Building ObjectStore C++ Applications

Release 6.1

February 2003
# Contents

<table>
<thead>
<tr>
<th>Preface</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Chapter 1</strong></td>
<td><strong>Overview of Building an Application</strong></td>
</tr>
<tr>
<td>Basic Steps for Building Applications</td>
<td>1</td>
</tr>
<tr>
<td>Flow Chart for Building Applications</td>
<td>3</td>
</tr>
<tr>
<td>Third-Party Compilers</td>
<td>4</td>
</tr>
<tr>
<td>Third-Party Libraries and Applications</td>
<td>5</td>
</tr>
<tr>
<td>ClearCase Virtual File System (MVFS)</td>
<td>5</td>
</tr>
<tr>
<td>ObjectStore Server and the Build Process</td>
<td>5</td>
</tr>
<tr>
<td>ObjectStore/SINGLE</td>
<td>6</td>
</tr>
</tbody>
</table>

| **Chapter 2** | **Working with Source Files** | 9 |
| Overview of Source Files | 9 |
| ObjectStore Header Files | 10 |
| Instantiation Problem with Template Collection Classes | 10 |
| os_Collection_declare() Macro | 11 |
| os_Collection_declare_no_class() Macro | 11 |
| os_Collection_declare_ptr_tdef() Macro | 12 |
| Required Order of #include Statements | 12 |
| Determining the Types in a Schema | 13 |
| The Types to Mark | 14 |
| The Types That Are Reachable | 14 |
| If You Are Not Sure a Type Is Reachable | 14 |
| Choosing Whether to Mark Types or Specify -mrscp | 15 |
| Ensuring a Complete Schema Source File | 15 |
Creating Schema Source Files ........................................ 15
Schema Source File Format ........................................ 16
Schema Source File Examples ....................................... 17
Schema Source File Must Be Compilable .............................. 18
When You Modify a Source File ...................................... 18

Chapter 3  Generating Schemas ........................................ 21
Overview ...................................................................... 22
Application Schemas .................................................. 22
Component Schemas ................................................... 23
Library Schemas ........................................................ 23
Compilation Schemas .................................................. 23
Database Schemas ...................................................... 23
Keeping Database Schemas and Application Schemas Compatible . 25
Schemas Are Platform Specific ....................................... 25
Invoking ossg to Generate a Schema ............................... 26
Changing the Default Preprocessor ................................. 33
Using a Temporary File to Send Arguments to ossg .......... 33
Example of Using the -runtime_dispatch Option ............ 34
Generating an Application or Component Schema ........... 35
Invoking ossg to Generate an Application Schema ............ 36
Using the Same Application Schema for Multiple Applications .... 36
Examples of Generating an Application Schema ............. 37
Locating a Schema Database ........................................ 38
Generating a Library Schema ....................................... 38
Invoking ossg to Generate a Library Schema .................... 39
Using Multiple Schema Source Files to Create a Library Schema ...... 39
Generating a Compilation Schema .................................. 40
Invoking ossg to Generate a Compilation Schema ............ 40
Using a Compilation Schema to Generate an Application Schema . . . 41
Hiding Code from the Schema Generator .......................... 41
Using long long Data Type .......................................... 42
Schema Incompatibilities (Windows) ............................... 42
Differing Support for #pragma pack(pop,N) ...................... 42
## Contents

Differing Support for Integral Extension Types ........................................ 43
Correcting Schema-Related Errors........................................................... 44
  Type Mismatch Errors ........................................................................ 44
  Persistent Allocation Errors .............................................................. 44
  ossg Run-Time Errors ...................................................................... 45
  Metaschema Mismatch Errors ............................................................ 45
  Schema Neutralization Errors ............................................................ 45
  Missing Virtual Function Table Pointer Problems .......................... 45
Handling #pragma Statements in Source Code........................................ 46
Utilities for Working with Schemas ....................................................... 46
Comparison of ossg Command Lines ................................................... 47
Comparison of Kinds of Schemas .......................................................... 48
Deploying Products with Protected Databases ...................................... 49
Using Rogue Wave with Solaris Sun C++ .......................................... 50
ossg Troubleshooting Tips .................................................................. 51
  Technical Support ............................................................................ 52

### Chapter 4

**Compiling, Linking, and Debugging Programs** .................................... 53

**UNIX** ............................................................................................... 53
  Linking with ObjectStore Libraries .................................................. 54
  Examples of Passing Libraries to the Linker .................................... 55
  Compiling and Linking on Solaris 2 ................................................... 56
  Sun C++ Compiler Options ................................................................. 56
  Sample Makefile Template ................................................................. 56
  Using Signal Handlers ..................................................................... 57
  Makefile for Building from Compilation Schemas ........................... 58
  Virtual Function Table Pointers ....................................................... 58
  Debugging Applications ................................................................ 59

**Windows** ......................................................................................... 59
  Linking with ObjectStore Libraries ................................................ 59
  Using Custom Build to Run ossg ..................................................... 59
  Ensuring That You Include Required Files ...................................... 60
  Make and Compiler Options ............................................................. 60
  Using the Standard Run-Time Library on Windows NT .................. 61

**Release 6.1**
Contents

Linking Your Files ............................................... 62
Building Applications ........................................... 62
Specifying Environment Variables ............................... 63
Debugging Your Application .................................... 64
Building ObjectStore/Microsoft Foundation Class (MFC) Applications 65
Class os_CString .................................................. 67
Building Applications on Machines Remote from the Server ..... 67
Porting ObjectStore Applications to Windows Platforms ....... 68
Windows debug and symbols Builds of ObjectStore .......... 69
Installing symbol.zip ............................................. 69
Moving an Application and Its Schema ......................... 71
Moving an Application Schema .................................. 72
Working with Virtual Function Table Pointers .................. 72
  Relocation ...................................................... 73
Missing vtbls. ..................................................... 73
  Symbols Missing When Linking ObjectStore Applications .... 73
  Run-Time Errors from Missing vtbls ......................... 75
Run-Time Type Information .................................... 76
Debugging Applications ......................................... 76
Retrofitting Makefiles .......................................... 77

Chapter 5  Building Applications for Multiple Platforms ........ 79
General Instructions ............................................. 80
When Is a Schema Neutral? ...................................... 83
  Causes of Varying Data Formats .............................. 83
  Creating Identical Data Formats .............................. 84
Restrictions ....................................................... 85
  Virtual Base Classes ........................................... 85
  Primitive Data Types .......................................... 85
  64-Bit Pointers ................................................ 85
  Pointers to Members ........................................... 85
  Base Class Initialization Order ............................... 85
  Parameterized Classes ........................................ 86
  Sizes for Data Types .......................................... 87
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-Bit and 64-Bit Integral Types.</td>
<td>88</td>
</tr>
<tr>
<td>ossg Neutralization Options.</td>
<td>89</td>
</tr>
<tr>
<td>Neutralizing the Schema</td>
<td>91</td>
</tr>
<tr>
<td>When to Use Neutralization Options</td>
<td>92</td>
</tr>
<tr>
<td>Versioned Architecture Sets</td>
<td>93</td>
</tr>
<tr>
<td>Additional Neutralization Considerations</td>
<td>93</td>
</tr>
<tr>
<td>Updating a Database Schema to Be Neutral</td>
<td>93</td>
</tr>
<tr>
<td>Command Line and Neutralization Examples</td>
<td>94</td>
</tr>
<tr>
<td>Using a Makefile to Obtain Neutralization Instructions</td>
<td>97</td>
</tr>
<tr>
<td>Building a Heterogeneous Application from a Neutral Schema</td>
<td>98</td>
</tr>
<tr>
<td>Listing Nondefault Object Layout Compiler Options</td>
<td>98</td>
</tr>
<tr>
<td>Compiler Option File Format.</td>
<td>99</td>
</tr>
<tr>
<td>Overriding Options Within the Compiler Option File</td>
<td>101</td>
</tr>
<tr>
<td>Sample Compiler Option File</td>
<td>101</td>
</tr>
<tr>
<td>Schema Generator Instructions.</td>
<td>101</td>
</tr>
<tr>
<td>Base Class Padding Macros</td>
<td>101</td>
</tr>
<tr>
<td>Dynamically Defined Padding Macros</td>
<td>102</td>
</tr>
<tr>
<td>Member Padding Macros</td>
<td>102</td>
</tr>
<tr>
<td>Templated Virtual Base Classes</td>
<td>103</td>
</tr>
<tr>
<td>Neutralizing the Allocation Order of Virtual Base Classes</td>
<td>104</td>
</tr>
<tr>
<td>Database Growth</td>
<td>105</td>
</tr>
<tr>
<td>Endian Types for ObjectStore Platforms</td>
<td>106</td>
</tr>
<tr>
<td><strong>Chapter 6</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Working with ObjectStore/Single</strong></td>
<td>109</td>
</tr>
<tr>
<td>ObjectStore/Single Features</td>
<td>110</td>
</tr>
<tr>
<td>ObjectStore/Single API</td>
<td>110</td>
</tr>
<tr>
<td>ObjectStore/Single Utilities</td>
<td>110</td>
</tr>
<tr>
<td>Dynamic Library Load Path</td>
<td>111</td>
</tr>
<tr>
<td>Application Development Sequence.</td>
<td>112</td>
</tr>
<tr>
<td>Server Log Propagation</td>
<td>113</td>
</tr>
<tr>
<td>Server Log Functions</td>
<td>113</td>
</tr>
<tr>
<td>Log File Guidelines</td>
<td>114</td>
</tr>
<tr>
<td>When to Intervene</td>
<td>115</td>
</tr>
<tr>
<td>Remote Access</td>
<td>115</td>
</tr>
</tbody>
</table>
## Contents

- Accessing Server Logs and Cache Files Through NFS ............ 116
- Packaging ObjectStore/Single Applications .................... 116
  - Cache and Server Log Files .................................. 116
  - Additional Considerations ................................. 117
  - Cache File Considerations ................................ 117
  - Server Log File Considerations ............................ 117
  - What You Should Tell Your Customers ..................... 118
  - Packaging an ObjectStore/Single Application ............. 118
  - Packaging with a VAR Product .............................. 118

## Index .......................................................... 119
Preface

Purpose  
*Building ObjectStore C++ Applications* provides information and instructions for generating schemas and for compiling, linking, and debugging applications that use the ObjectStore C++ application programming interface (API). This book describes ObjectStore Release 6.1.

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  
Much of this book provides information that applies to all ObjectStore platforms. Chapters 4 and 5 contain platform-specific chapters. Chapter 6 contains information specific to ObjectStore / Single.
Notation Conventions

This document uses the following conventions

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

ObjectStore on the World Wide Web

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

Technical Support

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

- You can send questions to support@objectstore.net. Remember to include your site ID in the body of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems.
- You can access the Technical Support Web site, which includes
A template for submitting a support request. This helps you provide the necessary details, which speeds response time.

Frequently asked questions (FAQs) that you can browse and query.

Online documentation for all ObjectStore products.

White papers and short articles about using ObjectStore products.

Sample code and examples.

The latest versions of ObjectStore products, service packs, and publicly available patches that you can download.

Access to an ObjectStore product matrix.

Support policies.

Local phone numbers and hours when support personnel can be reached.

**Education Services**

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

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

**Searchable Documents**

In addition to the online documentation that is included with your software distribution, the full set of product documentation is available on the ObjectStore Support Web server. The documentation is found at www.objectstore.net/documentation, and is listed by product. The site supports the most recent release and the previous supported release of ObjectStore documentation. Service Pack README files are also included to provide historical context for specific issues. Be sure to check this site for new information or documentation clarifications posted between releases.

**Your Comments**

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

Subject: Doc: Incorrect message on page 76 of reference manual
Chapter 1
Overview of Building an Application

This chapter provides general information on 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

- Basic Steps for Building Applications
- Flow Chart for Building Applications
- Third-Party Compilers
- Third-Party Libraries and Applications
- ClearCase Virtual File System (MVFS)
- ObjectStore Server and the Build Process
- ObjectStore / Single

Basic Steps for Building Applications

An ObjectStore application is a C++ program that uses ObjectStore. To make it possible for an application to use ObjectStore, you need to perform the following steps:

1. Modify the source.
   Modify your application source code to make ObjectStore API calls. See the C++ API User Guide for specific information.
Modify your makefile or project settings to find ObjectStore header files. See ObjectStore Header Files on page 10.

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 the

- 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 13 for details.

3 Generate a schema with ossg.

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

- Schema source file
- ObjectStore library schemas

The output from this step are 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 and the definitions of any get_os_typespec() member functions.

See Chapter 3, Generating Schemas, on page 21, for further information.

4 Compile the application schema source file.

Make sure your makefile enables you to compile the application schema source file. This step 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.
See Compiling, Linking, and Debugging Programs on page 53 for more information compiling schema files.

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 in the following diagram. 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 Multiple Platforms, on page 79.
You can use the third-party compilers listed as supported in the ObjectStore Release 6 Service Pack README file. The support matrix supplied there and on the Support Web site provide the most recent information about operating system revision and compiler support.
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 allows you 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 the following conditions are true:

- All persistent data is read or written inside a transaction.
- Library APIs are unique.
- ObjectStore operator delete comes first.
- There are no address space collisions.
- For Windows applications, the load locations of libraries do not collide.

