Writing ObjectStore Applications for UNIX Platforms

ObjectStore Release 6.1 for all platforms, February 2003

© 2003 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.

Allegrix, Leadership by Design, Object Design, ObjectStore, Progress, Powered by Progress, Progress Fast Track, Progress Profiles, Partners in Progress, Partners en Progress, Progress en Partners, Progress in Progress, P.I.P., Progress Results, ProVision, ProCare, ProtoSpeed, SmartBeans, SpeedScript, and WebSpeed are registered trademarks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries. A Data Center of Your Very Own, Apptivity, AppsAlive, AppServer, ASPen, ASP-in-a-Box, BPM, Cache-Forward, Empowerment Center, eXcelon, EXLN, Fathom, Future Proof, Progress for Partners, IntelliStream, Javlin, ObjectStore Browsers, OpenEdge, POSSE, POSSENET, Progress Dynamics, Progress Software Developers Network, RTEE, Schemadesigner, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Stylus, Stylus Studio, WebClient, Who Makes Progress, XIS, XIS Lite, and XPRESS are trademarks or service marks of Progress Software Corporation or one of its subsidiaries or affiliates in the U.S. and other countries.

Any other trademarks and service marks contained herein may be the property of their respective owners.
Contents

Preface ................................................. v

Writing ObjectStore Applications .............. 1
  Making Complex Objects Persistent .............. 1
  The ObjectStore API ............................... 2

Basic ObjectStore Operations ..................... 2
  Database Management ............................. 2
  Database Roots .................................... 3
  Persistent Objects ................................. 3
  Transaction Management ......................... 3

A Checklist of ObjectStore Operations .......... 4

Using ObjectStore: An Example Application .. 6
  iostreams Version ............................... 7
  Flaws in the iostreams Version ................. 12
  ObjectStore Version ............................. 12

Building ObjectStore Applications ............. 20
  Generating Schemas .............................. 20
  Compiling and Linking ......................... 22

Using the Collections Facility .................. 23
  An Example ...................................... 24
  Building the Collections Example ............. 31

An Abbreviated ObjectStore Reference ......... 32
  Compiler Options ................................ 33
  get_os_typespec() ............................... 33
  Linker Options .................................. 34
Preface

Purpose
Writing ObjectStore Applications (UNIX platforms) shows how to write and build simple database applications that use basic features of the ObjectStore C++ application programming interface (API), including the collections facility.

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

Release
This manual supports ObjectStore Release 6.1.

How This Book Is Organized
The book is organized in sections, as follows:

• Making Complex Objects Persistent on page 1 discusses the benefits of using ObjectStore to make objects persistent.

• Basic ObjectStore Operations on page 2 describes the basic operations that any ObjectStore application must perform.

• A Checklist of ObjectStore Operations on page 4 lists the ObjectStore operations that you must implement in your application and the APIs that you use to implement them.

• Using ObjectStore: An Example Application on page 6 presents and compares two versions of an example program — one using iostreams and the other using ObjectStore.

• Building ObjectStore Applications on page 20 describes how to build an ObjectStore application. It uses the program presented in the previous section as an example.

• Using the Collections Facility on page 23 provides an overview of the collections facility and shows how to use some of its features. It uses a
revised version of the ObjectStore program presented earlier to illustrate how to implement the collections facility.

- An Abbreviated ObjectStore Reference on page 32 provides a summary reference of all features of the API mentioned in this Guide. For detailed and complete information, refer to the C++ API Reference and to the C++ Collections Guide and Reference.

### Notation Conventions

This document uses the following conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier</td>
<td>Courier font indicates code, syntax, file names, API names, system output, and the like.</td>
</tr>
<tr>
<td><strong>Bold Courier</strong></td>
<td><strong>Bold Courier font</strong> is used to emphasize particular code.</td>
</tr>
<tr>
<td><strong>Italic Courier</strong></td>
<td><strong>Italic Courier font</strong> indicates the name of an argument or variable for which you must supply a value.</td>
</tr>
<tr>
<td>Sans serif</td>
<td>Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.</td>
</tr>
<tr>
<td><strong>Italic serif</strong></td>
<td>In text, <strong>italic serif typeface</strong> indicates the first use of an important term.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Brackets enclose optional arguments.</td>
</tr>
<tr>
<td>{a</td>
<td>b</td>
</tr>
<tr>
<td>...</td>
<td>Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.</td>
</tr>
</tbody>
</table>

### Example Programs

The example programs discussed in this manual are available online in the directory

$OS_ROOTDIR/examples/quick_start
The quick_start directory consists of subdirectories that contain the source files for each program, as well as README.TXT text 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 ObjectStore. For example, none of the programs executes more than one transaction at run time. But ObjectStore’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.

Also, none of the example programs is multithreaded or uses the ObjectStore sessions facility. For information about using threads and sessions in ObjectStore applications, see Chapter 3 of the Advanced C++ API User Guide.

**ObjectStore on the World Wide Web**

ObjectStore has its own Web site (www.objectstore.net) that provides a variety of useful information about products, news and events, special programs, support, and training opportunities.

*Technical Support*

When you purchase technical support, the following services are available to you:

- You can send questions to support@objectstore.net. Remember to include your site ID in the body of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems.
- 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.
  - Frequently asked questions (FAQs) that you can browse and query.
  - Online documentation for all ObjectStore products.
  - White papers and short articles about using ObjectStore products.
  - Sample code and examples.
  - The latest versions of ObjectStore products, service packs, and publicly available patches that you can download.
  - Access to an ObjectStore product matrix.
  - Support policies.
- Local phone numbers and hours when support personnel can be reached.

### Education Services

Use the ObjectStore education services site (www.objectstore.net/services/education) to learn about the standard course offerings and custom workshops.

