Oracle® C++ Call Interface

Programmer's Guide

Release 9.0.1

June 2001

Part No. A89860-01

ORACLE®

Oracle C++ Call Interface Programmer’s Guide, Release 9.0.1

Part No. A89860-01

Copyright © 2001, Oracle Corporation. All rights reserved.

Primary Authors: Den Raphaely, Joan Gregoire

Contributors: Sandeepan Banerjee, Krishna Itikarlapalli, Maura Joglekar, Ravi Kasamsetty, Roopa Kesari, Srinath Krishnaswamy, Shoaib Lari, Geoff Lee, Jack Melnick, Gayathri Priyalakshmi, Rajiv Ratnam, Ashok Shivarudraiah, Rekha Vallam.

Graphic Designer: Valarie Moore.

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle Corporation disclaims liability for any damages caused by such use of the Programs.

Oracle is a registered trademark, and Oracle C++ Call Interface, Oracle Call Interface, PL/SQL, Pro*C, Pro*C/C++, Oracle Net, and Trusted Oracle are trademarks or registered trademarks of Oracle Corporation. Other names may be trademarks of their respective owners.

Contents

Send Us Your Comments xxiii

Preface xxv

Part I OCCI Programmer's Guide

1 Introduction to OCCI

Overview of OCCI 1-2
Benefits of OCCI 1-2
Building an OCCI Application 1-3
Functionality of OCCI 1-4
Procedural and Nonprocedural Elements 1-4
Processing of SQL Statements 1-5
Data Definition Language Statements 1-5
Control Statements 1-6
Data Manipulation LanguageSQL Statements 1-6
Queries 1-6
Overview of PL/SQL 1-7
Special OCCI/SQL Terms 1-8
Object Support 1-9
Client-Side Object Cache 1-9
Runtime Environment for Objects 1-10
Associative and Navigational Interfaces 1-10
Metadata Class 1-11

Object Type Translator Utility 1-11

2 Relational Programming

Connecting to a Database 2-2
Creating and Terminating an Environment 2-2
Opening and Closing a Connection 2-3
Creating a Connection Pool 2-3
Executing SQL DDL and DML Statements 2-6
Creating a Statement Handle 2-6
Creating a Statement Handle to Execute SQL Commands 2-6
Reusing a Statement Handle 2-7
Terminating a Statement Handle 2-8
Types of SQL Statements in the OCCI Environment 2-8
Standard Statements 2-9
Parameterized Statements 2-9
Callable Statements 2-10
Streamed Reads and Writes 2-12
Modifying Rows Iteratively 2-12
Executing SQL Queries 2-13
Result Set 2-14
Specifying the Query 2-15
Optimizing Performance by Setting Prefetch Count 2-15
Executing Statements Dynamically 2-16
Status Definitions 2-17
Committing a Transaction 2-20
Error Handling 2-20
Null and Truncated Data 2-21
Advanced Relational Techniques 2-23
Utilizing a Shared Server Environment 2-23
Optimizing Performance 2-27

3 Object Programming

Overview of Object Programming 3-2
Working with Objects in OCCI 3-2
Persistent Objects 3-3

Transient Objects 3-4
Values 3-5
Representing Objects in C++ Applications 3-5
Creating Persistent and Transient Objects 3-5
Creating Object Representations using the OTT Utility 3-6
Developing an OCCI Object Application 3-7
Basic Object Program Structure 3-7
Basic Object Operational Flow 3-8
Overview of Associative Access 3-11
Using SQL to Access Objects 3-12
Inserting and Modifying Values 3-12
Overview of Navigational Access 3-13
Retrieving an Object Reference (REF) from the Database Server 3-13
Pinning an Object 3-14
Manipulating Object Attributes 3-15
Marking Objects and Flushing Changes 3-15
Marking an Object as Modified (Dirty) 3-15
Recording Changes in the Database 3-15
Overview of Complex Object Retrieval 3-16
Retrieving Complex Objects 3-17
Prefetching Complex Objects 3-19
Working with Collections 3-19
Fetching Embedded Objects 3-20
Nullness 3-21
Using Object References 3-22
Freeing Objects 3-22
Type Inheritance 3-22
Substitutability 3-24
NOT INSTANTIABLE Types and Methods 3-24
OCCI Support for Type Inheritance 3-25
OTT Support for Type Inheritance 3-25
A Sample OCCI Application 3-26

4 Datatypes

Overview of Oracle Datatypes 4-2

OCCI Type and Data Conversion 4-2
Internal Datatypes 4-3
Character Strings and Byte Arrays 4-4
Universal Rowid (UROWID) 4-4
External Datatypes 4-5
Description of External Datatypes 4-8
Data Conversions 4-20
Data Conversions for LOB Datatypes 4-22
Data Conversions for Date, Timestamp, and Interval Datatypes 4-22

5 Introduction to LOBs

Overview of LOBs 5-2
Internal LOBs (BLOBs, CLOBs, and NCLOBs) 5-2
External LOBs (BFILEs) 5-3
LOB Values and Locators 5-3
LOB Classes and Methods 5-4
Creating LOBs 5-7
Opening and Closing LOBs 5-8
Reading and Writing LOBs 5-10
Improving Read and Write Performance 5-14
Updating LOBs 5-15
Objects with LOB Attributes 5-16
Persistent Objects with LOB Attributes 5-16
Transient Objects with LOB Attributes 5-17

6 Metadata

Overview of Metadata 6-2
Notes on Types and Attributes 6-3
Describing Database Metadata 6-3
Metatdata Code Examples 6-5
Attribute Reference 6-9
Parameter Attributes 6-10
Table and View Attributes 6-11
Procedure, Function, and Subprogram Attributes 6-12
Package Attributes 6-13

Type Attributes 6-13
Type Attribute Attributes 6-15
Type Method Attributes 6-16
Collection Attributes 6-17
Synonym Attributes 6-19
Sequence Attributes 6-19
Column Attributes 6-20
Argument and Result Attributes 6-21
List Attributes 6-23
Schema Attributes 6-24
Database Attributes 6-24

7 How to Use the Object Type Translator Utility

Overview of the Object Type Translator Utility 7-2
How to Use the OTT Utility 7-2
Creating Types in the Database 7-10
Invoking the OTT Utility 7-10
Specifying OTT Parameters 7-10
Invoking the OTT Utility on the Command Line 7-12
Overview of the INTYPE File 7-14
OTT Utility Datatype Mappings 7-16
OTT Type Mapping Example for C++ 7-24
Overview of the OUTTYPE File 7-27
The OTT Utility and OCCI Applications 7-28
OTT Utility Parameters for C++ 7-30
OTT-Generated C++ Classes 7-31
Map Registry Function 7-47
Extending OTT C++ Classes 7-48
Example for Extending OTT Classes 7-49
Example OCCI Application 7-60
OTT Utility Reference 7-88
OTT Command Line Syntax 7-88
OTT Utility Parameters 7-90
Where OTT Parameters Can Appear 7-97
Structure of the INTYPE File 7-98

Nested #include File Generation 7-100
SCHEMA_NAMES Usage 7-103
Default Name Mapping 7-105
Restriction Affecting the OTT Utility: File Name Comparison 7-107

Part II OCCI API Reference

8 OCCI Classes and Methods

Summary of OCCI Classes 8-2
OCCI Classes and Methods 8-3
Bfile Class 8-5
Summary of Bfile Methods 8-6
close() 8-6
closeStream() 8-6
fileExists() 8-7
getDirAlias() 8-7
getFileName() 8-7
getStream() 8-8
isInitialized() 8-8
isNull() 8-8
isOpen() 8-9
length() 8-9
open() 8-9
operator=() 8-9
operator==() 8-10
operator!=() 8-10
read() 8-11
setName() 8-12
setNull() 8-12
Blob Class 8-13
Summary of Blob Methods 8-14
append() 8-14
close() 8-15
closeStream() 8-15
copy() 8-15

getChunkSize() 8-17
getStream() 8-17
isInitialized() 8-17
isNull() 8-18
isOpen() 8-18
length() 8-18
open() 8-18
operator=() 8-19
operator==() 8-19
operator!= () 8-20
read() 8-20
setEmpty() 8-21
setNull() 8-21
trim() 8-21
write() 8-22
writeChunk() 8-23
Bytes Class 8-24
Summary of Bytes Methods 8-24
byteAt() 8-24
getBytes() 8-25
isNull() 8-26
length() 8-26
setNull() 8-26
Clob Class 8-27
Summary of Clob Methods 8-28
append() 8-29
close() 8-29
closeStream() 8-29
copy() 8-29
getCharSetForm() 8-31
getCharSetId() 8-31
getChunkSize() 8-31
getStream() 8-31
isInitialized() 8-32
isNull() 8-32

isOpen() 8-33
length() 8-33
open() 8-33
operator=() 8-33
operator==() 8-34
operator!=() 8-34
read() 8-35
setEmpty() 8-36
setNull() 8-36
trim() 8-36
write() 8-37
writeChunk() 8-38
Connection Class 8-40
Summary of Connection Methods 8-40
changePassword() 8-41
commit() 8-41
createStatement() 8-41
flushCache() 8-42
getClientCharSet() 8-42
getClientNCHARCharSet() 8-42
getMetaData() 8-42
getOCIServer() 8-43
getOCIServiceContext() 8-43
getOCISession() 8-43
rollback() 8-43
terminateStatement() 8-44
ConnectionPool Class 8-45
Summary of ConnectionPool Methods 8-45
createConnection() 8-46
createProxyConnection() 8-46
getBusyConnections() 8-47
getIncrConnections() 8-47
getMaxConnections() 8-47
getMinConnections() 8-48
getOpenConnections() 8-48

getPoolName() 8-48
getTimeOut() 8-48
setErrorOnBusy() 8-48
setPoolSize() 8-49
setTimeOut() 8-49
terminateConnection() 8-50
Date Class 8-51
Summary of Date Methods 8-52
addDays() 8-53
addMonths() 8-54
daysBetween() 8-54
fromBytes() 8-54
fromText() 8-55
getDate() 8-55
getSystemDate() 8-56
isNull() 8-57
lastDay() 8-57
nextDay() 8-57
operator=() 8-57
operator==() 8-58
operator!=() 8-58
operator>() 8-59
operator>=() 8-59
operator<() 8-60
operator<=() 8-60
setDate() 8-61
setNull() 8-61
toBytes() 8-61
toText() 8-62
toZone() 8-62
Environment Class 8-64
Summary of Environment Methods 8-64
createConnection() 8-65
createConnectionPool() 8-65
createEnvironment() 8-66

getCurrentHeapSize () 8-67
getMap() 8-67
getOCIEnvironment() 8-68
terminateConnection () 8-68
terminateConnectionPool() 8-68
terminateEnvironment() 8-68
IntervalDS Class 8-70
Summary of IntervalDS Methods 8-72
fromText() 8-73
getDay() 8-73
getFracSec() 8-74
getHour() 8-74
getMinute() 8-74
getSecond() 8-74
isNull() 8-74
operator*() 8-75
operator*=() 8-75
operator=() 8-75
operator==() 8-76
operator!=() 8-76
operator/() 8-77
operator/=() 8-77
operator>() 8-77
operator>=() 8-78
operator<() 8-78
operator<=() 8-79
operator-() 8-79
operator-=() 8-80
operator+() 8-80
operator+=() 8-80
set() 8-81
setNull() 8-81
toText() 8-82
IntervalYM Class 8-83
Summary of IntervalYM Methods 8-84

- fromText() 8-85
- getMonth() 8-86
- getYear() 8-86
- isNull() 8-86
- operator*() 8-87
- operator*=() 8-87
- operator=() 8-87
- operator==() 8-88
- operator!=() 8-88
- operator/() 8-89
- operator/=() 8-89
- operator>() 8-90
- operator>=() 8-90
- operator<() 8-91
- operator<=() 8-91
- operator-() 8-92
- operator-=() 8-92
- operator+() 8-92
- operator+=() 8-93
- set() 8-93
- setNull() 8-93
- toText() 8-94

Map Class 8-95

- Summary of Map Methods 8-95
- put() 8-95

MetaData Class 8-97

- Summary of MetaData Methods 8-98
- getAttributeCount() 8-99
- getAttributeId() 8-99
- getAttributeType() 8-100
- getBoolean() 8-100
- getInt() 8-100
- getMetaData() 8-101
- getNumber() 8-101
- getRef() 8-101

getString() 8-102
getTimeStamp() 8-102
getUInt() 8-102
getVector() 8-103
operator=() 8-103
Number Class 8-104
Summary of Number Methods 8-107
abs() 8-110
arcCos() 8-110
arcSin() 8-110
arcTan() 8-110
arcTan2() 8-111
ceil() 8-111
cos() 8-111
exp() 8-111
floor() 8-112
fromBytes() 8-112
fromText() 8-112
hypCos() 8-113
hypSin() 8-113
hypTan() 8-113
intPower() 8-113
isNull() 8-114
ln() 8-114
log() 8-114
operator++() 8-114
operator++() 8-115
operator--() 8-115
operator--() 8-115
operator*() 8-115
operator/() 8-116
operator%() 8-116
operator+() 8-117
operator-() 8-117
operator-() 8-117

operator<() 8-118
operator<=() 8-118
operator>() 8-119
operator>=() 8-119
operator==() 8-120
operator!=() 8-120
operator=() 8-121
operator*=() 8-121
operator/=() 8-122
operator%=() 8-122
operator+=() 8-122
operator-=() 8-123
operator char() 8-123
operator double() 8-123
operator float() 8-123
operator int() 8-124
operator long() 8-124
operator long double() 8-124
operator short() 8-124
operator unsigned char() 8-125
operator unsigned int() 8-125
operator unsigned long() 8-125
operator unsigned short() 8-125
power() 8-126
prec() 8-126
round() 8-126
setNull() 8-127
shift() 8-127
sign() 8-127
sin() 8-127
sqareroot() 8-128
tan() 8-128
toBytes() 8-128
toText() 8-128
trunc() 8-129

PObject Class 8-130
Summary of PObject Methods 8-130
flush() 8-131
getConnection() 8-131
getRef() 8-131
isLocked() 8-132
isNull() 8-132
lock() 8-132
markDelete() 8-133
markModified() 8-133
operator=() 8-133
operator delete() 8-133
operator new() 8-134
pin() 8-135
setNull() 8-135
unmark() 8-135
unpin() 8-135
Ref Class 8-137
Summary of Ref Methods 8-138
clear() 8-138
getConnection() 8-138
getRef() 8-139
isNull() 8-139
markDelete() 8-139
operator->() 8-139
operator*() 8-140
operator==() 8-140
operator!=() 8-140
operator=() 8-141
ptr() 8-141
setPrefetch() 8-141
setLock() 8-142
unmarkDelete() 8-143
RefAny Class 8-144
Summary of RefAny Methods 8-144

clear() 8-144
getConnection() 8-145
getRef() 8-145
isNull() 8-145
markDelete() 8-145
operator=() 8-145
operator==() 8-146
operator!=() 8-146
unmarkDelete() 8-146
ResultSet Class 8-147
ResultSet() 8-147
Summary of RefAny Methods 8-148
cancel() 8-150
closeStream() 8-150
getBfile() 8-150
getBlob() 8-151
getBytes() 8-151
getCharSet() 8-152
getClob() 8-152
getColumnListMetaData() 8-152
getCurrentStreamColumn() 8-153
getCurrentStreamRow() 8-153
getCursor() 8-153
getDate() 8-154
getDouble() 8-154
getFloat() 8-154
getInt() 8-155
getIntervalDS() 8-155
getIntervalYM() 8-155
getMaxColumnSize() 8-156
getNumArrayRows() 8-156
getNumber() 8-156
getObject() 8-157
getRef() 8-157
getRowid() 8-158

getRowPosition() 8-158
getStatement() 8-158
getStream() 8-158
getString() 8-159
getTimestamp() 8-159
getUInt() 8-159
getVector() 8-160
isNull() 8-162
isTruncated() 8-162
next() 8-163
preTruncationLength() 8-164
setBinaryStreamMode() 8-164
setCharacterStreamMode() 8-164
setCharSet() 8-165
setDataBuffer() 8-165
setErrorOnNull() 8-166
setErrorOnTruncate() 8-167
setMaxColumnSize() 8-167
status() 8-168
SQLException Class 8-169
SQLException() 8-169
Summary of SQLException Methods 8-169
getErrorCode() 8-169
getMessage() 8-169
setErrorCtx() 8-170
Statement Class 8-171
Summary of Statement Methods 8-171
addIteration() 8-175
closeResultSet() 8-175
closeStream() 8-176
execute() 8-176
executeArrayUpdate() 8-177
executeQuery() 8-179
executeUpdate() 8-179
getAutoCommit() 8-179

getBfile() 8-180
getBlob() 8-180
getBytes() 8-180
getCharSet() 8-181
getClob() 8-181
getConnection() 8-181
getCurrentIteration() 8-182
getCurrentStreamIteration() 8-182
getCurrentStreamParam() 8-182
getCursor() 8-182
getDatabaseNCHARParam() 8-183
getDate() 8-183
getDouble() 8-184
getFloat() 8-184
getInt() 8-184
getIntervalDS() 8-185
getIntervalYM() 8-185
getMaxIterations() 8-186
getMaxParamSize() 8-186
getNumber() 8-186
getObject() 8-187
getOCIStatement() 8-187
getRef() 8-187
getResultSet() 8-188
getRowid() 8-188
getSQL() 8-188
getStream() 8-188
getString() 8-189
getTimestamp() 8-189
getUInt() 8-189
getUpdateCount() 8-190
getVector() 8-190
isNull() 8-192
isTruncated() 8-193
preTruncationLength() 8-193

registerOutParam() 8-193
setAutoCommit() 8-194
setBfile() 8-195
setBinaryStreamMode() 8-195
setBlob() 8-196
setBytes() 8-196
setCharacterStreamMode() 8-197
setCharSet() 8-197
setClob() 8-197
setDate() 8-198
setDatabaseNCHARParam() 8-198
setDataBuffer() 8-199
setDataBufferArray() 8-200
setDouble() 8-202
setErrorOnNull() 8-203
setErrorOnTruncate() 8-203
setFloat() 8-204
setInt() 8-204
setIntervalDS() 8-204
setIntervalYM() 8-205
setMaxIterations() 8-205
setMaxParamSize() 8-206
setNull() 8-206
setNumber() 8-207
setObject() 8-207
setPrefetchMemorySize() 8-208
setPrefetchRowCount() 8-208
setRef() 8-209
setRowid() 8-209
setSQL() 8-210
setString() 8-210
setTimestamp() 8-210
setUInt() 8-211
setVector() 8-211
status() 8-214

Stream Class 8-215
Summary of Stream Methods 8-215
readBuffer() 8-215
readLastBuffer() 8-216
writeBuffer() 8-216
writeLastBuffer() 8-217
status() 8-217
Timestamp Class 8-219
Summary of Timestamp Methods 8-221
fromText() 8-222
getDate() 8-222
getTime() 8-223
getTimeZoneOffset() 8-223
intervalAdd() 8-224
intervalSub() 8-224
isNull() 8-225
operator=() 8-225
operator==() 8-225
operator!=() 8-226
operator>() 8-226
operator>=() 8-227
operator<() 8-227
operator<=() 8-228
setDate() 8-228
setNull() 8-229
setTimeZoneOffset() 8-229
subDS() 8-230
subYM() 8-230
toText() 8-230

Part III Appendix

A OCCI Demonstration Programs

OCCI Demonstration Programs A-2
demo_rdbms.mk A-2

occiblob.cpp A-7
occiclob.cpp A-12
occicoll.cpp A-16
occidesc.cpp A-21
occidml.cpp A-30
occiinh.typ A-34
occiinh.cpp A-34
occiobj.typ A-41
occiobj.cpp A-41
occipobj.typ A-45
occipobj.cpp A-45
occipool.cpp A-50
occiproc.cpp A-52
occistre.cpp A-55

Index

Send Us Your Comments

Oracle C++ Call Interface Programmer's Guide, Release 9.0.1

Part No. A89860-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title and part number, and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev_us@oracle.com
- FAX: (650) 506-7227 Attn: Server Technologies Documentation Manager
- Postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, Mailstop 4op11
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

The Oracle C++ Call Interface (OCCI) is an application programming interface (API) that allows applications written in C++ to interact with one or more Oracle database servers. OCCI gives your programs the ability to perform the full range of database operations that are possible with an Oracle database server, including SQL statement processing and object manipulation.

This preface contains these topics:

- Audience
- Organization
- Related Documentation
- Conventions
- Documentation Accessibility

Audience

The *Oracle C++ Call Interface Programmer's Guide* is intended for programmers, system analysts, project managers, and other Oracle users who perform, or are interested in learning about, the following tasks:

- Design and develop database applications in the Oracle environment.
- Convert existing database applications to run in the Oracle environment.
- Manage the development of database applications.

To use this document, you need a basic understanding of object-oriented programming concepts, familiarity with the use of Structured Query Language (SQL), and a working knowledge of application development using C++.

Organization

This document contains:

PART I: OCCI Programming

Part 1 (Chapter 1 through Chapter 7) provides information about how to program with OCCI to build scalable application solutions that provide access to relational and object data in an Oracle database.

Chapter 1, "Introduction to OCCI"

This chapter introduces you to OCCI and describes special terms and typographical conventions that are used in describing OCCI.

Chapter 2, "Relational Programming"

This chapter gives you the basic concepts needed to develop an OCCI program. It discusses the essential steps each OCCI program must include, and how to retrieve and understand error messages

Chapter 3, "Object Programming"

This chapter provides an introduction to the concepts involved when using OCCI to access objects in an Oracle database server. The chapter includes a discussion of basic object concepts and object navigational access, and the basic structure of object-relational applications.

Chapter 4, "Datatypes"

This chapter discusses Oracle internal and external datatypes, and necessary data conversions.

Chapter 5, "Introduction to LOBs"

This chapter provides an intrduction to LOBs and the related classes and methods.

Chapter 6, "Metadata"

This chapter discusses how to use the `MetaData()` method to obtain information about schema objects and their associated elements.

Chapter 7, "How to Use the Object Type Translator Utility"

This chapter discusses the use of the Object Type Translator (OTT) to convert database object definitions to C++ representations for use in OCCI applications.

PART II: The Application Programmer's Interface Reference

Part 2 (Chapter 8) describes OCCI classes and methods.

Chapter 8, "OCCI Classes and Methods"

This chapter describes the OCCI classes and methods for C++.

PART III: Appendix

Part 3 (Appendix A) presents the OCCI demonstration programs.

Appendix A, "OCCI Demonstration Programs"

This appendix identifies the OCCI demonstration programs and provides the code for each.

Related Documentation

For more information, see these Oracle resources:

- *Oracle9i Database Concepts*
- *Oracle9i SQL Reference*
- *Oracle9i Database Administrator's Guide*
- *Oracle9i Application Developer's Guide - Object-Relational Features.*

- *Oracle9i Database New Features*
- *Oracle Call Interface Programmer's Guide*
- *Oracle9i Database Server Cache Concepts and Administration Guide*

Many books in the documentation set use the sample schemas of the seed database, which is installed by default when you install Oracle. Refer to *Oracle9i Sample Schemas* for information on how these schemas were created and how you can use them yourself.

In North America, printed documentation is available for sale in the Oracle Store at

```
http://oraclestore.oracle.com/
```

Customers in Europe, the Middle East, and Africa (EMEA) can purchase documentation from

```
http://www.oraclebookshop.com/
```

Other customers can contact their Oracle representative to purchase printed documentation.

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

```
http://technet.oracle.com/membership/index.htm
```

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

```
http://technet.oracle.com/docs/index.htm
```

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- Conventions in Code Examples

Conventions in Text

We use various conventions in text to help you more quickly identify special terms. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle9i Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, RMAN keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true.` Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

Conventions in Code Examples

Code examples illustrate SQL, PL/SQL, SQL*Plus, or other command-line statements. They are displayed in a monospace (fixed-width) font and separated from normal text as shown in this example:

```
SELECT username FROM dba_users WHERE username = 'MIGRATE';
```

The following table describes typographic conventions used in code examples and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`
...	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
`Italics`	Italicized text indicates placeholders or variables for which you must supply particular values.	`CONNECT SYSTEM/`*`system_password`* `DB_NAME = `*`database_name`*

Convention	Meaning	Example

`UPPERCASE`	Uppercase typeface indicates elements supplied by the system. We show these terms in uppercase in order to distinguish them from terms you define. Unless terms appear in brackets, enter them in the order and with the spelling shown. However, because these terms are not case sensitive, you can enter them in lowercase.	`SELECT last_name, employee_id FROM employees;`
		`SELECT * FROM USER_TABLES;`
		`DROP TABLE hr.employees;`

`lowercase`	Lowercase typeface indicates programmatic elements that you supply. For example, lowercase indicates names of tables, columns, or files.
	Note: Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.
		`SELECT last_name, employee_id FROM employees;`
		`sqlplus hr/hr`
		`CREATE USER mjones IDENTIFIED BY ty3MU9;`

Documentation Accessibility

Oracle's goal is to make our products, services, and supporting documentation accessible to the disabled community with good usability. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For additional information, visit the Oracle Accessibility Program Web site at

http://www.oracle.com/accessibility/

JAWS, a Windows screen reader, may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, JAWS may not always read a line of text that consists solely of a bracket or brace.

Part I

OCCI Programmer's Guide

This part contains the following chapters:

- Chapter 1, "Introduction to OCCI"
- Chapter 2, "Relational Programming"
- Chapter 3, "Object Programming"
- Chapter 4, "Datatypes"
- Chapter 5, "Introduction to LOBs"
- Chapter 6, "Metadata"
- Chapter 7, "How to Use the Object Type Translator Utility"

1 Introduction to OCCI

This chapter provides an overview of Oracle C++ Call Interface (OCCI) and introduces terminology used in discussing OCCI. You are provided with the background information needed to develop C++ applications that run in an Oracle environment.

The following topics are covered:

- Overview of OCCI
- Processing of SQL Statements
- Overview of PL/SQL
- Special OCCI/SQL Terms
- Object Support

Overview of OCCI

Oracle C++ Call Interface (OCCI) is an application program interface (API) that provides C++ applications access to data in an Oracle database. OCCI enables C++ programmers to utilize the full range of Oracle database operations, including SQL statement processing and object manipulation.

OCCI provides for:

- High performance applications through the efficient use of system memory and network connectivity
- Scalable applications that can service an increasing number of users and requests
- Comprehensive support for application development by using Oracle database objects, including client-side access to Oracle database objects
- Simplified user authentication and password management
- n-tiered authentication
- Consistent interfaces for dynamic connection management and transaction management in two-tier client/server environments or multitiered environments
- Encapsulated and opaque interfaces

OCCI provides a library of standard database access and retrieval functions in the form of a dynamic runtime library (OCCI classes) that can be linked in a C++ application at runtime. This eliminates the need to embed SQL or PL/SQL within third-generation language (3GL) programs.

Benefits of OCCI

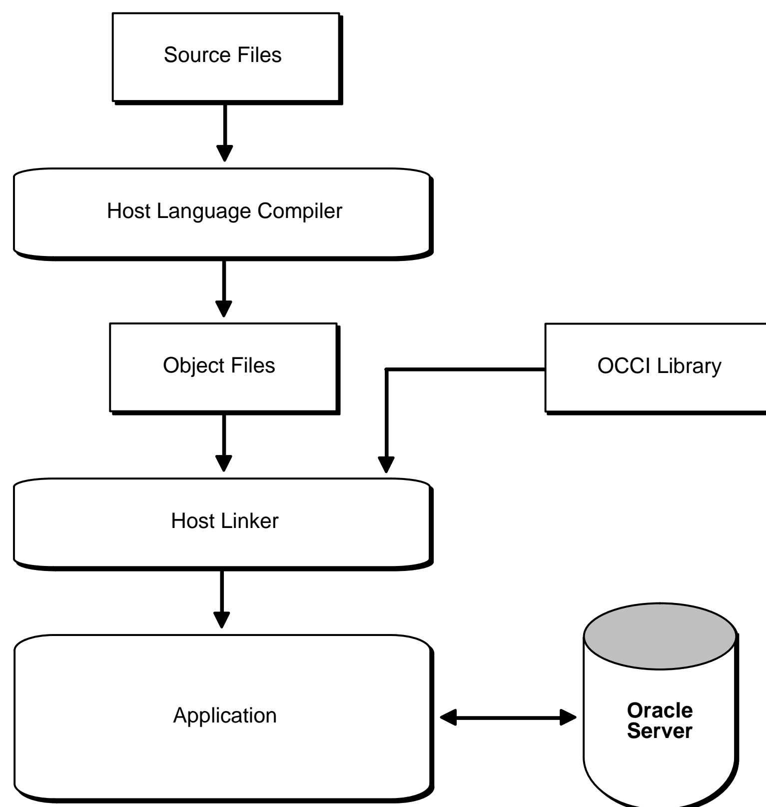
OCCI provides these significant advantages over other methods of accessing an Oracle database:

- Provides OCI functionality
- Leverages C++ and the Object Oriented Programming paradigm
- Easy to use
- Easy to learn for those familiar with JDBC
- Navigational interface to manipulate database objects of user-defined types as C++ class instances

Building an OCCI Application

As Figure 1–1 shows, you compile and link an OCCI program in the same way that you compile and link a nondatabase application.

Figure 1–1 The OCCI Development Process



Oracle supports most popular third-party compilers. The details of linking an OCCI program vary from system to system. On some platforms, it may be necessary to include other libraries, in addition to the OCCI library, to properly link your OCCI programs.

Functionality of OCCI

OCCI provides the following functionality:

- APIs to design a scalable, shared server application that can support large numbers of users securely
- SQL access functions, for managing database access, processing SQL statements, and manipulating objects retrieved from an Oracle database server
- Datatype mapping and manipulation functions, for manipulating data attributes of Oracle types

Procedural and Nonprocedural Elements

Oracle C++ Call Interface (OCCI) enables you to develop scalable, shared server applications on multitiered architecture that combine the nonprocedural data access power of structured query language (SQL) with the procedural capabilities of C++.

In a nonprocedural language program, the set of data to be operated on is specified, but what operations will be performed, or how the operations are to be carried out, is not specified. The nonprocedural nature of SQL makes it an easy language to learn and use to perform database transactions. It is also the standard language used to access and manipulate data in modern relational and object-relational database systems.

In a procedural language program, the execution of most statements depends on previous or subsequent statements and on control structures, such as loops or conditional branches, which are not available in SQL. The procedural nature of these languages makes them more complex than SQL, but it also makes them very flexible and powerful.

The combination of both nonprocedural and procedural language elements in an OCCI program provides easy access to an Oracle database in a structured programming environment.

OCCI supports all SQL data definition, data manipulation, query, and transaction control facilities that are available through an Oracle database server. For example, an OCCI program can run a query against an Oracle database. The queries can require the program to supply data to the database by using input (bind) variables, as follows:

```
SELECT name FROM employees WHERE empno = :empnumber
```

In the above SQL statement, `:empnumber` is a placeholder for a value that will be supplied by the application.

In an OCCI application, you can also take advantage of PL/SQL, Oracle's procedural extension to SQL. The applications you develop can be more powerful and flexible than applications written in SQL alone. OCCI also provides facilities for accessing and manipulating objects in an Oracle database server.

Processing of SQL Statements

One of the main tasks of an OCCI application is to process SQL statements. Different types of SQL statements require different processing steps in your program. It is important to take this into account when coding your OCCI application. Oracle recognizes several types of SQL statements:

- Data definition language (DDL) statements
- Control statements
  - Transaction control statements
  - Connection control statements
  - System control statements
- Data manipulation language (DML) statements
- Queries

Data Definition Language Statements

Data definition language (DDL) statements manage schema objects in the database. DDL statements create new tables, drop old tables, and establish other schema objects. They also control access to schema objects.

The following is an example of creating and specifying access to a table:

```
CREATE TABLE employees (
    name       VARCHAR2(20),
    ssn        VARCHAR2(12),
    empno      NUMBER(6),
    mgr        NUMBER(6),
    salary     NUMBER(6))

GRANT UPDATE, INSERT, DELETE ON employees TO donna
REVOKE UPDATE ON employees FROM jamie
```

DDL statements also allow you to work with objects in the Oracle database, as in the following series of statements which create an object table:

```
CREATE TYPE person_t AS OBJECT (
    name     VARCHAR2(30),
    ssn      VARCHAR2(12),
    address  VARCHAR2(50))

CREATE TABLE person_tab OF person_t
```

Control Statements

OCCI applications treat transaction control, connection control, and system control statements like DML statements.

Data Manipulation LanguageSQL Statements

Data manipulation language (DML) statements can change data in database tables. For example, DML statements are used to perform the following actions:

- Insert new rows into a table
- Update column values in existing rows
- Delete rows from a table
- Lock a table in the database
- Explain the execution plan for a SQL statement

DML statements can require an application to supply data to the database by using input (bind) variables. Consider the following statement:

```
INSERT INTO dept_tab VALUES(:1,:2,:3)
```

Either this statement can be executed several times with different bind values, or an array insert can be performed to insert several rows in one round-trip to the server.

DML statements also enable you to work with objects in the Oracle database, as in the following example, which inserts an instance of type `person_t` into the object table `person_tab`:

```
INSERT INTO person_tab
    VALUES (person_t('Steve May','123-45-6789','146 Winfield Street'))
```

Queries

Queries are statements that retrieve data from tables in a database. A query can return zero, one, or many rows of data. All queries begin with the SQL keyword `SELECT`, as in the following example:

```
SELECT dname FROM dept
    WHERE deptno = 42
```

Queries can require the program to supply data to the database server by using input (bind) variables, as in the following example:

```
SELECT name
    FROM employees
    WHERE empno = :empnumber
```

In the above SQL statement, `:empnumber` is a placeholder for a value that will be supplied by the application.

Overview of PL/SQL

PL/SQL is Oracle's procedural extension to the SQL language. PL/SQL processes tasks that are more complicated than simple queries and SQL data manipulation language statements. PL/SQL allows a number of constructs to be grouped into a single block and executed as a unit. Among these are the following constructs:

- One or more SQL statements
- Variable declarations
- Assignment statements
- Procedural control statements (`IF ... THEN ... ELSE` statements and loops)
- Exception handling

In addition to calling PL/SQL stored procedures from an OCCI program, you can use PL/SQL blocks in your OCCI program to perform the following tasks:

- Call other PL/SQL stored procedures and stored functions.
- Combine procedural control statements with several SQL statements, to be executed as a single unit.
- Access special PL/SQL features such as records, tables, cursor `FOR` loops, and exception handling
- Use cursor variables
- Access and manipulate objects in an Oracle database

A PL/SQL procedure or function can also return an output variable. This is called an **out bind variable**. For example:

```
BEGIN
    GET_EMPLOYEE_NAME(:1, :2);
END;
```

Here, the first parameter is an input variable that provides the ID number of an employee. The second parameter, or the out bind variable, contains the return value of employee name.

The following PL/SQL example issues a SQL statement to retrieve values from a table of employees, given a particular employee number. This example also demonstrates the use of placeholders in PL/SQL statements.

```
SELECT ename, sal, comm INTO :emp_name, :salary, :commission
    FROM emp
    WHERE ename = :emp_number;
```

Note that the placeholders in this statement are not PL/SQL variables. They represent input and output parameters passed to and from the database server when the statement is processed. These placeholders need to be specified in your program.

Special OCCI/SQL Terms

This guide uses special terms to refer to the different parts of a SQL statement. Consider the following example of a SQL statement:

```
SELECT customer, address
    FROM customers
    WHERE bus_type = 'SOFTWARE'
    AND sales_volume = :sales
```

This example contains these parts:

- A SQL *command*: `SELECT`
- Two *select-list items*: `customer` and `address`
- A *table name* in the `FROM` clause: `customers`
- Two *column names* in the `WHERE` clause: `bus_type` and `sales_volume`
- A *literal input value* in the `WHERE` clause: '`SOFTWARE`'
- A *placeholder* for an input (bind) variable in the `WHERE` clause: `:sales`

When you develop your OCCI application, you call routines that specify to the database server the value of, or reference to, input and output variables in your

program. In this guide, specifying the placeholder variable for data is called a **bind operation**. For input variables, this is called an **in bind operation**. For output variables, this is called an **out bind operation**.

Object Support

OCCI has facilities for working with *object types* and *objects*. An **object type** is a user-defined data structure representing an abstraction of a real-world entity. For example, the database might contain a definition of a `person` object. That object type might have *attributes*—`first_name`, `last_name`, and `age`—which represent a person's identifying characteristics.

The object type definition serves as the basis for creating **objects**, which represent instances of the object type. By using the object type as a structural definition, a `person` object could be created with the attributes `John`, `Bonivento`, and `30`. Object types may also contain *methods*—programmatic functions that represent the behavior of that object type.

OCCI provides a comprehensive API for programmers seeking to use the Oracle database server's object capabilities. These features can be divided into several major categories:

- Client-side object cache
- Runtime environment for objects
- Associative and navigational interfaces to access and manipulate objects
- Metadata class to describe object type metadata
- Object Type Translator (OTT) utility, which maps internal Oracle schema information to client-side language bind variables

See Also:

- *Oracle9i Database Concepts* and
- *Oracle9i Application Developer's Guide - Object-Relational Features* for a more detailed explanation of object types and objects

Client-Side Object Cache

The object cache is a client-side memory buffer that provides lookup and memory management support for objects. It stores and tracks objects which have been fetched by an OCCI application from the server to the client side. The client-side object cache is created when the OCCI environment is initialized in `object` mode.

Multiple applications running against the same server will each have their own object cache. The client-side object cache tracks the objects that are currently in memory, maintains references to objects, manages automatic object swapping and tracks the meta-attributes or type information about objects. The client-side object cache provides the following benefits:

- Improved application performance by reducing the number of client/server round-trips required to fetch and operate on objects
- Enhanced scalability by supporting object swapping from the client-side cache
- Improved concurrency by supporting object-level locking

Runtime Environment for Objects

OCCI provides a runtime environment for objects that offers a set of methods for managing how Oracle objects are used on the client side. These methods provide the necessary functionality for performing these tasks:

- Connecting to an Oracle database server in order to access its object functionality
- Allocating the client-side object cache and tuning its parameters
- Retrieving error and warning messages
- Controlling transactions that access objects in the database
- Associatively accessing objects through SQL
- Describing a PL/SQL procedure or function whose parameters or result are of Oracle object type system types

Associative and Navigational Interfaces

Applications that use OCCI can access objects in the database through several types of interfaces:

- SQL `SELECT`, `INSERT`, and `UPDATE` statements
- C++ pointers and references to access objects in the client-side object cache by traversing the corresponding references

OCCI provides a set of methods to support object manipulation by using SQL `SELECT`, `INSERT`, and `UPDATE` statements. To access Oracle objects, these SQL statements use a consistent set of steps as if they were accessing relational tables. OCCI provides methods to access objects by using SQL statements for:

- Binding object type instances and references as input and output variables of SQL statements and PL/SQL stored procedures
- Executing SQL statements that contain object type instances and references
- Fetching object type instances and references
- Retrieving column values from a result set as objects
- Describing a select-list item of an Oracle object type

OCCI provides a seamless interface for navigating objects, enabling you to manipulate database objects in the same way that you would operate on transient C++ objects. You can dereference the overloaded arrow (`->`) operator on an object reference to transparently materialize the object from the database into the application space.

Metadata Class

Each Oracle datatype is represented in OCCI by a C++ class. The class exposes the behavior and characteristics of the datatype by overloaded operators and methods. For example, the Oracle datatype `NUMBER` is represented by the `Number` class.

OCCI provides a metadata class that enables you to retrieve metadata describing database objects, including object types.

Object Type Translator Utility

The Object Type Translator (OTT) utility translates schema information about Oracle object types into client-side language bindings. That is, OTT translates object type information into declarations of host language variables, such as structures and classes. OTT takes an `intype` file which contains information about Oracle database schema objects as input. OTT generates an `outtype` file and the necessary header and implementation files that must be included in a C++ application that runs against the object schema. OTT has many benefits, including:

- Improving application developer productivity: OTT eliminates the need for application developers to write by hand the host language variables that correspond to schema objects.
- Maintaining SQL as the data definition language of choice: By providing the ability to automatically map Oracle database schema objects that are created by using SQL to host language variables, OTT facilitates the use of SQL as the data definition language of choice. This in turn allows Oracle to support a consistent, enterprise-wide model of the user’s data.

- Facilitating schema evolution of object types: OTT provides the ability to regenerate included header files when the schema is changed, allowing Oracle applications to support schema evolution.

OTT is typically invoked from the command line by specifying the `intype` file, the `outtype` file, and the specific database connection.

In summary, OCCI supports the following methods to handle objects in an Oracle database:

- Execution of SQL statements that manipulate object data and schema information
- Passing object references and instances as input variables in SQL statements
- Declaring object references and instances as variables to receive the output of SQL statements
- Fetching object references and instances from a database
- Describing the properties of SQL statements that return object instances and references
- Describing PL/SQL procedures or functions with object parameters or results
- Extending commit and rollback calls to synchronize object and relational functionality

2

Relational Programming

This chapter describes the basics of developing C++ applications using Oracle C++ Interface (OCCI) to work with data stored in standard relational databases.