ClearCase Virtual File System (MVFS)

If your ObjectStore applications reside in a ClearCase virtual file system (MVFS), they might have trouble locating an ObjectStore server. To overcome this problem, you should set up locator files or use ObjectStore / Single. For more information on using locator files, see Chapter 5, Using Locator Files to Set Up Server-Remote Databases, in Managing ObjectStore. For more information about ObjectStore/Single, see Chapter 6, Working with ObjectStore / Single, on page 109 of this book.

ObjectStore Server and the Build Process

The ObjectStore server is involved in the build process during schema generation. Because the schema is a regular ObjectStore database, ossg needs to be able to find an ObjectStore server when you invoke it to generate a 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.

You can run an application either as an enterprise ObjectStore application or as an ObjectStore/Single application, depending on how you set the library load path. See the summary comparison, below, for the library load path details.

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, ObjectStore/Single is most useful for applications that do not require

- Concurrency
- A networked server
- Rawfs support

ObjectStore/Single uses the same API as enterprise ObjectStore, and the rules for linking ObjectStore/Single applications are the same as for full ObjectStore, which minimizes 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 to package the application using ObjectStore/Single as a replacement. This replacement eases integration of embedded applications.
A summary comparison of enterprise ObjectStore and ObjectStore/Single follows:

<table>
<thead>
<tr>
<th>Full ObjectStore</th>
<th>ObjectStore/Single</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation requires root permission.</td>
<td>Installation does not require root permission.</td>
</tr>
<tr>
<td>Server is a separate process.</td>
<td>Server functions are integrated in a single library.</td>
</tr>
<tr>
<td>Cache manager is a separate process.</td>
<td>Cache manager functions are integrated in a single library.</td>
</tr>
<tr>
<td>Supports rawfs.</td>
<td>Does not support rawfs.</td>
</tr>
<tr>
<td>UNIX platforms use OS_ROOTDIR/lib/libos and libosdbu shared libraries.</td>
<td>UNIX platforms use OS_ROOTDIR/libsngl/libos and libosdbu shared libraries.</td>
</tr>
<tr>
<td>Windows platforms use %OS_ROOTDIR%\bin DLLs.</td>
<td>Windows platforms use %OS_ROOTDIR%\binsngl.</td>
</tr>
<tr>
<td>Transaction log files are created automatically by the server during ObjectStore installation.</td>
<td>Transaction log files are specified when an ObjectStore application runs.</td>
</tr>
<tr>
<td>Cache files are created automatically by the cache manager process.</td>
<td>Cache files are specified when an ObjectStore application runs.</td>
</tr>
<tr>
<td>The osserver process automatically manages crash recovery and server log propagation.</td>
<td>The 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</td>
</tr>
<tr>
<td>Uses server and cache manager parameter files.</td>
<td>Does not use server and cache manager parameter files.</td>
</tr>
</tbody>
</table>

See Chapter 6, Working with ObjectStore/Single, on page 109, 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
ObjectStore Header Files
Instantiation Problem with Template Collection Classes
Determining the Types in a Schema
Creating Schema Source Files
When You Modify a Source File

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 15.
ObjectStore Header Files

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

<table>
<thead>
<tr>
<th>If You Use This Feature</th>
<th>Include These Header Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any ObjectStore feature</td>
<td>ostore/ostore.hh</td>
</tr>
<tr>
<td>Collections</td>
<td>ostore/ostore.hh, ostore/coll.hh</td>
</tr>
<tr>
<td>Compactor</td>
<td>ostore/ostore.hh, ostore/compact.hh, ostore/coll.hh</td>
</tr>
<tr>
<td>Database utilities</td>
<td>ostore/ostore.hh, ostore/dbutil.hh</td>
</tr>
<tr>
<td>Metaobject protocol</td>
<td>ostore/ostore.hh, ostore/mop.hh</td>
</tr>
<tr>
<td>Relationships</td>
<td>ostore/ostore.hh, ostore/coll.hh, ostore/relat.hh</td>
</tr>
<tr>
<td>Schema evolution</td>
<td>ostore/ostore.hh, ostore/manschem.hh, ostore/schmevol.hh</td>
</tr>
<tr>
<td>Schema generation</td>
<td>ostore/ostore.hh, ostore/manschem.hh</td>
</tr>
</tbody>
</table>

Instantiation Problem with Template Collection Classes

Some compilers have a problem instantiating ObjectStore template collection classes correctly. The problem manifests itself as an error report indicating that one of the following template collection classes is undefined at link time:

- 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_CollectionDeclare()`
- `os_CollectionDeclareNoClass()`
- `os_CollectionDeclarePtrTdef()`

Use these macros to declare the more common cases of forward definitions for ObjectStore template collection classes.

**os_CollectionDeclare() Macro**

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

```c
os_CollectionDeclare(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 at which you use the `os_CollectionDeclare()` macro.

**os_CollectionDeclareNoClass() Macro**

Because `os_CollectionDeclare()` 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_CollectionDeclareNoClass()` preprocessor macro instead. For example:

```c
template<int size> class Fixed_Array;
typedef Fixed_Array<5> Fixed_Array_5;
os_CollectionDeclareNoClass(Fixed_Array_5);
```
You also use the `os_CollectionDeclare_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 include a statement of the following form in your source module:

```cpp
os_CollectionDeclare_no_class(int);
```

### os_CollectionDeclare_ptr_tdef() Macro

The `os_CollectionDeclare_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:

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

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

```cpp
os_CollectionDeclare_ptr_tdef(pPerson);
```

### Required Order of #include Statements

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

```cpp
#include <ostore/ostore.hh>
os_CollectionDeclare(Person);
os_CollectionDeclare(Employer);
#include <ostore/col.le.hh>
```

The following code does not work:

```cpp
#include <ostore/ostore.hh>
#include <ostore/col.le.hh>
os_CollectionDeclare(Person);
os_CollectionDeclare(Employer);
```

If your ObjectStore C++ application uses ObjectStore template collection classes and includes `<ostore/mop.hh` or `<ostore/schmevol.hh` or both, you must also include `<ostore/semoptwk.hh`. This header file provides invocations of the `os_CollectionDeclare()` 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:

```cpp
#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, you need not 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 73.

### Determining the Types in a Schema

The schema source file determines the types 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 equals 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 Invoking `ossg` to Generate a Schema on page 26.
Determining the Types in a Schema

The 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, index path, or restricted cursor.

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

The Types That 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, 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 whether the type is in the schema. If it is not, you must mark it. For information about `osexschm`, see `osexschm`: Displaying Class Names in a Schema in Chapter 4 of Managing ObjectStore.
Choosing Whether to Mark Types or Specify -mrscp

If you mark a type, 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. Because 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; 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 use your C++ compiler instead of ossg to find errors in your schema source file. When you are satisfied that your schema source file compiles without errors, use ossg to generate the schema.
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 (ossg).

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 13 to help you decide. Then follow these steps to create a schema source file. See Schema Source File Examples on page 17 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 of the files is at ObjectStore Header Files on page 10.
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 (vtbls) 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 the types to mark. The order in which you mark types does not matter.

Each call is on its own line and has the format

\[
\text{OS\_MARK\_SCHEMA\_TYPE(type-name);} \\
\text{OS\_MARK\_SCHEMA\_TYPE()}\text{ is a preprocessor macro. For additional information about }\text{OS\_MARK\_SCHEMA\_TYPE()}\text{ and }\text{OS\_MARK\_SCHEMA\_TYPESPEC()},\text{ see Chapter 4, System-Supplied Macros, in }\text{C++ API Reference.}
\]

6 Mark parameterized types that have multiple arguments with a call to the macro \text{OS\_MARK\_SCHEMA\_TYPESPEC()}. This macro is similar to \text{OS\_MARK\_SCHEMA\_TYPE()} in syntax and function. Note that you must enclose the type and its arguments in its own set of parentheses, which results in the double set of parentheses.

Each call is on its own line and has the format

\[
\text{OS\_MARK\_SCHEMA\_TYPESPEC((type-name<x,y>))};
\]

7 Save the schema source file.

**Schema Source File Examples**

```cpp
#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 \text{ostore.hh} and \text{manschem.hh} files are included. The \text{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 — \text{schedule}, \text{ticket}, and \text{passenger} — need to be marked. Each included class is marked on its own line with a call to the \text{OS\_MARK\_SCHEMA\_TYPE()} macro.

```cpp
#include <ostore/ostore.hh>
#include <ostore/coll.hh>
#include <ostore/manschem.hh>
#include "schmdefs.hh"
OS_MARK_SCHEMA_TYPE(drawing);
OS_MARK_SCHEMA_TYPE(view);
```
When You Modify a Source File

```
OS_MARK_SCHEMA_TYPE(layer);
OS_MARK_SCHEMA_TYPE(coordinates);
```

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

Schema Source File Must Be Compilable

The schema source file can contain only valid C++ code. It is good practice to make sure that your schema source file compiles before you use it as input to \texttt{ossg}. However, you should use your C++ compiler instead of \texttt{ossg} to find errors in your schema source file. (With some C++ compilers, you may need to add a dummy function to the schema source file, just for test purposes.) When you are satisfied that your schema source file compiles without errors, use \texttt{ossg} to generate the schema.

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
- A class that is in a library schema used to make the application schema

You also must 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 regenerate a schema automatically when required. Changes to source files have the same compiling and linking implications as they have 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 one of 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, Advanced Schema Evolution, in the *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 ossg schema generator to generate application, component, library, and compilation schemas.

Caution
Be sure you can successfully compile your code using a C++ compiler before you generate a schema. You should not use the schema generator to validate your code. When you are satisfied that your schema source code compiles without errors, use ossg to generate the schema.

The topics discussed in this chapter are

- Overview
- Invoking ossg to Generate a Schema
- Generating an Application or Component Schema
- Generating a Library Schema
- Generating a Compilation Schema
- Hiding Code from the Schema Generator
- Using long long Data Type
- Schema Incompatibilities (Windows)
- Correcting Schema-Related Errors
- Handling #pragma Statements in Source Code
- Utilities for Working with Schemas
- Comparison of ossg Command Lines
- Comparison of Kinds of Schemas
- Deploying Products with Protected Databases
- Using Rogue Wave with Solaris Sun C++
- ossg Troubleshooting Tips
Overview

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, called a `remote schema database`.

Each schema database must be accessible to an ObjectStore server.

Application Schemas

An application schema contains descriptions of

- Classes stored in or read from persistent memory by your application
- Classes stored in or read from persistent memory by a library with which your application links

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

An application schema is associated with an executable.

For simple applications, you can use a single invocation of `ossg` to generate an application schema. In more complex applications, you may want 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 schemas 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 schemas.

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. A library schema corresponds to a static library.

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

If you link your application with a library provided by ObjectStore, linking with the schema is handled automatically.

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.

A compilation schema corresponds to an object file and is used like an object file when an ObjectStore application is being built.

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 the types of the objects that have ever been stored or are expected to be stored in the database.
Remote Schema 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. You cannot use application, component, compilation, or library schemas as remote schema databases.

Schema installation

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 the 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. In your source code, you can specify incremental schema installation for a particular database.

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

<table>
<thead>
<tr>
<th>Batch Schema Installation</th>
<th>Incremental Schema Installation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema is larger.</td>
<td>Schema is smaller.</td>
</tr>
<tr>
<td>Administrative work is done at the beginning.</td>
<td>Administrative work is done in steps; there is greater potential for concurrency conflict because each incremental schema installation uses write locks as it modifies pages.</td>
</tr>
</tbody>
</table>
Keeping Database Schemas and Application Schemas Compatible

Each ObjectStore database has its own database schema. An ObjectStore 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 names, 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
- Both classes must have the same values for enum data members

ObjectStore flags differences as schema validation exceptions.

Multiple component schemas can be used in one application.

If two program schemas 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 shows up as a schema validation error.

When a new type is installed into a database schema, it is validated against all program schemas, and a schema validation error is 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 schema from one or more schemas that were built on other platforms, ossg aborts the process and displays a message indicating the reason. 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.
An application schema corresponds to an executable, a component schema corresponds to a DLL, a library schema corresponds to a static 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.

Invoking ossg to Generate a Schema

This section describes the `ossg` command syntax for generating application, component, library, and compilation schemas. Portions of the command line are on different lines to make them more understandable. This section covers the commonly used `ossg` options; for information about all `ossg` options, see `ossg: Generating Schemas in Chapter 4 of Managing ObjectStore`.

Syntax

**Application or component schema:**

```
ossg [neutralizer_options] [additional_options]
[{-assf app_schema_source_file |
-asof app_schema_object_file.obj}]
-asdb app_schema.adb schema_source_file [lib_schema.ldb]
```

**Library schema:**

```
ossg [neutralizer_options][additional_options]
-lsdb lib_schema.ldb schema_source_file
```

**Compilation schema:**

```
ossg [neutralizer_options] [additional_options]
-csdb comp_schema.cdb schema_source_file
```

**Preprocessing:**

```
ossg [compiler_options] [neutralizer_options]
-E schema_source_file
```

Options

The **neutralizer_options** are as follows:

```
[-arch set...] [{-padm | -padc}] [-nout filename] [-showsets]
[{-showd | -showw}] [-sopt option_file] [-ignore_vbo]
[-ptn name size] [{wchar_t2 | wchar_t4}]
```

The **additional_options** are as follows:

```
[[-mrlcp | -mrscp ]] [-rdp] [-no_weak_symbols]
[-weak_symbols] [-no_default_includes] [-mrscp] [-smf]
```

**Argument**

`schema_source_file`
Chapter 3: Generating Schemas

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.

This argument is almost always required; there is no default. These are the two cases when the option is not required:

- When you generate an application schema from a compilation schema, as described in Using a Compilation Schema to Generate an Application Schema on page 41
- If you specify one or more library schemas that contain all the persistent types that your application uses

Common Options
Following is a description of the common options used with the `ossg` command.

`compiler_options`

Specifies any options that would be passed to the compiler if you were compiling a schema source file instead of generating a schema from it. You should include any preprocessor options, such as `include` file paths and macro definitions, and compiler options that might affect object layout, such as packing options (for example, `/Zp4` for Visual C++).

Optional.

If you specify the `/I, -I, /D, or -D` option, do not include a space between the option and the argument. For example, on Windows, the following is correct:

```
ossg /I%OS_ROOTDIR%\include ...
```

On UNIX, do not specify the `-o` option on the `ossg` command line. On Windows, do not specify `/Tp` on the `ossg` command line.

`-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 without the need to package all your application’s `include` files.
Invoking ossg to Generate a Schema

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

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

additional_options

Use the following options to specify items such as the kind of schema you are building, how to handle vtbls, the path names to various files, and what classes are reachable.

\(-mrlcp \text{ or } -\text{make\_reachable\_library\_classes\_persistent}\)

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 \ossg, the application schema does not include any types. You must mark at least one type for there to be any reachable types.

If this option is not specified, the default is that only marked classes are persistently allocatable and accessible.

See also Determining the Types in a Schema on page 13.

\(-mrscp \text{ or } -\text{make\_reachable\_source\_classes\_persistent}\)

Causes every class that is both defined in the schema source file and 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 the 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 \text{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 -mrscc when you run ossg, the application schema does not include any types. You must mark at least one type for there to be any reachable types.

If this option is not specified, the default is that only marked classes are persistently allocatable and accessible.

See also Determining the Types in a Schema on page 13.

-no_default_includes or -I-

When you specify this option, ossg does not automatically specify any include directories to the C++ preprocessor. However, the preprocessor can have default include directories built in. If any directories are built into the preprocessor, ossg 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: /I%OS_ROOTDIR%\include

If this option is not specified, the default is that the preprocessor checks default directories for included files.

-no_weak_symbols

Disables mechanisms that suppress notification about missing vtbls. This option allows you to check whether any vtbl symbol referenced is undefined.

If you specify -rtdp maximal -no_weak_symbols, the linker outputs messages about what is missing. You can use this information to determine those additional classes that you need to mark. These missing
Invoking ossg to Generate a Schema

symbols are only a hint about what you might consider marking. They might also be the result of a link line error.

This default option specifies that the schema generator notify you of missing vtbls. Specify the option -weak_symbols to suppress this behavior.

-rtdp or -runtime_dispatch  {minimal | derived | full | maximal}

Specifies the classes for which the schema generator makes vtbls available.

minimal specifies marked classes, classes embedded in marked classes, and base classes of marked classes.

derived specifies the minimal set and the classes that derive from marked classes and classes embedded in the derived classes.

full specifies the derived set and 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.

See Example of Using the -runtime_dispatch Option on page 34.

If this option is not specified, the default is derived.

-smf or -store_member_functions

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 Metaobject Protocol (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.

When you use this option, 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.

If this option is not specified, 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 vtbls. This option overrides the default behavior described at `-no_weak_symbols` on page 29.

`-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`. On UNIX platforms, use the `-assf` option, which causes the schema generator to produce a source file that you must compile. On Windows, use the `-asof` option, which causes the schema generator to produce the object file directly.

This option is required when generating a schema. There is no default.

`-asdb app_schema.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 path name must be local to a host running an ObjectStore server.

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

This option is required when generating a schema. There is no default.

`-csdb comp_schema.cdb ...`

Builds the compilation schema database specified by `comp_schema.cdb`. Technical Support recommends, but does not require, that the path name end in `.cdb`.

This option is required when generating a compilation schema. There is no default.

`-lsdb lib_schema.ldb ...`
Invoking ossg to Generate a Schema

Builds the library schema database specified by

    lib_schema.ldb. The name must end in .ldb. This is 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.

If this option is not specified, the default is that library schemas are not
included.

neutralizer_options

Include any of the following options. These options allow you to
neutralize a schema for a heterogeneous application. You can include the
options in any order. For more information about the neutralization
options, see ossg Neutralization Options on page 89.

-arch set

The schema that is generated or updated will be neutralized to be
compatible with the platforms in the specified architecture set. This option
is required when you are neutralizing schema. There is no default. See
ossg Neutralization Options on page 89 for more information about the
-arch option.

-ignore_vbo

Suppress any warnings about differences in the allocation order of virtual
base classes. For more information, see Neutralizing the Allocation Order
of Virtual Base Classes on page 104.

-nout filename or
-neutral_info_output filename

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

Optional. The default is that the schema generator sends output to
stderr.

-padc or -pad_consistent | -padm or -pad_maximal

Neutralizer options that indicate the type of padding requested. See ossg
Neutralization Options on page 89 for additional information about these
options.

If this option is not specified, the default is -padc.
Chapter 3: Generating Schemas

- ptn name size or -portable_type_name name size

Specifies that a type, name, has a given size, in bytes, on all architectures in the architecture set specified with -arch.

Usually name is conditionally defined in a typedef and it must be an integral type.

-showsets

Lists the names of all of the architecture sets that can be specified as arguments to the -arch option. The listing also includes the names of platforms that are members of each set. For more information about architecture sets, see ossg Neutralization Options on page 89.

-showd or -show_difference | -showw or -show_whole

Neutralizer options that indicate the description level of the schema neutralization instructions.

If this option is not specified, the default is -show_whole.

-sopt option_file or -schema_options option_file

These options specify 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 98 for further information.

-wchar_t2 | -wchar_t4

Neutralizer options that specify the size of wchar_t to be either 2 or 4 bytes. If wchar_t on any of the platforms within the set specified by -arch differ in size, use a typedef to define them conditionally to the correct size. One of these options is required if a schema includes a wchar_t.

Changing the Default Preprocessor

You can use the OS_OSSG_CPP environment variable to specify a C preprocessor other than the configured default. The default preprocessor on UNIX platforms is the platform compiler with the appropriate preprocessing flags. The default preprocessor on Windows is cl.

Using a Temporary File to Send Arguments to ossg

You can specify the following option on an ossg command line when you are generating any kind of schema:

@filename
Invoking ossg to Generate a Schema

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 Windows NT, this option 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) <<

Example of Using the -runtime_dispatch Option

The -runtime_dispatch (-rtdp) option specifies the set of classes for which the schema generator makes vtbls 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 vtbls are available). Note that class H is not reachable from class D and, consequently, is not in the set unless you mark it explicitly.

<table>
<thead>
<tr>
<th>Option Specified</th>
<th>Classes in the Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimal</td>
<td>A and D</td>
</tr>
<tr>
<td>derived</td>
<td>A, D, E, and F</td>
</tr>
</tbody>
</table>
Generating an Application or Component Schema

Application and component schemas describe the classes stored in or read from persistent memory by an application or a library linked to the application.

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.

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 and the definitions for any get_os_typespec() member functions.

You must compile this file, then link the resulting object file with your application or DLL. (Chapter 4, Compiling, Linking, and Debugging Programs, on page 53, provides details on linking.)

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

---

<table>
<thead>
<tr>
<th>Option Specified</th>
<th>Classes in the Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>full</td>
<td>A, C, D, E, and F</td>
</tr>
<tr>
<td>maximal</td>
<td>A, B, C, D, E, F, and G</td>
</tr>
</tbody>
</table>
Following is an illustration of the application schema generation process, extending to just before the link step.

**Invoking ossg to Generate an Application Schema**

Use the following `ossg` command syntax to generate an application or component schema.

```
ossg [neutralizer_options] [additional_options] [{-assf app_schema_source_file | -asof app_schema_object_file.obj}] 
-asdb app_schema.adb schema_source_file [lib_schema.ldb ...]
```

See Invoking ossg to Generate a Schema on page 26 for information about `ossg` options.

**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 also share the application schema.

By default, when the first application accesses the database, ObjectStore installs the application schema. When subsequent applications access the database, ObjectStore does not need to perform installation and access is faster.

Examples of Generating an Application Schema

**UNIX**

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

The `-assf` option specifies 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 not marked in the schema source file. The `-assf` option specifies that `myassf.c` is the name of the application schema source file to be produced by `ossg`. The `-asdb` option specifies 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, therefore, includes the types in those libraries.

**Windows**

```
ossg -asof jetsch.obj -asdb jetsch.adb jetsch.cc
```

Because this is a Windows platform, the schema generator directly produces the application schema object file. The `-asof` option specifies 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`. 
Generating a Library Schema

Locating a Schema Database

An ObjectStore application or DLL contains a string that indicates the location of the associated schema database. If, for some reason, you need to move the schema database, you should use the `ossetasp` utility to patch the application or DLL with the path to the schema database’s new location. You can also use the `OS_SCHEMA_PATH` environment variable to accomplish this. If you move your application to another machine, you will need to use one of these methods to specify the new path.

For more information, refer to the following topics in Managing ObjectStore:

- `ossetasp`: Patching Executables with Application Schema Path Names
- `OS_SCHEMA_PATH`

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. Your application needs access to all types that are persistently allocated. These should be marked in either the application schema or the library schema.

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

You need not generate a library schema if you link with the libraries provided with ObjectStore.

The commands used in a makefile or project file to generate a library schema are similar to those used to generate 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.
Invoking ossg to Generate a Library Schema

Use the following ossg command syntax to generate a library schema.

```
ossg [neutralizer_options] [additional_options]
   -lsdb lib_schema.ldb schema_source_file
```

See Invoking ossg to Generate a Schema on page 26 for information about ossg options.

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 information from the schema source files `s1.cc` and `s2.cc` in the library schema `foo.ldb`, you 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 run ossg again, ossg adds to `foo.ldb`. It does not overwrite `foo.ldb`. If the modifications that you made to `s1.cc` are not consistent with what is already in `foo.ldb`, an exception will occur.

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.
Generating a Compilation Schema

When the schema generator generates an application schema, it internally creates a compilation schema first, 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.

The commands used in a makefile or project file to generate a compilation schema are similar to those used to generate 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.

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 you are 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 you are performing schema evolution.

Invoking `ossg` to Generate a Compilation Schema

Use the following `ossg` command syntax to generate a compilation schema.

```
ossg [neutralizer_options] [additional_options]
   -csdb comp_schema.cdb schema_source_file
```

See Invoking `ossg` to Generate a Schema on page 26 for information about `ossg` options.
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`.

Examples

On a UNIX platform:

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

On Windows:

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

Hiding Code from the Schema Generator

You can use the `_ODI_OSSG_` C preprocessor (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, when there is an error in code
    that 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.
Using long long Data Type

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.

Using long long Data Type

A `long long` is an `int` type with a length of 64 bits. The `long long` type is treated as a new type on SPARCompiler C++.

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.

If you develop applications that use 64-bit `int` types and that run on different platforms, see 32-Bit and 64-Bit Integral Types on page 88 for information on the `-portable_type_name` option.

Schema Incompatibilities (Windows)

Because of changes in Microsoft’s Visual C++ compiler from version 6 (vc6) to version 7 (vc7), users who upgrade their applications from vc6 to vc7, or who wish to use both versions, may see schema incompatibilities that affect schema generation. The following sections discuss these incompatibilities and their workarounds.

Differing Support for `#pragma pack(pop,N)`

Support for `#pragma pack(pop,N)`, where `N` is some alignment, differs between vc6 and vc7. In particular, the following set of pragmas results in different alignments between vc6 and vc7:

```c
#pragma pack(push,11,2)
#pragma pack(push,12,4)
#pragma pack(pop,1)
```
On vc6, these pragmas result in a two-byte alignment; whereas on vc7 they result in a one-byte alignment. The workaround for this incompatibility is to replace the last pragma with the either of two sets of pragmas. The following set ensures one-byte alignment that is compatible with vc7:

```c
#pragma pack(pop)
#pragma pack(1)
```

The following set ensures two-byte alignment that is compatible with vc6:

```c
#pragma pack(pop)
#pragma pack(2)
```

### Differing Support for Integral Extension Types

On vc6, the integral extension types are distinct types. On vc7, they are treated as typedefs for regular C++ types. Thus the following code behaves differently on vc6 and vc7:

```c
template<class T> class printer {
public:
    static void print(const char* source_name) {
        const type_info& ti = typeid(T);
        printf("in source %s, actual %s\n",ti.name());
    }
};
printer<__int16>::print("__int16");
printer<unsigned __int32>::print("unsigned __int32");
```

On vc6, the output is:

```
in source __int16, actual __int16
in source unsigned __int32, actual unsigned __int32
```

On vc7, the output is:

```
in source __int16, actual short
in source unsigned __int32, actual unsigned int
```

Furthermore, the symbol for print (or for the virtual function table, if there was one) is different. On vc6, the template argument for __int16 would be _F; on vc7, it would be F.

If you wish to share code or databases between vc6 and vc7 applications, you cannot use integral extension types in any context that appears in the schema, including template instantiations and virtual function table pointers. If you are not sharing code between vc6 and vc7 applications, you can continue to use the integral types and the “mangling” will follow the rules for the compiler version you are using.
Correcting Schema-Related Errors

The following sections discuss schema-related errors that you might need to resolve.

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 osexschm: Displaying Class Names in a Schema in Chapter 4 of Managing ObjectStore.

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 as follows:

- 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 osexschm: Displaying Class Names in a Schema in Chapter 4 of Managing ObjectStore.
Chapter 3: Generating Schemas

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 the way ossg handles type-mismatch errors during schema generation. Set the environment variable \texttt{OS\_COMP\_SCHEMA\_CHANGE\_ACTION} to one of the following values:

- \texttt{warn}
  - Reports a warning. The new type definition replaces the previous definition in the compilation schema. Default.
- \texttt{silent}
  - Specifies that errors are not reported. The new type definition replaces the previous definition in the compilation schema.
- \texttt{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 in which 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.

Schema Neutralization Errors

For information about handling schema neutralization errors, see ossg Neutralization Options on page 89.

Missing Virtual Function Table Pointer Problems

For information about missing virtual function tables, see Run-Time Errors from Missing vtbls on page 75.
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, ossg usually does not treat #pragma statements that occur inside a class definition 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.
- ossetrsp sets the path name of a remote schema database.
- ossevol updates a database and its schema so that it matches a revised application or component schema.

Information about all these utilities appears in Chapter 4, Utilities, of Managing ObjectStore.
Comparison of ossg Command Lines

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

<table>
<thead>
<tr>
<th>Kind of Schema</th>
<th>Syntax for ossg Command</th>
</tr>
</thead>
</table>
| Application or component | `ossg [compilation_options] [neutralizer_options]`  
| | `-cpp_fixup` `-final_asdb final_app_schema_db`  
| | `[-rtdp {minimal | derived | full | maximal}]`  
| | `[-store_member_functions]`  
| | `{-assf app_schema_source_file | -asof app_schema_object_file.obj}`  
| | `-asdb app_schema.adb schema_source_file [lib_schema.ldb ...]`  
| 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.adb comp_schema.cdb
```
## Comparison of Kinds of Schemas

<table>
<thead>
<tr>
<th>Kind of Schema</th>
<th>What Does It Contain?</th>
<th>How Is It Generated?</th>
<th>When Is It Used?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application</strong></td>
<td>Definitions of classes the application stores in or reads from persistent memory and classes persistently used by libraries the application links with.</td>
<td>Specify the <code>-asdb</code> option when invoking <code>ossg</code>. Naming convention recommended for an application schema: <code>yourchoice.adb</code></td>
<td>ObjectStore uses the application schema during run time to determine the layout of objects being transferred between the database and the application.</td>
</tr>
<tr>
<td><strong>Component</strong></td>
<td>Definitions of classes a DLL or shared library stores in or reads from persistent memory.</td>
<td>Specify the <code>-asdb</code> option when invoking <code>ossg</code>. Naming convention recommended for an application schema: <code>yourchoice.adb</code></td>
<td>A component schema can be loaded and unloaded dynamically during run time. An application can load multiple component schemas.</td>
</tr>
<tr>
<td><strong>Library</strong></td>
<td>Definitions of types that the library stores or retrieves in a persistent context.</td>
<td>Specify the <code>-lsdb</code> option when invoking <code>ossg</code>. Naming convention required for a library schema: <code>yourchoice.ldb</code></td>
<td>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.</td>
</tr>
</tbody>
</table>
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 Setting the Database’s Schema Key in Chapter 7, Database Access Control, of C++ API User Guide.

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 `OS_SCHEMA_KEY_LOW` in Managing ObjectStore.

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

Your application must include these characteristics because 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.

```bash
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++ 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 workarounds for this:

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

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


Chapter 3: Generating Schemas

#undef RW_COMPILE_INSTANTIATE

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

## ossg Troubleshooting Tips

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

### Check source files for errors

Do not use `ossg` to find syntax and semantic errors in your source files. Compile your sources with the C++ compiler before running `ossg` over your source files. With some compilers you may need to insert a dummy function for test purposes.

### 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 two 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 it can install virtual function table (vtbl) information in the schema correctly.
- If the `-store_member_functions` option of `ossg` is specified, the member function declarations must be visible to `ossg`.

If neither of these conditions applies, functions (including member functions) can be hidden from `ossg` using conditional compilation or exclusion.

### 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. In generating schemas, `ossg` sometimes uses large amounts of swap space, so Technical Support recommends that you make a large amount of swap space available when running `ossg`.

---

Release 6.1
When problems occur during schema generation (such as syntax errors, semantic errors, or any other errors), you should send the following to Technical Support:

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

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

For example:

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

After you produce the preprocessed source, please contact Technical Support for a password so you can put the source on Technical Support’s ftp server.
Chapter 4
Compiling, Linking, and Debugging Programs

This chapter provides information about compiling, linking, and debugging your application. The first two topics provide information specific to building applications on UNIX and Windows platforms. The remaining topics treat specific issues involved in building ObjectStore applications.

UNIX
Windows
Moving an Application and Its Schema
Working with Virtual Function Table Pointers
Missing vtbls
Run-Time Type Information
Debugging Applications
Retrofitting Makefiles

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 the C++ interface to ObjectStore Release 6.1.
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.

Requirements for Linking

You always link with the `libos` library. If you are using full ObjectStore, use `$OS_ROOTDIR/lib/libos`. If you are using ObjectStore / Single, use `$OS_ROOTDIR/libsngl/libos`.

When linking on Linux and Solaris 2, if your application uses multiple threads, you must link it with the threads library `libosth`. If you have a real threads package, this library can be found in `$OS_ROOTDIR/lib`. If you are not using a real threads package, you must use the stubs library in `$OS_ROOTDIR/lib/threadstub`. If you use the thread stub library, you must set your library path (such as `LD_LIBRARY_PATH`) to include `$OS_ROOTDIR/lib/threadstub` before `$OS_ROOTDIR/lib`. For information about compiling and linking multi-threaded applications on Solaris 2, see Compiling and Linking on Solaris 2 on page 56.

Single-threaded applications can be linked with the thread stub library, `libosths`.

You must link with additional ObjectStore libraries according to the features you use in your code, as shown in the following table. If your application is single-threaded, you can use the thread stub library (`libosths`) in place of the thread safe library (`libosth`) that is listed in the table. If there is more than one library, you must specify each of them in the order given. See Examples of Passing Libraries to the Linker on page 55.

<table>
<thead>
<tr>
<th>If Your Application Uses This Feature</th>
<th>Link with These Libraries, in This Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any ObjectStore feature</td>
<td>libos [libosth]</td>
</tr>
<tr>
<td>Collections</td>
<td>liboscol libos [libosth]</td>
</tr>
<tr>
<td>Compactor</td>
<td>liboscmp libos [libosth]</td>
</tr>
<tr>
<td>Database utilities:</td>
<td></td>
</tr>
<tr>
<td>• <code>os_dbutil::osverifydb()</code></td>
<td>liboscol libosmvp libostcl libosdbu libos [libosth]</td>
</tr>
<tr>
<td>• All other <code>os_dbutil</code> methods</td>
<td>libosdbu libos [libosth]</td>
</tr>
</tbody>
</table>
When You Are Linking Libraries

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 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 -losth
```

You cannot link with both libosse and liboscmp.

Examples of Passing Libraries to the Linker

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

```
-loscmp -losmop -losqry -loscol -los -losth
```

Note that -losth appears here and in the following link lines as a placeholder and should be replaced with -losth for single-threaded applications.

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

```
-losse -losqry -loscol -losmop -los -losth
```

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

```
-losdbu -los -losth
```

If you need all ObjectStore libraries, link with the libraries in this order:

```
{-losse | -loscmp} -losmop -losqry -loscol -los -losdbu -losth
```
Compiling and Linking on Solaris 2

On Solaris 2.x, if your 32-bit or 64-bit application uses multiple threads, you must compile it with the \texttt{-mt} option, or specify the \texttt{-lthread} option on the link line, and link with the \texttt{$OS_ROOTDIR/lib} version of the \texttt{libosth} library. Here is an example:

\begin{verbatim}
CC -mt -I$(OS_ROOTDIR)/include $(CCFLAGS) -o foo \ 
-L$(OS_ROOTDIR)/lib -los -losth
\end{verbatim}

If your application is single threaded, you can remove the \texttt{-mt} from the compile line or \texttt{-lthread} option from the link line and link with the \texttt{$OS_ROOTDIR/lib/threadstub} version of the \texttt{libosth} library, as in the following example:

\begin{verbatim}
CC -I$(OS_ROOTDIR)/include $(CCFLAGS) -o foo \ 
-L$(OS_ROOTDIR)/lib/threadstub -los -losth
\end{verbatim}

When running your application, make sure that \texttt{$OS_ROOTDIR/lib/threadstub} is before \texttt{$OS_ROOTDIR/lib} in your \texttt{LD_LIBRARY_PATH} environment variable.

Sun C++ Compiler Options

Sun C++ 5.4 has a compile-time option, \texttt{-instance=static} (formerly, \texttt{-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. Because the vtbls are static, ObjectStore cannot get at them and gets the wrong vtbl, which leads to an error.

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; nonetheless, it debugs single-threaded applications.

Sample Makefile Template

The makefile shown later in this section is a template for building ObjectStore applications. This makefile is for an application that uses queries and collections.

In an ObjectStore makefile, in the \texttt{LDLIBS} line, you must specify each library with which you are linking.
Then, in the line of the makefile in which `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.

**Note**

If you are using an online 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**

```makefile
include $(OS_ROOTDIR)/etc/ostore.lib.mk
APPLICATION_SCHEMA_PATH=app-schema-db
LDLIBS = $(OS_EXPORT) -losqry -loscol -los -losth
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)
.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

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`.

---

Release 6.1

57
These handlers are critical to ObjectStore’s operation. If your application disturbs them, it fails, possibly in a way that makes it difficult to determine the reason 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`. In this example, `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.

```makefile
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

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 40.

### Virtual Function Table Pointers

Virtual function table symbols (vtbls) need not 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 that were specified at link time as well as any that were dynamically opened. 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 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.

Linux

On Linux, the standard debugger is gdb. To ignore SIGBUS and SIGSEGV, respectively, specify:

```
handle 10 nostop noprint
handle 11 nostop noprint
```

Windows

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

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 most applications need is ostore.lib or the debug equivalent, ostored.lib. One of these libraries is normally linked automatically (depending on whether you are building a retail or debug application).

When building dump/load applications, you must also link with osdump.lib, osload.lib, osmsgs.lib, osmsgsys.lib, and ostrvrs.lib.

Using 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 to add the rules needed to run the ObjectStore schema generator in the automatically generated makefiles.

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 specify a Custom Build step for it as follows:
Ensuring 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

Following is an example of a compilation command:

mycode.obj: mycode.cpp
  cl /c /EHa /D_X86_=1 /DWIN32 /MD /Zi /vmg /vmv\1$(OS_ROOTDIR)\include mycode.cpp

The options /c, /D_X86_=1, and /DWIN32 are required in approximately this form by all compilations with Visual C++. ObjectStore supports only one form of pointer-to-member, so the options /vmg and /vmv are required.

The option /MD is required for you to use msvcrtd.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.
Chapter 4: Compiling, Linking, and Debugging Programs

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

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

Using 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 (or its debug equivalent msvcrtd.lib).

Each run-time library has its own allocation routines (malloc, operator new) and deallocation routines (free, operator delete). You cannot call an allocator from one library (for example, msvcr) 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. It can do this because all ObjectStore DLLs share the single allocator in MSVCRT.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 functions (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 those obtained from a third party) if 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 the way 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 functions such as 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 DLL.

The following requirements apply:

- You must link ObjectStore applications with the library ostore.lib. With normal ObjectStore installations, this is done automatically. For debug builds, ostored.lib is normally linked with automatically instead of ostore.lib.

- You must link with msvcrt.lib. This is the C run-time library with which all ObjectStore DLLs and executables are linked. With normal ObjectStore installations, this is done automatically. For debug builds, msvcrtd.lib is normally linked automatically instead of msvcrt.lib.