If you are in North America, you can call 1-800-477-6473 x4452 to register for classes. 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 ObjectStore Support Web server. The documentation is found at www.objectstore.net/documentation, and is listed by product. The site supports the most recent release and the previous supported release of ObjectStore documentation. 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

ObjectStore product development welcomes your comments about its documentation. Send any product feedback to support@objectstore.net. To expedite your documentation feedback, begin the subject with Doc:. For example:

Subject: Doc: Incorrect message on page 76 of reference manual
Writing ObjectStore Applications

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

- Persistence
- Basic ObjectStore operations, such as transactions and database management
- Writing and building an ObjectStore application
- Using the collections facility

Note

One purpose of this Guide is to expose the details of how you write simple ObjectStore 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 vi.

Making Complex Objects Persistent

ObjectStore offers a broad and varied range of products to meet your database needs. Its core competence, however, is quite simple: making complex objects persistent. ObjectStore 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 an ObjectStore database, they are ready for use and require no mapping code to move them from disk-resident format to memory-resident format.
Basic ObjectStore Operations

The ObjectStore API

To write an ObjectStore application, you use the application programming interface (API) that is part of ObjectStore. The API consists of C++ classes and macros that perform certain ObjectStore 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++.

The API consists of around 150 classes, most of which have a dozen or so public members. Most of the classes are designed for specialized tasks, making them 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 ObjectStore applications.

This Guide describes the basic ObjectStore 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 ObjectStore Reference on page 32.

Basic ObjectStore Operations

To use ObjectStore, 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 an ObjectStore database are familiar:

- Creating the database
- Opening the database
- Closing the database
What might be less obvious is that an ObjectStore 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 an ObjectStore application, you access the first object in a database by retrieving a database root. You create the root by using the ObjectStore 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 a query operation) 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 ObjectStore 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.

ObjectStore provides an overloading of new to create persistent objects and an overriding of delete to destroy them. ObjectStore 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 36 and ::operator delete() on page 36.

Transaction Management

Like most database management systems, ObjectStore requires that any access to persistent data occur 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.
By restricting persistent reads and writes within a transaction, ObjectStore can ensure the integrity of the database when two or more applications attempt to access the same data at the same time. Without a transaction, there is the risk that an application can see data in an intermediate state.

One problem that can occur during concurrent access can be illustrated from the example program presented in Using ObjectStore: An Example Application on page 6. The program enables the user to make dinner reservations at a restaurant. The database holds the number of tables currently available for reservations; the application updates this number each time the user makes a reservation. Suppose that two users try to reserve the same table at the same time. Without some form of concurrent access control such as that provided by ObjectStore's transaction facility, it is possible that the program would allow both customers to reserve the same table.

A Checklist of ObjectStore Operations

This section lists the items you must implement when you write an application to use ObjectStore. These items include the major operations described in Basic ObjectStore Operations on page 2 as well as minor but equally essential items, such as including ObjectStore header files.

- Include the required ObjectStore header files. All ObjectStore applications must include `ostore.hh`. For more specialized features of the API, you might have to include other ObjectStore header files. The ObjectStore C++ API Reference lists the required header files for all classes that require them. For an example, see “restaurant.hh” on page 13, line 5.

- Include all code that performs any ObjectStore operation within the following macros:
  - `OS_ESTABLISH_FAULT_HANDLER`
  - `OS_END_FAULT_HANDLER`

ObjectStore uses these macros to detect references to persistent memory. For example, if you use `new` to create an object, ObjectStore uses the macros to determine at run time whether to allocate persistent or transient memory.

The simplest way to use these macros is to insert `OS_ESTABLISH_FAULT_HANDLER` at the beginning of `main()` or `WinMain()` and to insert `OS_END_FAULT_HANDLER` at the end. For more information about the macros, see
OS_ESTABLISH_FAULT_HANDLER on page 42. For an example, see “init_db.cpp” on page 15, line 11 and line 33.

- Initialize ObjectStore by calling objectstore::initialize() before performing any ObjectStore operation. The simplest place to call this function is just after the OS_ESTABLISH_FAULT_HANDLER macro. For more information, see objectstore::initialize() on page 35. For an example, see “init_db.cpp” on page 15, 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 40. For an example, see “init_db.cpp” on page 15, line 21.
  - os_database::open() on page 40. For an example, see “main.cpp” on page 17, line 21.
  - os_database::close() on page 40. For an example, see “main.cpp” on page 17, line 44.

- Define the transaction boundaries of each sequence of statements that operate on persistent data as a logical unit. The simplest way to define transaction boundaries is to use the transaction macros, as described in OS_BEGIN_TXN() on page 37 and OS_END_TXN() on page 42. For an example, see “main.cpp” on page 17, line 24 and line 43.

- Create and destroy persistent data. To create a persistent object, use the ObjectStore overloading of operator new, as described in ::operator new() on page 36. For examples, see “restaurant.cpp” on page 13, line 25 (an object) and line 52 (a character array).

  ObjectStore overrides operator delete, so the same syntax can be used to destroy persistent or transient objects; see ::operator delete() on page 36. For an example, see “restaurant.hh” on page 13, line 16.

- Create a database root and associate it with the entry-point object in your database. ObjectStore provides the following methods for managing database roots:
  - os_database::create_root() on page 40. For an example, see “restaurant.cpp” on page 13, line 27.
  - os_database::find_root() on page 41. For an example, see “restaurant.cpp” on page 13, line 39.
  - os_database_root::get_value() on page 41. For an example, see “restaurant.cpp” on page 13, line 41.
Using ObjectStore: An Example Application

- os_database_root::set_value() on page 41. For an example, see “restaurant.cpp” on page 13, line 29.

- Perform an orderly shutdown of ObjectStore by calling objectstore::shutdown() just before your program exits. The simplest way to do this is to make the call just before the OS_END_FAULT_HANDLER macro. For more information, see objectstore::shutdown() on page 35. For an example, see “init_db.cpp” on page 15, line 32.