It includes the following topics:

- Connecting to a Database
- Executing SQL DDL and DML Statements
- Types of SQL Statements in the OCCI Environment
- Streamed Reads and Writes
- Executing SQL Queries
- Executing Statements Dynamically
- Committing a Transaction
- Error Handling
- Advanced Relational Techniques

Connecting to a Database

You have a number of different options with regard to how your application connects to the database. These options are discussed in the following sections:

- Creating and Terminating an Environment
- Opening and Closing a Connection
- Creating a Connection Pool
- Utilizing a Shared Server Environment

Creating and Terminating an Environment

All OCCI processing takes place in the context of the `Environment` class. An OCCI environment provides application modes and user-specified memory management functions. The following code example shows how you can create an OCCI environment:

```
Environment *env = Environment::createEnvironment();
```

All OCCI objects created with the `create`*`xxx`* methods (connections, connection pools, statements) must be explicitly terminated and so, when appropriate, you must also explicitly terminate the environment. The following code example shows how you terminate an OCCI environment.

```
Environment::terminateEnvironment(env);
```

In addition, an OCCI environment should have a scope that is larger than the scope of any objects created in the context of the that environment. This concept is demonstrated in the following code example:

```
const string userName = "SCOTT";
const string password = "TIGER";
const string connectString = "";

Environment *env = Environment::createEnvironment();
{
    Connection *conn = env->createConnection(userName, password, connectString);
    Statement *stmt = conn->createStatement("SELECT blobcol FROM mytable");
    ResultSet *rs = stmt->executeQuery();
    rs->next();
    Blob b = rs->getBlob(1);
    cout << "Length of BLOB : " << b.length();
```

```
    .
    .
    .
    stmt->closeResultSet(rs);
    conn->terminateStatement(stmt);
    env->terminateConnection(conn);
}
Environment::terminateEnvironment(env);
```

You can use the mode parameter of the `createEnvironment` method to specify that your application:

- Runs in a threaded environment (`THREADED_MUTEXED` or `THREADED_UNMUTEXED`)
- Uses objects (`OBJECT`)
- Utilizes shared data structures (`SHARED`)

The mode can be set independently in each environment.

Opening and Closing a Connection

The `Environment` class is the factory class for creating `Connection` objects. You first create an `Environment` instance, and then use it to enable users to connect to the database by means of the `createConnection` method.

The following code example creates an environment instance and then uses it to create a database connection for a database user `scott` with the password `tiger`.

```
Environment *env = Environment::createEnvironment();
Connection *conn = env->createConnection("scott", "tiger");
```

You must use the `terminateConnection` method shown in the following code example to explicitly close the connection at the end of the working session. In addition, the OCCI environment should be explicitly terminated.

```
env->terminateConnection(conn);
Environment::terminateEnvironment(env);
```

Creating a Connection Pool

For many shared server, middle-tier applications, connections to the database should be enabled for a large number of threads, each thread for a relatively short

duration. Opening a connection to the database for every thread would result in inefficient utilization of connections and poor performance.

By employing the **connection pooling** feature, your application can use database management system (DBMS) functionality to manage the connections. Oracle creates a small number of open connections, dynamically selects one of the free connections to execute a statement, and then releases the connection immediately after the execution. This relieves you from creating complex mechanisms to handle connections and optimizes performance in your application.

Creating a Connection Pool

To create a connection pool, you use the `createConnectionPool` method:

```
virtual ConnectionPool* createConnectionPool(
    const string &poolUserName,
    const string &poolPassword,
    const string &connectString ="",
    unsigned int minConn =0,
    unsigned int maxConn =1,
    unsigned int incrConn =1) = 0;
```

The following parameters are used in the previous method example:

- `poolUserName`: The owner of the connection pool
- `poolPassword`: The password to gain access to the connection pool
- `connectString =""`: The database name that specifies the database server to which the connection pool is related
- `minConn`: The minimum number of connections to be opened when the connection pool is created
- `maxConn`: The maximum number of connections that can be maintained by the connection pool. When the maximum number of connections are open in the connection pool, and all the connections are busy, an OCCI method call that needs a connection waits until it gets one, unless setErrorOnBusy() was called on the connection pool
- `incrConn`: The additional number of connections to be opened when all the connections are busy and a call needs a connection. This increment is implemented only when the total number of open connections is less than the maximum number of connections that can be opened in that connection pool

The following code example demonstrates how you can create a connection pool:

```
const string poolUserName = "SCOTT";
const string poolPassword = "TIGER";
const string connectString = "";
const string username = "SCOTT";
const string password = "TIGER";
unsigned int maxConn = 5;
unsigned int minConn = 3;
unsigned int incrConn = 2;

ConnectionPool *connPool = env->createConnectionPool(poolUserName, poolPassword,
    connectString, minConn, maxConn, incrConn);
```

See Also:

- Appendix A, "OCCI Demonstration Programs" and the code example occipool.cpp that demonstrates how to use the connection pool interface of OCCI

You can also configure all these attributes dynamically. This lets you design an application that has the flexibility of reading the current load (number of open connections and number of busy connections) and tune these attributes appropriately. In addition, you can use the `setTimeOut` method to time out the connections that are idle for more than the specified time. The DBMS terminates idle connections periodically so as to maintain an optimum number of open connections.

Each connection pool has a data structure (pool handle) associated with it. This pool handle stores the pool parameters. There is no restriction that one environment must have only one connection pool. There can be multiple connection pools in a single OCCI environment, and these can connect to the same or different databases. This is useful for applications requiring load balancing. However, note that since a pool handle requires memory, multiple connection pools consume more memory.

Proxy Connections

If you authorize the connection pool user to act as a proxy for other connections, then no password is required to log in database users who use one of the connections in the connection pool to act as a proxy on their behalf.

A proxy connection can be created by using either of the following methods:

```
ConnectionPool->createProxyConnection(const string &name,
    Connection::ProxyType proxyType =
    Connection::PROXY_DEFAULT);
```

or

```
ConnectionPool->createProxyConnection(const string &name,
    string roles[], int numRoles,
    Connection::ProxyType proxyType =
    Connection::PROXY_DEFAULT);
```

The following parameters are used in the previous method example:

- `roles[]`: The roles array specifies a list of roles to be activated after the proxy connection is activated for the client
- `Connection::ProxyType proxyType = Connection::PROXY_DEFAULT`: The enumeration `Connection::ProxyType` lists constants representing the various ways of achieving proxy authentication. `PROXY_DEFAULT` is used to indicate that `name` represents a database username and is the only proxy authentication mode currently supported.

Executing SQL DDL and DML Statements

SQL is the industry-wide language for working with relational databases. In OCCI you execute SQL commands by means of the `Statement` class.

Creating a Statement Handle

To create a `Statement` handle, call the `createStatement` method of the `Connection` object, as shown in the following example:

```
Statement *stmt = conn->createStatement();
```

Creating a Statement Handle to Execute SQL Commands

Once you have created a `Statement` handle, execute SQL commands by calling the `execute`, `executeUpdate`, `executeArrayUpdate`, or `executeQuery` methods on the `Statement`. These methods are used for the following purposes:

- `execute`: To execute all nonspecific statement types
- `executeUpdate`: To execute DML and DDL statements
- `executeQuery`: To execute a query

- `executeArrayUpdate`: To execute multiple DML statements

Creating a Database Table

Using the `executeUpdate` method, the following code example demonstrates how you can create a database table:

```
stmt->executeUpdate("CREATE TABLE basket_tab
    (fruit VARCHAR2(30), quantity NUMBER)");
```

Inserting Values into a Database Table

Similarly, you can execute a SQL `INSERT` statement by invoking the `executeUpdate` method:

```
stmt->executeUpdate("INSERT INTO basket_tab
    VALUES('MANGOES', 3)");
```

The `executeUpdate` method returns the number of rows affected by the SQL statement.

See Also:

- Appendix A, "OCCI Demonstration Programs" and the code example occidml.cpp that demonstrates how to perform insert, select, update, and delete operations of a table row by using OCCI

Reusing a Statement Handle

You can reuse a statement handle to execute SQL statements multiple times. For example, to repeatedly execute the same statement with different parameters, you specify the statement by the `setSQL` method of the `Statement` handle:

```
stmt->setSQL("INSERT INTO basket_tab VALUES(:1,:2)");
```

You may now execute this `INSERT` statement as many times as required. If at a later time you wish to execute a different SQL statement, you simply reset the statement handle. For example:

```
stmt->setSQL("SELECT * FROM basket_tab WHERE quantity >= :1");
```

Thus, OCCI statement handles and their associated resources are not allocated or freed unnecessarily. You can retrieve the contents of the current statement handle at any time by means of the `getSQL` method.

SHARED Mode

When a SQL statement is processed, certain underlying data is associated with the statement. This data includes information about statement text and bind data, as well as resultset and describe information for queries. This data remains the same from one execution of a statement to another, even if the statement is executed by different users.

When an OCCI environment is initialized in `SHARED` mode, common statement data is shared between multiple statement handles, thus providing memory savings for the application. This savings may be particularly valuable for applications that create multiple statement handles which execute the same SQL statement on different user sessions, either on the same or multiple connections.

To enable sharing of common metadata across multiple statement handles, create the `Environment` in `SHARED` mode.

Terminating a Statement Handle

You should explicitly terminate and deallocate a `Statement`:

```
Connection::conn->terminateStatement(Statement *stmt);
```

Types of SQL Statements in the OCCI Environment

There are three types of SQL statements in the OCCI environment:

- Standard Statements use SQL commands with specified values
- Parameterized Statements have parameters, or bind variables
- Callable Statements call stored PL/SQL procedures

The `Statement` methods are subdivided into those applicable to all statements, to parameterized statements, and to callable statements. Standard statements are a superset of parameterized statements, and parameterized statements are a superset of callable statements.

Standard Statements

Previous sections describe examples of both DDL and DML commands. For example:

```
stmt->executeUpdate("CREATE TABLE basket_tab
    (fruit VARCHAR2(30), quantity NUMBER)");
```

and

```
stmt->executeUpdate("INSERT INTO basket_tab
    VALUES('MANGOES', 3)");
```

These are each an example of a **standard statement** in which you explicitly define the values of the statement. So, in these examples, the `CREATE TABLE` statement specifies the name of the table (`basket_tab`), and the INSERT statement stipulates the values to be inserted `('MANGOES', 3)`.

Parameterized Statements

You can execute the same statement with different parameters by setting placeholders for the input variables of the statement. These statements are referred to as parameterized statements because they are able to accept input from a user or program by using parameters.

For example, suppose you want to execute an `INSERT` statement with different parameters. You first specify the statement by the `setSQL` method of the `Statement` handle:

```
stmt->setSQL("INSERT INTO basket_tab VALUES(:1, :2)");
```

You then call the `set`*`xxx`* methods to specify the parameters, where *`xxx`* stands for the type of the parameter. The following example invokes the `setString` and `setInt` methods to input the values of these types into the first and second parameters.

To insert a row:

```
stmt->setString(1, "Bananas");     // value for first parameter
stmt->setInt(2, 5);                // value for second parameter
```

Having specified the parameters, you insert values into the row:

```
stmt->executeUpdate();             // execute statement
```

To insert another row:

```
stmt->setString(1, "Apples");      // value for first parameter
stmt->setInt(2, 9);                // value for second parameter
```

Having specified the parameters, you again insert values into the row:

```
stmt->executeUpdate();             // execute statement
```

If your application is executing the same statement repeatedly, then avoid changing the input parameter types because a rebind is performed each time the input type changes.

Callable Statements

PL/SQL stored procedures, as their name suggests, are procedures that are stored on the database server for reuse by an application. By using OCCI, a call to a procedure which contains other SQL statements is referred to as a **callable statement**.

For example, suppose you wish to call a procedure (`countFruit`) that returns the quantity of a specified kind of fruit. To specify the input parameters of a PL/SQL stored procedure, call the `set`*`xxx`* methods of the `Statement` class as you would for parameterized statements.

```
stmt->setSQL("BEGIN countFruit(:1, :2); END:");
int quantity;
stmt->setString(1, "Apples");
    // specify the first (IN) parameter of procedure
```

However, before calling a stored procedure, you need to specify the type and size of any `OUT` and `IN/OUT` parameters by calling the `registerOutParam` method.

```
stmt->registerOutParam(2, Type::OCCIINT, sizeof(quantity));
    // specify the type and size of the second (OUT) parameter
```

You now execute the statement by calling the procedure:

```
stmt->executeUpdate();           // call the procedure
```

Finally, you obtain the output parameters by calling the relevant `get`*`xxx`* method:

```
quantity = stmt->getInt(2);     // get the value of the second (OUT) parameter
```

Callable Statements with Arrays as Parameters

A PL/SQL stored procedure executed through a callable statement can have array of values as parameters. The number of elements in the array and the dimension of elements in the array are specified through the `setDataBufferArray` method.

The following example shows the `setDataBufferArray` method:

```
void setDataBufferArray(int paramIndex,
    void *buffer,
    Type type,
    ub4 arraySize,
    ub4 *arrayLength,
    sb4 elementSize,
    sb2 *ind = NULL,
    ub2 *rc = NULL);
```

The following parameters are used in the previous method example:

- `paramIndex`: Parameter number
- `buffer`: Data buffer containing an array of values
- `Type`: Type of data in the data buffer
- `arraySize`: Maximum number of elements in the array
- `arrayLength`: Number of elements in the array
- `elementSize`: Size of the current element in the array
- `ind`: Indicator information
- `rc`: Return code

See Also:

- Appendix A, "OCCI Demonstration Programs" and the code example occiproc.cpp that demonstrates how to invoke PL/SQL procedures with bind parameters

Streamed Reads and Writes

Streamed data is of three kinds:

- A writable stream corresponds to an `IN` bind variable.
- A readable stream corresponds to an `OUT` bind variable.
- A bidirectional stream corresponds to an `IN/OUT` bind variable.

OCCI supports streamed parameters for parameterized and callable statements of all three kinds: `IN`, `OUT`, and `IN/OUT`.

Modifying Rows Iteratively

While you can issue the `executeUpdate` method repeatedly for each row, OCCI provides an efficient mechanism for sending data for multiple rows in a single network round-trip. To do this, use the `addIteration` method of the `Statement` class to perform batch operations that modify a different row with each iteration.

To execute INSERT, UPDATE, and DELETE operations iteratively, you must:

- Set the maximum number of iterations
- Set the maximum parameter size for variable length parameters

Setting the Maximum Number of Iterations

For iterative execution, first specify the maximum number of iterations that would be done for the statement by calling the `setMaxIterations` method:

```
Statement->setMaxIterations(int maxIterations)
```

You can retrieve the current maximum iterations setting by calling the `getMaxIterations` method.

Setting the Maximum Parameter Size

If the iterative execution involves variable length datatypes, such as `string` and `Bytes`, then you must set the maximum parameter size so that OCCI can allocate the maximum size buffer:

```
Statement->setMaxParamSize(int parameterIndex, int maxParamSize)
```

You do not need to set the maximum parameter size for fixed length datatypes, such as `Number` and `Date`, or for parameters that use the `setDataBuffer` method.

You can retrieve the current maximum parameter size setting by calling the `getMaxParamSize` method.

Executing an Iterative Operation

Once you have set the maximum number of iterations and (if necessary) the maximum parameter size, iterative execution using a parameterized statement is straightforward, as shown in the following example:

```
stmt->setSQL("INSERT INTO basket_tab VALUES(:1, :2)");

stmt->setString(1, "Apples");      // value for first parameter of first row
stmt->setInt(2, 6);                // value for second parameter of first row
stmt->addIteration();              // add the iteration

stmt->setString(1, "Oranges");     // value for first parameter of second row
stmt->setInt(1, 4);                // value for second parameter of second row

stmt->executeUpdate();             // execute statement
```

As shown in the example, you call the `addIteration` method after each iteration except the last, after which you invoke `executeUpdate` method. Of course, if you did not have a second row to insert, then you would not need to call the `addIteration` method or make the subsequent calls to the `setxxx` methods.

Iterative Execution Usage Notes

- Iterative execution is designed only for use in `INSERT`, `UPDATE` and `DELETE` operations that use either standard or parameterized statements. It cannot be used for callable statements and queries.
- The datatype cannot be changed between iterations. For example, if you use `setInt` for parameter 1, then you cannot use `setString` for the same parameter in a later iteration.

Executing SQL Queries

SQL query statements allow your applications to request information from a database based on any constraints specified. A result set is returned as a result of a query.

Result Set

Execution of a database query puts the results of the query into a set of rows called the result set. In OCCI, a SQL `SELECT` statement is executed by the `executeQuery` method of the `Statement` class. This method returns an `ResultSet` object that represents the results of a query.

```
ResultSet *rs = stmt->executeQuery("SELECT * FROM basket_tab");
```

Once you have the data in the result set, you can perform operations on it. For example, suppose you wanted to print the contents of this table. The `next` method of the `ResultSet` is used to fetch data, and the `get`*`xxx`* methods are used to retrieve the individual columns of the result set, as shown in the following code example:

```
cout << "The basket has:" << endl;

while (rs->next())
{
    string fruit = rs->getString(1);     // get the first column as string
    int quantity = rs->getInt(2);        // get the second column as int

    cout << quantity << " " << fruit << endl;
}
```

The `next` and `status` methods of the `ResultSet` class return an enumerated type of `Status`. The possible values of `Status` are:

- `DATA_AVAILABLE`
- `END_OF_FETCH = 0`
- `STREAM_DATA_AVAILABLE`

If data is available for the current row, then the status is `DATA_AVAILABLE`. After all the data has been read, the status changes to `END_OF_FETCH`.

If there are any output streams to be read, then the status is `STREAM_DATA_AVAILABLE` until all the stream data is successfully read, as shown in the following code example:

```
ResultSet *rs = stmt->executeQuery("SELECT * FROM demo_tab");
ResultSet::Status status = rs->status();     // status is DATA_AVAILABLE
while (rs->next())
{
    get data and process;
```

```
}
```

When the entire result set has been traversed, then the status changes to END_OF_FETCH which terminates the WHILE loop.

The following is an example for streams for a result set:

```
char buffer[4096];
ResultSet *rs = stmt->executeQuery
    ("SELECT col2 FROM tab1 WHERE col1 = 11");
ResultSet *rs = stmt->getResultSet ();

while (rs->next ())
{
    unsigned int length = 0;
    unsigned int size = 500;
    Stream *stream = rs->getStream (2, 4000);
    while (stream->status () == Stream::READY_FOR_READ)
    {
        length += stream->readBuffer (buffer +length, size);
    }
    cout << "Read "  << length << " bytes into the buffer" << endl;
}
```

Specifying the Query

The IN bind variables can be used with queries to specify constraints in the WHERE clause of a query. For example, the following program prints only those items that have a minimum quantity of 4:

```
stmt->setSQL("SELECT * FROM basket_tab WHERE quantity >= :1");
int minimumQuantity = 4;
stmt->setInt(1, minimumQuantity);     // set first parameter
ResultSet *rs = stmt->executeQuery();
cout << "The basket has:" << endl;

while (rs->next())
    cout << rs->getInt(2) << " " << rs->getString(1) << endl;
```

Optimizing Performance by Setting Prefetch Count

Although the ResultSet method retrieves data one row at a time, the actual fetch of data from the server need not entail a network round-trip for each row queried.

To maximize the performance, you can set the number of rows to prefetch in each round-trip to the server.

You effect this either by setting the number of rows to be prefetched (`setPrefetchRowCount`), or by setting the memory size to be used for prefetching (`setPrefetchMemorySize`).

If you set both of these attributes, then the specified number of rows are prefetched unless the specified memory limit is reached first. If the specified memory limit is reached first, then the prefetch returns as many rows as will fit in the memory space defined by the call to the `setPrefetchMemorySize` method.

By default, prefetching is turned on, and the database fetches an extra row all the time. To turn prefetching off, set both the prefetch row count and memory size to zero.

Note: Prefetching is not in effect if LONG columns are part of the query. Queries containing LOB columns *can* be prefetched, because the LOB locator, rather than the data, is returned by the query.

Executing Statements Dynamically

When you know that you need to execute a DML operation, you use the `executeUpdate` method. Similarly, when you know that you need to execute a query, you use `executeQuery`.

If your application needs to allow for dynamic events and you cannot be sure of which statement will need to be executed at run time, then OCCI provides the `execute` method. Invoking the `execute` method returns one of the following statuses:

- `UNPREPARED`
- `PREPARED`
- `RESULT_SET_AVAILABLE`
- `UPDATE_COUNT_AVAILABLE`
- `NEEDS_STREAM_DATA`
- `STREAM_DATA_AVAILABLE`

While invoking the `execute` method will return one of these statuses, you can also interrogate the statement by using the `status` method.

```
Statement stmt = conn->createStatement();
Statement::Status status = stmt->status();      // status is UNPREPARED
stmt->setSQL("select * from emp");
status = stmt->status();                        // status is PREPARED
```

If a statement handle is created with a SQL string, then it is created in a `PREPARED` state. For example:

```
Statement stmt = conn->createStatement("insert into foo(id) values(99)");
Statement::Status status = stmt->status();      // status is PREPARED
status = stmt->execute();            // status is UPDATE_COUNT_AVAILABLE
```

When you set another SQL statement on the `Statement`, the status changes to `PREPARED`. For example:

```
stmt->setSQL("select * from emp");              // status is PREPARED
status = stmt->execute();            // status is RESULT_SET_AVAILABLE
```

Status Definitions

This section describes the possible values of `Status` related to a statement handle:

- UNPREPARED
- PREPARED
- RESULT_SET_AVAILABLE
- UPDATE_COUNT_AVAILABLE
- NEEDS_STREAM_DATA
- STREAM_DATA_AVAILABLE

UNPREPARED

If you have not used the `setSQL` method to attribute a SQL string to a statement handle, then the statement is in an `UNPREPARED` state.

```
Statement stmt = conn->createStatement();
Statement::Status status = stmt->status();    // status is UNPREPARED
```

PREPARED

If a `Statement` is created with a SQL string, then it is created in a `PREPARED` state. For example:

```
Statement stmt = conn->createStatement("INSERT INTO demo_tab(id)
```

```
    VALUES(99)");
Statemnt::Status status = stmt->status();    // status is PREPARED
```

Setting another SQL statement on the `Statement` will also change the status to `PREPARED`. For example:

```
status = stmt->execute();                    // status is UPDATE_COUNT_AVAILABLE
stmt->setSQL("SELECT * FROM demo_tab");      // status is PREPARED
```

RESULT_SET_AVAILABLE

A status of `RESULT_SET_AVAILABLE` indicates that a properly formulated query has been executed and the results are accessible through a result set.

When you set a statement handle to a query, it is `PREPARED`. Once you have executed the query, the statement changes to `RESULT_SET_AVAILABLE`. For example:

```
stmt->setSQL("SELECT * from EMP");           // status is PREPARED
status = stmt->execute();                    // status is RESULT_SET_AVAILABLE
```

To access the data in the result set, issue the following statement:

```
ResultSet *rs = Statement->getResultSet();
```

UPDATE_COUNT_AVAILABLE

When a DDL or DML statement in a `PREPARED` state is executed, its state changes to `UPDATE_COUNT_AVAILABLE`, as shown in the following code example:

```
Statement stmt = conn->createStatement("INSERT INTO demo_tab(id) VALUES(99)");
Statemnt::Status status = stmt->status();    // status is PREPARED
status = stmt->execute();                   // status is UPDATE_COUNT_AVAILABLE
```

This status refers to the number of rows affected by the execution of the statement. It indicates that:

- The statement did not include any input or output streams.
- The statement was not a query but either a DDL or DML statement.

You can obtain the number of rows affected by issuing the following statement:

```
Statement->getUpdateCount();
```

Note that a DDL statement will result in an update count of zero (0). Similarly, an update that does not meet any matching conditions will also produce a count of zero (0). In such a case, you cannot infer the kind of statement that has been executed from the reported status.

NEEDS_STREAM_DATA

If there are any output streams to be written, the execute does not complete until all the stream data is completely provided. In such a case, the status changes to `NEEDS_STREAM_DATA` to indicate that a stream must be written. After writing the stream, call the `status` method to find out if more stream data should be written, or whether the execution has completed.

In cases in which your statement includes multiple streamed parameters, use the `getCurrentStreamParam` method to discover which parameter needs to be written.

If you are performing an iterative or array execute, then the `getCurrentStreamIteration` method reveals to which iteration the data is to be written.

Once all the stream data has been handled, the status changes to either `RESULT_SET_AVAILABLE` or `UPDATE_COUNT_AVAILABLE`.

STREAM_DATA_AVAILABLE

This status indicates that the application requires some stream data to be read in `OUT` or `IN/OUT` parameters before the execution can finish. After reading the stream, call the `status` method to find out if more stream data should be read, or whether the execution has completed.

In cases in which your statement includes multiple streamed parameters, use the `getCurrentStreamParam` method to discover which parameter needs to be read.

If you are performing an iterative or array execute, then the `getCurrentStreamIteration` method reveals from which iteration the data is to be read.

Once all the stream data has been handled, the status changes to `UPDATE_COUNT_AVAILABLE`.

The `ResultSet` class also has readable streams and it operates similar to the readable streams of the `Statement` class.

Committing a Transaction

All SQL DML statements are executed in the context of a transaction. An application causes the changes made by these statement to become permanent by either committing the transaction, or undoing them by performing a rollback. While the SQL `COMMIT` and `ROLLBACK` statements can be executed with the `executeUpdate` method, you can also call the `Connection::commit` and `Connection::rollback` methods.

If you want the DML changes that were made to be committed immediately, you can turn on the auto commit mode of the `Statement` class by issuing the following statement:

```
Statement::setAutoCommit(TRUE)
```

Once auto commit is in effect, each change is automatically made permanent. This is similar to issuing a commit right after each execution.

To return to the default mode, auto commit off, issue the following statement:

```
Statement::setAutoCommit(FALSE).
```

Error Handling

Each OCCI method is capable of returning a return code indicating whether the method was successful or not. In other words, an OCCI method can throw an exception. This exception is of `type SQLException`. OCCI uses the C++ Standard Template Library (STL), so any exception that can be thrown by the STL can also be thrown by OCCI methods.

The STL exceptions are derived from the standard `exception` class. The `exception::what()` method returns a pointer to the error text. The error text is guaranteed to be valid during the catch block

The `SQLException` class contains Oracle specific error numbers and messages. It is derived from the standard `exception` class, so it too can obtain the error text by using the `exception::what()` method.

In addition, the `SQLException` class has two methods it can use to obtain error information. The `getErrorCode` method returns the Oracle error number. The same error text returned by `exception::what()` can be obtained by the `getMessage` method. The `getMessage` method returns an STL string so that it can be copied like any other STL string.

Based on your error handling strategy, you may choose to handle OCCI exceptions differently from standard exceptions, or you may choose not to distinguish between the two.

If you decide that it is not important to distinguish between OCCI exceptions and standard exceptions, your catch block might look similar to the following:

```
catch (exception &excp)
{
    cerr << excp.what() << endl;
}
```

Should you decide to handle OCCI exceptions differently than standard exceptions, your catch block might look like the following:

```
catch (SQLException &sqlExcp)
{
    cerr <<sqlExcp.getErrorCode << ": " << sqlExcp.getErrorMessage() << endl;
}
catch (exception &excp)
{
    cerr << excp.what() << endl;
}
```

In the preceding catch block, SQL exceptions are caught by the first block and non-SQL exceptions are caught by the second block. If the order of these two blocks were to be reversed, SQL exceptions would never be caught. Since `SQLException` is derived from the standard `exception`, the standard `exception` catch block would handle the SQL exception as well.

> **See Also:** *Oracle9i Database Error Messages* for more information about Oracle erro messages.

Null and Truncated Data

In general, OCCI does not cause an exception when the data value retrieved by using the `get`*`xxx`* methods of the `ResultSet` class or `Statement` class is null or truncated. However, this behavior can be changed by calling the `setErrorOnNull` method or `setErrorOnTruncate` method. If the `setError`*`xxx`* methods are called with `causeException=TRUE`, then an `SQLException` is raised when a data value is null or truncated.

The default behavior is to not raise an `SQLException`. In this case, null data is returned as zero (0) for numeric values and null strings for character values.

For data retrieved through the `setDataBuffer` method and `setDataBufferArray` method, exception handling behavior is controlled by the presence or absence of indicator variables and return code variables as shown in Table 2–1, Table 2–2, and Table 2–3.

Table 2–1 Normal Data - Not Null and Not Truncated

	Indicator - not provided	Indicator - provided
Return code - not provided	`error = 0`	`error = 0` `indicator = 0`

Return code - provided	`error = 0`
	`return code = 0`
		`error = 0`
		`indicator = 0`
		`return code = 0`

Table 2–2 Null Data

	Indicator - not provided	Indicator - provided
Return code - not provided	`SQLException` `error = 1405`	`error = 0` `indicator = -1`

Return code - provided	`SQLException`
	`error = 1405`
	`return code = 1405`
		`error = 0`
		`indicator = -1`
		`return code = 1405`

Table 2–3 Truncated Data

	Indicator - not provided	Indicator - provided

Return code - not provided	`SQLException`
	`error = 1406`
		`SQLException`
		`error = 1406`
		`indicator = data_len`

Return code - provided	`error = 24345`
	`return code = 1405`
		`error = 24345`
		`indicator = data_len`
		`return code = 1406`

In Table 2–3, `data_len` is the actual length of the data that has been truncated if this length is less than or equal to `SB2MAXVAL`. Otherwise, the indicator is set to -2.

Advanced Relational Techniques

The following advanced techniques are discussed in this section:

- Utilizing a Shared Server Environment
- Optimizing Performance

Utilizing a Shared Server Environment

Thread Safety

Threads are lightweight processes that exist within a larger process. Threads each share the same code and data segments, but have their own program counters, machine registers, and stack. Global and static variables are common to all threads, and a mutual exclusivity mechanism may be required to manage access to these variables from multiple threads within an application.

Once spawned, threads run asynchronously to one another. They can access common data elements and make OCCI calls in any order. Because of this shared access to data elements, a mechanism is required to maintain the integrity of data being accessed by multiple threads. The mechanism to manage data access takes the form of mutexes (mutual exclusivity locks), which ensure that no conflicts arise between multiple threads that are accessing shared resources within an application. In OCCI, mutexes are granted on an OCCI environment basis.

This thread safety feature of the Oracle database server and OCCI library enables developers to use OCCI in a shared server environment with these added benefits:

- Multiple threads of execution can make OCCI calls with the same result as successive calls made by a single thread.
- When multiple threads make OCCI calls, there are no side effects between threads.
- Even if you do not write a shared server program, you do not pay any performance penalty for including thread-safe OCCI calls.
- Use of multiple threads can improve program performance. You can discern gains on multiprocessor systems where threads run concurrently on separate

processors, and on single processor systems where overlap can occur between slower operations and faster operations.

Thread Safety and Three-Tier Architectures

In addition to client/server applications, where the client can be a shared server program, a typical use of shared server applications is in three-tier (also called client-agent-server) architectures. In this architecture, the client is concerned only with presentation services. The agent (or application server) processes the application logic for the client application. Typically, this relationship is a many-to-one relationship, with multiple clients sharing the same application server.

The server tier in the three-tier architecture is an Oracle database server. The applications server (agent) is very well suited to being a shared server application server, with each thread serving a client application. In an Oracle environment, this middle-tier application server is an OCCI or precompiler program.

Implementing Thread Safety

In order to take advantage of thread safety by using OCCI, an application must be running on a thread-safe platform. Then the application must inform OCCI that the application is running in shared server mode by specifying `THREADED_MUTEXED` or `THREADED_UNMUTEXED` for the mode parameter of the `createEnvironment` method. For example, to turn on mutual exclusivity locking, issue the following statement:

```
Environment *env = Environment::createEnvironment(Environment::THREADED_
MUTEXED);
```

Note that once `createEnvironment` is called with `THREADED_MUTEXED` or `THREADED_UNMUTEXED`, all subsequent calls to the `createEnvironment` method must also be made with `THREADED_MUTEXED` or `THREADED_UNMUTEXED` modes.

If a shared server application is running on a thread-safe platform, then the OCCI library will manage mutexes for the application on a for each-OCCI-environment basis. However, you can override this feature and have your application maintain its own mutex scheme. This is done by specifying a mode value of `THREADED_UNMUTEXED` to the `createEnvironment` method.

Note:

- Applications running on non-thread-safe platforms should not pass a value of `THREADED_MUTEXED` or `THREADED_UNMUTEXED` to the `createEnvironment` method.
- If an application is single threaded, whether or not the platform is thread safe, the application should pass a value of `Environment::DEFAULT` to the `createEnvironment` method. This is also the default value for the mode parameter. Single threaded applications which run in `THREADED_MUTEXED` mode may incur performance degradation.

Shared Server Concurrency

As an application programmer, you have two basic options regarding concurrency in a shared server environment:

- Automatic serialization, in which you utilize OCCI's transparent mechanisms
- Application-provided serialization, in which you manage the contingencies involved in maintaining multiple threads

Automatic Serialization In cases where there are multiple threads operating on objects (connections and connection pools) derived from an OCCI environment, you can elect to let OCCI serialize access to those objects. The first step is to pass a value of `THREADED_MUTEXED` to the `createEnvironment` method. At this point, the OCCI library automatically acquires a mutex on thread-safe objects in the environment.

When the OCCI environment is created with `THREADED_MUTEXED` mode, then only the `Environment`, `Map`, `ConnectionPool`, and `Connection` objects are thread-safe. That is, if two threads make simultaneous calls on one of these objects, then OCCI serializes them internally. However, note that all other OCCI objects, such as `Statement`, `ResultSet`, `SQLException`, `Stream`, and so on, are not thread-safe as, applications should not operate on these objects simultaneously from multiple threads.

Note that the bulk of processing for an OCCI call happens on the server, so if two threads that use OCCI calls go to the same connection, then one of them could be blocked while the other finishes processing at the server.

Application-Provided Serialization In cases where there are multiple threads operating on objects derived from an OCCI environment, you can chose to manage serialization. The first step is to pass a value of `THREADED_UNMUTEXED` for the `createEnvironment` mode. In this case the application must mutual exclusively lock OCCI calls made on objects derived from the same OCCI environment. This has the advantage that the mutex scheme can be optimized based on the application design to gain greater concurrency.

When an OCCI environment is created in this mode, OCCI recognizes that the application is running in a shared server environment, but that OCCI need not acquire its internal mutexes. OCCI assumes that all calls to methods of objects derived from that OCCI environment are serialized by the application. You can achieve this two different ways:

- Each thread has its own environment. That is, the environment and all objects derived from it (connections, connection pools, statements, result sets, and so on) are not shared across threads. In this case your application need not apply any mutexes.
- If the application shares an OCCI environment or any object derived from the environment across threads, then it must serialize access to those objects (by using a mutex, and so on) such that only one thread is calling an OCCI method on any of those objects.

Basically, in both cases, no mutexes are acquired by OCCI. You must ensure that only one OCCI call is in process on any object derived from the OCCI environment at any given time when `THREADED_UNMUTEXED` is used.

Note:

- OCCI is optimized to reuse handles as much as possible. Since each environment has its own heap, multiple environments result in increased consumption of memory. Having multiple environments may imply duplicating work with regard to connections, connection pools, statements, and result set objects. This will result in further memory consumption.
- Having multiple connections to the server results in more resource consumptions on the server and network. Having multiple environments would normally entail more connections.

Optimizing Performance

When you provide data for bind parameters by the `setxxx` methods in parameterized statements, the values are copied into an internal data buffer, and the copied values are then provided to the database server for insertion. This data copying may be expensive, especially if large strings are involved. Also, for each new value, the string is reallocated, so there may be memory management overhead in repeated allocation and deallocation of strings.

For these reasons, OCCI provides several methods to help counter these performance drains. These methods are:

- `setDataBuffer`
- `executeArrayUpdate`
- `next` (of the `ResultSet` class)

setDataBuffer Method

For high performance applications, OCCI provides the `setDataBuffer` method whereby the data buffer is managed by the application. The following example shows the `setDataBuffer` method:

```
void setDataBuffer(int paramIndex,
    void *buffer,
    Type type,
    sb4 size,
    ub2 *length,
    sb2 *ind = NULL,
    ub2 *rc = NULL);
```

The following parameters are used in the previous method example:

- `paramIndex`: Parameter number
- `buffer`: Data buffer containing data
- `type`: Type of the data in the data buffer
- `size`: Size of the data buffer
- `length`: Current length of data in the data buffer
- `ind`: Indicator information. This indicates whether the data is `NULL` or not. For parameterized statements, a value of -1 means a `NULL` value is to be inserted. For data returned from callable statements, a value of -1 means `NULL` data is retrieved.

- `rc`: Return code. This variable is not applicable to data provided to the `Statement` method. However, for data returned from callable statements, the return code specifies parameter-specific error numbers.

Not all datatypes can be provided and retrieved by means of the `setDataBuffer` method. For instance, C++ Standard Library strings cannot be provided with the `setDataBuffer` interface. Currently, only the following types can be provided or retrieved:

There is an important difference between the data provided by the `set`*xxx* methods and `setDataBuffer` method. When data is copied in the `set`*xxx* methods, the original can change once the data is copied. For example, you can use a `setString(str1)` method, then change the value of `str1` prior to execute. The value of `str1` that is used is the value at the time `setString(str1)` is called. However, for data provided by means of the `setDataBuffer` method, the buffer must remain valid until the execution is completed.

If iterative executes or the `executeArrayUpdate` method is used, then data for multiple rows and iterations can be provided in a single buffer. In this case, the data for the *i*th iteration is at `buffer + (i-1) *size address` and the `length`, indicator, and return codes are at `*(length + i)`, `*(ind + i)`, and `*(rc + i)` respectively.

This interface is also meant for use with array executions and callable statements that have array or `OUT` bind parameters.

The same method is available in the `ResultSet` class to retrieve data without re-allocating the buffer for each fetch.

executeArrayUpdate Method

If all data is provided with the `setDataBuffer` methods or output streams (that is, no `set`*xxx* methods besides `setDataBuffer` or `getStream` are called), then there is a simplified way of doing iterative execution.

In this case, you should not call `setMaxIterations` and `setMaxParamSize`. Instead call the `setDataBuffer` (or `getStream`) method for each parameter with the appropriate size arrays to provide data for each iteration, followed by the `executeArrayUpdate(int arrayLength)` method. The `arrayLength` parameter specifies the number of elements provided in each buffer. Essentially, this is same as setting the number of iterations to `arrayLength` and executing the statement.

Since the stream parameters are specified only once, they can be used with array executes as well. However, if any `set`*xxx* methods are used, then the

`addIteration` method is called to provide data for multiple rows. To compare the two approaches, consider an example that inserts two employees in the `emp` table:

```
Statement *stmt = conn->createStatement("insert into emp (id, ename)
                                          values(:1, :2)");
char enames[2][] = {"SMITH", "MARTIN"};
ub2 enameLen[2];
for (int i = 0; i < 2; i++)
    enameLen[i] = strlen(enames[i] + 1);
stmt->setMaxIteration(2);               // set maximum number of iterations
stmt->setInt(1, 7369);                  // specify data for the first row
stmt->setDataBuffer(2, enames, OCCI_SQLT_STR, sizeof(ename[0]), &enameLen);
stmt->addIteration();
stmt->setInt(1, 7654);                  // specify data for the second row
// a setDatBuffer is unnecessary for the second bind parameter as data
// provided through setDataBuffer is specified only once.
stmt->executeUpdate();
```

However, if the first parameter could also be provided through the `setDataBuffer` interface, then, instead of the `addIteration` method, you would use the `executeArrayUpdate` method:

```
stmt ->setSQL("insert into emp (id, ename) values (:1, :2)");
char enames[2][] = {"SMITH", "MARTIN"};
ub2 enameLen[2];
for (int i = 0; i < 2; i++)
    enameLen[i] = strlen(enames[i] + 1);
int ids[2] = {7369, 7654};
ub2 idLen[2] = {sizeof(ids[0], sizeof(ids[1])};
stmt->setDataBuffer(1, ids, OCCIINT, sizeof(ids[0]), &idLen);
stmt->setDataBuffer(2, enames, OCCI_SQLT_STR, sizeof(ename[0]), &len);
stmt->executeArrayUpdate(2);            // data for two rows is inserted.
```

Array Fetch Using next Method

If the application is fetching data with only the `setDataBuffer` interface or the stream interface, then an array fetch can be executed. The array fetch is implemented by calling the `ResultSet->next(int numRows)` method. This causes up to `numRows` amount of data is fetched for each column. The buffers specified with the `setDataBuffer` interface should be big enough to hold data for multiple rows. Data for the *i*th row is fetched at `buffer + (i - 1) * size` location. Similarly, the length of the data is stored at `*(length + (i - 1))`.

```
int empno[5];
char ename[5][11];
```

```
ub2  enameLen[5];
ResultSet *resultSet = stmt->executeQuery("select empno, ename  from emp");
resultSet->setDataBuffer(1, &empno, OCCIINT);
resultSet->setDataBuffer(2, ename, OCCI_SQLT_STR, sizeof(ename[0]), enameLen);
rs->next(5);           // fetches five rows, enameLen[i] has length of ename[i]
```

3 Object Programming

This chapter provides information on how to implement object-relational programming using the Oracle C++ Call Interface (OCCI).

