

OBJECTSTORE

BUILDING C++ INTERFACE APPLICATIONS

RELEASE 5.1

March 1998

ObjectStore Building C++ Interface Applications

ObjectStore Release 5.1 for all platforms, March 1998

ObjectStore, Object Design, the Object Design logo, LEADERSHIP BY DESIGN, and Object Exchange are registered trademarks of Object Design, Inc. ObjectForms and Object Manager are trademarks of Object Design, Inc.

All other trademarks are the property of their respective owners.

Copyright © 1989 to 1998 Object Design, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

COMMERCIAL ITEM — The Programs are Commercial Computer Software, as defined in the Federal Acquisition Regulations and Department of Defense FAR Supplement, and are delivered to the United States Government with only those rights set forth in Object Design's software license agreement.

Data contained herein are proprietary to Object Design, Inc., or its licensors, and may not be used, disclosed, reproduced, modified, performed or displayed without the prior written approval of Object Design, Inc.

This document contains proprietary Object Design information and is licensed for use pursuant to a Software License Services Agreement between Object Design, Inc., and Customer.

The information in this document is subject to change without notice. Object Design, Inc., assumes no responsibility for any errors that may appear in this document.

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

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

Contents

	Preface	xi
Chapter 1	Overview of Building an Application	1
	General Instructions for Building Applications	2
	Flow Chart for Building Applications	4
	Third-Party Compilers You Can Use	5
	Third-Party Libraries and Applications	6
	Virtual File Systems	7
	ObjectStore Server and the Build Process	9
	ObjectStore/Single	10
Chapter 2	Working with Source Files	13
	Overview of Source Files	14
	ObjectStore Header Files	15
	Instantiation Problem with Template Collection Classes ..	16
	os_collection_declare() Macro	16
	os_collection_declare_no_class() Macro	17
	os_collection_declare_ptr_tdef() Macro	17
	Required Order of Include Statements	17
	Determining the Types in a Schema	19
	Which Types to Mark	19
	Which Types Are Reachable	19
	If You Are Not Sure a Type Is Reachable	20
	Choosing Whether to Mark Types or Specify -mrscp	20
	Ensuring a Complete Schema Source File	21

Contents

	Creating Schema Source Files	22
	Schema Source File Format	22
	Placing Calls to OS_MARK_SCHEMA_TYPE in a Function	23
	Schema Source File Examples	24
	Schema Source File Must Be Compilable	25
	OS/2 Notes	25
	When You Modify a Source File	27
Chapter 3	Generating Schemas	29
	Overview of Schema Generation	30
	Application Schemas	30
	Component Schemas	31
	Library Schemas	31
	Compilation Schemas	31
	Database Schemas	32
	Keeping Database Schemas and Application Schemas Compatible	33
	Schemas Are Platform Specific	33
	Generating an Application or Component Schema	35
	Invoking ossg to Generate an Application Schema	36
	Changing the Default Preprocessor	46
	OS/2 Platforms Have an Additional ossg Option	47
	Using a Temporary File to Send Arguments to ossg	47
	Specifying ObjectStore Library Schemas	48
	Specifying Remote Library Schemas	48
	How Long Does Schema Generation Take?	48
	If You Omit a Required Library Schema	49
	Example of Using the -runtime_dispatch Option	50
	Using the Same Application Schema for Multiple Applications	51
	Examples of Generating an Application Schema	51

Generating a Library Schema	53
Invoking ossg to Generate a Library Schema	53
Using Multiple Schema Source Files to Create a Library Schema	53
If Not Creating a Library Schema	54
Generating a Compilation Schema	55
Why Generate a Compilation Schema?	55
Invoking ossg to Generate a Compilation Schema	56
Using a Compilation Schema to Generate an Application Schema	56
Hiding Code from the Schema Generator	57
Unsupported Types	58
Limited Support for long long Data Type	58
Support for wchar_t Types	58
Restricting Use of Template Classes and Collections	59
Correcting Schema-Related Errors	60
Type Mismatch Errors	60
Persistent Allocation Errors	60
ossg Run-Time Errors	61
Metaschema Mismatch Errors	61
Schema Neutralization Errors	62
Missing Virtual Function Table Pointer Problems	62
Handling Pragma Statements in Source Code	63
Utilities for Working with Schemas	64
Comparison of ossg Command Lines	65
Comparison of Kinds of Schemas	66
Deploying Products with Protected Databases	68
Using Rogue Wave with Solaris Sun C++	69
ossg Troubleshooting Tips	70

Chapter 4	Compiling, Linking, and Debugging Programs . . .	73
	Using Standard Template Libraries	74
	How to Store STL Types Persistently	74
	Microsoft Visual C++ Restriction.	74
	Moving an Application and Its Schema	75
	Syntax	75
	Description	75
	Working with Virtual Function Table (VTBL) Pointers and Discriminant Functions.	77
	Relocation	78
	Missing VTBLs.	79
	Symbols Missing When Linking ObjectStore Applications.	79
	Run-Time Errors from Missing VTBLs.	80
	AIX C Set ++ — Virtual Function Table Pointers	82
	Using new and delete Operators with cfront	84
	Debugging Applications	85
	Dependency of Object Files on Header Files	86
	Retrofitting Makefiles	87
	UNIX.	88
	Linking with ObjectStore Libraries	88
	Examples of Passing Libraries to the Linker.	90
	DEC C++ 64-Bit Pointer Considerations.	91
	Troubleshooting Errors	101
	HP Requires Linker Options.	102
	+eh Mode Supported	102
	HP aC++ Source Files	102
	HP C++ Compiler Messages.	103
	SGI IRIX Compiler Option	103
	Sun C++ Compiler Options.	103
	Solaris 2 Linking	104
	Sample Makefile Template	105
	Using Signal Handlers	106
	Makefile for Building from Compilation Schemas	107
	Establishing Fault Handlers in POSIX Thread Environments	107

Virtual Function Table Pointers	108
Debugging Applications	108
Solaris C++ Search Paths	109
SGI Delta C++ Compiler	109
Windows	110
Linking with ObjectStore Libraries	110
Use Custom Build to Run oss g	110
Ensure That You Include Required Files	110
Make and Compiler Options	111
Use the Standard Run-Time Library on Windows NT	111
Linking Your Files	113
Sample Makefile	114
Sample Makefile for an Application That Uses Collections and Queries	115
Specifying Environment Variables	115
Debugging Your Application	116
Abnormal Application Exit	117
Building ObjectStore/Microsoft Foundation Class Applications	117
Class os_CString	119
Generating MFC Applications Using ObjectStore AppWizards	119
Using the Visual C++ Integrated Development Environment (IDE)	121
Using ObjectStore Within a DLL	121
Building Applications on Machines Remote from the Server	122
Porting ObjectStore Applications to Windows Platforms	123
Windows DEBUG and DDEBUG Builds of ObjectStore	124
Installing DEBUG.ZIP or DDEBUG.ZIP	125

Contents

OS/2	127
Using Compiler Options	127
Linking Your Files	128
Sample Makefile Without Library Schemas	129
Sample Makefile for an Application That Uses Collections and Queries	129
Debugging Your Application	130
Pass Source Files to ossg	131
Building Applications on Machines Remote from the Server	131
Chapter 5	
Building Applications for Use on Multiple Platforms	133
General Instructions	134
Which Platforms Can Be Heterogeneous?	136
When Is a Schema Neutral?	138
What Causes Data Formats to Vary?	138
How Can You Create Identical Data Formats?	139
Restrictions	140
Virtual Base Classes	140
Primitive Data Types	140
64-Bit Pointers	140
Floating-Point Data Conversion	141
Pointers to Members	141
Base Class Initialization Order	141
Parameterized Classes	141
Sizes for Data Types	143
General Restriction	144
ossg Neutralization Options	145

Neutralizing the Schema	148
When to Use Neutralization Options	149
Additional Neutralization Considerations	149
Updating a Database Schema to Be Neutral	150
Benefits of Compiler Groups	150
Command Line and Neutralization Examples	150
Using a Makefile to Obtain Neutralization Instructions	154
Building a Heterogeneous Application from a Neutral Schema	155
Neutralizing enums	155
Listing Nondefault Object Layout Compiler Options	156
Compiler Option File Format	156
Overriding Options Within the Compiler Option File	159
Sample Compiler Option File	160
Compiler Option File Example	160
Compiler Options That Aid Neutralization	160
Compiler Option Files for Architecture Sets	161
Description of Schema Generator Instructions	162
Base Class Padding Macros	162
Dynamically Defined Padding Macros	163
Member Padding Macros	163
Virtual Base Templates	164
Database Growth Resulting from Padding	165
Endian Types for ObjectStore Platforms	166

Contents

Chapter 6	Working with ObjectStore/Single	167
	ObjectStore/Single Features	168
	ObjectStore/Single API	168
	ObjectStore/Single Utilities	168
	Dynamic Library Load Path	169
	Application Development Sequence	170
	Server Log Propagation	171
	Server Log Functions	171
	Log File Guidelines	172
	When to Intervene	172
	Full ObjectStore oss erver Role	173
	Remote Access	174
	Accessing Server Logs and Cache Files Through NFS	174
	Packaging ObjectStore/Single Applications	175
	Cache and Server Log Files	175
	Additional Considerations	175
	Cache File Considerations	176
	Server Log File Considerations	176
	What Should You Tell Your Customers?	176
	Packaging an ObjectStore/Single Application	177
	Packaging with a VAR Product	177
	Index	179

Preface

Purpose	<i>ObjectStore Building C++ Interface Applications</i> provides information and instructions for generating schemas, compiling, linking, and debugging. This is for applications that use the ObjectStore C++ application programming interface (API). This book describes ObjectStore Release 5.1.
Audience	This book is for experienced C++ programmers who know how to build C++ applications on their platforms. Programmers who will be using makefiles are expected to be familiar with them.

How This Book Is Organized

The first half of this book provides information that applies to all ObjectStore platforms. The second half contains platform-specific chapters. For complete information, you must read the general chapters along with the chapter for your platform.

Notation Conventions

This document uses the following conventions:

<i>Convention</i>	<i>Meaning</i>
Bold	Bold typeface indicates user input or code.
Sans serif	Sans serif typeface indicates system output.
<i>Italic sans serif</i>	Italic sans serif typeface indicates a variable for which you must supply a value. This most often appears in a syntax line or table.

Convention	Meaning
<i>Italic serif</i>	In text, italic serif typeface indicates the first use of an important term.
[]	Brackets enclose optional arguments.
{ a b c }	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify <i>a</i> or <i>b</i> or <i>c</i> .
...	Three consecutive periods indicate that you can repeat the immediately previous item. In examples, they also indicate omissions.
 	Indicates that the operating system named inside the circle supports or does not support the feature being discussed.

ObjectStore Release 5.1 Documentation

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

Your use of ObjectStore documentation depends on your role and level of experience with ObjectStore. You can find an overview description of each book in the ObjectStore documentation set at URL <http://www.objectdesign.com>. Select **Products** and then select **Product Documentation** to view these descriptions.

Internet Sources of More Information

World Wide Web

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

Internet gateway

You can obtain such information as frequently asked questions (FAQs) from Object Design's Internet gateway machine as well as from the Web. This machine is called **ftp.odi.com** and its Internet

address is 198.3.16.26. You can use **ftp** to retrieve the FAQs from there. Use the login name **odiftp** and the password obtained from **patch-info**. This password also changes monthly, but you can automatically receive the updated password by subscribing to **patch-info**. See the **README** file for guidelines for using this connection. The FAQs are in the subdirectory **./FAQ**. This directory contains a group of subdirectories organized by topic. The file **./FAQ/FAQ.tar.Z** is a compressed **tar** version of this hierarchy that you can download.

Automatic email notification

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

Training

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

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

Your Comments

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

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

You can also fax your comments to 781.674.5440.

Preface

Chapter 1

Overview of Building an Application

This chapter provides an overview of building an ObjectStore application. The basic steps are the same, regardless of the compiler or platform you use.

The topics presented in this chapter are

General Instructions for Building Applications	2
Flow Chart for Building Applications	4
Third-Party Compilers You Can Use	5
Third-Party Libraries and Applications	6
Virtual File Systems	7
ObjectStore Server and the Build Process	9
ObjectStore / Single	10

General Instructions for Building Applications

An ObjectStore application is a C++ program that uses ObjectStore. For an application to use ObjectStore, you perform the following steps. These instructions assume that you are using a makefile to build your application.

1 Modify the source.

Modify your application source code to make ObjectStore API calls. See the *ObjectStore C++ API User Guide* for information about using ObjectStore APIs. Note that you must modify your makefile to find ObjectStore header files. See ObjectStore Header Files on page 15.

2 Create the schema source file.

The schema source file is a C++ file with a specified format used as input to the schema generator (**ossg**). The schema source file includes the files that define

- Classes that have instances stored by the application in persistent memory.
- Classes that have instances read by the application from persistent memory. You can include the type itself, or the base types of the class.
- Classes that appear in library interface query strings or index paths.

See Determining the Types in a Schema on page 19 for details.

3 Generate schema with **ossg**.

Modify your makefile to run the ObjectStore schema generator (**ossg**). The input for this step includes

- Schema source file
- ObjectStore library schemas

The output from this step is the

- Application schema database
- Application schema source file

If you are using Visual C++, the output is an object file referred to as the application schema object file. This file records the

location of the application schema database along with the names of the application's virtual function dispatch tables, the names of discriminant functions, and the definitions for any **get_os_typespec()** member functions.

See Chapter 3, Generating Schemas, on page 29, for further information.

4 Compile the application schema source file.

See Compiling, Linking, and Debugging Programs on page 73. Make sure your makefile enables you to compile the application schema source file. This creates the application schema object file.

When you use Visual C++, the schema generator creates the object file directly. On all other platforms, you must compile the application schema source file yourself.

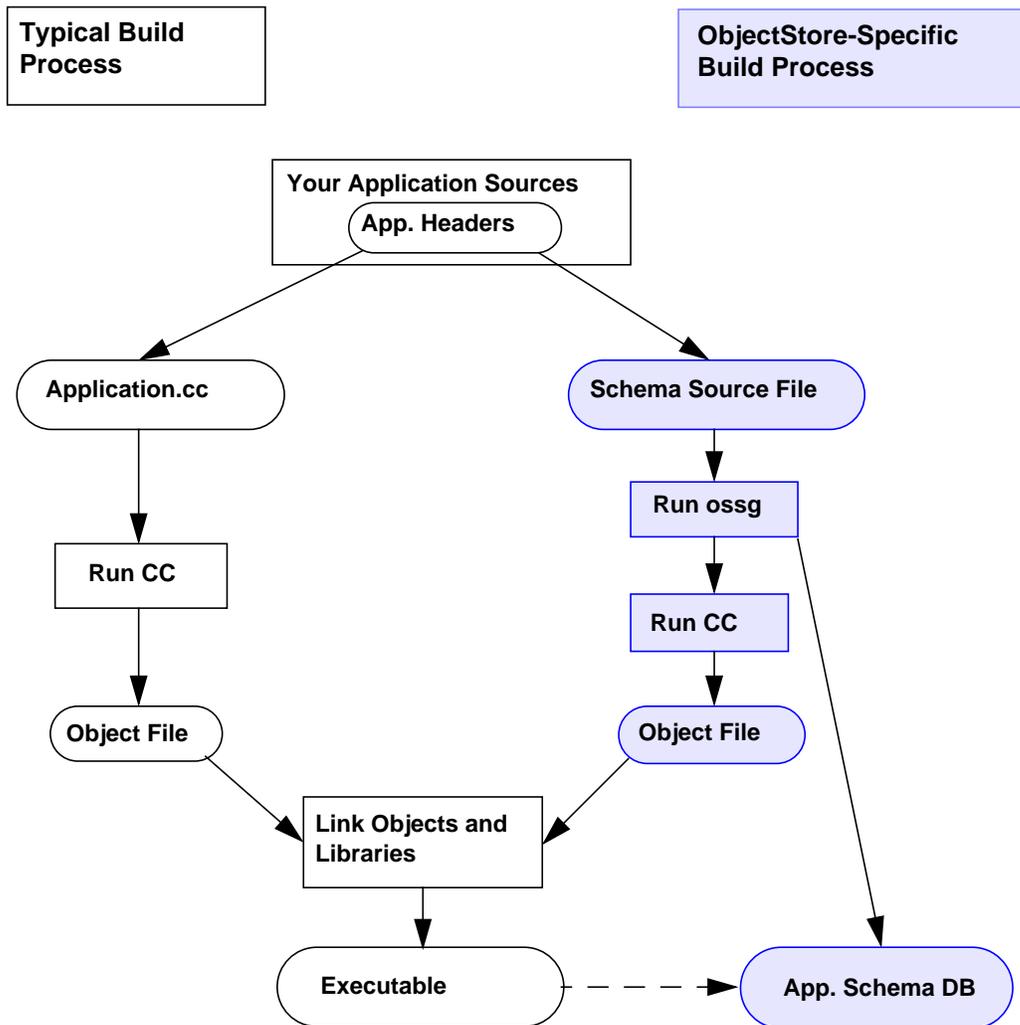
5 Link.

Make sure your makefile is modified to link the following (to create the executable):

- Application object files
- Application schema object file
- Application libraries
- ObjectStore libraries
- System libraries

Flow Chart for Building Applications

The workflow for building an ObjectStore application on one platform appears below. An understanding of how to build on one platform is essential to an understanding of how to build for multiple platforms. See Chapter 5, Building Applications for Use on Multiple Platforms, on page 133.



Third-Party Compilers You Can Use

Third-party compilers you can use include those in the following list. Consult the *ObjectStore C++ Interface Release Notes, Platforms and Compilers* in [Chapter 1](#) for the specific compiler version numbers supported by this release of ObjectStore.

- AIX C Set ++
- Digital UNIX DEC C++
- HP-UX HP C++
- IBM VisualAge C++ for OS/2
- SGI CC and NCC C++
- Sun SPARCompiler C++
- Solaris 2 Intel ProCompiler C++
- Microsoft Visual C++ 32-bit

Third-Party Libraries and Applications

If you use a third-party library or application and you want to use ObjectStore to store its types persistently, you must do one of the following:

- If it exists, obtain the vendor's version of the library or application and header files that use the ObjectStore API.
- Obtain the source code for the library or application and modify its calls where appropriate.

If the library provides the ability to write custom allocators, implement allocators that do persistent allocation.

When using a third-party library or application with an ObjectStore application, you must ensure that all persistent data is read/written inside a transaction.

Virtual File Systems

This section describes the use of ObjectStore with virtual file systems such as MVFS (part of ClearCase from Rational Software Corp.).

A virtual file system provides a user with a logical view of a file system. When you use a virtual file system, it appears as if all sources and executables reside in the current directory path, when only local modifications actually reside there. The purpose of a virtual file system is to hide the actual locations of files from users.

When you use ObjectStore with a virtual file system, you must specify pathnames that would work without the virtual file system. This is usually some kind of absolute pathname.

This requirement applies to any ObjectStore database stored under a virtual file system and includes any file that the Server must access. The Server must be able to access whatever pathname is provided. This can be a virtual file system pathname if the Server is running under a virtual file system.

You can run ObjectStore clients and utilities in directories where ObjectStore can find all files. However, the opening of a database is always performed by the Server. In a virtual file system, the Server cannot determine where a file actually resides unless you specify the true location.

You can run the Server under a virtual file system if there is only one *view* of the virtual file system and the view is shared by all users. It is not possible for the Server to recognize more than one view. In some virtual file systems, machines and users can have their own views.

You can maintain schema databases in a virtual file system because the schema generator embeds the absolute pathname of the schema in the database. However, when you generate schema databases you must specify pathnames that the Server can use.

Some virtual file systems, such as ClearCase, provide a way to return an absolute (nonvirtual) pathname for a given element in a virtual file system. This pathname can enable the ObjectStore Server to locate the element. It does mean the loss of the ability to

Virtual File Systems

use relative names (such as **eng_1.ldb** in the following example) as arguments to ObjectStore utilities.

Example

Here is an example of the type of message you might receive when ObjectStore cannot find a file. Notice that the file appears to be there.

```
> pwd
/home/clients/libschemas
> ls
test_1.ldb
eng_1.ldb
qa_1.ldb
> ossize eng_1.ldb
ossize: The database was not found
<err-0025-0351>The database
"/home/clients/libschemas/eng_1.ldb"
does not exist (err_database_not_found)
>
```

ObjectStore Server and the Build Process

The ObjectStore Server is involved in the build process only during schema generation. That is, the Server is involved only when you invoke **ossg** to generate an application, library, or compilation schema.

ObjectStore/Single

ObjectStore/Single is a stand-alone version of ObjectStore. It is a form of the ObjectStore client tailored for single-user nonnetworked use. The functional capability of an ObjectStore/Single application operating on file databases is virtually identical to that of other ObjectStore clients, and databases created with one kind of client are completely compatible with the other. However, you cannot run an application as both ObjectStore and ObjectStore/Single together on one machine because you must select a library load path as noted in the following discussion.

ObjectStore/Single includes the Server and Cache Manager functionality as part of the same library as the client application rather than as separate processes. Those who benefit from ObjectStore/Single are application developers who develop applications for a nonnetworked environment. Therefore it is most useful for applications that have no requirement for

- Concurrency
- Networked Server
- Rawfs support

ObjectStore/Single uses the same APIs as enterprise ObjectStore, and the rules for linking ObjectStore/Single applications are the same as for full ObjectStore, minimizing compatibility concerns for existing ObjectStore sites.

If you use dynamic library load paths, you can decide at execution time whether an application should be an enterprise ObjectStore or an ObjectStore/Single application. This allows you to develop applications using full ObjectStore, but package the application using ObjectStore/Single as a replacement. This replacement eases integration of embedded applications.

A summary comparison of full ObjectStore and ObjectStore/Single follows.

Full ObjectStore

Installation requires **root** permission.

ObjectStore/Single

Installation does not require **root** permission.

Full ObjectStore

Server is a separate process.

Cache Manager is a separate process.

Supports rawfs.

UNIX platforms use **OS_ROOTDIR/lib/libos** and **libosdbu** shared libraries.

Windows platforms use **%OS_ROOTDIR%\bin DLLs**.

OS/2 uses **%OS_ROOTDIR%\lib DLLs**.

Transaction log files are automatically created by Server during ObjectStore installation.

Cache files are automatically created by Cache Manager process.

osserver process automatically manages crash recovery and Server log propagation.

Uses Server and Cache Manager parameter files.

ObjectStore/Single

Server functions are integrated in single library.

Cache Manager functions are integrated in single library.

Does not support rawfs.

UNIX platforms use **OS_ROOTDIR/libsngr/libos** and **libosdbu** shared libraries.

Windows platforms use **%OS_ROOTDIR%\binsngl**.

OS/2 uses **%OS_ROOTDIR%\libsngr DLLs**.

User must specify transaction log files at each invocation of an application.

User must specify and clean up cache files.

User must ensure that unpropagated data in Server logs (for example, following a crash) is applied to the database by either

- Restarting the application
- Running the **osprop** utility

Does not use Server and Cache Manager parameter files.

See Chapter 6, Working with ObjectStore / Single, on page 167, for additional information about ObjectStore / Single.

ObjectStore/Single

Chapter 2

Working with Source Files

This chapter describes the source files you use to build ObjectStore applications.

The topics discussed are

Overview of Source Files	14
ObjectStore Header Files	15
Instantiation Problem with Template Collection Classes	16
Determining the Types in a Schema	19
Creating Schema Source Files	22
When You Modify a Source File	27

Overview of Source Files

You build an ObjectStore application from the following source files:

- Source files that contain code that you write.
- Header files, provided with ObjectStore, that you include in your source files.
- Header files that you write that define your persistent C++ classes.
- A schema source file that specifies your persistent classes for the schema generator. You create this file according to ObjectStore rules.

Building an ObjectStore application requires the generation of schema information. This is information about the classes of objects the application stores in or reads from persistent memory. ObjectStore generates schema information according to the schema source file that you create. See [Creating Schema Source Files](#) on page 22.

ObjectStore Header Files

ObjectStore provides header files that you must include in your source code. The ObjectStore features you use determine which header files to include. Be sure to include the files in the given order. You must always include `ostore/ostore.hh`.

<i>If You Use This Feature</i>	<i>Include These Header Files</i>
Any ObjectStore feature	<code>ostore/ostore.hh</code>
Collections	<code>ostore/ostore.hh</code> , <code>ostore/coll.hh</code>
Compactor	<code>ostore/ostore.hh</code> , <code>ostore/compact.hh</code>
Database utilities	<code>ostore/ostore.hh</code> , <code>ostore/dbutil.hh</code>
Metaobject protocol	<code>ostore/ostore.hh</code> , <code>ostore/mop.hh</code>
Relationships	<code>ostore/ostore.hh</code> , <code>ostore/coll.hh</code> , <code>ostore/relat.hh</code>
Schema evolution	<code>ostore/ostore.hh</code> , <code>ostore/manschem.hh</code> , <code>ostore/schmevol.hh</code>

Instantiation Problem with Template Collection Classes

Most AT&T cfront 3.0.1-based compilers have a problem correctly instantiating ObjectStore template collection classes. This is particularly true for HP C++. The problem manifests itself as an error report indicating that one of the following template collection classes is undefined at the time of instantiation:

```
os_Collection<your_class*>  
os_Set<your_class*>  
os_Bag<your_class*>  
os_List<your_class*>  
os_Array<your_class*>  
os_Cursor<your_class*>
```

The problem occurs even if you reference only one of the parameterized types in your application.

ObjectStore defines three preprocessor macros to help you work around this problem:

```
os_Collection_declare  
os_Collection_declare_no_class  
os_Collection_declare_ptr_tdef
```

Use these macros to declare the more common cases of ObjectStore template collection class forward definitions. Doing so works around the instantiation problem.

os_Collection_declare() Macro

The `os_Collection_declare` macro declares ObjectStore template collection classes that are parameterized by nontemplate classes. For example, if you intend to use `os_List<Person*>` in your application, you would use a statement of the following form in your source code:

```
os_Collection_declare(Person);
```

The macro automatically provides a forward definition of the class `Person`. There is no need for you to provide a full definition of the class `Person` at the point in the module where you use the `os_Collection_declare` preprocessor macro.

os_Collection_declare_no_class() Macro

Because `os_Collection_declare` is a text-substitution-based preprocessor macro, you cannot use it to work around the instantiation problem for ObjectStore template collection class parameters that are themselves template class instantiations.

In these cases, you must provide a forward definition of the template class and a typedef and use the `os_Collection_declare_no_class` preprocessor macro instead. For example:

```
template <int size> class Fixed_Array;
typedef Fixed_Array<5> Fixed_Array_5;
os_Collection_declare_no_class(Fixed_Array_5);
```

You also use the `os_Collection_declare_no_class` preprocessor macro to predeclare an ObjectStore template collection class parameterized by a fundamental type. For example, if you intend to use an `os_Array<int*>` in your application, you would include a statement of the following form in your source module:

```
os_Collection_declare_no_class(int);
```

os_Collection_declare_ptr_tdef() Macro

The `os_Collection_declare_ptr_tdef` preprocessor macro allows you to predeclare an ObjectStore template collection class parameterized by a typedef that names a pointer type. For example, you might define a typedef such as the following:

```
class Person;
typedef Person * pPerson;
```

To provide the necessary work-around declarations for the ObjectStore template collection class `os_Set<Person*>`, you can use a statement of the following form:

```
os_Collection_declare_ptr_tdef(pPerson);
```

Required Order of Include Statements

You must ensure that invocations of these macros appear in your source module before `<ostore/coll.hh>`. The following code works:

```
#include <ostore/ostore.hh>
os_Collection_declare(Person);
os_Collection_declare(Employer);
#include <ostore/coll.hh>
```

The following code does not work:

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
os_Collection_declare(Person);
os_Collection_declare(Employer);
```

ostore/semoptwk.hh

If your ObjectStore C++ application uses ObjectStore template collection classes and includes **ostore/mop.hh** or **ostore/schmevol.hh** or both, then you must also include **ostore/semoptwk.hh**. This header file provides invocations of the **os_Collection_declare** macro for types used in **ostore/mop.hh** and **ostore/schmevol.hh**.

As with your own invocations of the **os_Collection_declare** macro, the inclusion of **ostore/semoptwk.hh** must precede the inclusion of **ostore/coll.hh** in your source module. Also, if you include **schmevol.hh** or **mop.hh**, include them before **coll.hh**. For example:

```
#include <ostore/ostore.hh>
os_Collection_declare(Person);
os_Collection_declare(Employer);
#include <ostore/semoptwk.hh>
#include <ostore/schmevol.hh>
#include <ostore/coll.hh>
```

If you are not using ObjectStore template collection classes, it is not necessary for you to explicitly include **ostore/semoptwk.hh** even if your application uses **ostore/mop.hh** or **ostore/schmevol.hh** or both. Similarly, it is not necessary for you to explicitly include **ostore/coll.hh**, even though the schema evolution and metaobject protocol interfaces use ObjectStore template collection classes. When you are not using ObjectStore template collection classes in your application, the existing structure of the **ostore/mop.hh** and **ostore/schmevol.hh** header files is sufficient.

With Visual C++, there are additional considerations for building applications that use collections. See *Symbols Missing When Linking ObjectStore Applications* on page 79.

Determining the Types in a Schema

The schema source file determines the types that are in a schema. In the schema source file, you use a macro to mark the types to be included in the schema. After you run the schema generator, not only are these types in the schema, but any types that are *reachable* (defined below) from these types are also in the schema.

In other words, the types that you mark plus the types reachable from those types equal the types represented in the schema.

The types that you mark are the types on which you can perform persistent **new**. However, if you specify the **-make_reachable_source_classes_persistent (-mrscp)** option when you generate the schema, you can also perform persistent **new** on types in the schema source file that you did not mark. See **-mrscp** on the next page. See also *Generating an Application or Component Schema* on page 35.

Which Types to Mark

As a minimum, you should mark the following types:

- Classes on which the application might perform persistent **new** to create a direct instance of the class.
- Classes that have instances read by the application from persistent memory. You can mark the type itself or the base types of the class.
- Classes appearing in a query string or index path.

It is not necessary to mark ObjectStore classes except for collection classes.

Which Types Are Reachable

Type T is directly reachable from class C in each of these situations:

- T is in the definition of a member of C.
- T is a base class of C.
- C is a base class of T.
- T is nested in C.

- C is a template instantiation and T is an *actual* template parameter.
- C is a template instantiation and T is the class template for C.
- C is a class template and T is a *formal* template parameter.

Furthermore, a type that is directly reachable from a directly reachable type is considered to be reachable from the original type. For example, if T is directly reachable from C, and X is directly reachable from T, then X is reachable from C. There are no limits on the chain of reachability.

If You Are Not Sure a Type Is Reachable

When you want a type to be in the schema but you are not sure if the type is reachable, you can do either of the following:

- Mark it.
- Do not mark it. Run the schema generator. Then use the **osexschm** utility to determine if the type is in the schema. If it is not, you must mark it. For information about **osexschm**, see *ObjectStore Management, Chapter 4, osexschm: Displaying Class Names in a Schema*.

Choosing Whether to Mark Types or Specify `-mrscp`

If you mark a type, then you can perform a persistent **new** on that type. If you specify **-mrscp** (the shortened form of **-make_reachable_source_classes_persistent**) when you generate the schema, you can perform a persistent **new** on any type defined in the schema source file. Since the end result is the same, how do you choose whether to mark a particular type or to allow it to be persistently stored only through specification of **-mrscp**?

For each type that you mark, the schema generator does a certain amount of processing so that the type can be persistently stored. For each type that is reachable, but that is not itself marked, the schema generator does less processing, and the processing is not sufficient to allow persistent storage.

Suppose you specify **-mrscp** at schema generation time and then during execution you persistently store a type that you did not mark. At run time, additional processing is required so that you can persistently store such a type.

The benefit of specifying **-mrscp** is that it allows you to perform a persistent **new** for a type that you did not explicitly mark. The drawback is greater execution time and executable size overhead.

You should mark types that you persistently store. You can specify **-mrscp** in case you forget to mark a type that you persistently store.

Ensuring a Complete Schema Source File

Omissions in the schema source file can cause run-time errors. For example, you might try to persistently store a type that you did not mark or that is not in the schema. To avoid this, use the **osexschm** utility to ensure that all relevant types are in the schema.

Creating Schema Source Files

The schema source file specifies the C++ classes that your code reads from or writes to persistent memory. You create the schema source file according to a specified format. The schema source file can contain only valid C++ code. It is good practice to compile your schema source file to verify that it is compilable, but compilation is not required.

When you run the schema generator, you specify the name of the schema source file. Your executable program does not include the schema source file. The schema source file is only for input to the schema generator (**oss-g**).

Schema Source File Format

Before you create the schema source file, determine the types in your application that you are going to mark in the schema source file. Use the information in Determining the Types in a Schema on page 19 to help you decide. Then follow these steps to create a schema source file. See Schema Source File Examples on page 24 for sample code.

- 1 Create a text file.
- 2 In the text file, specify **#include** to include ObjectStore header files required by the features you use. The required order is on page 15.
- 3 Specify **#include** to include the **manschem.hh** file provided with ObjectStore.
- 4 Specify **#include** to include the files that define the following types:
 - The types that you are going to mark
 - Any types embedded in types that you are going to mark

You are not required to include the definitions for all reachable types. However, not including the class definition for a type that is in the application schema means that

- ObjectStore cannot check the class for compatibility with a database class definition (if one exists).

- ObjectStore cannot make virtual function table pointers (vftbls) and discriminant functions available for the class.
- Specifying the **-mrscp** option does not allow you to persistently allocate that type. The definition of the class must be in the schema source file or included directly or indirectly in the schema source file for **-mrscp** to allow that type to be persistently allocated.

For efficiency, create header files that contain only class definitions and include the header files in the schema source file. This speeds schema generation because there is nothing extra for the schema generator to examine.

- 5 Mark certain included types with a call to the macro **OS_MARK_SCHEMA_TYPE**.

Use the information on the previous pages to determine which types to mark. The order in which you mark types does not matter.

Each call is on its own line and has the format

```
OS_MARK_SCHEMA_TYPE(type-name);
```

OS_MARK_SCHEMA_TYPE is a preprocessor macro. For additional information about **OS_MARK_SCHEMA_TYPE()** and **OS_MARK_SCHEMA_TYPESPEC()**, see *ObjectStore C++ API Reference*, [Chapter 4, System-Supplied Macros](#).

- 6 Mark parameterized types with multiple arguments with a call to the macro **OS_MARK_SCHEMA_TYPESPEC**.

This macro is similar to **OS_MARK_SCHEMA_TYPE** in syntax and function, except that you must enclose the type and its arguments in parentheses.

Each call is on its own line and has the format

```
OS_MARK_SCHEMA_TYPESPEC((type-name<x,y>));
```

- 7 Save the schema source file.

Placing Calls to **OS_MARK_SCHEMA_TYPE** in a Function

In previous releases of ObjectStore, you were required to place the calls to **OS_MARK_SCHEMA_TYPE** or **OS_MARK_SCHEMA_TYPESPEC** in a dummy function.

This practice is now discouraged, and becomes obsolete in a future release. Since the default behavior is `-skip_function_body_parsing` (the `-sfbp` option), `ossg` does not see these marked types and prints a warning.

You can override the default using the `-parse_function_bodies` (`-hpfb`) option, but in general, placing calls to `OS_MARK_SCHEMA_TYPE` in a function is not good practice.

Schema Source File Examples

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/manschem.hh>
#include "ticket.hh" /*defines class ticket*/
#include "passenger.hh" /*defines class passenger*/
#include "schedule.hh" /*defines class schedule*/
OS_MARK_SCHEMA_TYPE(schedule);
OS_MARK_SCHEMA_TYPE(ticket);
OS_MARK_SCHEMA_TYPE(passenger);
```

As required, the `ostore.hh` and `manschem.hh` files are included. The `coll.hh` file is included because the application uses collections. For each marked type, the file in which it is defined is included. In this example, three classes, `schedule`, `ticket`, and `passenger`, need to be marked. Each included class is marked on its own line with a call to the `OS_MARK_SCHEMA_TYPE` macro.