Using ObjectStore: An Example Application

To illustrate how to use ObjectStore 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 7.
- An ObjectStore version, which uses ObjectStore to make persistent the object that contains the restaurant data; see ObjectStore Version on page 12.

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 ObjectStore version — that is, the basic ObjectStore 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 `$OS_ROOTDIR/examples/quick_start/iostreams`

**restaurant.hh**

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

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

```c++
// restaurant.cpp: implements members of Restaurant
// and defines globals
```
Using ObjectStore: An Example Application

```c
#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!
        return 0;
    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;
}

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
Using ObjectStore: An Example Application

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:

CC -c init_db.cpp restaurant.cpp
CC -o init_db init_db.o restaurant.o

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);
Writing ObjectStore Applications

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

```
CC -c main.cpp restaurant.cpp
CC -o reserve main.o restaurant.o
```

The following is a sample run:

```
$ 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 and 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.
- The program has no way to deal with concurrent access. It cannot be used by more than one user.

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

ObjectStore Version

The ObjectStore 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 this version of the program are in the directory

$OS_ROOTDIR/examples/quick_start/basic_os

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.)

restaurant.cpp

Here is the implementation file.
```cpp
#include "restaurant.hh"

#include "dbname.h";

const int Restaurant::PERSONSPERTABLE = 4;
const char* Restaurant::DB_ROOT = "restaurant_root";

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

// create a Restaurant object and a database root, and associate
// the two
void Restaurant::create_restaurant(const char* s, int t,
    os_database* db)
{
    // get a pointer to typespec for Restaurant class
    os_typespec* ts = Restaurant::get_os_typespec();
    // create Restaurant object
    Restaurant* restaurant = new(db, ts) Restaurant(s, t);
    // create database root
    os_database_root* db_root = db->create_root(DB_ROOT);
    // associate root with object
    db_root->set_value(restaurant, ts);
    // delete heap-allocated os_database_root object
    delete db_root;
}

// return a pointer to the Restaurant object that is associated with
// the database root
Restaurant* Restaurant::get_restaurant(os_database* db)
{
    // find root
    os_database_root* db_root = db->find_root(DB_ROOT);
    // retrieve and return associated object
    return (Restaurant*)db_root->get_value(
        Restaurant::get_os_typespec());
    // delete heap-allocated os_database_root object
    delete db_root;
}

// 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;
}
Writing ObjectStore Applications

char* p = new(os_cluster::with(this),
os_typespec::get_char(), len) char [len];
return strcpy(p, s);

// return the number of tables reserved, based on the number of
// persons in the party; 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)
            tables_to_reserve = 0;
        else
            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 Restaurant object and makes it persistent. Commentary follows the listing.

#include <cstring>

int main(int argc, char** argv)
{
    int n_tables;
    OS_ESTABLISH_FAULT_HANDLER {
        objectstore::initialize();
        // check for missing or bad arguments
    }
Using ObjectStore: An Example Application

```cpp
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_BEGIN_TXN(txn, 0, os_transaction::update) {
25
26 // create a Restaurant object
27 Restaurant::create_restaurant(argv[1], n_tables, db);
28
29 } OS_END_TXN(txn)
30 db->close();
31 delete db; // delete heap-allocated os_database object
32 objectstore::shutdown(); // cleanup
33 } OS_END_FAULT_HANDLER
34
35 return 0;
36}
```

**Commentary on init_db.cpp**

The structure and interface of the ObjectStore 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 ObjectStore. The key differences are:

- All code that performs ObjectStore operations is delimited by page-fault handling macros.
- **ObjectStore is initialized by a call to objectstore::initialize().**
- All code that accesses persistent data is contained within a transaction defined by the ObjectStore transaction macros.

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 ObjectStore 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 ObjectStore 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 ObjectStore 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` but is implemented by ObjectStore.

For a description of persistent `new`, see `::operator new()` on page 36.

The ObjectStore 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:

```c
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()`, Technical Support recommends using it to prevent mismatch errors; see `os_database_root::set_value()` on page 41.

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**

The `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.

```c
// main.cpp: processes restaurant reservations
#include "restaurant.hh"
#include <stdlib.h>
#include "dbname.h"
```
Using ObjectStore: An Example Application

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

Commentary on main.cpp

The ObjectStore 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 16. In other words, the
ObjectStore 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 ObjectStore 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

```cpp
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 13, 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 ObjectStore 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 13, line 29.

As part of cleanup, main.cpp deletes the os_database object to which db points; see “main.cpp” on page 17, line 45. This is a transient object that represents the persistent database. Deleting it does not delete the database.

Although explicit deletion in this example program is unnecessary (the storage will be released when the program exits), in a more realistic program...
such objects can cause serious memory leaks unless their lifetimes are carefully managed. For information about using the ObjectStore API for help in managing the lifetimes of transient objects, see Managing Lifetimes of ObjectStore Objects in Chapter 3 of the C++ API User Guide.

Building ObjectStore Applications

The steps for building an ObjectStore application are

1. Generate schemas.
2. Compile the source code.
3. Link the application.

Except for Step 1, building an ObjectStore 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. ObjectStore uses this schema information whenever an application accesses a database, and you use it in the remaining steps of the build process, as described in the following sections.

Note

Before building an application, you must have installed ObjectStore and correctly set all necessary environment variables. In particular, `OS_ROOTDIR` must be set to the top-level ObjectStore directory.

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 ObjectStore schema generator, `ossg`, 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 Determining the Types in a Schema in Chapter 2 of Building ObjectStore C++ Applications.
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 42.

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:

```cpp
OS_MARK_SCHEMA_TYPE(Restaurant);
```

The schema source file must also include the ObjectStore header files `ostore.hh` and `manschem.hh`, as well as the application header files that define the marked classes. Any other ObjectStore header files included in your application must also be included in the schema source file.

ObjectStore header files are order dependent, and `ostore.hh` must be included before any others; see ObjectStore Header Files in Chapter 2 of Building ObjectStore C++ Applications for the required order.

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

```cpp
#include "restaurant.hh"
#include <ostore/manschem.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:

```bash
CC -I${OS_ROOTDIR}/include -o schema_src.cpp
```

For information about the compiler options used in this command line, see Compiler Options on page 33.