The following topics are discussed:

- Overview of Object Programming
- Working with Objects in OCCI
- Representing Objects in C++ Applications
- Developing an OCCI Object Application
- Overview of Associative Access
- Overview of Navigational Access
- Overview of Complex Object Retrieval
- Working with Collections
- Using Object References
- Freeing Objects
- Type Inheritance
- A Sample OCCI Application

Overview of Object Programming

OCCI supports both the associative and navigational style of data access. Traditionally, third-generation language (3GL) programs manipulate data stored in a database by using the **associative access** based on the associations organized by relational database tables. In associative access, data is manipulated by executing SQL statements and PL/SQL procedures. OCCI supports associative access to objects by enabling your applications to execute SQL statements and PL/SQL procedures on the database server without incurring the cost of transporting data to the client.

Object-oriented programs that use OCCI can also make use of **navigational access** that is a key aspect of this programming paradigm. Applications model their objects as a set of interrelated objects that form graphs of objects, the relationships between objects implemented as references (REFs). Typically, an object application that uses navigational access first retrieves one or more objects from the database server by issuing a SQL statement that returns REFs to those objects. The application then uses those REFs to traverse related objects, and perform computations on these other objects as required. Navigational access does not involve executing SQL statements except to fetch the references of an initial set of objects. By using OCCI's API for navigational access, your application can perform the following functions on Oracle objects:

- Creating, accessing, locking, deleting, copying and flushing objects
- Getting references to objects and navigating through the references

This chapter gives examples that show you how to create a persistent object, access an object, modify an object, and flush the changes to the database server. It discusses how to access the object using both navigational and associative approaches.

Working with Objects in OCCI

Many of the programming principles that govern a relational OCCI application are the same for an object-relational application. An object-relational application uses the standard OCCI calls to establish database connections and process SQL statements. The difference is that the SQL statements that are issued retrieve object references, which can then be manipulated with OCCI's object functions. An object can also be directly manipulated as a value (without using its object reference).

Instances of an Oracle type are categorized into **persistent objects** and **transient objects** based on their lifetime. Instances of persistent objects can be further divided

into **standalone objects** and **embedded objects** depending on whether or not they are referenced by way of an object identifier.

Persistent Objects

A **persistent object** is an object which is stored in an Oracle database. It may be fetched into the object cache and modified by an OCCI application. The lifetime of a persistent object can exceed that of the application which is accessing it. Once it is created, it remains in the database until it is explicitly deleted. There are two types of persistent objects:

- A **standalone instance** is stored in a database table row, and has a unique object identifier. An OCCI application can retrieve a reference to a standalone object, pin the object, and navigate from the pinned object to other related objects. Standalone objects may also be referred to as **referenceable objects**.

  It is also possible to select a referenceable object, in which case you fetch the object *by value* instead of fetching it by reference.

- An **embedded instance** is not stored in a database table row, but rather is embedded within another structure. Examples of embedded objects are objects which are attributes of another object, or objects that exist in an object column of a database table. Embedded objects do not have object identifiers, and OCCI applications cannot get REFs to embedded instances.

  Embedded objects may also be referred to as **nonreferenceable objects** or **value instances**. You may sometimes see them referred to as **values**, which is not to be confused with scalar data values. The context should make the meaning clear.

The following SQL examples demonstrate the difference between these two types of persistent objects.

Creating Standalone Objects: Example

This code example demonstrates how a standalone object is created:

```
CREATE TYPE person_t AS OBJECT
    (name      varchar2(30),
     age       number(3));
CREATE TABLE person_tab OF person_t;
```

Objects that are stored in the object table `person_tab` are standalone objects. They have object identifiers and can be referenced. They can be pinned in an OCCI application.

Creating Embedded Objects: Example

This code example demonstrates how an embedded object is created:

```
CREATE TABLE department
    (deptno     number,
     deptname   varchar2(30),
     manager    person_t);
```

Objects which are stored in the `manager` column of the `department` table are embedded objects. They do not have object identifiers, and they cannot be referenced. This means they cannot be pinned in an OCCI application, and they also never need to be unpinned. They are always retrieved into the object cache *by value*.

Transient Objects

A **transient object** is an instance of an object type. Its lifetime cannot exceed that of the application. The application can also delete a transient object at any time.

The Object Type Translator (OTT) utility generates two `operator new` methods for each C++ class, as demonstrated in this code example:

```
class Person : public PObject {
    .
    .
    .
public:
     dvoid *operator new(size_t size);    // creates transient instance
     dvoid *operator new(size_t size, Connection &conn, string table);
                                          // creates persistent instance
}
```

The following code example demonstrates how a transient object can be created:

```
Person *p = new Person();
```

Transient objects cannot be converted to persistent objects. Their role is fixed at the time they are instantiated.

See Also:

- *Oracle9i Database Concepts* for more information about objects

Values

In the context of this manual, a **value** refers to either:

- A scalar value which is stored in a nonobject column of a database table. An OCCI application can fetch values from a database by issuing SQL statements.
- An embedded (nonreferenceable) object.

The context should make it clear which meaning is intended.

Note: It is possible to `SELECT` a referenceable object into the object cache, rather than pinning it, in which case you fetch the object *by value* instead of fetching it by reference.

Representing Objects in C++ Applications

Before an OCCI application can work with object types, those types must exist in the database. Typically, you create types with SQL DDL statements, such as `CREATE TYPE`.

Creating Persistent and Transient Objects

The following sections discuss how persistent and transient objects are created.

Creating a Persistent Object

Before you create a persistent object, you must have created the environment and opened a connection. The following example shows how to create a persistent object, `addr`, in the database table, `addr_tab`, created by means of a SQL statement:

```
CREATE TYPE ADDRESS AS OBJECT (state CHAR(2), zip_code CHAR(5));
CREATE TABLE ADDR_TAB of ADDRESS;
ADDRESS *addr = new(conn, "ADDR_TAB") ADDRESS("CA", "94065");
```

The persistent object is created in the database only when one of the following occurs:

- The transaction is committed (`Connection::commit()`)
- The object cache is flushed (`Connection::flushCache()`)
- The object itself is flushed (`PObject::flush()`)

Creating a Transient Object

An instance of the transient object `ADDRESS` is created in the following manner:

```
ADDRESS *addr_trans = new ADDRESS("MD", "94111");
```

Creating Object Representations using the OTT Utility

When your C++ application retrieves instances of object types from the database, it needs to have a client-side representation of the objects. The Object Type Translator (OTT) utility generates C++ class representations of database object types for you. For example, consider the following declaration of a type in your database:

```
CREATE TYPE address AS OBJECT (state CHAR(2), zip_code CHAR(5));
```

The OTT utility produces the following C++ class:

```
class ADDRESS : public PObject {

protected:
    string state;
    string zip;

public:
    void *operator new(size_t size);
    void *operator new(size_t size, const Session* sess, const string&
        table);
    string getSQLTypeName(size_t size);
    ADDRESS(void *ctx) : PObject(ctx) { };
    static void *readSQL(void *ctx);
    virtual void readSQL(AnyData& stream);
    static void writeSQL(void *obj, void *ctx);
    virtual void writeSQL(AnyData& stream);
}
```

These class declarations are automatically written by OTT to a header (`.h`) file that you name. This header file is included in the source files for an application to provide access to objects. Instances of a `PObject` (as well as instances of classes derived from `PObject`s) can be either transient or persistent. The methods `writeSQL` and `readSQL` are used internally by the OCCI object cache to linearize and delinearize the objects and are not to be used or modified by OCCI clients.

See Also:

- Chapter 7, "How to Use the Object Type Translator Utility" for more information about the OTT utility

Developing an OCCI Object Application

This section discusses the steps involved in developing a basic OCCI object application.

Basic Object Program Structure

The basic structure of an OCCI application that uses objects is similar to a relational OCCI application, the difference being object functionality. The steps involved in an OCCI object program include:

1. Initialize the `Environment`. Initialize the OCCI programming environment in object mode.

   Your application will most likely need to include C++ class representations of database objects in a header file. You can create these classes by using the Object Type Translator (OTT) utility, as described in Chapter 7, "How to Use the Object Type Translator Utility".

2. Establish a Connection. Use the environment handle to establish a connection to the database server.

3. Prepare a SQL statement. This is a local (client-side) step, which may include binding placeholders. In an object-relational application, this SQL statement should return a reference (`REF`) to an object.

4. Access the object.

   a. Associate the prepared statement with a database server, and execute the statement.

   b. By using navigational access, retrieve an object reference (`REF`) from the database server and pin the object. You can then perform some or all of the following:

      * Manipulate the attributes of an object and mark it as **dirty** (modified)
      * Follow a reference to another object or series of objects
      * Access type and attribute information
      * Navigate a complex object retrieval graph

* Flush modified objects to the database server

c. By using associative access, you can fetch an entire object *by value* by using SQL. Alternately, you can select an embedded (nonreferenceable) object. You can then perform some or all of the following:

* Insert values into a table
* Modify existing values

5. Commit the transaction. This step implicitly writes all modified objects to the database server and commits the changes.

6. Free statements and handles not to be reused or reexecute prepared statements again.

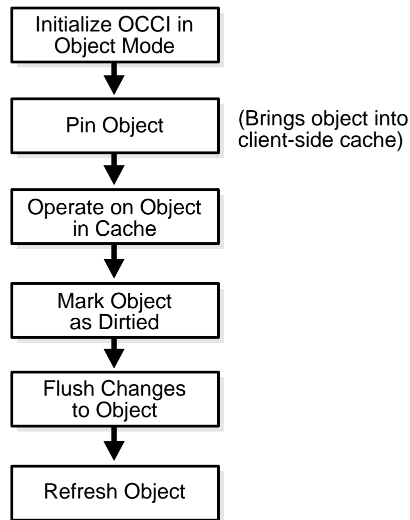
See Also:

- Chapter 2, "Relational Programming" for information about using OCCI to connect to a database server, process SQL statements, and allocate handles
- Chapter 7, "How to Use the Object Type Translator Utility" for information about the OTT utility
- Chapter 8, "OCCI Classes and Methods" for descriptions of OCCI relational functions and the `Connect` class and the `getMetaData` method

Basic Object Operational Flow

Figure 3–1 shows a simple program logic flow for how an application might work with objects. For simplicity, some required steps are omitted.

Figure 3–1 Basic Object Operational Flow



The steps shown in Figure 3–1 are discussed in the following sections:

Initialize OCCI in Object Mode

If your OCCI application accesses and manipulates objects, then it is essential that you specify a value of `OBJECT` for the `mode` parameter of the `createEnvironment` method, the first call in any OCCI application. Specifying this value for `mode` indicates to OCCI that your application will be working with objects. This notification has the following important effects:

- The object run-time environment is established
- The object cache is set up

Note: If the `mode` parameter is not set to `OBJECT`, any attempt to use an object-related function will result in an error.

The following code example demonstrates how to specify the `OBJECT` `mode` when creating an OCCI environment:

```
Environment *env;
```

```
Connection *con;
Statement *stmt;

env = Environment::createEnvironment(Environment::OBJECT);
con = env->createConnection(userName, password, connectString);
```

Your application does not have to allocate memory when database objects are loaded into the object cache. The object cache provides transparent and efficient memory management for database objects. When database objects are loaded into the object cache, they are transparently mapped into the host language (C++) representation.

The object cache maintains the association between the object copy in the object cache and the corresponding database object. Upon `commit`, changes made to the object copy in the object cache are automatically propagated back to the database.

The object cache maintains a look-up table for mapping references to objects. When an application dereferences a reference to an object and the corresponding object is not yet cached in the object cache, the object cache automatically sends a request to the database server to fetch the object from the database and load it into the object cache. Subsequent dereferences of the same reference are faster since they are to the object cache itself and do not incur a round-trip to the database server.

The object cache maintains a pin count for each persistent object in the object cache. When an application dereferences a reference to an object, the pin count of the object pointed to by the reference is incremented. The subsequent dereferencing of the same reference to the object does not change the pin count. Until the reference to the object goes out of scope, the object will continue to be pinned in the object cache and be accessible by the OCCI client.

The pin count functions as a reference count for the object. The pin count of the object becomes zero (0) only when there are no more references referring to this object, during which time the object becomes eligible for garbage collection. The object cache uses a least recently used algorithm to manage the size of the object cache. This algorithm frees objects with a pin count of 0 when the object cache reaches the maximum size.

Pin Object

In most situations, OCCI users do not need to explicitly pin or unpin the objects because the object cache automatically keeps track of the pin counts of all the objects in the cache. As explained earlier, the object cache increments the pin count when a reference points to the object and decrements it when the reference goes out of scope or no longer points to the object.

But there is one exception. If an OCCI application uses `Ref<T>::ptr()` method to get a pointer to the object, then the `pin` and `unpin` methods of the `PObject` class can be used by the application to control pinning and unpinning of the objects in the object cache.

Operate on Object in Cache

Note that the object cache does not manage the contents of object copies; it does not automatically refresh object copies. Your application must ensure the validity and consistency of object copies.

Flush Changes to Object

Whenever changes are made to object copies in the object cache, your application is responsible for flushing the changed object to the database.

Memory for the object cache is allocated on demand when objects are loaded into the object cache.

The client-side object cache is allocated in the program's process space. This object cache is the memory for objects that have been retrieved from the database server and are available to your application.

Note: If you initialize the OCCI environment in object mode, your application allocates memory for the object cache, whether or not the application actually uses object calls.

There is only one object cache allocated for each OCCI environment. All objects retrieved or created through different connections within the environment use the same physical object cache. Each connection has its own logical object cache.

Overview of Associative Access

You can employ SQL within OCCI to retrieve objects, and to perform DML operations:

- Using SQL to Access Objects
- Inserting and Modifying Values

See Also:

- Appendix A, "OCCI Demonstration Programs" and the code examples occiobj.typ and occiobj.cpp for an illustration of the concepts covered in this section

Using SQL to Access Objects

In the previous sections we discussed navigational access, where SQL is used only to fetch the references of an initial set of objects and then navigate from them to the other objects. Here we will discuss how to fetch the objects using SQL.

The following example shows how to use the `ResultSet::getObject` method to fetch objects through associative access where it gets each object from the table, `addr_tab`, using SQL:

```
string sel_addr_val = "SELECT VALUE(address) FROM ADDR_TAB address";

ResultSet *rs = stmt->executeQuery(sel_addr_val);

while (rs->next())
{
    ADDRESS *addr_val = rs->getObject(1);
    cout << "state: " << addr_val->getState();
}
```

The objects fetched through associative access are termed value instances and they behave just like transient objects. Methods such as `markModified`, `flush`, and `markDeleted` are applicable only for persistent objects.

Any changes made to these objects are not reflected in the database.

Inserting and Modifying Values

We have just seen how to use SQL to access objects. OCCI also provides the ability to use SQL to insert new objects or modify existing objects in the database server through the `Statement::setObject` method interface.

The following example creates a transient object `Address` and inserts it into the database table `addr_tab`:

```
ADDRESS *addr_val = new address("NV", "12563");  // new a transient instance
stmt->setSQL("INSERT INTO ADDR_TAB values(:1)");
stmt->setObject(1, addr_val);
stmt->execute();
```

Overview of Navigational Access

By using navigational access, you engage in a series of operations:

- Retrieving an Object Reference (REF) from the Database Server
- Pinning an Object
- Manipulating Object Attributes
- Marking Objects and Flushing Changes

See Also:

- Appendix A, "OCCI Demonstration Programs" and the code examples occipobj.typ and occipobj.cpp for an illustration of the concepts covered in this section

Retrieving an Object Reference (REF) from the Database Server

In order to work with objects, your application must first retrieve one or more objects from the database server. You accomplish this by issuing a SQL statement that returns references (REFs) to one or more objects.

Note: It is also possible for a SQL statement to fetch embedded objects, rather than REFs, from a database.

The following SQL statement retrieves a REF to a single object address from the database table addr_tab:

```
string sel_addr = "SELECT REF(address) FROM addr_tab address
    WHERE zip_code = '94065'";
```

The following code example illustrates how to execute the query and fetch the REF from the result set.

```
ResultSet *rs = stmt->executeQuery(sel_addr);
rs->next();
Ref<address> addr_ref = rs->getRef(1);
```

At this point, you could use the object reference to access and manipulate the object or objects from the database.

See Also:

- "Executing SQL DDL and DML Statements" on page 2-6 for general information about preparing and executing SQL statements

Pinning an Object

Upon completion of the fetch step, your application has a `REF` to an object. The actual object is not currently available to work with. Before you can manipulate an object, it must be **pinned**. Pinning an object loads the object into the object cache, and enables you to access and modify the object's attributes and follow references from that object to other objects. Your application also controls when modified objects are written back to the database server.

Note: This section deals with a simple pin operation involving a single object at a time. For information about retrieving multiple objects through complex object retrieval, see the section Overview of Complex Object Retrieval on page 3-16.

OCCI requires only that you dereference the `REF` in the same way you would dereference any C++ pointer. Dereferencing the `REF` transparently materializes the object as a C++ class instance.

Continuing the `Address` class example from the previous section, assume that the user has added the following method:

```
string  Address::getState()
{
    return state;
}
```

To dereference this `REF` and access the object's attributes and methods:

```
string state = addr_ref->getState();     // -> pins the object
```

The first time `Ref<T> (addr_ref)` is dereferenced, the object is pinned, which is to say that it is loaded into the object cache from the database server. From then on, the behavior of `operator ->` on `Ref<T>` is just like that of any C++ pointer `(T *)`. The object remains in the object cache until the `REF (addr_ref)` goes out of scope. It then becomes eligible for garbage collection.

Now that the object has been pinned, your application can modify that object.

Manipulating Object Attributes

Manipulating object attributes is no different from that of accessing them as shown in the previous section. Let us assume the `Address` class has the following user defined method that sets the `state` attribute to the input value:

```
void Address::setState(string new_state)
{
   state = new_state;
}
```

The following example shows how to modify the state attribute of the object, addr:

```
addr_ref->setState("PA");
```

As explained earlier, the first invocation of the `operator ->` on `Ref<T>` loads the object if not already in the object cache.

Marking Objects and Flushing Changes

In the example in the previous section, an attribute of an object was changed. At this point, however, that change exists only in the client-side cache. The application must take specific steps to ensure that the change is written to the database.

Marking an Object as Modified (Dirty)

The first step is to indicate that the object has been modified. This is done by calling the `markModified` method on the object (derived method of `PObject`). This method marks the object as **dirty** (modified).

Continuing the previous example, after object attributes are manipulated, the object referred to by `addr_ref` can be marked dirty as follows:

```
addr_ref->markModified()
```

Recording Changes in the Database

Objects that have had their dirty flag set must be flushed to the database server for the changes to be recorded in the database. This can be done in three ways:

- Flush a single object marked dirty by calling the method `flush`, a derived method of `PObject`.
- Flush the entire object cache using the `Connection::flushCache` method. In this case, OCCI traverses the dirty list maintained by the object cache and flushes all the dirty objects.
- Commit a transaction by calling the `Connection::commit` method. Doing so also traverses the dirty list and flushes the objects to the database server. The dirty list includes newly created persistent objects

Overview of Complex Object Retrieval

In the examples discussed earlier, only a single object was fetched or pinned at a time. In these cases, each pin operation involved a separate database server round-trip to retrieve the object.

Object-oriented applications often model their problems as a set of interrelated objects that form graphs of objects. These applications process objects by starting with some initial set of objects and then using the references in these objects to traverse the remaining objects. In a client/server setting, each of these traversals could result in costly network round-trips to fetch objects.

The performance of such applications can be increased through the use of **complex object retrieval** (COR). This is a prefetching mechanism in which an application specifies some criteria (content and boundary) for retrieving a set of linked objects in a single network round-trip.

Note: Using COR does not mean that these prefetched objects are pinned. They are fetched into the object cache, so that subsequent pin calls are local operations.

A **complex object** is a set of logically related objects consisting of a root object, and a set of objects each of which is prefetched based on a given depth level. The **root object** is explicitly fetched or pinned. The **depth level** is the shortest number of references that need to be traversed from the root object to a given prefetched object in a complex object.

An application specifies a complex object by describing its content and boundary. The fetching of complex objects is constrained by an environment's **prefetch limit**, the amount of memory in the object cache that is available for prefetching objects.

Note: The use of complex object retrieval does not add functionality; it only improves performance, and so its use is optional.

Retrieving Complex Objects

An OCCI application can achieve COR by setting the appropriate attributes of a `Ref<T>` before dereferencing it using the following methods:

```
// prefetch attributes of the specified type name up to the the specified depth
Ref<T>::setPrefetch(const string &typeName, unsigned int depth);
// prefetch all the attribute types up to the specified depth.
Ref<T>::setPrefetch(unsigned int depth);
```

The application can also choose to fetch all objects reachable from the root object by way of REFs (transitive closure) to a certain depth. To do so, set the level parameter to the depth desired. For the preceding two examples, the application could also specify `(PO object REF, OCCI_MAX_PREFETCH_DEPTH)` and `(PO object REF, 1)` respectively to prefetch required objects. Doing so results in many extraneous fetches but is quite simple to specify, and requires only one database server round-trip.

As an example for this discussion, consider the following type declaration:

```
CREATE TYPE customer(...);
CREATE TYPE line_item(...);
CREATE TYPE line_item_varray as VARRAY(100) of REF line_item;
CREATE TYPE purchase_order AS OBJECT
( po_number          NUMBER,
  cust               REF customer,
  related_orders     REF purchase_order,
  line_items         line_item_varray)
```

The `purchase_order` type contains a scalar value for `po_number`, a `VARRAY` of `line_items`, and two references. The first is to a `customer` type and the second is to a `purchase_order` type, indicating that this type may be implemented as a linked list.

When fetching a complex object, an application must specify the following:

- A reference to the desired root object

- One or more pairs of type and depth information to specify the boundaries of the complex object. The type information indicates which REF attributes should be followed for COR, and the depth level indicates how many levels deep those links should be followed.

In the case of the purchase_order object in the preceding example, the application must specify the following:

- The reference to the root purchase_order object
- One or more pairs of type and depth information for customer, purchase_order, or line_item

An application prefetching a purchase order will very likely need access to the customer information for that purchase order. Using simple navigation, this would require two database server accesses to retrieve the two objects.

Through complex object retrieval, customer can be prefetched when the application pins the purchase_order object. In this case, the complex object would consist of the purchase_order object and the customer object it references.

In the previous example, if the application wanted to prefetch a purchase order and the related customer information, the application would specify the purchase_order object and indicate that customer should be followed to a depth level of one as follows:

```
Ref<PURCHASE_ORDER> poref;
poref.setPrefetch("CUSTOMER",1);
```

If the application wanted to prefetch a purchase order and all objects in the object graph it contains, the application would specify the purchase_order object and indicate that both customer and purchase_order should be followed to the maximum depth level possible as follows:

```
Ref<PURCHASE_ORDER> poref;
poref.setPrefetch("CUSTOMER", OCCI_MAX_PREFETCH_DEPTH);
poref.setPrefetch("PURCHASE_ORDER", OCCI_MAX_PREFETCH_DEPTH);
```

where OCCI_MAX_PREFETCH_DEPTH specifies that all objects of the specified type reachable through references from the root object should be prefetched.

If an application wanted to prefetch a purchase order and all the line items associated with it, the application would specify the purchase_order object and

indicate that `line_items` should be followed to the maximum depth level possible as follows:

```
Ref<PURCHASE_ORDER> poref;
poref.setPrefetch("LINE_ITEM", 1);
```

Prefetching Complex Objects

After specifying and fetching a complex object, subsequent fetches of objects contained in the complex object do not incur the cost of a network round-trip, because these objects have already been prefetched and are in the object cache. Keep in mind that excessive prefetching of objects can lead to a flooding of the object cache. This flooding, in turn, may force out other objects that the application had already pinned leading to a performance degradation instead of performance improvement.

Note: If there is insufficient memory in the object cache to hold all prefetched objects, some objects may not be prefetched. The application will then incur a network round-trip when those objects are accessed later.

The `SELECT` privilege is needed for all prefetched objects. Objects in the complex object for which the application does not have `SELECT` privilege will not be prefetched.

Working with Collections

Oracle supports two kinds of collections - variable length arrays (ordered collections) and nested tables (unordered collections). OCCI maps both of them to a Standard Template Library (STL) vector container, giving you the full power, flexibility, and speed of an STL vector to access and manipulate the collection elements. The following is the SQL DDL to create a `VARRAY` and an object that contains an attribute of type `VARRAY`.

```
CREATE TYPE ADDR_LIST AS VARRAY(3) OF REF ADDRESS;
CREATE TYPE PERSON AS OBJECT (name VARCHAR2(20), addr_l ADDR_LIST);
```

Here is the C++ class declaration generated by OTT:

```
class PERSON : public PObject
{
```

```
    protected:
        string name;
        vector< Ref< ADDRESS > > addr_1;

    public:
        void *operator new(size_t size);
        void *operator new(size_t size,
        const Session* sess,
        const string& table);
        string getSQLTypeName(size_t size);
        PERSON (void *ctx) : PObject(ctx) { };
        static void *readSQL(void *ctx);
        virtual void readSQL(AnyData& stream);
        static void writeSQL(void *obj, void *ctx);
        virtual void writeSQL(AnyData& stream);
}
```

See Also:

- Appendix A, "OCCI Demonstration Programs" and the code examples occicoll.cpp for an illustration of the concepts covered in this section

Fetching Embedded Objects

If your application needs to fetch an embedded object—an object stored in a column of a regular table, rather than an object table—you cannot use the `REF` retrieval mechanism. Embedded instances do not have object identifiers, so it is not possible to get a reference to them. This means that they cannot serve as the basis for object navigation. There are still many situations, however, in which an application will want to fetch embedded instances.

For example, assume that an `address` type has been created.

```
CREATE TYPE address AS OBJECT
( street1             varchar2(50),
  street2             varchar2(50),
  city                varchar2(30),
  state               char(2),
  zip                 number(5))
```

You could then use that type as the datatype of a column in another table:

```
CREATE TABLE clients
```

```
( name          varchar2(40),
  addr          address)
```

Your OCCI application could then issue the following SQL statement:

```
SELECT addr FROM clients
WHERE name='BEAR BYTE DATA MANAGEMENT'
```

This statement would return an embedded `address` object from the `clients` table. The application could then use the values in the attributes of this object for other processing. The application should execute the statement and fetch the object in the same way as described in the section "Overview of Associative Access" on page 3-11.

Nullness

If a column in a row of a database table has no value, then that column is said to be NULL, or to contain a NULL. Two different types of NULLs can apply to objects:

- Any attribute of an object can have a NULL value. This indicates that the value of that attribute of the object is not known.
- An object may be **atomically NULL**. This means that the value of the entire object is unknown.

Atomic nullness is not the same thing as nonexistence. An atomically NULL object still exists, its value is just not known. It may be thought of as an existing object with no data.

For every type of object attribute, OCCI provides a corresponding class. For instance, `NUMBER` attribute type maps to the `Number` class, `REF` maps to `RefAny`, and so on. Each and every OCCI class that represents a data type provides two methods:

- `isNull` — returns whether the object is null
- `setNull` — sets the object to null

Similarly, these methods are inherited from the `PObject` class by all the objects and can be used to access and set atomically null information about them.

Using Object References

OCCI provides the application with the flexibility to access the contents of the objects using their pointers or their references. OCCI provides the `PObject::getRef` method to return a reference to a persistent object. This call is valid for persistent objects only.

Freeing Objects

OCCI users can use the overloaded `PObject::operator new` to create the persistent objects. It is the user's responsibility to free the object by calling the `PObject::operator delete` method.

Note that freeing the object from the object cache is different from deleting the object from the database server. To delete the object from the database server, the user needs to call the `PObject::markDelete` method. The operator `delete` just frees the object and reclaims the memory in the object cache but it does not delete the object from the database server.

Type Inheritance

Type inheritance of objects has many similarities to inheritance in C++ and Java. You can create an object type as a subtype of an existing object type. The subtype is said to inherit all the attributes and methods (member functions and procedures) of the supertype, which is the original type. Only single inheritance is supported; an object cannot have more than one supertype. The subtype can add new attributes and methods to the ones it inherits. It can also override (redefine the implementation) of any of its inherited methods. A subtype is said to extend (that is, inherit from) its supertype.

See Also:

- *Oracle9i Application Developer's Guide - Object-Relational Features* for a more complete discussion of this topic

As an example, a type `Person_t` can have a subtype `Student_t` and a subtype `Employee_t`. In turn, `Student_t` can have its own subtype, `PartTimeStudent_t`. A type declaration must have the flag `NOT FINAL` so that it can have subtypes. The default is `FINAL`, which means that the type can have no subtypes.

All types discussed so far in this chapter are `FINAL`. All types in applications developed before release 9.0 are `FINAL`. A type that is `FINAL` can be altered to be `NOT FINAL`. A `NOT FINAL` type with no subtypes can be altered to be `FINAL`. `Person_t` is declared as `NOT FINAL` for our example:

```
CREATE TYPE Person_t AS OBJECT
( ssn NUMBER,
  name VARCAHR2(30),
  address VARCHAR2(100)) NOT FINAL;
```

A subtype inherits all the attributes and methods declared in its supertype. It can also declare new attributes and methods, which must have different names than those of the supertype. The keyword `UNDER` identifies the supertype, like this:

```
CREATE TYPE Student_t UNDER Person_t
( deptid NUMBER,
  major  VARCHAR2(30)) NOT FINAL;
```

The newly declared attributes `deptid` and `major` belong to the subtype `Student_t`. The subtype `Employee_t` is declared as, for example:

```
CREATE TYPE Employee_t UNDER Person_t
( empid NUMBER,
  mgr   VARCHAR2(30));
```

See Also:

- "OTT Support for Type Inheritance" on page 3-25 for the classes generated by OTT for this example.

This subtype `Student_t`, can have its own subtype, such as `PartTimeStudent_t`:

```
CREATE TYPE PartTimeStuden_t UNDER Student_t
( numhours NUMBER) ;
```

See Also:

- Appendix A, "OCCI Demonstration Programs" and the code examples occiinh.typ and occiinh.cpp for an illustration of the concepts covered in this section

Substitutability

The benefits of polymorphism derive partially from the property substitutability. Substitutability allows a value of some subtype to be used by code originally written for the supertype, without any specific knowledge of the subtype being needed in advance. The subtype value behaves to the surrounding code just like a value of the supertype would, even if it perhaps uses different mechanisms within its specializations of methods.

Instance substitutability refers to the ability to use an object value of a subtype in a context declared in terms of a supertype. REF substitutability refers to the ability to use a `REF` to a subtype in a context declared in terms of a `REF` to a supertype.

`REF` type attributes are substitutable, that is, an attribute defined as `REF T` can hold a `REF` to an instance of `T` or any of its subtypes.

Object type attributes are substitutable, that is, an attribute defined to be of (an object) type `T` can hold an instance of `T` or any of its subtypes.

Collection element types are substitutable, that is, if we define a collection of elements of type `T`, then it can hold instances of type `T` and any of its subtypes. Here is an example of object attribute substitutability:

```
CREATE TYPE Book_t AS OBJECT
( title VARCHAR2(30),
  author Person_t     /* substitutable */);
```

Thus, a Book_t instance can be created by specifying a title string and a `Person_t` (or any subtype of `Person_t`) object:

```
Book_t('My Oracle Experience',
        Employee_t(12345, 'Joe', 'SF', 1111, NULL))
```

NOT INSTANTIABLE Types and Methods

A type can be declared `NOT INSTANTIABLE`, which means that there is no constructor (default or user defined) for the type. Thus, it will not be possible to construct instances of this type. The typical usage would be to define instantiable subtypes for such a type. Here is how this property is used:

```
CREATE TYPE Address_t AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USAddress_t UNDER Address_t(...);
CREATE TYPE IntlAddress_t UNDER Address_t(...);
```

A method of a type can be declared to be NOT INSTANTIABLE. Declaring a method as NOT INSTANTIABLE means that the type is not providing an implementation for that method. Further, a type that contains any NOT INSTANTIABLE methods must necessarily be declared as NOT INSTANTIABLE. For example:

```
CREATE TYPE T AS OBJECT
(
  x NUMBER,
  NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
) NOT INSTANTIABLE;
```

A subtype of NOT INSTANTIABLE can override any of the NOT INSTANTIABLE methods of the supertype and provide concrete implementations. If there are any NOT INSTANTIABLE methods remaining, the subtype must also necessarily be declared as NOT INSTANTIABLE.

A NOT INSTANTIABLE subtype can be defined under an instantiable supertype. Declaring a NOT INSTANTIABLE type to be FINAL is not useful and is not allowed.

OCCI Support for Type Inheritance

The following calls support type inheritance.

Connection::getMetaData()

This method provides information specific to inherited types. Additional attributes have been added for the properties of inherited types. For example, you can get the supertype of a type.

Bind and Define Functions

The setRef, setObject and setVector methods of the Statement class are used to bind REF, object, and collections respectively. All these functions support REF, instance, and collection element substitutability. Similarly, the corresponding get*xxx* methods to fetch the data also support substitutability.

OTT Support for Type Inheritance

Class declarations for objects with inheritance are similar to the simple object declarations except that the class is derived from the parent type class and only the fields corresponding to attributes not already in the parent class are included. The structure for these declarations is as follows:

```
class <typename> : public <parentTypename> {
```

```
protected:
<OCCItype1> <attributename1>;
    .
    .
    .
<OCCItypen> <attributenamen>;

public:

void *operator new(size_t size);
void *operator new(size_t size, const Session* sess, const string& table);
string getSQLTypeName(size_t size);
<typename> (void *ctx) : <parentTypename>(ctx) { };
static void *readSQL(void *ctx);
virtual void readSQL(AnyData& stream);
static void writeSQL(void *obj, void *ctx);
virtual void writeSQL(AnyData& stream);

}
```

In this structure, all the variables are the same as in the simple object case. `parentTypename` refers to the name of the parent type, that is, the class name of the type from which typename inherits.

A Sample OCCI Application

Following is a sample OCCI application that uses some of the features discussed in this chapter.

First we list the SQL DDL and then the OTT mappings.

```
CREATE TYPE FULL_NAME AS OBJECT (first_name CHAR(20), last_name CHAR(20));
CREATE TYPE ADDRESS AS OBJECT (state CHAR(20), zip CHAR(20));
CREATE TYPE ADDRESS_TAB  AS VARRAY(3) OF REF ADDRESS;
CREATE TYPE PERSON AS OBJECT (id NUMBER, name FULL_NAME,
    curr_addr REF ADDRESS, prev_addr_l ADDRESS_TAB);
CREATE TYPE STUDENT UNDER  PERSON (school_name CHAR(20));
```

Let us assume OTT generates `FULL_NAME`, `ADDRESS`, `PERSON`, and `STUDENT` class declarations in `demo.h`. The following sample OCCI application will extend the classes generated by OTT and will add some user defined methods.

```
/************************** myDemo.h *****************************************/

#include demo.h

// declarations for the MyFullName class.
class MyFullname : public FULLNAME {
    public:
        MyFullname(string first_name, string last_name);
        void displayInfo();
}

// declarations for the MyAddress class.
class MyAddress : public ADDRESS {
    public:
        MyAddress(string state_i, string zip_i);
        void displayInfo();
}

// declarations for the MyPerson class.
class MyPerson : public PERSON {
    public:
        MyPerson(Number id_i, MyFullname *name_i,
            Ref<MyAddress>& addr_i);
        void move(const Ref<MyAddress>& new_addr);
        void displayInfo();
}

/**************************myDemo.cpp*************************************/

/* initialize MyFullName */
MyFullName::MyFullname(string first_name, string last_name)
    : FirstName(first_name), LastName(last_name)
{ }

/* display all the information in MyFullName */
void MyFullName::displayInfo()
{
    cout << "FIRST NAME is" << FirstName << endl;
    cout << "LAST NAME is" << LastName << endl;
}

/*********************************************************************/
// method implementations for MyAddress class.
/********************************************************************/
```

```
/* initialize MyAddress */
MyAddress::MyAddress(string state_i, string zip_i)
    : state(state_i), zip(zip_i)
{ }

/* display all the information in MyAddress */
void MyAddress::displayInfo()
{
    cout << "STATE is" << state << endl;
    cout << "ZIP is" << zip << endl;
}

/*********************************************************************/
// method implementations for MyPerson class.
/********************************************************************/

/* initialize MyPerson */
MyPerson::MyPerson(Number id_i,
MyFullName* name_i, const Ref<MyAddress>& addr_i)
    : id(id_i), name(name_i), curr_addr(addr_i)
{ }

/* Move Person from curr_addr to new_addr */
void MyPerson::move(const Ref<MyAddress>& new_addr)
{
    prev_addr_l.push_back(curr_addr);     // append curr_addr to the vector
    curr_addr = new_addr;
    this->mark_modified();                // mark the object as dirty
}

/*  Display all the information of MyPerson */
void MyPerson::displayInfo()
{
    cout << "ID is" << CPerson::id << endl;
    name->displayInfo();

    // de-referencing the Ref attribute using -> operator
    curr_addr->displayInfo();
    cout << "Prev Addr List: " << endl;
    for (int i = 0; i < prev_addr_l.size(); i++)
    {
        // access the collection elements using [] operator
        prev_addr_l[i]->displayInfo();
    }
}
```

```
/************************************************************************/
// main function of this OCCI application.
// This application connects to the database as scott/tiger, creates
// the Person (Joe Black) whose Address is in CA, and commits the changes.
// The Person object is then retrieved from the database and its
// information is displayed. A second Address object is created (in PA),
// then the previously retrieved Person object (Joe Black) is moved to
// this new address. The Person object is then displayed again.
/***********************************************************************/
int main()
{
    Environment *env = Environment::createEnvironment()
    Connection *conn = env->createConnection("scott", "tiger");

    /* Call the OTT generated function to register the mappings */
    RegisterMappings(env);

    /* create a persistent object of type ADDRESS in the database table,
        ADDR_TAB */
    MyAddress *addr1 = new(conn, "ADDR_TAB")
    MyAddress("CA", "94065");
    MyFullName name1("Joe", "Black");

    /* create a persistent object of type Person in the database table,
        PERSON_TAB */
    MyPerson *person1 = new(conn, "PERSON_TAB")
    MyPerson(1,&name1,addr1->getRef());

    /* commit the transaction which results in the newly created objects, addr,
        person1 being flushed to the server */
    conn->commit();

    Statement *stmt = conn->createStatement();

    ResultSet *resultSet
        = stmt->executeQuery("SELECT REF(Person) from person_tab where id = 1");

    ResultSetMetaData rsMetaData = resultSet->getMetaData();

    if (Types::POBJECT != rsMetaData.getColumnType(1))
        return -1;

    Ref<MyPerson> joe_ref = (Ref<MyPerson>) resultSet.getRef(1);
```

```
    joe_ref->displayInfo();

    /* create a persistent object of type ADDRESS, in the database table,
        ADDR_TAB */
    MyAddress *new_addr1 = new(conn, "ADDR_TAB") MyAddress("PA", "92140");
    joe_ref->move(new_addr1->getRef());
    joe_ref->displayInfo();

    /* commit the transaction which results in the newly created object,
        new_addr and the dirty object, joe to be flushed to the server. Note
        that joe was marked dirty in move(). */
    conn->commit();

    /* The following delete statements delete the objects only from the
        application cache. To delete the objects from server, mark_deleted()
        should be used. */
    delete addr1;
    delete person1;
    delete new_addr1;

    conn->closeStatement(stmt);
    env->terminateConnection(conn);
    Environment::terminateEnvironment(env);
    return 0;
}
```

4

Datatypes

This chapter is a reference for Oracle datatypes used by Oracle C++ Call Interface applications. This information will help you understand the conversions between internal and external representations of data that occur when you transfer data between your application and the database server.

This chapter includes the following topics:

- Overview of Oracle Datatypes
- Internal Datatypes
- External Datatypes
- Data Conversions

Overview of Oracle Datatypes

Accurate communication between your C++ program and the Oracle database server is critical. OCCI applications can retrieve data from database tables by using SQL queries or they can modify existing data through the use of SQL `INSERT`, `UPDATE`, and `DELETE` functions. To facilitate communication between the host language C++ and the database server, you must be aware of how C++ datatypes are converted to Oracle datatypes and back again.

In the Oracle database, values are stored in columns in tables. Internally, Oracle represents data in particular formats called internal datatypes. `NUMBER`, `VARCHAR2`, and `DATE` are examples of Oracle internal datatypes.

OCCI applications work with host language datatypes, or external datatypes, predefined by the host language. When data is transferred between an OCCI application and the database server, the data from the database is converted from internal datatypes to external datatypes.

OCCI Type and Data Conversion

OCCI defines an enumerator called `Type` that lists the possible data representation formats available in an OCCI application. These representation formats are called external datatypes. When data is sent to the database server from the OCCI application, the external datatype indicates to the database server what format to expect the data. When data is requested from the database server by the OCCI application, the external datatype indicates the format of the data to be returned.

For example, on retrieving a value from a `NUMBER` column, the program may be set to retrieve it in `OCCIINT` format (a signed integer format into an integer variable). Or, the client might be set to send data in `OCCIFLOAT` format (floating-point format) stored in a C++ float variable to be inserted in a column of `NUMBER` type.