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/dbutil.hh>
#include <ostore/manschem.hh>
#include "schmdefs.hh"
OS_MARK_SCHEMA_TYPE(drawing);
OS_MARK_SCHEMA_TYPE(view);
OS_MARK_SCHEMA_TYPE(layer);
OS_MARK_SCHEMA_TYPE(coordinates);
```

This schema source file includes the always required ObjectStore header files (`ostore.hh` and `manschem.hh`) along with the `coll.hh` and `dbutil.hh` header files, since the application uses collections and database utilities. It then includes the `schmdefs.hh` file that, in this example, contains the definitions of all classes in the application. The classes defined in `schmdefs.hh` that need to be marked are `drawing`, `view`, `layer`, and `coordinates`. Each of these classes is marked on its own line with a call to the `OS_MARK_SCHEMA_TYPE` macro.

Schema Source File Must Be Compilable

The schema source file can contain only valid C++ code. However, you should not use the schema generator to find errors in your schema source file. It is good practice to make sure that your schema source file compiles before you use it as input to **ossg**.

This is true even if you are using a C compiler. If you have valid C code that is invalid C++ code, you must modify it or hide it from the schema generator. For example, here is a valid C **struct** definition:

```
struct class
{
};
```

This is invalid C++ because **class** is a reserved keyword in C++. To hide such code, use comments or the **_ODI_OSSG_** macro. See Hiding Code from the Schema Generator on page 57 for more information about using this macro.

If you are generating persistent C types, for example, **structs**, and compiling them using a C compiler, then you must include those C types in the schema source file.

Most C constructs (including **struct** definitions) are valid C++ constructs, but some are not. If invalid C++ constructs appear in your included header files, you might have to use preprocessor directives to ensure that they are not visible in the schema source file. For example, specify the following in the schema source file to isolate the invalid code:

```
#ifndef __cplusplus
    /* invalid C++ code */
#endif
```

OS/2 Notes

Create a schema header file, also called a class definition file. In this file, place the definitions for all classes that you want in the application schema. While only one schema header file is allowed, it can contain **#include** statements for files that actually contain the class definitions. Include the schema header file in the schema source file. When you invoke **ossg** to generate the application

schema, you specify the **-cd** option with the name of this class definition file.

The reason for the class definition file is that the **icc** compiler does not make public a number of symbols that ObjectStore needs to link your persistent data to your program at run time. However, these symbols are available to files that include the class definitions. So the application schema source file produced by the schema generator must include the class definitions for all persistent classes your application might need. Therefore, an OS/2 schema source file contains

- **#include <ostore/ostore.hh>**
- **#include <ostore/manschem.hh>**
- **#include *schema_header_file***
- **OS_MARK_SCHEMA_TYPE** call for each type you want to mark

Sample OS/2 schema source file

For example, suppose an application's classes are defined in

- **mydefs1.hh**
- **mydefs2.hh**
- **mydefs3.hh**

You must create a single header file that contains

```
#include "mydefs1.hh"  
#include "mydefs2.hh"  
#include "mydefs3.hh"
```

If you name this header file **alldefs.hh**, you must pass **-cd alldefs.hh** to **ossg**. The schema source file would look something like this:

```
#include <ostore/ostore.hh>  
#include <ostore/manschem.hh>  
#include "alldefs.hh"  
OS_MARK_SCHEMA_TYPE(foo)  
OS_MARK_SCHEMA_TYPE(bar)
```

The schema generator then marks the schema types in the usual way. In addition, it places code in the generated file that arranges for vtbls for all the affected classes to be defined in the schema object file. This means they are accessible to ObjectStore at application run time. (Other platforms provide mechanisms for explicitly instantiating vtbls.)

When You Modify a Source File

Suppose you have already generated the schema for your application, and then you modify a type description. You must regenerate the application schema after changing

- A class that is marked in the schema source file
- A class that is reachable from a marked class

You must also regenerate the application schema when you add a library that has a library schema to your application. If you delete a library, you should regenerate the schema to remove the clutter.

You can set up rules in a makefile to automatically regenerate schema when required. Changes to source files have the same compiling and linking implications as they would in any other application.

If you make an incompatible change to the schema, and the old definition is present in the schema database you are regenerating, schema generation fails with an error message identifying the incompatible change. You can choose from three methods for handling this situation:

- When the incompatible change is intentional, and you do not need to access databases with the old definitions, delete the schema database and rerun **ossg** for successful schema generation.
- When the incompatible change was unintentional, and undesirable, reverse the changes to the source files and rerun **ossg** for successful schema generation.
- When the incompatible change is necessary and you need to access old databases created with older revisions of your schema, see [Chapter 8, Schema Evolution](#), in *ObjectStore Advanced C++ API User Guide* for specific details.

When You Modify a Source File

Chapter 3

Generating Schemas

This chapter provides instructions for using the schema generator to generate application, component, library, and compilation schemas.

Caution

Be sure you can successfully compile your code before you generate a schema. You should not use the schema generator to validate your code.

The topics discussed in this chapter are

Overview of Schema Generation	30
Generating an Application or Component Schema	35
Generating a Library Schema	53
Generating a Compilation Schema	55
Hiding Code from the Schema Generator	57
Unsupported Types	58
Restricting Use of Template Classes and Collections	59
Correcting Schema-Related Errors	60
Handling Pragma Statements in Source Code	63
Utilities for Working with Schemas	64
Comparison of ossg Command Lines	65
Comparison of Kinds of Schemas	66
Deploying Products with Protected Databases	68
Using Rogue Wave with Solaris Sun C++	69
ossg Troubleshooting Tips	70

Overview of Schema Generation

A schema contains information about a set of classes. ObjectStore defines these kinds of schemas:

- Application schemas
- Component schemas
- Library schemas
- Compilation schemas
- Database schemas

You use the ObjectStore schema generator to generate application, component, library, and compilation schemas. ObjectStore creates database schemas.

ObjectStore stores each application, component, library, and compilation schema in its own ObjectStore database. ObjectStore stores database schemas in the associated database or in a separate database that you specify.

Each schema database must be accessible to an ObjectStore Server.

Application Schemas

An application schema contains descriptions of

- Classes the application stores in or reads from persistent memory
- Classes that a library that your application links with stores in or reads from persistent memory

ObjectStore uses the application schema during run time to

- Determine the layout of objects being transferred between the database and the application
- Validate the database schema to ensure that the application schema matches the database schema

For simple applications, you can use a single invocation of **ossg** to generate an application schema. In more complex applications, you might need to use library schemas to store schema information before constructing the application schema.

Component Schemas

A component schema is a type of application schema that can be loaded and unloaded dynamically at run time. Typically, a component schema is also a self-contained schema associated with a DLL.

The rules for generating and using component schemas are identical to those for application schemas, with these differences:

- Multiple component schema can be in effect at the same time in a single program.
- Batch schema installation is not supported. You must use incremental schema installation with component schema.

Library Schemas

If your application uses a library that stores or retrieves persistent data, and the library does not supply its own component schema, use the schema generator to create a library schema for that library. When you generate the application schema, you specify the library schema. This allows the schema generator to generate an application schema that contains information for all persistently used types.

It is particularly important that a library schema contain definitions of persistently allocated types that users of the library do not have access to.

In addition to the library schemas you create, ObjectStore provides library schemas for its libraries that use persistent data. If you link your application with an ObjectStore library that has a library schema, you must specify that library schema when you generate the application schema. See *Specifying ObjectStore Library Schemas* on page 48.

Compilation Schemas

A compilation schema works like a library schema. A compilation schema contains information about the application's persistent types, but does not contain information about any persistent types used by any libraries that the application links with. Earlier releases of ObjectStore on some platforms required a compilation schema before you could generate an application schema. With ObjectStore Release 5.0 and later, you still can create a compilation

schema and generate the application schema from the compilation schema, but it is no longer required.

Database Schemas

ObjectStore creates a database schema from the application and component schemas of all applications that allocate objects in the database. The database schema consists of the definitions of all types of objects that have ever been stored, or are expected to be stored, in the database.

Normally, ObjectStore stores a database schema in segment 0 of the database. (In each database, segment 0 is a special segment that is reserved for ObjectStore use.)

You can, however, specify an alternative database to contain the database schema. You do this when you create the database. The database whose schema is stored in another database is referred to as a remote schema database. The database that contains the schema belonging to the remote schema database is referred to as the schema database. The schema for a remote schema database resides in segment 0 of the schema database.

An application augments a database schema through batch (the default) or incremental schema installation.

With batch schema installation, the first time an application accesses a database, each class in the application's schema that can be persistently allocated is added to the database's schema (if it is not already present in the database schema). Subsequent execution of the application does not install schema in that database unless the application's schema changes (as evidenced by a change in the internally stored date of the application schema database).

With incremental schema installation, a class is added to a database's schema only when the first instance of that class is allocated in the database. You can specify incremental schema installation for a particular database in your source code.

The following table is a simple comparison of batch and incremental schema installation:

Batch Schema Installation	Incremental Schema Installation
Schema is larger.	Schema is smaller.

Batch Schema Installation

Administrative work is done at the beginning.

Incremental Schema Installation

Administrative work is done in steps; there is potential for concurrency conflict because each incremental schema installation modifies some pages and so uses some write locks.

Keeping Database Schemas and Application Schemas Compatible

Each ObjectStore database has its own database schema. Each With the exception of component schema, anObjectStore application is associated with one application schema or multiple component schemas. Many users can share an application schema and use it with different applications. When an application opens a database, the application's schema must be *compatible* with the database's schema. Compatibility means that if a class exists in both schemas,

- Its data members must have identical definitions and ordering in each schema.
- Both class definitions must either define at least one virtual function, or virtual functions must be absent from both definitions.

ObjectStore flags differences as schema validation exceptions.

Multiple Component Schemas

Multiple component schemas can be used in one application.

If two program schemata that are loaded into the same complete program schema define types with the same name, the type definitions must be identical.

This is only checked for types that appear in the schema of a database that is in use. An error will show up as a schema validation error.

When a new type is installed into a database schema, it will be validated against all program schemata and a schema validation error will be signaled if there are multiple inconsistent definitions.

Schemas Are Platform Specific

After you generate a schema, you can use it only on the platform on which you generated it. If you try to generate an application

Overview of Schema Generation

schema from one or more schemas that were built on other platforms, **ossg** aborts the process and displays a message indicating why. If you want to use a library schema on multiple platforms, you must generate it on each platform on which you want to use it.

Consider that an application schema corresponds to an executable, a component schema corresponds to a DLL, a library schema corresponds to a library, and a compilation schema corresponds to an object file. Just as executables, libraries, and object files are platform specific, so are their accompanying schema databases.

Generating an Application or Component Schema

Input To generate an application schema, specify your schema source file as input to **ossg**. If you are linking with libraries that have library schemas, you must also specify those library schemas as input to **ossg**. This procedure is also true for component schema.

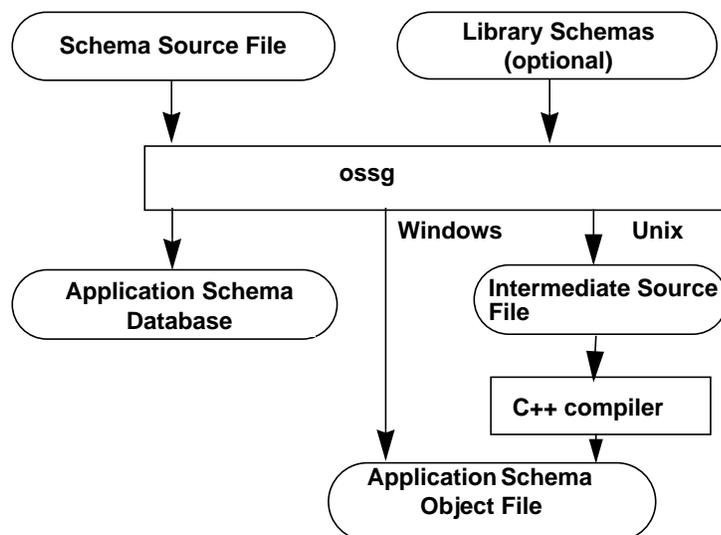
Output The output from **ossg** is

- Application schema database. The schema generator creates the application schema and stores it in an ObjectStore database.
- Intermediate schema source file. This file records the location of the application schema database along with the names of the application's virtual function dispatch tables, the names of discriminant functions, and the definitions for any **get_os_typespec()** member functions.

You must compile this file and then link the resulting object file with your application. (Chapter 4, *Compiling, Linking, and Debugging Programs*, on page 73, provides the details for linking.)

When you use **ossg** on Windows platforms, the schema generator creates the application schema object file directly.

Following is an illustration of the application schema generation process, extending to just before the link step.



Invoking ossg to Generate an Application Schema

The **ossg** command syntax for generating application, library, and compilation schemas appears below. Portions of the command line are on different lines only to make them more understandable.

```
ossg [neutralizer_options] [additional_app_options] [{"-assf"
app_schema_source_file | "-asof app_schema_object_file.obj"}]
-asdb app_schema_db schema_source_file [lib_schema_db1 ... lib_
schema_dbn]-cd class_definition_file
```

```
ossg [neutralizer_options] [additional_options] -lsdb lib_schema_db
schema_source_file
```

```
ossg [neutralizer_options] [additional_options] -csdb comp_schema_db
schema_source_file
```

neutralizer_options: [-arch setn...] [{"-padm | -padc"}] [-ostyp] [-nout file_name][{"-showd | -showw"}] [-nor] [-sopt]

additional_app_options: [{"-mrlcp | -mrscp"}] [-rtdp] [-no_weak_symbols][{-weak_symbols} [additional_options]]

additional_options: [-no_default_includes] [-mrscp] [additional_options]: [{"-smf"}] [{"-sfbp | -pfb"}]

```
ossg [compilation_options] [neutralizer_options] [-cpp_fixup]
```

<i>compilation_options</i>	<p>[-final_asdb <i>final_app_schema_db</i>] [-E <i>schema_source_file</i>]</p> <p>Specifies any options that would be passed to the compiler if you were compiling a schema source file instead of generating schema from it. You should include any preprocessor options such as include file paths and macro definitions, as well as compiler options that might affect object layout, such as packing options (for example, /Zp4 for Visual C++). Optional.</p> <p>If you specify the /I, -I (uppercase I), /D, or -D option, do not include a space between the option and the argument. For example, on OS/2 the following is correct:</p> <p>ossg /\$(OS_ROOTDIR)include ...</p> <p>On UNIX, do not specify the -o option on the ossg command line.</p> <p>On Windows, do not specify /Tp on the ossg command line.</p> <p>On OS/2, when you specify any option that takes an argument do not put a space between the option and the argument.</p>
<i>neutralizer_options</i>	<p>Include any of the options in ossg Neutralization Options on page 145. These options allow you to neutralize a schema for a heterogeneous application. You can include them in any order.</p>
-arch set<i>n</i>	<p>Optional. The default is that neutralization is not done. The schema that is generated or updated will be neutralized to be compatible with the architectures in the specified set. See ossg Neutralization Options on page 145 for explanations of the set options.</p> <p>Required when you are neutralizing schema. No default.</p>

-cpp_fixup

Allows preprocessor output to contain spaces inside C++ tokens. Specify this option if your preprocessor inserts a space between consecutive characters that form C++ tokens. For example, if your preprocessor changes :: to : ., you can specify this option so that the schema generator allows the inserted space and correctly reads the preprocessor output.

You might generate an application schema from a compilation schema and library schemas. In this case, you do not need this option because there is no source code input to the schema generator, which means that the preprocessor is not involved.

Optional. The default is that **ossg** does not allow a space in a C++ token such as :: or .*.

-E *schema_source_file*

The **-E** option causes **ossg** to preprocess the input file and send the preprocessed output to standard output.

This option is useful for debugging **ossg** parsing problems, because it allows you to see the results of any preprocessing that might have occurred without generating the schema. It is also useful when you report **ossg** problems to Technical Support because it allows the problem to be reproduced by Object Design without the need to package all your application's **include** files.

When specifying the **-E** argument, if you also specify schema databases on the same command line, a warning is issued:

<warn-0038-0006> The option -E which generates a preprocessed source has been specified. No schema will be generated from this command.

-final_asdb *final_app_schema_db*

Specifies a location for the application schema database that is different from the location you specify with the **-asdb** option. The schema generator writes the location you specify with the **-final_asdb** option into the application schema source file (application schema object file for Visual C++). Use this option when you cannot specify the desired location with the **-asdb** option. The **-asdb** option is still required and that is where the schema generator places the application schema.

This option is useful when you plan to store the application schema database as a derived object in a ClearCase versioned object base (VOB). The schema generator cannot place the application schema database directly in a ClearCase VOB. If you specify the **-final_asdb** option with the desired location, you avoid the need to run the **ossetasp** utility, which patches an executable so that it looks for its application schema in a database that you specify.

You can also use this option to specify a relative path.

After you run **oss** with the **-final_asdb** option, remember to move the application schema to the database you specify with **-final_asdb**.

You must specify an absolute pathname with **-final_asdb**.

Optional. The default is that the schema generator writes the pathname that you specify for **-asdb** in the application schema source file (object file for Visual C++).

Note: This option works for component schemas that are generated with **-asdb**, as well.

-mrlcp or **-make_reachable_library_classes_persistent**

Causes every class in the application schema that is reachable from a persistently marked class to be persistently allocatable and accessible.

This option is supplied for compatibility purposes only. The use of the **-mrlcp** option is discouraged. Specify **-mrscp** instead.

When you specify this option, you cannot neutralize the schema for use with a heterogeneous application. If you are building a heterogeneous application, you must either mark every persistent class in the schema source file or specify the **-mrscp** option.

If you do not mark any types in the schema source file and you specify **-mrlcp** when you run **oss**, then the application schema does not include any types. You must mark at least one type for there to be any reachable types.

Optional. The default is that only marked classes are persistently allocatable and accessible.

See also *Determining the Types in a Schema* on page 19.

-mrscp or **-make_reachable_source_classes_persistent**

Causes every class that is either

- Defined in the schema source file
- Reachable from a persistently marked class

to be persistently allocatable and accessible.

The difference between **-mrscp** and **-mrlcp** is that when you specify **-mrscp**, it applies to the schema when **ossg** is translating from source to schema. This allows the schema generator to recognize which types you plan to allocate persistently. The **-mrlcp** option applies to the application schema after the merging of constituent schemas.

The benefit of specifying the **-mrscp** option is that it allows you to perform a persistent **new** for a type that you did not explicitly mark in the schema source file. The drawback is greater execution time and executable size overhead.

If you do not mark any types in the schema source file and you specify **-mrscp** when you run **ossg**, then the application schema does not include any types. You must mark at least one type for there to be any reachable types.

Optional. The default is that only marked classes are persistently allocatable and accessible.

See also Determining the Types in a Schema on page 19.

-no_default_includes or
-I-

When you specify this option, **oss**g does not automatically specify any include directories to the C++ preprocessor. However, the preprocessor can have default include directories built in. If there are any directories that are built into the preprocessor, **oss**g does check these built-in directories. Typically, the preprocessor uses built-in include paths to find standard include files such as **stdio.h**.

When you specify this option, you must explicitly specify directories that contain included files.

For example, on some UNIX systems, when you do not specify this option, the C++ preprocessor looks for include files in the **/usr/include** directory.

Note that if you want the schema generator to pass the ObjectStore include directory to the preprocessor as a directory for finding included files, you must always specify it. For example:

UNIX: **-I\$OS_ROOTDIR/include**

Windows and OS/2: **-I%OS_ROOTDIR%\include**

The **-I-** option is the letter I as in Include. Specifying **-I-** is the same as specifying **-no_default_includes**.

Optional. The default is that the preprocessor checks default directories for included files.

-nor

A neutralizer option that prevents the schema generator from instructing you to reorganize your code as part of neutralization. This is useful for minimizing changes outside your header file, working with unfamiliar classes, or simply padding formats.

See **oss**g Neutralization Options on page 145 for more information.

-neutral_info_output filename or
-nout filename

A neutralizer option that indicates the name of the file to which neutralization instructions are directed.

Optional. Default is that the schema generator sends output to **stderr**.

-no_weak_symbols

Default. Notifies you of missing vftbls and discriminants, allowing you to check whether a referenced vtbl or discriminant function symbol is undefined.

If you specify **-rt dp maximal -no_weak_symbols**, the linker outputs messages about what is missing. You can use this information to determine which additional classes you need to mark. These missing symbols are only a hint about what you might consider marking. They might also be the result of a link line error.

Specify the option **-weak_symbols** to suppress this behavior.

-pad_maximal or **-padm** /
-pad_consistent or **-padc**

Neutralizer options that indicate the type of padding requested. See *ossg Neutralization Options* on page 145 for additional information.

Optional. Default is **-padc**.

-parse_function_bodies or **-pfb**

This option ensures that any types that are marked inside a function are parsed by **ossg**. If you do not explicitly use this option and you have any types marked inside functions, an error is reported. See *ossg Troubleshooting Tips* on page 70 for further information.

Optional. The default is that the **-sfbp** option is set.

-runtime_dispatch or -rtdp { minimal derived full maximal }	Specifies the classes for which the schema generator makes vftbls and discriminant functions available. minimal specifies marked classes, classes embedded in marked classes, and base classes of marked classes. derived specifies the minimal set plus classes that derive from marked classes and classes embedded in the derived classes. full specifies the derived set plus the transitive closure over base classes, derived classes, and classes that are the targets of pointers or references. The full specification does not include nested classes or enclosing classes unless they meet one of the previous criteria. maximal specifies the full set plus nested types. In previous ObjectStore releases, this was the default. If your application used an earlier release of ObjectStore and you do not specify this option, you might need to mark classes that you did not previously mark. See Example of Using the -runtime_dispatch Option on page 50. Optional. The default is derived .
-show_difference or -showd -show_whole or -showw	Neutralizer options that indicate the description level of the schema neutralization instructions. Optional. Default is -show_whole .
-skip_function_body_parsing or -sfbp	Optional. Specifies that code within function bodies is not parsed. By default this option is in effect.
-schema_options <i>option_file</i> or -sopt	A neutralizer option that specifies a file in which you list compiler options being used on platforms other than the current platform. See Listing Nondefault Object Layout Compiler Options on page 156 for further information.

- store_member_functions** or **-smf** Causes **ossg** to create an instance of **os_member_function** for each member function in each class in the schema source file. It then puts these instances in the list of class members, which includes member types and member variables.
- This is useful when you intend to use the MOP to inspect the member functions. If you are not planning to inspect member functions, you should not specify this option, because it wastes disk space.
- This also means that additions and deletions of member functions are schema changes and affect validation.
- When you generate an application schema, you might specify a library or compilation schema. If you want to capture the member functions from the library or compilation schema, you must have specified the **-store_member_functions** option when you generated the library or compilation schema. You must also specify the **-store_member_functions** option when you generate the application schema.
- Optional. The default is that **ossg** generates a schema that includes member types and member variables, but not member functions.
- weak_symbols** Suppresses notification about missing vftbls and discriminants. This option overrides the default behavior described at **-no_weak_symbols** on page 43.
- assf** *app_schema_source_file* or
-asof *app_schema_object_file.obj* Specifies the name of the application schema source file or application schema object file to be produced by **ossg**. For all compilers except Visual C++, the schema generator produces a source file that you must compile. When you use Visual C++, the schema generator directly produces the object file.
- Required. No default.

-asdb *app_schema_db.adb*

Specifies the name of the application schema database to be produced by **ossg**. If the schema database exists and is compatible with the type information in the input files, the database is not modified.

This pathname must be local to a host running an ObjectStore Server.

The pathname should have the extension `.`. If you want to specify an existing application schema database with **ossg**, the application schema must have `.adb` as its extension.

Required. No default.

schema_source_file

Specifies the C++ source file that designates all the types you want to include in the schema. It should include all classes that the application uses in a persistent context.

No default. Required except in two cases:

- When you generate an application schema from a compilation schema as described on page 56
- If you specify one or more library schemas that contain all the persistent types that your application uses

lib_schema.ldb ...

Specifies the pathname of a library schema database. The name must end in `.ldb`. This can be an ObjectStore-provided library schema or a library schema that you created with **ossg**.

The schema generator reads schema information from the library schema database specified and modifies the application schema database to include the library schema information. You can specify zero or more library schema databases.

Optional. The default is that library schemas are not included.

Changing the Default Preprocessor

Except on OS/2, you can use the **OS_OSSG_CPP** environment variable to specify a C preprocessor other than the configured default. The following table shows the default preprocessor on each platform:

OS/2	icc
------	------------

UNIX	cpp
Windows	cl

OS/2 Platforms Have an Additional ossg Option

On OS/2 platforms, when you invoke **ossg**, you must specify the **-cd** option with the name of the class definition file for the application. The class definition file, also called the schema header file, contains the definitions for all classes that you want in the schema. While only one class definition file is allowed, it can contain **#include** statements for files that actually contain the class definitions. You must include the class definition file in the schema source file.

The reason for this option is that the **icc** compiler does not make public a number of symbols that ObjectStore needs to link your persistent data to your program at run time. However, these symbols are available to any file that includes your class definitions. Consequently, the application schema source file produced by the schema generator must include the class definitions for all persistent classes your application might need. For example:

```
ossg -assf ossschema.cpp -cd schmdefs.hh -asdb myschema.adb \
myschema $(OS_ROOTDIR)\lib\os_coll.lib
```

Using a Temporary File to Send Arguments to ossg

You can specify the following option on an **ossg** command line:

@filename

It does not matter what kind of schema you are generating.

When the schema generator encounters this option, it reads additional options and arguments from **filename**. Note that the command line passed to the compiler for preprocessing is the fully expanded command line, so those options that are passed to the compiler must still meet command-line length restrictions.

Be sure that you do not insert a space between **@** and **filename**. You can specify this option on any platform.

On OS/2 and Windows NT, this is commonly used to avoid problems with the length of a command line in Microsoft's **nmake** program. The usual **nmake** syntax is

```
ossg -other-args @<<  
$(LONG_ARGS1)  
$(LONG_ARGS2)  
<<
```

Specifying ObjectStore Library Schemas

ObjectStore provides library schemas for the ObjectStore libraries that store or retrieve persistent data. If you will be linking your application with an ObjectStore library that has a schema, you must specify that library schema when you generate the application schema. The following table shows when to specify an ObjectStore-provided library schema. When you specify a library schema, you must always specify the full path. For example, `$(OS_ROOTDIR)/lib/liboscol.ldb` on UNIX and `$(OS_ROOTDIR)\liblos_coll.ldb` on Windows and OS/2.

<i>For This Feature</i>	<i>Specify This Library Schema</i>		<i>Link with This Library</i>	
	<i>UNIX</i>	<i>Windows or OS/2</i>	<i>UNIX</i>	<i>Windows or OS/2</i>
Collections	<code>liboscol.ldb</code>	<code>os_coll.ldb</code>	<code>-loscol</code>	<code>ostore.lib</code>
Compactor	<code>liboscmp.ldb</code>	<code>oscmpct.ldb</code>	<code>-loscmp</code>	<code>ostore.lib</code>
Queries	<code>libosqry.ldb</code>	<code>osquery.ldb</code>	<code>-losqry</code>	<code>ostore.lib</code>
Schema evolution	<code>libosse.ldb</code>	<code>ossevol.ldb</code>	<code>-losse</code>	<code>ostore.lib</code>
	<code>libosqry.ldb</code>	<code>osquery.ldb</code>	<code>-losqry</code>	
	<code>liboscol.ldb</code>	<code>os_coll.ldb</code>	<code>-loscol</code>	

Specifying Remote Library Schemas

A client that uses a remote Server uses ObjectStore library schemas that are stored on the remote Server. Consequently, when generating the application schema, be sure to specify absolute pathnames for library schemas.

How Long Does Schema Generation Take?

The time it takes to generate an application schema is difficult to predict. There are many variables that affect how long schema generation takes, such as the platform and the size of the schema. Usually, the time to generate your schema is comparable to the time to compile the equivalent source code.

If You Omit a Required Library Schema

If you generate an application schema without specifying a required library schema, you might not notice the omission until the application fails with a message indicating that classes are missing from the application schema.

If the collections library schema is required, and you omit it, the linker typically displays a message indicating that there are missing `get_os_typespec` functions. If `os_tinyarray::get_os_typespec()` is among them, this probably indicates that you failed to include the collections library schema while building the application schema.

If the types in a library are not persistently used,

- It might not matter if the corresponding library schema is not specified when the application schema is generated. For example, there might be types that are persistent in some situations but not in others. Also, if there are `get_os_typespec()` functions, the library schema is required. Consequently, Object Design recommends that you always specify a library schema when you use a type in that library, whether or not the types are persistently used.
- The database schema might be larger than necessary when batch schema installation is used.

Missing collections
class library example

Here is an example that shows what happens when there is a missing collections library schema. The exact mode of failure differs on each platform. This example was generated on Sun using the ProCompiler.

Here is a trivial application:

```
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
main ()
{
    OS_ESTABLISH_FAULT_HANDLER
    objectstore::initialize();
    os_collection::initialize();
    OS_END_FAULT_HANDLER
    return 0;
}
```

As you can see, it does not do much, but it does make a reference to the collections library. Because this application does not

persistently create or access any types, the schema source file is trivial:

Schema source file

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

Build the application schema with the usual command (forgetting to explicitly include the collections library schema):

```
oss -assf os_nullschm.cc -asdb nullschm.adb nullschema.cc
```

Compile the application (`test.cc`) and the application schema source file (`os_nullschm.cc`) and link the application as shown. As you can see, there are a large number of undefined `get_os_typespec` functions that are missing (about 100 lines of messages are not shown) because the collections library schema was not included while `oss` was building the application schema.

Running the ProCompiler

```
CC -pta -vdelx -mt -o test test.o os_nullschm.o -loscol -los -losthr
Undefined symbol first referenced in file
static _RH_ref<_RH_ref_slot_SHV>::get_os_typespec(void)
...
...
```

The exact message varies depending on which compiler you are using, but the elements common among the messages are the undefined `get_os_typespec` functions.

Example of Using the `-runtime_dispatch` Option

The `-runtime_dispatch` (`-rtdp`) option specifies the set of classes for which the schema generator makes vftbls and discriminant functions available. For example, suppose you define the following classes:

```
class A {
public:
    class B { public: G* gp; };
    E* ep;
};
class C : public A { };
class D : public A { };
class E { };
class F : public D { public: E e; };
class G { public: E e; };
class H { };
```

If you mark class D, the following classes are in the set (that is, their vftbls and discriminant functions are available). Note that

class H is not reachable from class D and consequently is not in the set unless you explicitly mark it.

Option Specified	Classes in the Set
Minimal	A and D
Derived	A, D, E and F
Full	A, C, D, E and F
Maximal	A, B, C, D, E, F and G

Using the Same Application Schema for Multiple Applications

Multiple applications can share a single application schema. You can set this up by following these steps:

- 1 Generate the application schema that you want the applications to share.
- 2 Compile the application schema source file produced by the schema generator. (If you are using Visual C++, the compiler directly produces the application schema object file.)
- 3 Link the application schema object file into each application that you want to use this schema. When applications share an application schema object file, they consequently share the application schema.

When the first application accesses the database, ObjectStore validates the application schema. When subsequent applications access the database, ObjectStore does not need to perform validation and access is faster.

Examples of Generating an Application Schema

UNIX

```
ossg -assf myschema.cc -asdb myschema.adb schema_source.cc
```

The **-assf** option indicates the name of the application schema source, **myschema.cc**. You must specify a name for the application schema source file. There is no default. The **-asdb** option indicates the name of the database (**myschema.adb**) to contain the application schema. The schema source file is **schema_source.cc**.