### Building Applications

In makefiles and project build settings, 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` (and `/MDd`) implies `/D_DLL`, so pass the `/D_DLL` to `ossg`. Note the way `$(COMPILER_OPTS)` is used for both compilation and schema generation in the following makefile fragment.

```makefile
!include $(OS_ROOTDIR)\etc\ostore.mak
OBJECTS=note.obj
EXECUTABLES=note.exe
APPLICATION_SCHEMA_PATH=note.adb
all: $(EXECUTABLES)
mschema.obj: schema.cc
   `ossg` -asof mschema.obj -asdb /
   $(APPLICATION_SCHEMA_PATH) /
   $(COMPILER_OPTS)schema.cc
note.obj: note.cc
   `$(COMPILER) $(COMPILER_OPTS) note.cc
note.exe: $(OBJECTS) mschema.obj
   `$(CL_LINK)` /OUT:note.exe $(OBJECTS) mschema.obj
clean:
   `osrm` -f $(APPLICATION_SCHEMA_PATH)
   -del $(OBJECTS) osschema.*
```

If you are building applications from within the Visual C++ IDE, you do not use makefiles, but you need to specify custom build rules for generating schemas. For example, an application’s custom build settings for the schema source might be

```bash
ossg -asof mschema.obj -asdb mschema.adb schema.osg
```

### Specifying Environment Variables

The following environment variables are set automatically 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**
Windows

%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

ObjectStore handles all access violations to determine whether 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 an archive file symbol.zip on the distribution CD-ROM. Unzip this file to the %OS_ROOTDIR% directory.

To obtain a back trace using the debug DLLs, put the debug DLL directory in the front of the path environment, then run the msdev debugger (in VC++ 7, the debugger is devenv/debug.exe ).

For example, using myapps and VC++ 6, you would issue the following commands:

C:\myapps> set path = %OS_ROOTDIR%\symbol\bin;%path%
C:\myapps> start msdev myapps.exe
Setting a Breakpoint
If your application exits with an unhandled TIX exception, you can set a breakpoint to obtain a stack trace before 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 hard-coded 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 Chapter 3, Environment Variables in Managing ObjectStore.
- 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, then examine the stack using the Visual C++ debugger.

Example
Following is a code sample illustrating the way to use set_unhandled_exception_hook():

```cpp
#include <iostream.h>
#include <ostore/ostore.hh>
void break_hook(tix_exception *err, os_int32, char* message){
  cout << "Set breakpoint 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 (MFC)
Applications

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

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, follow these guidelines:

**Issue 1**
Copy the function `AfxWinMain` from `MFC\SRC\WINMAIN.CPP` to an application source file and add the initialization calls to the copy. Doing so overrides the `AfxWinMain` in the unchanged MFC DLL.

**Issue 2**
Put the `OS_ESTABLISH_FAULT_HANDLER` and `OS_END_FAULT_HANDLER` macros in the thread functions passed to `CWinThread::CreateThread`.

**Issue 3**
Modify `stdafx.h`. 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 overloading of `new`. To eliminate these problems, redefine `DEBUG_NEW` to be `new` as opposed to `new(__FILE__,__LINE)`, 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`:

```cpp
#if defined _AFX_NO_DEBUG_CRT
#else
#endif
#define DEBUG_NEW new
#endif
```

66 Building ObjectStore C++ Applications
#ifdef new
#undef new
#define new new
#endif

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

```c
// 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::free_memory(p);}  
#endif
```

**Issue 4**

Never pass persistent data to library calls. Pass copies instead or use os_with_mapped.

**Issue 5**

Do not store MFC classes persistently. See Class os_CString, next.

### 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.

### 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 path name. There are two ways to do this.

You can specify ObjectStore server-relative path names, 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 path names, the ossg command line would look like this:

**Example 1**

```plaintext
ossg -asdb foo:c:\appdir\appschema.adb -assf osschema.cc
```

This example assumes the following:

- The server is on a remote machine named `foo`.
- There is a directory on `foo`’s C drive called `appdir`.

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 path names 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:

```plaintext
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 `ossg` command line:

```plaintext
ossg -asdb x:\appdir\appschema.adb -assf osschema.cc
```

This example assumes the following:

- The server is on a remote machine named `bar`.
- The application is in `/usr/appdir` on `bar`.

### Porting ObjectStore Applications to Windows Platforms

This section presents guidelines for porting ObjectStore code to a Windows environment.

**Threads**

You must use the Visual C++ C run-time functions `beginthread` and `endthread` to initialize the Visual C++ C run-time library. `CreateThread` and `TerminateThread` do not properly initialize the Visual C++ C run-time library.

`objectstore::initialize()` must be called only once, even if the application uses multiple threads.
Visual C++ makes `double` and `long double` 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)
```

Windows debug and symbols 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. The installation automatically includes a debug build that uses the debugging version of the Visual C++ run-time and 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.

You can also install a symbol build manually to aid in debugging your applications. The symbol 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 Technical Support to track down a problem. It is compiled unoptimized and has extra error-checking code built in.

Installing symbol.zip

To install `symbol.zip`, follow these steps:

1. Install ObjectStore Release 6.1 with the `SETUP` program. See Installing ObjectStore for Windows documentation for instructions.
2. Shut down the ObjectStore server and cache manager using the ObjectStore `SETUP` program. Answer Yes to the question about shutting down servers, and then exit from `SETUP`.
3. Go to the `%OS_ROOTDIR%` directory.
4. Unzip the `symbol.zip` file from the distribution CD-ROM. Make sure the application you use for unzipping is set to restore the subdirectories properly. For example, use the `-d` option with `pkunzip`:
pkunzip -d symbol.zip

5 Run the ObjectStore SETUP program to start the server. In the first setup dialog box, select the Setup Server option. In the menu Choosing to start ObjectStore services automatically, click Yes. A Confirm Message dialog box asks if you want to start the services right now. Click Yes.

The retail, symbol, and debug Builds
To use the retail or symbol build, compile your application using the /MD option. 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. To switch to the symbol build from the retail build, add %OS_ROOTDIR%\symbol\bin to your path environment variable, before the normal %OS_ROOTDIR%\bin entry. To switch back to the retail build, remove the %OS_ROOTDIR%\symbol\bin entry in your path environment variable.

To use the debug build, compile your application using the /MDd option. 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.

The following table compares the features of retail, symbols, and debug versions of ObjectStore:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Retail</th>
<th>Symbols</th>
<th>Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation with INSTALL?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Optimized?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ObjectStore symbols available?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Drop-in capability?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Compiling and Linking Applications
Use the following information when compiling and linking applications:

<table>
<thead>
<tr>
<th>Run-time library (DLL)</th>
<th>Retail</th>
<th>Symbols</th>
<th>Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>VC++ 6</td>
<td>msvcrt.dll</td>
<td>msvcrt.dll</td>
<td>msvcrtld.dll</td>
</tr>
<tr>
<td></td>
<td>msvcr70.dll</td>
<td>msvcr70.dll</td>
<td>msvcr70d.dll</td>
</tr>
<tr>
<td>VC++ 7</td>
<td>msvcrt.dll</td>
<td>msvcrt.dll</td>
<td>msvcrtld.dll</td>
</tr>
<tr>
<td></td>
<td>msvcr70.dll</td>
<td>msvcr70.dll</td>
<td>msvcr70d.dll</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compile options</th>
<th>Retail</th>
<th>Symbols</th>
<th>Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/MD</td>
<td>/MD</td>
<td>/MDd</td>
</tr>
</tbody>
</table>
* The `/MDd` option defines the symbol `_DEBUG`, which determines the link library used by ObjectStore. If you want to use `msvcrtd.dll` but not `ostored.lib`, define the symbol `_ODI_FORCE_OSTORE_LIB`, for example, with the compiler option:

```
/D " _ODI_FORCE_OSTORE_LIB"
```

## Moving an Application and Its Schema

If you want to move an ObjectStore application to another machine, use the `ossetasp` utility to patch the executable so that it looks for its application schema in an application schema database that you specify. Normally you leave the application schema on the ObjectStore server host.

### Syntax

```
ossetasp -p executable
ossetasp executable database
```

### Option `-p executable`

Instructs `ossetasp` to display the path name of the specified executable’s application schema database.

### Arguments `executable`

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

### `database`

Specifies the path name of an application schema database. ObjectStore patches the specified executable so that 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.
Working with Virtual Function Table Pointers

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 path name of the application schema database. Be sure to specify the name of the server host.

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

Moving an Application Schema

If you want to move a database and its application schema to a machine other than the server host, you need to use a locator file. See Chapter 5, Using Locator Files to Set Up Server-Remote Databases in Managing ObjectStore.

Working with Virtual Function Table Pointers

At run time, ObjectStore needs to know the locations of virtual function tables in your application program’s executable.

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` or, on some platforms, `vft`). 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, because 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.
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, if ObjectStore needs vtbls, it finds them in tables that map class names to references to vtbls. 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, at program start-up time, or some combination of these, depending on the platform. At each of these steps, the referenced vtbls are searched for in the executable and, if found, are entered into the tables. At run time, ObjectStore uses these tables to find items for relocation.

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 21, 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. On non-Windows platforms, these symbols begin with __vtbl or __vft or end with __vtbl; on Windows, the symbols begin with ??_?. These errors are reported 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*) {
    foo_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 because 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 the following, class A might be missing its vft:

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

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,

```cpp
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 workaround is to declare this and link it. For example:

```cpp
void foo_force_vfts(void*) {
    foo_force_vfts(new os_Set<missing-type>);
}
There are additional considerations for building applications that use collections. See Instantiation Problem with Template Collection Classes on page 10.

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. Because 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 the class 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 to explicitly mark it with \texttt{OS_MARK_SCHEMA_TYPE()}

Inline virtual functions

If all virtual functions of a class are inline, either because they are defined fully in the class specification or with the \texttt{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 and, therefore, not available to ObjectStore) because the locations of the virtual functions were not filled in by \texttt{ossg}.

The solution to this problem is to put an out-of-line virtual function in each class with a missing vtbl. You either can modify an inline function or add a
trivial out-of-line function. To determine the classes that 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 Common Options on page 27.

A class that meets the following conditions might also need the addition of an out-of-line virtual function:

- Has at least one virtual base class
- Has at least one virtual function defined by a base class
- Does not define any virtual functions

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

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

-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 variable. See `OS_TRACE_MISSING_VTBLS` in Chapter 3 of Managing ObjectStore.

### Run-Time Type Information

ObjectStore supports the use, on Solaris and Linux platforms, of C++ run-time type information (RTTI) on persistent instances.

### Debugging Applications

In addition to native debuggers, alternatives for debugging applications include the following:

- **Setting 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 at a later time
Chapter 4: Compiling, Linking, and Debugging Programs

See OS_TRACE_MISSING_VTBLs in Chapter 3 of Managing ObjectStore.

- Running the osverifydb utility after an application creates a database. This allows you to check for invalid pointers.
- Examining incorrect database information with the ObjectStore Inspector.
- Using the ObjectStore Performance Expert (OPE) as a debugging tool, if performance is the issue.

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 more difficult for someone to subvert schema protection by analyzing information provided by the debugger.

For more information about debugging ObjectStore UNIX applications, see Debugging Applications on page 59; for Windows applications, see Debugging Your Application on page 64. In addition, see Troubleshooting in Managing ObjectStore.

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 online version of an ObjectStore makefile from ObjectStore documentation sources, make sure that you have tabs, not spaces, at the beginning of appropriate lines.
Retrofitting Makefiles
Chapter 5
Building Applications for 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 making your application portable.

It covers the following topics:

- General Instructions 80
- When Is a Schema Neutral? 83
- Restrictions 85
- ossg Neutralization Options 89
- Neutralizing the Schema 91
- Listing Nondefault Object Layout Compiler Options 98
- Schema Generator Instructions 101
- Endian Types for ObjectStore Platforms 106
You can build an ObjectStore application on multiple platforms, 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 you build 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 85 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 set (or -arch set) option for the set of compilers on which the application must run — in most cases, all32 or all64. If you are neutralizing the application to run on 32-bit architectures, specify all32; if you are neutralizing the application to run on 64-bit architectures, specify all64.
   b Specify other ossg options, as needed.
      A description of the schema generator neutralization options is on ossg Neutralization Options on page 89.

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 step 1 and step 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, 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` 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.
General Instructions

Your Application Sources
  App. Headers

Application.cc

Schema Source File

Run cc

Run ossg

Modify Application

Produced application schema?
  no

Perform Neutralization

Heterogeneous Build Process

Heterogeneous Applications
  yes

Built on all platforms?
  yes

Neutral

App. Schema DB

Heterogeneous Applications
  no

Copy source files to new platform
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.

Causes of Varying Data Formats

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 \texttt{int} data type is 2-byte alignment instead of 4-byte alignment:

\begin{verbatim}
struct X {
    char status;
    int value;
};
\end{verbatim}

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 that of the Intel microprocessor line. If an integer is written on one platform, its format might need to be converted before it is readable on another platform.

Other features that can cause data formats to vary are

- Virtual function table (vtbl) pointers
- Virtual base classes
- Zero-length base classes
Creating Identical Data Formats

When you run `ossg` 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 `ossg` 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:

```cpp
class X {
    char a;
    int i;
    char b;
    int j;
};
```

The best way to neutralize this code is to reorder the members:

```cpp
class X {
    int i;
    int j;
    char a;
    char b;
};
```

Because some applications depend on the order of members within a class, you decide whether 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 `ossize` 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 in the way compilers lay out such objects. For information about neutralizing virtual base classes, see Templated Virtual Base Classes on page 103 and Neutralizing the Allocation Order of Virtual Base Classes on page 104.

Primitive Data Types

The compiler support for Sun C++ and gcc3 includes 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.

In general, you should use the floating-point types float and double if you are concerned about heterogeneity issues.

64-Bit Pointers

If you want to neutralize a schema containing 64-bit pointers so that they work on a 32-bit platform, use 64-bit soft pointers and 32-bit soft pointers. For more information about using soft pointers, see the Advanced C++ API User Guide.

Pointers to Members

Pointers to function members (PTOMFs) are not supported on any platforms. Pointers to data members (PTODMs) are supported on all platforms — with the exception that the pointers must not point to members of virtual base classes.

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 and 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:

```cpp
template <class T> class Y
{
    public:
        char a;
        T b;
    }
Y<char> y_char;
Y<double> y_double;
```

The `Y<char>` template requires no additional padding for it 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 will not be used. Instead, a replacement class that you specify should be used. These replacement classes are known as template specializations. Following is an example of where one might be useful:

```cpp
/* 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;
    };
```

Following is the template specialization you would define for the example:

```cpp
class Y<double>
{
```
Chapter 5: Building Applications for Multiple Platforms

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. Because 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.