An OCCI application binds input parameters to a `Statement`, by calling a `set`*`xxx`* method (the external datatype is implicitly specified by the method name), or by calling the `registerOutParam`, `setDataBuffer`, or `setDataBufferArray` method (the external datatype is explicitly specified in the method call). Similarly, when data values are fetched through a `ResultSet` object, the external representation of the retrieved data must be specified. This is done by calling a `get`*`xxx`* method (the external datatype is implicitly specified by the method name) or by calling the `setDataBuffer` method (the external datatype is explicitly specified in the method call).

Note: There are more external datatypes than internal datatypes. In some cases, a single external datatype maps to a single internal datatype; in other cases, many external datatypes map to a single internal datatype. The many-to-one mapping provides you with added flexibility.

See Also:

- External Datatypes on page 4-5

Internal Datatypes

The internal (built-in) datatypes provided by Oracle are listed in this section.

Table 4–1 lists the Oracle internal datatypes and maximum internal length of each:

Table 4–1 Oracle Internal Datatypes

Internal Datatype	Code	Maximum Internal Length
BFILE	114	4 gigabytes
CHAR, NCHAR	96	2000 bytes
DATE	12	7 bytes
INTERVAL DAY TO SECOND REF	183	11 bytes
INTERVAL YEAR TO MONTH REF	182	5 bytes
LONG	8	2 gigabytes (2^31-1 bytes)
LONG RAW	24	2 gigabytes (2^31-1 bytes)
NUMBER	2	21 bytes
RAW	23	2000 bytes
REF	111	
REF BLOB	113	4 gigabytes
REF CLOB, REF NCLOB	112	4 gigabytes
ROWID	11	10 bytes
TIMESTAMP	180	11 bytes
TIMESTAMP WITH LOCAL TIME ZONE	231	7 bytes

Table 4–1 Oracle Internal Datatypes (Continued)

Internal Datatype	Code	Maximum Internal Length
TIMESTAMP WITH TIME ZONE	181	13 bytes
UROWID	208	4000 bytes
User-defined type (object type, VARRAY, nested table)	108	
VARCHAR2, NVARCHAR2	1	4000 bytes

See Also:

- *Oracle9i SQL Reference*
- *Oracle9i Database Concepts*

Character Strings and Byte Arrays

You can use five Oracle internal datatypes to specify columns that contain either characters or arrays of bytes: CHAR, VARCHAR2, RAW, LONG, and LONG RAW.

CHAR, VARCHAR2, and LONG columns normally hold character data. RAW and LONG RAW hold bytes that are not interpreted as characters, for example, pixel values in a bitmapped graphics image. Character data can be transformed when passed through a gateway between networks. For example, character data passed between machines by using different languages (where single characters may be represented by differing numbers of bytes) can be significantly changed in length. Raw data is never converted in this way.

The database designer is responsible for choosing the appropriate Oracle internal datatype for each column in a table. You must be aware of the many possible ways that character and byte-array data can be represented and converted between variables in the OCCI program and Oracle database tables.

Universal Rowid (UROWID)

The universal rowid (UROWID) is a datatype that can store both the logical and the physical rowid of rows in Oracle tables and in foreign tables, such as DB2 tables accessed through a gateway. Logical rowid values are primary key-based logical identifiers for the rows of index organized tables.

To use columns of the UROWID datatype, the value of the COMPATIBLE initialization parameter must be set to 8.1 or higher.

The following OCCI_SQLT types can be bound to universal rowids:

- OCCI_SQLT_CHR (VARCHAR2)
- OCCI_SQLT_VCS (VARCHAR)
- OCCI_SQLT_STR (null terminated string)
- OCCI_SQLT_LVC (long VARCHAR)
- OCCI_SQLT_AFC (CHAR)
- OCCI_SQLT_AVC (CHARZ)
- OCCI_SQLT_VST (string)
- OCCI_SQLT_RDD (ROWID descriptor)

External Datatypes

Communication between the host OCCI application and the Oracle database server is through the use of external datatypes. Specifically, external datatypes are mapped to C++ datatypes.

Table 4–2 lists the Oracle external datatypes, the C++ equivalent (what the Oracle internal datatype is usually converted to), and the corresponding OCCI type:

Table 4–2 External Datatypes, C++ Datatypes, and OCCI Types

External Datatype	Code	C++ Datatype	OCCI Type
Binary FILE	114	OCILobLocator	OCCI_SQLT_FILE
Binary LOB	113	OCILobLocator	OCCI_SQLT_BLOB
CHAR	96	char[n]	OCCI_SQLT_AFC
Character LOB	112	OCILobLocator	OCCI_SQLT_CLOB
CHARZ	97	char[n+1]	OCCI_SQLT_RDD
DATE	12	char[7]	OCCI_SQLT_DAT
FLOAT	4	float, double	OCCIFLOAT
16 bit signed INTEGER	3	signed short, signed int	OCCIINT
32 bit signed INTEGER	3	signed int, signed long	OCCIINT
8 bit signed INTEGER	3	signed char	OCCIINT
INTERVAL DAY TO SECOND	190	char[11]	OCCI_SQLT_INTERVAL_DS

n Indicates variable length, depending on program requirements (or the operating system in the case of ROWID).

Table 4–2 External Datatypes, C++ Datatypes, and OCCI Types (Continued)

External Datatype	Code	C++ Datatype	OCCI Type
INTERVAL YEAR TO MONTH	189	char[5]	OCCI_SQLT_INTERVAL_YM
LONG	8	char[n]	OCCI_SQLT_LNG
LONG RAW	24	unsigned char[n]	OCCI_SQLT_LBI
LONG VARCHAR	94	char[n+sizeof(integer)]	OCCI_SQLT_LVC
LONG VARRAW	95	unsigned char[n+sizeof(integer)]	OCCI_SQLT_LVB
NAMED DATA TYPE	108	struct	OCCI_SQLT_NTY
NUMBER	2	unsigned char[21]	OCCI_SQLT_NUM
RAW	23	unsigned char[n]	OCCI_SQLT_BIN
REF	110	OCIRef	OCCI_SQLT_REF
ROWID	11	OCIRowid	OCCI_SQLT_RID
ROWID descriptor	104	OCIRowid	OCCI_SQLT_RDD
null-terminated STRING	5	char[n+1]	OCCI_SQLT_STR
TIMESTAMP	187	char[11]	OCCI_SQLT_TIMESTAMP
TIMESTAMP WITH LOCAL TIME ZONE	232	char[7]	OCCI_SQLT_TIMESTAMP_LTZ
TIMESTAMP WITH TIME ZONE	188	char[13]	OCCI_SQLT_TIMESTAMP_TZ
UNSIGNED INT	68	unsigned	OCCIUNSIGNED_INT
VARCHAR	9	char[n+sizeof(short integer)]	OCCI_SQLT_VCS
VARCHAR2	1	char[n]	OCCI_SQLT_CHR
VARNUM	6	char[22]	OCCI_SQLT_VNU
VARRAW	15	unsigned char[n+sizeof(short integer)]	OCCI_SQLT_VBI
Most of the following external datatypes are represented as C++ classes in OCCI. Please refer to Chapter 8, "OCCI Classes and Methods" for additional information.			
OCCI BFILE		Bfile	OCCIBFILE
OCCI BLOB		Blob	OCCIBLOB
OCCI BOOL		bool	OCCIBOOL

n Indicates variable length, depending on program requirements (or the operating system in the case of ROWID).

Table 4–2 External Datatypes, C++ Datatypes, and OCCI Types (Continued)

External Datatype	Code	C++ Datatype	OCCI Type
OCCI BYTES		Bytes	OCCIBYTES
OCCI ROWID		Bytes	OCCIROWID
OCCI CHAR		char	OCCICHAR
OCCI CLOB		Clob	OCCICLOB
OCCI DATE		Date	OCCIDATE
OCCI DOUBLE		double	OCCIDOUBLE
OCCI FLOAT		float	OCCIFLOAT
OCCI INTERVALDS		IntervalDS	OCCIINTERVALDS
OCCI INTERVALYM		IntervalYM	OCCIINTERVALYM
OCCI INT		int	OCCIINT
OCCI METEDATA		MetaData	OCCIMETADATA
OCCI NUMBER		Number	OCCINUMBER
OCCI REF		Ref	OCCIREF
OCCI REFANY		RefAny	OCCIREFANY
OCCI CURSOR		ResultSet	OCCICURSOR
OCCI STRING		STL string	OCCISTRING
OCCI VECTOR		STL vector	OCCIVECTOR
OCCI STREAM		Stream	OCCISTREAM
OCCI TIMESTAMP		Timestamp	OCCITIMESTAMP
OCCI UNSIGNED INT		unsigned int	OCCIUNSIGNED_INT
OCCI POBJECT		user defined types (generated by the Object Type Translator)	OCCIPOBJECT

n Indicates variable length, depending on program requirements (or the operating system in the case of ROWID).

Note: The TIMESTAMP and TIMESTAMP WITH TIME ZONE datatypes are collectively known as datetimes. The INTERVAL YEAR TO MONTH and INTERVAL DAY TO SECOND are collectively known as intervals.

Please note the usage of the types in the following methods of the `Statement` class:

- `registerOutParam`: Only types of the form `OCCI`*xxx* (for example, `OCCIDOUBLE`, `OCCICURSOR`, and so on) on the `occiCommon.h` file are permitted. However, there are some exceptions. `OCCIANYDATA`, `OCCIMETADATA`, `OCCISTREAM`, and `OCCIBOOL` are not permitted.
- `setDataBuffer()` and `setDataBufferArray`: Only types of the form `OCCI_SQLT_`*xxx* (for example, `OCCI_SQLT_INT`) in the `occiCommon.h` file are permitted.

Please note the usage of the types in the following methods of the `ResultSet` class:

- `setDataBuffer()` and `setDataBufferArray`: Only types of the form `OCCI_SQLT_`*xxx* (for example, `OCCI_SQLT_INT`) in the `occiCommon.h` file are permitted.

Description of External Datatypes

This section provides a description for each of the external datatypes.

BFILE

The external datatype `BFILE` allows read-only byte stream access to large files on the file system of the database server. A BFILE is a large binary data object stored in operating system files outside database tablespaces. These files use reference semantics. The Oracle server can access a BFILE provided the underlying server operating system supports stream-mode access to these operating system files.

BLOB

The external datatype `BLOB` stores unstructured binary large objects. A BLOB can be thought of as a bitstream with no character set semantics. BLOBs can store up to 4 gigabytes of binary data.

BLOB datatypes have full transactional support. Changes made through OCCI participate fully in the transaction. BLOB value manipulations can be committed or rolled back. You cannot save a BLOB locator in a variable in one transaction and then use it in another transaction or session.

CHAR

The external datatype `CHAR` is a string of characters, with a maximum length of 2000 characters. Character strings are compared by using blank-padded comparison semantics.

CHARZ

The external datatype `CHARZ` is similar to the `CHAR` datatype, except that the string must be null terminated on input, and Oracle places a null terminator character at the end of the string on output. The null terminator serves only to delimit the string on input or output. It is not part of the data in the table.

CLOB

The external datatype `CLOB` stores fixed-width or varying-width character data. A CLOB can store up to 4 gigabytes of character data. CLOBs have full transactional support. Changes made through OCCI participate fully in the transaction. CLOB value manipulations can be committed or rolled back. You cannot save a CLOB locator in a variable in one transaction and then use it in another transaction or session.

DATE

The external datatype `DATE` can update, insert, or retrieve a date value using the Oracle internal date binary format, which contains seven bytes, as listed in Table 4–3:

Table 4–3 Format of the DATE Datatype

	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Meaning:	Century	Year	Month	Day	Hour	Minute	Second
Example (01-JUN-2000, 3:17PM):	120	100	6	1	16	18	1
Example (01-JAN-4712 BCE):	53	88	1	1	1	1	1

The century and year bytes (1 and 2) are in *excess-100 notation*. Dates BCE (Before Common Era) are less than 100. Dates in the Common Era, 0 and after, are greater than 100. For dates 0 and after, the first digit of both bytes 1 and 2 merely signifies that it is of the Common Era.

For byte 1, the second and third digits of the century are calculated as the year (an integer) divided by 100. With integer division, the fractional portion is discarded. The following calculation is for the year 1992:

```
1992 / 100 = 19
```

For byte 1, `119` represents the twentieth century, 1900 to 1999. A value of `120` would represent the twenty-first century, 2000 to 2099.

For byte 2, the second and third digits of the year are calculated as the year modulo 100. With a modulo division, the nonfractional portion is discarded:

```
1992 % 100 = 92
```

For byte 2, `192` represents the ninety-second year of the current century. A value of `100` would represent the zeroth year of the current century.

The year 2000 would yield 120 for byte 1 and 100 for byte 2.

For years prior to 0 CE, centuries and years are represented by the difference between 100 and the number. So 01-JAN-4712 BCE is century 53 because 100 - 47 = 53. The year is 88 because the 100 - 12 = 88.

Valid dates begin at 01-JAN-4712 BCE. The month byte ranges from 1 to 31, the hour byte ranges from 1 to 24, and the second byte ranges from 1 to 60.

Note: If no time is specified for a date, the time defaults to midnight: 1, 1, 1.

When you enter a date in binary format by using the external datatype `DATE`, the database does not perform consistency or range checking. All data in this format must be validated before input.

Note: There is little need for the external datatype `DATE`. It is more convenient to convert `DATE` values to a character format, because most programs deal with dates in a character format, such as DD-MON-YYYY. Instead, you may use the `Date` datatype.

When a `DATE` column is converted to a character string in your program, it is returned in the default format mask for your session, or as specified in the `INIT.ORA` file.

Note that this datatype is different from `OCCI DATE` which corresponds to a C++ `Date` datatype.

FLOAT

The external datatype FLOAT processes numbers with fractional parts. The number is represented in the host system's floating-point format. Normally, the length is 4 or 8 bytes.

The internal format of an Oracle number is decimal. Most floating-point implementations are binary. Oracle, therefore, represents numbers with greater precision than floating-point representations.

INTEGER

The external datatype INTEGER is used for converting numbers. An external integer is a signed binary number. Its size is operating system-dependent. If the number being returned from Oracle is not an integer, then the fractional part is discarded, and no error is returned. If the number returned exceeds the capacity of a signed integer for the system, then Oracle returns an overflow on conversion error.

> **Note:** A rounding error may occur when converting between FLOAT and NUMBER. Using a FLOAT as a bind variable in a query may return an error. You can work around this by converting the FLOAT to a string and using the OCCI type OCCI_SQLT_CHR or the OCCI type OCCI_SQLT_STR for the operation.

INTERVAL DAY TO SECOND

The external datatype INTERVAL DAY TO SECOND stores the difference between two datetime values in terms of days, hours, minutes, and seconds. Specify this datatype as follows:

```
INTERVAL DAY [(day_precision)]
   TO SECOND [(fractional_seconds_precision)]
```

This example uses the following placeholders:

- *day_precision*: Number of digits in the DAY datetime field. Accepted values are 1 to 9. The default is 2.
- *fractional_seconds_precision*: Number of digits in the fractional part of the SECOND datetime field. Accepted values are 0 to 9. The default is 6.

To specify an INTERVAL DAY TO SECOND literal with nondefault day and second precisions, you must specify the precisions in the literal. For example, you might

specify an interval of 100 days, 10 hours, 20 minutes, 42 seconds, and 22 hundredths of a second as follows:

```
INTERVAL '100 10:20:42.22' DAY(3) TO SECOND(2)
```

You can also use abbreviated forms of the `INTERVAL DAY TO SECOND` literal. For example:

`INTERVAL '90' MINUTE`	`maps to INTERVAL '00 00:90:00.00' DAY TO SECOND(2)`
`INTERVAL '30:30' HOUR TO MINUTE`	`maps to INTERVAL '00 30:30:00.00' DAY TO SECOND(2)`
`INTERVAL '30' SECOND(2,2)`	`maps to INTERVAL '00 00:00:30.00' DAY TO SECOND(2)`

INTERVAL YEAR TO MONTH

The external datatype `INTERVAL YEAR TO MONTH` stores the difference between two datetime values by using the `YEAR` and `MONTH` datetime fields. Specify `INTERVAL YEAR TO MONTH` as follows:

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

The placeholder *`year_precision`* is the number of digits in the `YEAR` datetime field. The default value of *`year_precision`* is 2. To specify an `INTERVAL YEAR TO MONTH` literal with a nondefault *`year_precision`*, you must specify the precision in the literal. For example, the following `INTERVAL YEAR TO MONTH` literal indicates an interval of 123 years, 2 months:

```
INTERVAL '123-2' YEAR(3) TO MONTH
```

You can also use abbreviated forms of the `INTERVAL YEAR TO MONTH` literal. For ex

`INTERVAL '10' MONTH`	`maps to INTERVAL '0-10' YEAR TO MONTH`
`INTERVAL '123' YEAR(3)`	`maps to INTERVAL '123-0' YEAR(3) TO MONTH`

LONG

The external datatype `LONG` stores character strings longer than 4000 bytes and up to 2 gigabytes in a column of datatype `LONG`. Columns of this type are only used for storage and retrieval of long strings. They cannot be used in methods, expressions,

or WHERE clauses. LONG column values are generally converted to and from character strings.

LONG RAW

The external datatype LONG RAW is similar to the external datatype RAW, except that it stores up to 2 gigabytes.

LONG VARCHAR

The external datatype LONG VARCHAR stores data from and into an Oracle LONG column. The first four bytes contain the length of the item. The maximum length of a LONG VARCHAR is 2 gigabytes.

LONG VARRAW

The external datatype LONG VARRAW store data from and into an Oracle LONG RAW column. The length is contained in the first four bytes. The maximum length is 2 gigabytes.

NCLOB

The external datatype NCLOB is a national character version of a CLOB. It stores fixed-width, multibyte national character set character (NCHAR), or varying-width character set data. An NCLOB can store up to 4 gigabytes of character text data.

NCLOBs have full transactional support. Changes made through OCCI participate fully in the transaction. NCLOB value manipulations can be committed or rolled back. You cannot save an NCLOB locator in a variable in one transaction and then use it in another transaction or session.

You cannot create an object with NCLOB attributes, but you can specify NCLOB parameters in methods.

NUMBER

You should not need to use NUMBER as an external datatype. If you do use it, Oracle returns numeric values in its internal 21-byte binary format and will expect this format on input. The following discussion is included for completeness only.

Oracle stores values of the NUMBER datatype in a variable-length format. The first byte is the exponent and is followed by 1 to 20 mantissa bytes. The high-order bit of the exponent byte is the sign bit; it is set for positive numbers and it is cleared for negative numbers. The lower 7 bits represent the exponent, which is a base-100 digit with an offset of 65.

To calculate the decimal exponent, add 65 to the base-100 exponent and add another 128 if the number is positive. If the number is negative, you do the same, but subsequently the bits are inverted. For example, -5 has a base-100 exponent = 62 (0x3e). The decimal exponent is thus (~0x3e) -128 - 65 = 0xc1 -128 -65 = 193 -128 -65 = 0.

Each mantissa byte is a base-100 digit, in the range 1 to 100. For positive numbers, the digit has 1 added to it. So, the mantissa digit for the value 5 is 6. For negative numbers, instead of adding 1, the digit is subtracted from 101. So, the mantissa digit for the number -5 is 96 (101 - 5). Negative numbers have a byte containing 102 appended to the data bytes. However, negative numbers that have 20 mantissa bytes do not have the trailing 102 byte. Because the mantissa digits are stored in base-100, each byte can represent two decimal digits. The mantissa is normalized; leading zeroes are not stored.

Up to 20 data bytes can represent the mantissa. However, only 19 are guaranteed to be accurate. The 19 data bytes, each representing a base-100 digit, yield a maximum precision of 38 digits for an internal datatype `NUMBER`.

Note that this datatype is different from `OCCI NUMBER` which corresponds to a C++ `Number` datatype.

OCCI BFILE

See Also:

- Chapter 8, "OCCI Classes and Methods", Bfile Class on page 8-5

OCCI BLOB

See Also:

- Chapter 8, "OCCI Classes and Methods", Blob Class on page 8-13

OCCI BYTES

See Also:

- Chapter 8, "OCCI Classes and Methods", Bytes Class on page 8-24

OCCI CLOB

See Also:

- Chapter 8, "OCCI Classes and Methods", Clob Class on page 8-27

OCCI DATE

See Also:

- Chapter 8, "OCCI Classes and Methods", Date Class on page 8-51

OCCI INTERVALDS

See Also:

- Chapter 8, "OCCI Classes and Methods", IntervalDS Class on page 8-70

OCCI INTERVALYM

See Also:

- Chapter 8, "OCCI Classes and Methods", IntervalYM Class on page 8-83

OCCI NUMBER

See Also:

- Chapter 8, "OCCI Classes and Methods", Number Class on page 8-104

OCCI POBJECT

See Also:

- Chapter 8, "OCCI Classes and Methods", PObject Class on page 8-130

OCCI REF

See Also:

- Chapter 8, "OCCI Classes and Methods", Ref Class on page 8-137

OCCI REFANY

See Also:

- Chapter 8, "OCCI Classes and Methods", RefAny Class on page 8-144

OCCI STRING

The external datatype OCCI STRING corresponds to an STL string.

OCCI TIMESTAMP

See Also:

- Chapter 8, "OCCI Classes and Methods", Timestamp Class on page 8-219

OCCI VECTOR

The external datatype `OCCI VECTOR` is used to represent collections, for example, a nested table or `VARRAY`. `CREATE TYPE num_type as VARRAY OF NUMBER(10)` can be represented in a C++ application as `vector<int>`, `vector<Number>`, and so on.

RAW

The external datatype `RAW` is used for binary data or byte strings that are not to be interpreted or processed by Oracle. `RAW` could be used, for example, for graphics character sequences. The maximum length of a `RAW` column is 2000 bytes.

When `RAW` data in an Oracle table is converted to a character string, the data is represented in hexadecimal code. Each byte of `RAW` data is represented as two characters that indicate the value of the byte, ranging from 00 to FF. If you input a character string by using `RAW`, then you must use hexadecimal coding.

REF

The external datatype `REF` is a reference to a named datatype. To allocate a `REF` for use in an application, declare a variable as a pointer to a `REF`.

ROWID

The external datatype `ROWID` identifies a particular row in a database table. The `ROWID` is often returned from a query by issuing a statement similar to the following example:

```
SELECT ROWID, var1, var2 FROM db
```

You can then use the returned `ROWID` in further `DELETE` statements.

If you are performing a `SELECT` for an `UPDATE` operation, then the `ROWID` is implicitly returned.

STRING

The external datatype `STRING` behaves like the external datatype `VARCHAR2` (datatype code 1), except that the external datatype `STRING` must be null-terminated.

Note that this datatype is different from `OCCI STRING` which corresponds to a C++ STL string datatype.

TIMESTAMP

The external datatype `TIMESTAMP` is an extension of the `DATE` datatype. It stores the year, month, and day of the `DATE` datatype, plus hour, minute, and second values. Specify the `TIMESTAMP` datatype as follows:

```
TIMESTAMP [(fractional_seconds_precision)]
```

The placeholder *`fractional_seconds_precision`* optionally specifies the number of digits in the fractional part of the `SECOND` datetime field and can be a number in the range 0 to 9. The default is 6. For example, you specify `TIMESTAMP(2)` as a literal as follows:

```
TIMESTAMP '1997-01-31 09:26:50.10'
```

Note that this datatype is different from `OCCI TIMESTAMP`.

TIMESTAMP WITH LOCAL TIME ZONE

The external datatype TIMESTAMP WITH TIME ZONE (TSTZ) is a variant of TIMESTAMP that includes an explicit time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). Specify the TIMESTAMP WITH TIME ZONE datatype as follows:

```
TIMESTAMP(fractional_seconds_precision) WITH TIME ZONE
```

The placeholder *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6.

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of the TIME ZONE offsets stored in the data.

TIMESTAMP WITH TIME ZONE

The external datatype TIMESTAMP WITH TIME ZONE is a variant of TIMESTAMP that includes a time zone displacement in its value. The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). Specify the TIMESTAMP WITH TIME ZONE datatype as follows:

```
TIMESTAMP [(fractional_seconds_precision)] WITH TIME ZONE
```

The placeholder *fractional_seconds_precision* optionally specifies the number of digits in the fractional part of the SECOND datetime field and can be a number in the range 0 to 9. The default is 6. For example, you might specify TIMESTAMP(0) WITH TIME ZONE as a literal as follows:

```
TIMESTAMP '1997-01-31 09:26:50+02.00'
```

UNSIGNED INT

The external datatype UNSIGNED INT is used for unsigned binary integers. The size in bytes is operating system dependent. The host system architecture determines the order of the bytes in a word. If the number being output from Oracle is not an integer, the fractional part is discarded, and no error is returned. If the number to be returned exceeds the capacity of an unsigned integer for the operating system, Oracle returns an overflow on conversion error.

VARCHAR

The external datatype VARCHAR store character strings of varying length. The first two bytes contain the length of the character string, and the remaining bytes contain the actual string. The specified length of the string in a bind or a define call must include the two length bytes, meaning the largest VARCHAR string is 65533 bytes long, not 65535. For converting longer strings, use the LONG VARCHAR external datatype.

VARCHAR2

The external datatype VARCHAR2 is a variable-length string of characters up to 4000 bytes.

VARNUM

The external datatype VARNUM is similar to the external datatype NUMBER, except that the first byte contains the length of the number representation. This length value does not include the length byte itself. Reserve 22 bytes to receive the longest possible VARNUM. You must set the length byte when you send a VARNUM value to the database.

Table 4–4 VARNUM Examples

Decimal Value	Length Byte	Exponent Byte	Mantissa Bytes	Terminator Byte
0	1	128	N/A	N/A
5	2	193	6	N/A
-5	3	62	96	102
2767	3	194	28, 68	N/A
-2767	4	61	74, 34	102
100000	2	195	11	N/A
1234567	5	196	2, 24, 46, 68	N/A

VARRAW

The external datatype VARRAW is similar to the external datatype RAW, except that the first two bytes contain the length of the data. The specified length of the string in a bind or a define call must include the two length bytes. So the largest VARRAW string that can be received or sent is 65533 bytes, not 65535. For converting longer strings, use the LONG VARRAW datatype.

Data Conversions

Table 4–5 lists the supported conversions from Oracle internal datatypes to external datatypes, and from external datatypes to internal column representations. Note the following conditions:

- A `REF` stored in the database is converted to `OCCI_SQLT_REF` on output
- `OCCI_SQLT_REF` is converted to the internal representation of a `REF` on input
- A named datatype stored in the database is converted to `OCCI_SQLT_NTY` (and represented by a C structure in the application) on output
- `OCCI_SQLT_NTY` (represented by a C structure in an application) is converted to the internal representation of the corresponding datatype on input
- A `LOB` and a `BFILE` are represented by descriptors in OCCI applications, so there are no input or output conversions

Table 4–5 Data Conversions

	Internal Datatypes							
	1	2	8	11	12	23	24	96
External Datatypes	**VARCHAR2**	**NUMBER**	**LONG**	**ROWID**	**DATE**	**RAW**	**LONG RAW**	**CHAR**
1 `VARCHAR`	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3]	
2 `NUMBER`	I/O[4]	I/O	I					I/O[4]
3 `INTEGER`	I/O[4]	I/O	I					I/O[4]
4 `FLOAT`	I/O[4]	I/O	I					I/O[4]
5 `STRING`	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3, 5]	I/O
6 `VARNUM`	I/O[4]	I/O	I					I/O[4]
7 `DECIMAL`	I/O[4]	I/O	I					I/O[4]

Conversion valid for...

I = Input only.

O = Output only.

I/O = Input or Output.

Notes:

1. For input, host string must be in Oracle ROWID format. On output, column value is returned in Oracle ROWID format.
2. For input, host string must be in the Oracle DATE character format. On output, column value is returned in Oracle DATE format.
3. For input, host string must be in hexadecimal format. On output, column value is returned in hexadecimal format.
4. For output, column value must represent a valid number.
5. Length must be less than or equal to 2000 characters.
6. On input, column value is stored in hexadecimal format. On output, column value must be in hexadecimal format.

Table 4–5 Data Conversions (Continued)

	Internal Datatypes							
	1	**2**	**8**	**11**	**12**	**23**	**24**	**96**
External Datatypes	**VARCHAR2**	**NUMBER**	**LONG**	**ROWID**	**DATE**	**RAW**	**LONG RAW**	**CHAR**
8 LONG	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3, 5]	I/O
9 VARCHAR	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3, 5]	I/O
10 ROWID	I		I	I/O				I
12 DATE	I/O		I		I/O			I/O
15 VARRAW	I/O[6]		I[5, 6]			I/O	I/O	I/O[6]
23 RAW	I/O[6]		I[5, 6]			I/O	I/O	I/O[6]
24 LONG RAW	O[6]		I[5, 6]			I/O	I/O	O[6]
68 UNSIGNED	I/O[4]	I/O	I					I/O[4]
94 LONG VARCHAR	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3, 5]	I/O
95 LONG VARRAW	I/O[6]		I[5, 6]			I/O	I/O	I/O[6]
96 CHAR	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I[3, 5]	I/O
97 CHARZ	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I[3, 5]	I/O
104 ROWID Desc.	I(1)			I/O				I(1)
OCCI Number	I/O[4]	I/O	I					I/O[4]
OCCI Bytes	I/O[6]		I[5, 6]			I/O	I/O	I/O[6]
OCCI Date	I/O		I		I/O			I/O
OCCI Timestamp								
STL string	I/O	I/O	I/O	I/O[1]	I/O[2]	I/O[3]	I/O[3]	

Conversion valid for...

I = Input only.

O = Output only.

I/O = Input or Output.

Notes:

1. For input, host string must be in Oracle ROWID format. On output, column value is returned in Oracle ROWID format.
2. For input, host string must be in the Oracle DATE character format. On output, column value is returned in Oracle DATE format.
3. For input, host string must be in hexadecimal format. On output, column value is returned in hexadecimal format.
4. For output, column value must represent a valid number.
5. Length must be less than or equal to 2000 characters.
6. On input, column value is stored in hexadecimal format. On output, column value must be in hexadecimal format.

Data Conversions for LOB Datatypes

Table 4–6 Data Conversions for LOBs

	INTERNAL DATATYPES	
EXTERNAL DATATYPES	**CLOB**	**BLOB**
VARCHAR	I/O	
CHAR	I/O	
LONG	I/O	
LONG VARCHAR	I/O	
STL STRING	I/O	
RAW		I/O
VARRAW		I/O
LONG RAW		I/O
LONG VARRAW		I/O
OCCI BYTES		I/O

Data Conversions for Date, Timestamp, and Interval Datatypes

You can also use one of the character data types for the host variable used in a fetch or insert operation from or to a datetime or interval column. Oracle will do the conversion between the character data type and datetime/interval data type for you.

Data Conversions for Date, Timestamp, and Interval Datatypes

	Internal Types						
External Types	**VARCHAR,CHAR**	**DATE**	**TS**	**TSTZ**	**TSLTZ**	**INTERVAL YEAR TO MONTH**	**NTERVAL DAY TO SECOND**
VARCHAR2, CHAR	I/O	I/O	I/O	I/O	I/O	I/O	I/O
STL STRING	I/O	I/O	I/O	I/O	I/O	I/O	I/O
DATE	I/O	I/O	I/O	I/O	I/O	-	-
OCCI DATE	I/O	I/O	I/O	I/O	I/O	-	-
ANSI DATE	I/O	I/O	I/O	I/O	I/O	-	-

Data Conversions for Date, Timestamp, and Interval Datatypes (Continued)

	Internal Types						
External Types	**VARCHAR,CHAR**	**DATE**	**TS**	**TSTZ**	**TSLTZ**	**INTERVAL YEAR TO MONTH**	**NTERVAL DAY TO SECOND**
TIMESTAMP (TS)	I/O	I/O	I/O	I/O	I/O	-	-
OCCI TIMESTAMP	I/O	I/O	I/O	I/O	I/O	-	-
TIMESTAMP WITH TIME ZONE (TSTZ)	I/O	I/O	I/O	I/O	I/O	-	-
TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)	I/O	I/O	I/O	I/O	I/O		
INTERVAL YEAR TO MONTH	I/O	-	-	-	-	I/O	-
OCCI INTERVALYM	I/O	-	-	-	-	I/O	-
INTERVAL DAY TO SECOND	I/O	-	-	-	-	-	I/O
OCCI INTERVALDS	I/O	-	-	-	-	-	I/O

Note: When assigning a source with time zone to a target without a time zone, the time zone portion of the source is ignored. On assigning a source without a time zone to a target with a time zone, the time zone of the target is set to the session's default time zone

(0) When assigning an Oracle DATE to a TIMESTAMP, the TIME portion of the DATE is copied over to the TIMESTAMP. When assigning a TIMESTAMP to Oracle DATE, the TIME portion of the result DATE is set to zero. This is done to encourage migration of Oracle DATE to ANSI compliant DATETIME data types

(1) When assigning an ANSI DATE to an Oracle DATE or a TIMESTAMP, the TIME portion of the Oracle DATE and the TIMESTAMP are set to zero. When assigning an Oracle DATE or a TIMESTAMP to an ANSI DATE, the TIME portion is ignored

(2) When assigning a DATETIME to a character string, the DATETIME is converted using the session's default DATETIME format. When assigning a character string to a DATETIME, the string must contain a valid DATETIME value based on the session's default DATETIME format

(3) When assigning a character string to an INTERVAL, the character string must be a valid INTERVAL character format.

1. When converting from TSLTZ to CHAR, DATE, TIMESTAMP and TSTZ, the value will be adjusted to the session time zone.

2. When converting from `CHAR`, `DATE`, and `TIMESTAMP` to `TSLTZ`, the session time zone will be stored in memory.
3. When assigning `TSLTZ` to `ANSI DATE`, the time portion will be zero.
4. When converting from `TSTZ`, the time zone which the time stamp is in will be stored in memory.
5. When assigning a character string to an interval, the character string must be a valid interval character format.

5 Introduction to LOBs

The following topics are covered in this chapter:

- Overview of LOBs
- LOB Classes and Methods
- Objects with LOB Attributes

Overview of LOBs

Oracle C++ Call Interface (OCCI) includes classes and methods for performing operations on large objects (LOBs). LOBs are either internal or external depending on their location with respect to the database.

Internal LOBs (BLOBs, CLOBs, and NCLOBs)

Internal LOBs are stored inside database tablespaces in a way that optimizes space and enables efficient access. Internal LOBs use copy semantics and participate in the transactional model of the server. You can recover internal LOBs in the event of transaction or media failure, and any changes to an internal LOB value can be committed or rolled back. In other words, all the ACID[1] properties that pertain to using database objects also pertain to using internal LOBs.

Internal LOB Datatypes

There are three SQL datatypes for defining instances of internal LOBs:

- BLOB: A LOB whose value is composed of unstructured binary (raw) data
- CLOB: A LOB whose value is composed of character data that corresponds to the database character set defined for the Oracle database
- NCLOB: A LOB whose value is composed of character data that corresponds to the national character set defined for the Oracle database

Copy Semantics

Internal LOBs, whether persistent or temporary, use copy semantics. When you insert or update a LOB with a LOB from another row in the same table, the LOB value is copied so that each row has a copy of the LOB value.

The use of copy semantics results in both the LOB locator and the LOB value being copied, not just the LOB locator.

Internal LOBs are divided into persistent LOBs and temporary LOBs.

---

[1] ACID = Access Control Information Directory. This is the attribute that determines who has what type of access and to what directory data. It contains a set of rules for structural and content access items. For more information, refer to the Oracle Internet Directory Administrator's Guide.

External LOBs (BFILEs)

External LOBs (BFILES) are large binary data objects stored in operating system files outside database tablespaces. These files use reference semantics. Apart from conventional secondary storage devices such as hard disks, BFILEs may also be located on tertiary block storage devices such as CD-ROMs, PhotoCDs and DVDs.

The BFILE datatype allows read-only byte stream access to large files on the file system of the database server.

Oracle can access BFILEs provided that the underlying server operating system supports stream mode access to these operating system files.

Note:

- External LOBs do not participate in transactions. Any support for integrity and durability must be provided by the underlying file system as governed by the operating system.
- A single external LOB must reside on a single device. It may not, for instance, be striped across a disk array.

External LOB Datatypes

There is one SQL datatype for declaring instances of external LOBs, called a BFILE. A BFILE is a LOB whose value is composed of binary (raw) data and is stored outside the database tablespaces in a server-side operating system file.

Reference Semantics

External LOBs (BFILEs) use reference semantics. When a BFILE associated with a column of a row in a table is copied to another column, only the BFILE locator is copied, not the actual operating system file that houses the BFILE.

LOB Values and Locators

The size of the LOB value, among other things, dictates where it is stored. The LOB value is either stored inline with the row data or outside the row.

A LOB locator is stored inline with the row data and indicates where the LOB value is stored.

Inline Storage of the LOB Value

Data stored in a LOB is termed the LOB value. The value of an internal LOB may or may not be stored inline with the other row data. If you do not set `DISABLE STORAGE IN ROW`, and if the internal LOB value is less than approximately 4,000 bytes, then the value is stored inline.Otherwise, it is stored outside the row. Since LOBs are intended to be large objects, inline storage will only be relevant if your application mixes small and large LOBs.

The LOB value is automatically moved out of the row once it extends beyond approximately 4,000 bytes.

LOB Locators

Regardless of where the value of the internal LOB is stored, a LOB locator is stored in the row. You can think of a LOB locator as a pointer to the actual location of the LOB value. A LOB locator is a locator to an internal LOB while a BFILE locator is a locator to an external LOB.

- **Internal LOB Locators**: For internal LOBs, the LOB column stores a locator to the LOB value stored in a database tablespace. Each internal LOB column and attribute for a given row has its own unique LOB locator and a distinct copy of the LOB value stored in the database tablespace.
- **External LOB Locators**: For external LOBs (BFILEs), the LOB column stores a locator to the external operating system file that houses the BFILE. Each external LOB column and attribute for a given row has its own BFILE locator. However, two different rows can contain a BFILE locator that points to the same operating system file.

See Also:

- *Oracle9i Application Developer's Guide - Large Objects (LOBs)*, for general information about LOBs and the LOB interfaces available
- *Oracle9i Supplied PL/SQL Packages Reference.*, for information on how to use LOBS with the `dbms_lob` package

LOB Classes and Methods

The classes and methods in Table 5–1are available for LOB operations.

Table 5–1 OCCI LOB Classes and Methods

Class	Method	Purpose
Bfile Class on page 8-5	close() closeStream() fileExists() getDirAlias() getFileName() getStream() isInitialized() isNull() isOpen() length() open() operator=() operator==() operator!=() read() setName() setNull()	To access data in external LOBs (BFILEs)

Table 5–1 OCCI LOB Classes and Methods (Continued)

Class	Method	Purpose
Blob Class on page 8-13	append() close() closeStream() copy() getChunkSize() getStream() isInitialized() isNull() isOpen() length() open() operator=() operator==() operator!= () read() setEmpty() setNull() trim() write() writeChunk()	To manipulate internal LOB (BLOB) values and locators

Table 5–1 OCCI LOB Classes and Methods (Continued)

Class	Method	Purpose
Clob Class on page 8-13	append() close() closeStream() copy() getCharSetForm() getCharSetId() getChunkSize() getStream() isInitialized() isNull() isOpen() length() open() operator=() operator==() operator!=() read() setEmpty() setNull() trim() write() writeChunk()	To manipulate internal LOB (CLOB and NCLOB) values and locators

See Also: Chapter 8, "OCCI Classes and Methods" for detailed information about each class and method.

Creating LOBs

To create an internal or external LOB, initialize a new LOB locator in the database. Based on the type of LOB you want to create, use one of the following classes:

- `Bfile`

- `Blob`
- `Clob`

You can then use the related methods, as appropriate, to access the LOB value.

> **Note:** Whenever you want to modify an internal LOB column or attribute (write, copy, trim, and so forth), you must lock the row containing the LOB. One way to do this is to use a `SELECT...FOR UPDATE` statement to select the locator before performing the operation.

For any LOB write command to be successful, a transaction must be open. This means that if you commit a transaction before writing the data, then you must lock the row again (by reissuing the `SELECT ... FOR UPDATE` statement, for example), because the `COMMIT` closes the transaction.

Opening and Closing LOBs

OCCI provides methods to explicitly open and close internal and external LOBs:

- `Bfile::open()` and `Bfile::close()`
- `Blob::open()` and `Blob::close()`
- `Clob::open()` and `Clob::close()`

Additional methods are available to check whether a particular LOB is already open:

- `Bfile::isOpen()`
- `Blob::isOpen()`
- `Clob::isOpen()`

These methods allow an OCCI application to mark the beginning and end of a series of LOB operations so that specific processing (for example, updating indexes, and so on) can be performed when a LOB is closed.

Note: For internal LOBs, the concept of openness is associated with a LOB and not the LOB locator. The LOB locator does not store any information about whether the LOB to which it refers is open. It is possible for more than one LOB locator to point to the same open LOB. However, for external LOBs, openness is associated with a specific external LOB locator. Hence, more than one open can be performed on the same BFILE using different external LOB locators.