```
ossg -mrscp -assf myassf.c -asdb my.adb mine.c my.ldb your.ldb
```

The **-mrscp** option is included, so all classes that are defined in the schema source file and that are reachable from marked classes are persistently allocatable and accessible. This is true even if they are

Generating an Application or Component Schema

not marked in the schema source file. The **-assf** option indicates that **myassf.c** is the name of the application schema source file to be produced by **ossg**. The **-asdb** option indicates that **my.adb** is the name of the application schema database to be generated. The schema source file is **mine.c**. The application schema is generated with the **my.ldb** and **your.ldb** library schema databases and consequently will include the types in those libraries.

Windows

```
ossg -asof jetsch.obj -asdb jetsch.adb jetsch.cc \ %OS_  
ROOTDIR%\lib\os_coll.ldb
```

Since this is a Windows platform, the schema generator directly produces the application schema object file. The **-asof** option indicates the name of the application schema object file, **jetsch.obj**. The **-asdb** option indicates that the application schema database to be generated by **ossg** is **jetsch.adb**. The schema source file is **jetsch.cc**. Because this application uses ObjectStore collections, the collections library schema, **lib\os_coll.ldb**, is specified.

OS/2

```
ossg -assf paris.cc -asdb paris.adb paris_sch_source.cc\  
-cd paris.hh %OS_ROOTDIR%\lib\oscmpct.ldb
```

On OS/2, you must specify the **-cd** option with the name of the class definition file. The application schema source file that **ossg** produces will be named **paris.cc**. The application schema database will be in **paris.adb**. The schema source file is **paris_sch_source.cc**. The class definition file is **paris.hh**. The application uses the compactor feature, so you must specify the library schema for the compactor library, **oscmpct.ldb**.

Generating a Library Schema

If you create a library that allocates or reads persistent data, you should create a library schema. A library schema contains descriptions of the types that the library stores or retrieves in a persistent context.

You specify the library schema when you generate the application schema for applications that use the library. ObjectStore adds the types defined in the library schema to the application schema.

In addition to the library schemas you create, there are library schemas that ObjectStore provides. Specifying ObjectStore Library Schemas on page 48 describes library schemas provided with ObjectStore.

Makefiles

A makefile that generates a library schema is almost the same as a makefile that generates an application schema. The major differences are that when you want to generate a library schema you

- Specify the **-lsdb** option on the **ossg** command line. Do not specify the **-asdb** option.
- Do not specify the **-assf** or **-asof** option. Generating a library schema does not involve generating the application schema source (or object) file.

For an example of a makefile that generates an application schema, see the section discussing your platform in Chapter 4, *Compiling, Linking, and Debugging Programs*, on page 73.

Invoking ossg to Generate a Library Schema

Refer to *Invoking ossg to Generate an Application Schema* on page 36 for the forms of the **ossg** command to generate a library schema. On OS/2, the **-cd** option is only required when you generate an application schema.

Using Multiple Schema Source Files to Create a Library Schema

You can create a library schema from multiple schema source files. To do this, you incrementally build the library schema from one schema source file at a time. For example, to include

Generating a Library Schema

information from the schema source files **s1.cc** and **s2.cc** in the library schema **foo.ldb**, you would use two commands:

```
ossg -lsdb foo.ldb s1.cc  
ossg -lsdb foo.ldb s2.cc
```

This creates the **foo.ldb** library schema. It contains the types marked in each of the schema source files **s1.cc** and **s2.cc**.

If you modify a source file from which you created a library schema and run **ossg** again, the schema generator adds to the library schema. It does not overwrite the existing library schema. If you do not want to add to the existing library schema, you must remove the old schema or specify a name for a new library schema.

For example, if you modify **s1.cc** after you create **foo.ldb** and then you run **ossg** again, **ossg** adds to **foo.ldb**. It does not overwrite **foo.ldb**.

Example

```
ossg -lsdb part.ldb partschm.cc
```

This command creates the **part.ldb** library schema and stores it in an ObjectStore database. The **part.ldb** library schema contains descriptions of the types marked in the **partschm.cc** schema source file. When you generate an application schema for an application that will link with the **part** library, you must specify the **part.ldb** library schema.

If you specify neutralization arguments when you generate a library schema, you must specify the same neutralization arguments each time you run **ossg** to update that library schema.

If Not Creating a Library Schema

Your application needs access to all types that are persistently allocated. These should be marked either in the application schema or the library schema. If they are not marked in one or the other, providing access to the necessary header files can be quite complex.

Generating a Compilation Schema

When the schema generator generates an application schema, it internally creates a compilation schema first and then builds the application schema from the compilation schema.

A compilation schema contains information about the classes in your application's source files that are read from or written to persistent memory. A compilation schema differs from an application schema in that the compilation schema does not include information about the classes used by libraries your application links with.

You can use **ossg** to explicitly generate a compilation schema before you generate the application schema.

A makefile that generates a compilation schema is almost the same as a makefile that generates an application schema. The major differences are that when you want to generate a compilation schema you

- Specify the **-csdb** option on the **ossg** command line. Do not specify the **-asdb** option.
- Do not specify the **-assf** or **-asof** option. Generating a compilation schema does not involve generating the application schema source (or object) file.

For an example of a makefile that generates an application schema, see the section for your platform in Chapter 4, *Compiling, Linking, and Debugging Programs*, on page 73.

Why Generate a Compilation Schema?

There is no requirement for you to generate a compilation schema. However, you might want to generate a compilation schema in the following situations:

- When building a very large application. You can split the schema source file into multiple files so that you need not regenerate the entire schema when one type description changes.
- When performing schema evolution. See *ObjectStore C++ API User Guide*, [Chapter 8, Schema Evolution, Generating a Compilation Schema](#) for additional information.

Invoking ossg to Generate a Compilation Schema

Use the syntax described in Invoking ossg to Generate an Application Schema on page 36 to create a compilation schema. On OS/2, the **-cd** option is not required when you generate a compilation schema.

Example

```
ossg -csdb my.cdb myschema.cc
```

The name of the compilation schema that **ossg** produces is **my.cdb**. The schema source file is **myschema.cc**.

Using a Compilation Schema to Generate an Application Schema

To generate an application schema from a compilation schema, specify the name of a compilation schema in place of the schema source file when you invoke **ossg**. For example:

```
ossg -assf engine_schema.cc -asdb engine.adb libmine.cdb
```

On Windows, you would enter

```
ossg -asof engine_schema.obj -asdb engine.adb libmine.cdb
```

OS/2

On OS/2, you must specify the **-cd** option when you generate an application schema from a compilation schema. See OS/2 Platforms Have an Additional ossg Option on page 47.

Hiding Code from the Schema Generator

You can use the `_ODI_OSSG_` `cpp` macro to hide code from the schema generator. This is useful

- When your source files include code that is
 - Compiler specific (not ANSI C++)
 - Not compilable by `ossg`; for example, code that has an error but the code is not pertinent to schema generation
- To speed up `ossg`

Specify the `_ODI_OSSG_` macro in a directive in your source code. For example:

```
#ifndef _ODI_OSSG_  
/* code you do not want the schema generator to see */  
#endif
```

When you run the schema generator, `ossg` passes your code to the C preprocessor (`cpp`) before it generates the schema. When `ossg` does this, it also passes a definition for `_ODI_OSSG_` to `cpp`. Consequently, `_ODI_OSSG_` is always defined. The result is that `cpp` removes the code between the two directives (`#ifndef _ODI_OSSG_` and `#endif`) from the code that it passes back to the schema generator. The schema generator never operates on the code between the directives.

Unsupported Types

The following paragraphs describe types unsupported in ObjectStore Release 5.1.

Limited Support for long long Data Type

A **long long** is an **int** type with a length of 64 bits. DEC C++ , SPARCompiler C++, and SGI C++ treat **long long** as a new type.

On platforms that support **long long**, the schema generator recognizes this type in a source fed through it. On platforms that do not support **long long**, the schema generator signals an error when it encounters this type.

However, you cannot perform a persistent **new** for an object that includes the **long long** type. In other words, you cannot mark a type in the schema source file that includes a **long long** member. An error occurs during schema generation if you do.

Support for wchar_t Types

The schema generator does not fully support the **wchar_t** type. Consequently, Object Design discourages the use of this type.

You can persistently store an object that has a **wchar_t** member type. However, code that requires implicit conversions or type promotions that involve **wchar_t** is not guaranteed to be recognized by the schema generator. You might need to conditionalize such code so that the schema generator ignores it.

You cannot neutralize a schema that includes the **wchar_t** type. The schema generator treats **wchar_t** types in the same manner as the architecture/compiler platform for which it is generating schema. Different platforms treat this type in different ways.

Schema evolution and queries of classes containing **wchar_t** types are not supported.

Restricting Use of Template Classes and Collections

There are two ObjectStore preprocessor macros that you can pass to **oss**g to restrict the use of ObjectStore template classes and collections.

- **__NO_TEMPLATES__**
- **OS_NO_COLLECTION_TEMPLATES**

ObjectStore header files contain numerous definitions and uses of template classes. Specify one of these macros on the **oss**g command line when you do not want ObjectStore to define class templates.

The use of these macros does not affect your ability to define and use your own template classes. These macros hide only the definitions of ObjectStore templates.

__NO_TEMPLATES__

When you specify **__NO_TEMPLATES__** on the **oss**g command line, before the specification of any ObjectStore include files, template classes normally defined by ObjectStore are not available for use by your application. These templates include collection templates, reference templates, and relationship templates. The advantage of this is that compilation time is faster. The disadvantage is that you need to use the generic versions of the classes, which do not provide the security of type safety.

OS_NO_COLLECTION_TEMPLATES

When you specify **OS_NO_COLLECTION_TEMPLATES** on the **oss**g command line, your application can use parameterized references and relationships but not parameterized collections.

It is not possible to allow the use of collections without allowing the use of template classes.

Caution

The **mop.hh** and **schmevol.hh** header files use template classes unconditionally. Do not specify the **__NO_TEMPLATES__** macro or the **OS_NO_COLLECTION_TEMPLATES** macro when your application uses the MOP or schema evolution.

Correcting Schema-Related Errors

This section presents information about schema-related errors that you might need to resolve:

- Type mismatch errors
- Persistent allocation errors
- **ossg** run-time errors
- Metaschema mismatch errors

Type Mismatch Errors

The most common schema-related error can occur when an attempt is made to reconcile type definitions from different sources. This can happen when ObjectStore builds any kind of schema.

Similarly, you might get an exception when an application is run against a database, if the class definitions in the application schema are incompatible with class definitions already present in the database schema. When this happens, you might have to either change the application to match the database or evolve the schema database.

When an error occurs due to a type mismatch, the easiest way to get more information about the mismatch is to use the **osexschm** utility. See *ObjectStore Management, Chapter 4, osexschm: Displaying Class Names in a Schema*.

Persistent Allocation Errors

The following exceptions might occur at run time:

```
<err-0025-0022> Persistent new requested for type "XXX", which has not  
been marked as a legitimate type for persistent new.  
<err-0025-0021> Persistent new requested for type "XXX", which was not  
found in the application schema.
```

ObjectStore detects these errors when the application attempts to persistently allocate a class that was not part of its application schema or a class that was not marked. Common sources of this error are

- The persistent allocation was done in a library and the library schema was not supplied to **ossg**.

- **ossg** was not used.
- **ossg** was used, but the type was not marked with a call to the **OS_MARK_SCHEMA_TYPE** macro.

Use the **osexschm** utility to establish the absence of the class in the appropriate schema. See *ObjectStore Management*, [Chapter 4](#), [osexschm: Displaying Class Names in a Schema](#).

ossg Run-Time Errors

When the schema generator builds an application schema, errors can occur at link time if the class definitions present in the compilation and library schemas are not compatible.

When the schema generator builds a compilation schema, errors can occur because a class definition used in a persistent context in one file differs from the class definition of a class with the same name in another file. This situation normally causes a compile-time error.

For library and compilation schemas, you can determine how **ossg** handles type-mismatch errors during schema generation. Set the environment variable **OS_COMP_SCHEMA_CHANGE_ACTION** to one of the following values:

warn	Reports a warning. The new type definition replaces the previous definition in the compilation schema. Default.
silent	Not reported. The new type definition replaces the previous definition in the compilation schema.
error	Reports an error. The schema generation is eventually terminated, and the compilation schema remains unchanged.

Metaschema Mismatch Errors

The metaschema consists of class definitions internal to ObjectStore that are used to describe user classes. A metaschema mismatch usually indicates an incorrectly built program or database, where the program, libraries, and databases do not correspond to the same ObjectStore release of the software. For this reason, you are most likely to see these errors at ObjectStore release boundaries.

Correcting Schema-Related Errors

Schema Neutralization Errors

For information about handling schema neutralization errors, see *ossg Neutralization Options* on page 145.

Missing Virtual Function Table Pointer Problems

For information about missing virtual function tables, see *Run-Time Errors from Missing VTBLs* on page 80.

Handling Pragma Statements in Source Code

The schema generator recognizes the **#pragma** statements that it encounters in your schema source files and interprets them as the compiler would in most cases. However, for pragma statements that occur inside a class definition, **oss-g** does not usually treat them as the compiler would.

In particular, a **#pragma** statement that is nested in a class does not take effect until the start of the next nonnested class. For nonpersistent classes, this might not be a problem. For persistent classes, be sure that all **#pragma** statements that affect structure layout occur outside the class. Otherwise, the object layout defined by the compiler and the ObjectStore schema might be inconsistent.

Utilities for Working with Schemas

The following ObjectStore utilities help you manage schemas:

- **osexschm** lists the classes in an application, component, compilation, database, or library schema.
- **osscheq** compares schemas.
- **ossetasp** patches an executable to use a specified application or component schema database. This utility is described on page 75. OS/2 does not support this utility.
- **ossetrsp** sets the pathname of a remote schema database.
- **ossevol** updates a database schema to reflect modifications to your application or component schema.

Information about all these utilities appears in *ObjectStore Management*, [Chapter 4, Utilities](#).

Comparison of **ossg** Command Lines

Each time you run **ossg**, you can generate one kind of schema.

Kind of Schema

Application

Syntax for ossg Command

```
ossg [compilation_options] [neutralizer_options]
[-cpp_fixup] [-final_asdb final_app_schema_db]
[{-mrlcp | -mrscp}] [-no_default_includes] [-no_weak_symbols]
[-rtdp {minimal | derived | full | maximal}]
[-store_member_functions]
{-assf app_schema_source_file or -asof app_schema_object_file.obj}
-asdb app_schema_database schema_source_file
[lib_schema.ldb ...]
```

On OS/2, you must also specify

```
-cd class_definition_file
```

Library

```
ossg [compilation_options] [neutralizer_options] [-cpp_fixup] [-mrscp]
[-no_default_includes] [-store_member_functions]
-lsdb lib_schema.ldb schema_source_file
```

Compilation

```
ossg [compilation_options] [neutralizer_options] [-cpp_fixup] [-mrscp]
[-no_default_includes] [-store_member_functions]
-csdb comp_schema.cdb schema_source_file
```

After you generate a compilation schema, you can use the compilation schema to generate the application schema. In this scenario, the name of the compilation schema replaces the name of the schema source file in the **ossg** command syntax for generating an application schema. The syntax is

```
ossg {-assf app_schema_source_file | -asof app_schema_object_
file.obj}
-asdb app_schema_database.adb comp_schema.cdb
```

Conventions for application schema source file names

In previous releases of ObjectStore, **.os_schema.cc** and **.os_schema.o** were often used as the names for the application schema source file and the application schema object file. This was always a convention and not a requirement. The initial dot hid the files from directory listings.

This convention is not observed in Release 5.1. The Release 5.1 convention is to use names without the initial dot. Again, this is a convention and not a requirement.

Comparison of Kinds of Schemas

Kind of Schema	What Does It Contain?	How Is It Generated?	When Is It Used?
Application			
You specify a database in which to store it.	Definitions of classes the application stores in or reads from persistent memory, and classes persistently used by libraries the application links with.	Specify the -asdb option when invoking ossg . Naming convention recommended for an application schema: <i>yourchoice.adb</i> ¹	ObjectStore uses the application schema during run time to determine the layout of objects being transferred between the database and the application.
Component			
Library			
You specify a database in which to store it. ¹	Definitions of types that the library stores or retrieves in a persistent context.	Specify the -lsdb option when invoking ossg . Naming convention required for a library schema: <i>yourchoice.ldb</i>	You can specify a library schema when you generate an application schema for an application that uses the corresponding library. ObjectStore adds the types defined in the library schema to the application schema.
Database			
ObjectStore stores it in the database itself or in a remote database if you specify one when you create the database.	Definitions of all C++ types that have ever been, or are expected to be, stored in the database.	ObjectStore creates a database schema from application schemas of the applications that access the database. An application augments a database schema through batch (the default) or incremental schema installation.	When an application accesses a database, ObjectStore checks to make sure that the definitions of classes that are in both the application schema and the database schema are the same.

Kind of Schema	What Does It Contain?	How Is It Generated?	When Is It Used?
Compilation	Definitions of classes the application stores in or reads from persistent memory. (It does not include classes used by libraries the application links with.)	Specify the -csdb option when invoking ossg . Explicitly creating a compilation schema is optional. Naming convention recommended for a compilation schema: <i>yourchoice.cdb</i>	If you explicitly create a compilation schema, you specify it to generate the application schema. ObjectStore always creates a compilation schema, and uses it to create the application schema.

¹ If you want to specify an existing application schema database with **ossg**, Object Design recommends that the application schema have **.adb** as its extension.

Deploying Products with Protected Databases

If you want to restrict access to a database's data and metadata, you can use ObjectStore's schema protection facility. This facility allows you to associate a *schema key* (a pair of integers) with a database. After you associate a database with a schema key, an application must supply the key to access data in the database. A database with a schema key is considered to be a protected database. See *ObjectStore C++ API User Guide*, [Chapter 7, Database Access Control](#), [Schema Keys](#).

When you deploy a product that generates a schema for a protected database, you must write an application that does the following:

- 1 Programmatically sets the environment variables **OS_SCHEMA_KEY_LOW** and **OS_SCHEMA_KEY_HIGH** to the correct schema key. See *ObjectStore Management*, [OS_SCHEMA_KEY_LOW](#).
- 2 Spawns a child process that
 - Inherits the settings for the **OS_SCHEMA_KEY_LOW** and **OS_SCHEMA_KEY_HIGH** variables
 - Generates a schema for the protected database

The reason your application must include these characteristics is that the schema generator is not a database utility that you can call programmatically.

Using Rogue Wave with Solaris Sun C++

When you use Sun C++ on a Solaris 2 system, you might encounter the following problem when you try to generate a schema and your schema source file includes Rogue Wave header files.

```
ossg -csdb fiddle:/home/bow/aa.cdb -I$OS_ROOTDIR/include
a.cc"/opt/SUNWspro/SC3.0.1/include/CC/rw/tislist.h", line 216: Error:
Trying to open encrypted file
"/opt/SUNWspro/SC3.0.1/include/CC/rw/tislist.cc" while
preprocessing."/opt/SUNWspro/SC3.0.1/include/CC/rw/tislist.h", line
216: Error: Could not open include file "
rw/tislist.cc"."/opt/SUNWspro/SC3.0.1/include/CC/rw/tpslist.h", line 244:
Error: Trying to open encrypted file
"/opt/SUNWspro/SC3.0.1/include/CC/rw/tpslist.cc" while
preprocessing."/opt/SUNWspro/SC3.0.1/include/CC/rw/tpslist.h", line
244: Error: Could not open include file "
rw/tpslist.cc".
```

The Sun C++ 4.0.1 package includes Rogue Wave header files. However, the template functions are in encrypted form in accompanying `.cc` files. If you include the `.hh` file in a compilation, everything works fine. If you include the `.hh` file in a schema source file, the schema generator tries to preprocess the source file using `CC`. But if you do not have a source license, the encrypted source file causes an error.

There are two work arounds for this:

- Buy a Rogue Wave source license.
- Modify the schema source file to avoid including the encrypted sources, which are not needed anyway. In the schema source file, include the following two lines before including any other header files:

```
#include <rw/compiler.h>
#undef RW_COMPILE_INSTANTIATE
```

These two lines cause only the class declarations to be seen and not the member functions.

ossg Troubleshooting Tips

The following information provides guidelines for avoiding problems when using **ossg**.

Precompile

Do not use **ossg** to find syntax and semantic errors in your source files. Compile your sources with the compiler before running **ossg** over your source files.

Member function declarations

Function declarations can be complex. For example, are template functions instantiated using inheritance or nesting? **ossg** never needs to see nonmember functions. There are only three circumstances requiring **ossg** to see and process member function declarations:

- If a class has at least one virtual member function, **ossg** needs to see at least one virtual member function declaration so that it can install virtual function table (vtbl) information in the schema correctly.
- If there are discriminant functions (for unions) **ossg** must see them so they can be installed in the schema.
- If the **-store_member_functions** option of **ossg** is specified, the member function declarations must be visible to **ossg**.

If none of these conditions apply, functions (including member functions) can be hidden from **ossg** using conditional compilation or exclusion.

Skip function body parsing

In ObjectStore 5.0 and subsequent releases, by default **ossg** skips all function body parsing and processing (the **-sfbp** option is the default). If function bodies are not parsed, **ossg**'s process time and use of heap storage during execution will decrease from previous releases. The only reason to process function bodies is if a user has marked types for persistent **new** inside function bodies. If a user has done so, **ossg** will not see those types and will issue the following warning:

<warn-0038-0003>One or more types have been marked in a function body, but function bodies are not being parsed. These types will not be marked in the schema. Please move the marking of all types outside of function bodies (preferred) or specify -parse_function_bodies.

The two options available to the user in this case are to move the **OS_MARK_SCHEMA_TYPE** macros outside function bodies or to specify the **-parse_function_bodies** option.

Modularize schema information

Just as it is good programming practice to modularize code, it is preferable to separate schema information into separate compilation or library schemas that can be combined as needed to create application schemas for specific applications.

Swap space

Parsing and analysis of C++ code can be very complex. For generation of schema, **ossg** sometimes uses large amounts of swap space, so Object Design recommends that you make a large amount of swap space available when running **ossg**.

What to send to Technical Support

When a problem occurs during schema generation (such as syntax errors, semantic errors, or any other errors), you should send the following to Object Design Technical Support :

- A copy of the problem code in the form of preprocessed source
- The error information
- Platform and release information

This information will probably be needed to diagnose the problem. To produce the preprocessed source, run **ossg** over the source (as if you were generating schema), using the **-E** option and redirecting the output to a file. Since you are not generating schemas, **ossg** will ignore all schema generation-related options and only use the pertinent options such as **-I** and **-D**.

For example:

```
ossg -E -D... -I... source.cc > source.ii
```

ossg Troubleshooting Tips

Chapter 4

Compiling, Linking, and Debugging Programs

This chapter provides information about compiling, linking, and debugging your application.

The earlier topics address all platforms. The last three topics are each dedicated to a specific platform or group of platforms.

Using Standard Template Libraries	74
Moving an Application and Its Schema	75
Working with Virtual Function Table (VTBL) Pointers and Discriminant Functions	77
Missing VTBLs	79
Using new and delete Operators with cfront	84
Debugging Applications	85
Dependency of Object Files on Header Files	86
Retrofitting Makefiles	87
UNIX	88
Windows	110
OS/2	127

Using Standard Template Libraries

In the ObjectStore development environment, you use the schema generation utility **oss**g to create schema for the classes that need to be stored persistently. The front end of **oss**g is actually a C++ language parser. As input, it takes a source file that includes C++ header files, which in turn define the layout of the application objects. You must inform **oss**g which of those classes you want to *mark* so that you can create persistent instances of them at a later time.

ObjectStore can now be used to store STL objects persistently. There are a number of STL class library implementations available that can be used with ObjectStore. For example, ObjectSpace, Rogue Wave, and Visual C++ offer such implementations.

How to Store STL Types Persistently

In order to store STL types persistently, take the following steps:

- 1 Include the STL header files as well as the ObjectStore headers in the schema source file, and then mark the STL types. See [Creating Schema Source Files](#) on page 22.
- 2 Provide an STL allocator for persistent storage. The ObjectStore distribution does not include allocators, but does include an example that shows you how to create your own.

An example of an STL allocator using ObjectStore is also included in the directory `$OS_ROOTDIR/examples/stl`. Also see the discussion of standard template libraries in [Persistent new and delete](#) in [Chapter 2, Persistence](#), of the *ObjectStore C++ API User Guide*.

Microsoft Visual C++ Restriction

ObjectStore Release 5.1 supports ObjectSpace STL, but does not yet support Visual C++ version 5.0 STL.

Moving an Application and Its Schema



Syntax

The **ossetasp** utility patches an executable so that it looks for its application schema in a database that you specify.

ossetasp -p *executable*

ossetasp *executable database*

-p Instructs **ossetasp** to display the pathname of the specified executable's application schema database. Do not specify *database* in the command line when you include **-p**.

executable Specifies the pathname of an executable. On Windows systems, this can also be the pathname of a DLL.

database Specifies the pathname of an application schema database. ObjectStore patches the specified executable so it uses this application schema.

Description

When the schema generator generates an application schema, ObjectStore stores the actual string given as the **-asdb** argument to **ossg** (or the **-final_asdb** argument, if specified). When the application starts, it uses that string to find the application schema database.

When you move or copy an ObjectStore application to a machine that is not running a Server, leave the application schema database on the Server host. Normally, the application schema database must be local to the Server.

After you copy or move an application to another machine, you must patch the executable so that it can find the application schema database. Run the **ossetasp** utility with the absolute pathname of the application schema database. Be sure to specify the name of the Server host.

A locator file allows a database and its application schema to be on a machine other than the Server host. See *ObjectStore Management*, [Chapter 5, Using Locator Files to Set Up Server-Remote Databases](#).

Moving an Application and Its Schema

Windows NT

On Windows NT systems, you can run the **ossetasp** utility on any executable or DLL that contains schema (that is, that has a schema object file produced by **ossg** linked into it).

Restrictions

This utility is available on all platforms except OS/2. On OS/2, as well as on all other platforms, you can use [objectstore::get_application_schema_pathname\(\)](#). See the *ObjectStore C++ API Reference*, [Chapter 2, Class Library](#), for details.

Working with Virtual Function Table (VTBL) Pointers and Discriminant Functions

There are two special cases in which ObjectStore needs to know, at run time, the locations of information in your application program's executable:

- Virtual function tables (vtbls)
- Union discriminant functions

Vtbls

When you declare a class to have virtual functions or, in some cases, to have virtual base classes, it acquires an invisible data member, the *virtual function table pointer*. (*Virtual function table* is usually abbreviated as *vtbl*, pronounced *veetable*. On some platforms vtbls are called *vfts* for virtual function tables. *Vtbl* and *vft* indicate the same thing.) The vtbl points to a table of function pointers that the application uses to dispatch calls to virtual functions. The C++ compiler arranges for the correct function pointers to be placed in the virtual function table.

Persistent storage

When you persistently store an object belonging to a class with virtual functions, ObjectStore cannot store the vtbl pointer literally, since it is a pointer to the text or data segment of the current executable — a transient pointer. When the same program is run another time, or a different program opens the database, the vtbl might have a different location.

Discriminants

For some union types used persistently, ObjectStore requires that you provide an associated *discriminant function*, which indicates the field of the union currently in use. The function is used by the application at run time when a union is brought into virtual memory from persistent storage. (See [Discriminant Functions](#) in the *ObjectStore Advanced C++ API User Guide* for additional information about discriminant functions.) To handle a discriminated union, ObjectStore must know the address of the union discriminant function for the union. This is similar to the way that ObjectStore handles vtbls.

Relocation

When ObjectStore reads in an object with virtual functions, it supplies an appropriate vtbl pointer from the current application. This is called *vtbl relocation*.

When your application references a persistent object of a class with virtual functions, ObjectStore must fill in the vtbl pointer in the object. To fill in the vtbl pointer, ObjectStore must know the address of the vtbl for the class. Virtual function tables are not stored in databases; they are part of your executable.

During relocation, ObjectStore might need vtbls and discriminant functions. It finds them in tables that map class names to references to both vtbls and discriminant functions. The schema generator generates a C++ source file (or object file for Visual C++) containing these tables that relate your schema to your application.

These tables are filled in during application link or postlink or at program start-up time, or some combination of these, depending on the platform. At each of these steps, the referenced vtbls and discriminants are searched for in the executable and, if found, are entered into the tables. At run time, ObjectStore can use these tables to find items for relocation.

On cfront platforms, the `os_postlink` executable performs this job. On other platforms, the compiler does it. On some platforms, this search might be done at run time based on the currently available DLLs.

Missing VTBLs

Depending on your platform, missing vtbls can cause errors at compile time or at run time. It is better to find such errors at compile time. By default **ossg** reports these errors at compile time. An optional **ossg** flag, **-weak_symbols**, can be used to suppress the default behavior. See Chapter 3, Generating Schemas, on page 29, for detailed information about **ossg** syntax.

Symbols Missing When Linking ObjectStore Applications

Sometimes when linking, particularly with optimizations enabled, you are told that various symbols required by the application schema object file are missing. These symbols on non-Windows platforms (including OS/2) begin with `__vtbl` or `__vft`, or end with `__vtbl`. On Windows the symbols begin with `??_7`. This happens because ObjectStore needs access to the virtual function tables (vfts) for some classes, and the C++ compiler does not recognize these tables as being needed. The easiest way to get these symbols is to add a nonstatic dummy function such as the following