After successful compilation, you can delete the generated object file. It is not needed for building an ObjectStore application.

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

```bash
ossg -I${OS_ROOTDIR}/include -assf schema.cpp \ -asdb schema.adb schema_src.cpp
```
For information about the `ossg` options used in this command line, see `ossg` on page 43. Note that the schema source file (`schema_src.cpp` in the example) does not require an option; however, it must appear at the end of the command line.

After processing, `ossg` produces two files:

- An application schema database; in this example, `schema.adb`. This database contains type information about the objects your application can store persistently.
- An application schema source file; in this example, `schema.cpp`. You must compile this file separately, as described in the next section.

### Compiling and Linking

After you have successfully run `ossg`, you are ready to compile and link your application. Compiling and linking an ObjectStore 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:

```bash
CC -I${OS_ROOTDIR}/include -c main.cpp restaurant.cpp init_db.cpp schema.cpp main.cpp
```

Note that, as shown here, you must also compile the application schema source file produced by `ossg`.

For information about the compiler options in this command line, see Compiler Options on page 33.

**Link line**

Here is the command line for linking the application object files into an executable:

```bash
CC -vdelx -mt -L${OS_ROOTDIR}/lib -o reserve main.o restaurant.o schema.o -los -losth
```

For information about the linker options used in this command line, see Linker Options on page 34.

The command line for linking `init_db` is essentially the same as the one for linking `reserve`.

**Running the application**

The following is a sample run:

```bash
$ reserve 11
Reserved 3 tables at Il Falchetto
```
Using the Collections Facility

The ObjectStore 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, which would provide navigational access to additional objects in the database; see, for example, the sample program in Chapter 2 of the C++ API User Guide.

But ObjectStore provides a better way — the collections facility. This facility enables applications to group database objects in a collection and then to perform queries on the collection.

A **collection** is an object (such as a set or list) that can group other objects (such as `Restaurant` objects). ObjectStore 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 C++ Collections Guide and Reference.

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, the most important of which is a **query**. A query is used to return one or more elements of a collection, based on selection criteria that you specify. For example, you could use a query to find all `Restaurant` objects that can accommodate a party of 30.

To use the collections API, an application must

- Include the `coll.hh` header file just after `ostore.hh`.
- Call `os_collection::initialize()` just after `objectstore::initialize()`.

---

**Release 6.1**
Using the Collections Facility

- Link with the collections libraries; see Building the Collections Example on page 31.

For other requirements that might apply, see Requirements for Collections Applications in Chapter 1 of the 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 and to perform queries on those objects. To make the program more interesting, another data member (city) has been added to the Restaurant class, identifying 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)

All source files for the program — including those for additional applications, such as init_db, which populates the collection — are in the directory

$OS_ROOTDIR/examples/quick_start/coll_os

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

```
01// restaurant.hh: defines the Restaurant class
02#include <iostream.h>
03#include <string.h>
04#include <stdlib.h>
05#include <ostore/ostore.hh>
06#include <ostore/coll.hh>
07
08// forward declarations
09class Restaurant;
10
11// global function
12ostream& operator<<(ostream& os, Restaurant& r);
13
14class Restaurant {
15private:
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)
```
Here is the implementation file:

```
// restaurant.cpp: implements members of Restaurant
// and defines globals
#include "restaurant.hh"
#include "dbname.h"
const char* DB_ROOT = "restaurant_root";
const int Restaurant::QS_SIZE = 100;
const int Restaurant::PERSONSPERTABLE = 4;

Restaurant::Restaurant(const char* n, const char* c, int t):
    name(dupl_string(n)), city(dupl_string(c))
{
    tables = t;
}
```

```
Using the Collections Facility

017\}
018
019// return a pointer to a new Restaurant object
020Restaurant* Restaurant::create_restaurant(const char* n,
021 const char* c, int t, os_database* db)
022\}
023 // get a pointer to typespec for Restaurant class
024 os_typespec* ts = Restaurant::get_os_typespec();
025 // create Restaurant object
026 return new(db, ts) Restaurant(n, c, t);
027\}
028
029// Create a collection object and a root, and associate the two;
030// return a pointer to the collection object
031os_set* Restaurant::create_restaurant_set(os_database* db)
032\}
033 // get a typespec for os_set
034 os_typespec* rest_ts = os_set::get_os_typespec();
035
036 // create a collection for Restaurant objects
037 os_set* rest_set = new(db, rest_ts) os_set();
038
039 // create a database root and associate it with the collection
040 os_database_root* db_root = db->create_root(DB_ROOT);
041 db_root->set_value(rest_set, rest_ts);
042
043 // delete the heap-allocated os_database_root object
044 delete db_root;
045
046 return rest_set;
047\}
048
049// allocate storage to hold the string in s, copy s to the
050// newly allocated storage, and return a pointer to the storage
051char* Restaurant::dupl_string(const char* s)
052\}
053 int len = strlen(s)+1;
054 char* p = new(os_cluster::with(this), os_typespec::get_char(),
055 len) char [len];
056 return strcpy(p, s);
057\}
058
059// return the number of tables reserved, based on the number of
060// persons in the party; otherwise, 0
061int Restaurant::make_reservation(int n_persons)
062\}
063 int tables_to_reserve = 0;
064 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;

// return the Restaurant object in the collection "coll" whose
// name data member matches "n". This function can be made
// more general-purpose by adding an int argument as a token
// to select alternate data members for queries.
Restaurant* Restaurant::query_pick(const char* n,
os_database* db, os_set* coll)
{
  char qstr[QS_SIZE];
  sprintf(qstr, "!strcmp(name, "%s")", n);
  return (Restaurant*)coll->query_pick("Restaurant*", qstr, db);
}

// return a collection of objects whose city data member matches
// "c". See the comment on Restaurant::query_pick() for making
// this function more general purpose.
os_collection& Restaurant::query(const char* c,
os_database* db, os_set* coll)
{
  char qstr[QS_SIZE];
  sprintf(qstr, "!strcmp(city, "%s")", c);
  return coll->query("Restaurant*", qstr, db);
}

// overloading of << for displaying information about a restaurant
ostream& operator<<(ostream& os, Restaurant& r)
{
  cout << r.get_name() << " in " << r.get_city()
Using the Collections Facility