Note: If LOB operations are not wrapped inside `open()` and `close()` method calls, any extensible indexes on the LOB are updated as LOB modifications are made, and thus are always valid and may be used at any time. If the LOB is modified between a set of `open()` and `close()` method calls, triggers are not fired for individual LOB modifications. Triggers are only fired after the `close()` method call, so indexes are not updated then, and thus are not valid in between the `open()` and `close()` method calls.

If an application does not wrap LOB operations between a set of `open()` and `close()` method calls, then each modification to the LOB implicitly opens and closes the LOB, thereby firing any triggers associated with changes to the LOB.

Restrictions for Opening and Closing LOBs

The LOB opening and closing mechanism has the following restrictions:

- An application must close all previously opened LOBs before committing a transaction. Failing to do so results in an error. If a transaction is rolled back, then all open LOBs are discarded along with the changes made (the LOBs are not closed), so associated triggers are not fired.
- While there is no limit to the number of open internal LOBs, there is a limit on the number of open files. Note that assigning an already opened locator to another locator does not count as opening a new LOB.
- It is an error to open or close the same internal LOB twice within the same transaction, either with different locators or with the same locator.
- It is an error to close a LOB that has not been opened.

Note: The definition of a transaction within which an open LOB value must be closed is one of the following:

- Between `SET TRANSACTION` and `COMMIT`
- Between `DATA MODIFYING DML` or `SELECT ... FOR UPDATE` and `COMMIT`
- Within an autonomous transaction block

A LOB opened when there is no transaction must be closed before the end of session. If there are LOBs open at the end of session, then the openness is discarded and no triggers of extensible indexes are fired.

Reading and Writing LOBs

OCCI provides methods for reading and writing LOBS. For nonstreamed reads and writes, the following methods are used:

- `Bfile::read()`
- `Blob::read()` and `Blob::write()`
- `Clob::read()` and `Clob::write()`

For streamed reads and writes, the following methods are used:

- `Bfile::getStream()`
- `Blob::getChunkSize()`, `Blob::getStream()`, and `Blob::writeChunk()`
- `Clob::getChunkSize()`, `Clob::getStream()`, and `Clob::writeChunk()`

The remainder of this section provides code examples for streamed and unstreamed reads and writes.

Nonstreamed Read

The following code example demonstrates how to obtain data from an internal LOB (in this example, a BLOB) that is not null by using a nonstreamed read:

```
ResultSet *rset=stmt->executeQuery("SELECT ad_composite FROM print_media
                                   WHERE product_id=6666");
while(rset->next())
```

```
{
    Blob blob=rset->getBlob(1);
    if(blob.isNull())
        cerr <<"Null Blob"<<endl;
    else
    {
        blob.open(OCCI_LOB_READONLY);

        const unsigned int BUFSIZE=100;
        char buffer[BUFSIZE];
        unsigned int readAmt=BUFSIZE;
        unsigned int offset=1;

        //reading readAmt bytes from offset 1
        blob.read(readAmt,buffer,BUFSIZE,offset);

        //process information in buffer
        .
        .
        .
        blob.close();
    }
}
stmt->closeResultSet(rset);
```

Reading all information from a BLOB without using streams, as in the preceding code example, requires that you keep track of the read offset and the amount remaining to be read, and pass these values to the `read()` method.

The following code example demonstrates how to read data from a BFILE, where the BFILE locator is not null, by using a nonstreamed read:

```
ResultSet *rset=stmt->executeQuery("SELECT ad_graphic FROM print_media
                                    WHERE product_id=6666");
while(rset->next())
{
    Bfile file=rset->getBfile(1);
    if(bfile.isNull())
        cerr <<"Null Bfile"<<endl;
    else
    {
        //display the directory alias and the file name of the BFILE
        cout <<"File Name:"<<bfile.getFileName()<<endl;
        cout <<"Directory Alias:"<<bfile.getDirAlias()<<endl;
```

```
        if(bfile.fileExists())
        {
            unsigned int length=bfile.length();
            char *buffer=new char[length];
            bfile.read(length, buffer, length, 1);
            //read all the contents of the BFILE into buffer, then process
            .
            .
            .
            delete[] buffer;
        }
        else
            cerr <<"File does not exist"<<endl;
    }
}
stmt->closeResultSet(rset);
```

Nonstreamed Write

The following code example demonstrates how to write data to an internal LOB (in this example, a BLOB) that is not null by using a nonstreamed write:

```
ResultSet *rset=stmt->executeQuery("SELECT ad_composite FROM print_media
                                    WHERE product_id=6666 FOR UPDATE");
while(rset->next())
{
    Blob blob=rset->getBlob(1);
    if(blob.isNull())
        cerr <<"Null Blob"<<endl;
    else
    {
        blob.open(OCCI_LOB_READWRITE);

        const unsigned int BUFSIZE=100;
        char buffer[BUFSIZE];
        unsigned int writeAmt=BUFSIZE;
        unsigned int offset=1;

        //writing writeAmt bytes from offset 1
        //contents of buffer are replaced after each writeChunk(),
        //typically with an fread()
        while(<fread "BUFSIZE" bytes into buffer succeeds>)
        {
            blob.writeChunk(writeAmt, buffer, BUFSIZE, offset);
            offset += writeAmt;
```

```
        }
        blob.writeChunk(<remaining amt>, buffer, BUFSIZE, offset);

        blob.close();
    }
}
stmt->closeResultSet(rset);
conn->commit();
```

In the preceding code example, the `writeChunk()` method is enclosed by the `open()` and `close()` methods. The `writeChunk()` method operates on a LOB that is currently open and ensures that triggers do not fire for every chunk read. The `write()` method can be used in place of the `writeChunk()` method in the preceding example; however, the `write()` method implicitly opens and closes the LOB.

Streamed Read

The following code example demonstrates how to obtain data from an internal LOB (in this example, a BLOB) that is already populated by using a streamed read:

```
ResultSet *rset=stmt->executeQuery("SELECT ad_composite FROM print_media
                                    WHERE product_id=6666");
while(rset->next())
{
    Blob blob=rset->getBlob(1);
    if(blob.isNull())
        cerr <<"Null Blob"<<endl;
    else
    {
        Stream *instream=blob.getStream(1,0);
        //reading from offset 1 to the end of the BLOB

        unsigned int size=blob.getChunkSize();
        char *buffer=new char[size];

        while((unsigned int length=instream->readBuffer(buffer,size))!=-1)
        {
            //process "length" bytes read into buffer
            .
            .
            .
        }
        delete[] buffer;
```

```
        blob.closeStream(instream);
    }
}
stmt->closeResultSet(rset);
```

Streamed Write

The following code example demonstrates how to write data to an internal LOB (in this example, a BLOB) that is already populated by using a streamed write:

```
ResultSet *rset=stmt->executeQuery("SELECT ad_composite FROM print_media
                                    WHERE product_id=6666 FOR UPDATE");
while(rset->next())
{
    Blob blob=rset->getBlob(1);
    if(blob.isNull())
        cerr <<"Null Blob"<<endl;
    else
    {
        char buffer[BUFSIZE];
        Stream *outstream=blob.getStream(1,0);

        //writing from buffer beginning at offset 1 until
        //a writeLastBuffer() method is issued.
        //contents of buffer are replaced after each writeBuffer(),
        //typically with an fread()
        while(<fread "BUFSIZE" bytes into buffer succeeds>)
            ostream->writeBuffer(buffer,BUFSIZE);
        ostream->writeLastBuffer(buffer,<remaining amt>);
        blob.closeStream(outstream);
    }
}
stmt->closeResultSet(rset);
conn->commit();
```

Improving Read and Write Performance

Read and write performance of internal LOBs can be improved by using either of the following methods:

- `getChunkSize()` method
- `writeChunk()` method

When to Use the getChunkSize() Method

Take advantage of the `getChunkSize()` methods of the `Blob` and `Clob` classes to improve the performance of internal LOB read and write operations. The `getChunkSize()` method returns the usable chunk size in bytes for BLOBs and in characters for CLOBs and NCLOBs. When a read or write is done by using data whose size is a multiple of the usable chunk size and starts on a chunk boundary, performance improves. You can specify the chunk size for a LOB column when you create a table that contains the LOB.

Calling the `getChunkSize()` method returns the usable chunk size of the LOB. An application can batch a series of write operations until an entire chunk can be written, rather than issuing multiple LOB write calls that operate on the same chunk.

To read through the end of a LOB, use the `read()` method with an amount of 4 GB. This avoids the round-trip involved with first calling the `getLength()` method because the `read()` method with an amount of 4 GB reads until the end of the LOB is reached.

> **Note:** For LOBs which store varying width characters, the `GetChunkSize()` method returns the number of Unicode characters that fit in a LOB chunk.

When to Use the writeChunk() Method

OCCI provides a shortcut to make it more efficient to write data to the end of a LOB. The `writeAppend()` methods of the `Blob` and `Clob` classes enables an application to append data to the end of a LOB without first requiring a call to the `getLength()` method to determine the starting point for a call to the `write()` method.

Updating LOBs

The following code example demonstrates how to update an internal LOB (in this example, a CLOB) to empty:

```
Clob clob(conn);
clob.setEmpty();
stmt->setSQL("UPDATE print_media SET ad_composite = :1
              WHERE product_id=6666");
stmt->setClob(1, clob);
stmt->executeUpdate();
```

```
conn->commit();
```

The following code example demonstrates how to update a BFILE:

```
Bfile bfile(conn);
bfile.setName("MEDIA_DIR", "img1.jpg");
stmt->setSQL("UPDATE print_media SET ad_graphic = :1
              WHERE product_id=6666");
stmt->setBfile(1, bfile);
stmt->executeUpdate();
conn->commit();
```

Objects with LOB Attributes

An OCCI application can use the overloaded operator `new()` to create a persistent or transient object with a LOB attribute. By default, all LOB attributes are constructed by using the default constructor and initialized to null.

Persistent Objects with LOB Attributes

It is possible to use OCCI to create a new persistent object with a LOB attribute. To do so, follow these steps:

1. Create a persistent object with a LOB attribute.

   ```
   Person *p=new(conn,"PERSON_TAB")Person();
   ```

2. Initialize the `Blob` object to empty.

   ```
   p->imgBlob = Blob(conn);
   p->imgBlob.setEmpty();
   ```

   If appropriate, then use the corresponding methods (`set`*`xxx`* methods and `get`*`xxx`* methods) on the `Person` object to accomplish the same thing.

3. Mark the `Blob` object as dirty.

   ```
   p->markModified();
   ```

4. Flush the object.

   ```
   p->flush();
   ```

5. Repin the object after obtaining a REF to it, thereby retrieving a refreshed version of the object from the database and acquiring an initialized LOB.

```
Ref<Person> r = p->getRef();
delete p;
p = r.ptr();
```

6. Write the data.

```
p->imgBlob.write( ... );
```

To create a persistent object with BFILE attributes, follow these steps:

1. Create a persistent object with a LOB attribute.

```
Person *p=new(conn,"PERSON_TAB")Person();
```

2. Initialize the `Bfile` object to empty.

```
p->imgBfile = Bfile(conn);
p->setName(<Directory Alias>,<File Name>);
```

3. Mark the object as dirty.

```
p->markModified();
```

4. Flush the object.

```
p->flush();
```

5. Read the data.

```
p->imgBfile.read( ... );
```

Transient Objects with LOB Attributes

An application can call the overloaded `new()` method and create a transient object with an internal LOB (BLOB, CLOB, NCLOB) attribute. However, you cannot perform any operations (for example, read or write) on the LOB attribute because transient LOBs are not currently supported. Calling the overloaded `new()` method to create a transient internal LOB type does not fail, but the application cannot use any LOB operations with the transient LOB.

An application can, however, create a transient object with a FILE attribute and use the FILE attribute to read data from the file stored in the server's file system. The application can also call the overloaded `new()` method to create a transient FILE and use that FILE to read from the server's file.

6 Metadata

This chapter describes how to retrieve metadata about result sets or the database as a whole.

It includes the following topics:

- Overview of Metadata
- Describing Database Metadata
- Attribute Reference

Overview of Metadata

Database objects have various attributes that describe them, and you obtain information about a particular schema object by performing a `DESCRIBE` operation for the object. The result can be accessed as an object of the `Metadata` class in that you can use class methods to get the actual values of an object. You accomplish this by passing object attributes as arguments to the various methods of the `Metadata` class.

You can perform an explicit `DESCRIBE` operation on the database as a whole, on the types and properties of the columns contained in a `ResultSet` class or on any of the following schema and subschema objects:

- Tables
- Views
- Procedures
- Functions
- Packages
- Types
- Type Attributes
- Type Methods
- Collections
- Synonyms
- Sequences
- Columns
- Argument
- Results
- Lists

You must specify the type of the attribute you are looking for. By using the `getAttributeCount`, `getAttributeId`, and `getAttributeType` methods of the `MetaData` class, you can scan through each available attribute.

All `DESCRIBE` information is cached until the last reference to it is deleted. Users are in this way prevented from accidentally trying to access `DESCRIBE` information that is already freed.

You obtain metadata by calling the `getMetaData` method on the `Connection` class in case of an explicit describe, or by calling the `getColumnListMetaData` method on the `ResultSet` class to get the metadata of the result set columns. Both methods return a `MetaData` object with the described information. The `MetaData` class provides the `get`*`xxx`* methods to access this information.

Notes on Types and Attributes

When performing `DESCRIBE` operations, be aware of the following issues:

- The `ATTR_TYPECODE` returns typecodes that represent the type supplied when you created a new type by using the `CREATE TYPE` statement. These typecodes are of the enumerated type `OCCITypeCode`, which are represented by `OCCI_TYPECODE` constants.

> **Note:** Internal PL/SQL types (boolean, indexed table) are not supported.

- The `ATTR_DATA_TYPE` returns types that represent the datatypes of the database columns. These values are of enumerated type `OCCIType`. For example, `LONG` types return `OCCI_SQLT_LNG` types.

Describing Database Metadata

Describing database metadata is equivalent to an explicit `DESCRIBE` operation. The object to describe must be an object in the schema. In describing a type, you call the `getMetaData` method from the connection, passing the name of the object or a `RefAny` object. To do this, you must initialize the environment in the `OBJECT` mode. The `getMetaData` method returns an object of type `MetaData`. Each type of `MetaData` object has a list of attributes that are part of the describe tree. The describe tree can then be traversed recursively to point to subtrees containing more information. More information about an object can be obtained by calling the `get`*`xxx`* methods.

If you need to construct a browser that describes the database and its objects recursively, then you can access information regarding the number of attributes for each object in the database (including the database), the attribute ID listing, and the attribute types listing. By using this information, you can recursively traverse the describe tree from the top node (the database) to the columns in the tables, the attributes of a type, the parameters of a procedure or function, and so on.

For example, consider the typical case of describing a table and its contents. You call the `getMetaData` method from the connection, passing the name of the table to be described. The `MetaData` object returned contains the table information. Since you are aware of the type of the object that you want to describe (table, column, type, collection, function, procedure, and so on), you can obtain the attribute list as shown in Table 6–1. You can retrieve the value into a variable of the type specified in the table by calling the corresponding `get`*`xxx`* method.

Table 6–1 Attribute Groupings

Attribute Type	Description
Parameter Attributes on page 6-10	Attributes belonging to all elements
Table and View Attributes on page 6-11	Attributes belonging to tables and views
Procedure, Function, and Subprogram Attributes on page 6-12	Attributes belonging to procedures, functions, and package subprograms
Package Attributes on page 6-13	Attributes belonging to packages
Type Attributes on page 6-13	Attributes belonging to types
Type Attribute Attributes on page 6-15	Attributes belonging to type attributes
Type Method Attributes on page 6-16	Attributes belonging to type methods
Collection Attributes on page 6-17	Attributes belonging to collection types
Synonym Attributes on page 6-19	Attributes belonging to synonyms
Sequence Attributes on page 6-19	Attributes belonging to sequences
Column Attributes on page 6-20	Attributes belonging to columns of tables or views
Argument and Result Attributes on page 6-21	Attributes belonging to arguments / results
List Attributes on page 6-23	Attributes that designate the list type

Table 6–1 Attribute Groupings (Continued)

Attribute Type	Description
Schema Attributes on page 6-24	Attributes specific to schemas
Database Attributes on page 6-24	Attributes specific to databases

Metatdata Code Examples

This section provides code examples for obtaining:

- `Connection` metadata
- `ResultSet` metadata

Connection Metadata Code Examples

The following code example demonstrates how to obtain metadata about attributes of a simple database table:

```
/* Create an environment and a connection to the HR database */
.
.
.
/* Call the getMetaData method on the Connection object obtained above */
MetaData emptab_metaData = connection->getMetaData("EMPLOYEES",
                                                   MetaData::PTYPE_TABLE);
/* Now that you have the metadata information on the EMPLOYEES table,
   call the getxxx methods using the appropriate attributes
*/
/* Call getString */
cout<<"Schema:"<<(emptab_metaData.getString(MetaData::ATTR_OBJ_SCHEMA))<<endl;

if(emptab_metaData.getInt(emptab_metaData::ATTR_PTYPE)==MetaData::PTYPE_TABLE)
    cout<<"EMPLOYEES is a table"<<endl;
else
    cout<<"EMPLOYEES is not a table"<<endl;

/* Call getInt to get the number of columns in the table */
int columnCount=emptab_metaData.getInt(MetaData::ATTR_NUM_COLS);
cout<<"Number of Columns:"<<columnCount<<endl

/* Call getTimestamp to get the timestamp of the table object */
Timestamp tstamp = emptab_metaData.getTimestamp(MetaData::ATTR_TIMESTAMP);
```

```
/* Now that you have the value of the attribute as a Timestamp object,
   you can call methods to obtain the components of the timestamp */
int year;
unsigned int month, day;
tstamp.getData(year, month, day);

/* Call getVector for attributes of list type,
for example ATTR_LIST_COLUMNS
*/
vector<MetaData>listOfColumns;
listOfColumns=emptab_metaData.getVector(MetaData::ATTR_LIST_COLUMNS);

/* Each of the list elements represents a column metadata,
   so now you can access the column attributes
*/
for (int i=0;i<listOfColumns.size();i++
{
    MetaData columnObj=listOfColumns[i];
    cout<<"Column Name:"<<(columnObj.getString(MetaData::ATTR_NAME))<<endl;
    cout<<"Data Type:"<<(columnObj.getInt(MetaData::ATTR_DATA_TYPE))<<endl;
    .
    .
    .
    /* and so on to obtain metadata on other column specific attributes */
}
```

The following code example demonstrates how to obtain metadata about a database table with a column containing a user-defined type:

```
/* Create an environment and a connection to the HR database */
.
.
.
/* Call the getMetaData method on the Connection object obtained above */
MetaData custtab_metaData = connection->getMetaData("CUSTOMERS",
    MetaData::PTYPE_TABLE);

/* Now that you have the metadata information on the CUSTOMERS table,
   call the getxxx methods using the appropriate attributes
*/
/* Call getString */
cout<<"Schema:"<<(custtab_metaData.getString(MetaData::ATTR_OBJ_SCHEMA))<<endl;

if(custtab_metaData.getInt(custtab_metaData::ATTR_PTYPE)==MetaData::PTYPE_TABLE)
    cout<<"CUSTOMERS is a table"<<endl;
```

```
else
    cout<<"CUSTOMERS is not a table"<<endl;

/* Call getVector to obtain a list of columns in the CUSTOMERS table */
vector<MetaData>listOfColumns;
listOfColumns=custtab_metaData.getVector(MetaData::ATTR_LIST_COLUMNS);

/* Assuming that the metadata for the column cust_address_typ
   is the fourth element in the list...
*/
MetaData customer_address=listOfColumns[3]

/* Now you can obtain the metadata for the customer_address attribute */
int typcode = customer_address.getInt(MetaData::ATTR_TYPECODE):
if(typcode==OCCI_TYPECODE_OBJECT)
    cout<<"customer_address is an object type"<<endl;
else
    cout<<"customer_address is not an object type"<<endl;

string objectName=customer_address.getString(MetaData::ATTR_OBJ_NAME);

/* Now that you have the name of the address object,
   the metadata of the attributes of the type can be obtained by using
   getMetaData on the connection by passing the object name
*/
MetaData address = connection->getMetaData(objectName);

/* Call getVector to obtain the list of the address object attributes */
vector<MetaData> attributeList =
                 address.getVector(MetaData::ATT_LIST_TYPE_ATTRS);

/* and so on to obtain metadata on other address object specific attributes */
```

The following code example demonstrates how to obtain metadata about an object when using a reference to it:

Assuming the following schema structure:

```
Type ADDRESS(street VARCHAR2(50), city VARCHAR2(20));
Table Person(id NUMBER, addr REF ADDRESS);


/* Create an environment and a connection to the HR database */
.
.
```

```
.
/* Call the getMetaData method on the Connection object obtained above */
MetaData perstab_metaData = connection->getMetaData("Person",
    MetaData::PTYPE_TABLE);

/* Now that you have the metadata information on the Person table,
   call the getxxx methods using the appropriate attributes
*/
/* Call getString */
cout<<"Schema:"<<(perstab_metaData.getString(MetaData::ATTR_OBJ_SCHEMA))<<endl;

if(perstab_metaData.getInt(perstab_metaData::ATTR_PTYPE)==MetaData::PTYPE_TABLE)
    cout<<"Person is a table"<<endl;
else
    cout<<"Person is not a table"<<endl;

/* Call getVector to obtain the list of columns in the Person table
*/
vector<MetaData>listOfColumns;
listOfColumns=perstab_metaData.getVector(MetaData::ATTR_LIST_COLUMNS);

/* Each of the list elements represents a column metadata,
   so now get the datatype of the column by passing ATTR_DATA_TYPE
   to getInt
*/
for(int i=0;i<numCols;i++)
{
int dataType=colList[i].getInt(MetaData::ATTR_DATA_TYPE);
/* If the datatype is a reference, get the Ref and obtain the metadata
   about the object by passing the Ref to getMetaData
*/
if(dataType==SQLT_REF)
    RefAny refTdo=colList[i].getRef(MetaData::ATTR_REF_TDO);

/* Now you can obtain the metadata about the object as shown below
MetaData tdo_metaData=connection->getMetaData(refTdo);

/* Now that you have the metadata about the TDO, you can obtain the metadata
   about the object
*/
```

Resultset Metadata Code Example

The following code example demonstrates how to obtain metadata about a select list from a `ResultSet` object:

```
/* Create an environment and a connection to the database */
.
.
.
/* Create a statement and associate it with a select clause */
string sqlStmt="SELECT * FROM EMPLOYEES";
Statement *stmt=conn->createStatement(sqlStmt);

/* Execute the statement to obtain a ResultSet */
ResultSet *rset=stmt->executeQuery();

/* Obtain the metadata about the select list */
vector<MetaData>cmd=rset->getColumnListMetaData();

/* The metadata is a column list and each element is a column metaData */
int dataType=cmd[i].getInt(MetaData::ATTR_DATA_TYPE);
.
.
.
```

The `getMetaData` method is called for the `ATTR_COLLECTION_ELEMENT` attribute only.

Attribute Reference

This section describes the attributes belonging to schema and subschema objects. The following attribute groupings are presented:

- Parameter Attributes
- Table and View Attributes
- Procedure, Function, and Subprogram Attributes
- Package Attributes
- Type Attributes
- Type Attribute Attributes
- Type Method Attributes
- Collection Attributes
- Synonym Attributes
- Sequence Attributes

- Column Attributes
- Argument and Result Attributes
- List Attributes
- Schema Attributes
- Database Attributes

Parameter Attributes

All elements have some attributes specific to that element and some generic attributes. Table 6–2 describes the attributes that belong to all elements:

Table 6–2 Attributes Belonging to All Elements

Attribute	Description	Attribute Datatype
ATTR_OBJ_ID	Object or schema ID	unsigned int
ATTR_OBJ_NAME	Object, schema, or database name	string
ATTR_OBJ_SCHEMA	Schema where object is located	string
ATTR_OBJ_PTYPE	Type of information described by the parameter.	int

Possible Values	Description
PTYPE_TABLE	Table
PTYPE_VIEW	View
PTYPE_PROC	Procedure
PTYPE_FUNC	Function
PTYPE_PKG	Package
PTYPE_TYPE	Type
PTYPE_TYPE_ATTR	Attribute of a type
PTYPE_TYPE_COLL	Collection type information
PTYPE_TYPE_METHOD	A method of a type
PTYPE_SYN	Synonym
PTYPE_SEQ	Sequence
PTYPE_COL	Column of a table or view
PTYPE_ARG	Argument of a function or procedure

Table 6–2 Attributes Belonging to All Elements (Continued)

Attribute	Description		Attribute Datatype
	PTYPE_TYPE_ARG	Argument of a type method	
	PTYPE_TYPE_RESULT	Results of a method	
	PTYPE_SCHEMA	Schema	
	PTYPE_DATABASE	Database	
ATTR_TIMESTAMP	The TIMESTAMP of the object this description is based on (Oracle DATE format)		Timestamp

The sections that follow list attributes specific to different types of elements.

Table and View Attributes

A parameter for a table or view (type PTYPE_TABLE or PTYPE_VIEW) has the following type-specific attributes described in Table 6–3:

Table 6–3 Attributes Belonging to Tables or Views

Attribute	Description	Attribute Datatype
ATTR_OBJID	Object ID	unsigned int
ATTR_NUM_COLS	Number of columns	int
ATTR_LIST_COLUMNS	Column list (type PTYPE_LIST)	vector<MetaData>
ATTR_REF_TDO	REF to the TDO of the base type in case of extent tables	RefAny
ATTR_IS_TEMPORARY	Identifies whether the table or view is temporary	bool
ATTR_IS_TYPED	Identifies whether the table or view is typed	bool
ATTR_DURATION	Duration of a temporary table. Values can be: OCCI_DURATION_SESSION (session) OCCI_DURATION_TRANS (transaction) OCCI_DURATION_NULL (table not temporary)	int

The additional attributes belonging to tables are described in Table 6–4.

Table 6–4 Attributes Specific to Tables

Attribute	Description	Attribute Datatype
ATTR_DBA	Data block address of the segment header	unsigned int
ATTR_TABLESPACE	Tablespace the table resides on	int
ATTR_CLUSTERED	Identifies whether the table is clustered	bool
ATTR_PARTITIONED	Identifies whether the table is partitioned	bool
ATTR_INDEX_ONLY	Identifies whether the table is index only	bool

Procedure, Function, and Subprogram Attributes

A parameter for a procedure or function (type PTYPE_PROC or PTYPE_FUNC) has the type-specific attributes described in Table 6–5.

Table 6–5 Attributes Belonging to Procedures or Functions

Attribute	Description	Attribute Datatype
ATTR_LIST_ARGUMENTS	Argument list Refer to List Attributes on page 6-23.	vector<MetaData>
ATTR_IS_INVOKER_RIGHTS	Identifies whether the procedure or function has invoker-rights.	int

The additional attributes belonging to package subprograms are described in Table 6–6.

Table 6–6 Attributes Belonging to Package Subprograms

Attribute	Description	Attribute Datatype
ATTR_NAME	Name of procedure or function	string
ATTR_OVERLOAD_ID	Overloading ID number (relevant in case the procedure or function is part of a package and is overloaded). Values returned may be different from direct query of a PL/SQL function or procedure.	int

Package Attributes

A parameter for a package (type PTYPE_PKG) has the type-specific attributes described in Table 6–7.

Table 6–7 Attributes Belonging to Packages

Attribute	Description	Attribute Datatype
ATTR_LIST_SUBPROGRAMS	Subprogram list Refer to List Attributes on page 6-23.	vector<MetaData>
ATTR_IS_INVOKER_RIGHTS	Identifies whether the package has invoker-rights	bool

Type Attributes

A parameter for a type (type PTYPE_TYPE) has attributes described in Table 6–8.

Table 6–8 Attributes Belonging to Types

Attribute	Description	Attribute Datatype
ATTR_REF_TDO	Returns the in-memory ref of the type descriptor object for the type, if the column type is an object type. ADD MORE	**RefAny**
ATTR_TYPECODE	Typecode. Can be OCCI_TYPECODE_OBJECT or OCCI_TYPECODE_NAMEDCOLLECTION. Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_COLLECTION_TYPECODE	Typecode of collection if type is collection; invalid otherwise. Can be OCCI_TYPECODE_VARRAY or OCCI_TYPECODE_TABLE. Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_VERSION	A null terminated string containing the user-assigned version	string
ATTR_IS_FINAL_TYPE	Identifies whether this is a final type	bool
ATTR_IS_INSTANTIABLE_TYPE	Identifies whether this is an instantiable type	bool

Table 6–8 Attributes Belonging to Types (Continued)

Attribute	Description	Attribute Datatype
ATTR_IS_SUBTYPE	Identifies whether this is a subtype	bool
ATTR_SUPERTYPE_SCHEMA_NAME	Name of the schema containing the supertype	string
ATTR_SUPERTYPE_NAME	Name of the supertype	string
ATTR_IS_INVOKER_RIGHTS	Identifies whether this type is invoker-rights	bool
ATTR_IS_INCOMPLETE_TYPE	Identifies whether this type is incomplete	bool
ATTR_IS_SYSTEM_TYPE	Identifies whether this is a system type	bool
ATTR_IS_PREDEFINED_TYPE	Identifies whether this is a predefined type	bool
ATTR_IS_TRANSIENT_TYPE	Identifies whether this is a transient type	bool
ATTR_IS_SYSTEM_GENERATED_TYPE	Identifies whether this is a system-generated type	bool
ATTR_HAS_NESTED_TABLE	Identifies whether this type contains a nested table attribute	bool
ATTR_HAS_LOB	Identifies whether this type contains a LOB attribute	bool
ATTR_HAS_FILE	Identifies whether this type contains a FILE attribute	bool
ATTR_COLLECTION_ELEMENT	Handle to collection element Refer to Collection Attributes on page 6-17	MetaData
ATTR_NUM_TYPE_ATTRS	Number of type attributes	unsigned int
ATTR_LIST_TYPE_ATTRS	List of type attributes Refer to List Attributes on page 6-23	vector<MetaData>
ATTR_NUM_TYPE_METHODS	Number of type methods	unsigned int
ATTR_LIST_TYPE_METHODS	List of type methods Refer to List Attributes on page 6-23	vector<MetaData>
ATTR_MAP_METHOD	Map method of type Refer to Type Method Attributes on page 6-16	MetaData

Table 6–8 Attributes Belonging to Types (Continued)

Attribute	Description	Attribute Datatype
ATTR_ORDER_METHOD	Order method of type Refer to Type Method Attributes on page 6-16	MetaData

Type Attribute Attributes

A parameter for an attribute of a type (type PTYPE_TYPE_ATTR) has the attributes described in Table 6–9.

Table 6–9 Attributes Belonging to Type Attributes

Attribute	Description	Attribute Datatype
ATTR_DATA_SIZE	Maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.	int
ATTR_TYPECODE	Typecode Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_DATA_TYPE	Datatype of the type attribute Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_NAME	A pointer to a string that is the type attribute name	string
ATTR_PRECISION	Precision of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply by NUMBER.	int
ATTR_SCALE	Scale of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply as NUMBER.	int

Table 6–9 Attributes Belonging to Type Attributes (Continued)

Attribute	Description	Attribute Datatype
ATTR_TYPE_NAME	A string that is the type name. The returned value will contain the type name if the datatype is SQLT_NTY or SQLT_REF. If the datatype is SQLT_NTY, then the name of the named datatype's type is returned. If the datatype is SQLT_REF, then the type name of the named datatype pointed to by the REF is returned.	string
ATTR_SCHEMA_NAME	String with the schema name under which the type has been created	string
ATTR_REF_TDO	Returns the in-memory REF of the TDO for the type, if the column type is an object type.	RefAny
ATTR_CHARSET_ID	Character set ID, if the type attribute is of a string or character type	int
ATTR_CHARSET_FORM	Character set form, if the type attribute is of a string or character type	int
ATTR_FSPRECISION	The fractional seconds precision of a datetime or interval	int
ATTR_LFPRECISION	The leading field precision of an interval	int

Type Method Attributes

A parameter for a method of a type (type PTYPE_TYPE_METHOD) has the attributes described in Table 6–10.

Table 6–10 Attributes Belonging to Type Methods

Attribute	Description	Attribute Datatype
ATTR_NAME	Name of method (procedure or function)	string
ATTR_ENCAPSULATION	Encapsulation level of the method (either OCCI_TYPEENCAP_PRIVATE or OCCI_TYPEENCAP_PUBLIC)	int
ATTR_LIST_ARGUMENTS	Argument list	vector<MetaData>

Table 6–10 Attributes Belonging to Type Methods (Continued)

Attribute	Description	Attribute Datatype
ATTR_IS_CONSTRUCTOR	Identifies whether the method is a constructor	bool
ATTR_IS_DESTRUCTOR	Identifies whether the method is a destructor	bool
ATTR_IS_OPERATOR	Identifies whether the method is an operator	bool
ATTR_IS_SELFISH	Identifies whether the method is selfish	bool
ATTR_IS_MAP	Identifies whether the method is a map method	bool
ATTR_IS_ORDER	Identifies whether the method is an order method	bool
ATTR_IS_RNDS	Identifies whether "Read No Data State" is set for the method	bool
ATTR_IS_RNPS	Identifies whether "Read No Process State" is set for the method	bool
ATTR_IS_WNDS	Identifies whether "Write No Data State" is set for the method	bool
ATTR_IS_WNPS	Identifies whether "Write No Process State" is set for the method	bool
ATTR_IS_FINAL_METHOD	Identifies whether this is a final method	bool
ATTR_IS_INSTANTIABLE_METHOD	Identifies whether this is an instantiable method	bool
ATTR_IS_OVERRIDING_METHOD	Identifies whether this is an overriding method	bool

Collection Attributes

A parameter for a collection type (type PTYPE_COLL) has the attributes described in Table 6–11.

Table 6–11 Attributes Belonging to Collection Types

Attribute	Description	Attribute Datatype
ATTR_DATA_SIZE	Maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.	int
ATTR_TYPECODE	Typecode Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_DATA_TYPE	The datatype of the type attribute Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_NUM_ELEMENTS	Number of elements in an array. Only valid for collections that are arrays.	unsigned int
ATTR_NAME	A pointer to a string that is the type attribute name	string
ATTR_PRECISION	Precision of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply as NUMBER.	int
ATTR_SCALE	Scale of numeric type attributes. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply as NUMBER.	int
ATTR_TYPE_NAME	String that is the type name. The returned value will contain the type name if the datatype is SQLT_NTY or SQLT_REF. If the datatype is SQLT_NTY, then the name of the named datatype's type is returned. If the datatype is SQLT_REF, then the type name of the named datatype pointed to by the REF is returned	string
ATTR_SCHEMA_NAME	String with the schema name under which the type has been created	string

Table 6–11 Attributes Belonging to Collection Types (Continued)

Attribute	Description	Attribute Datatype
ATTR_REF_TDO	Maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.	RefAny
ATTR_CHARSET_ID	Typecode Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_CHARSET_FORM	The datatype of the type attribute Refer to Notes on Types and Attributes on page 6-3.	int

Synonym Attributes

A parameter for a synonym (type PTYPE_SYN) has the attributes described in Table 6–12.

Table 6–12 Attributes Belonging to Synonyms

Attribute	Description	Attribute Datatype
ATTR_OBJID	Object ID	unsigned int
ATTR_SCHEMA_NAME	Null-terminated string containing the schema name of the synonym translation	string
ATTR_NAME	Null-terminated string containing the object name of the synonym translation	string
ATTR_LINK	Null-terminated string containing the database link name of the synonym translation	string

Sequence Attributes

A parameter for a sequence (type PTYPE_SEQ) has the attributes described in Table 6–13.

Table 6–13 Attributes Belonging to Sequences

Attribute	Description	Attribute Datatype
ATTR_OBJID	Object ID	unsigned int
ATTR_MIN	Minimum value (in Oracle number format)	Number

Table 6–13 Attributes Belonging to Sequences (Continued)

Attribute	Description	Attribute Datatype
ATTR_MAX	Maximum value (in Oracle number format)	Number
ATTR_INCR	Increment (in Oracle number format)	Number
ATTR_CACHE	Number of sequence numbers cached; zero if the sequence is not a cached sequence (in Oracle number format)	Number
ATTR_ORDER	Identifies whether the sequence is ordered?	bool
ATTR_HW_MARK	High-water mark (in Oracle number format)	Number

Column Attributes

A parameter for a column of a table or view (type PTYPE_COL) has the attributes described in Table 6–14.

Table 6–14 Attributes Belonging to Columns of Tables or Views

Attribute	Description	Attribute Datatype
ATTR_DATA_SIZE	Column length in codepoints. The number of codepoints allowed in the column.	int
ATTR_DATA_TYPE	Type of length semantics of the column. Valid values are 0 for byte-length semantics and 1 for codepoint-length semantics.	int
ATTR_NAME	Maximum size of the column. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.	string
ATTR_PRECISION	The datatype of the column Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_SCALE	Pointer to a string that is the column name	int

Table 6–14 Attributes Belonging to Columns of Tables or Views (Continued)

Attribute	Description	Attribute Datatype
ATTR_IS_NULL	The precision of numeric columns. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, NUMBER(p, s) can be represented simply as NUMBER.	bool
ATTR_TYPE_NAME	Scale of numeric columns. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply as NUMBER.	string
ATTR_SCHEMA_NAME	Returns 0 if null values are not permitted for the column	string
ATTR_REF_TDO	Returns a string that is the type name. The returned value will contain the type name if the datatype is SQLT_NTY or SQLT_REF. If the datatype is SQLT_NTY, then the name of the named datatype's type is returned. If the datatype is SQLT_REF, then the type name of the named datatype pointed to by the REF is returned	RefAny
ATTR_CHARSET_ID	Returns a string with the schema name under which the type has been created	int
ATTR_CHARSET_FORM	The REF of the TDO for the type, if the column type is an object type	int

Argument and Result Attributes

A parameter for an argument or a procedure or function type (type PTYPE_ARG), for a type method argument (type PTYPE_TYPE_ARG), or for method results (type PTYPE_TYPE_RESULT) has the attributes described in Table 6–15.

Table 6–15 Attributes Belonging to Arguments / Results

Attribute	Description	Attribute Datatype
ATTR_NAME	Returns a pointer to a string which is the argument name	string

Table 6–15 Attributes Belonging to Arguments / Results (Continued)

Attribute	Description	Attribute Datatype
ATTR_POSITION	Position of the argument in the argument list. Always returns 0.	int
ATTR_TYPECODE	Typecode Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_DATA_TYPE	Datatype of the argument Refer to Notes on Types and Attributes on page 6-3.	int
ATTR_DATA_SIZE	Size of the datatype of the argument. This length is returned in bytes and not characters for strings and raws. Returns 22 for NUMBER.	int
ATTR_PRECISION	Precision of numeric arguments. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply as NUMBER.	int
ATTR_SCALE	Scale of numeric arguments. If the precision is nonzero and scale is -127, then it is a FLOAT; otherwise a NUMBER(p, s). If precision is 0, then NUMBER(p, s) can be represented simply as NUMBER.	int
ATTR_LEVEL	Datatype levels. This attribute always returns 0.	int
ATTR_HAS_DEFAULT	Indicates whether an argument has a default	int
ATTR_LIST_ARGUMENTS	The list of arguments at the next level (when the argument is of a record or table type)	vector<MetaData>
ATTR_IOMODE	Indicates the argument mode. Valid values are 0 for IN (OCCI_TYPEPARAM_IN), 1 for OUT (OCCI_TYPEPARAM_OUT), and 2 for IN/OUT (OCCI_TYPEPARAM_INOUT)	int
ATTR_RADIX	Returns a radix (if number type)	int
ATTR_IS_NULL	Returns 0 if null values are not permitted for the column	int

Table 6–15 Attributes Belonging to Arguments / Results (Continued)

Attribute	Description	Attribute Datatype
ATTR_TYPE_NAME	Returns a string that is the type name, or the package name in the case of package local types. The returned value contains the type name if the datatype is SQLT_NTY or SQLT_REF. If the datatype is SQLT_NTY, then the name of the named datatype's type is returned. If the datatype is SQLT_REF, then the type name of the named datatype pointed to by the REF is returned.	string
ATTR_SCHEMA_NAME	For SQLT_NTY or SQLT_REF, returns a string with the schema name under which the type was created, or under which the package was created in the case of package local types	string
ATTR_SUB_NAME	For SQLT_NTY or SQLT_REF, returns a string with the type name, in the case of package local types	string
ATTR_LINK	For SQLT_NTY or SQLT_REF, returns a string with the database link name of the database on which the type exists. This can happen only in the case of package local types, when the package is remote.	string
ATTR_REF_TDO	Returns the REF of the TDO for the type, if the argument type is an object	RefAny
ATTR_CHARSET_ID	Returns the character set ID if the argument is of a string or character type	int
ATTR_CHARSET_FORM	Returns the character set form if the argument is of a string or character type	int

List Attributes

A list type of attribute can be described for all the elements in the list. In case of a function argument list, position 0 has a parameter for return values (PTYPE_ARG).

The list is described iteratively for all the elements. The results are stored in a C++ `vector<MetaData>`. Call the `getVector` method to describe list type of attributes. Table 6–16 displays the list attributes.