```
void foo_force_vfts(void*){
    force_vfts(new A);
    foo_force_vfts(new B);
    ...
}
```

Creating instances of a class causes the class's vfts, as well as those of bases that have out-of-line default constructors, to be created in this file.

Abstract classes

If one of your classes is abstract, a variant of the above approach is needed, since you cannot allocate an abstract class. You can provide an out-of-line constructor for the class, or you can allocate a nonabstract derived class in such a way that inline constructors are used for the abstract class. For example, if the original class definitions were

```
class A {
    virtual void foo() = 0;
};
class B : public A {
    virtual void foo();
};
```

then class A might be missing its vft. However, an unoptimized `new B` would call A's inline default constructor, which would reference the vft for A. But if class B had an out-of-line constructor, this would not work. Then it would be easiest to make an out-of-line constructor for A:

```
class A {
    virtual void foo() = 0;
    A(){}
    friend void force_vfts(void*);
    A(void*);
};
class B : public A {
    virtual void foo();
};
```

and define `A::A(void*){} in some file.`

Instantiating
collection classes

If you are using a parameterized collection class, you must instantiate the other collection classes because they have casts to each other. A work around is to declare this and link it. For example:

```
void foo_force_vfts(void*) {
    foo_force_vfts(new os_Set<missing-type>);
    foo_force_vfts(new os_List<missing-type>);
    foo_force_vfts(new os_Array<missing-type>);
    foo_force_vfts(new os_Bag<missing-type>);
    foo_force_vfts(new os_Collection<missing-type>);
}
```

There are additional considerations for building applications that use collections. See Instantiation Problem with Template Collection Classes on page 16.

Run-Time Errors from Missing VTBLs

On some platforms (without weak symbol support), you find out about the missing vtbls at link time. The vtbls are marked in the schema output file, but are not marked in the application. This is frequently the case for parameterized collections classes (`os_Set`, `os_List`, and so on).

Sometimes an executable does not have vtbls for all classes with virtual functions in the schema. When a vtbl pointer for a class is not available, ObjectStore fills in the vtbl pointer for the class's instance with a special vtbl that signals an error when any of the virtual functions is called.

No constructor	Missing vtbls can occur when your application calls a virtual function on an instance of a class for which no constructor call appears in the source. Since a call to the class's constructor does not appear in the source, the linker does not recognize the class as being used and does not link in its implementation. But an ObjectStore application can use a class whose constructor it never calls by reading an instance of the class from a database. To avoid this situation, put a call to the class's constructor inside a dummy function that is never called.
Class not in schema	Missing vtbls can also occur when the class is not included in the application's schema, either because its definition was not included in the source or because the class was only reachable from explicitly marked classes by means of void* pointers. In this case, the solution is to include a definition of the class, or explicitly mark it with OS_MARK_SCHEMA_TYPE() .
Inline virtual functions when using nonfront compilers	<p>Nonfront compilers include C Set ++, DEC C++, SGI C++, Sun C++, VisualAge C++, and Visual C++.</p> <p>When you are using a nonfront compiler (except on OS/2), if all virtual functions of a class are inline, either because they are defined fully in the class specification or with the inline keyword, the compiler treats the virtual function table as static. Because the virtual function table is viewed as static, the vtbl pointers for such a class are not available (that is, not seen globally, therefore not available to ObjectStore) because the locations of the virtual functions were not filled in by ossg.</p> <p>The solution to this problem is to put an out-of-line virtual function in each class with a missing vtbl. You can either modify an inline function or you can add a trivial noninline function. To determine which classes need an out-of-line virtual function, you can run ossg with the -no_weak_symbols option. This identifies missing vtbls at link time rather than at run time. For information about this option, see page 43.</p> <p>A class that meets the following conditions might also need the addition of a noninline virtual function:</p> <ul style="list-style-type: none">• Has at least one virtual base class• Has at least one virtual function defined by a base class• Does not define any virtual functions

Missing VTBLs

This is because the vtbl from the base class might become invalid as a result of the derivation.

Visual C++

When you are using Visual C++, an alternative solution is to export a class that has only inline virtual functions.

OS/2

On OS/2, correct vtbls are available for classes that have only inline virtual functions because you run **ossg** with the **-cd** option.

-rtdp option

Missing vtbls can also occur depending on what you specify for the **-rtdp** option when you generate the schema.

To obtain a list of missing vtbls at run time, set the **OS_TRACE_MISSING_VTBLs** environment parameter. See *ObjectStore Management*, [OS_TRACE_MISSING_VTBLs](#).

AIX C Set ++ — Virtual Function Table Pointers

When the ObjectStore client reads a page from the Server into your application, it must store correct virtual function table pointers to those objects that have virtual function tables (vfts). To do this, the client must have the addresses of the vfts for the classes in your schema.

ObjectStore derives these addresses as part of schema generation. The application schema source file generated by **ossg** contains **extern** declarations of the vft symbols. The schema generator stores their addresses in a table when your program starts execution.

Normally, vfts are declared **extern** in all modules that reference them except in the module that defines the first noninline virtual function. That module defines the vft as a global symbol.

However, there are several cases in which there is no module that corresponds to the first noninline virtual function. For example, all the class's virtual functions can be inline. In these cases, C Set ++ generates a static vft in each module that allocates an object of the class. In such a case, the application schema source file cannot link to the vft, since it is static in some other module or modules.

You can tell if you have such classes with the **-qinfo:vft** argument to C Set ++. If it informs you of any classes with static links, and if you want to store them persistently, you must either add an out-of-line virtual function to each such class, or use the command-line arguments that control vft allocation to make the vfts

accessible. Aside from making ObjectStore work, it makes your executable smaller. Here is an example of `-qinfo:vft` output:

```
class A {
public:
    virtual foo();
    int a;
    A() {a=0;}
};

class B : public virtual A {
public:
    virtual foo();
    int b;
    B() {b=1;}
};

class C : public B {
    int c;
    C() {c=2;}
};

int B::foo() {}
```

```
x1C_r -qinfo=vft -c x.C
```

```
"x.C", line 29.1: 1540-017: (W) Return value of type "int" is expected.
```

```
"x.C", line 1.1: 1540-281: (I) The virtual function table for "A" will be defined where "A::foo()
```

```
"x.C", line 10.1: 1540-280: (I) The virtual function table for "B" is defined with "extern" links.
```

```
"x.C", line 20.1: 1540-280: (I) The virtual function table for "C" is defined with "extern" links.
```

The two C Set ++ arguments are `-qvftable` and `-qnovftable`.

`-qvftable` instructs C Set ++ to allocate global vfts for all classes visible in the compilation unit. `-qnovftable` forces all vfts to be referenced externally.

The simplest way to make C Set ++ work with ObjectStore is to create an additional source file that includes the definitions for classes that have static links. Compile it with `-qvftable`, and link it. This does not avoid the extra static copies of the table in any modules that allocate objects of this class, but it does allow the application schema source file to link to them.

To avoid extra copies, you have to compile all ordinary sources with `-qnovftable`, and then add `-qvftable` to the appropriate sources so that each vft is defined exactly one time.

Using **new** and **delete** Operators with cfront

A design limitation of the C++ language fails to match overloadability of the **new** operator with a corresponding overloadability of the **delete** operator. Therefore, to make deletion of persistent objects work transparently, ObjectStore must take control of some of the internals of the C++ storage allocation run-time environment, including `__vec_delete`.

Note that there is still a limitation that ObjectStore cannot completely address — if a shared library that is not itself linked with the ObjectStore library uses C++ allocation, it will not share the copy of `_new_handler` provided by ObjectStore.

Debugging Applications

In addition to native debuggers, alternatives for debugging applications include

- Set the **OS_TRACE_MISSING_VTBLS** environment variable. If you run the application with **OS_TRACE_MISSING_VTBLS** set, ObjectStore catalogs vtbls that
 - Were not found at initialization
 - Might result in **err_missing_vtbl** errors later on

See *ObjectStore Management*, [Chapter 3, OS_TRACE_MISSING_VTBLS](#).

- Run the **osverifydb** utility after an application creates a database. This allows you to check for invalid pointers.
- If the database information is incorrect, you can examine it with the ObjectStore Inspector.
- If performance is the issue, you can use the ObjectStore Performance Expert (OPE) as a debugging tool.

If you use schema protection, you can remove all symbol names from your application so that use of a debugger does not display the names of functions on the stack. This makes it harder for someone to subvert schema protection by analyzing information provided by the debugger.

Dependency of Object Files on Header Files

You should maintain complete dependencies of object files on header files. Object Design recommends automating this by using a dependency generation tool in your makefiles.

In previous releases on some platforms, ObjectStore provided the **osmakedep** command. This is no longer the case. Your compiler vendor should supply you with a configuration management tool.

Retrofitting Makefiles

When you use a makefile to build an ObjectStore application, if you start with a makefile that you use for another application, be sure to follow the instructions in this book for specifying libraries. ObjectStore brings in certain system libraries that are not explicitly specified.

If you copy an on-line version of an ObjectStore makefile from ObjectStore documentation sources, make sure that you have tabs at the beginning of appropriate lines and not spaces.

UNIX

This section provides information for compiling and linking ObjectStore applications on UNIX platforms. Unless a particular UNIX platform is named in the heading, the material refers to all UNIX platforms that support ObjectStore C++ interface Release 5.1.

Linking with ObjectStore Libraries

ObjectStore includes libraries that you must link with when you build your application. Libraries allow multiple programs to share code without redundantly compiling the source. Applications use libraries by specifying them at link time.

Requirement

You always link with the **libos** library. If you are using full ObjectStore, you use **\$OS_ROOTDIR/lib/libos**; if you are using ObjectStore/Single, you use **\$OS_ROOTDIR/libsingle/libos**.

You must also link with either the **libosthr** or **libosths** libraries. You must link with additional ObjectStore libraries according to the features you use in your code, as shown in the following table. If there is more than one library, you must specify them in the order given. See Examples of Passing Libraries to the Linker on page 90.

<i>If Your Application Uses This Feature</i>	<i>Link with These Libraries in This Order</i>
Any ObjectStore feature	libos {libosthr or libosths}
Collections	liboscol libos {libosthr or libosths}
Compactor	liboscmp libos {libosthr or libosths}
Database utilities:	
• os_dbutil::osverifydb()	liboscol libosmop libostcl libosdbu libos {libosthr or libosths}
• All other os_dbutil methods	libosdbu libos {libosthr or libosths}
MOP	libosmop liboscol libos {libosthr or libosths}
Queries and indexes	libosqry liboscol libos {libosthr or libosths}
Relationships	liboscol libos {libosthr or libosths}
Schema evolution	libosse libosqry liboscol libos {libosthr or libosths}
C run-time library	libos {libosthr or libosths}

Threads library

The **libosthr** or **libosths** library is for threads. Be sure to use the correct threads library for your platform:

Platform	Threads Library
AIX	libosthr
Digital UNIX	libosthr
HP-UX	libosthr or libosths
IRIX	libosths
Solaris 2	libosthr or libosths

If you have a real threads package, link with the **libosthr** library. The **libosths** library is a stub threads library.

IRIX shared process (**sproc**) restrictions

Use **pthreads** but not **sprocs** in ObjectStore Release 5 and later. ObjectStore 5.0 for IRIX does not support concurrent ObjectStore operations (for example, transactions) in more than one process in a shared process group, where the processes are sharing virtual memory. If you want your program to use ObjectStore 5.0, it is best to restrict all ObjectStore operations to a single process in the shared process group. If you want ObjectStore operations in a number of the processes, the program must minimally ensure that

- All file descriptors are shared in the group.
- All accesses to ObjectStore using external interfaces, and potential page faults, are single-threaded through a single process at any given time.

C++ run-time library

You must link **libos** before the C++ run-time library (often called **libC**). On some platforms it is possible for the order to come out wrong when you link with shared libraries that use shared libraries. Linking with **libos** first prevents this problem.

Specifying libraries

Use the **-l** (lowercase l as in *link*) option to pass library names to the linker. When you specify an ObjectStore library, do not include the **lib** portion of the library name. For example:

```
CC -L$(OS_ROOTDIR)/lib -o my_exec main.o os_schema.o foo.o -los
```

Note that the **-L\$(OS_ROOTDIR)/lib** option begins with an uppercase L as in *Library*.

Exclusive libraries

You cannot link with both **libosse** and **liboscmp**.

UNIX

HP-UX and DCE

This release of ObjectStore works with HP-UX 10. DCE, bundled with HP-UX 10, provides thread support on HP-UX. If you use threads and link with DCE, you must use **libosthr** instead of **libosths**.

When using threads (**libosthr**), while ObjectStore is running in its fault handler, DCE masks most signals. Therefore, when DCE is linked into an ObjectStore application, ObjectStore's fault handler reenables the following signals:

- **SIGHUP**
- **SIGINT**
- **SIGQUIT**
- **SIGBUS**
- **SIGSEGV**
- **SIGVTALRM**

If an application wants to reenable a different set of signals, this default behavior can be overridden using the following interfaces:

```
extern "C" void _ODI_set_reenable_mask(unsigned int new_mask);  
extern "C" unsigned int _ODI_get_reenable_mask();
```

Consult Object Design Technical Support before using these interfaces.

The mask is a bit vector of the signal values. A value of 1 indicates that the corresponding signal is enabled when ObjectStore's fault handler is entered.

Note that **_ODI_get_reenable_mask** returns 0 if **_ODI_set_reenable_mask** has not been called.

Also note that regardless of the **reenable_mask** value, ObjectStore always reenables the signal being handled (**SIGSEGV** or **SIGBUS**).

Examples of Passing Libraries to the Linker

When building an application that uses compaction, MOP, queries, and collections, specify ObjectStore libraries like this:

```
-loscmp -losmop -losqry -loscol -los {-losthr | losths}
```

When building an application that uses the schema evolution feature, specify libraries like this:

```
-losse -losqry -loscol -los {-losthr | losths}
```

When building an application that uses database utilities, specify the libraries like this:

-losdbu -los {-losthr | losths}

If you need all ObjectStore libraries and you need a stub threads library because you do not have real threads, link with the libraries in this order:

{-losse | -loscmp} -losmop -losqry -loscol -los -losdbu -losu -losths

If you need all ObjectStore libraries and you have a real threads package, link with the libraries in this order:

{-losse | -loscmp} -losmop -losqry -loscol -los -losdbu -losu -losthr

DEC C++ 64-Bit Pointer Considerations

Since the Digital UNIX operating system environment supports 64-bit pointers, using this compiler with ObjectStore presents unique decisions for application writers. Fortunately, ObjectStore Release 5.1 and the DEC C++ 5.1 compiler include options that offer alternatives depending on your particular application needs.

Usually, DEC C++ uses 64 bits for all pointers in generated code. The eXtended Truncated Address Support Option (**xtaso**), and other **#pragma** directives and compiler options, let code with 32-bit pointers coexist within this 64-bit operating system environment. If you use only **-xtaso**, and have 64-bit pointers in the objects stored in your database, then your database will not be heterogeneous.

Note that within a compilation unit, **pragma** statements allow the user to switch between 32-bit and 64-bit classes selectively. However, **ossg** deals with these **pragma** statements at the granularity of a complete top-level class.

Software version requirements

You must be running Digital UNIX version 4.0A and the DEC C++ compiler version 5.5-005 general release to take advantage of the ObjectStore 5.0 and DEC 5.5 compiler features described here. You should also install the DEC patch OSF360-350222. This patch includes the previously required 350108 patch.

If you are uncertain what versions of software are running on your system, check them with the following commands.

To determine the operating system version, enter

uname -a

The response should be of the form

```
V3.2 148
```

where 148 corresponds to 3.2.C.

To determine the version of the compiler, enter

```
cxx -V | tail -1
```

The result should be of the form

```
DEC C++ V5.1-1
```

for DEC OSF/1 (Alpha) or greater.

How to use the **-xtaso** option

Use of **-xtaso** options, pragma statements, and the **-taso** linker option allows application programs that make intensive use of memory to make efficient use of operating system resources. It allows for the manipulation of objects containing 32-bit pointers. This also enables the use of libraries, such as the ObjectStore libraries, that were built using 32-bit pointers. ObjectStore libraries were built in this manner to support heterogeneous access to databases between this platform and those that do not support 64-bit pointers. In order to use both 64-bit and 32-bit pointers compatibly, it is necessary to limit the application to a 31-bit virtual address space. It is important to select one of these **-taso** options when using ObjectStore even if you want 64-bit pointers to be the default, because you need to ensure that your ObjectStore application uses 31-bit virtual address space. If you do not specify the **-taso** option, this is not the case.

This means that when you write ObjectStore applications, you need to use some of the methods described in this section to accommodate 32- and 64-bit pointers. Specifically, the considerations you must address are

- Header files
- System-defined function declarations with pointers in the signature
- ObjectStore collections class libraries

The sections that follow describe techniques and methods you should use to avoid compilation errors that might result from mixing 32- and 64-bit pointers.

Compiler options and pointer size directives

To use 32-bit pointers, you can use command-line compiler options and/or **#pragma** preprocessor directives. For example, ObjectStore's collections are implemented with 32-bit pointers to support heterogeneous access. Therefore, you need to use these options to ensure correct conversion between 64- and 32-bit pointers.

The pointer size compiler options are **-xtaso**, **-xtaso_short**, **-vptr_size**, and **-vptr_size_short**. You can accommodate 32- and 64-bit code using the **-xtaso** and **-vptr_size** or **-xtaso_short** and **-vptr_size_short** compiler options. The following table describes the compiler options:

<i>Compiler Option</i>	<i>Description</i>
-xtaso	Sets the default pointer size of the compilation unit to 64 bits (for all pointers except virtual function and virtual base pointers in a C++ class object). This is the normal default unless overridden by -xtaso_short .
-xtaso_short	Sets the default pointer size of the compilation unit to 32 bits (for all pointers except virtual function and virtual base pointers in a C++ class object).
-vptr_size	Makes 64 bits the default size of virtual function and virtual base pointers in a C++ class object. This is the default unless overridden by -vptr_size_short .
-vptr_size_short	Makes 32 bits the default size of virtual function and virtual base pointers in a C++ class object.

Whenever any of these options is specified on the command line, the following actions occur:

- **#pragma pointer_size** is enabled. This pragma statement only has an effect if you specify a pointer size compiler option on the command line.
- The **cxx** command automatically passes the **-taso** option to the linker. This option causes the linker to load the executable in the lower 31-bit addressable virtual address range. This means the program is limited to a 31-bit virtual address space whenever you use any of the pointer-size compiler options.

When using **#pragma** directives to control pointer size, and these compiler options, you must ensure that the pointer size of any particular pointer is used consistently across compilation units.

This is especially true when you call functions in any library compiled with different pointer sizes. The **#pragma** directives are defined below.

To use ObjectStore with the DEC C++ compiler, you must select one of the following methods of protecting pointer size assumptions:

- Use the **-xtaso** option and wrap all pointer declarations that refer to other pointers so that the referring pointer size and the referred-to pointer size agree.
In **foo****, the **foo*** that is pointed to must agree with the **foo**** declaration.
- Use ObjectStore typedefs to allow the code to compile as required independently of the **-xtaso** option. Do not use these with a persistent class.
- Use **-xtaso_short** and force everything to 32-bit pointers.

Considerations in deciding which of these approaches is the most suitable for your application are the relative importance of heterogeneity and whether disk space is an issue. (64-bit executables and libraries use considerably more disk space than 32-bit executables and libraries.)

If heterogeneity is not a consideration for your application and you are not concerned with the amount of memory used for 64-bit pointers, you can take advantage of the 64-bit operating system environment and compile using the **-xtaso** option. This option makes 64 bits the default pointer size and uses 32-bit pointers for particular declarations. In such a case, it is possible to use third-party class libraries that only support 64-bit pointers. See “Using the **-xtaso** option” on page 96 for details about using this option.

If heterogeneity is an important consideration for your application, you can easily take advantage of ObjectStore’s 32-bit pointers in this 64-bit operating system environment by compiling with the **-xtaso_short** option. This option makes 32 bits the default pointer size and uses 64-bit pointers for particular declarations. The 32-bit pointer data type allows you to minimize the amount of memory used by dynamically allocated pointers, and assists in porting applications that contain assumptions about pointer sizes. See “Using the **-xtaso** option” on page 96 for details about this option.

#pragma pointer_size directive

The **#pragma pointer_size** directive controls pointer size allocation for the following:

- References
- Pointers to member declarations
- Implied **this** pointer argument declarations in member function declarations
- Function declarations
- Array declarations

oss reminder

Remember that **oss** deals with these pragma statements at the granularity of a complete top-level class. This consideration also applies to embedded members and inherited base classes.

For this pragma statement to have any effect, specify **-xtaso**, **-xtaso_short**, **-vptr_size**, or **-vptr_size_short** on the **cxx** command.

#Pragma Directive	Description
pointer_size	Controls the pointer size of all pointers except virtual function and virtual base pointers in a C++ class object. Has an effect only if you specify one or more of the pointer-size compiler options.
required_pointer_size	Has the same effect as #pragma pointer_size but is always enabled, whether or not you specify any pointer-size compiler options.
required_vptr_size	Controls the size of virtual function and virtual base pointers in a C++ class object. Always enabled, whether or not you specify any pointer size compiler options.

This pragma statement has the following syntax:

```
#pragma pointer_size {long|64}
#pragma pointer_size {short|32}
#pragma pointer_size restore
#pragma pointer_size save
```

The **#pragma** syntax options are defined in the following table.

<i>Option</i>	<i>Description</i>
long or 64	Sets as 64 bits all pointer sizes in declarations that follow this directive until the compiler encounters another #pragma pointer_size directive.
short or 32	Sets as 32 bits all pointer sizes in declarations that follow this directive until the compiler encounters another #pragma pointer_size directive.
restore	Restores the saved pointer size.
save	Saves the current pointer size onto a pushdown stack.

The **save** and **restore** options are particularly useful for specifying mixed pointer support and for protecting header files that interface to older objects. Objects compiled with multiple pointer size pragma statements are not compatible with old object files, and the compiler cannot detect that incompatible objects are being combined.

Using the **-xtaso** option

The **-xtaso** compiler option is useful when heterogeneity is not a consideration. Using **-xtaso** enables you to use the 32-bit pointer size only for selected pointers. The **-vptr_size** option makes 64 bits the default size of virtual function and virtual base pointers in a C++ class object (64 bits is the normal default). This **-xtaso** option also enables **#pragma pointer_size** and passes **-taso** to the linker.

With this approach, most pointers in your application are 64 bits, and 32 bits are used for selected pointers. To use this method, compile with **-xtaso** to enable **#pragma pointer_size** and cause the **cxx** command to pass **-taso** to the linker. In addition, use **#pragma pointer_size**, **#pragma required_pointer_size**, and **#pragma required_vptr_size** to control the pointer sizes for particular declarations. For example, to save space in an object, declare a class as follows:

```
#pragma pointer_size save
#pragma required_vptr_size save
#pragma pointer_size short
#pragma required_vptr_size long
class Table_Node {
    char *table; // 32 bit pointers
    Table_Node *next;
    Table_Node *(Table_Node::*search)(char *);
    // pointer to member has
    // 2 32 bit fields
public:
```

```

    void insert_node(char *);
    Table_Node *extract_node(char *);
    Table_Node *search_forward(char *);
    Table_Node *search_backward(char *);
};
#pragma pointer_size restore
#pragma required_vptr_size restore

```

When you use this approach, it is important to specify the pointer size `#pragma` directives *after* any `#include` directives so the header files that assume 64-bit pointer sizes are not affected.

Using typedefs

In order to provide an interface to an application using 64-bit pointers and the ObjectStore library, which uses 32-bit pointers, a number of new typedefs are available.

Use typedefs when building an application using the `-xtaso` option to avoid compilation errors that result from incompatible interface definitions. The following code example produces such a compilation error:

```

#include <ostore/ostore.hh>
int main(int, char **)
{
    os_int32 max_servers;
    os_server **servers;
    os_int32 n_servers;

    objectstore::get_all_servers(max_servers, servers, n_servers);
}
$ cxx example.cc -c -xtaso

```

The error message that results might look as follows:

```

example.cc:9: error: In this statement, the referenced type of the pointer
value "servers" is "long pointer to os_server", which is not compatible with
"short pointer to os_server".
Compilation terminated with errors.

```

The use of the typedefs enables the compiler to perform the correct pointer conversions to interface to the ObjectStore libraries. More precisely, the typedefs listed on the next page solve the situation of a compound pointer, that is, a *pointer-to-a-pointer*, which cannot be automatically converted by the compiler. The compiler automatically converts the *pointer-to* but does not convert what it points to, the *a-pointer* part.

An example of such a typedef is `os_server_p`. It is defined as an `os_server*`.

If you modify the example, replacing `os_server **servers` with `os_server_p *servers`, the program compiles cleanly:

```
#include <ostore/ostore.hh>
int main(int, char **)
{
    os_int32 max_servers;
    os_server_p *servers;
    os_int32 n_servers;
    objectstore::get_all_servers(max_servers, servers, n_servers);
}

$ cxx example.cc -c -xtaso
```

Note that this affects C++ references as well as pointers.

The comprehensive list of typedefs supplied by the ObjectStore header files follows.

```
typedef char const* os_char_const_p;
typedef char* os_char_p;
typedef objectstore_exception& objectstore_exception_r;
typedef os_bound_query const& os_bound_query_const_r;
typedef os_canonical_ptom& os_canonical_ptom_r;
typedef os_coll_query& os_coll_query_r;
typedef os_coll_rep_descriptor const& os_coll_rep_descriptor_const_r;
typedef os_coll_rep_descriptor const* os_coll_rep_descriptor_const_p;
typedef os_collection const& os_collection_const_r;
typedef os_collection* os_collection_p;
typedef os_cursor const& os_cursor_const_r;
typedef os_database_root& os_database_root_r;
typedef os_int32& os_int32_r;
typedef os_old_reference& os_old_reference_r;
typedef os_old_reference_version& os_old_reference_version_r;
typedef os_rawfs_entry* os_rawfs_entry_p;
typedef os_reference& os_reference_r;
typedef os_reference_local& os_reference_local_r;
typedef os_reference_version& os_reference_version_r;
typedef os_transaction const* os_transaction_const_p;
typedef os_typed_pointer_void const& os_typed_pointer_void_const_r;
typedef tix_exception* tix_exception_p;
```

```
typedef void const* os_void_const_p;
typedef void const*& os_void_const_p_r;
typedef void* os_void_p;
typedef void*& os_void_p_r;
```

Using the **-xtaso_short** option

The **-xtaso_short** option makes 32 bits the default pointer size. In this mode you need to use some mechanism to accommodate code that expects 64-bit pointers. If you select this method, remember that references to other libraries can be more complicated since any library with compound pointers requires you to use **#pragmas**.

To use this option, compile with the **-xtaso_short** and **-vptr_size_short** options on the **cxx** command. The **-vptr_size_short** option makes 32 bits the default size of virtual function and virtual base pointers in a C++ class object. The **-vptr_size_short** option also enables **#pragma pointer_size** and passes **-taso** to the linker.

It is important to protect header files so that when the header file is included in a compilation (using **#include**), the pointer size assumptions are the same as those made when the code associated with the header files was compiled.

#pragmas and function declarations

In the default C++ installation, none of the system header files, including those for the standard C library, is protected, but see [“Using the header file protection option”](#) for an automated method of providing this protection. The alternative methods for doing this are to

- Modify each header file
- Use the header file protection option provided with the DEC C++ compiler

If you include a system-defined function declaration with pointers in its signature — without including it from a protected header file — you need to protect it with a **#pragma pointer_size** directive.