```
template <class T> class linked_list
{
    public:
        T* data;
        char flag;
        linked_lists<T*> next;
};
```

Used by this type.

```
linked_list<int>
```

Template specialization based on schema generator instructions. This provides a neutral storage format.

```
class linked_list <int> {
    public:
        T* data;
        char flag;
        char _os_pad_1[3];
        linked_lists<T*> next;
};
```

Alternatively, you can modify the template this way. Instantiations of this template would be neutral.

```
template <class T> class linked_list {
    public:
        T* data;
        char flag;
        char _os_pad_1[3];
        linked_lists<T*> next;
};
```

Sizes for Data Types

A neutralized schema requires that a data type be the same size on all platforms for which it has been neutralized. When the types involved are classes, the schema generator can determine the way 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,
**Restrictions**

`ossg` displays messages similar to the 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

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.

**32-Bit and 64-Bit Integral Types**

Within a given architectural set, different type names are used to denote integer types of the same size. For example, Solaris uses `long long` to mean 64-bit integers, Visual C++ uses `__int64`, while other 64-bit platforms use `long`. To handle these differences correctly, use the `-portable_type_name` option on the `ossg` command line and conditionally define a type to the correctly sized type for each platform. The syntax for this option is

```
-portable_type_name name size
```

where `name` is the name of your type and `size` is the type’s size in bytes. The option can be abbreviated `-ptn`.

**Example**

To use a 64-bit integer called `myInt64` on Solaris and Windows platforms, add the following `typedef` to your application code:

```c
#if defined __OS_SOL2
typedef  long long myInt64
#else
typedef __int64 myInt64
#endif
```
Then add the following to your `ossg` command line:

```
-portable_type_name myInt64 8
```

## `ossg` Neutralization Options

You neutralize a schema by running `ossg` with neutralization options. The `-arch set` option is required; the other options are not. This section describes the neutralization options.

**-arch set**

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 `set` to be one of the following architecture sets:

- `all32` — All 32-bit architectures
- `all64` — All 64-bit architectures
- A versioned architecture set supported by `ossg`; see Versioned Architecture Sets on page 93.
- A forced-order architecture set — for use when allocation order is a consideration when generating schema for applications that use virtual base classes. For more information, see Neutralizing the Allocation Order of Virtual Base Classes.
- A user-defined set specified by the `OS_USER_ARCH_SET` environment variable; see `OS_USER_ARCH_SET` in Chapter 3 of *Managing ObjectStore*.

You can use `ossg`’s `-showsets` option to list the platforms supported by any of the ObjectStore-defined architecture sets.

**-ignore_vbo**

Suppress any warnings about differences in the allocation order of virtual base classes. For more information, see Neutralizing the Allocation Order of Virtual Base Classes.

**-nout filename or -neutral_info_output filename**

Indicates the name of the file to which neutralization instructions are directed.
**ossg Neutralization Options**

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

- **nor** or **-noreorg**

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.

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

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**.

- **ptn name size** or **-portable_type_name name size**

  Specifies that a type, `name`, has a given `size`, in bytes, on all architectures in the architecture set specified with **-arch**.

  Usually `name` is conditionally defined in a `typedef` and it must be an integer type.

- **-sopt option_file** or **-schema_options option_file**

  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 consider them. See Compiler Option File Format on page 99 for details about the content of the option file.

Optional. No default.
Chapter 5: Building Applications for Multiple Platforms

-showsets
Lists all the names of architecture sets that can be specified as arguments to the -arch option. The listing also includes the names of platforms that are members of each set.

-showd -show_difference or -showw or -show_whole
Indicates the description level of the schema neutralization instructions. Optional. Default is -show_whole.

-wchar_t2 or -wchar_t4
Specifies the size of wchar_t to be either 2 or 4 bytes. If wchar_t on any of the platforms within the set specified by -arch differ in size, use a typedef to define them conditionally to the correct size.

Required if a schema includes a wchar_t.

Neutralizing the Schema

Follow these instructions to generate a neutral application schema:

1 Run the schema generator with neutralizer options to determine the changes in your source files (usually header files) that will 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 Invoking ossg to Generate a Schema on page 26. (Portions of the command line are on different lines only for clarity.)
Neutralizing the Schema

```
ossg [compilation_options] neutralizer_options
[other_schema_generator_options]
{-assf app_schema_source_file |
  -asof app_schema_object_file.obj}
-asdb app_schema.adb 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 go to step 5.

4 Continue to modify your source files according to ossg instructions, then run ossg until the schema generator produces your application schema successfully.

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 option whenever you run ossg. This ensures that changes do not cause platform conflicts.

Specifying the -arch set 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 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 option, the schema generator cannot determine that the alignment must be larger on certain platforms.
Versioned Architecture Sets

For most applications that require neutralization, it is sufficient to invoke `ossg -arch` with one of the architecture sets described in `ossg Neutralization Options` on page 89 — namely, `all32` or `all64`. However, the contents of these sets are subject to change from release to release, as support for a particular platform on ObjectStore is added or removed. And changes in platforms can require schema changes that the user may find undesirable. For example, a user who neutralizes with `ossg -arch all64` may find that, at a later release of ObjectStore, a platform was added to the `all64` set that results in schema changes requiring schema evolution.

If you know that you will always neutralize with the same architecture set as defined at a particular release and that you will not be adding more platforms at later releases, you can use one of the versioned architecture sets defined by ObjectStore. Unlike the `all32` and `all64` sets, a versioned set always includes the same set of platforms — and only those platforms — from release to release.

The name of a versioned set consists of one of the three standard sets (`all32` or `all64`), to which an ObjectStore release number is appended as a suffix. For example, the versioned set `all32_610` includes only those 32-bit platforms that were supported at Release 6.1 of ObjectStore.

You can use `ossg`'s `-showsets` option to list the ObjectStore-defined architecture sets, including all versioned sets, as well as the platforms defined for each set.

Additional Neutralization Considerations

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 4-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.
Neutralizing the Schema

In this case, you must also update the database schema and the data so they match the neutralized application schema. To do this, use the ossevol utility or a custom evolution application using the schema evolution library.

Command Line and Neutralization Examples

Following is a sample ossg command line on a Sun SPARC system:

```
ossg -arch all32 -padm -assf proj_schema.cc \
-asdb progschema.adb proj_schema_source.cc
```

The schema will be neutralized for use on platforms belonging to all32. Maximal padding will be done. Because 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.

Neutralization example

```
#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 all32 for schema neutralization. These examples assume that you are running ossg on a UNIX platform. To try these examples on a Windows NT platform, specify -asof instead of -assf. Also, on a Windows platform, the specification of the path for ObjectStore header files is -I%OS_ROOTDIR%\include, instead of the way it is in the examples.

```
ossg -assf hetero.cc -asdb hetero.adb -arch all32 \
-1%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_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.
Chapter 5: Building Applications for Multiple Platforms

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:

```cpp
#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 `ossg` 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` option specified:

```
<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:

```cpp
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();
};
```
Neutralizing the Schema

Next, modify the original `ossg` command line by adding the `-show_difference` option. The only effect that this has is on the way that the changes are presented:

```
ossg -assf hetero.assf -asdb hetero.adb -arch all32 \\
- show_difference -I$OS_ROOTDIR/include hetero.cc
```

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, this kind of display might be more easily understood.

For most classes, the default (`-show_whole`) display behavior is probably simpler.

Virtual base class example

Following is a more complicated class involving virtual base classes:

```
#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 `all32` architecture group.
Chapter 5: Building Applications for Multiple Platforms

Schema generator command line

```
ossg -assf hetero.assf -asdb hetero2.adb -arch all32 \
-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:

```c
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 the current example, you can see that the schema generator replaces the virtual base classes 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 classes introducing templates are named slightly differently.

The particular template that the schema generator chooses for a given virtual base class depends on the characteristics of the virtual base class 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 `ossg` 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 instructions for making neutralization changes to your source code.

```
all: my_exec
  CC -o my_exec main.o os_schema.o foo.o bar.o
  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.
    ossg -arch all32 -showd -mrscp -asdb my_exec.adb \
```

Release 6.1
Building a Heterogeneous Application from a Neutral Schema

After you neutralize a schema, you must follow these 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.

Listing Nondefault Object Layout Compiler Options

When you are building C++ applications, you might encounter circumstances in which 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 the default object layout required on all platforms in the application’s heterogeneity set. The schema generator uses the compiler options file when determining the changes that are necessary for schema neutralization. You use the `-schema_options options_file` command-line argument to tell `ossg` the schema options file to use.

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.
Chapter 5: Building Applications for Multiple Platforms

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:

  \[\text{compiler_spec} \mid \text{architecture_spec}\] \[(\text{class_list})\] \text{option}

  The \text{compiler_spec} variable indicates the compiler with which you are using the specified option.

  The \text{architecture_spec} variable indicates both the compiler and the platform that you are using. The possible values are in the following table.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Value of architecture_spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Windows NT Visual C++</td>
<td>intel_win32_msoft \ or \ visualc++</td>
</tr>
<tr>
<td>Intel Windows 98 Visual C++</td>
<td></td>
</tr>
<tr>
<td>HP-UX aC++</td>
<td>compiler_hpacc</td>
</tr>
</tbody>
</table>

  The \text{class_list} variable lists one or more classes that the specified compiler option or \text{pragma} operates on. The list can include any application classes 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 class names.

  The \text{option} variable can be a compiler switch or a \text{pragma} statement.

The option Variable

When the \text{option} variable is a compiler switch, it takes one of the following forms:

- \text{switch compiler\_switch}
- \text{switch compiler\_switch compiler\_switch\_value}
- \text{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.

<table>
<thead>
<tr>
<th>compiler_spec or architecture_spec</th>
<th>compiler_switch</th>
<th>compiler_switch_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>visualc++ or intel_win32_msoft</td>
<td>/vmb</td>
<td>Not applicable</td>
</tr>
<tr>
<td></td>
<td>/vmg</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/vms</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/vmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/vmv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/Zpn</td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>compiler_spec or architecture_spec</th>
<th>directive</th>
<th>directive_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>visualc++ or intel_win32_msoft</td>
<td>pack</td>
<td>(n)</td>
</tr>
<tr>
<td></td>
<td>pointers_to_members</td>
<td>best_case</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_generality,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>single_inheritance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_generality,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>multiple_inheritance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>full_generality,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtual_inheritance</td>
</tr>
</tbody>
</table>

Listing Nondefault Object Layout Compiler Options

Building ObjectStore C++ Applications
Overriding Options Within the Compiler Option File

You can specify compiler options and _pragma_ statements 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. Consider the following example of a compiler option file for Windows compilers:

```
intel_win32_msoft /Zp8
intel_win32_msoft (classX, classY) _pragma pack (4)
```

When this compiler option file is in effect, _ossg_ assumes that the Windows compiler aligns all types in the schema on 8-byte boundaries, except for _classX_ and _classY_, which the compiler aligns on 4-byte boundaries.

Sample Compiler Option File

```
visualc++ switch /Zp
intel_win32_msoft _pragma pointers_to_members best_case
visualc++ (ClassX) _pragma pack (1)
```

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 provided by the schema generator.

This section provides a description of the macros that the schema generator instructs you to use. An explanation of changes to templated virtual base classes is also included.

Some macros have an _(_x_)_ at the end. The neutralizer provides instructions for replacing the _x_ with a meaningful value.

Base Class Padding Macros

On platforms for which 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 warning message names the specific platforms. When you receive it, add the padding and run `ossg` again.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Inserts Padding to Compensate For</th>
</tr>
</thead>
<tbody>
<tr>
<td>os_base_pad_vftbl</td>
<td>4- or 8-byte vtbls placed at the start of an object</td>
</tr>
<tr>
<td>os_base_pad_vftbl8</td>
<td>4- or 8-byte vtbls placed at the start of an object</td>
</tr>
<tr>
<td>os_base_pad_vbptr(x)</td>
<td>4- or 8-byte virtual base class pointers placed at the start of an object</td>
</tr>
<tr>
<td>os_base_pad_vbptr8(x)</td>
<td>4- or 8-byte virtual base class pointers placed at the start of an object</td>
</tr>
<tr>
<td>os_base_pad_vbtbl</td>
<td>4-byte virtual base class tables placed at the start of an object</td>
</tr>
</tbody>
</table>

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 to determine how that macro would be expanded on other platforms.

The following table describes the member padding macros that ObjectStore provides:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Inserts Padding to Compensate For</th>
</tr>
</thead>
<tbody>
<tr>
<td>os_pad_vftbl_start</td>
<td>4- or 8-byte vtbls placed at the start of an object</td>
</tr>
<tr>
<td>os_pad_vftbl_start8</td>
<td>4- or 8-byte vtbls placed at the start of an object</td>
</tr>
<tr>
<td>os_pad_vftbl_end</td>
<td>4- or 8-byte vtbls placed at the end of an object</td>
</tr>
<tr>
<td>os_pad_vftbl_end8</td>
<td>4- or 8-byte vtbls placed at the end of an object</td>
</tr>
<tr>
<td>os_pad_vbtbl</td>
<td>8-byte vtbls in classes containing no data members</td>
</tr>
<tr>
<td>os_pad_vbtbl_only8</td>
<td>8-byte vtbls in classes containing no data members</td>
</tr>
<tr>
<td>os_pad_vbtbl1</td>
<td>4- or 8-byte virtual base class tables placed at the end of an object</td>
</tr>
<tr>
<td>os_pad_vbtbl8</td>
<td>4- or 8-byte virtual base class tables placed at the end of an object</td>
</tr>
</tbody>
</table>
Templated Virtual Base Classes

Class layout varies so widely 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 templated base classes to introduce virtual base classes. For example, suppose you begin with this class:

```cpp
class A : public virtual B {}
```

The schema generator might instruct you to convert this to something like the following line. This varies according to the platforms that are involved.