Table 6–16 Values for ATTR_LIST_TYPE

Possible Values	Description
ATTR_LIST_COLUMNS	Column list
ATTR_LIST_ARGUMENTS	Procedure or function arguments list

Table 6–16 Values for ATTR_LIST_TYPE (Continued)

Possible Values	Description
ATTR_LIST_SUBPROGRAMS	Subprogram list
ATTR_LIST_TYPE_ATTRIBUTES	Type attribute list
ATTR_LIST_TYPE_METHODS	Type method list
ATTR_LIST_OBJECTS	Object list within a schema
ATTR_LIST_SCHEMAS	Schema list within a database

Schema Attributes

A parameter for a schema type (type PTYPE_SCHEMA) has the attributes described in Table 6–17.

Table 6–17 Attributes Specific to Schemas

Attribute	Description	Attribute Datatype
ATTR_LIST_OBJECTS	List of objects in the schema	string

Database Attributes

A parameter for a database (type PTYPE_DATABASE) has the attributes described in Table 6–18.

Table 6–18 Attributes Specific to Databases

Attribute	Description	Attribute Datatype
ATTR_VERSION	Database version	string
ATTR_CHARSET_ID	Database character set ID from the server handle	int
ATTR_NCHARSET_ID	Database native character set ID from the server handle	int
ATTR_LIST_SCHEMAS	List of schemas (type PTYPE_SCHEMA) in the database	vector<MetaData>
ATTR_MAX_PROC_LEN	Maximum length of a procedure name	unsigned int
ATTR_MAX_COLUMN_LEN	Maximum length of a column name	unsigned int

Table 6–18 Attributes Specific to Databases (Continued)

Attribute	Description	Attribute Datatype
ATTR_CURSOR_COMMIT_BEHAVIOR	How a COMMIT operation affects cursors and prepared statements in the database. Values are: OCCI_CURSOR_OPEN for preserving cursor state as before the commit operation and OCCI_CURSOR_CLOSED for cursors that are closed on COMMIT, although the application can still reexecute the statement without preparing it again.	int
ATTR_MAX_CATALOG_NAMELEN	Maximum length of a catalog (database) name	int
ATTR_CATALOG_LOCATION	Position of the catalog in a qualified table. Valid values are OCCI_CL_START and OCCI_CL_END.	int
ATTR_SAVEPOINT_SUPPORT	Identifies whether the database supports savepoints. Valid values are OCCI_SP_SUPPORTED and OCCI_SP_UNSUPPORTED.	int
ATTR_NOWAIT_SUPPORT	Identifies whether the database supports the nowait clause. Valid values are OCCI_NW_SUPPORTED and OCCI_NW_UNSUPPORTED.	int
ATTR_AUTOCOMMIT_DDL	Identifies whether the autocommit mode is required for DDL statements. Valid values are OCCI_AC_DDL and OCCI_NO_AC_DDL.	int
ATTR_LOCKING_MODE	Locking mode for the database. Valid values are OCCI_LOCK_IMMEDIATE and OCCI_LOCK_DELAYED.	int

See Also:

- Appendix A, "OCCI Demonstration Programs" and the code example occidesc.cpp for an illustration of the concepts covered in this chapter

7

How to Use the Object Type Translator Utility

This chapter discusses the Object Type Translator (OTT) utility, which is used to map database object types, LOB types, and named collection types to C structures and C++ class declarations for use in OCCI, OCI, and Pro*C/C++ applications.

This chapter includes the following topics:

- How to Use the OTT Utility
- Creating Types in the Database
- Invoking the OTT Utility
- Overview of the INTYPE File
- OTT Utility Datatype Mappings
- Overview of the OUTTYPE File
- The OTT Utility and OCCI Applications
- Example OCCI Application
- OTT Utility Reference

Overview of the Object Type Translator Utility

The Object Type Translator (OTT) utility assists in the development of applications that make use of user-defined types in an Oracle database server.

Through the use of SQL `CREATE TYPE` statements, you can create object types. The definitions of these types are stored in the database and can be used in the creation of database tables. Once these tables are populated, an Oracle C++ Call Interface (OCCI), Oracle Call Interface (OCI), or Pro*C/C++ programmer can access objects stored in the tables.

An application that accesses object data must be able to represent the data in a host language format. This is accomplished by representing object types as structures in C or as classes in C++.

You could code structures or classes manually to represent database object types, but this is time-consuming and error-prone. The OTT utility simplifies this step by automatically generating the appropriate structure declarations for C or the appropriate classes for C++.

For Pro*C/C++, the application only needs to include the header file generated by the OTT utility. In OCI, the application also needs to call an initialization function generated by the OTT utility.

For OCCI, the application must include and link the following files:

- Include the header file containing the generated class declarations
- Include the header file containing the prototype for the function to register the mappings
- Link with the C++ source file containing the static methods to be called by OCCI while instantiating the objects
- Link with the file containing the function to register the mappings with the environment and call this function

For C, in addition to creating C structures that represent stored datatypes, the OTT utility also generates parallel indicator structures that indicate whether an object type or its fields are null. This is not the case for C++.

How to Use the OTT Utility

To translate database types to C or C++ representation, you must explicitly invoke the OTT utility. In addition, OCI programmers must initialize a data structure called

the Type Version Table with information about the user-defined types required by the program. Code to perform this initialization is generated by the OTT utility

In Pro*C/C++, the type version information is recorded in the OUTTYPE file which is passed as a parameter to Pro*C/C++.

OCCI programmers must invoke the function to register the mappings with the environment. This function is generated by the OTT utility.

On most operating systems, the OTT utility is invoked on the command line. It takes as input an INTYPE file, and it generates an OUTTYPE file and one or more C header files or one or more C++ header files and C++ method files. An optional implementation file is generated for OCI programmers. For OCCI programmers, an additional C++ methods file to register mappings is generated along with its corresponding header file containing the prototype.

Example for C

The following example is of a command line that invokes the OTT utility and generates C structs:

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=c hfile=demo.h
```

This command causes the OTT utility to connect to the database with username `scott` and password `tiger`, and to translate database types to C structures, based on instructions in the INTYPE file, `demoin.typ`. The resulting structures are output to the `header` file, `demo.h`, for the host language (C) specified by the code parameter. The OUTTYPE file, `demoout.typ`, receives information about the translation.

Each of these parameters is described in more detail in later sections of this chapter.

Sample demoin.typ and demoout.typ files

This is an example of a `demoin.typ` file:

```
CASE=LOWER
TYPE employee
```

This is an example of a `demoout.typ` file:

```
CASE = LOWER
TYPE SCOTT.EMPLOYEE AS employee
    VERSION = "$8.0"
    HFILE = demo.h
```

In this example, the `demoin.typ` file contains the type to be translated, preceded by the keyword `TYPE`. The structure of the `OUTTYPE` file is similar to the `INTYPE` file, with the addition of information obtained by the OTT utility.

Once the OTT utility has completed the translation, the header file contains a C structure representation of each type specified in the `INTYPE` file, and a null indicator structure corresponding to each type.

Let us assume the employee type listed in the `INTYPE` file is defined as follows:

```
CREATE TYPE employee AS OBJECT
(
    name        VARCHAR2(30),
    empno       NUMBER,
    deptno      NUMBER,
    hiredate    DATE,
    salary      NUMBER
);
```

The header file, `demo.h`, generated by the OTT utility includes, among other items, the following declarations:

```
struct employee
{
    OCIString * name;
    OCINumber empno;
    OCINumber deptno;
    OCIDate   hiredate;
    OCINumber salary;
};
typedef struct emp_type emp_type;

struct employee_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd deptno;
    OCIInd hiredate;
    OCIInd salary;
};
typedef struct employee_ind employee_ind;
```

Note: Parameters in the INTYPE file control the way generated structures are named. In this example, the structure name employee matches the database type name employee. The structure name is in lowercase because of the line CASE=lower in the INTYPE file.

See Also: "OTT Utility Datatype Mappings" on page 7-16 for more information about types.

Example for C++

The following example is of an OTT command that invokes the OTT utility and generates C++ classes:

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=cpp
hfile=demo.h cppfile=demo.cpp mapfile=RegisterMappings.cpp
```

This command causes the OTT utility to connect to the database as username `scott` with password `tiger`, and use the `demoin.typ` file as the `INTYPE` file, and the `demoout.typ` file as the `OUTTYPE` file. The resulting declarations are output to the file `demo.h` in C++, specified by the `CODE=cpp` parameter, the method implementations written to the file `demo.cpp`, and the functions to register mappings is written to `RegisterMappings.cpp` with its prototype written to `RegisterMappings.h`.

By using the same `demoin.typ` file and `employee` type as in the previous section, the OTT utility generates the following files:

- `demo.h`
- `demo.cpp`
- `RegisterMappings.h`
- `RegisterMappings.cpp`
- `demoout.typ`

The contents of these files are displayed in the following sections:

- Contents of the demo.h File
- Contents of the demo.cpp File
- Contents of the RegisterMappings.h File
- Contents of the RegisterMappings.cpp File

- Contents of the demoout.typ File

Contents of the demo.h File

```
#ifndef DEMO_ORACLE
# define DEMO_ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif


/************************************************************/
//  generated declarations for the EMPLOYEE object type.
/************************************************************/

class employee : public oracle::occi::PObject {

protected:

   OCCI_STD_NAMESPACE::string name;
   oracle::occi::Number empno;
   oracle::occi::Number deptno;
   oracle::occi::Date hiredate;
   oracle::occi::Number salary;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   employee();

   employee(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);
```

```
   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

#endif
```

Contents of the demo.cpp File

```
#ifndef DEMO_ORACLE
# include "demo.h"
#endif


/*****************************************************************/
//  generated method implementations for the EMPLOYEE object type.
/*****************************************************************/

void *employee::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *employee::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.EMPLOYEE");
}

OCCI_STD_NAMESPACE::string employee::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.EMPLOYEE");
}

employee::employee()
{
}

void *employee::readSQL(void *ctxOCCI_)
{
  employee *objOCCI_ = new employee(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
```

```
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void employee::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   name = streamOCCI_.getString();
   empno = streamOCCI_.getNumber();
   deptno = streamOCCI_.getNumber();
   hiredate = streamOCCI_.getDate();
   salary = streamOCCI_.getNumber();
}

void employee::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  employee *objOCCI_ = (employee *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void employee::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
```

```
   streamOCCI_.setString(name);
   streamOCCI_.setNumber(empno);
   streamOCCI_.setNumber(deptno);
   streamOCCI_.setDate(hiredate);
   streamOCCI_.setNumber(salary);
}
```

Contents of the RegisterMappings.h File

```
#ifndef REGISTERMAPPINGS_ORACLE
# define REGISTERMAPPINGS_ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif

#ifndef DEMO_ORACLE
# include "demo.h"
#endif

void RegisterMappings(oracle::occi::Environment* envOCCI_);

#endif
```

Contents of the RegisterMappings.cpp File

```
#ifndef REGISTERMAPPINGS_ORACLE
# include "registermappings.h"
#endif

void RegisterMappings(oracle::occi::Environment* envOCCI_)
{
  oracle::occi::Map *mapOCCI_ = envOCCI_->getMap();
  mapOCCI_->put("SCOTT.EMPLOYEE", employee::readSQL, employee::writeSQL);
}
```

Contents of the demoout.typ File

```
CASE = LOWER
MAPFILE = RegisterMappings.cpp
MAPFUNC = RegisterMappings
```

```
TYPE SCOTT.EMPLOYEE AS employee
  VERSION = "$8.0"
  HFILE = demo.h
```

See Also: Example for Extending OTT Classes on page 7-49 for a complete C++ example.

Creating Types in the Database

The first step in using the OTT utility is to create object types or named collection types and store them in the database. This is accomplished through the use of the SQL `CREATE TYPE` statement.

The following is an example of statements that create objects:

```
CREATE TYPE FULL_NAME AS OBJECT (first_name CHAR(20), last_name CHAR(20));
CREATE TYPE ADDRESS AS OBJECT (state CHAR(20), zip CHAR(20));
CREATE TYPE ADDRESS_TAB AS VARRAY(3) OF REF ADDRESS;
CREATE TYPE PERSON AS OBJECT (id NUMBER, name FULL_NAME, curr_addr REF ADDRESS,
    prev_addr_l ADDRESS_TAB) NOT FINAL;
CREATE TYPE STUDENT UNDER PERSON (school_name CHAR(20));
```

Invoking the OTT Utility

After creating types in the database, the next step is to invoke the OTT utility.

Specifying OTT Parameters

You can specify OTT parameters either on the command line or in a configuration file. Certain parameters can also be specified in the `INTYPE` file.

If you specify a parameter in more than one place, then its value on the command line takes precedence over its value in the `INTYPE` file. The value in the `INTYPE` file takes precedence over its value in a user-defined configuration file, which takes precedence over its value in the default configuration file.

Parameter precedence then is as follows:

1. OTT command line
2. Value in `INTYPE` file
3. User-defined configuration file
4. Default configuration file

For global options (that is, options on the command line or options at the beginning of the INTYPE file before any TYPE statements), the value on the command line overrides the value in the INTYPE file. (The options that can be specified globally in the INTYPE file are CASE, INITFILE, INITFUNC, MAPFILE and MAPFUNC, but not HFILE or CPPFILE.) Anything in the INTYPE file in a TYPE specification applies to a particular type only and overrides anything on the command line that would otherwise apply to the type. So if you enter TYPE person HFILE=p.h, then it applies to person only and overrides the HFILE on the command line. The statement is not considered a command line parameter.

Setting Parameters on the Command Line

Parameters (also called options) set on the command line override any parameters or option set elsewhere.

> **See Also:** "Invoking the OTT Utility" on page 7-10 for more information

Setting Parameters in the INTYPE File

The INTYPE file gives a list of types for the OTT utility to translate.

The parameters CASE, CPPFILE, HFILE, INITFILE, INITFUNC, MAPFILE, and MAPFUNC can appear in the INTYPE file. See "Overview of the INTYPE File" on page 7-14 for more information.

> **See Also:** "Overview of the INTYPE File" on page 7-14 for more information

Setting Parameters in the Configuration File

A configuration file is a text file that contains OTT parameters. Each nonblank line in the file contains one parameter, with its associated value or values. If more than one parameter is put on a line, then only the first one will be used. No blank space is allowed on any nonblank line of a configuration file.

A configuration file can be named on the command line. In addition, a default configuration file is always read. This default configuration file must always exist, but can be empty. The name of the default configuration file is ottcfg.cfg, and the location of the file is operating system-specific.

> **See Also:** Your operating system-specific documentation for more information about the location of the default configuration file

Invoking the OTT Utility on the Command Line

On most platforms, the OTT utility is invoked on the command line. You can specify the input and output files and the database connection information at the command line, among other things.

> **See Also:** Your operating system-specific documentation to see how to invoke the OTT utility on your operating system

Invoking the OTT Utility for C++

The following is an example of invoking the OTT utility that generates C++ classes:

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=cpp
hfile=demo.h cppfile=demo.cpp mapfile=RegisterMappings.cpp
```

> **Note:** No spaces are permitted around the equals sign (=) on the OTT command line.

Description of Elements Used on the OTT Command Line

The following sections describe the elements of the command lines used in these examples:

- ott Command
- userid Parameter
- intype Parameter
- outtype Parameter
- code Parameter
- hfile Parameter
- cppfile Parameter
- mapfile Parameter

> **See Also:** "OTT Utility Reference" on page 7-88 for a detailed discussion of the various OTT command line parameters.

ott Command Causes the OTT utility to be invoked. It must be the first item on the command line.

userid Parameter Specifies the database connection information that the OTT utility will use. In both of the preceding examples, the OTT utility attempts to connect with username `scott` and password `tiger`.

intype Parameter Specifies the name of the `INTYPE` file. In both of the preceding examples, the name of the `INTYPE` file is specified as `demoin.typ`.

outtype Parameter Specifies the name of the `OUTTYPE` file. When the OTT utility generates the header file, it also writes information about the translated types into the `OUTTYPE` file. This file contains an entry for each of the types that is translated, including its version string, and the header file to which its C or C++ representation is written.

In both of the preceding examples, the name of the `OUTTYPE` file is specified as `demoout.typ`.

Note: If the file specified by the `OUTTYPE` parameter already exists, it will be overwritten when the OTT utility runs, with one exception: if the contents of the file as generated by the OTT utility are identical to the contents of the existing `OUTTYPE` file, the OTT utility will not actually write to the file. This preserves the modification time of the file so that UNIX `make` utility and similar facilities on other platforms do not perform unnecessary recompilations.

---

code Parameter Specifies the target language for the translation. The following values are valid:

- ANSI_C (for ANSI C)
- C (equivalent to ANSI_C)
- KR_C (for Kernighan & Ritchie C)
- CPP (for C++)

There is currently no default value, so this parameter is required.

Structure declarations are identical for the C language options: C, ANSI_C, and KR_C. The style in which the initialization function is defined in the `INITFILE` file depends on whether KR_C is used. If the `INITFILE` parameter is not used, then the C language options are equivalent.

Note: In the previous example for C, the target language is specified as C (`code=c`). In the previous example for C++, the language is C++ (`code=cpp`) and both `CPPFILE` and `MAPFILE` parameters are specified.

If you are generating C++ classes by setting the `CODE` parameter to `cpp`, then you must use the `CPPFILE` and the `MAPFILE` parameters.

hfile Parameter Specifies the name of the C or C++ header file to which the generated C structures or C++ classes are written.

Note: If the file specified by the `HFILE` parameter already exists, it is overwritten, with one exception: if the contents of the file as generated by the OTT utility are identical to the contents of the existing `HFILE` file, the OTT utility does not actually write to the file. This preserves the modification time of the file so that UNIX `make` utility and similar facilities on other platforms do not perform unnecessary recompilations.

See Also: "OTT Command Line Syntax" on page 7-88 for information about the `CPPFILE` and `MAPFILE` parameters

cppfile Parameter Specifies the name of the C++ source file into which the method implementations are written. The methods generated in this file are called by OCCI while instantiating the objects and are not to be called directly in the an application. This parameter is only needed for OCCI applications.

mapfile Parameter Specifies the name of the C++ source file into which the function to register the mappings with the environment is written. A corresponding header file is created containing the prototype for the function. This function to register mappings is only used for OCCI applications.

Overview of the INTYPE File

When you run the OTT utility, the `INTYPE` file tells the OTT utility which database types should be translated. The `INTYPE` file also controls the naming of the generated structures or classes. You can either create an `INTYPE` file or use the `OUTTYPE` file of a previous invocation of the OTT utility. If you do not use an

INTYPE file, then all types in the schema to which the OTT utility connects are translated.

The following is an example of a user-created INTYPE file:

```
CASE=LOWER
TYPE employee
    TRANSLATE SALARY$ AS salary
              DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

In the first line, the CASE parameter indicates that generated C identifiers should be in lowercase. However, this CASE parameter is only applied to those identifiers that are not explicitly mentioned in the INTYPE file. Thus, employee and ADDRESS would always result in C structures employee and ADDRESS, respectively. The members of these structures are named in lowercase.

The lines that begin with the TYPE keyword specify which types in the database should be translated. In this case, the EMPLOYEE, ADDRESS, ITEM, PERSON, and PURCHASE_ORDER types are set to be translated.

The TRANSLATE ... AS keywords specify that the name of an object attribute should be changed when the type is translated into a C structure. In this case, the SALARY$ attribute of the employee type is translated to salary.

The AS keyword in the final line specifies that the name of an object type should be changed when it is translated into a structure. In this case, the purchase_order database type is translated into a structure called p_o.

If you do not use AS to translate a type or attribute name, then the database name of the type or attribute will be used as the C identifier name, except that the CASE parameter will be observed, and any characters that cannot be mapped to a legal C identifier character will be replaced by an underscore character (_). Consider the following reasons for translating a type or attribute:

- The name contains characters other than letters, digits, and underscores
- The name conflicts with a C keyword
- The type name conflicts with another identifier in the same scope. This can happen, for example, if the program uses two types with the same name from different schemas.
- The programmer prefers a different name

The OTT utility may need to translate additional types that are not listed in the `INTYPE` file. This is because the OTT utility analyzes the types in the `INTYPE` file for type dependencies before performing the translation, and it translates other types as necessary. For example, if the `ADDRESS` type were not listed in the `INTYPE` file, but the `Person` type had an attribute of type `ADDRESS`, then the OTT utility would still translate `ADDRESS` because it is required to define the `Person` type.

Note: As of release 9.0.1, you may indicate whether the OTT utility is to generate required object types that are not specified in the `INTYPE` file. Set `TRANSITIVE=FALSE` so the OTT utility will not to generate required object types. The default is `TRANSITIVE=TRUE`.

A normal case insensitive SQL identifier can be spelled in any combination of uppercase and lowercase in the `INTYPE` file, and is not quoted.

Use quotation marks, such as `TYPE "Person"` to reference SQL identifiers that have been created in a case sensitive manner, for example, `CREATE TYPE "Person"`. A SQL identifier is case sensitive if it was quoted when it was declared. Quotation marks can also be used to refer to a SQL identifier that is an OTT-reserved word, for example, `TYPE "CASE"`. In this case, the quoted name must be in uppercase if the SQL identifier was created in a case insensitive manner, for example, `CREATE TYPE Case`. If an OTT-reserved word is used to refer to the name of a SQL identifier but is not quoted, then the OTT utility will report a syntax error in the `INTYPE` file.

See Also:

- "Structure of the INTYPE File" on page 7-98 for a more detailed specification of the structure of the `INTYPE` file and the available options.
- "CASE Parameter" on page 7-91 for further information regarding the `CASE` parameter

OTT Utility Datatype Mappings

When the OTT utility generates a C structure or a C++ class from a database type, the structure or class contains one element corresponding to each attribute of the object type. The datatypes of the attributes are mapped to types that are used in Oracle object data types. The datatypes found in Oracle include a set of predefined,

primitive types and provide for the creation of user-defined types, like object types and collections.

The set of predefined types includes standard types that are familiar to most programmers, including number and character types. It also includes large object datatypes (for example, BLOB or CLOB).

Oracle also includes a set of predefined types that are used to represent object type attributes in C structures or C++ classes. As an example, consider the following object type definition, and its corresponding OTT-generated structure declarations:

```
CREATE TYPE employee AS OBJECT
(   name        VARCHAR2(30),
    empno       NUMBER,
    deptno      NUMBER,
    hiredate    DATE,
    salary$     NUMBER);
```

The OTT utility, assuming that the `CASE` parameter is set to `LOWER` and there are no explicit mappings of type or attribute names, produces the following output:

```
struct employee
{   OCIString * name;
    OCINumber empno;
    OCINumber deptno;
    OCIDate   hiredate;
    OCINumber salary_;
};
typedef struct emp_type emp_type;
struct employee_ind
{   OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd deptno;
    OCIInd hiredate;
    OCIInd salary_;
}
typedef struct employee_ind employee_ind;
```

The datatypes in the structure declarations—`OCIString`, `OCINumber`, `OCIDate`, `OCIInd`—are used here to map the datatypes of the object type attributes. The `number` datatype of the `empno` attribute, maps to the `OCINumber` datatype, for example. These datatypes can also be used as the types of bind and define variables.

See Also:

- Oracle Call Interface Programmer's Guide for further information about the use of datatypes, including object datatypes, in OCI applications

Mapping Object Datatypes to C

This section describes the mappings of object attribute types to C types, as generated by the OTT utility Table 7–1 lists the mappings from types that can be used as attributes to object datatypes that are generated by the OTT utility.

See Also: "OTT Utility Type Mapping Example for C" on page 7-20 includes examples of many of these different mappings.

Table 7–1 C Object Datatype Mappings for Object Type Attributes

Object Attribute Types	C Mapping
`BFILE`	`OCIBFileLocator*`
`BLOB`	`OCIBlobLocator*`
`CHAR(n)`, `CHARACTER(n)`	`OCIString*`
`CLOB`	`OCIClobLocator*`
`DATE`	`OCIDate*`
`DEC`, `DEC(n)`, `DEC(n,n)`	`OCINumber`
`DECIMAL`, `DECIMAL(n)`, `DECIMAL(n,n)`	`OCINumber`
`FLOAT`, `FLOAT(n)`, `DOUBLE PRECISION`	`OCINumber`
`INT`, `INTEGER`, `SMALLINT`	`OCINumber`
`INTERVAL DAY TO SECOND`	`OCIInterval`
`INTERVAL YEAR TO MONTH`	`OCIInterval`
Nested Object Type	`C` name of the nested object type
`NESTED TABLE`	Declared by using `typedef;` equivalent to `OCITable*`
`NUMBER`, `NUMBER(n)`, `NUMBER(n,n)`	`OCINumber`
`NUMERIC`, `NUMERIC(n)`, `NUMERIC(n,n)`	`OCINumber`
`RAW`	`OCIRaw*`

Table 7–1 C Object Datatype Mappings for Object Type Attributes (Cont.)

Object Attribute Types	C Mapping
REAL	OCINumber
REF	Declared by using typedef; equivalent to OCIRef*
TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime*
VARCHAR(n)	OCIString*
VARCHAR2(n)	OCIString*
VARRAY	Declared by using typedef; equivalent to OCIArray*

Note: For REF, VARRAY, and NESTED TABLE types, the OTT utility generates a typedef. The type declared in the typedef is then used as the type of the data member in the structure declaration. For examples, see the next section, "OTT Utility Type Mapping Example for C".

If an object type includes an attribute of a REF or collection type, then a typedef for the REF or collection type is first generated. Then the structure declaration corresponding to the object type is generated. The structure includes an element whose type is a pointer to the REF or collection type.

If an object type includes an attribute whose type is another object type, then the OTT utility first generates the nested object type. It then maps the object type attribute to a nested structure of the type of the nested object type.

The C datatypes to which the OTT utility maps nonobject database attribute types are structures, which, except for OCIDate, are opaque.

Mapping Object Datatypes to C++

This section describes the mappings of object attribute types to C++ types generated by the OTT utility. Table 7–2 lists the mappings from types that can be used as attributes to object datatypes that are generated by the OTT utility.

Table 7–2 C++ Object Datatype Mappings for Object Type Attributes

Object Attribute Types	C++ Mapping
BFILE	Bfile
BLOB	Blob
CHAR(n), CHARACTER(n)	string
CLOB	Clob
DATE	Date
DEC, DEC(n), DEC(n,n)	Number
DECIMAL, DECIMAL(n), DECIMAL(n,n)	Number
FLOAT, FLOAT(n), DOUBLE PRECISION	Number
INT, INTEGER, SMALLINT	Number
INTERVAL DAY TO SECOND	IntervalDS
INTERVAL YEAR TO MONTH	IntervalYM
Nested Object Type	C++ name of the nested object type
NESTED TABLE	vector<attribute_type>
NUMBER, NUMBER(n), NUMBER(n,n)	Number
NUMERIC, NUMERIC(n), NUMERIC(n,n)	Number
RAW	Bytes
REAL	Number
REF	Ref<attribute_type>
TIMESTAMP,TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE	Timestamp
VARCHAR(n)	string
VARCHAR2(n)	string
VARRAY	vector<attribute_type>

OTT Utility Type Mapping Example for C

The example in this section demonstrates the various type mappings created by the OTT utility for a C program.

The example assumes that the following database types are created:

```
CREATE TYPE my_varray AS VARRAY(5) of integer;

CREATE TYPE object_type AS OBJECT
(object_name VARCHAR2(20));

CREATE TYPE other_type AS OBJECT
(object_number NUMBER);

CREATE TYPE my_table AS TABLE OF object_type;

CREATE TYPE many_types AS OBJECT
(   the_varchar     VARCHAR2(30),
    the_char        CHAR(3),
    the_blob        BLOB,
    the_clob        CLOB,
    the_object      object_type,
    another_ref     REF other_type,
    the_ref         REF many_types,
    the_varray      my_varray,
    the_table       my_table,
    the_date        DATE,
    the_num         NUMBER,
    the_raw         RAW(255));
```

The example also assumes that an `INTYPE` file exists and that it includes the following:

```
CASE = LOWER
TYPE many_types
```

The OTT utility would then generate the following C structures:

> **Note:** Comments are provided here to help explain the structures. These comments are not part of the OTT utility output.

```
#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE

#ifndef OCI_ORACLE
#include <oci.h>
```

```
#endif

typedef OCIRef many_types_ref;
typedef OCIRef object_type_ref;
typedef OCIArray my_varray;            /* part of many_types */
typedef OCITable my_table;             /* part of many_types*/
typedef OCIRef other_type_ref;
struct object_type                     /* part of many_types */
{
   OCIString * object_name;
};
typedef struct object_type object_type;

struct object_type_ind                 /*indicator struct for*/
{                                      /*object_types*/
   OCIInd _atomic;
   OCIInd object_name;
};
typedef struct object_type_ind object_type_ind;

struct many_types
{
   OCIString *        the_varchar;
   OCIString *        the_char;
   OCIBlobLocator *   the_blob;
   OCIClobLocator *   the_clob;
   struct object_type the_object;
   other_type_ref *   another_ref;
   many_types_ref *   the_ref;
   my_varray *        the_varray;
   my_table *         the_table;
   OCIDate            the_date;
   OCINumber          the_num;
   OCIRaw *           the_raw;
};
typedef struct many_types many_types;

struct many_types_ind                  /*indicator struct for*/
{                                      /*many_types*/
   OCIInd _atomic;
   OCIInd the_varchar;
   OCIInd the_char;
   OCIInd the_blob;
   OCIInd the_clob;
   struct object_type_ind the_object;   /*nested*/
```

```
   OCIInd another_ref;
   OCIInd the_ref;
   OCIInd the_varray;
   OCIInd the_table;
   OCIInd the_date;
   OCIInd the_num;
   OCIInd the_raw;
};
typedef struct many_types_ind many_types_ind;

#endif
```

Notice that even though only one item was listed for translation in the `INTYPE` file, two object types and two named collection types were translated. This is because the OTT utility parameter TRANSITIVE Parameter, has the default value of `TRUE`. When `TRANSITIVE=TRUE`, the OTT utility automatically translates any types which are used as attributes of a type being translated, in order to complete the translation of the listed type.

This is not the case for types which are only accessed by a pointer or by reference in an object type attribute. For example, although the `many_types` type contains the attribute `another_ref REF other_type`, a declaration of structure `other_type` was not generated.

This example also illustrates how `typedefs` are used to declare `VARRAY`, `NESTED TABLE`, and `REF` types.

In the above example, the `typedefs` occur near the beginning of the file generated by the OTT utility:

```
typedef OCIRef many_types_ref;
typedef OCIRef object_type_ref;
typedef OCIArray my_varray;
typedef OCITable my_table;
typedef OCIRef other_type_ref;
```

In the structure `many_types`, the `VARRAY`, `NESTED TABLE`, and `REF` attributes are declared:

```
struct many_types
{
   ...
   other_type_ref *   another_ref;
   many_types_ref *   the_ref;
```

```
   my_varray *        the_varray;
   my_table *         the_table;
   ...
}
```

OTT Type Mapping Example for C++

The following is an example of the OTT type mappings for C++, given the types created in the example in the previous section, and an `INTYPE` file that includes the following:

```
CASE = LOWER
TYPE many_types
```

The OTT utility generates the following C++ class declarations:

```
#ifndef MYFILENAME_ORACLE
# define MYFILENAME_ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif


/************************************************************/
//  generated declarations for the OBJECT_TYPE object type.
/************************************************************/

class object_type : public oracle::occi::PObject {

protected:

   OCCI_STD_NAMESPACE::string object_name;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   object_type();
```

```
   object_type(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

/************************************************************/
//  generated declarations for the OTHER_TYPE object type.
/************************************************************/

class other_type : public oracle::occi::PObject {

protected:

   oracle::occi::Number object_number;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   other_type();

   other_type(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};
```

```
/************************************************************/
//  generated declarations for the MANY_TYPES object type.
/************************************************************/

class many_types : public oracle::occi::PObject {

protected:

   OCCI_STD_NAMESPACE::string the_varchar;
   OCCI_STD_NAMESPACE::string the_char;
   oracle::occi::Blob the_blob;
   oracle::occi::Clob the_clob;
   object_type * the_object;
   oracle::occi::Ref< other_type > another_ref;
   oracle::occi::Ref< many_types > the_ref;
   OCCI_STD_NAMESPACE::vector< oracle::occi::Number > the_varray;
   OCCI_STD_NAMESPACE::vector< object_type * > the_table;
   oracle::occi::Date the_date;
   oracle::occi::Number the_num;
   oracle::occi::Bytes the_raw;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   many_types();

   many_types(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

#endif
```

For C++, when TRANSITIVE=TRUE, the OTT utility automatically translates any types that are used as attributes of a type being translated, including types that are only being accessed by a pointer or REF in an object type attribute. Even though only the many_types object was specified in the INTYPE file for the C++ example, a class declaration was generated for all the object types, including the other_type object, which was only accessed by a REF in the many_types object.

Overview of the OUTTYPE File

The OUTTYPE file is named on the OTT command line. When the OTT utility generates a C or C++ header file, it also writes the results of the translation into the OUTTYPE file. This file contains an entry for each of the translated types, including its version string and the header file to which its C or C++ representation was written.

The OUTTYPE file from one OTT utility run can be used as the INTYPE file for a subsequent invocation of the OTT utility.

For example, consider the following simple INTYPE file used earlier in this chapter:

```
CASE=LOWER
TYPE employee
    TRANSLATE SALARY$ AS salary
              DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

In this INTYPE file, the programmer specifies the case for OTT-generated C identifiers, and provides a list of types that should be translated. In two of these types, naming conventions are specified.

The following example shows what the OUTTYPE file looks like after running the OTT utility:

```
CASE = LOWER
TYPE EMPLOYEE AS employee
  VERSION = "$8.0"
  HFILE = demo.h
  TRANSLATE SALARY$ AS salary
            DEPTNO AS department
TYPE ADDRESS AS ADDRESS
  VERSION = "$8.0"
```

```
  HFILE = demo.h
TYPE ITEM AS item
  VERSION = "$8.0"
  HFILE = demo.h
TYPE "Person" AS Person
  VERSION = "$8.0"
  HFILE = demo.h
TYPE PURCHASE_ORDER AS p_o
  VERSION = "$8.0"
  HFILE = demo.h
```

When examining the contents of the `OUTTYPE` file, you might discover types listed that were not included in the `INTYPE` file specification. For example, consider the case where the `INTYPE` file only specified that the `person` type was to be translated:

```
CASE = LOWER
TYPE PERSON
```

If the definition of the `person` type includes an attribute of type `address`, then the `OUTTYPE` file includes entries for both `PERSON` and `ADDRESS`. The `person` type cannot be translated completely without first translating `address`.

The OTT utility analyzes the types in the `INTYPE` file for type dependencies before performing the translation, and translates other types as necessary.

Note: As of release 9.0.1, you may indicate whether the OTT utility is to generate required object types that are not specified in the `INTYPE` file. Set `TRANSITIVE=FALSE` so the OTT utility will not to generate required object types. The default is `TRANSITIVE=TRUE`.

See Also: "Invoking the OTT Utility" on page 7-10 for details on these parameters.

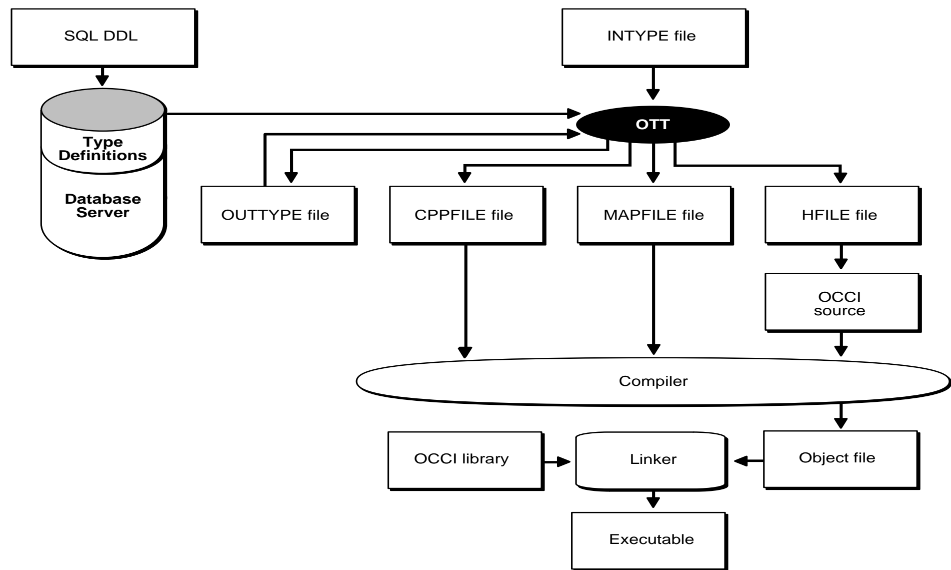
The OTT Utility and OCCI Applications

The OTT utility generates objects and maps SQL datatypes to C++ classes. The OTT utility also implements a few methods called by OCCI when instantiating objects and a function that is called in the OCCI application to register the mappings with the environment. These declarations are stored in a header file that you include

(`#include`) in your OCCI application. The prototype for the function that registers the mappings is written to a separate header file that you also include in your OCCI application.The method implementations are stored in a C++ source code file (with extension `.cpp`) that is linked with the OCCI application. The function that registers the mappings is stored in a separate C++ (`.cpp`) file that is also linked with the application.

Figure 7–1 shows the steps involved in using the OTT utility with OCCI. These steps are described following the figure.

Figure 7–1 The OTT Utility with OCCI



1. Create the type definitions in the database by using the SQL DLL.
2. Create the `INTYPE` file that contains the database types to be translated by the OTT utility.
3. Specify that C++ should be generated and invoke the OTT utility.

   The OTT utility then generates the following files:

- A header file (with the extension .h) that contains C++ class representations of object types. The filename is specified on the OTT command line by the HFILE parameter.
- A header file containing the prototype of the function (MAPFUNC) that registers the mappings.
- A C++ source file (with the extension .cpp) that contains the static methods to be called by OCCI while instantiating the objects. Do not call these methods directly from your OCCI application. The filename is specified on the OTT command line by the CPPFILE parameter.
- A file that contains the function used to register the mappings with the environment (with the extension .cpp). The filename is specified on the OTT command line by the MAPFILE parameter.
- A file (the OUTTYPE file) that contains an entry for each of the translated types, including the version string and the file into which it is written. The filename is specified on the OTT command line by the OUTTYPE parameter.

4. Write the OCCI application and include the header files created by the OTT utility in the OCCI source code file.

   The application declares an environment and calls the function MAPFUNC to register the mappings.

5. Compile the OCCI application to create the OCCI object code, and link the object code with the OCCI libraries to create the program executable.

OTT Utility Parameters for C++

To generate C++ using the OTT utility, the CODE parameter must be set to CODE=CPP. Once CODE=CPP is specified, you are required to specify the CPPFILE and MAPFILE parameters to define the filenames for the method implementation file and the mappings registration function file. The name of the mapping function is derived by the OTT utility from the MAPFILE or you may specify the name with the MAPFUNC parameter. ATTRACCESS is also an optional parameter that can be specified to change the generated code.

The following parameters are specific to C++ only and control the generation of C++ classes:

- CPPFILE
- MAPFILE
- MAPFUNC

- ATTRACCESS

See Also: "OTT Utility Parameters" on page 7-90 for details on these parameters.

OTT-Generated C++ Classes

When the OTT utility generates a C++ class from a database object type, the class declaration contains one element corresponding to each attribute of the object type. The datatypes of the attribute are mapped to types that are used in Oracle object datatypes, as defined in Table 7–2 on page 7-20.

For each class, two new operators, a getSQLTypeName method, two constructors, two readSQL methods and two writeSQL methods are generated. The getSQLTypeName method, the constructor, the readSQL and writeSQL methods are called by OCCI while unmarshalling and marshalling the object data.

By default, the OTT-generated C++ class for an object type is derived from the PObject class and so the generated constructor in the class also derives from the PObject class. For inherited database types, the class is derived from the parent type class as is the generated constructor and only the elements corresponding to attributes not already in the parent class are included.

Class declarations that include the elements corresponding to the database type attributes and the method declarations are included in the header file generated by the OTT utility. The method implementations are included in the CPPFILE file generated by the OTT utility.

The following is an example of the C++ classes generated by the OTT utility as a result of this series of steps:

1. Define the types:

```
CREATE TYPE FULL_NAME AS OBJECT (first_name CHAR(20), last_name CHAR(20));
CREATE TYPE ADDRESS AS OBJECT (state CHAR(20), zip CHAR(20));
CREATE TYPE ADDRESS_TAB AS VARRAY(3) of REF ADDRESS;
CREATE TYPE PERSON AS OBJECT (id NUMBER, name FULL_NAME, curr_addr REF ADDRESS,
   prev_addr_l ADDRESS_TAB) NOT FINAL;
CREATE TYPE STUDENT UNDER PERSON (school_name CHAR(20));
```

2. Provide an INTYPE file:

```
CASE = SAME
MAPFILE = RegisterMappings_3.cpp
TYPE FULL_NAME AS FullName
  TRANSLATE first_name as FirstName
```

```
            last_name as LastName
TYPE ADDRESS
TYPE PERSON
TYPE STUDENT
```

3. Invoke the OTT utility:

```
ott userid=scott/tiger intype=demoin_3.typ outype=demoout_3.typ code=cpp
hfile=demo_3.h cppfile=demo_3.cpp
```

This example produces a header file named `demo_3.h`, a C++ source file named `demo_3.cpp`, and an `OUTTYPE` file named `demoout_3.typ`. These files generated by the OTT utility are displayed in the following sections:

- Example of a Header File Generated by the OTT Utility: demo_3.h
- Example of a C++ Source File Generated by the OTT Utility: demo_3.cpp
- Example of an OUTTYPE File Generated by the OTT Utility: demoout_3.typ

Example of a Header File Generated by the OTT Utility: demo_3.h

This section contains the header file generated by the OTT utility (named `demo_3.h`) based on information contained in the previous section.