```
#pragma pointer_size save
#pragma pointer_size long
extern "C" {
int getrusage (
    int who,
    struct rusage *r_usage );
}
```

Modifying each header file

#pragma pointer_size restore

To modify each header file, use the **#pragma** environment directive as shown:

```
#pragma __environment save          // Save pointer size
#pragma __environment header_defaults // set to system defaults

// existing header file
#pragma__environment restore // Restore pointer size
```

Using the header file protection option

With the header file protection option, you can place special header files in a directory. DEC C++ processes these special header files before and after each file included with the **#include** directive. These special header files are named

- **__DECC_include_prologue.h**
- **__DECC_include_epilogue.h**

The compiler checks for files with these special names when processing **#include** directives. If the special prologue file exists in the same directory as a file with the **#include** directive, the contents of the prologue file are processed just before the file included with the **#include** directive. Similarly, if the epilogue file exists in the same directory as the file included with the **#include** directive, it is processed just after that file.

For example, if the source code has **#include <stdio.h>**, the order of file processing would be

- 1 **/usr/include/__DECC_include_prologue.h**
- 2 **/usr/include/stdio.h**
- 3 **/usr/include/__DECC_include_epilogue.h**
- 4 Whatever follows **#include <stdio.h>**

For convenience, you can protect header files using the script

```
/usr/lib/cmplrs/cxx/protect_system_headers.sh
```

This script creates, in all directories in a directory tree that contain header files, symbolic links to special header prologue and epilogue files.

The default directory tree root assumed by the script is **/usr/include**, but you can specify other roots.

Embedded
ObjectStore members

All ObjectStore libraries use 32-bit pointers to support heterogeneous access, and therefore must be used in 32-bit mode.

This means that the ObjectStore headers are protected so that 32 bits is the default pointer size. Any time an ObjectStore-defined object is embedded in an application object, that class — or at least the collection declaration — must be protected as well. This occurs most commonly in the collection class library, but it also applies to other ObjectStore classes such as `os_Reference`. Also note that any ObjectStore relationship is an embedded collection.

Troubleshooting Errors

When you are working in a mixed 32- and 64-bit environment, there are a variety of conditions that can cause errors. Several error message samples and the conditions that caused them are described below. You might see such errors if you have combined `-xtaso` and `-xtaso_short` incorrectly, or failed to specify the `-taso` option to the linker.

- In an example such as `$OS_ROOTDIR/examples/coll`, if you change the `TFLAGS` to be only `-xtaso`, and your `OSSHEMA_FLAGS` remain `-xtaso_short`, your code compiles without any warning messages. However, an attempt to run the code produces an error such as

No handler for exception:

Miscellaneous ObjectStore error

An inconsistency was detected during the allocation of an object of type part. The compiler believes the allocation size should be 40 bytes, while the schema believes it should be 32 bytes. Verify that the arguments to 'new' match the type and/or array count of the object being allocated.

(err_misc)

Abort process (core dumped)

- If you try to use `oss` without any `-taso` flags, the following error occurs:

```
oss -assf osschema.cc -asdb coll.adb -I/usr/local/ostore/4.0.0.C/  
include/schema.cc /usr/local/ostore/4.0.0.C/lib/liboscol.ldb  
<err-0004-0008>
```

The following class definitions in library schema `"/usr/local/ostore/4.0.0.C/lib/liboscol.ldb"`

were inconsistent with the corresponding definitions obtained from other library or compilation schemas specified for building the application schema:

`"os_set"` from the schema source file `"/usr/local/ostore/4.0.0.C/include/ostore/coll/coll_int.hh"`

the class size changed from 24 to 40

UNIX

```
"os_tinyarray" from the schema source file "/usr/local/ostore/4.0.0.C/include/ostore/coll/query.hh"...  
*** Exit 1  
Stop.
```

HP Requires Linker Options

You must supply **-WI** (this is a lowercase l), **-E** to **cc** or **CC**. This option is necessary so that symbols are exported correctly. If this is not done, virtual function table pointers (vtbls) are not available when ObjectStore relocation occurs. For this reason, your application might run correctly the first time, but signal an exception when run the second time. The exception would be

```
No handler for exception:  
Attempt to call virtual function without vtbl.  
Vtbl for type os_packed_list not linked into application. (err_missing_vtbl)  
IOT trap (core dumped)
```

Here is an example of an HP makefile link command line:

```
$(CCC) +eh -WI,-E -g -o test test.o os_schema.o $(LDLIBS)
```

+eh Mode Supported

ObjectStore only supports **+eh** mode of HP CC in Release 5.1.

HP aC++ Source Files

HP recommends that you name HP aC++ source files with an extension of either **.c** or **.C**, possibly followed by additional characters. This applies to schema source files as well because they are preprocessed for **oss** using **aCC**.

Note that HP recommends that you use extensions consisting only of **.c** or **.C** without additional characters, because while the compiler accepts the additional characters, other HP tools and environments might not.

If you compile only, each C++ source file produces an object file whose prefix is identical to the source file but with a **.o** suffix. However, if you compile and link a single source file into an executable program in one step, the **.o** file is automatically deleted.

HP C++ Compiler Messages

When using the HP C++ compiler, you might receive the messages below when building your application. Although the application seems to run correctly, removing the **-g** option from the compilation and link phases resolves the errors. You might see these messages during the compilation phase:

```
CC: "myprog.C", line 82: warning: debug.object_ids: weird
vtable->vclass for os_Collection <const MyObject*> (198)
CC: "myprog.C", line 8: warning: debug.emit_variable: bad address found
for name ABC::xyz::ex (251)
```

Also, you might see these warnings during the link phase:

```
pxdb: [cu: 0 index: 0x136] can't link template/expansion
pxdb: [cu: 0 index: 0x14f] can't link template/expansion
pxdb: [cu: 0 index: 0x168] can't link template/expansion
pxdb: [cu: 0 index: 0x181] can't link template/expansion
```

You might see the following warnings during compilation of a source file that uses stack transactions (that is, that contains **OS_BEGIN_TXN/OS_END_TXN** macros):

```
CC: "myprog.cc", line 342: warning: label in block with destructors (2048)
```

Such warnings, where they concern the stack transaction macros, can safely be ignored.

SGI IRIX Compiler Option

On SGI IRIX 6.2 you must use the **-n32** option to the compiler. The **-32** and **-64** options are not currently supported.

Sun C++ Compiler Options

Sun C++ 4.0 has a compile-time option, **-pto**, that creates all template instantiations in the current object file. Do not use this option when developing ObjectStore applications, because it makes everything (including vtbls) static. Since the vtbls are static, ObjectStore cannot get at them and gets the wrong vtbl, which leads to an error.

-vdelx compiler option

When you are using SPARC ProCompiler C++, you must always specify the **-vdelx** compiler option. The **-vdelx** option to **CC** generates the correct calling sequence for persistent vector deletes. For example:

```
delete [] persistent_array;
```

Without `-vdelx`, the compiler generates a direct call to the Sun vector delete routine. This routine returns an error message indicating that it did not allocate the array:

```
error: delete [] does not correspond to any `new`
```

If you use **CC** to link (by means of **ld**), then the linker receives the correct libraries. Be sure to study the **CC** and **ld** man pages for details, especially if you invoke **ld** directly.

On Solaris 2.x systems, you must compile (and link) ObjectStore applications with real thread support. Specify `-losthr` on the link line to supply the proper library for real thread support. The **libosths** library, a stub threads library, can also be used with Solaris 2.x systems. Using this library can result in higher performance in some circumstances because it turns off thread locking.

Debugging with DBX

To use the multithread-related (MT) commands of DBX to debug a multithreaded application, you need to obtain a separate license from Sun. However, you are not required to use a special version of DBX. MT features are part of the standard DBX. You can debug without the multithreaded debugging commands. The DBX debugger reports an error if you do not have threading licenses, but nonetheless it debugs single-threaded applications.

Solaris 2 Linking

ObjectStore supports linking with and without threads. This means there are two ObjectStore thread support libraries on Solaris 2 — **libosths** and **libosthr**.

- Use the **libosthr** library in linking an application that links with `-mt` (or `-lthread`).
- Use the **libosths** library in linking an application that does not link with `-mt` (or `-lthread`).

Here is an example of the same program (**foo**) compiled and linked in both configurations.

On Solaris 2.x, you must always specify the `-mt` compiler option on the **CC** command line. This is required for a successful compilation.

Linking with threads

With threads:

```
CC -mt -I$(OS_ROOTDIR)/include $(CCFLAGS) -o foo  
-L$(OS_ROOTDIR)/lib -los -losthr
```

Note that to link an application to use threads,

- Use **-mt** (or **-lthread**) on the link line.
- Link with **libosthr**.

Without threads:

```
CC -I$(OS_ROOTDIR)/include $(CCFLAGS) -o foo  
-L$(OS_ROOTDIR)/lib -los -losths
```

Note that to link an application that does not use threads,

- Do not use **-mt** on the link line.
- Do not explicitly place **-lthread** in the link line.
- Link with **libosths**.

Linking without
threads

Sample Makefile Template

The makefile on the next page is a template for building ObjectStore applications. This makefile is for an application that uses queries and collections.

In an ObjectStore makefile, in the **LDLIBS** line, you must specify each library with which you are linking.

Then, in the line of the makefile where **ossg** generates the application schema, you must specify the library schemas needed by your application schema. For each library schema that you specify in the **ossg** command line, you must specify the corresponding library in the **LDLIBS** line.

Note that the reverse is not true. For each library that you specify in the **LDLIBS** line, you do not necessarily specify a library schema in the **ossg** command line. This is because every library does not necessarily have a library schema. Only those libraries that store or retrieve persistent data have associated library schemas.

OS_POSTLINK and
os_postlink

OS_POSTLINK is a macro provided with ObjectStore. It calls the **os_postlink** command that fixes vtbls and discriminants in the executable, if needed. While **os_postlink** does not actually do anything on some platforms, Object Design recommends that you always include it so that its absence does not cause a problem if you move the application to another platform. For more

UNIX

information about `os_postlink`, see Working with Virtual Function Table (VTBL) Pointers and Discriminant Functions on page 77.

Tabs and spaces

If you are using an on-line version of this book, and you copy a makefile and try to use it, make sure that there are tabs and not spaces at the beginning of relevant lines.

Application schema database

In makefiles, you should not specify an existing ObjectStore database as the application schema database. Doing so can corrupt your build process if the Server log has not been propagated to the database.

Makefile template

```
include $(OS_ROOTDIR)/etc/ostore.lib.mk
APPLICATION_SCHEMA_PATH=app-schema-db
LDLIBS = $(OS_EXPORT) -losqry -loscol -los -losths [other libraries]
SOURCES = .cc files
OBJECTS = .o files
EXECUTABLES = executables
CCC=CC

all: ${EXECUTABLES}

executable: $(OBJECTS) os_schema.o
    $(CCC) -o executable $(OBJECTS) os_schema.o \
        $(LDLIBS)
    $(OS_POSTLINK) executable

.o files: .cc files
    ${CCC} ${CPPFLAGS} -c .cc files

os_schema.o: os_schema.cc
    $(CCC) ${CPPFLAGS} -c os_schema.cc

os_schema.cc: schema.cc
    ossg -assf os_schema.cc -asdb $(APPLICATION_SCHEMA_
PATH) \
    ${CPPFLAGS} schema.cc $(OS_ROOTDIR)/lib/libosqry.ldb \
    $(OS_ROOTDIR)/lib/liboscol.ldb

clean:
    osrm -f ${APPLICATION_SCHEMA_PATH}
    rm -f ${EXECUTABLES} ${OBJECTS} os_schema.*
```

Using Signal Handlers

At run time, ObjectStore sets a handler for the UNIX **SIGSEGV** signal. On some platforms, it also sets a handler for **SIGBUS**.

These handlers are critical to ObjectStore's operation. If your application disturbs them, it will fail, and it might fail in a way that makes it difficult to determine why it failed.

If you must temporarily change the state of the handler for **SIGSEGV** and **SIGBUS**, be sure to save and restore the complete state. You cannot do this with the **signal** entry point; you must call **sigaction** and save the contents of the structure returned as the old handler state.

Makefile for Building from Compilation Schemas

The following makefile fragment shows a less common use of **ossg**. **ossg** builds a compilation schema in two steps from two schema source files. It then builds the application schema source file and the application schema database from the compilation schema. Finally, the makefile compiles the application schema source file and links it into the executable. Note that the double colon allows you to define the same target twice.

```
all: my_exec

my_exec: main.o os_schema.o foo.o bar.o
    CC -o my_exec main.o os_schema.o foo.o bar.o -los
    $(OS_POSTLINK) my_exec

my_exec.cdb:: schema_source1.o
    ossg -csdb my_exec.cdb $(CPPFLAGS) schema_source1.cc
    touch schema_source1.o

my_exec.cdb:: schema_source2.o
    ossg -csdb my_exec.cdb $(CPPFLAGS) schema_source2.cc
    touch schema_source2.o

os_schema.cc: my_exec.cdb
    ossg -asdb my_exec.adb -assf os_schema.cc my_exec.cdb
```

For more information, see *Generating a Compilation Schema* on page 55.

Establishing Fault Handlers in POSIX Thread Environments

On some UNIX systems, the POSIX thread environment gives each thread its own set of UNIX signal handlers. On such systems, the ObjectStore handlers for **SIGSEGV** (and in some cases **SIGBUS**) must be established in each thread.

ObjectStore provides two macros to help you do this:

- **OS_ESTABLISH_FAULT_HANDLER** establishes the start of the fault handler block.
- **OS_END_FAULT_HANDLER** ends the fault handler block.

UNIX

Affected platforms

On the Digital UNIX, SGI IRIX, and HP-UX platforms, use the **OS_ESTABLISH_FAULT_HANDLER** and **OS_END_FAULT_HANDLER** macros at the beginning and end of any thread that performs ObjectStore operations. This is required because HP-UX and Digital UNIX signal handlers are installed strictly on a per-thread basis and are not inherited across **pthread_create** calls.

A typical function that uses these macros in an ObjectStore application would look like this:

```
void thread_1() {  
    OS_ESTABLISH_FAULT_HANDLER  
    ...your code...  
    OS_END_FAULT_HANDLER  
    return value;  
}
```

Unaffected platforms

The AIX and Solaris 2 platforms do not require the use of these macros because the signal handlers are inherited across **pthread_create** calls.

You can benefit from using these macros, even on platforms not requiring them. This practice helps ensure portability of code, and also guards against potential problems resulting from future changes to your operating system.

Virtual Function Table Pointers

Virtual function table and discriminant function symbols do not need to be in the base executable. In fact, the schema object file does not have to be in the base executable. It can be in shared libraries.

When ObjectStore is fully initialized, it examines each of the shared libraries specified at link time that was opened by **shl_load** or **cxxshl_load**. ObjectStore searches the shared libraries for the symbols that identify the tables in the schema object file. After it finds these symbols, it searches again to find as many vtbls and discriminants as it can.

Debugging Applications

When debugging applications on UNIX systems, be sure to instruct the debugger to send **SIGBUS** and **SIGSEGV** signals through to the application. ObjectStore expects to be handed those

exceptions, as opposed to having the debugger catch them as if they were errors.

SGI IRIX	On IRIX, the standard debugger is odbx (old dbx) or cvd , which is part of CASEVision/Workshop 2.4 (a layered product).
HP-UX	On HP-UX, the standard debugger is xdb . Specify z 10 irs z 11 irs
AIX	On AIX, the standard debugger is dbx . Specify ignore 10 ignore 11
Digital UNIX	On Digital UNIX, the standard debugger is decladebug . Specify ignore SEGV It is not necessary to ignore SIGBUS .

Solaris C++ Search Paths

On Solaris 2 with ProCompiler C++ 4.0.1, the schema generator (**ossg**) uses a script as the default preprocessor. This script invokes **CC** and expects to find the ProCompiler C++ 4.0.1 compiler in your search path. If the script finds the wrong **CC** when called by **ossg**, you receive a message such as the following. If you correct your search path, this should fix the problem.

/usr/include/stddef.h line 30: syntax error on input" (103)

SGI Delta C++ Compiler

ObjectStore cannot persistently store objects in Delta format. You can use the Delta C++ compiler, but you cannot use ObjectStore to store Delta objects persistently in the database.

Windows

This section provides information you need to compile and link ObjectStore applications using the Microsoft Visual C++ 32-bit edition compiler.

Note that you must use the 32-bit edition of Visual C++.

Linking with ObjectStore Libraries

ObjectStore includes libraries that you must link with when you build your application. Libraries allow multiple programs to share code without redundantly compiling the source. Applications use libraries by specifying them at link time.

On Windows platforms, the only ObjectStore library is **ostore.lib**. You must link with this library every time you build an ObjectStore application.

Use Custom Build to Run ossg

If you want to run the ObjectStore schema generator within the IDE (Integrated Development Environment), you must use a Custom Build step. This is to add the rules needed to run the ObjectStore schema generator in its automatically generated makefiles.

How to customize
Visual C++

To use the Custom Build feature, name your schema source file with an extension other than **.cpp** or **.c**, (**.osg**, for example). Include the schema source file in your project, and set up the Custom Build step appropriately. See the **MFC** example discussed in Building ObjectStore/Microsoft Foundation Class Applications on page 117. Also see Using the Visual C++ Integrated Development Environment (IDE) on page 121.

Ensure That You Include Required Files

The ObjectStore installation program modifies your **INCLUDE** environment variable to include the **%OS_ROOTDIR%\include** directory. Therefore, under normal circumstances, you should not need to specify this directory in a compilation command.

If you edit your environment to remove **%OS_ROOTDIR%\include** from your **INCLUDE** environment variable, you must add the following argument to your compilation and **ossg** commands.

-I%OS_ROOTDIR%\include

If you include in your makefiles the makefile shipped with ObjectStore, %OS_ROOTDIR%\etc\ostore.mak, you can use the makefile macro **COMPILER_OPTS** to get the proper compilation options.

Make and Compiler Options

Here is an example of a compilation command:

```
mycode.obj: mycode.cpp
  cl -c -W3 -EHa -G4 -D_X86_=1 -DWIN32 -MD -Zi -vmg -vmv\
  I$(OS_ROOTDIR)\include mycode.cpp
```

Requirements

The options **-c**, **-W3**, **-G4**, **-D_X86_=1**, and **-DWIN32** are required in approximately this form by all compilations with Visual C++. You can adjust the warning level lower with **-W2** or higher with **-W4**, but **-W3** is the default for Visual C++. You can also optimize for the 386 with **-G3** or the Pentium with **-G5**, but **-G4** is the default for Visual C++. You can use any optimization arguments.

The option **-MD** is required in order for you to use **msvcrt.lib**, which is required by ObjectStore.

The implementation of TIX exceptions depends on C++ exceptions. All ObjectStore modules should be compiled with the **/EHa** option. ObjectStore header files that rely on C++ exception handling use a pragma statement to ensure that **/EHa** is used. If you do not specify **/EHa** when you compile files that use those headers, you receive an error such as

```
osdraw.cpp(168) : error C4530: C++ exception handler used, but unwind semantics are not enabled. Specify -EHa
```

Do not ignore this error.

The option **-Zi** causes debugging information to be put in the project database. You can also use the **-Z7** or **-Zd** option.

All compiler and linker options not mentioned explicitly can be set as you want.

Use the Standard Run-Time Library on Windows NT

All ObjectStore applications for Windows must link with the standard Visual C++ run-time library **msvcrt.lib**.

Run-time libraries

Each run-time library has its own allocation (**malloc**, **operator new**) and deallocation (**free**, **operator delete**) routines. You cannot call an allocator from one library (for example, **msvcrt**) and deallocate that object in any other library (for example, **LIBC.LIB** or **LIBCMT.LIB**).

ObjectStore is linked with **msvcrt.lib**, which allows ObjectStore to freely allocate objects in any ObjectStore DLL and deallocate them in any other ObjectStore DLL. This is because all ObjectStore DLLs share the single allocator in **MSVCRT40.DLL**.

Because ObjectStore is linked with **msvcrt.lib**, it is easy for most applications that also link with **msvcrt.lib** to deallocate objects that are allocated by ObjectStore APIs (for example, **os_collection::query()**).

Compiling DLLs

Any part of any application that links against ObjectStore must use **msvcrt.lib** and only **msvcrt.lib** as its run-time library. Consequently, you must specify the **-MD** option when you compile a DLL that calls ObjectStore. This option is not required when you compile other DLLs (your own or obtained from a third party), provided you are careful about the issues surrounding shared C run-time constructs.

You can use a library DLL that is not compiled with the **-MD** option. Keep in mind that it has a separate copy of the C run-time library and therefore you cannot share certain objects between that DLL and the rest of the application.

Compiling DLLs without **msvcrt.lib**

When you do not compile a DLL with the **-MD** option, you cannot share pointers to standard input/output files, C++ streams, and the like. Without the **-MD** option, there can be conflicts about how to delete a shared object. Some library DLLs (like **WSOCK32**, the Windows Sockets DLL) were compiled with the **-MD** option. Others, like **MFC**, were not, but you can compile them with the **-MD** option.

If you must link with **LIBC.LIB** or **LIBCMT.LIB**, you can do one of two things:

- Ask the vendor of the software that uses **LIBC.LIB** or **LIBCMT.LIB** to use **msvcrt.lib** instead. You cannot design an application as a set of cooperating DLLs if the vendor does not support **msvcrt.lib**. For example, if you want to use something

that needs to share C run-time objects but was not compiled with the **-MD** option, you need to contact that vendor.

- Encapsulate all code that uses ObjectStore in one or more DLLs. In other words, instead of directly calling ObjectStore APIs like **os_database::open()** from the executable, call your own **OpenOSDatabase()** function that is in a DLL that was linked with **msvcrt.lib**. You can then link the executable with **LIBC.LIB** or **LIBCMT.LIB**. You must ensure that any objects passed between the executable and the DLL are deallocated by the correct allocator.

Linking Your Files

After you generate an application schema, you can link the application object files, application schema object file, and ObjectStore libraries to create an executable or dynamic link library (DLL).

Required

The following requirements apply:

- You must link ObjectStore applications with the library **ostore.lib**. To do so you can either add **%OS_ROOTDIR%\lib** to your **LIB** environment variable and put **ostore.lib** on the link command line, or you can put **%OS_ROOTDIR%\lib\ostore.lib** on the link command line.

The reference to **ostore.lib** can be anywhere on the link command line as long as it appears *before* any explicit references to **msvcrt.lib** (the C run-time library). If you are using MFC, see Building ObjectStore/Microsoft Foundation Class Applications on page 117.

- You must link with **msvcrt.lib**. This is the C run-time library with which all ObjectStore DLLs and executables are linked.

Link example

Use a command line like the following, which is shown using the response file syntax used by **nmake**:

Linking an executable file would look like this:

```
link @<<
-NODEFAULTLIB -machine:i386 -subsystem:windows -debug
-debug-type:cv -out:myapp.exe myapp.obj myschema.obj ostore.lib
msvcrt.lib kernel32.lib user32.lib gdi32.lib winspool.lib
<<
```

Linking a DLL would look like this:

```
link @<<
-NODEFAULTLIB -dll -machine:i386 -debug -debugtype:cv
-out:myapp.dll myapp.obj myschema.obj ostore.lib msvcrt.lib
kernel32.lib user32.lib gdi32.lib winspool.lib
<<
```

myschema.obj in the previous two link commands is the application schema object file output from the schema generator (**ossg**).

In both examples, the **%OS_ROOTDIR%\lib** directory must be listed in your **LIB** environment variable. This is normally set up by the ObjectStore installation program.

Sample Makefile

In makefiles, do *not* specify an existing ObjectStore database as the application schema database. Doing so can corrupt your build process if the Server log has not been propagated to the database.

The schema generator (**ossg**) is a C++ compiler front end that parses C++ source code to obtain information. This means that you must pass the same compiler flags to **ossg** as you pass to the compiler when you are compiling your source. It is especially important to duplicate the **-I**, **-D**, and **-Zp** arguments. Also, remember that **-MD** implies **-D_DLL**, so pass the **-D_DLL** to **ossg**. Note how **\$(COMPILER_OPTS)** is used for both compilation and schema generation in the following makefile fragment.

```
include $(OS_ROOTDIR)\etc\ostore.mak

OBJECTS=note.obj
EXECUTABLES=note.exe

APPLICATION_SCHEMA_PATH=note.adb

all: $(EXECUTABLES)

myschema.obj: schema.cc
    ossg -asof myschema.obj -asdb $(APPLICATION_SCHEMA_
PATH) \
    $(COMPILER_OPTS) schema.cc

note.obj: note.cc
    $(COMPILER) $(COMPILER_OPTS) note.cc

note.exe: $(OBJECTS) myschema.obj
    $(CL_LINK) /OUT:note.exe $(OBJECTS) myschema.obj \
    $(OS_ROOTDIR)\lib\ostore.lib

clean:
    -osrm -f $(APPLICATION_SCHEMA_PATH)
```

```
-del $(OBJECTS) osschema.*
```

Sample Makefile for an Application That Uses Collections and Queries

The following makefile adds the specification of two library schemas to the preceding makefile. This is for a Windows application that uses collections and queries. Note that in ObjectStore Release 3, you only needed to specify the collections library schema, which included the query library schema. In ObjectStore Release 5.1, if you use both collections and queries, you must specify a library schema for each feature.

Sample file

```
include $(OS_ROOTDIR)\etc\ostore.mak

OBJECTS=note.obj
EXECUTABLES=note.exe

APPLICATION_SCHEMA_PATH=note.adb

all: $(EXECUTABLES)

myschema.obj: schema.cc
    ossg -asof myschema.obj -asdb $(APPLICATION_SCHEMA_
    PATH) \
        $(COMPILER_OPTS) schema.cc \$(OS_
    ROOTDIR)\lib\osquery.ldb 1
        $(OS_ROOTDIR)\lib\os_coll.ldb 2

note.obj: note.cc
    $(COMPILER) $(COMPILER_OPTS) note.cc

note.exe: $(OBJECTS) myschema.obj
    $(CL_LINK) /OUT:note.exe $(OBJECTS) myschema.obj \
        $(OS_ROOTDIR)\lib\ostore.lib

clean:
    -osrm -f $(APPLICATION_SCHEMA_PATH)
    -del $(OBJECTS) osschema.*
```

Notes on sample file

¹Add query library schema.

²Add collections library schema.

Specifying Environment Variables

The following environment variables are automatically set by the ObjectStore installation program:

OS_ROOTDIR	ObjectStore root directory.
PATH	%OS_ROOTDIR%\bin is added to this environment variable.

INCLUDE %OS_ROOTDIR%\include is added to this environment variable (only for development installations).

LIB %OS_ROOTDIR%\lib is added to this environment variable (only for development installations).

You can edit these environment variables using the **Control Panel System** applet.

On Windows NT, ObjectStore adds these variables to the *system* environment.

Debugging Your Application

Visual C++ debugger

ObjectStore handles all access violations to determine if they are persistent memory accesses. If you handle access violations in the Visual C++ debugger you would disrupt this, so leave this exception unhandled in the debugger. You might see multiple exception messages such as the following:

First-Chance Exception in *yourprog.exe*: 0xC0000005: Access Violation.

First-Chance Exception in *yourprog.exe*: (*something.DLL*) 0xC0000005: Access Violation.

You can safely ignore these messages.

Under **Debug | Exceptions**, the default for exception **C0000005 Access Violation** is **Stop if not handled**. Do not change this to **Stop Always**. If you do, ObjectStore cannot function normally.

Obtaining a stack trace

On Windows NT, to obtain a complete stack trace, use the debug versions of the ObjectStore DLLs.

The debug DLLs are available on the distribution CDROM.

To obtain a back trace using the debug DLLs, put the debug DLL directory in the front of the path environment and then run the **msvc** debugger.

For example, using **myapps** on drive C and a CDROM on drive E, you would issue the following commands:

```
C:\myapps> set path = e:\windows\debnt\bin;%path%
C:\myapps> start msvc myapps.exe
```

Setting a breakpoint

If your application exits with an unhandled TIX exception, you can set a breakpoint to obtain a stack trace prior to the stack's being unwound. The `OS_DEF_BREAK_ACTION` environment variable allows you to do this. When you set this variable to 1, ObjectStore reaches a hardcoded breakpoint immediately before an exception is signaled. This works with Visual C++'s just-in-time debugging.

Abnormal Application Exit

In case of abnormal ObjectStore application exits, you might want to get a stack trace at the point of failure before cleanup handlers are run. To do this, you can do one of the following:

- Set the environment variable `OS_DEF_BREAK_ACTION`. See *ObjectStore Management, Chapter 3, Environment Variables*.
- Include code in your program that uses the static method for the class `tix_exception` called `set_unhandled_exception_hook()`; see `%OS_ROOTDIR%\include\ostore\tix.hh`.

This method allows you to set a function to be called before the ObjectStore exception handler unwinds the call stack or exits from the program. You can set a breakpoint in this function and then examine the stack using the Visual C++ debugger. Here is a code sample illustrating how to use this function:

```
#include <iostream.h>
#include <ostore/ostore.hh>
void break_hook(tix_exception *err, os_int32, char* message){
cout << "Set break point here" << endl;
cout << "Have a look at the stack " << endl;
// You may also want to put up a message box that displays
// the contents of the message parameter.
}
main(int argc, char *argv[])
{ OS_ESTABLISH_FAULT_HANDLER
  os_database *db1;
  objectstore::initialize();
  tix_exception::set_unhandled_exception_hook(break_hook);
  do_something_fun();
  OS_END_FAULT_HANDLER
}
```

Building ObjectStore/Microsoft Foundation Class Applications

To build ObjectStore/MFC applications, users need to address a number of issues, as described here. (See *Generating MFC*

Applications Using ObjectStore AppWizards on page 119 for related information.)

- 1 Putting **OS_ESTABLISH_FAULT_HANDLER** and **objectstore::initialize** code in **WinMain**.
- 2 Putting **OS_ESTABLISH_FAULT_HANDLER** in threads created with **CWinThread::CreateThread**.
- 3 Integrating ObjectStore's overloading of **operator new** and **delete** with MFC's **DEBUG_NEW** macro.
- 4 Adding knowledge of nonmapped persistent pointers to MFC's valid-address checking.
- 5 Adding support for persistent **new** of MFC types.

To ensure that these issues do not cause problems using MFC, add the following to your code.

Issue 1

Resolve issue 1 by copying the function **AfxWinMain** from **MFC\SRCS\WINMAIN.CPP** to an application source file, and adding the initialization calls to the copy. This will override the **AfxWinMain** in the unchanged **MFC DLL**.

Issue 2

Resolve issue 2 by putting the **OS_ESTABLISH_FAULT_HANDLER** and **OS_END_FAULT_HANDLER** macros in the thread functions passed to **CWinThread::CreateThread**.

Issue 3

Resolve issue 3 by modifying **stdafx.h** as follows. The reason for doing this is that MFC uses **#define new DEBUG_NEW** to activate the MFC debugging **malloc**, and that causes problems for the ObjectStore overloads of **new**. To eliminate these problems, redefine **DEBUG_NEW** to be **new** as opposed to **new(__FILE__, __LINE)** and then insert an inline **operator delete** that calls ObjectStore's internal **persistent delete** function. This, in turn, checks for transient pointers and calls **_ODI_free** if needed.

To do this, add the following to **stdafx.h** after including **ostore.hh** and **afxwin.h**:

```
// If we let DEBUG_NEW keep the definition that it's given in
// afxwin.h,
// and let the "#define new DEBUG_NEW" from afx.h stay, then our
// overloads of new won't be recognized by the compiler.