```cpp
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`; 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. Following are the class names:

- `os_vb_f<>`
- `os_vb_fs<>`
- `os_vb_fbs<>`
- `os_vb_fbsc<>`
- `os_vb_fbcsd<>`

### Table: Macro Inserts Padding to Compensate For

<table>
<thead>
<tr>
<th>Macro</th>
<th>Inserts Padding to Compensate For</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>os_pad_vbptr_start(x)</code></td>
<td>4- or 8-byte virtual base class 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.</td>
</tr>
<tr>
<td><code>os_pad_vbptr_start8(x)</code></td>
<td>4- or 8-byte virtual base class 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.</td>
</tr>
<tr>
<td><code>os_pad_mem_ptr8(x)</code></td>
<td>Data member pointers that are 8, 12, or 16 bytes long.</td>
</tr>
<tr>
<td><code>os_pad_mem_ptr12(x)</code></td>
<td></td>
</tr>
<tr>
<td><code>os_pad_mem_ptr16(x)</code></td>
<td></td>
</tr>
</tbody>
</table>
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. Because 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 class 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.

### Neutralizing the Allocation Order of Virtual Base Classes

On some platforms, classes that use virtual base classes are allocated in a different order from that used by other platforms. Consider the following example of declarations of classes that use virtual base classes:

```cpp
class A {};  
class B {};  
class C {};  
class D {};  
class E : virtual public A, virtual public B {};  
class F : virtual public C, virtual public D {};  
class G : virtual public E, virtual public F {};  
```

All platforms will allocate classes \( E \) and \( F \) in the following order:

- \( E \): A, B  
- \( F \): C, D  

Most platforms will allocate class \( G \) in post traversal order:

- \( G \): A, B, E, C, D, F  

Some platforms, however, will allocate \( G \) in inheritance order:

- \( G \): E, A, B, F, C, D  

In a neutralized environment with heterogeneous access, this difference in allocation order poses a problem because the virtual bases are not allocated in the same order. Thus, user data is not at the same offset on all platforms.

If you are neutralizing against platforms that use a different order when allocating classes that use virtual base classes, `ossg` will emit detailed instructions for editing the schema source file to ensure the same allocation order across all platforms in the architecture set. Specifically, the instructions will tell the user to substitute “forced-order” base classes for any virtual base
classes that have virtual base classes. The instructions will include the replacement forced-order classes as well as helper classes (os_fo<>) that are used as template specializations.

The names of the forced-order classes are the same as those described in Templated Virtual Base Classes on page 103, except that each is appended with the _fo suffix, as follows:

- os_vb_f_fo<>  
- os_vb_fs_fo<>  
- os_vb_fbs_fo<>  
- os_vb_fbsc_fo<>  
- os_vb_fbscd_fo<>  

To return to the original example, after the schema source file has been edited to include the forced-order base classes, instances of class G will be allocated in post traversal order on all platforms.

Instead of using the forced-order classes to neutralize virtual base classes, you may be able to use the all32vbtrav architecture set. For more information, see Growth Resulting from the Forced-Order Classes on page 106.

Note

The ossg utility will continue to emit warning messages concerning differences in allocation order even after the forced-order base classes have been added. To suppress the messages, invoke ossg with the -ignore_vbo option. For information about this and other ossg options, see Invoking ossg to Generate a Schema on page 26.

Database Growth

The your database can grow in size as a result of the following:

- Padding added by the compiler
- Growth that results from the use of the forced-order classes during neutralization

The following sections discuss both sources of database growth.

Growth Resulting from Padding

Database growth as a result of padding source files is hard to predict. Whether there is any growth depends on the following:

- The kinds of data and structures in the schema
Endian Types for ObjectStore Platforms

- The platforms that the application can run on

The schema databases supplied with ObjectStore hardly ever grow as a result of padding. Applications that include virtual base classes and applications but not the os_virtual_behavior base class are more likely to grow.

One way to try to determine any effect on database size is to neutralize the schema, 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 size of the database to the size of an equivalent preneutralized database.

Growth Resulting from the Forced-Order Classes

A side-effect of using the forced-order classes to normalize allocation order is that they can cause your database to grow in size. As of Release 6.1 of ObjectStore, the only platform that uses inheritance order when allocating virtual base classes is linux3 (gcc3). If you are neutralizing against 32-bit platforms but not linux3, you can reduce growth of the database by excluding linux3 from the 32-bit architecture set when generating the neutralized schema.

To reduce database growth, specify all32vbtrav as the architecture set when invoking ossg -arch. This architecture set includes only those 32-bit platforms that use post traversal order when allocating virtual base classes — that is, all 32-bit platforms except linux3. You can also use the all32vbinherit architecture set when neutralizing against only 32-bit platforms that use inheritance order when allocating virtual base classes — currently, linux3.

Endian Types for ObjectStore Platforms

Endian type specifies whether the high-order byte or the low-order byte is first. This information is provided as general information. You need not 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 the way to lay out objects in a way that can be used by all machines involved.

The following lists show the endian type for each ObjectStore platform. In a big-endian type, the high-order bit is first.

Big-Endians
- HP
- Sun SPARC

Little-Endians
- Intel (Windows and linux)
Endian Types for ObjectStore Platforms
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. ObjectStore / Single does not support rawfs.

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

The topics discussed in this chapter are

ObjectStore / Single Features 110
Application Development Sequence 112
Server Log Propagation 113
Remote Access 115
Packaging ObjectStore / Single Applications 116

ObjectStore/Single Features

Each invocation of an ObjectStore / Single application requires a server log file. Applications for UNIX platforms also require a cache file. ObjectStore / Single applications normally supply these files automatically.

ObjectStore/Single API

The following functions in the class objectstore can be used in creating ObjectStore / Single applications. The C++ API Reference describes these functions in detail.

- objectstore::embedded_server-available()
- objectstore::get_cache_file() on page 26 — UNIX only
- objectstore::get_log_file()
- objectstore::get_propagate_on_commit()
- objectstore::network_servers-available()
- objectstore::propagate_log()
- objectstore::set_cache_file()— UNIX only
- objectstore::set_log_file()
- objectstore::shutdown()
- objectstore::set_propagate_on_commit()

ObjectStore/Single Utilities

The following ObjectStore utilities can be used with ObjectStore / Single. The following sections in Managing ObjectStore describe each utility in detail.

- osaffiliate: Changing Database Affiliations
- oschgrp: Changing Database Group Names
- oschmod: Changing Database Permissions
Chapter 6: Working with ObjectStore/Single

- oschown: Changing Database Owners
- oscompact: Compacting Databases
- osexschm: Displaying Class Names in a Schema
- oshostof: Displaying Database Host Name
- osls: Displaying Directory Content
- osmkdir: Creating a Rawfs Directory
- osprop: Propagating Server Logs
- osrm: Removing Databases and Rawfs Links
- osrmdir: Removing a Rawfs Directory
- osscheq: Comparing Schemas
- ossetasp: Patching Executables with Application Schema Path Names
- ossetrsp: Setting a Remote Schema Path Name
- ossevol: Evolving Schemas
- ossize: Displaying Database Size
- osverifydb: Verifying Pointers and References in a Database
- osversion: Displaying the ObjectStore Version

Use the standard operating system copy facilities instead of the oscopy utility to copy ObjectStore/Single databases.

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:

<table>
<thead>
<tr>
<th>Platform</th>
<th>Library Load Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>$OS_ROOTDIR/libsngl:/$OS_ROOTDIR/lib</td>
</tr>
<tr>
<td>Windows</td>
<td>%OS_ROOTDIR%/binsngl</td>
</tr>
</tbody>
</table>
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 117 for a discussion of the issues involved in using these environment variables.

The two environment variables are

- `OS_CACHE_FILE`
- `OS_LOG_FILE`

You can use the `OS_CACHE_FILE` environment variable 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. If you use neither the environment variable nor `objectstore::set_cache_file()`, ObjectStore / Single applications supply a default file named `objectstore_6_cache` in the current working directory.

You can use the `OS_LOG_FILE` environment variable early in development, instead of the 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. If you use neither the environment variable nor `objectstore::set_log_file()`, ObjectStore / Single applications supply a default file named `tplog.odi` in the current working directory of the program.

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 The Server Transaction Log in Chapter 1 of Managing ObjectStore for additional information about server logs.

In this discussion, the term server refers to both the separate oserver process of full ObjectStore and the functionally equivalent part of ObjectStore / Single 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, 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 oscopy.

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 general rule, 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 generally not a problem for full ObjectStore because there is only one log file per oserver process. However, in ObjectStore / Single, because
Server Log Propagation

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 work with logs:

- By default, ObjectStore / Single forces the log to be propagated immediately when a transaction commits. The result is that the period of time when there is committed data in the log is usually quite short. This behavior can be overridden as described in `OS_DISABLE_PROPAGATE_ON_COMMIT`, following.

- During startup, 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 `ossvrshrd` is executed.

- The utility `osprop` propagates data from the log files named in the command invocation. Following successful propagation, the log files are removed automatically.

- The other utilities that provide meaningful information as ObjectStore / Single tools (`ossize` 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.

OS_DISABLE_PROPAGATE_ON_COMMIT

The environment variable `OS_DISABLE_PROPAGATE_ON_COMMIT` controls when ObjectStore propagates committed data from the transaction log to a database. By default, this environment variable is false and data is propagated to a database when a top-level transaction commits. If this environment variable is set to true, the data is propagated when a database is closed and when an application ends normally. If an application ends prematurely, some of the data might not have been propagated to the database. This data is propagated the next time ObjectStore is initialized or on a call to `propagate_log()`.
For some applications, setting this environment variable to true can improve performance because data is written to the database less frequently. However, it increases the risk of data loss, which can happen if an application ends prematurely and the transaction log is deleted before the data is propagated to the database.

Applications can override this environment variable by using `objectstore::set_propagate_on_commit()`. For more information, see `objectstore::get_propagate_on_commit()` and `objectstore::set_propagate_on_commit()` in the C++ API Reference.

When to Intervene

An ObjectStore/Single application (or any of the ObjectStore/Single utilities) tries to ensure that all the data in its server log is propagated and that the log is removed before the application exits.

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 keep in mind that any time an ObjectStore/Single application or utility is initialized with an existing server log, ObjectStore conducts propagation automatically. The `osprop` utility is the simplest possible ObjectStore/Single application — it starts, does propagation, then shuts down.

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.

Remote Access

ObjectStore/Single allows remote access to databases through NFS. The path name lookup algorithm differs from the one full ObjectStore uses in that
it does not consider remote mount points when expanding paths, so all paths appear to be local.

The different path name 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

If you want to prevent an ObjectStore/Single application from accessing databases on remote file systems, set the `OS_DISALLOW_OSSINGLE_REMOTE_DATABASES` environment variable to any nonnull value except 0. See also `OS_DISALLOW_OSSINGLE_REMOTE_DATABASES` in *Managing ObjectStore*.

### 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.

**UNIX**

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 removed inadvertently
Chapter 6: Working with ObjectStore/Single

Additional Considerations

The ObjectStore / Single environment variables (OS_LOG_FILE and OS_CACHE_FILE) should be used only 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 other ObjectStore / Single applications 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 because there are the additional factors of user permissions and server log file persistence. See the following sections for a discussion of what your application must consider in selecting a cache and log file.

Cache File Considerations

When designing and deploying an ObjectStore/Single application on UNIX platforms, pay particular attention to the cache file in terms of the following:

- 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 to 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 the way the log file name should be communicated to an external agent in case recovery is needed
- Specifying the way 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 You Should 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 before your application installation.

Contact Technical Support if you have recommendations about ways in which the current cache file and server log API could be enhanced to ease this requirement.