```
#ifndef DEMO_3_ORACLE
# define DEMO_3_ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif


/************************************************************/
//  generated declarations for the FULL_NAME object type.
/************************************************************/

class FullName : public oracle::occi::PObject {

protected:

   OCCI_STD_NAMESPACE::string FirstName;
   OCCI_STD_NAMESPACE::string LastName;

public:
```

```
   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   FullName();

   FullName(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

/************************************************************/
//  generated declarations for the ADDRESS object type.
/************************************************************/

class ADDRESS : public oracle::occi::PObject {

protected:

   OCCI_STD_NAMESPACE::string STATE;
   OCCI_STD_NAMESPACE::string ZIP;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   ADDRESS();

   ADDRESS(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };
```

```
   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

/************************************************************/
//  generated declarations for the PERSON object type.
/************************************************************/

class PERSON : public oracle::occi::PObject {

protected:

   oracle::occi::Number ID;
   FullName * NAME;
   oracle::occi::Ref< ADDRESS > CURR_ADDR;
   OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< ADDRESS > > PREV_ADDR_L;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   PERSON();

   PERSON(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};
```

```
/************************************************************/
//  generated declarations for the STUDENT object type.
/************************************************************/

class STUDENT : public PERSON {

protected:

   OCCI_STD_NAMESPACE::string SCHOOL_NAME;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   STUDENT();

   STUDENT(void *ctxOCCI_) : PERSON (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

#endif
```

Example of a C++ Source File Generated by the OTT Utility: demo_3.cpp

This section contains the C++ source file generated by the OTT utility (named `demo_3.cpp`) based on information contained in the previous section.

```
#ifndef DEMO_3_ORACLE
# include "demo_3.h"
#endif
```

```
/*****************************************************************/
//  generated method implementations for the FULL_NAME object type.
/*****************************************************************/

void *FullName::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *FullName::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.FULL_NAME");
}

OCCI_STD_NAMESPACE::string FullName::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.FULL_NAME");
}

FullName::FullName()
{
}

void *FullName::readSQL(void *ctxOCCI_)
{
  FullName *objOCCI_ = new FullName(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
```

```
  return (void *)objOCCI_;
}

void FullName::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   FirstName = streamOCCI_.getString();
   LastName = streamOCCI_.getString();
}

void FullName::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  FullName *objOCCI_ = (FullName *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void FullName::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setString(FirstName);
   streamOCCI_.setString(LastName);
}

/*****************************************************************/
//  generated method implementations for the ADDRESS object type.
/*****************************************************************/

void *ADDRESS::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *ADDRESS::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
```

```
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.ADDRESS");
}

OCCI_STD_NAMESPACE::string ADDRESS::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.ADDRESS");
}

ADDRESS::ADDRESS()
{
}

void *ADDRESS::readSQL(void *ctxOCCI_)
{
  ADDRESS *objOCCI_ = new ADDRESS(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void ADDRESS::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   STATE = streamOCCI_.getString();
   ZIP = streamOCCI_.getString();
}

void ADDRESS::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  ADDRESS *objOCCI_ = (ADDRESS *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);
```

```
  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void ADDRESS::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setString(STATE);
   streamOCCI_.setString(ZIP);
}

/*****************************************************************/
//  generated method implementations for the PERSON object type.
/*****************************************************************/

void *PERSON::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *PERSON::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.PERSON");
}

OCCI_STD_NAMESPACE::string PERSON::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.PERSON");
}

PERSON::PERSON()
{
   NAME = (FullName *) 0;
```

```
}

void *PERSON::readSQL(void *ctxOCCI_)
{
  PERSON *objOCCI_ = new PERSON(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void PERSON::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   ID = streamOCCI_.getNumber();
   NAME = (FullName *) streamOCCI_.getObject();
   CURR_ADDR = streamOCCI_.getRef();
   getVector(streamOCCI_, PREV_ADDR_L);
}

void PERSON::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  PERSON *objOCCI_ = (PERSON *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
```

```
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void PERSON::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setNumber(ID);
   streamOCCI_.setObject(NAME);
   streamOCCI_.setRef(CURR_ADDR);
   setVector(streamOCCI_, PREV_ADDR_L);
}

/*****************************************************************/
//  generated method implementations for the STUDENT object type.
/*****************************************************************/

void *STUDENT::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *STUDENT::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.STUDENT");
}

OCCI_STD_NAMESPACE::string STUDENT::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.STUDENT");
}

STUDENT::STUDENT()
{
}

void *STUDENT::readSQL(void *ctxOCCI_)
{
  STUDENT *objOCCI_ = new STUDENT(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
```

```
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void STUDENT::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   PERSON::readSQL(streamOCCI_);
   SCHOOL_NAME = streamOCCI_.getString();
}

void STUDENT::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  STUDENT *objOCCI_ = (STUDENT *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void STUDENT::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   PERSON::writeSQL(streamOCCI_);
   streamOCCI_.setString(SCHOOL_NAME);
}
```

Example of an OUTTYPE File Generated by the OTT Utility: demoout_3.typ

This section contains the OUTTYPE file generated by the OTT utility (named `demoout_3.typ`) based on information contained in the previous section:

```
CASE = SAME
MAPFILE = RegisterMappings_3.cpp
MAPFUNC = RegisterMappings

TYPE SCOTT.FULL_NAME AS FullName
  VERSION = "$8.0"
  HFILE = demo_3.h
TRANSLATE FIRST_NAME AS FirstName
          LAST_NAME AS LastName

TYPE SCOTT.ADDRESS AS ADDRESS
  VERSION = "$8.0"
  HFILE = demo_3.h

TYPE SCOTT.PERSON AS PERSON
  VERSION = "$8.0"
  HFILE = demo_3.h

TYPE SCOTT.STUDENT AS STUDENT
  VERSION = "$8.0"
  HFILE = demo_3.h
```

Example with ATTRACCESS=PRIVATE

To demonstrate the difference in generated code when `ATTRACCESS=PRIVATE`, consider an `INTYPE` file that contains:

```
CASE = SAME
TYPE PERSON
```

The OTT utility generates the following header file:

```
#ifndef DEMO_4_ORACLE
# define DEMO_4 _ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif


/************************************************************/
```

```
//  generated declarations for the FULL_NAME object type.
/************************************************************/

class FULL_NAME : public oracle::occi::PObject {

private:

   OCCI_STD_NAMESPACE::string FIRST_NAME;
   OCCI_STD_NAMESPACE::string LAST_NAME;

public:

   OCCI_STD_NAMESPACE::string getFirst_name() const;

   void setFirst_name(const OCCI_STD_NAMESPACE::string &value);

   OCCI_STD_NAMESPACE::string getLast_name() const;

   void setLast_name(const OCCI_STD_NAMESPACE::string &value);

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   FULL_NAME();

   FULL_NAME(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

/************************************************************/
//  generated declarations for the ADDRESS object type.
/************************************************************/
```

```
class ADDRESS : public oracle::occi::PObject {

private:

   OCCI_STD_NAMESPACE::string STATE;
   OCCI_STD_NAMESPACE::string ZIP;

public:

   OCCI_STD_NAMESPACE::string getState() const;

   void setState(const OCCI_STD_NAMESPACE::string &value);

   OCCI_STD_NAMESPACE::string getZip() const;

   void setZip(const OCCI_STD_NAMESPACE::string &value);

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   ADDRESS();

   ADDRESS(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

/************************************************************/
//  generated declarations for the PERSON object type.
/************************************************************/

class PERSON : public oracle::occi::PObject {

private:
```

```
   oracle::occi::Number ID;
   FULL_NAME * NAME;
   oracle::occi::Ref< ADDRESS > CURR_ADDR;
   OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< ADDRESS > > PREV_ADDR_L;

public:

   oracle::occi::Number getId() const;

   void setId(const oracle::occi::Number &value);

   FULL_NAME * getName() const;

   void setName(FULL_NAME * value);

   oracle::occi::Ref< ADDRESS > getCurr_addr() const;

   void setCurr_addr(const oracle::occi::Ref< ADDRESS > &value);

   OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< ADDRESS > >& getPrev_addr_l();

   const OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< ADDRESS > >& getPrev_
addr_l() const;

   void setPrev_addr_l(const OCCI_STD_NAMESPACE::vector< oracle::occi::Ref<
ADDRESS > > &value);

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   PERSON();

   PERSON(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);
```

```
   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

#endif
```

Since `ATTRACCESS=PRIVATE`, the access given to the attributes is `private` and the accessor (get*xxx*) and the mutator (set*xxx*) methods are generated for each of the attributes.

Map Registry Function

One function to register the mappings with the environment is generated by the OTT utility. The function contains the mappings for all the types translated by the invocation of the OTT utility. The function name is either specified in the `MAPFUNC` parameter or, if that parameter is not specified, derived from `MAPFILE` parameter. The only argument to the function is the pointer to `Environment`.

The function uses the provided `Environment` to get `Map` and then registers the mapping of each translated type.

Given the database type and `INTYPE` file listed in the previous section, and specifying `MAPFILE=RegisterMappings_3.cpp`, the map registering function generated takes the following form:

```
#ifndef REGISTERMAPPINGS_3_ORACLE
# include "registermappings_3.h"
#endif

void RegisterMappings_3(oracle::occi::Environment* envOCCI_)
{
  oracle::occi::Map *mapOCCI_ = envOCCI_->getMap();
  mapOCCI_->put("SCOTT.FULL_NAME", FullName::readSQL, FullName::writeSQL);
  mapOCCI_->put("SCOTT.ADDRESS", ADDRESS::readSQL, ADDRESS::writeSQL);
  mapOCCI_->put("SCOTT.PERSON", PERSON::readSQL, PERSON::writeSQL);
  mapOCCI_->put("SCOTT.STUDENT", STUDENT::readSQL, STUDENT::writeSQL);
}
```

The prototype of the register mapping function is written to a corresponding header file, `RegisterMapping.h`, and looks like the following:

```
#ifndef REGISTERMAPPINGS_3_ORACLE
# define REGISTERMAPPINGS_3_ORACLE
```

```
#ifndef OCCI_ORACLE
# include <occi.h>
#endif

#ifndef DEMO_3_ORACLE
# include "demo_3.h"
#endif

void RegisterMappings_3(oracle::occi::Environment* envOCCI_);

#endif
```

Extending OTT C++ Classes

To enhance the functionality of a class generated by the OTT utility, you can derive new classes. You can also add methods to a class, but Oracle does not recommend doing so due to an inherent risk.

Caution: Oracle recommends that you do not add methods to an OTT-generated class. If you ever run the OTT utility again to regenerate the class, the methods you added previously are overwritten. This is true even if you direct the OTT utility output to a separate file because you will have to merge changes into the main file.

For an example of deriving a new class from an OTT-generated class, assume you want to generate the class `CAddress` from the SQL object type `ADDRESS`. Assume also that you want to write a class MyAddress to represent `ADDRESS` objects. The `MyAddress` class can be derived from `CAddress`.

To perform this, the OTT utility must alter the code it generates:

- By using the `MyAddress` class instead of the `CAddress` class to represent attributes whose database type is `ADDRESS`
- By using the `MyAddress` class instead of the `CAddress` class to represent vector and `REF` elements whose database type is `ADDRESS`
- By using the `MyAddress` class instead of the `CAddress` class as the base class for database object types that are inherited from `ADDRESS`. Even though a

derived class is a subtype of `MyAddress`, the `readSQL` and `writeSQL` methods called are those of the `CAddress` class.

> **Note:** When a class is both extended and used as a base class for another generated class, the *inheriting* type class and the *inherited* type class must be generated in separate files.

To use the OTT utility to generate the `CAddress` class (that you derive the `MyAddress` class from), the following clause must be specified in the `TYPE` statement:

```
TYPE ADDRESS GENERATE CAdress AS MyAddress
```

Example for Extending OTT Classes

Given the database types `FULL_NAME`, `ADDRESS`, `PERSON`, and `STUDENT` as they were created before and changing the `INTYPE` file to include the `GENERATE ... AS` clause:

```
CASE = SAME
MAPFILE = RegisterMappings_5.cpp

TYPE FULL_NAME GENERATE CFullName AS MyFullName
  TRANSLATE first_name as FirstName
            last_name as LastName

TYPE ADDRESS GENERATE CAddress AS MyAddress
TYPE PERSON GENERATE CPerson AS MyPerson
TYPE STUDENT GENERATE CStudent AS MyStudent
```

The following C++ source file (with the extension `.cpp`) is generated by the OTT utility:

```
#ifndef MYFILENAME_ORACLE
# define MYFILENAME_ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif


/************************************************************/
//  generated declarations for the FULL_NAME object type.
```

```
/************************************************************/

class CFullName : public oracle::occi::PObject {

protected:

   OCCI_STD_NAMESPACE::string FirstName;
   OCCI_STD_NAMESPACE::string LastName;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   CFullName();

   CFullName(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

/************************************************************/
//  generated declarations for the ADDRESS object type.
/************************************************************/

class CAddress : public oracle::occi::PObject {

protected:

   OCCI_STD_NAMESPACE::string STATE;
   OCCI_STD_NAMESPACE::string ZIP;

public:
```

```
   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   CAddress();

   CAddress(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

/************************************************************/
//  generated declarations for the PERSON object type.
//
//  Note the type name for the "name" attribute is MyFullName
//  and not CFullName, the "curr-addr" attribute is Ref< MyAddress >
//  and not Ref< CAddress >, and the "prev_addr_l" attribute is
//  vector< Ref< MyAddress > >.
/************************************************************/

class CPerson : public oracle::occi::PObject {

protected:

   oracle::occi::Number ID;
   MyFullName * NAME;
   oracle::occi::Ref< MyAddress > CURR_ADDR;
   OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > > PREV_ADDR_L;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);
```

```
   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   CPerson();

   CPerson(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

/************************************************************/
//  generated declarations for the STUDENT object type.
//
//  Note the parent class for CStudent is MyPerson and not
//  CPerson
/************************************************************/

class CStudent : public MyPerson {

protected:

   OCCI_STD_NAMESPACE::string SCHOOL_NAME;

public:

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   CStudent();

   CStudent(void *ctxOCCI_) : MyPerson (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);
```

```
   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

#endif
```

The method implementations are as follows:

```
#ifndef MYFILENAME_ORACLE
# include "myfilename.h"
#endif


/*****************************************************************/
//  generated method implementations for the FULL_NAME object type.
/*****************************************************************/

void *CFullName::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CFullName::operator new(size_t size, const oracle::occi::Connection *
sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.FULL_NAME");
}

OCCI_STD_NAMESPACE::string CFullName::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.FULL_NAME");
}

CFullName::CFullName()
{
}

void *CFullName::readSQL(void *ctxOCCI_)
```

```
{
  CFullName *objOCCI_ = new CFullName(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CFullName::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   FirstName = streamOCCI_.getString();
   LastName = streamOCCI_.getString();
}

void CFullName::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CFullName *objOCCI_ = (CFullName *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}
```

```
void CFullName::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setString(FirstName);
   streamOCCI_.setString(LastName);
}

/*****************************************************************/
//  generated method implementations for the ADDRESS object type.
/*****************************************************************/

void *CAddress::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CAddress::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.ADDRESS");
}

OCCI_STD_NAMESPACE::string CAddress::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.ADDRESS");
}

CAddress::CAddress()
{
}

void *CAddress::readSQL(void *ctxOCCI_)
{
  CAddress *objOCCI_ = new CAddress(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
```

```
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CAddress::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   STATE = streamOCCI_.getString();
   ZIP = streamOCCI_.getString();
}

void CAddress::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CAddress *objOCCI_ = (CAddress *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CAddress::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setString(STATE);
   streamOCCI_.setString(ZIP);
}

/*****************************************************************/
//  generated method implementations for the PERSON object type.
//
//  Note the type used in the casting in the readSQL method is
//  MyFullName and not CFullName.
/*****************************************************************/
```

```
void *CPerson::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CPerson::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.PERSON");
}

OCCI_STD_NAMESPACE::string CPerson::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.PERSON");
}

CPerson::CPerson()
{
   NAME = (MyFullName *) 0;
}

void *CPerson::readSQL(void *ctxOCCI_)
{
  CPerson *objOCCI_ = new CPerson(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CPerson::readSQL(oracle::occi::AnyData& streamOCCI_)
{
```

```
   ID = streamOCCI_.getNumber();
   NAME = (MyFullName *) streamOCCI_.getObject();
   CURR_ADDR = streamOCCI_.getRef();
   getVector(streamOCCI_, PREV_ADDR_L);
}

void CPerson::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CPerson *objOCCI_ = (CPerson *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CPerson::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setNumber(ID);
   streamOCCI_.setObject(NAME);
   streamOCCI_.setRef(CURR_ADDR);
   setVector(streamOCCI_, PREV_ADDR_L);
}

/*****************************************************************/
//  generated method implementations for the STUDENT object type.
//
//  Note even though CStudent derives from MyPerson, the readSQL
//  and writeSQL methods called are those of CPerson.
/*****************************************************************/

void *CStudent::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}
```

```
void *CStudent::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.STUDENT");
}

OCCI_STD_NAMESPACE::string CStudent::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.STUDENT");
}

CStudent::CStudent()
{
}

void *CStudent::readSQL(void *ctxOCCI_)
{
  CStudent *objOCCI_ = new CStudent(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CStudent::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   CPerson::readSQL(streamOCCI_);
   SCHOOL_NAME = streamOCCI_.getString();
}

void CStudent::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
```

```
  CStudent *objOCCI_ = (CStudent *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CStudent::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   CPerson::writeSQL(streamOCCI_);
   streamOCCI_.setString(SCHOOL_NAME);
}
```

Example OCCI Application

This OCCI application example extends the OTT-generated C++ classes and translates inherited object types. Each class in this application contains a constructor to initialize class objects and a method to display the values assigned to the attributes of the object. The `MyPerson` class also has a method to change the `curr_addr` attribute. All the classes here are derived from the generated classes.

Create the needed types and tables for the OCCI application as illustrated in the following code example:

```
connect scott/tiger
create type full_name as object (first_name char(20), last_name char(20));
create type address as object (state char(20), zip char(20));
create type address_tab as varray(3) of ref address;
create type person as object (id number, name full_name,
    curr_addr ref address, prev_addr_l address_tab) not final;
create type student under person (school_name char(20));

/* tables needed in the user-written occi application */
create table addr_tab of address;
```

```
create table person_tab of person;
create table student_tab of student;
```

The `INTYPE` file provided to the OTT utility contains the following information:

```
CASE = SAME
MAPFILE = registerMappings.cpp

TYPE FULL_NAME GENERATE CFullName AS MyFullName
HFILE=cfullname.h
CPPFILE=cfullname.cpp
  TRANSLATE first_name as FirstName
            last_name as LastName

TYPE ADDRESS GENERATE CAddress AS MyAddress
HFILE=caddress.h
CPPFILE=caddress.cpp

TYPE PERSON GENERATE CPerson AS MyPerson
HFILE=cperson.h
CPPFILE=cperson.cpp

TYPE STUDENT GENERATE CStudent AS MyStudent
HFILE=cstudent.h
CPPFILE=cstudent.cpp
```

Note: `PERSON` and `STUDENT` must be generated in separate files because `PERSON` is an extended class and it is the base class for `STUDENT`.

To invoke the OTT utility, use the following command line statement:

```
ott userid=scott/tiger code=cpp attraccess=private intype=demoin.typ
outtype=demoout.typ
```

Note: `attraccess=private` is specified because accessors and mutators are used to access the attributes.

The files generated by the OTT utility for this example are shown in the following sections:

- Example of the Declarations for Object Type FULL_NAME: cfullname.h
- Example of the Declarations for Object Type ADDRESS: caddress.h
- Example of Declarations for Object Type PERSON: cperson.h
- Example for Declarations for Object Type STUDENT: cstudent.h
- Example of C++ Source File for Object Type FULLNAME: cfullname.cpp
- Example of C++ Source File for Object Type ADDRESS: caddress.cpp
- Example of C++ Source File for Object Type PERSON: cperson.cpp
- Example of C++ Source File for Object Type STUDENT: cstudent.cpp
- Example of Register Mappings Header File: registerMappings.h
- Example of C++ Source File for Register Mappings: registerMappings.cpp
- Example of User-Written Extension File: myfullname.h
- Example of User-Written Extension File: myaddress.h
- Example of User-Written Extension File: myperson.h
- Example of User-Written Extension File: mystudent.h
- Example of User-Written Extension File: mydemo.cpp
- Output Generated by Example OTT Application

Example of the Declarations for Object Type FULL_NAME: cfullname.h

```
#ifndef CFULLNAME_ORACLE
# define CFULLNAME_ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif


/************************************************************/
//  generated declarations for the FULL_NAME object type.
/************************************************************/

class CFullName : public oracle::occi::PObject {
```

```
private:

   OCCI_STD_NAMESPACE::string FirstName;
   OCCI_STD_NAMESPACE::string LastName;

public:

   OCCI_STD_NAMESPACE::string getFirstname() const;

   void setFirstname(const OCCI_STD_NAMESPACE::string &value);

   OCCI_STD_NAMESPACE::string getLastname() const;

   void setLastname(const OCCI_STD_NAMESPACE::string &value);

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   CFullName();

   CFullName(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

#endif
```

Example of the Declarations for Object Type ADDRESS: caddress.h

```
#ifndef CADDRESS_ORACLE
# define CADDRESS_ORACLE

#ifndef OCCI_ORACLE
```

```
# include <occi.h>
#endif


/************************************************************/
//  generated declarations for the ADDRESS object type.
/************************************************************/

class CAddress : public oracle::occi::PObject {

private:

   OCCI_STD_NAMESPACE::string STATE;
   OCCI_STD_NAMESPACE::string ZIP;

public:

   OCCI_STD_NAMESPACE::string getState() const;

   void setState(const OCCI_STD_NAMESPACE::string &value);

   OCCI_STD_NAMESPACE::string getZip() const;

   void setZip(const OCCI_STD_NAMESPACE::string &value);

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   CAddress();

   CAddress(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};
```

```
#endif
```

Example of Declarations for Object Type PERSON: cperson.h

```
#ifndef CPERSON_ORACLE
# define CPERSON_ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif

#ifndef MYFULLNAME_ORACLE
# include "myfullname.h"
#endif

#ifndef MYADDRESS_ORACLE
# include "myaddress.h"
#endif


/************************************************************/
//  generated declarations for the PERSON object type.
/************************************************************/

class CPerson : public oracle::occi::PObject {

private:

   oracle::occi::Number ID;
   MyFullName * NAME;
   oracle::occi::Ref< MyAddress > CURR_ADDR;
   OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > > PREV_ADDR_L;

public:

   oracle::occi::Number getId() const;

   void setId(const oracle::occi::Number &value);

   MyFullName * getName() const;

   void setName(MyFullName * value);
```

```
   oracle::occi::Ref< MyAddress > getCurr_addr() const;

   void setCurr_addr(const oracle::occi::Ref< MyAddress > &value);

   const OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > >& getPrev_
addr_l() const;

   OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > >& getPrev_addr_
l();

   void setPrev_addr_l(const OCCI_STD_NAMESPACE::vector< oracle::occi::Ref<
MyAddress > > &value);

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   CPerson();

   CPerson(void *ctxOCCI_) : oracle::occi::PObject (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

#endif
```

Example for Declarations for Object Type STUDENT: cstudent.h

```
#ifndef CSTUDENT_ORACLE
# define CSTUDENT_ORACLE

#ifndef OCCI_ORACLE
# include <occi.h>
#endif
```

```
#ifndef MYPERSON_ORACLE
# include "myperson.h"
#endif

/************************************************************/
//  generated declarations for the STUDENT object type.
/************************************************************/

class CStudent : public MyPerson {

private:

   OCCI_STD_NAMESPACE::string SCHOOL_NAME;

public:

   OCCI_STD_NAMESPACE::string getSchool_name() const;

   void setSchool_name(const OCCI_STD_NAMESPACE::string &value);

   void *operator new(size_t size);

   void *operator new(size_t size, const oracle::occi::Connection * sess,
      const OCCI_STD_NAMESPACE::string& table);

   OCCI_STD_NAMESPACE::string getSQLTypeName() const;

   CStudent();

   CStudent(void *ctxOCCI_) : MyPerson (ctxOCCI_) { };

   static void *readSQL(void *ctxOCCI_);

   virtual void readSQL(oracle::occi::AnyData& streamOCCI_);

   static void writeSQL(void *objOCCI_, void *ctxOCCI_);

   virtual void writeSQL(oracle::occi::AnyData& streamOCCI_);

};

#endif
```

Example of C++ Source File for Object Type FULLNAME: cfullname.cpp

```
#ifndef CFULLNAME_ORACLE
# include "cfullname.h"
#endif


/*****************************************************************/
//  generated method implementations for the FULL_NAME object type.
/*****************************************************************/

OCCI_STD_NAMESPACE::string CFullName::getFirstname() const
{
  return FirstName;
}

void CFullName::setFirstname(const OCCI_STD_NAMESPACE::string &value)
{
  FirstName = value;
}

OCCI_STD_NAMESPACE::string CFullName::getLastname() const
{
  return LastName;
}

void CFullName::setLastname(const OCCI_STD_NAMESPACE::string &value)
{
  LastName = value;
}

void *CFullName::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CFullName::operator new(size_t size, const oracle::occi::Connection *
sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.FULL_NAME");
}

OCCI_STD_NAMESPACE::string CFullName::getSQLTypeName() const
{
```

```
  return OCCI_STD_NAMESPACE::string("SCOTT.FULL_NAME");
}

CFullName::CFullName()
{
}

void *CFullName::readSQL(void *ctxOCCI_)
{
  CFullName *objOCCI_ = new CFullName(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CFullName::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   FirstName = streamOCCI_.getString();
   LastName = streamOCCI_.getString();
}

void CFullName::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CFullName *objOCCI_ = (CFullName *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
```

```
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CFullName::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setString(FirstName);
   streamOCCI_.setString(LastName);
}
```

Example of C++ Source File for Object Type ADDRESS: caddress.cpp

```
#ifndef CADDRESS_ORACLE
# include "caddress.h"
#endif


/*****************************************************************/
//  generated method implementations for the ADDRESS object type.
/*****************************************************************/

OCCI_STD_NAMESPACE::string CAddress::getState() const
{
  return STATE;
}

void CAddress::setState(const OCCI_STD_NAMESPACE::string &value)
{
  STATE = value;
}

OCCI_STD_NAMESPACE::string CAddress::getZip() const
{
  return ZIP;
}

void CAddress::setZip(const OCCI_STD_NAMESPACE::string &value)
{
  ZIP = value;
}
```

```
void *CAddress::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CAddress::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.ADDRESS");
}

OCCI_STD_NAMESPACE::string CAddress::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.ADDRESS");
}

CAddress::CAddress()
{
}

void *CAddress::readSQL(void *ctxOCCI_)
{
  CAddress *objOCCI_ = new CAddress(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CAddress::readSQL(oracle::occi::AnyData& streamOCCI_)
{
```

```
   STATE = streamOCCI_.getString();
   ZIP = streamOCCI_.getString();
}

void CAddress::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CAddress *objOCCI_ = (CAddress *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CAddress::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setString(STATE);
   streamOCCI_.setString(ZIP);
}
```

Example of C++ Source File for Object Type PERSON: cperson.cpp

```
#ifndef CPERSON_ORACLE
# include "cperson.h"
#endif


/*****************************************************************/
//  generated method implementations for the PERSON object type.
/*****************************************************************/

oracle::occi::Number CPerson::getId() const
{
  return ID;
}
```

```
void CPerson::setId(const oracle::occi::Number &value)
{
  ID = value;
}

MyFullName * CPerson::getName() const
{
  return NAME;
}

void CPerson::setName(MyFullName * value)
{
  NAME = value;
}

oracle::occi::Ref< MyAddress > CPerson::getCurr_addr() const
{
  return CURR_ADDR;
}

void CPerson::setCurr_addr(const oracle::occi::Ref< MyAddress > &value)
{
  CURR_ADDR = value;
}


const OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > >&
CPerson::getPrev_addr_l() const
{
  return PREV_ADDR_L;
}


OCCI_STD_NAMESPACE::vector< oracle::occi::Ref< MyAddress > >&  CPerson::getPrev_
addr_l()
{
  return PREV_ADDR_L;
}

void CPerson::setPrev_addr_l(const OCCI_STD_NAMESPACE::vector<
oracle::occi::Ref< MyAddress > > &value)
{
  PREV_ADDR_L = value;
}
```

```
void *CPerson::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CPerson::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.PERSON");
}

OCCI_STD_NAMESPACE::string CPerson::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.PERSON");
}

CPerson::CPerson()
{
   NAME = (MyFullName *) 0;
}

void *CPerson::readSQL(void *ctxOCCI_)
{
  CPerson *objOCCI_ = new CPerson(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CPerson::readSQL(oracle::occi::AnyData& streamOCCI_)
```

```
{
   ID = streamOCCI_.getNumber();
   NAME = (MyFullName *) streamOCCI_.getObject();
   CURR_ADDR = streamOCCI_.getRef();
   getVector(streamOCCI_, PREV_ADDR_L);
}

void CPerson::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CPerson *objOCCI_ = (CPerson *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CPerson::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   streamOCCI_.setNumber(ID);
   streamOCCI_.setObject(NAME);
   streamOCCI_.setRef(CURR_ADDR);
   setVector(streamOCCI_, PREV_ADDR_L);
}
```

Example of C++ Source File for Object Type STUDENT: cstudent.cpp

```
#ifndef CSTUDENT_ORACLE
# include "cstudent.h"
#endif


/*****************************************************************/
//  generated method implementations for the STUDENT object type.
/*****************************************************************/
```

```
OCCI_STD_NAMESPACE::string CStudent::getSchool_name() const
{
  return SCHOOL_NAME;
}

void CStudent::setSchool_name(const OCCI_STD_NAMESPACE::string &value)
{
  SCHOOL_NAME = value;
}

void *CStudent::operator new(size_t size)
{
  return oracle::occi::PObject::operator new(size);
}

void *CStudent::operator new(size_t size, const oracle::occi::Connection * sess,
  const OCCI_STD_NAMESPACE::string& table)
{
  return oracle::occi::PObject::operator new(size, sess, table,
            (char *) "SCOTT.STUDENT");
}

OCCI_STD_NAMESPACE::string CStudent::getSQLTypeName() const
{
  return OCCI_STD_NAMESPACE::string("SCOTT.STUDENT");
}

CStudent::CStudent()
{
}

void *CStudent::readSQL(void *ctxOCCI_)
{
  CStudent *objOCCI_ = new CStudent(ctxOCCI_);
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (streamOCCI_.isNull())
      objOCCI_->setNull();
    else
      objOCCI_->readSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
```

```
    delete objOCCI_;
    excep.setErrorCtx(ctxOCCI_);
    return (void *)NULL;
  }
  return (void *)objOCCI_;
}

void CStudent::readSQL(oracle::occi::AnyData& streamOCCI_)
{
   CPerson::readSQL(streamOCCI_);
   SCHOOL_NAME = streamOCCI_.getString();
}

void CStudent::writeSQL(void *objectOCCI_, void *ctxOCCI_)
{
  CStudent *objOCCI_ = (CStudent *) objectOCCI_;
  oracle::occi::AnyData streamOCCI_(ctxOCCI_);

  try
  {
    if (objOCCI_->isNull())
      streamOCCI_.setNull();
    else
      objOCCI_->writeSQL(streamOCCI_);
  }
  catch (oracle::occi::SQLException& excep)
  {
    excep.setErrorCtx(ctxOCCI_);
  }
  return;
}

void CStudent::writeSQL(oracle::occi::AnyData& streamOCCI_)
{
   CPerson::writeSQL(streamOCCI_);
   streamOCCI_.setString(SCHOOL_NAME);
}
```

Example of Register Mappings Header File: registerMappings.h

```
#ifndef REGISTERMAPPINGS_ORACLE
# define REGISTERMAPPINGS_ORACLE

#ifndef OCCI_ORACLE
```

```
# include <occi.h>
#endif

#ifndef CFULLNAME_ORACLE
# include "cfullname.h"
#endif

#ifndef CADDRESS_ORACLE
# include "caddress.h"
#endif

#ifndef CPERSON_ORACLE
# include "cperson.h"
#endif

#ifndef CSTUDENT_ORACLE
# include "cstudent.h"
#endif

void registerMappings(oracle::occi::Environment* envOCCI_);

#endif
```

Example of C++ Source File for Register Mappings: registerMappings.cpp

```
#ifndef REGISTERMAPPINGS_ORACLE
# include "registerMappings.h"
#endif

void registerMappings(oracle::occi::Environment* envOCCI_)
{
  oracle::occi::Map *mapOCCI_ = envOCCI_->getMap();
  mapOCCI_->put("SCOTT.FULL_NAME", CFullName::readSQL, CFullName::writeSQL);
  mapOCCI_->put("SCOTT.ADDRESS", CAddress::readSQL, CAddress::writeSQL);
  mapOCCI_->put("SCOTT.PERSON", CPerson::readSQL, CPerson::writeSQL);
  mapOCCI_->put("SCOTT.STUDENT", CStudent::readSQL, CStudent::writeSQL);
}
```

Note: This demo extends types that are used as attributes and base classes of other generated types. `cperson.h` `#includes` `myfullname.h` and `myaddress.h`. `cstudent.h` `#includes` `myperson.h`. `PERSON` and `STUDENT` must be generated in separate files because `PERSON` is an extended class and it is the base class for `STUDENT`.

Example of User-Written Extension File: myfullname.h

```
#ifndef MYFULLNAME_ORACLE
# define MYFULLNAME_ORACLE

using namespace oracle::occi;
using namespace std;

#ifndef CFULLNAME_ORACLE
#include "cfullname.h"
#endif

/************************************************************************/
// declarations for the MyFullName class.
/************************************************************************/
class MyFullName : public CFullName {
 public:
   MyFullName(string first_name, string last_name);
   void displayInfo();
};

#endif
```

Example of User-Written Extension File: myaddress.h

```
#ifndef MYADDRESS_ORACLE
# define MYADDRESS_ORACLE

using namespace oracle::occi;
using namespace std;

#ifndef CADDRESS_ORACLE
#include "caddress.h"
#endif
```

```
/************************************************************************/
// declarations for the MyAddress class.
/************************************************************************/
class MyAddress : public CAddress {
 public:
   MyAddress(string state_i, string zip_i);
   void displayInfo();
};

#endif
```

Example of User-Written Extension File: myperson.h

```
#ifndef MYPERSON_ORACLE
# define MYPERSON_ORACLE

using namespace oracle::occi;
using namespace std;

#ifndef CPERSON_ORACLE
#include "cperson.h"
#endif

/************************************************************************/
// declarations for the MyPerson class.
/************************************************************************/
class MyPerson : public CPerson {
 public:
   MyPerson();
   MyPerson(void *ctxOCCI_) : CPerson(ctxOCCI_) { };
   MyPerson(Number id_i, MyFullName *name_i,const Ref<MyAddress>& addr_i);
   void move(const Ref<MyAddress>& new_addr);
   void displayInfo();
};

#endif
```

Example of User-Written Extension File: mystudent.h

```
#ifndef MYSTUDENT_ORACLE
# define MYSTUDENT_ORACLE

using namespace oracle::occi;
```

```
using namespace std;

#ifndef CSTUDENT_ORACLE
#include "cstudent.h"
#endif

/************************************************************************/
// declarations for the MyStudent class.
/************************************************************************/
class MyStudent : public CStudent {
 public:
   MyStudent(Number id_i, MyFullName *name_i, Ref<MyAddress>& addr_i, string
school_name);
   void displayInfo();
} ;

#endif
```

Example of User-Written Extension File: mydemo.cpp

```
#include <occi.h>
using namespace oracle::occi;
using namespace std;

#include "registerMappings.h"

#include "myfullname.h"
#include "myaddress.h"
#include "myperson.h"
#include "mystudent.h"

/************************************************************************/
// method implementations for MyFullName class.
/***********************************************************************/

/* initialize MyFullName */
MyFullName::MyFullName(string first_name, string last_name)
{
  setFirstname(first_name);
  setLastname(last_name);
}

/* display all the information in MyFullName */
```

```
void MyFullName::displayInfo()
{
   cout << "FIRST NAME is:  " << getFirstname() << endl;
   cout << "LAST NAME is:  " << getLastname() << endl;
}

/*********************************************************************/
// method implementations for MyAddress class.
/********************************************************************/

/* initialize MyAddress */
MyAddress::MyAddress(string state_i, string zip_i)
{
   setState(state_i);
   setZip(zip_i);
}

/* display all the information in MyAddress */
void MyAddress::displayInfo()
{
   cout << "STATE is:  " << getState() << endl;
   cout << "ZIP is:  " << getZip() << endl;
}

/**********************************************************************/
// method implementations for MyPerson class.
/**********************************************************************/
/* Default constructor needed because CStudent inherits from MyPerson */
MyPerson::MyPerson(){}

/* initialize MyPerson */
MyPerson::MyPerson(Number id_i,  MyFullName* name_i, const Ref<MyAddress>& addr_
i)
{
   setId(id_i);
   setName(name_i);
   setCurr_addr(addr_i);
}

/* Move Person from curr_addr to new_addr */
void MyPerson::move(const Ref<MyAddress>& new_addr)
{
  // append curr_addr to the vector
  getPrev_addr_l().push_back(getCurr_addr());
  setCurr_addr(new_addr);
```

```
  this->markModified();
}

/*  Display all the information of MyPerson */
void MyPerson::displayInfo()
{
  cout << "-----------------------------------  " << endl;

  cout << "ID is:  " << (int)getId() << endl;

  getName()->displayInfo();

  // de-referencing the Ref attribute using -> operator
  getCurr_addr()->displayInfo();

  cout << "Prev Addr List: " << endl;
  for (int i = 0; i < getPrev_addr_l().size(); i++)
  {
     // access the collection elements using [] operator
     getPrev_addr_l()[i]->displayInfo();
  }
}

/*********************************************************************/
// method implementations for MyStudent class.
/********************************************************************/

/* initialize MyStudent */
MyStudent::MyStudent(Number id_i, MyFullName *name_i, Ref<MyAddress>& addr_i,
string school_name_i)
{
   setId(id_i);
   setName(name_i);
   setCurr_addr(addr_i);
   setSchool_name(school_name_i);

}

/* display the information in MyStudent */
void MyStudent::displayInfo()
{
   MyPerson::displayInfo();
   cout << "SCHOOL NAME is:  " << getSchool_name() << endl;
}
```

```
void process(Connection *conn)
{
  /* creating a persistent object of type Address in the connection,
     conn, and the database table, ADDR_TAB */
  MyAddress *addr1 = new(conn, "ADDR_TAB") MyAddress("CA", "94065");

  /* commit the transaction which results in the newly created object, addr1,
     being flushed to the server */
  conn->commit();

  MyFullName name1("Joe", "Black");

  /* creating a persistent object of type Person in the connection,
     conn, and the database table, PERSON_TAB */
  MyPerson *person1 = new(conn, "PERSON_TAB") MyPerson(1,&name1,
                             addr1->getRef());

  /* commit the transaction which results in the newly created object,
     person1 being flushed to the server */
  conn->commit();

  Statement *stmt = conn->createStatement(
    "SELECT REF(per) from person_tab per ");

  ResultSet *resultSet = stmt->executeQuery();

  if (!resultSet->next())
  {
     cout << "No record found \n";
  }

  RefAny joe_refany = resultSet->getRef(1);
  Ref <MyPerson> joe_ref(joe_refany);

  /* de-referencing Ref using ptr() operator.  operator -> and operator *
   also could be used to de-reference the Ref. As part of de-referencing,
   if the referenced object is not found in the application cache, the object
   data is retrieved from the server and unmarshalled into Person instance
   through MyPerson::readSQL() method. */

  MyPerson *joe = joe_ref.ptr();
  joe->displayInfo();

  /* creating a persistent object of type MyAddress, in the connection,
     conn and the database table, ADDR_TAB */
```

```
MyAddress *new_addr1 = new(conn, "ADDR_TAB") MyAddress("PA", "92140");
conn->commit();

joe->move(new_addr1->getRef());
joe->displayInfo();

/* commit the transaction which results in the newly created object,
   new_addr1 and the dirty object, joe to be flushed to the server.  */
conn->commit();

MyAddress *addr2 = new(conn, "ADDR_TAB") MyAddress("CA", "95065");
MyFullName name2("Jill", "White");
Ref<MyAddress> addrRef = addr2->getRef();
MyStudent *student2 = new(conn, "STUDENT_TAB") MyStudent(2, &name2,
                           addrRef, "Stanford");
conn->commit();