```cpp
tables reserved;

// check for missing or bad arguments
if (argc != 3) {
  cerr << "USAGE: reserve <name> <n-persons>\n";
  return 1;
}

// initialize ObjectStore and collections facility
objectstore::initialize();

// open database
os_database* db = os_database::open(DB_NAME);

os_database_root* db_root;

// start update transaction
OS_BEGIN_TXN(txn, 0, os_transaction::update) {
  // find the database root
  db_root = db->find_root(DB_ROOT);

  // retrieve collection of restaurants associated with root
  os_set* rset = (os_set*)db_root->get_value(os_set::get_os_typespec());

  // query the restaurant named on command line
  Restaurant* r = Restaurant::query_pick(argv[1], db, rset);

  // restaurant in database?
  if (r) { // yes
    tables reserved;
    return os;
  }
```

Here is the source listing for main application, `reserve`. Commentary follows the listing.
// try to book a reservation
tables_reserved = r->make_reservation(atoi(argv[2]));

// confirm or deny reservation
if (tables_reserved)
    cout << "Reserved " << tables_reserved << " tables.\n";
else // not enough tables
    cout << "Sorry, all booked.\n";

else // name not found in database
    cout << "Can\'t find " << argv[1] << " in database.\n";

} OS_END_TXN(txn)

db->close();

// delete heap-allocated objects
delete db;
delete db_root;

objectstore::shutdown(); // cleanup

return 0;

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 queries to retrieve Restaurant objects

Collection as entry-point object

The noncollections 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 (at "restaurant.cpp" on page 13), 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_set() function (see the collections version of restaurant.cpp -- at "restaurant.cpp" on page 25 --, line 31) 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_set) using persistent new on line 37.
Using the Collections Facility

- It gets a pointer to a newly created database root by calling `create_root()` on line 40.
- It associates the root with `rest_set` by calling `set_value()` on line 41.

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, `query()`) to access the elements of the collection. For an example, see “main.cpp” on page 28, line 31-34.

Querying the collection

The program provides two ways to query the collection:

- By calling `Restaurant::query_pick()` to query on the name of a restaurant. This function returns the one `Restaurant` object with the matching name, if one exists in the database; see “restaurant.cpp” on page 25, line 90.
- By calling `Restaurant::query()` to query on the name of a city. This function returns a collection of `Restaurant` objects with the matching city; see “restaurant.cpp” on page 25, line 102.

Calling `query_pick()`

The main application (`reserve`) calls `Restaurant::query_pick()`; see “main.cpp” on page 28, line 38. The application expects the name of a restaurant as one of its command-line arguments. It passes this argument (`argv[1]`) to `query_pick()`, which returns a pointer to the `Restaurant` object, if an object with a matching name member is found.

The `Restaurant::query_pick()` function calls `sprintf()`, using the value that was passed as an argument to construct the query string; see “restaurant.cpp” on page 25, line 95. The query string specifies the condition that must be met to satisfy the query: the strings in argument `n` and in the name member of the `Restaurant` class must match. After the query string is constructed, it is passed to `os_set::query_pick()` on line 96. If the query is satisfied, this function returns a pointer to a `Restaurant` object.

Processing the query string

The call to `sprintf()` to construct the query string is satisfactory in a program like this one, which executes no more than one query at run time; see “restaurant.cpp” on page 25, line 95. But a more realistic application could execute the same query thousands of times, and calling `sprintf()` each time would result in poor performance — not to mention the risk of writing past the declared boundary of `qstr`.

A better way to process a query string is to use ObjectStore’s `preanalyzed` query. Using preanalyzed queries can amortize much of the work of analyzing a query string over a number of query operations. In addition,
they can be made persistent and thus be reused over different invocations of
the application, or by different applications. For more information, see
Preanalyzed Queries in Chapter 6 of C++ Collections Guide and Reference.

In the collections version of the program, the number of elements in the
collection is so small that the time it takes to query the collection is negligible.
But in a more realistic program — one having several collections, each with
thousands of elements — query processing could seriously degrade
performance.

To optimize query processing, ObjectStore provides indexing, which
minimizes the number of objects examined in response to a query. To
implement query optimization, you use the collections API to add an index
to a collection. Once added, the index is maintained for all objects
subsequently inserted in the collection or removed from it.

Here are the lines you would code in `create_restaurant_set()` to add an
index on the `name` data member of the `Restaurant` class in the example
program:

```cpp
os_index_path& nspec = os_index_path::create("Restaurant*",
"name", db);
rest_set->add_index(nspec, os_index_path::no_duplicates|
     os_index_path::signal_duplicates);
```

For more information about query optimization and indexing, see Chapter 7
of the 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 ObjectStore version:

1 Run `ossg` 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 `ossg` for the
noncollections version of the program; see Generating Schemas on page 20.
The changes to the `Restaurant` class (for example, the addition of the `city`
member) require you to run `ossg` again to generate fresh schemas. However,
the same schema source file can be used as input to `ossg`; see “schema_
src.cpp” on page 21 for the listing.

Here are the command lines for generating schemas, compiling, and linking
the `reserve` application:
Note that the `ossg` and compiler command lines are the same as those used for the noncollections version of the program; see Building ObjectStore Applications on page 20. However, the link line specifies three additional library options not used for linking the noncollections version:

```
-losqry -loscol -losmop
```

These options must be specified when you are linking any ObjectStore application that uses the collections facility. Also, these options must be specified in the order shown here, preceding the `-los` and `-losth` options. For additional information about linking ObjectStore applications, see Chapter 4 of Building ObjectStore C++ Applications.

### 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:

```
reserve "Il Falchetto" 17
Reserved 5 tables.
```

### An Abbreviated ObjectStore Reference

This section provides reference information for features of the ObjectStore 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 overloads, 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 ObjectStore documentation. However, for detailed and complete information about the ObjectStore API, refer to the *ObjectStore C++ API Reference* and to the 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>Causes the compiler to compile without linking.</td>
</tr>
<tr>
<td>-I</td>
<td>Specifies the location of the ObjectStore include directory. You can use this option more than once on the same command line to specify additional include directories.</td>
</tr>
</tbody>
</table>

For more information, see the following:

- Chapter 4 of *Building ObjectStore C++ Applications* for detailed information about compiling ObjectStore applications and about other options you might need to use
- “Compile line” on page 22 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, using the following format:

```c
static os_typespec* get_os_typespec();
```

You do not implement this function. The ObjectStore schema generator (ossg) implements it when you generate schemas for your application.

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

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

where `class_name` specifies the name of the class in which you declared it. 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), enabling you to use it to return a typespec argument when you use ::operator new() to create a collections object.
An Abbreviated ObjectStore Reference

For more information

See the following:

- `os_typespec::get_char()` on page 43 for information about supplying the typespec argument for fundamental types, such as `char`
- `restaurant.hh` (see “restaurant.hh” on page 13), line 26, for an example declaration
- `restaurant.cpp` (see “restaurant.cpp” on page 13), line 23, for an example call

Linker Options

The following table lists and describes the linker options used for linking the example programs presented in this Guide.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-L</td>
<td>Specifies the location of the ObjectStore lib directory.</td>
</tr>
<tr>
<td>-los</td>
<td>Links the core ObjectStore library into the executable. This option is required for linking any ObjectStore application.</td>
</tr>
<tr>
<td>-loscol</td>
<td>Links the collections library into the executable. This option is required for applications that use the collections facility.</td>
</tr>
<tr>
<td>-losmop</td>
<td>Links the metaobject protocol (MOP) library into the executable. This option is required for applications that use the collections facility.</td>
</tr>
<tr>
<td>-losqry</td>
<td>Links the queries and indexing library into the executable. This option is required for applications that use the collections facility.</td>
</tr>
<tr>
<td>-losth</td>
<td>Links ObjectStore thread primitives into the executable. This option is required for linking an ObjectStore application, even if the application does not use multiple threads.</td>
</tr>
<tr>
<td>-mt</td>
<td>Prepares the executable to use threads. This option is required for linking any ObjectStore application, even if the application does not use multiple threads.</td>
</tr>
</tbody>
</table>
The library options are order sensitive and must be specified in the order as shown in the example link lines:

- For an example link line for an application that does not use the collections facility, see Compiling and Linking on page 22.
- For an example link line for an application that does use the collections facility, see Building the Collections Example on page 31.

For more information See also Chapter 4 of Building ObjectStore C++ Applications for detailed information about linking ObjectStore applications and about other options you might need to use.

**objectstore::initialize()**

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

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

For more information See “main.cpp” on page 17, line 12, for an example.

**objectstore::shutdown()**

```cpp
static void shutdown();
```

Conducts an orderly shutdown of ObjectStore, such as cleaning up internal memory use by ObjectStore and closing any network connections that the application might have made. Technical Support recommends that all ObjectStore applications call this function before they exit.

For more information See “main.cpp” on page 17, line 46, for an example.
::operator delete()

::operator delete() is a global function that you can use to release either transient or persistent storage. ObjectStore 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 13, line 16, for an example of the delete operator used to delete a persistent character array.

::operator new()

ObjectStore 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. This can be a pointer to an os_database, os_segment, or os_cluster object; or an os_cluster_ with object, which is returned by os_cluster::with().

- **typespec** is a pointer to an os_typespec object that provides ObjectStore 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_cluster::with() on page 37
- os_typespec::get_char() on page 43 for information about typespecs for fundamental types
• get_os_typespec() on page 33 for information about typespecs for user-defined types
• “restaurant.cpp” on page 13, line 25, for an example of new used to allocate storage for a user-defined type
• “restaurant.cpp” on page 13, line 52, for an example of new used to allocate storage for a fundamental type

**OS_BEGIN_TXN()**

Use this macro to begin a *lexical* transaction and OS_END_TXN() to end and commit it. A lexical transaction is called as such because its boundaries are fixed at run time according to where the transaction macros are positioned in the source code. A lexical transaction provides automatic retry in case of a deadlock, which occurs when two or more transactions attempt to access the same data item at the same time.

Here is the syntax for defining a lexical transaction:

```
OS_BEGIN_TRANSACTION(txn-tag, expr, txn-type) {
    // your transaction code goes here
} OS_END_TRANSACTION(txn-tag)
```

The arguments have the following meanings:

• **txn-tag** is an identifier that uniquely identifies this transaction. The tag specified in OS_BEGIN_TXN() and OS_END_TXN() must be the same.
• **expr** is used for getting exception information. Specify 0 if you do not use this argument.
• **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 on these arguments, see:

• OS_BEGIN_TXN() in Chapter 4 of the ObjectStore C++ API Reference for more information about the expr argument
• Chapter 5 of the C++ API User Guide for detailed information about transactions, including dynamic transactions
• “main.cpp” on page 17, line 24, for an example of a lexical transaction

**os_cluster::with()**