#ifdef _AFX_NO_DEBUG_CRT
#ifdef DEBUG_NEW
#undef DEBUG_NEW
```

```

#define DEBUG_NEW new
#endif
#ifndef new
#undef new
#define new new
#endif
#endif

```

Then, add the following at the end of `stdafx.h`:

```

// This declaration taken from ostore.h */
extern "C" void _OSSYSCALL objectstore_delete(void *);
// In DEBUG mode, MFC has its own operator delete. We need to get
// first dibs on the deletion if the thing being deleted is persistent.
//
#if defined(_DEBUG) && !defined(_AFX_NO_DEBUG_CRT)
inline void operator delete(void* p)
{
    objectstore_delete(p);
}
#endif

```

Issue 4 Resolve issue 4 by never passing persistent data to library calls. Pass copies instead.

Issue 5 Resolve issue 5 by not storing MFC classes persistently. See [Class `os_CString`](#) below.

Class `os_CString`

ObjectStore provides a shadow class of `CString` called `os_CString` that does the job of storing `CString` objects persistently. These classes can be used interchangeably for most purposes.

Class `os_CString` has the same layout and member functions, but it includes `get_os_typespec()` members and `cast-to-CString` members. This means that users can easily pass them from the database to MFC and back. To use `os_CString`, include `<ostore/oscstring.h>` in your source code.

You can find the source code for `os_CString` in the directory `%OS_ROOTDIR%\examples\ospmfc`.

Generating MFC Applications Using ObjectStore AppWizards

ObjectStore includes a custom AppWizard for VC++ 5.0. This wizard can be used to generate MFC applications that use ObjectStore.

When an application is generated with this wizard, all the code needed for ObjectStore is inserted into your project. The following is what is inserted in addition to standard code provided by VC++:

- Copy of **AfxWinMain()** that establishes ObjectStore fault handling and calls initialization functions
- Code to **stdafx.h** that includes ObjectStore header files and ensures that the correct **operator new** is called
- Skeleton for ObjectStore schema in **schema.scm** file
- ObjectStore headers and libraries that are added to the makefile

Ensure that the following environment variables are set:

- **INCLUDE** must include **%OS_ROOTDIR%\INCLUDE**.
- **LIB** must include **%OS_ROOTDIR%\LIB**.

These are set by ObjectStore installation.

The source code for the ObjectStore AppWizard can be found in the directory **%OS_ROOTDIR%\examples\ostoreaw**.

Using the AppWizard

Complete the following sequence to create an ObjectStore MFC application with the AppWizard:

- 1 Start Microsoft Developer Studio.
- 2 In the **File** menu, select **New**.
- 3 From the dialog box displayed, select the **Projects** tab and select **ObjectStore AppWizard**.
- 4 Enter the project name and click **OK**.
- 5 In these steps, select the MFC-specific options you prefer.
- 6 Check **Use ObjectStore Collections** if you want your application to use ObjectStore collections. This will include collections-related files and initialize ObjectStore collections at the right place.
- 7 Click on **Finish** to generate a project with correct ObjectStore include files and initializations.

Read the **readme.txt** file provided with this new project and make changes as described there. This project also copies a file **schema.scm** in your project. This is a skeleton for a schema file needed for ObjectStore. You can change this file and add **oss**

compilation options in your project, as described in the **readme.txt** provided in the project.

Manual steps

The ObjectStore AppWizard cannot add the schema compilation build step to the project. You must do this following the directions in **readme.txt**.

You must add **OS_ESTABLISH_FAULT_HANDLER** to the top-level threads functions when using threads.

Using the Visual C++ Integrated Development Environment (IDE)

Follow these steps to use the VC++ IDE to generate schema.

- 1 Choose a file name extension for your schema source file other than **.cpp**, for example, **.osg**.
- 2 Set up a Custom Build rule that specifies that the schema object file is to be built from the schema source file, by using the command

```
oss <other oss args> schema.osg
```

The ObjectStore MFC example is set up to use this technique.

Using ObjectStore Within a DLL

This section provides examples of how to use ObjectStore in a DLL.

In the directory **%OS_ROOTDIR%\examples\dll**, there are two subdirectories, **LIBSCHM** and **NOSCHM**. These two directories demonstrate two ways to use ObjectStore from within a DLL.

Each of these two directories contains two subdirectories, **PROG** and **LIB**. The **LIB** directory creates a DLL called **people.dll** that uses ObjectStore. The **PROG** directory creates a program that uses **people.dll**.

The difference between the directories **LIBSCHM** and **NOSCHM** is as follows:

- The directory **LIBSCHMLIB** creates a DLL that uses ObjectStore and a library schema to go with the DLL. The directory **LIBSCHMPROG** uses the library schema in addition to its own schema sources to create an application schema.
- The directory **NOSCHM** is an example of how you create a DLL that uses ObjectStore but is to be linked with programs that are

not using ObjectStore. The objective is to avoid concern with schema generation and linking rules in the application makefiles.

This approach is useful if you are in a development environment where a single person or a small group of people are the only developers using ObjectStore, or if your product is a persistent library that will be used by developers who do not have any knowledge of ObjectStore.

Building Applications on Machines Remote from the Server

You can build an application on a machine that is remote from the ObjectStore Server.

All databases, including application schemas and library schemas, must physically reside on the same machine as the Server. The schema generator expects to be able to connect to a Server and produces an error message if it cannot do so. Consequently, you must provide the schema generator with a Server-relative pathname. There are two ways to do this.

You can specify ObjectStore Server-relative pathnames even if you are not using NFS or another file-sharing option. This does not rely on any file system protocol. Instead, this syntax is recognized by ObjectStore tools. Using Server-relative pathnames, the `oss` command line would look like this:

Example 1

```
oss -asdb foo:c:\appdir\appschema.adb -assf ossschema.cc /  
-cd schmdefs.hh myschema foo:c:\ostore\lib\os_coll.lib
```

This example assumes that

- The Server is on a remote machine named **foo**.
- There is a directory on **foo**'s C drive called **appdir**.
- ObjectStore is installed on **foo**'s C drive in the **ostore** directory.

This method makes no assumptions about the availability of a remote file system protocol.

If you can use Windows networking to connect to a network drive on the Server, you can use pathnames that start with that network drive letter.

Example 2

If you are on a system that supports NFS, you can mount a directory on the Server as follows:

nfs use x: bar:/usr

This mounts the **/usr** file system of a remote Server machine called **bar** on the local directory **x**. Having done that, you use the following form for your **oss** command line:

```
oss -asdb x:\appdir\appschema.adb -assf ossschema.cc \  
-cd schmdefs.hh myschema.cc x:\ostore\lib\os_coll.ldb \  
x:\ostore\lib\os_query.ldb
```

This example assumes that

- The Server is on a remote machine named **bar**.
- The application is in **/usr/appdir** on **bar**.
- ObjectStore is installed in **/usr/ostore** on **bar**.

Porting ObjectStore Applications to Windows Platforms

This section presents guidelines for porting ObjectStore code to a Windows environment.

Macros for fault and exception handling

ObjectStore must handle all memory access violations, because some of those access violations are actually references to persistent memory in an ObjectStore database. On UNIX systems, ObjectStore can register a signal handler for such access violations. But on Windows, there is no function for registering a handler that also works when you are debugging an ObjectStore application; the **SetUnhandledExceptionFilter** function does not work when you are using the Visual C++ debugger.

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

ObjectStore provides two macros to use in establishing a fault handler:

- **OS_ESTABLISH_FAULT_HANDLER** establishes the start of the fault handler block.
- **OS_END_FAULT_HANDLER** ends the fault handler block.

These macros expand to nothing on platforms that do not require their use. Using these macros, a typical **main** function in an ObjectStore application would look like this:

Windows

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

A **WinMain** function would also look like the preceding example.

Threads

You must use the Visual C++ C run-time functions **_beginthread** and **_endthread**, as they properly initialize the Visual C++ C run-time library. **CreateThread** and **TerminateThread** do not properly initialize the Visual C++ C run-time library.

objectstore::initialize() need only be called once, even though an application has multiple threads.

long double and
warning C4069

Visual C++ makes **doubles** and **long doubles** the same size (eight bytes) and issues a warning whenever a **long double** is encountered. For example:

```
c:\ostore\include\ostore\mop.hh(1041) : warning C4069: long double is
the same precision as double
```

To avoid these warnings, ObjectStore header files use **pragma** statements that disable them. You can enable this warning for your code by adding the following after the ObjectStore include files:

```
#pragma warning ( default : 4069 )
```

/NODEFAULTLIB
option

The default libraries should not be used with multithreaded programs. The **/NODEFAULTLIB** or **/NOD** option tells the Visual C++ linker not to search the default libraries. During compilation, you can use the **/ZI** (lowercase *l* as in *library*) option to suppress default library search records in the object files.

Windows DEBUG and DDEBUG Builds of ObjectStore

The build of ObjectStore for Windows installed by the **SETUP** program is a retail release that was compiled optimized, without extra error checking and without debugging symbols, to make the smallest, most efficient installation package. Two debugging versions of ObjectStore that you can install manually to aid in debugging your applications are also included.

debug

The **debug** build is a drop-in replacement for the retail build, and can be used to obtain symbolic stack trace information that can

help you debug your application or that might be required by Object Design Technical Support to track down a problem. It is compiled unoptimized, and has some extra error-checking code built in.

ddebug

The **ddebug** build uses the debugging version of the Visual C++ run time, and so is compatible with the debug versions of the Microsoft Foundation Class and Visual C++ libraries. This build is most useful to application developers who are building and debugging a new application, because they can symbolically debug an entire application, including the run-time and MFC libraries, if used.

Installing DEBUG.ZIP or DDEBUG.ZIP

To install **DEBUG.ZIP** or **DDEBUG.ZIP**, follow these steps:

- 1 Install ObjectStore Release 5.1 with the **SETUP** program. See ObjectStore installation for Windows NT documentation for instructions.
- 2 Shut down the ObjectStore Server and Cache Manager by using the ObjectStore **SETUP** program. Answer **Yes** to the question about shutting down servers, then exit from **SETUP**.
- 3 Go to the **%OS_ROOTDIR%** directory.
- 4 Rename **bin** and **binsngl** directories (from the command prompt window or **Windows Explorer**) to **retail.bin** and **retail.binsngl**.
- 5 Unzip the file (**DEBUG.ZIP** or **DDEBUG.ZIP**) from the command prompt by typing the following command. The **-d** option creates and restores the directories included in the zip file.
pkunzip -d debug.zip
- 6 Run the ObjectStore **SETUP** program to start the Server. In the first setup dialog, select the **Setup Server** option. In the menu **Choosing to start ObjectStore services automatically**, select **Yes**. Then a **Confirm Message** dialog asks if you want to start the services right now. Select **Yes**.

To use the retail or debug builds, compile your application using the **/MD** switch. Using **Project | Settings** in Developer Studio, select **C/C++, Category Code Generation, Use runtime library Multithreaded DLL**. This automatically selects **ostore.lib** as a

default library. You can then switch between retail and debug builds by changing your **PATH**.

To use the **ddebug** build, install the **ddebug** libraries as directed in Installing DEBUG.ZIP or DDEBUG.ZIP on page 125. Then compile your application using the **/MDd** switch. Using **Project | Settings** in Developer Studio, select **C/C++, Category Code Generation, Use runtime library Debug Multithreaded DLL**. This automatically selects **ostored.lib** as a default library. To run the resulting application, ensure that the **ddebug** build is in your **PATH**.

The following table compares the features of retail ObjectStore, **debug**, and **ddebug**:

<i>Characteristic</i>	<i>Retail</i>	debug	ddebug
Installation with INSTALL?	Yes	No	No
Optimized?	Yes	No	Yes
ObjectStore symbols available?	No	Yes	Yes
Drop-in capability?	Yes	Yes	No

Compiling and linking applications

Use the following information when compiling and linking applications:

	<i>Retail</i>	debug	ddebug
Run-time libraries (DLL)	msvcrt.dll	msvcrt.dll	msvcrt.d.dll
Compile Options	/MD	/MD	/MDd *
Link Library	ostore.lib	ostore.lib	ostored.lib
ObjectStore libraries (DLLs)	O4....DLL	O4....DLL	D4....DLL

* The **/MDd** option defines the symbol **_DEBUG**, which determines the link library used by ObjectStore.

OS/2

This section provides information about compiling and linking ObjectStore applications using the VisualAge C++ compiler, **icc**, on OS/2.

Using Compiler Options

Required

You must always use these compiler options:

/Gd Dynamically link run-time library
/Gm Multithreaded libraries

Prohibited

You cannot use these compiler options with ObjectStore:

/Gr Device driver
/Rn Subsystem environment

You cannot use these compiler options when compiling code that includes ObjectStore header files:

/EHa No exception support
/H Short external names
/Ms Use **_System** calling sequence
/Sc Use cfront language standard
/Sg Set margins
/Sp Pack structures
/Sq Use sequence numbers

Specifying default language for files

Use **/Tdp** to compile all source and unrecognized files that follow on the command line as C++ files. You can specify **/Td** anywhere on the command line to return to the default rules for the files that follow it.

No rules

There are no rules for compiler and linker options not explicitly mentioned. You can decide how to set them.

Note: Previous releases of ObjectStore on OS/2 required the **/Su4** compiler option. While it is not required for ObjectStore Release 5.1 applications, you must continue to specify it if you want your application to be compatible with databases that were created with ObjectStore Release 3. The **/Su4** option is required for compatibility whether or not you upgraded your databases to be

ObjectStore Release 5.1 databases. An alternative to specifying this option is to perform schema evolution on the existing databases.

Linking Your Files

After you generate the application schema, you can link your application object files, ObjectStore libraries, and the application schema object file into an executable.

Required library

You must link ObjectStore applications with the library **ostore.lib**. To do so, you can either add **%OS_ROOTDIR%\lib** to your **LIB** environment variable and put **ostore.lib** on the link command line, or you can put **%OS_ROOTDIR%\lib\ostore.lib** on the link command line.

The reference to **ostore.lib** can be anywhere on the link command line as long as it appears before any explicit references to **CPPOM30I.LIB** (the C++ run-time library).

You must call the function **objectstore::initialize()** before calling any other ObjectStore interface function.

Case sensitivity

ObjectStore applications run correctly whether or not linking is case sensitive. Object Design recommends case-sensitive linking.

Compiling the application schema source file

You must compile the application schema source file produced by **oss** into an object file. This file includes ObjectStore header files and none of your header files, so the compilation command line is simple. For example:

```
icc /C /Gd /Gm myschema.cpp
```

This produces a file named **myschema.obj**.

Linking object files into an executable

A typical link command line is

```
icc /Tdp /B"STACK:32768" /Femy_exec.exe myschema.obj \ my_
ob1.obj my_ob2.obj %OS_ROOTDIR%\lib\ostore.lib
```

Sample makefile

In makefiles, do *not* specify an existing ObjectStore database as the application schema database. Doing so can corrupt your build process if the Server log has not been propagated to the database.

```
myschema.obj: myschema.cpp
  ossg -assf myschema.cpp -cd schmdefs.hh -asdb
myschema.adb \
  myschema $(OS_ROOTDIR)\lib\os_coll.ldb
icc /C /Gd /Gm myschema.cpp
```

```
my_exec.exe: my_exec.obj myschema.obj
icc /Tdp /B"STACK:32768" /Femy_exec.exe my_exec.obj \
myschema.obj $(OS_ROOTDIR)\liblostore.lib
```

Sample Makefile Without Library Schemas

```
OBJECTS=note.obj
EXECUTABLES=note.exe

APPLICATION_SCHEMA_PATH=note.adb

all: $(EXECUTABLES)

myschema.obj: myschema.cc
icc /C /Gd /Gm /Ti myschema.cc

myschema.cc: schema.cc
ossg -assf myschema.cc -cd schmdefs.hh \
-asdb $(APPLICATION_SCHEMA_PATH) \
$(CPPFLAGS) schema.cc

note.obj: note.cc
icc /C /Gd /Gm /Ti note.cc

note.exe: $(OBJECTS) myschema.obj
icc /Tdp /B"STACK:32768" /Fenote.exe $(OBJECTS)
myschema.obj \
$(OS_ROOTDIR)\liblostore.lib

clean:
-osrm -f $(APPLICATION_SCHEMA_PATH)
-del $(OBJECTS) osschema.*
```

Sample Makefile for an Application That Uses Collections and Queries

The following makefile adds the specification of two library schemas to the previous makefile. This is for an OS/2 application that uses collections and queries. Note that, in Release 3, you only needed to specify the collections library schema, which included the query library schema. In ObjectStore Release 5.1, if you use both collections and queries you must specify a library schema for each feature.

```
OBJECTS=note.obj
EXECUTABLES=note.exe

APPLICATION_SCHEMA_PATH=note.adb

all: $(EXECUTABLES)

myschema.obj: myschema.cc
icc /C /Gd /Gm /Ti myschema.cc

myschema.cc: schema.cc
ossg -assf myschema.cc -cd schmdefs.hh \
```

OS/2

```

Add query library schema
Add collections library schema

-asdb $(APPLICATION_SCHEMA_PATH) \
$(CPPFLAGS) schema.cc \
$(OS_ROOTDIR)\lib\osquery.lib \
$(OS_ROOTDIR)\lib\os_coll.lib

note.obj: note.cc
icc /C /Gd /Gm /Ti note.cc

note.exe: $(OBJECTS) myschema.obj
icc /Tdp /B"STACK:32768" /Fnote.exe $(OBJECTS)
myschema.obj \
$(OS_ROOTDIR)\lib\ostore.lib

clean:
-osrm -f $(APPLICATION_SCHEMA_PATH)
-del $(OBJECTS) osschema.*
```

Debugging Your Application

By default, **IPMD** halts for every memory exception and asks whether to proceed. Because ObjectStore applications take memory exceptions regularly in the course of persistent storage management, it is often convenient to disable this behavior.

You can do this by using the OS/2 environment variable **PMDEXCEPT**. When this environment variable is set to **1**, the debugger unconditionally passes ObjectStore exceptions to ObjectStore.

However, if your application fails with a general protection fault, setting **PMDEXCEPT** causes the program to exit before you can debug the error.

If you need to debug this kind of problem, set a breakpoint on the first instruction in **O4LOW.DLL**. This breakpoint occurs only if there is a general protection fault that ObjectStore declines to handle (such as dereferencing a null pointer).

Obtaining a stack trace

To obtain a complete stack trace, use the debug versions of the ObjectStore DLLs. The debug DLLs are available on the distribution CDROM.

To obtain a back trace using the debug DLLs, put the debug DLL directory in the front of the **LIBPATH** environment, reboot, and then run the **IPMD** debugger.

Pass Source Files to ossg

If you pass object modules instead of source modules to **ossg** on the command line, the schema generator displays a parsing error. Be sure to pass source modules to **ossg**.

Building Applications on Machines Remote from the Server

You can build an application on a machine that is remote from the ObjectStore Server.

All databases, including application schemas and library schemas, must physically reside on the same machine as the Server. The schema generator expects to be able to connect to a Server and produces an error message if it cannot do so. Consequently, you must provide the schema generator with a Server-relative pathname. There are two ways to do this.

You can specify ObjectStore Server-relative pathnames even if you are not using NFS or another file sharing option. This does not rely on any file system protocol. Instead, this syntax is recognized by ObjectStore tools. Using Server-relative pathnames, the **ossg** command line would look like this:

Example 1

```
ossg -asdb foo:c:\appdir\appschema.adb -assf ossschema.cc /  
-cd schmdefs.hh myschema foo:c:\ostore\lib\os_coll.ldb
```

This example assumes that

- The Server is on a remote machine named **foo**.
- There is a directory on **foo**'s C drive called **appdir**.
- ObjectStore is installed on **foo**'s C drive in the **ostore** directory.

This method is preferable to the file sharing method because it is more portable. It makes no assumptions about the availability of a file system protocol.

Example 2

If you are on a system that supports NFS, you can mount a directory on the Server as follows:

```
mount x: foo:c:\
```

This mounts the C drive of a remote Server machine called **foo** on the local directory **x**. Having done that, you use the following form for your **ossg** command line:

```
ossg -asdb x:\appdir\appschema.adb -assf ossschema.cc \
```

OS/2

```
-cd schmdefs.hh myschema.cc x:\ostore\lib\os_coll.ldb \  
x:\ostore\lib\os_query.ldb
```

The same assumptions apply here as in the previous example.

Chapter 5

Building Applications for Use on Multiple Platforms

You can build an ObjectStore application on multiple platforms and then use it to store and update data interchangeably on any of these platforms. Applications that run on more than one platform are considered to be heterogeneous.

This chapter provides instructions for building heterogeneous applications. This chapter does not provide information about how to make your application portable.

It covers the following topics:

General Instructions	134
Which Platforms Can Be Heterogeneous?	136
When Is a Schema Neutral?	138
Restrictions	140
ossg Neutralization Options	145
Neutralizing the Schema	148
Listing Nondefault Object Layout Compiler Options	156
Description of Schema Generator Instructions	162
Endian Types for ObjectStore Platforms	166

General Instructions

You can build an ObjectStore application on multiple platforms and then use it to store and update data interchangeably on any of these platforms. This is referred to as *heterogeneity*. Applications that allow heterogeneity are considered to be *heterogeneous*.

To make an application heterogeneous, you must neutralize its schema for all platforms and then build the application on each platform. *Neutralization* is the process of modifying a schema so that it has identical data formats on each platform that runs the application. This is necessary because different compilers lay out data in different ways.

When building a heterogeneous application, consider address space limitations on all platforms. Database access patterns might work on some platforms but not on others.

You can start with an application that runs on one platform or you can create a new application. If you are building a new application, see the limitations in Restrictions on page 140 before you design your application.

When you have a working application, follow these steps to make it heterogeneous. As always, you can use the command-line interface or a makefile.

- 1 Run the schema generator to determine what you must do to neutralize your application.
 - a Specify the **-architecture setn** (or **-arch setn**) option for the set of compilers on which the application must run.
 - b You can also specify other **oss** options.

A description of the schema generator neutralization options is on page 140.

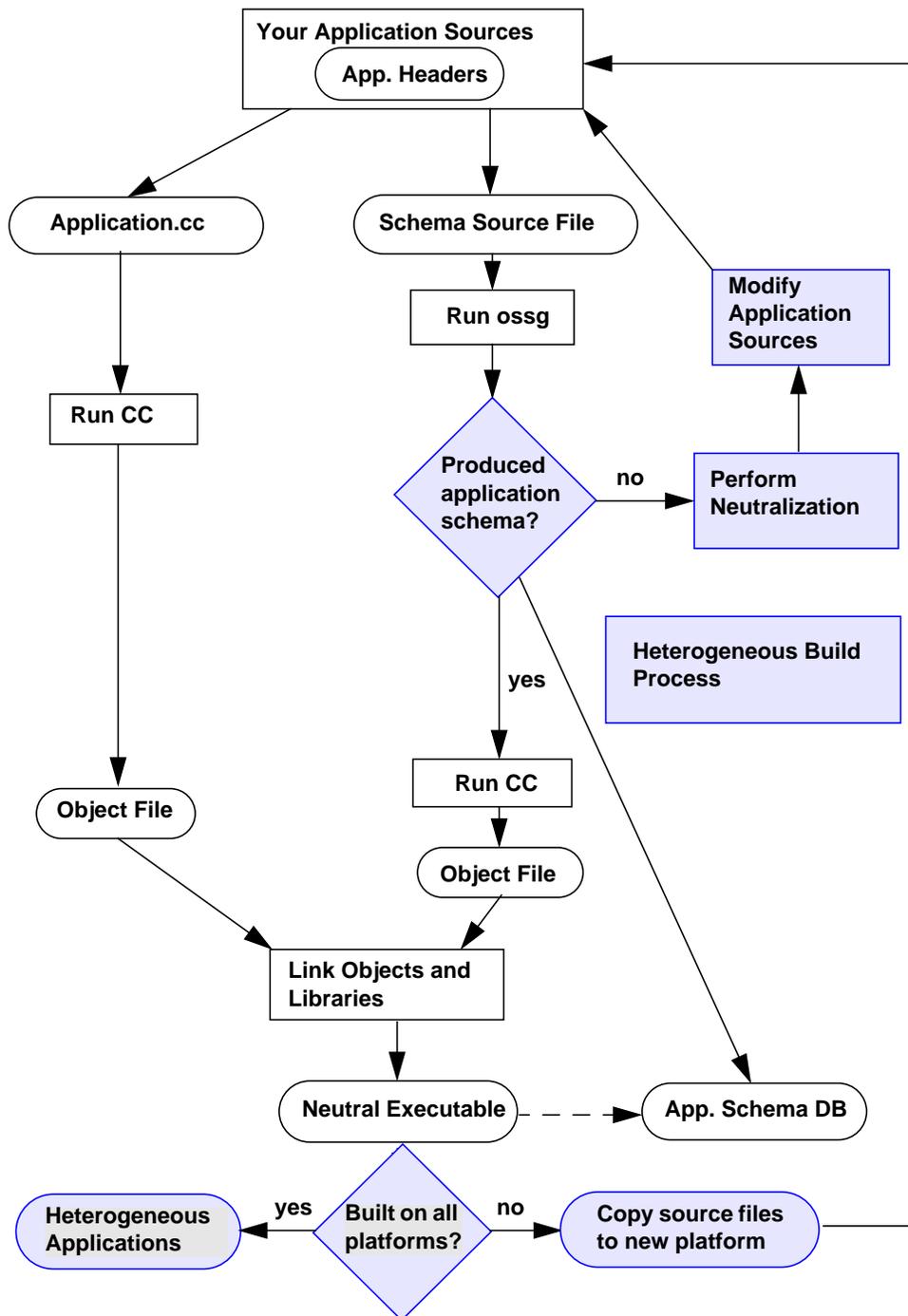
- 2 Modify your application source files according to the instructions you receive from the schema generator. Some instructions require you to insert macros in your source code. Be sure to enter the exact name specified by the schema generator.
- 3 Repeat steps 1 and 2 until the schema generator no longer displays instructions to change your source files.

If you follow the neutralization instructions correctly, you should need to repeat step 1 only once and step 2 not at all.

- 4 Recompile your application source files.
- 5 Compile the application schema source file generated by **ossg**. Be sure to do this after the schema generator has successfully produced the application schema database.
When you use Visual C++, the schema generator creates the application schema object file directly. On all other platforms, you must compile the application schema source file yourself.
- 6 Link the application object files and the application schema object file and any required libraries to create a neutralized executable.
- 7 When your neutralized application works on the first platform, copy the source files to each additional platform.
- 8 Build the application on each platform:
 - a Run **ossg**. Always specify the **-arch set***n* option. You should not receive additional instructions to modify your source files, but this catches any changes that affect neutralization.
 - b Compile.
 - c Link.

The following figure illustrates the workflow for neutralizing an application and then building it on multiple platforms.

Which Platforms Can Be Heterogeneous?



You can build an application that runs under all operating systems listed. The compilers you can use are also shown. See the *ObjectStore C++ Interface Release Notes* for ObjectStore Release 5.1 for the latest versions of supported compilers.

Normally, you cannot build an application that can run on both 32-bit and 64-bit platforms. The only exception to this rule is that Digital UNIX can operate heterogeneously with 32-bit platforms. The compiler on this 64-bit platform allows you to mix 32-bit and 64-bit pointers through pragma statements in the source code. When all pointers in persistent classes are forced to be of the 32-bit variety, you can share these classes with 32-bit platforms. The following table includes the supported compilers.

Platform	Operating System	Compiler
AXP (Alpha)	Digital UNIX	DEC C++
HP 700 and HP 800	HP-UX	HP C++
IBM RS/6000	AIX	IBM C Set ++
Intel	OS/2 Warp 3.0	VisualAge C++
	Windows NT 3.5 and Windows 95	Visual C++
	Solaris 2	Sun ProCompiler C++
SGI MIPS	IRIX	SGI C++
Sun SPARC	Solaris 2 (includes SunOS 5)	Sun SPARCompiler C++

When Is a Schema Neutral?

When an application is heterogeneous, its persistent objects are laid out in memory in identical formats for all platforms accessing the objects. The sizes and offsets of all data elements must be identical in both the creating architecture and the accessing architecture.

What Causes Data Formats to Vary?

Compilers on different platforms can use different rules for laying out objects. Machine architectures might have differing requirements for alignment of data types. This can cause differing object layouts. For example, the following class might have a different layout in memory if the required alignment of the `int` data type is two-byte alignment instead of four-byte alignment:

```
struct X {  
    char status;  
    int value;  
};
```

Layout incompatibilities are related to

- Alignment requirements for primitive data types, classes, or structs
- Bit-field packing rules
- Hidden compiler data structures

Additionally, even when a class is defined so that it is identically laid out on all platforms, the fundamental nature of the processor can result in differing formats of the data values themselves. For example, the standard SPARC architecture uses a byte order different from the Intel microprocessor line. If an integer is written on one platform, its format might need to be converted for it to be readable on another platform.

Other features that can cause data formats to vary are

- Virtual function table pointers (vtbls)
- Virtual base classes
- Zero-length base classes

How Can You Create Identical Data Formats?

When you run **oss** with neutralization options, the schema generator determines where padding is needed to create identical data formats. The schema generator examines the schema for a set of compilers that you specify. It then displays instructions for you to insert padding macros in your source files. After you modify your source files, you run **oss** again. If you followed the instructions correctly, the schema is now neutralized for the compilers in the specified set.

The schema generator always proposes adding explicit padding to your classes to achieve neutralization. In many cases, however, you can reduce the size of the end result by reordering the members of a class to eliminate some or all padding. For example:

```
class X {  
    char a;  
    int i;  
    char b;  
    int j;  
};
```

The best way to neutralize this code is to reorder the members:

```
class X {  
    int i;  
    int j;  
    char a;  
    char b;  
};
```

Because some applications depend on the order of members within a class, it is up to you to decide whether or not this method is appropriate for each case.

After you neutralize a schema, a class in the schema is the same size on each platform on which you run the application. If you run the **oss** utility, the returned value is the actual size of the object in the database. Objects belonging to a particular class are the same size on each platform on which they exist.

Restrictions

When designing applications, it is important to be familiar with the restrictions on heterogeneity.

Virtual Base Classes

When you are using several types of compilers, the use of virtual base classes makes neutralization particularly complex because of differences among the compilers in the way they lay out objects.

You cannot use virtual base classes when some but not all of your compilers use a cfront or SGI layout.

Applications compiled with Sun ProCompiler C++, AIX C Set ++, DEC C++, VisualAge C++, and Visual C++ can use virtual base classes in a heterogeneous environment.

Primitive Data Types

The compilers OS/2 VisualAge C++ (**icc**), Sun C++, and AIX C Set ++ (**x1C**) each support a **long double** data type that has no counterpart on any other platform. If you use the **long double** type, you cannot neutralize your application. Use the **double** data type instead.

If you expect to run a heterogeneous application on a Digital UNIX platform, avoid using the **long** data type. On the Alpha platform, **long** data types are 64 bits, rather than the usual 32 bits. Types can only be neutralized for architectures that use the same size for the type.

In general, you should use 32-bit integers and floating-point formats that are four or eight bytes.

64-Bit Pointers

You cannot neutralize a schema containing 64-bit pointers so that it works on a 32-bit platform. See DEC C++ 64-Bit Pointer Considerations on page 91 for detailed information about mixing pointer size.

Floating-Point Data Conversion

During database access, it is not known whether or not the data might have been written by an application running on another platform. ObjectStore automatically converts data if necessary so that data is in the correct format for your application. There is, however, an exception to this rule.

ObjectStore does not convert floating-point data formats. To do so would introduce inaccuracies in your data. When floating-point formats are different but the data size is the same, ObjectStore only swaps bytes, which causes the result to be meaningless.

Pointers to Members

On some platforms, you cannot use pointers to members in persistent classes.

DEC C++ does not allow pointer-to-data-members (PTODMs) or pointer-to-member-functions (PTOMFs) to be persistently maintained. Visual C++ PTOMFs cannot be persistently maintained.

On other platforms, you can store PTODMs persistently and use them heterogeneously, with this exception: cfront and Sun C++ PTODM formats do not support PTODMs that refer to a member of a virtual base class.

Base Class Initialization Order

Occasionally, you might receive neutralization instructions from **ossg** that involve the reordering of base classes. Following these instructions might disturb dependencies in your source files. If this occurs, you might need to modify your source code to accommodate changes in the base class initialization order as well as to accommodate neutralization.

Parameterized Classes

The schema generator instructs you to neutralize class instantiations rather than the template class itself. This is because you can create a template specialization with different parameters that produces a different layout. This can present a problem. For example:

```
template <class T> class Y
```

Restrictions

```
{
public:
    char a;
    T b;
};
Y<char> y_char;
Y<double> y_double;
```

The `Y<char>` template requires no additional padding to be neutral. However, the `Y<double>` template does require padding. To define the schema as neutral, you can follow the neutralizer instructions and define a template *specialization* for `Y<double>`.

Template specialization

A template specialization is a special kind of replacement class. In certain cases, you might want to specify that for a template with a particular set of template arguments, the instantiation generated from the template not be used. Instead, a replacement class that you specify should be used. These replacement classes are known as template specializations. Here is an example of where one might be useful:

```
/* a simple template class that holds a data item */
template<class T> class Data {
public:
    Data(T& the_data) : data(the_data) {}
    T data;
};

/* a specialization that knows to strdup string data */
class Data<char*> {
public:
    Data(char* the_data) : data (the_data ? strdup(the_data) : 0) {}
    ~Data() { if (data) free(data); }
    char* data;
};
```

Here is the template specialization you would define for the example:

```
class Y<double>
{
public:
    char a;
    char_os_pad_1[7]
    double b;
};
```

You might find that the schema generator instructs you to specialize each class instantiation in the same way. In this case, it is easier for you to change the template itself. Since the schema

generator does not provide instructions for modifying class templates, it is up to you to decide when this is appropriate. For example:

Original template definition	<pre>template <class T> class linked_list { public: T* data; char flag; linked_lists<T*> next; };</pre>
Used by this type	<code>linked_list<int></code>
Template specialization based on schema generator instructions. This provides a neutral storage format.	<pre>class linked_list <int> { public: T* data; char flag; char _os_pad_1[3]; linked_lists<T*> next; };</pre>
Alternatively, you can modify the template this way. Instantiations of this template would be neutral.	<pre>template <class T> class linked_list { public: T* data; char flag; char _os_pad_1[3]; linked_lists<T*> next; };</pre>

Sizes for Data Types

A neutralized schema requires that a data type be the same size on all platforms. When the types involved are classes, the schema generator can determine how to achieve the same size. But when the types involved are primitive types, the schema generator cannot provide instructions for padding to achieve the same size. In this case, **ossg** displays messages similar to the ones following. You might receive multiple versions of the second message.