Packaging an ObjectStore/Single Application

UNIX

If you choose to package the ObjectStore/Single components with your own libraries, you must include the ObjectStore/Single libraries from libsgl (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.

Windows

On Windows NT, all the ObjectStore/Single components are installed in the \binsgl directory. Choose the DLLs and ObjectStore utilities that 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 117.
Index

A
allocation order 104
application schema databases
description 2
option to ossg 31
specification in makefiles 57
application schema object files
description 2
option to ossg 31
application schema source files
description 2
option to ossg 31
application schemas
C preprocessor, changing default 33
comparison with other schemas 48
database schemas, compatibility with 25
description 22
generating 35
generating from a compilation schema 41
moving 71
multiple applications, using same for 36
ossg command-line examples 37
ossg, invoking 26
applications
debugging 76
heterogeneous 80
moving 71
third-party 5
Windows
porting to 68
-arch option to ossg
generating schema 32
neutralization 89
architecture sets 89
versioned 93
architectures
code specifications 99
neutralization options 89
neutralization, specifying for 32
-asdb option to ossg 31
-asof option to ossg 31
-assf option to ossg 31

B
breakpoints
Windows application abnormal exit 65
building heterogeneous applications 98
building ObjectStore applications
compiling 53
debugging 53
flow chart 4
generating schemas 21
linking 53
neutralizing schemas 91
overview 1
server involvement 5

C
C preprocessor
  defaults 33
C++ compiler
  schema source file contents 18
  changing default preprocessor 33
  cl preprocessor 33
ClearCase 5
coll.hh header file
  when to use 10
  with os_Collection_declare macro 12
collections
  coll.hh header file 10
  template class, undefined error report 10
  template instantiation problem 10
  UNIX, order for specifying libraries 54
compact.hh header file
  when to use 10
compactor
  compact.hh header file 10
  UNIX, order for specifying libraries 54
compatibility
  application and database schemas 25
  compilation options
    neutralizing with nondefault layouts 98
to osg
  application schemas 27
compilation schemas
  comparison with other schemas 48
description 23
generating 40
specifying when invoking osg 31
compiler options
  Windows 60
compiler options files
  description 98
  format 99
  overriding options 101
  pragmas 100
  switches 99
compilers
  third-party 4
  crash recovery 7
creating schema source files
  instructions 15
-csdb option to osg 31

D
data formats 83
database growth, reducing 105
database schemas
  comparison with other schemas 48
  compatibility with application schemas 25
description 23
  remote 24
database utilities
  dbutil.hh header file 10
  UNIX, order for specifying libraries 54
databases
  See also protected databases
dbutil.hh header file
  when to use 10
debug build 69
debugging
  overview 76
  Windows 64
defaults
  C preprocessor 33
DLLs
  compiling 61
debug versions
    Windows 64
  linking 62
  load path
    ObjectStore/Single 109
  using 7
drop-in build 69
dynamic libraries
   See DLLs

E
endian types 106
environment variables
   OS_CACHE_FILE
      ObjectStore / Single 112
   OS_COMP_SCHEMA_CHANGE_ACTION
      example 45
   OS_LOG_FILE
      ObjectStore / Single 112
   OS_TRACE_MISSING_VTBLS
      debugging applications 76
      Windows 63
exception handling
   Windows compiler options 60

F
@filename option to ossg 33
forced-order base classes 104

G
generating schemas
   application 35
   compilation 40
   library 38
   overview 22
   vtbls
      example of making available 34
      making available 30
   growth of database 105

H
header files
   coll.hh 18
      required order 13
      when to use 10
   compact.hh

   when to use 10
   dbutil.hh 18
      when to use 10
   manschem.hh 18
      creating schema source files 16
   mop.hh
      required order 13
      when to use 10
   ObjectStore 10
   ostore.hh 18
      when to use 10
   ostore/manschem.hh
      when to use 10
   relat.hh
      when to use 10
      required order 12
   Rogue Wave on Solaris 50
   sample #include statements 17
   schmdefs.hh 18
   schmevol.hh
      required order 13
      when to use 10
   semoptwk.hh
      when to use 12
      user-defined 9
heterogeneity 82
heterogeneous applications
   building 98
   description 80
   description of schema generator
      instructions 101
   endian types 106
   neutralization options 89
   neutralizing schema
      instructions 91
   nondefault object layout options 98
   restrictions 85

I
-I- option to ossg
   application schemas 29
IDE for Visual C++
makefiles 59

-ignore_vbo option to ossg

generating schema 32

INCLUDE environment variable

Windows 63

include paths

-no_default_includes option to ossg 29

specifying to ossg 27

indexes

UNIX, order for specifying libraries 55

inline virtual functions

missing vtbls 75

Integrated Development Environment

See IDE for Visual C++

L

LIB environment variable

Windows 63

libos library order 54

liboscmp library 54

liboscol library order 54

libosdbu library order 54

libosmop library order 54

libosqry library order 55

libosstd library order 55

libostcl library order 54

libosthr library order 54

libraries 54

third-party 5

UNIX

specifying 54

Windows

requirement 59

standard run-time 61

library schemas

comparison with other schemas 48

description 23

generating 38

specifying when invoking ossg 31

link order 54

linking

UNIX 54

Windows 62

log files, ObjectStore/Single

forcing data propagation for

applications 114

issues 117

long double data type, support for 85

long long type support 42

-Isdb option to ossg 31

M

macro arguments

specifying to ossg 27

macros, system-supplied

base class padding 101

member padding 102

_ODI_OSSG_ 41

os_Collection_declare 11

os_Collection_declare_no_class 11

os_Collection_declare_ptr_tdef 12

OS_MARK_SCHEMA_TYPE() 16

OS_MARK_SCHEMA_TYPESPEC() 17

-make_reachable_library_classes_

persistent option to ossg 28

-make_reachable_source_classes_

persistent option to ossg 28

application schemas 28

whether to mark 15

makefiles

compilation schema generation 40

library schema generation 38

neutralizing schemas 97

retrofitting 77

tabs and spaces 57

UNIX

building compilation schemas 58

description 56

template 57

Windows example 63

Windows IDE 59

Building ObjectStore C++ Applications
manschem.hh header file
    creating schema source files 16
marking types
    how to 16
    reason for 13
-MD option to Visual C++ 60
memory
    access violations
        Windows debugger 64
metaobject protocol
    mop.hh header file 10
    UNIX, specifying libraries 55
MOP
    See metaobject protocol
mop.hh header file
    when to use 10
moving application schemas 71
-mrlcp option to ossg 28
-mrscp option to ossg
    application schemas 28
    comparison with marking types 15
msvcrt.lib run-time library 61
multithreaded applications on Solaris
    compiling and linking 56
    debugging 56

N
-no_default_includes option to ossg
    generating schema 32
    neutralization 89
neutralization
    layout order 104
neutralizing inheritance paths 103
neutralizing schemas
    class instantiations 86
    considerations 93
    definition 80
    endian types 106
    examples 94
    instructions 91
    makefiles 97
    nondefault object layout options 98
    options 89
    how ossg creates identical formats 83
    restrictions 85
    schema generator instructions
        description 101
NFS mounts
    building applications on Windows 68
    -no_default_includes option to ossg
        application schemas 29
    -no_weak_symbols option to ossg 29
    -nor option to ossg 90
    -noreorg option to ossg 90
    -nout option to ossg 89
O
ObjectStore stand-alone
    See ObjectStore / Single
ObjectStore / Single
    API 110
    application packaging 118
    definition 109
    end-user information 118
    interoperability with full
        ObjectStore 116
    libraries 7
    log files 114
    remote database access 116
    server log files 116
    use of NFS with 116
    using environment variables 117
    utilities 110
objectstore, the class
    functions that can be used with
        ObjectStore / Single 110
    _ODI_OSSG_, the macro 41
os_Array, the class
    undefined error 10
os_Bag, the class
    undefined error 10
os_base_pad_XXX macros 102
OS_CACHE_FILE environment variable
use in developing ObjectStore/Single applications 112
use with objectstore::set_cache_file 112

os_Collection, the class
undefined error 10

os_Collection_DECLARE, the macro
description 11
example 12

os_Collection_DECLARE_NO_CLASS, the macro
11

os_Collection_DECLARE_PTR_TDEF, the macro 12

OS_COMP_SCHEMA_CHANGE_ACTION
environment variable
example 45

os_CString, the class 67

os_Cursor, the class
undefined error 10

os_List, the class
undefined error 10

OS_LOG_FILE environment variable
and objectstore::set_log_file() 112

OS_MARK_SCHEMA_TYPE(), the macro
creating schema source files 16

OS_MARK_SCHEMA_TYPESPEC(), the macro
creating schema source files 17

OS_OSSG_CPP environment variable
C preprocessor
changing 33
os_pad_XXX macros 102

OS_ROOTDIR environment variable
Windows 63
%OS_ROOTDIR\include directory 60

os_Set, the class
undefined error 10

OS_TRACE_MISSING_VTBLS environment variable
documentation 86

debugging applications 76
os_vb_virtual base introduction
templates 103
ossetasp utility 71
ossg utility
@filename option 33
comparison of command lines 47
generating application schemas 26
generating compilation schemas 40
neutralization options 89
nondefault object layout options 98
options 26
platforms in architecture sets 32
temporary file, using to send arguments 33
UNIX, building compilation schemas 58

ostore.hh header file
when to use 10

ostore.lib library
how to link with 62
Windows, linking 59

ostore/manschem.hh header file 10

P
-pad_consistent option to ossg
generating schema 32
neutralization 90
-pad_maximal option to ossg
generating schema 32
neutralization 90
-padc option to ossg 90
padding 105
padding macros
base class 101
database growth 105
dynamically defined 102
members 102
-padm option to ossg 90
parameterized classes
neutralizing schemas 86
PATH environment variable

124

Building ObjectStore C++ Applications
Windows 63
persistence
allowing allocation of reachable types
application schemas 28
platforms
neutralization 89
platforms specifying architecture sets 32
pointer restrictions 85
-portable_type_name option to ossg
generating schema 33
neutralization 90
preprocessors
options, specifying 27
propagation
ObjectStore / Single log files 113
protected databases
generating schemas for 49
-ptn option to ossg
generating schema 33
neutralization 90

Q
queries
UNIX, order for specifying libraries 55

R
reachable types
specifying -mrlcp 28
specifying -mrscp
application schemas 28
relat.hh header file
when to use 10
relationships
relat.hh header file 10
UNIX, order for specifying libraries 55
relocating vtbls 73
Rogue Wave header file problem 50
-rtdp option to ossg
how to use 30
example 34
RTTI (run-time type information) 76
run-time type information 76
--runtime_dispatch option to ossg
how to use 30
example 34

S
schema databases
description 22
schema evolution
ostore/manschem.hh header file 10
schmefol.hh header file 10
UNIX, order for specifying libraries 55
schema generation
See generating schemas
schema generator
See ossg utility
schema source files
creating 15
description 13
marking types 14
specifying for ossg
application schemas 26
using more than one 39
-schema_options option to ossg
description 33, 90
how to use 98
schemas
See also application schemas
See also compilation schemas
See also component schemas
See also database schemas
See also generating schemas
See also library schemas
See also neutralizing schemas
comparison of kinds 48
database schema
updating 93
hiding code from ossg 41
installation 24
introduction 22
metaschema mismatch errors 45
moving 71
neutralization
definition 80
eexamples 94
instructions 91
ossg run-time errors 45
persistent allocation errors 44
protected databases 49
type mismatch errors 44
vtbls, missing 75

template specializations 86

templates
collection classes instantiation
problem 10
nonvirtual instantiations 97
virtual base templates 103
when to modify 87
third-party applications, compilers, and
libraries 4

tthreads
compiling and linking on Solaris 56
UNIX, libraries 54
Visual C++ 68
troubleshooting
Rogue Wave header file problem 50
Windows
abnormal application exit 65
symbols, missing 73
types
how to mark 16
long double, support for 85

Building ObjectStore C++ Applications
Index

long long 42
reachable 14
which to mark 14

U
UNIX
libraries
  specifying 56
library schemas
  specifying 56
linking 54
makefile for building compilation
  schema 58
makefile template 56
ObjectStore libraries 54
signal handlers 57
vtbl pointers 58

V
versioned architecture sets 93
virtual base classes 104
  neutralization example 96
  neutralization restrictions 85
  neutralizing inheritance paths 103
  neutralizing layout order 104
virtual file systems 5
virtual functions
  missing vtbls 75
  vtbls 72
Visual C++
  application schema object file 81
double data type 69
long double data type 69
required options 60
threads 68
vtbls
  inline virtual functions 75
  making them available 30
    example 34
  missing 75
notification about missing ones 29
relocation 73
UNIX, shared libraries 58

W
-wchar_t2 option to ossg 91
-wchar_t4 option to ossg 91
-weak_symbols option to ossg 31
Windows
  abnormal application exit 65
  applications, porting to 68
  breakpoints, setting 65
  debugging 64
  include directory 60
  libraries, run-time 61
  library, required 59
  linking 62
  makefile, sample 63
  ossg, running within IDE 59
  symbols, missing 73
Windows NT
  architecture code 99

Z
-Zn options to Visual C++ 61

Release 6.1