```
static os_cluster_with with(const void* object);
```
This function can be used as the first argument to `::operator new()` when you want to allocate storage as close as possible to a previously allocated object — an allocation optimization known as clustering. The `with()` function returns an `os_cluster_with` object.

For more information

See the following:

- `::operator new()` on page 36
- “restaurant.cpp” on page 13, line 52, for an example

**os_collection::initialize()**

```cpp
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 35 for information about initializing ObjectStore
- “main.cpp” on page 28, line 21, for an example

**os_collection::query()**

```cpp
os_collection& query(
    char* element-type-name,
    char* query-string,
    os_database* schema-database);
```

This function returns a reference to a heap-allocated collection that contains those elements of the implicit `this` argument that satisfy the selection criteria specified in `query-string`.

The arguments have the following meanings:

- `element-type-name` is the type name of the object being queried, which must be the type of the elements in the collection.
- `query-string` is a C++ control expression indicating the query’s selection criteria — that is, the condition that must be met for an object to be selected.
- `schema-database` is a pointer to the database whose schema contains all the types mentioned in the selection criteria expressed in `query-string`. Typically, this database is also the database in which the collection resides.
The `query()` function is a member of the base class `os_collection` and is therefore inherited by the collections classes (for example, `os_set`) that derive from `os_collection`.

Because the return value is a reference to a heap-allocated object, you should delete the object when it is no longer needed to prevent a memory leak.

For more information

See the following:

- `os_collection::query()` in Chapter 8 of the C++ Collections Guide and Reference for a full description of this function and its overladings
- “restaurant.cpp” on page 25, line 108, for an example

### `os_collection::query_pick()`

```cpp
os_collection& query_pick(
    char* element-type-name,
    char* query-string,
    os_database* schema-database);
```

This function returns a pointer to an element of the implicit `this` argument that satisfies the selection criteria specified in `query-string`. If no element satisfies the selection criteria or if the collection is empty, `query_pick()` returns 0.

The arguments have the following meanings:

- `element-type-name` is the type name of the object being queried, which must be the type of the elements in the collection.
- `query-string` is a C++ control expression indicating the query’s selection criteria — that is, the condition that must be met for an object to be selected.
- `schema-database` is a pointer to the database whose schema contains all the types mentioned in the selection criteria expressed in `query-string`. Typically, this database is also the database in which the collection resides.

The `query_pick()` function is a member of the base class `os_collection` and is therefore inherited by the collections classes (for example, `os_set`) that derive from `os_collection`.

For more information

See the following:

- `os_collection::query_pick()` in Chapter 8 of the C++ Collections Guide and Reference for a full description of this function and its overladings
- “restaurant.cpp” on page 25, line 96, for an example
An Abbreviated ObjectStore Reference

**os_database::close()**

```c
void close();
```

Closes the database specified by the `this` argument.

For more information

See “main.cpp” on page 17, line 44, for an example.

**os_database::create()**

```c
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 overloads of this function that will cause it to overwrite an existing database at `pathname`.

For more information

See “init_db.cpp” on page 15, line 21, for an example.

**os_database::create_root()**

```c
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 41
- `os_database_root::set_value()` on page 41 for information about associating the root with an entry-point object
- “restaurant.cpp” on page 13, line 27, for an example

**os_database::open()**

```c
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 overloads of this function that will cause it to create a database.

For more information

See “main.cpp” on page 17, 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 40 for information about creating a root
• “restaurant.cpp” on page 13, 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, ObjectStore checks that it matches the typespec argument specified in os_database_root::set_value(). If this argument is unspecified or 0, ObjectStore does not check for matching typespecs.

For more information
See the following:
• os_database_root::set_value() on page 41 for information about associating a root with an entry-point object
• “restaurant.cpp” on page 13, 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, ObjectStore checks that it matches the typespec argument specified in os_database_root::get_value(). If this argument is unspecified or 0, ObjectStore does not check for matching typespecs.

For more information
See the following:
• os_database_root::get_value() on page 41
• “restaurant.cpp” on page 13, line 29, for an example
OS_END_FAULT_HANDLER
All applications that perform macro ObjectStore operations must use this macro and the OS_ESTABLISH_FAULT_HANDLER macro.
For more information See OS_ESTABLISH_FAULT_HANDLER on page 42.

OS_END_TXN()
The OS_END_TXN() macro is used to define the end of a lexical transaction.
For more information See OS_BEGIN_TXN() on page 37.

OS_ESTABLISH_FAULT_HANDLER
ObjectStore requires applications to use this macro and the OS_END_FAULT_HANDLER macro so that it can detect references to persistent storage. All applications performing ObjectStore operations must use these two 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 ObjectStore operations.
For more information See “init_db.cpp” on page 15, line 11 and line 33, 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:

- Determining the Types in a Schema in Chapter 2 of Building ObjectStore C++ Applications for more information about classes you must mark
- “schema_src.cpp” on page 21 for an example
OS_ROOTDIR

The OS_ROOTDIR environment variable specifies the top-level directory in the part of the file-system hierarchy that contains ObjectStore files. This variable is set when you install ObjectStore and is required to run ObjectStore. It is used to identify (among other things) the location of ObjectStore header files.

For more information

See OS_ROOTDIR in Chapter 3 of Managing ObjectStore for more information.

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 33 for getting a typespec for a user-defined type
- ::operator new() on page 36
- “restaurant.cpp” on page 13, line 52, for an example

ossg

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