```
<err-0013-0006>The following neutralization problems occurred during
the compilation of file m.cc:
failures for class E:
<err-0013-0011>Components of the class have differing sizes:
Data member e
```

Restrictions

In general, you must do one of the following:

- Choose an alternative data type that has a uniform size across the platforms in the architecture set.
- Use command-line options or pragma statements that force the data type to be a uniform size.

An example of nonstandard type sizes is when you use **enum** types and your architecture set includes OS/2. On OS/2, use the **/Su4** option to force **enums** to have a size of four bytes.

General Restriction

Occasionally, there are classes that require a special alignment that the schema generator cannot provide. Generally this refers to certain special cases of classes with virtual bases (usually multiple virtual bases) where at least one base class requires double alignment (usually with **set3** heterogeneous).

If you carefully followed the neutralization instructions and you still receive the **Neutralization failure** message, it might mean that your schema includes such a class. In this situation, contact Object Design Technical Support for assistance in neutralizing your schema.

An example of when this might occur is when you are using virtual base classes that contain **doubles**. Even with a single compiler (cfront is the most likely one), there might be cases where the schema generator cannot adequately align the virtual base classes for all platforms.

ossg Neutralization Options

You neutralize a schema by running **ossg** with neutralization options. The **-arch set***n* option is required; the other options are not. The table describes the neutralization options.

-arch set*n*

The schema that is generated or updated will be neutralized to be compatible with the architectures in the specified set. Applications running on these architectures can then access a database associated with the schema.

Required when you are neutralizing schema. No default.

You can specify one of the following sets.

set1 *Some 32-bit architectures*

HP-UX HP C++
IBM VisualAge C++ for OS/2
Intel Solaris 2 Sun C++
Intel Windows NT and Windows 95
RS/6000 AIX C Set ++
SGI IRIX SGI C++
SPARC 2 Sun C++

set2 *set1 without cfront architectures*

IBM VisualAge C++ for OS/2
Intel Solaris 2 Sun C++
Intel Windows NT Visual C++
Intel Windows 95 Visual C++
RS/6000 AIX C Set ++
SPARC Solaris 2 Sun C++

set3 *cfront architecture*

HP-UX HP C++
RS/6000 AIX C Set ++
SPARC 2 Sun C++

set4 *Some IBM architectures*

IBM VisualAge C++ for OS/2
RS/6000 AIX C Set ++

oss-g Neutralization Options

set5 **set1** plus Digital UNIX DEC C++

Restriction: Your schema cannot contain a data member of type **long**.

set6 **set2** plus Digital UNIX DEC C++

Restriction: Your schema cannot contain a data member of type **long**.

set7 **set1** with Windows NT Alpha

set8 **set2** with Windows NT Alpha

set9 **set5** with Windows NT Alpha

set10 **set6** with Windows NT Alpha

set11 **set6** with SGI and HP-UX support

-neutral_info_output *filename*
or **-nout** *filename*

Indicates the name of the file to which neutralization instructions are directed.

Optional. Default is that the schema generator sends output to **stderr**.

-noreorg or **-nor**

Prevents the schema generator from instructing you to reorganize your code as part of neutralization. This is useful for minimizing changes outside your header file, working with unfamiliar classes, or simply padding formats.

When you include **-noreorg**, your application might not make the best use of its space. In fact, it is seldom possible to neutralize a schema without reorganizing classes.

When you use virtual base classes, it is very unlikely that you can neutralize your schema when you include this option.

Optional. The default is that the schema generator provides reorganization instructions.

-pad_maximal or **-padm**
-pad_consistent or **-padc**

Indicates the type of padding requested.

-pad_maximal or **-padm** indicates that maximal padding should be done for any ObjectStore-supported architecture. This means all padding, even padding that the various compilers would add implicitly.

-pad_consistent or **-padc** indicates that padding should be done only if required to generate a consistent layout for the specified architectures.

Optional. Default is **-padc**.

-schema_options <i>option_file</i> or -sopt <i>option_file</i>	Specifies a file in which you list compiler options being used on platforms other than the current platform. The options in this file usually override the default layout of objects, so it is important for the schema generator to take them into account. See page 156 for details about the content of the option file. Optional. No default.
-show_difference or -showd -show_whole or -showw	Indicates the description level of the schema neutralization instructions. Optional. Default is -show_whole .

Neutralizing the Schema

Follow these instructions to generate a neutral application schema.

- 1 Run the schema generator with neutralizer options to determine what changes in your source files (usually header files) allow your application to be heterogeneous.

When you initiate the schema generator, you specify the set of platforms for which you want to neutralize. The schema generator examines the classes in the schema to ensure that the interpretation of the class definitions yields identical layout results on all platforms in the specified set.

When there are layout discrepancies, the schema generator determines the changes needed to produce a neutral layout, and provides instructions for modifying your source files.

To invoke **ossg**, use the same format you use to generate an application, library, or compilation schema. The only difference is the addition of neutralizer options. The following format shows the addition of neutralization options when generating an application schema. You can also add neutralization options when you generate a library or compilation schema. For an explanation of the **ossg** command line, see Generating an Application or Component Schema on page 35. (Portions of the command line are on different lines only for clarity.)

```
ossg [compilation_options] neutralizer_options  
[other_schema_generator_options]  
{-assf app_schema_source_file | -asof app_schema_object_file}  
-asdb app_schema_database schema_source_file  
[lib_schema.ldb ... ]
```

- 2 Change your source files according to the instructions from the schema generator.
- 3 Run **ossg** again.
If you followed the neutralization instructions correctly, you should now have a neutralized schema and you can skip step 4.
- 4 Continue to modify your source files according to **ossg** instructions and then run **ossg** until the schema generator successfully produces your application schema.

5 Recompile your source files.

When you run the schema generator with neutralization options, neutralization is limited to the classes defined in the schema source file and any classes defined in an include file that is directly or indirectly included in the schema source file. This means that when **ossg** generates an application schema, the schema generator does not examine class definitions that are in existing schemas. This limitation has no meaning when **ossg** generates a library or compilation schema because an existing schema is not involved.

When to Use Neutralization Options

After you neutralize your application, you should continue to include the **-arch set*n*** option whenever you run **ossg**. This ensures that changes do not cause platform conflicts.

Specifying the **-arch set*n*** option creates internal information needed by the relocation subsystem. This information ensures access to the correct virtual function tables.

If you remove the **-arch set*n*** option, you might add data that is not properly aligned for a particular platform. The schema generator normally selects the most restrictive alignment. Without the **-arch set*n*** option, the schema generator cannot determine that the alignment must be larger on certain platforms.

Additional Neutralization Considerations

Be sure to generate an application schema that is neutral for each platform on which you plan to run the application. When a schema is not explicitly neutralized for a platform, you might receive run-time schema validation errors when you invoke the application on that platform.

Be sure to mark all required data types in the schema source file. When you fail to mark a required type, the schema generator cannot detect incompatibilities.

The schema generator might require neutralization to prevent straddling pointers (pointers that span pages). This might be necessary

- On platforms that do not require four-byte alignment of pointers
- When a pragma or compiler option causes tighter packing

Updating a Database Schema to Be Neutral

Suppose you have an existing application on one platform that already stores information in a database. Now you want to neutralize the application schema to create a heterogeneous application. In this case, you must also update the database schema and the data so it matches the neutralized application schema. To do this, use the `ossevol` utility or a custom evolution application using the schema evolution library.

Benefits of Compiler Groups

There are several reasons to group the cfront platforms separately.

- The amount of padding necessary is usually less than with other, larger groupings. The result is smaller objects and smaller databases than would otherwise be possible.
- These groups are consistent with the heterogeneous architectures in ObjectStore Release 3.0. If you are running an ObjectStore Release 3.0 heterogeneous application, you can use one of these compiler groups to minimize, and often eliminate, the padding usually needed to upgrade to Release 4, and subsequently to Release 5.1.
- `set3` is a separate group because cfront handles virtual base classes very differently from other compilers.

The more platforms for which you neutralize your schema, the more space you use. If you neutralize for fewer platforms, you decrease the amount of space you use but you cannot run your application on as many types of platforms.

Command Line and Neutralization Examples

Here is a sample `oss` command line on a Sun SPARC system:

`oss` command line

```
oss -arch set1 -padm -assf proj_schema.cc \  
-asdb progschema.adb \  
proj_schema_source.cc $(OS_ROOTDIR)/lib/liboscol.ldb \  
$(OS_ROOTDIR)/lib/libosqry.ldb
```

The schema will be neutralized for use on platforms belonging to **set1**. Maximal padding will be done. Since this is not being invoked on a Windows platform, the schema generator produces an application schema source file (**prog_schema.cc**) that you must compile. The application schema will include the type information from the two library schemas specified.

Neutralization
example

Here is a schema source file:

```
#include <ostore/ostore.hh>
#include <ostore/manschem.hh>
class A {
public:
    virtual void fun1();
    double d;
    A();
};
OS_MARK_SCHEMA_TYPE(A);
```

First, try running the schema generator and specifying **-arch set1** for schema neutralization. These examples assume that you are running **oss** on a UNIX platform. To try these examples on a Windows NT platform, you would specify **-asof** instead of **-assf**. Also, on Windows and OS/2 platforms, the specification of the path for ObjectStore header files would be **-I%OS_ROOTDIR%\include** instead of the way it is in the examples.

```
oss -assf hetero.cc -asdb hetero.adb -arch set1 \
-I$OS_ROOTDIR/include hetero.cc
```

Schema generator
output

```
<err-0013-0002>The schema must be neutralized in order to operate
heterogeneously with the architectures specified:
The following schema modifications must occur:
class A :
    public os_virtual_behavior /* New */
{
    public:
    char _os_pad_0[4]; /* New */
    double d;
};
```

In the output, the schema generator does not display the member functions. It displays only the data members and nested types in the class definitions.

Also, the **/* New */** comment in the output flags the changes you need to make. You must edit the schema source file to have this content:

Neutralizing the Schema

Modified schema
source file

```
#include <ostore/ostore.hh>
#include <ostore/manschem.hh>
class A : public os_virtual_behavior {
public:
    virtual void fun1();
    double d;
    char _os_pad_0[4];
    A();
};
OS_MARK_SCHEMA_TYPE(A);
```

Now the previous **oss** command line generates the schema with no error messages.

Use of **-noreorg**

Suppose that the first run of the schema generator also had the **-noreorg** switch specified:

```
oss -assf hetero.assf -asdb hetero.adb -noreorg -arch set1 \
-!$OS_ROOTDIR/include hetero.cc
```

<err-0013-0002>The schema must be neutralized in order to operate heterogeneously with the architectures specified:
The following schema modifications must occur:

```
class A
{
    public:
    os_pad_vftbl_start /* New */
    char _os_pad_0[4]; /* New */
    double d;
    char _os_pad_1[4]; /* New */
    os_pad_vftbl_end /* New */
};
```

Working from this output, edit the original class description as follows:

Modified class
description

```
class A {
public:
    virtual void fun1();
    os_pad_vftbl_start
    char _os_pad_0[4];
    double d;
    char _os_pad_1[4];
    os_pad_vftbl_end
    A();
};
```

Next, modify the original **oss** command line by adding the **-show_difference** option. The only effect that this has is on the way that the changes are presented:

Use of
-show_difference

```
ossg -assf hetero.assf -asdb hetero.adb -arch set1 -show_
difference\
-!$OS_ROOTDIR/include hetero.cc
```

<err-0013-0002>The schema must be neutralized in order to operate heterogeneously with the architectures specified:

The following schema modifications must occur:

Changes for class A:

Add os_virtual_behavior as the first base class

Add a padding member as the first member: char _os_pad_0[4];

As you can see, the output in this case is simpler. For complex classes, the difference mode of display might be more easily understood.

For most classes, the default (-show_whole) display behavior is probably simplest.

Virtual base example

Here is a more complicated class involving virtual bases:

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

class A {
public:
    virtual void fun1(int);
    A();
};

class B {
public:
    virtual void fun2(char);
    B();
};

class C {
public:
    virtual void fun3(void*);
    C();
};

class D : public virtual A, public virtual B, public C
{
public:
    D();
    virtual void fun1(int);
};
OS_MARK_SCHEMA_TYPE(D);
```

In this case, neutralize with the **set2** architecture group because heterogeneity of classes with virtual bases between cfront and non-cfront-type platforms is not supported.

Neutralizing the Schema

Schema generator
command line

```
ossd -assf hetero.assf -asdb hetero2.adb -arch set2 \  
-I$OS_ROOTDIR/include hetero2.cc
```

```
<err-0013-0002>The schema must be neutralized in order to operate  
heterogeneously with the architectures specified:  
The following schema modifications must occur:
```

```
class/* file: m.cc line: 19 */  
class D :  
    public os_vb_fbsc<0,A> /* New */,  
    public C,  
    public os_vb_fbs<0,B> /* New */  
{  
};
```

Nonvirtual template
instantiations

In this example, you can see that the schema generator replaces the virtual bases with nonvirtual template instantiations. These instantiations virtually inherit from classes A and B in a way that ensures layout compatibility. Note that the two virtual base introducing templates are named slightly differently.

The particular template that the schema generator chooses for a given virtual base depends on the characteristics of the virtual base and the derived class.

Caution

When you modify files according to neutralization instructions be sure to follow the instructions exactly. In particular, when a class inherits from other classes the order of inheritance must be specified in the exact way expected by the schema neutralizer.

Using a Makefile to Obtain Neutralization Instructions

You can use a single makefile to invoke **ossd** and compile and link your code. The schema generator sends a nonzero return code when source changes are required for neutralization. This nonzero return causes **make** to stop processing. The schema generator displays neutralization instructions for changes to your source code.

UNIX makefile
example

```
all: my_exec  
  
my_exec: main.o os_schema.o foo.o bar.o  
    CC -o my_exec main.o os_schema.o foo.o bar.o -los -loscol  
    $(OS_POSTLINK) my_exec  
  
os_schema.cc: schema_source.o  
  
# os_schema.cc must depend on all headers that  
# schema_source.cc depends on.  
# You must also set up default rules so that schema_source.cc
```

```
# compiles to schema_source.o.  
ossd -arch set2 -showd -mrscp -asdb my_exec.adb \  
-assf os_schema.cc $(CPPFLAGS) schema_source.cc \  
$(OS_ROOTDIR)/lib/os_col.lib
```

Building a Heterogeneous Application from a Neutral Schema

After you neutralize a schema, there are two more steps before you have a heterogeneous application:

- 1 Finish building your application on the original platform.
- 2 After your application works on the first platform, build it on each platform on which you want it to work.

Neutralizing enums

OS/2 is the only platform on which you can specify multiple **enum** sizes. All other platforms use four bytes for an **enum**.

If you are neutralizing for any architecture set except **set3** (which does not include OS/2), the schema generator assumes that the schema is to be used on OS/2. Consequently, unless you specify otherwise, the schema generator expects OS/2 to use one byte for small-valued **enums**. This causes a neutralization error.

Regardless of whether or not you intend to use the schema on OS/2, you must do one of the following:

- Specify a dummy value inside the **enum** that guarantees that the **enum** uses four bytes. For example:

```
enum embedd_enum { enum_val1, enum_val2, enum_val3,  
enum_val4 = 100000 };
```

- On the **ossd** command line, specify **-schema_options** with the name of a compiler option file. In the compiler option file, specify that you will be using the **/Su4** option on OS/2. See Listing Nondefault Object Layout Compiler Options on page 156 for details.

Listing Nondefault Object Layout Compiler Options

When building C++ applications, you might encounter circumstances where compiler options or pragma statements are needed to alter default object layout rules. When you are building heterogeneous applications, this is usually true because each compiler has its own layout rules that might be incompatible in some way with the other compilers to be used. For nonheterogeneous applications, all that is required is that you specify any such compiler command options when invoking **ossg**. For heterogeneous applications the problem is more complicated, because **ossg** needs to know about all options or pragma statements that are used on any of the platforms in the heterogeneity set. You must provide this information to **ossg** by using a schema options file.

In the options file, you maintain a list of any compiler options and pragma statements that alter default object layout required on all platforms in the application's heterogeneity set. The schema generator uses the compiler option file when determining what changes are necessary for schema neutralization. You tell **ossg** what schema options file to use by the **-schema_options** *options_file* command-line argument.

See Compiler Option Files for Architecture Sets on page 161 for information about the contents of basic compiler option files for specific platform architectures. These describe what you should use as an absolute minimum for each architecture set.

If there are compiler options that you use on the current platform, specify them on the **ossg** command line as well as in the schema options file.

Compiler Option File Format

The compiler options file has the following structure:

- Each line contains one option.
- Comments and blank lines are allowed. A number sign (#) signals a comment and must be the first nonblank character in a comment line.
- An option has the following form:

{ *compiler_spec* | *architecture_spec* } [(*class_list*)] *option*

The *compiler_spec* variable indicates the compiler with which you are using the specified option.

The *architecture_spec* indicates both the compiler and the platform that you are using. The possible values are in the following table.

Architecture	<i>architecture_spec</i>
Digital UNIX DEC C++	axp_unix_dec
IBM VisualAge C++ for OS/2	intel_os2_visualage
Intel Windows NT Visual C++	intel_win32_msoft or
Intel Windows 95 Visual C++	visualc++
RS/6000 AIX C Set ++	rs6000_visualage

The *class_list* variable lists one or more classes that the specified compiler option or pragma operates on. Classes in the list can be all application classes included in the schema. If you do not specify a class, the option applies to all possible classes.

Enclose the class list in parentheses and insert a space between two class names.

The *option* variable can be a compiler switch or a pragma statement.

Compiler switches

When the *option* variable is a compiler switch, it has one of the following forms:

```
switch compiler_switch
switch compiler_switch compiler_switch_value
switch compiler_switch '=' compiler_switch_value
```

The compiler switches you can specify appear in the following table. See your compiler documentation for an explanation of each switch. These switches apply to an entire compilation rather than to a specific class.

Listing Nondefault Object Layout Compiler Options

<i>compiler_spec</i> or <i>architecture_spec</i>	<i>compiler_switch</i>	<i>compiler_switch_value</i>
axp_unix_dec	-nomember_alignment -vptr_size_short -xtaso_short -Zpn	Not applicable
intel_os2_visualage	/Sp /Sp1 (default) /Sp2 /Sp4 /Sp+ /Sp- /Su+ (default) /Su- /Su1 /Su2 /Su4	Not applicable
rs6000_visualage	-qalign= -qenum=	full packed power int (default) small
visualc++ or intel_win32_msoft	/vmb /vmg /vms /vmm /vmv /Zpn	Not applicable

Pragma statements

When the *option* variable is a pragma statement, it has the following form:

pragma *directive directive_value*

The pragma statements you can specify appear in the following table. See your compiler documentation for an explanation of these pragma statements.

<i>compiler_spec or architecture_spec</i>	<i>directive</i>	<i>directive_value</i>
axp_unix_dec	member_alignment	Not applicable
	nomember_alignment	Not applicable
	pack	Not applicable
	pointer_size	long
		short
		32
		64
	required_pointer_size	long
		short
		32
		64
		required_vptr_size
		short
		32
		64
intel_os2_visualage	pack	<i>(n)</i>
rs6000_visualage	options	ldbl128
	options align=	power
		full
		packed
	options enum=	smallest
		int
visualc++ or intel_win32_msoft	pack	<i>(n)</i>
	pointers_to_members	best_case
		full_generality,single_inheritance
		full_generality,multiple_inheritance
		full_generality,virtual_inheritance

Overriding Options Within the Compiler Option File

You can specify compiler options and pragmas at various levels. Specifications for a specific platform or compiler can override a specification that applies to all platforms or a set of platforms. Specifications for specific classes can override a specification for

all classes. The order of the options in the compiler option file is not significant. For example:

```
intel_os2_visualage switch /Sp  
intel_os2_visualage (classA classB classC) switch /Sp2
```

For all classes, **ossg** assumes that the compiler uses the **/Sp** switch. Except for **classA**, **classB**, and **classC**, **ossg** assumes the **/Sp2** switch.

Sample Compiler Option File

```
visualc++switch /Zpn  
intel_os2_visualageswitch /Sp4  
intel_win32_msoftpragma pointers_to_members best_case  
visualc++ (ClassX) pragma pack (1)  
rs6000_csetpragma options align=full
```

Compiler Option File Example

Suppose you want to use the following class in an application that runs on several platforms:

```
class A {  
public:  
    enum { X,Y,Z} id;  
};
```

The platforms you intend to use are OS/2, Windows NT, and Solaris 2. This class will not be neutral unless you specify the **/Su4** option on OS/2. (This option forces **enums** to be of size **int**.) You need a schema option file that contains the line

```
intel_os2_visualage option /Su4
```

In this example, the name of the schema option file is **schm.opt**. When you generate the schema on OS/2, specify the **/Su4** option. When you run **ossg** on the other platforms, specify **schm.opt**. Specifying the schema option file indicates to the schema generator exactly how objects are laid out on all platforms.

Compiler Options That Aid Neutralization

Certain platforms have default behavior in their compilers that causes problems in neutralization. The schema generator normally reports these problems as members of different sizes. In this situation, you might need to specify compiler options that force the compiler to yield a compatible object layout.

OS/2	<p>On OS/2 with the VisualAge C++ compiler, enum types default to the smallest size that can contain all values of the enumeration. Because of this, you must do one of the following:</p> <ul style="list-style-type: none"> • Pad the enumerations with large enumerator values. For example, enum Color { RED, BLUE, GREEN, BIG_COLOR=1<<30} • Specify the /Su4 compiler switch.
DEC AXP	<p>On DEC AXP systems that run Digital UNIX, pointers default to 64 bits. For this platform to operate heterogeneously with 32-bit platforms, you must build applications with compiler switches or pragmas that force the compiler to use 32-bit pointers at the appropriate time. To do this, use -xtaso or -xtaso_short on the compiler command line, as described in DEC C++ 64-Bit Pointer Considerations on page 91.</p>
Compiler option file	<p>When you use one of these compiler options, remember to include it in the compiler option file. In fact, if you specify -arch set1 for neutralization but you do not intend to use OS/2, you still need to specify compiler options that would be needed on OS/2.</p>

Compiler Option Files for Architecture Sets

This section provides basic compiler option files that are a minimum starting point for each of the architecture sets. Note that file contents are listed for **set1** through **set6**. In the event that you are using any set beyond **set6** (**set7** through **set11**), be aware that these sets are based on the first six. See *ossg Neutralization Options* on page 145 to clarify the relationship between the sets.

<i>Set</i>	<i>Compiler Option File Contents</i>
set1	# force enums to be 4 bytes on OS/2 intel_os2_visualage switch /Su4
set2	# force enums to be 4 bytes on OS/2 intel_os2_visualage switch /Su4
set3	# no options needed
set4	# force enums to be 4 bytes on OS/2 intel_os2_visualage switch /Su4

Set	Compiler Option File Contents
set5	# force enums to be 4 bytes on OS/2 intel_os2_visualage switch /Su4 # force pointers to be 4 bytes on AXP under DEC UNIX axp_unix_dec switch -xtaso_short axp_unix_dec switch -vptr_short
set6	# force enums to be 4 bytes on OS/2 intel_os2_visualage switch /Su4 # force pointers to be 4 bytes on AXP under DEC UNIX axp_unix_dec switch -xtaso_short axp_unix_dec switch -vptr_short

Description of Schema Generator Instructions

When run with neutralization options, the schema generator instructs you to insert ObjectStore padding macros and make other changes in your source files. The neutralization instructions are explicit. Be sure to use the exact macro name the schema generator provides.

This section provides a description of the macros that the schema generator instructs you to use. An explanation of changes to virtual base templates is also included.

Some macros have (x) at the end. The neutralizer provides instructions for replacing the x with a meaningful value.

Base Class Padding Macros

On platforms where the base class produces no padding, the schema generator cannot recognize that the padding macro exists. However, the schema generator can determine that, if the macro were present, the schema would be neutral. This allows the schema generator to both

- Produce the schema for the current platform successfully
- Warn you that additional padding is needed on other platforms

The message names the specific platforms. Add the padding and run `oss` again.

Macro	Inserts Padding to Compensate for
<code>os_base_pad_vftbl</code>	Four- or eight-byte vftbls placed at the start of an object
<code>os_base_pad_vftbl8</code>	
<code>os_base_pad_vbptr(x)</code>	Four- or eight-byte virtual base pointers placed at the start of an object
<code>os_base_pad_vbptr8(x)</code>	
<code>os_base_pad_vtbl</code>	Four-byte vtbls placed at the start of an object

Dynamically Defined Padding Macros

The schema generator does some dynamic naming of macros when inheritance precludes reusing a macro name. Dynamically named macros all begin with `_os_pad_`.

Member Padding Macros

Each member padding macro does one of the following:

- Defines a pad data member whose name is based on the macro name.
- Defines a nested class whose name is based on the macro name. Although no padding occurs on the neutralization platform, this allows the schema generator to recognize that the macro was used and determine how that macro would be expanded on other platforms.

The following table describes the member padding macros that ObjectStore provides.

Macro	Inserts Padding to Compensate For
<code>os_pad_vftbl_start</code>	Four- or eight-byte vtbls placed at the start of an object.
<code>os_pad_vftbl_start8</code>	
<code>os_pad_vftbl_end</code>	Four- or eight-byte vtbls placed at the end of an object.
<code>os_pad_vftbl_end8</code>	
<code>os_pad_vftbl_only8</code>	Eight-byte vtbls in classes containing no data members.
<code>os_pad_vtbl</code>	Four- or eight-byte virtual base tables placed at the end of an object.
<code>os_pad_vtbl8</code>	

Macro	Inserts Padding to Compensate For
os_pad_vbptr_start(x) os_pad_vbptr_start8(x)	Four- or eight-byte virtual base pointers placed at the start of an object. Takes an integer argument that must be unique for any use of the macro within the class.
os_pad_vbptr_end(x) os_pad_vbptr_end8(x)	Four- or eight-byte virtual base pointers placed at the end of an object. Takes an integer argument that must be unique for any use of the macro within the class.
os_pad_mem_ptr8(x) os_pad_mem_ptr12(x) os_pad_mem_ptr16(x)	Data member pointers that are 8, 12, or 16 bytes long.

Virtual Base Templates

Class layout varies so greatly among compilers that in many cases there is no way to achieve a compatible layout by padding alone. Often, complex inheritance paths that differ from platform to platform are needed.

Rather than require you to maintain multiple parallel class definitions and manually check that the correct neutralizations have been applied, the schema generator instructs you to use standard template base classes to introduce virtual bases. For example, suppose you begin with this class:

```
class A : public virtual B {}
```

The schema generator might instruct you to convert this to something like the line following. This varies depending on which platforms are involved.

```
class A : public os_vb_fbs<0,B> {}
```

Sometimes, instead of using a particular virtual base class, the schema generator instructs you to use a specific class whose name starts with **os_vb_**. This class takes two arguments. The first is an **int**, and you should insert the exact **int** that the schema generator provides. The second is the name of the virtual base class that you are replacing. Here are the class names:

- **os_vb_f<>**
- **os_vb_fs<>**
- **os_vb_fbs<>**

- `os_vb_fbsc<>`
- `os_vb_fbcstd<>`

This technique has the effect of virtually inheriting from class **B**, but hides all platform-dependent details needed to make the class neutral.

An important effect of this technique is that class **B** must have a public or protected default constructor. Since it is the responsibility of the most derived class to actually invoke the correct constructor, it should not be difficult to provide a public or protected default constructor.

The virtual base introduction templates have the prefix `os_vb_`. The schema generator *flattens* these template instantiations so that your class definitions are retained and the classes have compatible schema representations.

Database Growth Resulting from Padding

Database growth as a result of padding source files is hard to predict. Whether or not there is any growth depends on

- Kinds of data and structures in the schema
- Which platforms the application can run on

For example, the schema databases supplied with ObjectStore hardly grow. Applications that include virtual base classes and applications that do not include the `os_virtual_behavior` base class will grow more.

One way to try to determine any effect on database size would be to neutralize the schema and then run the `osexschm` utility on the preneutralized schema and postneutralized schema. Compare class sizes weighted by their relative frequency of occurrence in the database. You can also make a test run of the neutralized application and compare the database size to an equivalent preneutralized database.

Endian Types for ObjectStore Platforms

Endian type specifies whether the high-order byte is first or the low-order byte is first. This information is provided as general information. You do not need to be concerned with endian types when you neutralize schemas.

When you use the schema generator to neutralize for a group of platforms, the schema generator takes into account all platforms you want to use and the endian type for each platform. When the schema generator does this, it helps determine how to lay out objects in a way that can be used by all machines involved.

The following table shows the endian type for ObjectStore platforms. In a big-endian type, the high-order bit is first.

<i>Big-Endians</i>	<i>Little-Endians</i>
HP	DEC AXP
RS/6000	Intel
SGI	
Sun SPARC	

Chapter 6

Working with ObjectStore/Single

ObjectStore/Single is a form of the ObjectStore client tailored for single-user, nonnetworked use. The functional capability of an ObjectStore/Single application operating on file databases is virtually identical to that of a full ObjectStore client. Databases created with one kind of client are completely compatible with the other. However, full ObjectStore and ObjectStore/Single are not intended to run together.

As a stand-alone version of ObjectStore, ObjectStore/Single includes the Server and Cache Manager functionality as part of the same library as the ObjectStore client, rather than as separate processes. Also, ObjectStore/Single does not support rawfs file systems.

By using dynamic library load paths, you can decide at execution time whether an application should be a full ObjectStore or an ObjectStore/Single application. This allows you to develop applications using full ObjectStore, but package the application using ObjectStore/Single as a replacement. This replacement eases integration of embedded applications.

ObjectStore/Single Features

Each invocation of an ObjectStore/Single application requires a Server log file. Applications for UNIX platforms also require a cache file. Users must be prepared to specify the cache and Server log files at each execution of an ObjectStore/Single application.

ObjectStore/Single API

The following functions in the class **objectstore** can be used in creating ObjectStore/Single applications:

- [objectstore::embedded_server_available\(\)](#)
- [objectstore::get_cache_file\(\)](#) — UNIX only
- [objectstore::get_log_file\(\)](#)
- [objectstore::network_servers_available\(\)](#)
- [objectstore::propagate_log\(\)](#)
- [objectstore::set_cache_file\(\)](#) — UNIX only
- [objectstore::set_log_file\(\)](#)
- [objectstore::shutdown\(\)](#)

ObjectStore/Single Utilities

The following ObjectStore utilities can be used with ObjectStore/Single. *ObjectStore Management* describes each utility in detail.