```
ossg -Iinclude-path -assf app-schema-src.cpp \
-asdb app-schema-db.adb schema-src-file.cpp
```

The arguments have the following meanings:

- -I specifies the path of any include directories to be used by ossg. Typically, this path consists of the OS_ROOTDIR environment variable followed by include. For example:
  ```
  -I$OS_ROOTDIR/include
  ```
  However, you can specify -Iinclude-path more than once on the same command line.
An Abbreviated ObjectStore Reference

- **-assf** specifies the name of the application schema source file to be produced by `ossg`. You must compile this file.
- **-asdb** specifies the name of the application schema database to be produced by `ossg`.
- **schema-src-file.cpp** specifies the C++ schema source file that you must supply as input to `ossg`. 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 20 for information about the schema source file
- `ossg`: Generating Schemas in Chapter 4 of *Managing ObjectStore* for a full description of `ossg` and all of its options
- Chapter 3 of *Building ObjectStore C++ Applications* for detailed information about using `ossg` to generate schemas
- “`ossg command line`” on page 21 for an example `ossg` command line
Index

A
aborting a transaction 3
API
  checklist 4
  overview 2
  reference information 32
application schemas 22
applications
  compiling 22
  generating schemas 20
  linking 22
associating roots and objects 41

B
basic operations
  example program 12
  listed 2
building applications
  basic ObjectStore 20
  collections 31

C
checklist of ObjectStore operations 4
close() member of os_database
description 40
  example 10
clustering
database
  basic operations 2
  os_database methods 5
database roots
  creating 40
  described 3
E

entry-point object 41
finding 41
operations 5
os_database_root methods 5

deadlock 37
delete operator
description 36
example 18

E

elements of a collection 23
entry-point object
collection as 29
described 3
example 19
retrieving 41
environment variables
OS_ROOTDIR 43
example programs
basic ObjectStore version 12
collections version 24
iostreams version 7

F

find_root() member of os_database
description 41
example 14

G

get_char() member of os_typespec
description 43
example 15
get_os_typespec()
description 33
example 13
get_value() member of os_database_root
description 41
example 14

global functions
operator delete() 36
operator new() 36

H

header files
applications using collections 23
coll.hh 23
in basic applications 4
in schema source file 21
manschem.hh 21
ostore.hh 4

I

implementing ObjectStore 6
include files 4
indexing and query optimization 31
init_db application
basic ObjectStore version 15
iostreams version 9
initialization
collections facility 23
ObjectStore 5
initialize() member of objectstore
description 35
example 15
required 5
initialize() member of os_collection
description 38
example 28
iostreams example 7

L

lexical transactions 37
linking
application using collections 31
basic application 22
command-line options 34
Index

M
macros
  OS_BEGIN_TXN 37
  OS_END_FAULT_HANDLER 42
  OS_END_TXN 42
  OS_ESTABLISH_FAULT_HANDLER 42
  OS_MARK_SCHEMA_TYPE 42
managing database operations 2
manschem.hh header file 21
marking types 21

N
new operator
  description 36
  example 14

O
ObjectStore
  API 2
  basic operations 2
  building an application 20
objectstore::initialize()
    description 35
    example 15
    required call 5
objectstore::shutdown()
    description 35
    example 18
open() member of os_database
    description 40
    example 18
operator delete()
  deleting transient objects 19
    description 36
    example 18
operator new()
    description 36
    example 14
optimizing
queries 31
storage allocation 38
options
  compiler 33
  linker 34
  ossg utility 43
OS_BEGIN_TXN macro
  description 37
  example 16
os_cluster::with()
  description 38
  example 15
os_collection::initialize()
  description 38
  example 28
os_collection::query()
  description 38
  example 25
os_collection::query_pick()
  description 39
  example 25
os_database::close()
  description 40
  example 10
os_database::create()
  description 40
  example 16
os_database::create_root()
  description 40
  example 14
os_database::find_root()
  description 41
  example 14
os_database::open()
  description 40
  example 18
os_database_root::get_value()
  description 41
  example 14
os_database_root::set_value()
  description 41
Writing ObjectStore Applications

example 14
OS_END_FAULT_HANDLER macro
description 42
element 16
OS_END_TXN macro
description 42
element 16
OS_ESTABLISH_FAULT_HANDLER macro
description 42
element 15
OS_MARK_SCHEMA_TYPE macro
description 42
element 21
OS_ROOTDIR environment variable
building applications 20
description 43
os_typespec::get_char() description 43
element 15
ossg utility
description 43
element command line 21
ostore.hh header file
in basic applications 4
in collections applications 23
in schema source file 21

P
page-fault macros 4
OS_END_FAULT_HANDLER 42
OS_ESTABLISH_FAULT_HANDLER 42
persistence
creating persistent objects 3
operator delete() 36
operator new() 36
overview 1
page-fault macros 4
populating a collection 23
preanalyzed queries 30

Q
queries
described 23
optimizing 31
query() member of os_collection
description 38
element 25
query_pick() member of os_collection
description 39
element 25

R
reservations application
basic ObjectStore version 12
collections version 24
iostreams version 7
rolling back a transaction 3

S
schema source file
checking for errors 21
element 21
OS_MARK_SCHEMA_TYPE macro 42
schemas 20
set_value() member of os_database_root
description 41
element 14
shutdown() member of objectstore
description 35
element 18

T
transactions
aborting 3
committing 3
concurrent access 4
deadlock 37
delimiting 5
example 16
lexical 37
managing 3
OS_BEGIN_TXN 37
OS_END_TXN 42
rolling back 3
transient objects
  compared to persistent objects 1
deleting 19
typespec argument
  new 36
    os_database_root::get_value() 41
    os_database_root::set_value() 41
typespecs
  fundamental types 43
    get_os_typespec() 33
    os_typespec::get_char() 43
  reasons for using 17
  user-defined types 33

U
utilities
  ossg 43

W

  with() member of os_cluster
    description 38
    example 15