- [oschangedbref](#): Changing External Database References
- [oschgrp](#): Changing Database Group Names
- [oschmod](#): Changing Database Permissions
- [oschown](#): Changing Database Owners
- [oscompact](#): Compacting Databases
- [oscp](#): Copying Databases
- [osexschm](#): Displaying Class Names in a Schema
- [oshostof](#): Displaying Database Host Name
- [osls](#): Displaying Directory Content
- [osmkdir](#): Creating a Rawfs Directory
- [osmv](#): Moving Directories and Databases

- [osprmgc](#): Trimming Persistent Relocation Maps
- [osprop](#): Propagating Server Logs
- [osrm](#): Removing Databases and Rawfs Links
- [osrmdir](#): Removing a Rawfs Directory
- [osscheq](#): Comparing Schemas
- [ossetasp](#): Patching Executable with Application Schema Pathname
- [ossetrsp](#): Setting a Remote Schema Pathname
- [ossevol](#): Evolving Schemas
- [ossg](#): Generating Schemas
- [ossize](#): Displaying Database Size
- [osupgrm](#): Upgrading PRM Formats
- [osverifydb](#): Verifying Pointers and References in a Database
- [osversion](#): Displaying the ObjectStore Version in Use

Dynamic Library Load Path

The tools run as full ObjectStore or ObjectStore/Single applications, depending on the kind of client and database utilities libraries found when the user's dynamic library load path is resolved at execution time.

For ObjectStore/Single, specify the following:

Platform	Library Load Path
UNIX	<code>\$OS_ROOTDIR/libsngl:\$OS_ROOTDIR/lib</code>
Windows	<code>%OS_ROOTDIR%binsngl</code>
OS/2	<code>%OS_ROOTDIR%libsngl</code>

Application Development Sequence

You should first develop your applications and schema using distributed ObjectStore, and then run your applications with the ObjectStore/Single libraries for final testing and delivery.

You can use two environment variables for proof of concept when you want to convert a full ObjectStore application to an ObjectStore/Single application. These environment variables are strictly for development purposes and are not supported if included in your final application. See Additional Considerations on page 175 for a discussion of the issues involved in using these environment variables.

The two environment variables are

- **OS_CACHE_FILE**
- **OS_LOG_FILE**

OS_CACHE_FILE

The **OS_CACHE_FILE** environment variable can be used early in development instead of the entry point `objectstore::set_cache_file` to specify the cache file for an ObjectStore/Single application execution. The entry point takes precedence over the environment variable. Use of the environment variable in the final application is not supported, and is potentially troublesome to your customers.

OS_LOG_FILE

The **OS_LOG_FILE** environment variable can be used early in development instead of the new entry point `objectstore::set_log_file` to specify the Server log file for an ObjectStore/Single application execution. The entry point takes precedence over the environment variable. Use of the environment variable in the final application is not supported, and is potentially troublesome to your customers.

The user must take responsibility for ensuring that the Server log information is propagated when moving a database from a full ObjectStore environment to ObjectStore/Single, or the reverse.

Server Log Propagation

This section discusses log propagation in both ObjectStore / Single and full ObjectStore.

ObjectStore / Single provides transaction consistency for databases just as full ObjectStore does. Both ObjectStore / Single and full ObjectStore use a *Server log* as a tool to provide transaction consistency. See [Description of the Server Transaction Log](#) in *ObjectStore Management* for additional information about Server logs.

In this discussion, the term *Server* refers to both the separate **osserver** process of full ObjectStore and the functionally equivalent part of ObjectStore / Single that is contained in the client library. The term is meaningful for ObjectStore / Single because, while ObjectStore / Single does not have or need a separate-process Server, the requisite work is done in a part of the client library internals known as the *embedded Server*.

Server Log Functions

You must be aware of several important aspects of Server logs.

Commit compared to propagation

A log might contain data that has been committed but not yet written to one or more databases. Logically, the log information is part of the databases, and so it must be propagated, that is, written into the actual databases, before those databases can be used for more work. Users of full ObjectStore might have encountered this issue when trying to copy file databases with an operating system command such as **cp** rather than **oscp**.

Databases are marked internally when there is unpropagated data in the log file, so that if an ObjectStore application tries to use that database with a log different from the one holding the not-yet-propagated data, an error is reported. The significance of this information is that you must be careful not to delete a log that has unpropagated information in it.

Server log size

Server logs can grow large fairly quickly. As a rule of thumb, a log's size is proportional to the amount of data that has been modified in the largest transaction since that log was created. The Server cannot shrink its log. The log size is not generally a problem for full ObjectStore since there is only one log file per

osserver process. However, in ObjectStore/Single, because each execution of an ObjectStore/Single process must have its own log, the log size can quickly become a disk space management problem. Thus you must be conscientious about finding and propagating (and thereby removing) old log files.

Log File Guidelines

The following ObjectStore behaviors help you deal with logs:

- ObjectStore/Single always forces the log to be propagated immediately when a transaction commits. The result of this is that the period of time when there is committed data in the log is usually quite short.
- During start-up, both ObjectStore/Single and full ObjectStore's **osserver** process propagate data in the log before doing any other work.
- During shutdown (either during a normal process exit or when **objectstore::shutdown** is called), ObjectStore/Single attempts to propagate any remaining data in the log and remove the log. Full ObjectStore's **osserver** process similarly propagates the log but does not delete it when **ossvrshd** is executed.
- The utility **osprop** propagates data from the log files named in the command invocation. Following successful propagation, the log files are automatically removed.
- The other utilities that provide meaningful information as ObjectStore/Single tools (**oscp**, **ossiz**, and **osverifydb**) all accept a log file explicitly named on the command line. If a log file is specified, and is writable, propagation of the log occurs automatically before the indicated operation is carried out.

When to Intervene

An ObjectStore/Single application (or any of the ObjectStore/Single utilities) will try to ensure that all the data in its Server log is propagated and the log removed before it exits.

Important note

The absence of a log after the program ends is both normal and a guarantee that all committed data is physically in the affected databases.

Conversely, the presence of a log after the program ends is an indication that the databases should be considered to be in an

inconsistent state. When this happens, run the **osprop** utility on that Server log immediately.

Also bear in mind that any time an ObjectStore/Single application or utility is initialized with an existing Server log, ObjectStore automatically conducts propagation. The **osprop** utility is the simplest possible ObjectStore/Single application — it starts, does propagation, and then shuts down.

Cautions to observe

Note the following considerations:

- It is the responsibility of the application to keep track of Server logs. Databases do not contain any record of where an associated log is located.
- Databases that have not been propagated should not be moved.
- When the Server does log propagation, it silently discards data associated with databases not found.

Full ObjectStore osserver Role

The most reliable way to ensure that data has been propagated into a database last used by an **osserver** process is to shut down the Server using **ossvrshd**. As a practical matter, though, the **osserver** process normally propagates data into a database when no client process has the database open.

Other than by shutting down the **osserver** process, it is not easy for users to know if log information has been completely propagated into a database.



However, the Server will hold a database open with a **write** file lock if unpropagated data lies in the log, so UNIX system tools such as **ofiles** or **lsdf** can be used to infer the Server log state.

Remote Access

ObjectStore/Single allows remote access to databases through NFS. The pathname lookup algorithm differs from full ObjectStore in that it does not consider remote mount points when expanding paths, so all paths appear to be local.

The different pathname lookup behavior between ObjectStore/Single and full ObjectStore could be a concern for those who

- Use cross-database pointers and cross-database references
- Operate on databases via NFS mounts
- Want to interoperate between full ObjectStore and ObjectStore/Single

Accessing Server Logs and Cache Files Through NFS

It is acceptable, though not recommended, for an ObjectStore/Single application to use a Server log that is reached through NFS.



Attempting to use a cache file through NFS is not supported and might generate run-time errors (for example, a failure in `mmap`), depending on the host platform.

Packaging ObjectStore/Single Applications

This section provides a checklist of the requirements for packaging ObjectStore/Single applications for UNIX and Windows platforms.

Cache and Server Log Files

ObjectStore/Single makes the application completely responsible for identification of each program's cache and Server log file. This burden is not trivial. Every instance of a program run must have a unique cache and Server log. (Note that cache files are required for UNIX applications only.) You must be sure that

- Old cache files do not accumulate.
- Server logs with unpropagated data (due to application crashes) are propagated and not inadvertently removed.

Additional Considerations

The ObjectStore/Single environment variables (**OS_LOG_FILE** and **OS_CACHE_FILE**) should only be used early in development when you are prepared for things to go wrong. They are not safe for production, and they can create interoperability problems. Nonetheless, they can be useful to you very early in the process of moving a project from full ObjectStore to ObjectStore/Single.

Picking cache and log file names for an application should not be done statically. The names chosen must be in the context of whatever other ObjectStore/Single applications happen to be running at the same time on the host. Name collisions must be avoided. All of these concerns are the application's responsibility.

This issue is more complicated than the cache manager's job in managing the commseg and cache file pools for networked ObjectStore, since there are the additional factors of user permissions and Server log file persistence. To summarize, your application should address the following issues when selecting a cache and log file.

Cache File Considerations



- Specifying a location with sufficient disk space
- Picking a unique name (for example, **tmpnam** or **tempnam**)
- Deciding whether to immediately delete cache files after a program run or create a pool so that they can be reused

In case of an application crash, or if cache files are being pooled, you need to specify a mechanism for cleanup (the work done by **oscmrf** for full ObjectStore).

Server Log File Considerations

- Specifying a location with sufficient disk space
- Deciding where to put a backup of the log file
- Determining how the log file name should be communicated to an external agent in case recovery is needed
- Specifying how to manage recovery when the log file remains after the application exits
- Devising a mechanism to safeguard the log file until recovery is accomplished

What Should You Tell Your Customers?

What is enough disk space and what is going to be backed up are the points that need to be communicated clearly to the application. The end-user site must determine where there is enough disk space and what should be backed up. If you do not consider the other items, at a minimum, your application should be installed in a space that allows all cache and log files to be generated in a subdirectory of the product installation or in another directory on the same disk.

Be sure to communicate this to your customers prior to your application installation.

Contact Object Design if you have recommendations about how the current cache file and Server log API could be enhanced to ease this requirement.

Packaging an ObjectStore/Single Application



If you choose to package the ObjectStore/Single components with your own libraries, you must include the ObjectStore/Single libraries from **libsngl** (both **libos** and **libosdbu**), plus whatever additional libraries from **lib** are useful to your application (such as **liboscol**, **libosmop**, and so on). Do not include **lib/libos** or **lib/libosdbu**.

Also include the ObjectStore utilities.



On Windows NT, all the ObjectStore/Single components are installed in the **\binsngl** directory. Choose which DLLs and ObjectStore utilities your application requires and add them to your packaging procedure.

Packaging with a VAR Product

When ObjectStore/Single is packaged with a VAR's product, the only other installation issues are related to the items presented in the discussion in Additional Considerations on page 175.

Packaging ObjectStore/Single Applications

Index

A

AIX

- architecture code 157
- virtual function tables 82

ANSI exceptions 102

application schema databases

- description 2
- option to **ossg** 46
- specification in makefiles 106

application schema object files

- description 3
- option to **ossg** 45

application schema source files

- description 2
- naming convention 65
- option to **ossg** 45

application schemas

- C preprocessor, changing default 46
- cd** option on OS/2 47
- comparison with other schemas 66
- database schemas, compatibility
 - with 33
- description 30
- generating 35
- generating from a compilation
 - schema 56
- library schemas, omitting 49
- moving 75

- multiple applications, using same for 51

ossg command-line examples 51

ossg, invoking 36

specifying library schemas 48

applications

- debugging 85
- heterogeneous 134
- moving 75
- third-party 6
- Windows

- porting to 123

-arch setn option to **ossg** 37, 145

architectures

- code specifications 157
- neutralization, specifying for 37, 145
- starter compiler options files 161

-asdb option to **ossg** 46

-asof option to **ossg** 45

-assf option to **ossg** 45

B

breakpoints

- OS/2 130

- Windows application abnormal exit 117

building heterogeneous applications 155

building ObjectStore applications

- compiling 73

- debugging 73

C

- flow chart 4
- generating schemas 29
- linking 73
- neutralizing schemas 148
- overview 2
- Server involvement 9

C

- C compiler
 - schema source file contents 25
- C run-time library
 - UNIX, order for specifying libraries 88
- cd** option to **oss**g 47
- cfront
 - supported compilers 137
- changing default preprocessor 46
- cl** preprocessor 47
- clients
 - virtual file systems 7
- coll.hh** header file
 - when to use 15
 - with **os_Collection_declare** macro 17
- collections
 - coll.hh** header file 15
 - libraries and library schemas needed 48
 - restricting use of 59
 - template class, undefined error report 16
 - template instantiation problem 16
 - UNIX, order for specifying libraries 88
 - Windows makefile 115
- compact.hh** header file
 - when to use 15
- compactator
 - compact.hh** header file 15
 - libraries and library schemas needed 48
 - UNIX, order for specifying libraries 88
- compatibility
 - application and database schemas 33
- compilation options
 - HP 102
 - neutralizing with nondefault

- layouts 156
- OS/2 127
- Sun C++ 103
- to **oss**g
 - application schemas 37
 - Windows 111
- compilation schemas
 - comparison with other schemas 66
 - description 31
 - generating 55
- compiler options files
 - description 156
 - example 160
 - format 156
 - neutralization 160
 - overriding options 159
 - pragmas 158
 - starter file for each architecture set 161
 - switches 157
- compilers
 - heterogeneous 137
 - icc** public symbols 47
 - third-party 5
- cpp** preprocessor 47
- cpp_fixup** option to **oss**g
 - application schemas 38
- crash recovery 11
- creating schema source files
 - examples 24
 - instructions 22

D

- data formats 138
- database schemas
 - comparison with other schemas 66
 - compatibility with application schemas 33
 - description 32
- database utilities
 - dbutil.hh** header file 15
 - UNIX, order for specifying libraries 88

databases
See also protected databases

dbutil.hh header file
 when to use 15

DDE4MBSI.LIB library 128

ddebug build 125

debug build 124

debugging
 OS/2 130
 overview 85
 Windows 116

DEC C++
 pointer restrictions 141

defaults
 C preprocessor 46

delta format objects 109

dependencies
 object files on header files 86

Digital UNIX
 architecture code 157

discriminant functions
 when building applications 77
 example of making them available 50
 making them available 44
 notification about missing ones 43
 UNIX, shared libraries 108

DLLs
 compiling 112
 debug versions
 OS/2 130
 Windows 116
 linking 113
 load path
 ObjectStore/Single 167
 using 11
 using ObjectStore from within 121

dynamic libraries
See DLLs

E

endian types 166

environment variables
OS_CACHE_FILE
 ObjectStore/Single 170
OS_COMP_SCHEMA_CHANGE_ACTION
 example 61
OS_LOG_FILE
 ObjectStore/Single 170
OS_TRACE_MISSING_VTBLS
 debugging applications 85
PMDEXCEPT
 OS/2 130
 Windows 115

exception handling
 macros for 107
 Windows compiler options 111
 Windows macros 123

F

fault handlers
 UNIX macros 107
 Windows 123

@filename option to **ossg** 47

-final_asdb option to **ossg** 39

G

/Gd compiler option on OS/2 127

generating schemas
 application 35
 compilation 55
 library 53
 overview 30
 Server involvement 9
 vtbls and discriminant functions, making
 available 44
 vtbls and discriminants, example of
 making available 50

get_os_typespec()
 missing functions message 49

H

/Gm compiler option on OS/2 127

H

header files

coll.hh 24

 required order 18

 when to use 15

compact.hh

 when to use 15

dbutil.hh 24

 when to use 15

dependencies 86

manschem.hh 24

 creating schema source files 22

mop.hh

 required order 18

 when to use 15

ObjectStore 15

OS/2 class definition file 25

ostore.hh 24

 when to use 15

ostore/manschem.hh

 when to use 15

relat.hh

 when to use 15

required order 18

Rogue Wave on Solaris 69

sample **#include** statements 24

schmdefs.hh 24

schmevol.hh

 required order 18

 when to use 15

semoptwk.hh

 when to use 18

user-defined 14

heterogeneity 136

heterogeneous applications

 allowable platforms 137

 building 155

 description 134

 description of schema generator

 instructions 162

 endian types 166

 mixing 32-bit and 64-bit platforms 137

 neutralization options 145

 neutralizing schema

 instructions 148

 nondefault object layout options 156

 restrictions 140

HP CC

+eh mode 102

-hpfb option to **ossg** 24

HP-UX

 linker options 102

 missing vtbls 102

I

-I option to **ossg**

 application schemas 42

icc preprocessor 46

IDE for Visual C++

 makefiles 110

INCLUDE environment variable on

 Windows 116

include paths

-no_default_includes option to **ossg** 42

 specifying to **ossg** 37

indexes

 UNIX, order for specifying libraries 88

inline virtual functions

 missing vtbls 81

Integrated Development Environment

 See IDE for Visual C++

L

LIB environment variable on Windows 116

libos library order 88

liboscmp library 88

liboscmp.ldb library schema

 when to use 48

liboscol library order 88

- liboscol.ldb** library schema
 - UNIX example 106
 - when to use 48
 - libosdbu** library order 88
 - libosmop** library order 88
 - libosqry** library order 88
 - libosqry.ldb** library schema
 - UNIX example 106
 - when to use 48
 - libosse** library order 88
 - libosse.ldb** library schema
 - when to use 48
 - libostcl** library order 88
 - libosthr** library order 88
 - libosths** library order 88
 - libraries 88
 - OS/2 128
 - third-party 6
 - UNIX
 - specifying 88
 - Windows
 - requirement 110
 - standard run-time 111
 - using ObjectStore within DLL 121
 - library schemas
 - comparison with other schemas 66
 - description 31
 - generating 53
 - ObjectStore-provided 48
 - omitting one, example 49
 - remote 48
 - specifying to generate application schema 48
 - specifying when invoking **oss**g 46
 - when to create one 54
 - link order 88
 - linking
 - OS/2 128
 - UNIX 88
 - Windows 113
 - log files, ObjectStore/Single
 - forcing data propagation for applications 172
 - issues 176
 - size of Server log file 171
 - long** data type 140
 - long double** data type 140
 - long long** type support 58
 - loscmp** library
 - when to use 48
 - loscol** library
 - when to use 48
 - losqry** library
 - when to use 48
 - losse** library
 - when to use 48
- ## M
- macro arguments
 - specifying to **oss**g 37
 - macro, system-supplied
 - base class padding 162
 - member padding 163
 - __NO_TEMPLATES__** preprocessor macro 59
 - _ODI_OSSG_** 57
 - os_Collection_declare** 16
 - os_Collection_declare_no_class** 17
 - os_Collection_declare_ptr_tdef** 17
 - OS_END_FAULT_HANDLER** 107
 - OS_ESTABLISH_FAULT_HANDLER** 107
 - OS_MARK_SCHEMA_TYPE()** 23
 - OS_MARK_SCHEMA_TYPESPEC()** 23
 - OS_NO_COLLECTION_TEMPLATES** preprocessor macro 59
 - OS_POSTLINK** 105
 - make_reachable_library_classes_** persistent option to **oss**g 40

N

- make_reachable_source_classes_persistent** option to **oss**
 - application schemas 41
 - whether to mark 20
- makefiles
 - compilation schema generation 55
 - dependency of object files on header files 86
 - library schema generation 53
 - neutralizing schemas 154
 - OS/2 example 128
 - retrofitting 87
 - tabs and spaces 106
 - UNIX
 - building compilation schemas 107
 - description 105
 - template 106
 - Windows example 114
 - Windows IDE 110
- manschem.hh** header file
 - creating schema source files 22
- marking types
 - how to 23
 - reason for 19
- MD** option to Visual C++ 111
- memory access violations
 - Windows debugger 116
- metaobject protocol
 - mop.hh** header file 15
 - UNIX, specifying libraries 88
- MOP
 - See metaobject protocol
- mop.hh** header file
 - when to use 15
- moving application schemas 75
- mrlcp** option to **oss** 40
- mrscp** option to **oss**
 - application schemas 41
 - comparison with marking types 20
- msvcrt.lib** run-time library 111
- mt** compiler option 103

- multithreaded applications
 - Solaris, debugging 104
 - Windows 124

N

- neutral_info_output** option to **oss** 42, 146
- neutralizing schemas
 - class instantiations 141
 - compiler options needed 160
 - considerations 149
 - definition 134
 - endian types 166
 - examples 151
 - how **oss** creates identical formats 138
 - instructions 148
 - makefiles 154
 - nondefault object layout options 156
 - options 145
 - restrictions 140
 - schema generator instructions,
 - description 162
 - virtual base templates 164
- NFS mounts
 - building applications on OS/2 122, 131
- no_default_includes** option to **oss**
 - application schemas 42
- __NO_TEMPLATES__** preprocessor macro 59
- no_weak_symbols** option to **oss** 43
- nor** option to **oss** 146
- noreorg** option to **oss** 146
- nout** option to **oss** 146

O

- objects
 - delta format 109
- ObjectStore libraries 11
- ObjectStore stand-alone
 - See ObjectStore / Single

- ObjectStore/Single
 - API 168
 - application packaging 177
 - cache files 175
 - definition 167
 - embedded Server 171
 - end-user information 176
 - interoperability with full
 - ObjectStore 174
 - libraries 11
 - log files 172, 175
 - remote database access 174
 - use of NFS with 174
 - using environment variables 175
 - utilities 168
- objectstore**, the class
 - functions that can be used with
 - ObjectStore/Single 168
- _ODI_OSSG_**, the macro 57
- OS/2
 - building applications remote from
 - Server 122, 131
 - cd** option to **oss**g 47
 - compiler options 127
 - for neutralization 161
 - debugging 130
 - library schemas 48
 - linking 128
 - makefile example 128
 - schema header file 25
 - /Su4** compiler option 127
- os_Array**, the class
 - undefined error 16
- os_Bag**, the class
 - undefined error 16
- os_base_pad_xxx** macros 163
- OS_CACHE_FILE** environment variable
 - use in developing ObjectStore/stSingle
 - applications 170
 - use with **objectstore::set_cache_file** 170
- os_coll.ldb** library schema
 - OS/2 makefile example 130
 - when to use 48
- os_Collection**, the class
 - undefined error 16
- os_Collection_declare**, the macro
 - description 16
 - example 17
- os_Collection_declare_no_class**, the macro 17
- os_Collection_declare_ptr_tdef**, the macro 17
- OS_COMP_SCHEMA_CHANGE_ACTION**
 - environment variable
 - example 61
- os_Cursor**, the class
 - undefined error 16
- OS_END_FAULT_HANDLER**, the macro
 - description 107
 - Windows, example 123
- OS_ESTABLISH_FAULT_HANDLER**, the macro
 - description 107
 - Windows, example 123
- os_List**, the class
 - undefined error 16
- OS_LOG_FILE** environment variable 170
 - use with **objectstore::set_log_file** 170
- OS_MARK_SCHEMA_TYPE()**, the macro
 - creating schema source files 23
 - examples 24
- OS_MARK_SCHEMA_TYPESPEC()**, the macro
 - creating schema source files 23
- OS_NO_COLLECTION_TEMPLATES**
 - preprocessor macro 59
- OS_OSSG_CPP** environment variable
 - C preprocessor, changing C
 - preprocessor 46
- os_pad_xxx_xxx** macros 163

P

- os_postlink** executable
 - description 105
 - vtbl relocation 78
- OS_POSTLINK**, the macro
 - description 105
 - example 106
- OS_ROOTDIR** environment variable
 - Windows 115
- %OS_ROOTDIR%include** directory 110
- os_Set**, the class
 - undefined error 16
- OS_TRACE_MISSING_VTBLS** environment variable
 - debugging applications 85
- os_vb_** virtual base introduction
 - templates 164
- oscmpct.ldb** library schema
 - when to use 48
- osmakedep** command 86
- osquery.ldb** library schema
 - OS/2 makefile example 130
 - when to use 48
- ossetasp** utility 75
- ossevol.ldb** library schema
 - when to use 48
- ossg** utility
 - @filename** option 47
 - comparison of command lines 65
 - generating application schemas 36
 - generating compilation schemas 56
 - nondefault object layout options 156
 - options 36
 - platforms in architecture sets 37, 145
 - temporary file, using to send arguments 47
 - UNIX, building compilation schemas 107
- ostore.hh** header file
 - when to use 15

- ostore.lib** library
 - how to link with 113
 - OS/2, linking 128
 - when to use 48
 - Windows, linking 110
- ostore/manschem.hh** header file 15
- overloaded operators
 - cfront compilers 84

P

- pad_consistent** option to **ossg** 43, 146
- pad_maximal** option to **ossg** 43, 146
- padc** option to **ossg** 146
- padding macros
 - base class 162
 - database growth 165
 - dynamically defined 163
 - members 163
- padm** option to **ossg** 146
- parameterized classes
 - instantiation problem 16
 - neutralizing schemas 141
 - restricting use of 59
- parse_function_bodies** option to **ossg** 24
- PATH** environment variable on
 - Windows 115
- pathnames
 - when using virtual file systems 7
- persistence
 - allowing allocation of reachable types
 - application schemas 41
- platforms
 - architecture sets 37, 145
 - supporting heterogeneous applications 137
- PMDEXCEPT** OS/2 environment variable 130
- pointers
 - DEC C++ restrictions 141
- POSIX thread environment 107

preprocessor macros
`__NO_TEMPLATES__` 59
`OS_NO_COLLECTION_TEMPLATES` 59

preprocessors
 allowing spaces in C++ tokens
 application schemas 38
 default on each platform 46
 options, specifying 37
 propagation
 ObjectStore/Single log files 171
 protected databases
 generating schemas for 68

Q

queries
 libraries and library schemas needed 48
 UNIX, order for specifying libraries 88
 Windows makefile 115

R

reachable types
 description 19
 specifying `-mrlcp` 40
 specifying `-mrscp`
 application schemas 41
`relat.hh` header file
 when to use 15
 relationships
`relat.hh` header file 15
 UNIX, order for specifying libraries 88
 relocating vtbls 78
 remote library schemas 48
 remote Servers
 building applications on OS/2 122, 131
 Rogue Wave header file problem 69
`-rtdp` option to `oss`
 how to use 44
 example 50

`-runtime_dispatch` option to `oss`
 how to use 44
 example 50

S

schema databases
 description 30
 virtual file systems 7
 schema evolution
 libraries and library schemas needed 48
`ostore/manschem.hh` header file 15
`schmevol.hh` header file 15
 UNIX, order for specifying libraries 88
 schema generation
 See generating schemas
 schema generator
 See `oss` utility
 schema source files
 creating 22
 description 19
 marking types 19
 specifying for `oss`
 application schemas 46
 using more than one 53
`-schema_options` option to `oss`
 description 44, 147
 starter file for neutralization 161
 how to use 156
 schemas
 See also application schemas
 See also compilation schemas
 See also database schemas
 See also generating schemas
 See also library schemas
 comparison of kinds 66
 database schema, updating 150
 hiding code from `oss` 57
 introduction 30
 metaschema mismatch errors 61
 moving 75
 neutralization definition 134

S

- neutralization examples 150
- neutralization instructions 148
- OS/2 class definition header file 25
- oss**g run-time errors 61
- persistent allocation errors 60
- protected databases 68
- type mismatch errors 60
- types, determining 19
- vtbls, missing 80
- semoptwk.hh** header file 18
- Server logs
 - See log files, ObjectStore/Single
- Servers
 - build process involvement 9
 - building applications remotely on OS/2 122, 131
 - embedded Server for ObjectStore/Single 171
 - virtual file systems 7
- setting breakpoints on Windows 117
- sfbp** option to **oss**g 24
- shared libraries
 - UNIX
 - vtbl and discriminant function symbols 108
- shared library load path 10
- show_difference** option to **oss**g 44, 147
- show_whole** option to **oss**g 44, 147
- showd** option to **oss**g 147
- showw** option to **oss**g 147
- SIGBUS** signals
 - fault handlers 107
 - modifying state 106
- signal handlers
 - UNIX 106
- SIGSEGV** signals
 - handlers 107
 - modifying state 106
- 64-bit platforms
 - general restriction 137
 - pointer restriction 140
- skip_function_body_parsing** option to **oss**g 24
- Solaris
 - compiler options 103
 - default preprocessor 109
 - Rogue Wave header file problem 69
- sopt** option to **oss**g
 - description 147
 - how to use 156
- source files
 - contents 14
 - ObjectStore header files 15
 - schema source files 22
- stack traces
 - using debug DLLs
 - OS/2 130
 - Windows 116
 - Windows
 - abnormal application exit 117
- standard template library support 74
- STL class libraries
 - ObjectSpace 74
 - Rogue Wave 74
 - Visual C++ 74
- struct**
 - and schema source file 25
- /Su4** compiler option on OS/2
 - compatibility with previous versions 127
- Sun C++
 - compilation options 103

T

- template specializations 142
- templates
 - collection classes instantiation
 - problem 16
 - nonvirtual instantiations 154
 - restricting use of 59
 - virtual base templates 164
 - when to modify 142
- third-party applications, compilers, and libraries 5
- thread environment, POSIX 107
- threads
 - UNIX, libraries 88
 - Visual C++ 124
- troubleshooting
 - HP-UX
 - missing vtbls 102
 - OS/2 general protection fault 130
 - OS/2 link failure 131
 - Rogue Wave header file problem 69
 - Solaris, syntax error on input 109
 - Windows
 - abnormal application exit 117
 - symbols, missing 79
- types
 - long double** 140
 - long long** 58
 - reachable 19
 - wchar_t** 58
 - which to mark 19
- types, how to mark 23

U

- union discriminant functions
 - See discriminant functions
- UNIX
 - fault handlers in POSIX thread
 - environment 107
 - libraries
 - specifying 105
 - library schema
 - specifying 105
 - library schemas 48
 - linking 88
 - makefile for building compilation
 - schema 107
 - makefile template 105
 - ObjectStore libraries 88
 - signal handlers 106
 - vtbl pointers 107
- utilities
 - virtual file systems 7

V

V

- vdex** compiler option 103
- VFTs on AIX 82
- virtual base classes
 - neutralization example 153
 - neutralization restrictions 140
- virtual base templates
 - neutralization issues 164
- virtual file systems 7
- virtual functions
 - missing vtbls 81
 - vtbls 77
- virtual table pointers
 - See vtbls
- Visual C++
 - application schema object file 135
 - double** data type 124
 - long double** data type 124
 - /NODEFAULTLIB** option 124
 - required options 111
 - STL restriction 74
 - threads 124
- VisualAge C++
 - architecture code 157
 - options for neutralization 161
- vtbls
 - AIX/CSET 82
 - description 77
 - HP-UX, missing 102
 - inline virtual functions 81
 - making them available 44
 - making them available, example 50
 - missing 80
 - notification about missing ones 43
 - os_postlink** executable 105
 - relocation 78
 - UNIX, shared libraries 108

W

- wchar_t** types 58
- Windows
 - abnormal application exit 117
 - applications, porting to 123
 - breakpoints, setting 117
 - debugging 116
 - include** directory 110
 - libraries, run-time 111
 - library schemas 48
 - library, required 110
 - linking 113
 - makefile, sample 114
 - memory access violations 123
 - ossg**, running within IDE 110
 - symbols, missing 79
 - using ObjectStore within DLL 121
- Windows NT
 - architecture code 157

Z

- Zn** options to Visual C++ 111