Statement *stmt2 = conn->createStatement(
 "SELECT REF(Student) from student_tab Student where id = 2");

ResultSet *resultSet2 = stmt2->executeQuery();
if (!resultSet2->next())
{
   cout << "No record found \n";
}

RefAny jillrefany = resultSet2->getRef(1);
Ref <MyStudent> jill_ref(jillrefany);

MyStudent *jill = jill_ref.ptr();

cout << jill->getPrev_addr_l().size();

jill->displayInfo();

MyAddress *new_addr2 = new(conn, "ADDR_TAB") MyAddress("CO", "80021");
conn->commit();

jill->move(new_addr2->getRef());

jill->displayInfo();

jill->markModified();

conn->commit();
```

```
  /* The following delete statements delete the objects only from the
     application cache.  To delete the objects from the server, mark_deleted()
     should be used. */
  delete person1;
  delete addr1;
  delete new_addr1;
  delete addr2;
  delete student2;
  delete new_addr2;

  conn->terminateStatement(stmt);
  conn->terminateStatement(stmt2);

}

/*************************************************************************/
// main function of this OCCI application.
// This application connects to the database as scott/tiger, creates
// the Person (Joe Black) whose Address is in CA, and commits the changes.
// The Person object is then retrieved from the database and its
// information is displayed. A second Address object is created (in PA),
// then the previously retrieved Person object (Joe Black) is moved to
// this new address. The Person object is then displayed again.
// The similar commands are executed for "Jill White", a Student at Stanford,
// who is moved from CA to CO.
/*************************************************************************/

int main()
{

  Environment *env = Environment::createEnvironment(Environment::OBJECT );

  /* Call the OTT generated function to register the mappings */
  registerMappings(env);

  Connection *conn = env->createConnection("scott","tiger","");

  process(conn);

  env->terminateConnection(conn);
  Environment::terminateEnvironment(env);

  return 0;
}
```

Note: Each extension class declaration must #include the generated header file of the class it is extending. For example, `myfullname.h` must #include `cperson.h` and `mystudent.h` must #include `cstudent.h`. `mydemo.cpp` must #include `registerMappings.h` in order to call the `registerMapping` function to register the mappings.

Output Generated by Example OTT Application

The output generated from the example OCCI application is:

```
-----------------------------------
ID is:  1
FIRST NAME is:  Joe
LAST NAME is:  Black
STATE is:  CA
ZIP is:  94065
Prev Addr List:
-----------------------------------
ID is:  1
FIRST NAME is:  Joe
LAST NAME is:  Black
STATE is:  PA
ZIP is:  92140
Prev Addr List:
STATE is:  CA
ZIP is:  94065
-----------------------------------
ID is:  2
FIRST NAME is:  Jill
LAST NAME is:  White
STATE is:  CA
ZIP is:  95065
Prev Addr List:
SCHOOL NAME is:  Stanford
-----------------------------------
ID is:  2
FIRST NAME is:  Jill
LAST NAME is:  White
STATE is:  CO
```

```
ZIP is:  80021
Prev Addr List:
STATE is:  CA
ZIP is:  95065
SCHOOL NAME is:  Stanford
```

OTT Utility Reference

Behavior of the OTT utility is controlled by parameters that are specified either on the OTT command line or in a CONFIG file. Certain parameters may also appear in the INTYPE file.

This section provides detailed information about the following topics:

- OTT Command Line Syntax
- OTT Utility Parameters
- Where OTT Parameters Can Appear
- Structure of the INTYPE File
- Nested #include File Generation
- SCHEMA_NAMES Usage
- Default Name Mapping
- Restriction Affecting the OTT Utility: File Name Comparison

OTT Command Line Syntax

The OTT command line interface is used when explicitly invoking the OTT utility to translate database types into C structures or C++ classes. This is always required when developing OCI, OCCI, or Pro*C/C++ applications that use objects.

An OTT command line statement consists of the command OTT, followed by a list of OTT utility parameters.

The parameters that can appear on the OTT command line statement are listed alphabetically as follows:

```
[ATTRACCESS={PRIVATE|PROTECTED}]

[CASE={SAME|LOWER|UPPER|OPPOSITE}]

CODE={C|ANSI_C|KR_C|CPP}
```

```
[CONFIG=filename]
[CPPFILE=filename]
[ERRTYPE=filename]
[HFILE=filename]
[INITFILE=filename]
[INITFUNC=filename]
[INTYPE=filename]
[MAPFILE=filename]
[MAPFUNC=filename]
OUTTYPE=filename
[SCHEMA_NAMES={ALWAYS|IF_NEEDED|FROM_INTYPE}]
[TRANSITIVE={TRUE|FALSE}]
[USERID=username/password[@db_name]]
```

Note: Generally, the order of the parameters following the OTT command does not matter. The `OUTTYPE` and `CODE` parameters are always required.

The `HFILE` parameter is almost always used. If omitted, then `HFILE` must be specified individually for each type in the `INTYPE` file. If the OTT utility determines that a type not listed in the `INTYPE` file must be translated, then an error will be reported. Therefore, it is safe to omit the `HFILE` parameter only if the `INTYPE` file was previously generated as an OTT `OUTTYPE` file.

If the `INTYPE` file is omitted, then the entire schema will be translated. See the parameter descriptions in the following section for more information.

The following is an example of an OTT command line statement (enter it as one line):

```
ott userid=scott/tiger intype=in.typ outtype=out.typ code=c hfile=demo.h
errtype=demo.tls case=lower
```

Each of the OTT command line parameters is described in the following section.

OTT Utility Parameters

Enter parameters on the OTT command line using the following format:

```
parameter=value
```

In this example, `parameter` is the literal parameter string and `value` is a valid parameter setting. The literal parameter string is not case sensitive.

Separate command line parameters by using either spaces or tabs.

Parameters can also appear within a configuration file, but, in that case, no whitespace is permitted within a line, and each parameter must appear on a separate line. Additionally, the parameters `CASE`, `CPPFILE`, `HFILE`, `INITFILE`, `INTFUNC`, `MAPFILE`, and `MAPFUNC` can appear in the `INTYPE` file.

OTT utility parameters are described in the following sections:

- ATTRACCESS Parameter
- CASE Parameter
- CODE Parameter
- CONFIG Parameter
- CPPFILE Parameter
- ERRTYPE Parameter
- HFILE Parameter
- INITFILE Parameter
- INITFUNC Parameter
- INTYPE Parameter
- MAPFILE Parameter
- MAPFUNC Parameter
- OUTTYPE Parameter
- SCHEMA_NAMES Parameter
- TRANSITIVE Parameter
- USERID Parameter

ATTRACCESS Parameter

For C++ only. This parameter instructs the OTT utility whether to generate PRIVATE or PROTECTED access for the type attributes. If PRIVATE is specified, the OTT utility generates an accessor (get*xxx*) and mutator (set*xxx*) method for each of the type attributes.

```
ATTRACCESS=PRIVATE|PROTECTED
```

The default is PROTECTED.

CASE Parameter

This parameter affects the case of certain C or C++ identifiers generated by the OTT utility. The valid values of CASE are SAME, LOWER, UPPER, and OPPOSITE.

If CASE=SAME, the case of letters is not changed when converting database type and attribute names to C or C++ identifiers.

If CASE=LOWER, then all uppercase letters are converted to lowercase.

If CASE=UPPER, then all lowercase letters are converted to uppercase.

If CASE=OPPOSITE, then all uppercase letters are converted to lowercase, and all lowercase letters are converted to uppercase.

```
CASE=[SAME|LOWER|UPPER|OPPOSITE]
```

This parameter affects only those identifiers (attributes or types not explicitly listed) not mentioned in the INTYPE file. Case conversion takes place after a legal identifier has been generated.

The case of the C structure identifier for a type specifically mentioned in the INTYPE file is the same as its case in the INTYPE file. For example, consider this line included in the INTYPE file:

```
TYPE Worker
```

The OTT utility generates the following C structure:

```
struct Worker {...};
```

On the other hand, consider an INTYPE file that includes the following line:

```
TYPE wOrKeR
```

The OTT utility generates this C structure, which follows the case specified in the INTYPE file:

```
struct wOrKeR {...};
```

Case insensitive SQL identifiers not mentioned in the INTYPE file will appear in uppercase if CASE=SAME, and in lowercase if CASE=OPPOSITE. A SQL identifier is case insensitive if it was not quoted when it was declared.

CODE Parameter

This parameter indicates which host language is to be output by the OTT utility. CODE=C is equivalent to CODE=ANSI_C. CODE=CPP must be specified for the OTT utility to generate C++ code for OCCI applications.

```
CODE=C|KR_C|ANSI_C|CPP
```

This parameter is required, and there is no default value. You must specify one of the host languages.

CONFIG Parameter

This parameter specifies the name of the OTT configuration file to be used. The configuration file lists commonly used parameter specifications. Parameter specifications are also read from a system configuration file found in an operating system-dependent location. All remaining parameter specifications must appear either on the command line or in the INTYPE file.

CONFIG=*filename*

Note: The CONFIG parameter can only be specified on the OTT command line. It is not allowed in the CONFIG file.

CPPFILE Parameter

For C++ only. This parameter specifies the name of the C++ source file that will contain the method implementations generated by the OTT utility.

This parameter is required under the following conditions:

- A type not mentioned in the INTYPE file must be generated and two or more CPPFILEs are being generated. In this case, the unmentioned type goes in the CPPFILE specified on the command line.

- The INTYPE parameter is not specified, and you want the OTT utility to translate all the types in the schema.

The restrictions to this are similar to that of the existing HFILE parameter restrictions already in place for Pro*C/C++ and OCI. This parameter is optional when the CPPFILE is specified for individual types in the INTYPE file.

```
CPPFILE=filename
```

ERRTYPE Parameter

If you supply this parameter, then a listing of the INTYPE file is written to the ERRTYPE file, along with all information and error messages. Information and error messages are sent to the standard output whether or not the ERRTYPE parameter is specified.

Essentially, the ERRTYPE file is a copy of the INTYPE file with error messages added. In most cases, an error message will include a pointer to the text that caused the error.

If the filename specified for the ERRTYPE parameter on the command line does not include an extension, a platform-specific extension such as .TLS or .tls is automatically added.

```
ERRTYPE=filename
```

HFILE Parameter

This parameter specifies the name of the header (.h) file to be generated by the OTT utility. The HFILE specified on the command line contains the declarations of types that are mentioned in the INTYPE file but whose header files are not specified there.

This parameter is required unless the header file for each type is specified individually in the INTYPE file. This parameter is also required if a type not mentioned in the INTYPE file must be generated because other types require it, and these other types are declared in two or more different files.

If the filename specified for the HFILE parameter on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as H or .h is automatically added.

```
HFILE=filename
```

INITFILE Parameter

For OCI only. This parameter specifies the name of the initialization file to be generated by the OTT utility. If you omit this parameter, then the initialization file will not be generated.

For Pro*C/C++ programs, the `INITFILE` is not necessary, because the SQLLIB run-time library performs the necessary initializations. An OCI programmer must compile and link the `INITFILE` files, and must call the initialization functions when an environment handle is created.

If the filename specified for the `INITFILE` parameter on the command line or in the `INTYPE` file does not include an extension, a platform-specific extension such as `C` or `.c` is automatically added.

```
INITFILE=filename
```

INITFUNC Parameter

For OCI only. This parameter specifies the name of the initialization function to be generated by the OTT utility.

This parameter is optional. If you omit this parameter, then the name of the initialization function is derived from the name of the `INITFILE`.

```
INITFUNC=filename
```

INTYPE Parameter

This parameter specifies the name of the file from which to read the list of object type specifications. The OTT utility translates each type in the list.

```
INTYPE=filename
```

`INTYPE=` may be omitted if `USERID` and `INTYPE` are the first two parameters, in that order, and `USERID=` is omitted. For example,

```
OTT username/password filename...
```

If the `INTYPE` parameter is not specified, all types in the user's schema will be translated.

The `INTYPE` file can be thought of as a makefile for type declarations. It lists the types for which C structure declarations or C++ classes are needed. The format of the `INTYPE` file is described in "Structure of the INTYPE File" on page 7-98.

If the filename specified for the INTYPE parameter on the command line does not include an extension, a platform-specific extension such as TYP or .typ is automatically added.

See Also: "Structure of the INTYPE File" on page 7-98 for more information about the format of the INTYPE file

MAPFILE Parameter

For C++ only. This parameter specifies the name of the mapping file (.cpp) and corresponding header file (.h) that is generated by the OTT utility. The .cpp file contains the implementation of the functions to register the mappings while the .h file contains the prototype for the function.

This parameter is required for the generation of C++. If you specify CODE=CPP, then you must also specify a value for the MAPFILE parameter. Otherwise, the OTT utility generates an error.

This parameter may be specified either on the command line or in the INTYPE file.

```
MAPFILE=filename
```

MAPFUNC Parameter

For C++ only. This parameter specifies the name of the function to be used to register the mappings generated by the OTT utility.

This parameter is optional for the generation of C++. If this parameter is omitted, then the name of the function to register the mappings is derived from the filename specified in the MAPFILE parameter.

This parameter may be specified either on the command line or in the INTYPE file.

```
MAPFUNC=functionname
```

OUTTYPE Parameter

This parameter specifies the name of the file into which the OTT utility writes type information for all the object datatypes it processes. This file includes all types explicitly named in the INTYPE file, and may include additional types that are translated because they are used in the declarations of other types that need to be translated. This file may be used as an INTYPE file in a future invocation of the OTT utility.

```
OUTTYPE=filename
```

If the INTYPE and OUTTYPE parameters refer to the same file, then the new INTYPE information replaces the old information in the INTYPE file. This provides a convenient way for the same INTYPE file to be used repeatedly in the cycle of altering types, generating type declarations, editing source code, precompiling, compiling, and debugging.

This parameter is required.

If the filename specified for the OUTTYPE parameter on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as TYP or .typ is automatically added.

SCHEMA_NAMES Parameter

This parameter offers control in qualifying the database name of a type from the default schema with a schema name in the OUTTYPE file. The OUTTYPE file generated by the OTT utility contains information about the types processed by the OTT utility, including the type names.

```
SCHEMA_NAMES=ALWAYS|IF_NEEDED|FROM_INTYPE
```

The default value is ALWAYS.

See Also: "SCHEMA_NAMES Usage" on page 7-103 for further information

TRANSITIVE Parameter

This parameter indicates whether type dependencies not explicitly listed in the INTYPE file are to be translated. Valid values are TRUE and FALSE.

```
TRANSITIVE=TRUE|FALSE
```

The default value is TRUE.

If TRANSITIVE=TRUE is specified, then types needed by other types and not mentioned in the INTYPE file are generated.

If TRANSITIVE=FALSE is specified, then types not mentioned in the INTYPE file are not generated, even if they are used as attribute types of other generated types.

USERID Parameter

This parameter specifies the Oracle username, password, and optional database name (Oracle Net database specification string). If the database name is omitted, the default database is assumed.

```
USERID=username/password[@db_name]
```

If this is the first parameter, then `USERID=` may be omitted as shown:

```
OTT username/password ...
```

This parameter is optional. If you omit this parameter, then the OTT utility automatically attempts to connect to the default database as user `OPS$`*`username`*, where *`username`* is the user's operating system username.

Where OTT Parameters Can Appear

Supply OTT parameters on the command line, in a `CONFIG` file named on the command line, or both. Some parameters are also allowed in the `INTYPE` file.

The OTT utility is invoked as follows:

```
OTT parameters
```

You can name a configuration file on the command line with the CONFIG parameter as follows:

```
CONFIG=filename
```

If you name this parameter on the command line, then additional parameters are read from the configuration file named *`filename`*.

In addition, parameters are also read from a default configuration file that resides in an operating system-dependent location. This file must exist, but can be empty. If you choose to enter data in the configuration file, note that no white space is allowed on a line and parameters must be entered one to a line.

If the OTT utility is executed without any arguments, then an online parameter reference is displayed.

The types for the OTT utility to translate are named in the file specified by the `INTYPE` parameter. The parameters `CASE`, `CPPFILE`, `HFILE`, `INITFILE`, `INITFUNC`, `MAPFILE`, and `MAPFNC` may also appear in the `INTYPE` file. `OUTTYPE`

files generated by the OTT utility include the CASE parameter, and include the INITFILE, and INITFUNC parameters if an initialization file was generated or the MAPFILE and MAPFUNC parameters if C++ codes was generated. The OUTTYPE file, as well as the CPPFILE for C++, specifies the HFILE individually for each type.

The case of the OTT command is operating system-dependent.

Structure of the INTYPE File

The INTYPE and OUTTYPE files list the types translated by the OTT utility and provide all the information needed to determine how a type or attribute name is translated to a legal C or C++ identifier. These files contain one or more type specifications. These files also may contain specifications of the following options:

- CASE
- CPPFILE
- HFILE
- INITFILE
- INITFUNC
- MAPFILE
- MAPFUNC

If the CASE, INITFILE, INITFUNC, MAPFILE, or MAPFUNC options are present, then they must precede any type specifications. If these options appear both on the command line and in the INTYPE file, then the value on the command line is used.

See Also: "Overview of the OUTTYPE File" on page 7-27 for an example of a simple user-defined INTYPE file and of the full OUTTYPE file that the OTT utility generates from it

INTYPE File Type Specifications

A type specification in the INTYPE file names an object datatype that is to be translated. The following is an example of a user-created INTYPE file:

```
TYPE employee
  TRANSLATE SALARY$ AS salary
            DEPTNO AS department
TYPE ADDRESS
TYPE PURCHASE_ORDER AS p_o
```

The structure of a type specification is as follows:

```
TYPE type_name
[GENERATE type_identifier]
[AS type_identifier]
[VERSION [=] version_string]
[HFILE [=] hfile_name]
[CPPFILE [=] cppfile_name]
[TRANSLATE{member_name [AS identifier]}...]
```

The `type_name` syntax follows this form:

```
[schema_name.]type_name
```

In this syntax, *`schema_name`* is the name of the schema that owns the given object datatype, and *`type_name`* is the name of the type. The default schema, if one is not specified, is that of the userID invoking the OTT utility. To use a specific schema, you must use schema_name.

The components of the type specification are:

- *`type_name`*: Name of the object datatype.
- *`type_identifier`*: C / C++ identifier used to represent the class. The `GENERATE` clause is used to specify the name of the class that the OTT utility generates. The AS clause specifies the name of the class that you write. The `GENERATE` clause is typically used to extend a class. The `AS` clause, when optionally used without the `GENERATE` clause, specifies the name of the C structure or the C++ class that represents the user-defined type.
- *`version_string`*: Version string of the type that was used when the code was generated by the previous invocation of the OTT utility. The version string is generated by the OTT utility and written to the `OUTTYPE` file, which can later be used as the `INTYPE` file in later invocations of the OTT utility. The version string does not affect how the OTT utility operates, but can be used to select which version of the object datatype is used in the running program.
- *`hfile_name`*: Name of the header file into which the declarations of the corresponding class are written. If you omit the `HFILE` clause, then the file specified by the command line `HFILE` parameter is used.
- *`cppfile_name`*: Name of the C++ source file into which the method implementations of the corresponding class is written. If you omit the `CPPFILE` clause, the file specified by the command line `CPPFILE` parameter is used.

- *member_name*: Name of an attribute (data member) that is to be translated to the identifier.
- *identifier*: C / C++ identifier used to represent the attribute in the program. You can specify identifiers in this way for any number of attributes. The default name mapping algorithm is used for the attributes not mentioned.

An object datatype may need to be translated for one of two reasons:

- It appears in the INTYPE file.
- It is required to declare another type that must be translated, and the TRANSITIVE parameter is set to TRUE.

If a type that is not mentioned explicitly is required by types declared in exactly one file, then the translation of the required type is written to the same files as the explicitly declared types that require it.

If a type that is not mentioned explicitly is required by types declared in two or more different files, then the translation of the required type is written to the global HFILE file.

Note: As of release 9.0.1, you may indicate whether the OTT utility is to generate required object types that are not specified in the INTYPE file. Set TRANSITIVE=FALSE so the OTT utility will not to generate required object types. The default is TRANSITIVE=TRUE.

Nested #include File Generation

HFILE files generated by the OTT utility #include other necessary files, and #define a symbol constructed from the name of the file. This symbol #defined can then be used to determine if the related HFILE file has already been #included. Consider, for example, a database with the following types:

```
create type px1 AS OBJECT (col1 number, col2 integer);
create type px2 AS OBJECT (col1 px1);
create type px3 AS OBJECT (col1 px1);
```

The INTYPE file contains the following information:

```
CASE=lower
type px1
  hfile tott95a.h
```

```
type px3
  hfile tott95b.h
```

You invoke the OTT utility as follows:

```
ott scott/tiger tott95i.typ outtype=tott95o.typ code=c
```

The OTT utility then generates the following two header files, named `tott95a.h` and `tott95b.h`.

The content of the `tott95a.h` file is as follows:

```
#ifndef TOTT95A_ORACLE
#define TOTT95A_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif
typedef OCIRef px1_ref;
struct px1
{
   OCINumber col1;
   OCINumber col2;
}
typedef struct px1 px1;
struct px1_ind
{
   OCIInd _atomic;
   OCIInd col1;
   OCIInd col2;
}
typedef struct px1_ind px1_ind;
#endif
```

The content of the `tott95b.h` file is as follows:

```
#ifndef TOTT95B_ORACLE
#define TOTT95B_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#ifndef TOTT95A_ORACLE
#include "tott95a.h"
#endif
typedef OCIRef px3_ref;
```

```
struct px3
{
   struct px1 col1;
};
typedef struct px3 px3;
struct px3_ind
{
   OCIInd _atomic;
   struct px1_ind col1
};
typedef struct px3_ind px3_ind;
#endif
```

In the tott95b.h file, the symbol `TOTT95B_ORACLE` is `#define`d at the beginning of the file. This enables you to conditionally `#include` this header file in another file. To accomplish this, you would use the following construct:

```
#ifndef TOTT95B_ORACLE
#include "tott95b.h"
#endif
```

By using this technique, you can `#include tott95b.h` in, say `foo.h`, without having to know whether some other file `#include`d in `foo.h` also `#include`s `tott95b.h`.

After the definition of the symbol `TOTT95B_ORACLE`, the file oci.h is `#include`d. Every `HFILE` generated by the OTT utility includes `oci.h`, which contains type and function declarations that the Pro*C/C++ or OCI programmer will find useful. This is the only case in which the OTT utility uses angle brackets in a `#include`.

Next, the file `tott95a.h` is included because it contains the declaration of `struct px1`, that `tott95b.h` requires. When the `INTYPE` file requests that type declarations be written to more than one file, the OTT utility determines which other files each `HFILE` must `#include`, and generates each necessary `#include`.

Note that the OTT utility uses quotes in this `#include`. When a program including `tott95b.h` is compiled, the search for `tott95a.h` begins where the source program was found, and will thereafter follow an implementation-defined search rule. If `tott95a.h` cannot be found in this way, then a complete filename (for example, a UNIX absolute path name beginning with a slash character (/)) should be used in the `INTYPE` file to specify the location of `tott95a.h`.

SCHEMA_NAMES Usage

This parameter affects whether the name of a type from the default schema, to which the OTT utility is connected, is qualified with a schema name in the OUTTYPE file.

The name of a type from a schema other that the default schema is always qualified with a schema name in the OUTTYPE file.

The schema name, or its absence, determines in which schema the type is found during program execution.

There are three valid values for the SCHEMA_NAMES parameter:

- SCHEMA_NAMES=ALWAYS (default)

  All type names in the OUTTYPE file are qualified with a schema name.

- SCHEMA_NAMES=IF_NEEDED

  The type names in the OUTTYPE file that belong to the default schema are not qualified with a schema name. As always, type names belonging to other schemas are qualified with the schema name.

- SCHEMA_NAMES=FROM_INTYPE

  A type mentioned in the INTYPE file is qualified with a schema name in the OUTTYPE file if, and only if, it was qualified with a schema name in the INTYPE file. A type in the default schema that is not mentioned in the INTYPE file but that is generated because of type dependencies, is written with a schema name only if the first type encountered by the OTT utility that depends on it is also written with a schema name. However, a type that is not in the default schema to which the OTT utility is connected is always written with an explicit schema name.

The OUTTYPE file generated by the OTT utility is the Pro*C/C++ INTYPE file. This file matches database type names to C structure names. This information is used at runtime to make sure that the correct database type is selected into the structure. If a type appears with a schema name in the OUTTYPE file (Pro*C/C++ INTYPE file), then the type is found in the named schema during program execution. If the type appears without a schema name, then the type is found in the default schema to which the program connects, which may be different from the default schema the OTT utility used.

Example of SCHEMA_NAMES Parameter Usage

Consider an example where the SCHEMA_NAMES parameter is set to FROM_INTYPE, and the INTYPE file contains the following:

```
TYPE Person
TYPE joe.Dept
TYPE sam.Company
```

The Pro*C/C++ application that uses the OTT-generated structures uses the types sam.Company, joe.Dept, and Person. Person without a schema name refers to the Person type in the schema to which the application is connected.

If the OTT utility and the application both connect to schema joe, then the application uses the same type (joe.Person) that the OTT utility uses. If the OTT utility connects to schema joe but the application connects to schema mary, then the application uses the type mary.Person. This behavior is appropriate only if the same CREATE TYPE Person statement has been executed in schema joe and schema mary.

On the other hand, the application uses type joe.Dept regardless of which schema the application is connected to. If this is the behavior you want, then be sure to include schema names with your type names in the INTYPE file.

In some cases, the OTT utility translates a type that the user did not explicitly name. For example, consider the following SQL declarations:

```
CREATE TYPE Address AS OBJECT
(
street    VARCHAR2(40),
city      VARCHAR(30),
state     CHAR(2),
zip_code  CHAR(10)
);

CREATE TYPE Person AS OBJECT
(
name      CHAR(20),
age       NUMBER,
addr      ADDRESS
);
```

Suppose that the OTT utility connects to schema joe, SCHEMA_NAMES=FROM_INTYPE is specified, and the user's INTYPE files include either

```
TYPE Person
```

or

```
TYPE joe.Person
```

The INTYPE file does not mention the type `joe.Address`, which is used as a nested object type in type `joe.Person`.

If `Type Person` appears in the `INTYPE` file, then `TYPE Person` and `TYPE Address` appears in the `OUTTYPE` file.

If `TYPE joe.Person` appears in the `INTYPE` file, then `TYPE joe.Person` and `TYPE joe.Address` appear in the `OUTTYPE` file.

If the `joe.Address` type is embedded in several types translated by the OTT utility, but it is not explicitly mentioned in the `INTYPE` file, then the decision of whether to use a schema name is made the first time the OTT utility encounters the embedded `joe.Address` type. If, for some reason, the user wants type `joe.Address` to have a schema name but does not want type `Person` to have one, then you must explicitly request this in the `INTYPE` file as follows:

```
TYPE      joe.Address
```

In the usual case in which each type is declared in a single schema, it is safest for you to qualify all type names with schema names in the `INTYPE` file.

Default Name Mapping

When the OTT utility creates a C or C++ identifier name for an object type or attribute, it translates the name from the database character set to a legal C or C++ identifier. First, the name is translated from the database character set to the character set used by the OTT utility. Next, if a translation of the resulting name is supplied in the `INTYPE` file, that translation is used. Otherwise, the OTT utility translates the name character-by-character to the compiler character set, applying the character case specified in the CASE parameter. The following text describes this in more detail.

When the OTT utility reads the name of a database entity, the name is automatically translated from the database character set to the character set used by the OTT utility. In order for the OTT utility to read the name of the database entity successfully, all the characters of the name must be found in the OTT character set, although a character may have different encodings in the two character sets.

The easiest way to guarantee that the character set used by the OTT utility contains all the necessary characters is to make it the same as the database character set. Note, however, that the OTT character set must be a superset of the compiler character set. That is, if the compiler character set is 7-bit ASCII, then the OTT character set must include 7-bit ASCII as a subset, and if the compiler character set is 7-bit EBCDIC, then the OTT character set must include 7-bit EBCDIC as a subset. The user specifies the character set that the OTT utility uses by setting the `NLS_LANG` environment variable, or by some other operating system-specific mechanism.

Once the OTT utility has read the name of a database entity, it translates the name from the character set used by the OTT utility to the compiler's character set. If a translation of the name appears in the `INTYPE` file, then the OTT utility uses that translation.

Otherwise, the OTT utility attempts to translate the name as follows:

1. If the OTT character set is a multibyte character set, all multibyte characters in the name that have single-byte equivalents are converted to those single-byte equivalents.
2. The name is converted from the OTT character set to the compiler character set. The compiler character set is a single-byte character set such as US7ASCII.
3. The case of letters is set according to how the `CASE` parameter is defined, and any character that is not legal in a C or C++ identifier, or that has no translation in the compiler character set, is replaced by an underscore character (_). If at least one character is replaced by an underscore, then the OTT utility gives a warning message. If all the characters in a name are replaced by underscores, the OTT utility gives an error message.

Character-by-character name translation does not alter underscores, digits, or single-byte letters that appear in the compiler character set, so legal C or C++ identifiers are not altered.

Name translation may, for example, translate accented single-byte characters such as *o* with an umlaut or an *a* with an accent grave to *o* or *a*, with no accent, and may translate a multibyte letter to its single-byte equivalent. Name translation will typically fail if the name contains multibyte characters that lack single-byte equivalents. In this case, the user must specify name translations in the `INTYPE` file.

The OTT utility will not detect a naming clash caused by two or more database identifiers being mapped to the same C name, nor will it detect a naming problem where a database identifier is mapped to a C keyword.

Restriction Affecting the OTT Utility: File Name Comparison

Currently, the OTT utility determines if two files are the same by comparing the filenames provided by the user either on the command line or in the INTYPE file. But one potential problem can occur when the OTT utility needs to know if two filenames refer to the same file. For example, if the OTT-generated file foo.h requires a type declaration written to foo1.h, and another type declaration written to /private/smith/foo1.h, then the OTT utility should generate one #include if the two files are the same, and two #includes if the files are different. In practice, though, it concludes that the two files are different, and generates two #includes as follows:

```
#ifndef FOO1_ORACLE
#include "foo1.h"
#endif
#ifndef FOO1_ORACLE
#include "/private/smith/foo1.h"
#endif
```

If foo1.h and /private/smith/foo1.h are different files, then only the first one will be included. If foo1.h and /private/smith/foo1.h are the same file, then a redundant #include will be written.

Therefore, if a file is mentioned several times on the command line or in the INTYPE file, then each mention of the file should use exactly the same filename.

Part II

OCCI API Reference

This part contains one chapter:

- Chapter 8, "OCCI Classes and Methods"

8 OCCI Classes and Methods

This chapter describes the OCCI classes and methods for C++.

The topics discussed include:

- Summary of OCCI Classes on page 8-2
- OCCI Classes and Methods on page 8-3

Summary of OCCI Classes

Table 8–1 provides a brief description of all the OCCI classes. This section is followed by detailed descriptions of each class and its methods.

Table 8–1 OCCI Classes

Class	Description
Bfile Class on page 8-5	Provides access to a SQL BFILE value.
Blob Class on page 8-13	Provides access to a SQL BLOB value.
Bytes Class on page 8-24	Examines individual bytes of a sequence for comparing bytes, searching bytes, and extracting bytes.
Clob Class on page 8-27	Provides access to a SQL CLOB value.
Connection Class on page 8-40	Represents a connection with a specific database.
ConnectionPool Class on page 8-45	Represents a connection pool with a specific database.
Date Class on page 8-51	Specifies abstraction for SQL DATE data items. Also provides formatting and parsing operations to support the OCCI escape syntax for date values.
Environment Class on page 8-64	Provides an OCCI environment to manager memory and other resources of OCCI objects. An OCCI driver manager maps to an OCI environment handle.
IntervalDS Class on page 8-70	Represents a period of time in terms of days, hours, minutes, and seconds.
IntervalYM Class on page 8-83	Represents a period of time in terms of year and months.
Map Class on page 8-95	Used to store the mapping of the SQL structured type to C++ classes.
MetaData Class on page 8-97	Used to determine types and properties of columns in a `ResultSet`, that of existing schema objects in the database, or the database as a whole.
Number Class on page 8-104	Provides control over rounding behavior.

Class	Description
PObject Class on page 8-130	When defining types, enables specification of persistent or transient instances. Class instances derived from `PObject` can be either persistent or transient. If persistent, a class instance derived from `PObject` inherits from the `PObject` class; if transient, there is no inheritance.
Ref Class on page 8-137	The mapping in C++ for the SQL REF value, which is a reference to a SQL structured type value in the database.
RefAny Class on page 8-144	The mapping in C++ for the SQL REF value, which is a reference to a SQL structured type value in the database.
ResultSet Class on page 8-147	Provides access to a table of data generated by executing an OCCI `Statement`.
SQLException Class on page 8-169	Provides information on database access errors.
Statement Class on page 8-171	Used for executing SQL statements, including both query statements and insert / update / delete statements.
Stream Class on page 8-215	Used to provide streamed data (usually of the LONG datatype) to a prepared DML statement or stored procedure call.
Timestamp Class on page 8-219	Specifies abstraction for SQL TIMESTAMP data items. Also provides formatting and parsing operations to support the OCCI escape syntax for time stamp values.

OCCI Classes and Methods

OCCI classes are defined in the `oracle::occi` namespace. An OCCI class name within the `oracle::occi` namespace can be referred to in one of three ways:

- Use the scope resolution operator (`::`) for each OCCI class name.
- Use the `using` declaration for each OCCI class name.
- Use the `using` directive for all OCCI class name.

Scope Resolution Operator

The scope resolution operator (`::`) is used to explicitly specify the `oracle::occi` namespace and the OCCI class name. To declare `myConnection`, a `Connection` object, using the scope resolution operator, you would use the following syntax:

```
oracle::occi::Connection myConnection;
```

using Declaration

The `using` declaration is used when the OCCI class name can be used in a compilation unit without conflict. To declare the OCCI class name in the `oracle::occi` namespace, you would use the following syntax:

```
using oracle::occi::Connection;
```

`Connection` now refers to `oracle::occi::Connection`, and `myConnection` can be declared as:

```
Connection myConnection;
```

using Directive

The `using` directive is used when all OCCI class names can be used in a compilation unit without conflict. To declare all OCCI class names in the `oracle::occi` namespace, you would use the following syntax:

```
using oracle::occi;
```

Then, just as with the `using` declaration, the following declaration would now refer to the OCCI class `Connection`:

```
Connection myConnection;
```

Bfile Class

The `Bfile` class defines the common properties of objects of type `BFILE`. A BFILE is a large binary file stored in an operating system file outside of the Oracle database. A `Bfile` object contains a logical pointer to a BFILE, not the BFILE itself.

Methods of the `Bfile` class enable you to perform specific tasks related to `Bfile` objects.

Methods of the `ResultSet` and `Statement` classes, such as `getBfile()` and `setBfile()`, enable you to access a SQL BFILE value.

To create a null `Bfile` object, use the syntax:

```
Bfile();
```

The only methods valid on a null `Bfile` object are `setNull()`, `isNull()`, and `operator=()`.

To create an uninitialized `Bfile` object, use the syntax:

```
Bfile(const Connection *connectionp);
```

An uninitialized `Bfile` object can be initialized by:

- The `setName()` method. The BFILE can then be modified by inserting this BFILE into the table and then retrieving it using `SELECT ... FOR UPDATE`. The `write()` method will modify the BFILE; however, the modified data will be flushed to the table only when the transaction is committed. Note that an insert is not required.
- Assigning an initialized `Bfile` object to it.

To create a copy of an existing `Bfile` object, use the syntax:

```
Bfile(const Bfile &srcBfile);
```

Summary of Bfile Methods

Method	Summary
close() on page 8-6	Close a previously opened BFILE.
closeStream() on page 8-6	Close the stream obtained from the BFILE.
fileExists() on page 8-7	Test whether the BFILE exists.
getDirAlias() on page 8-7	Return the directory alias of the BFILE.
getFileName() on page 8-7	Return the name of the BFILE.
getStream() on page 8-8	Return data from the BFILE as a `Stream` object.
isInitialized() on page 8-8	Test whether the `Bfile` object is initialized.
isNull() on page 8-8	Test whether the `Bfile` object is atomically null.
isOpen() on page 8-9	Test whether the BFILE is open.
length() on page 8-9	Return the number of bytes in the BFILE.
open() on page 8-9	Open the BFILE with read-only access.
operator=() on page 8-9	Assign a BFILE locator to the `Bfile` object.
operator==() on page 8-10	Test whether two `Bfile` objects are equal.
operator!=() on page 8-10	Test whether two `Bfile` objects are not equal.
read() on page 8-11	Read a specified portion of the BFILE into a buffer.
setName() on page 8-12	Set the directory alias and file name of the BFILE.
setNull() on page 8-12	Set the `Bfile` object to atomically null.

close()

This method closes a previously opened BFILE.

Syntax

```
void close();
```

closeStream()

This method closes the stream obtained from the BFILE.

Syntax

```
void closeStream(Stream *stream);
```

Parameters

stream

The stream to be closed.

fileExists()

This method tests whether the BFILE exists. If the BFILE exists, then true is returned; otherwise, false is returned.

Syntax

```
bool fileExists() const;
```

getDirAlias()

This method returns a string containing the directory alias associated with the BFILE.

Syntax

```
string getDirAlias() const;
```

getFileName()

This method returns a string containing the file name associated with the BFILE.

Syntax

```
string getFileName() const;
```

getStream()

This method returns a `Stream` object read from the BFILE. If a stream is already open, it is disallowed to open another stream on the `Bfile` object. The stream must be closed before performing any `Bfile` object operations.

Syntax

```
Stream* getStream(unsigned int offset = 1,
    unsigned int amount = 0);
```

Parameters

offset

The starting position at which to begin reading data from the BFILE. If *`offset`* is not specified, the data is written from the beginning of the BLOB.

Valid values are:

Numbers greater than or equal to 1.

amount

The total number of bytes to be read from the BFILE; if *`amount`* is 0, the data will be read in a streamed mode from input *`offset`* until the end of the BFILE.

isInitialized()

This method tests whether the `Bfile` object has been initialized. If the `Bfile` object has been initialized, then true is returned; otherwise, false is returned.

Syntax

```
bool isInitialized() const;
```

isNull()

This method tests whether the `Bfile` object is atomically null. If the `Bfile` object is atomically null, then true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

isOpen()

This method tests whether the BFILE is open. The BFILE is considered to be open only if it was opened by a call on this `Bfile` object. (A different `Bfile` object could have opened this file as more than one open can be performed on the same file by associating the file with different `Bfile` objects). If the BFILE is open, then true is returned; otherwise, false is returned.

Syntax

```
bool isOpen() const;
```

length()

This method returns the number of bytes (inclusive of the end of file marker) in the BFILE.

Syntax

```
unsigned int length() const;
```

open()

This method opens an existing BFILE for read-only access. This function is meaningful the first time it is called for a `Bfile` object.

Syntax

```
void open();
```

operator=()

This method assigns a `Bfile` object to the current `Bfile` object. The source `Bfile` object is assigned to this `Bfile` object only when this `Bfile` object gets stored in the database.

Syntax

```
Bfile& operator=(const Bfile &srcBfile);
```

Parameters

srcBfile

The `Bfile` object to be assigned to the current `Bfile` object.

operator==()

This method compares two `Bfile` objects for equality. The `Bfile` objects are equal if they both refer to the same BFILE. If the `Bfile` objects are null, then false is returned. If the `Bfile` objects are equal, then true is returned; otherwise, false is returned.

Syntax

```
bool operator==(const Bfile &srcBfile) const;
```

Parameters

srcBfile

The `Bfile` object to be compared with the current `Bfile` object.

operator!=()

This method compares two `Bfile` objects for inequality. The `Bfile` objects are equal if they both refer to the same BFILE. If the `Bfile` objects are not equal, then true is returned; otherwise, false is returned.

Syntax

```
bool operator!=(const Bfile &srcBfile) const;
```

Parameters

srcBfile
The `Bfile` object to be compared with the current `Bfile` object.

read()

This method reads a part or all of the BFILE into the buffer specified, and returns the number of bytes read.

Syntax
```
unsigned int read(unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1) const;
```

Parameters

amt
The number of bytes to be read.

Valid values are:

Numbers greater than or equal to 1.

buffer
The buffer that the BFILE data is to be read into.

Valid values are:

A number greater than or equal to *amt*.

bufsize
The size of the buffer that the BFILE data is to be read into.

Valid values are:

A number greater than or equal to *amt*.

offset

The starting position at which to begin reading data from the BFILE. If *offset* is not specified, the data is written from the beginning of the BFILE.

setName()

This method sets the directory alias and file name of the BFILE.

Syntax

```
void setName(const string &dirAlias,
    const string &fileName);
```

Parameters

dirAlias

The directory alias to be associated with the BFILE.

fileName

The file name to be associated with the BFILE.

setNull()

This method sets the `Bfile` object to atomically null.

Syntax

```
void setNull();
```

Blob Class

The `Blob` class defines the common properties of objects of type BLOB. A BLOB is a large binary object stored as a column value in a row of a database table. A `Blob` object contains a logical pointer to a BLOB, not the BLOB itself.

Methods of the `Blob` class enable you to perform specific tasks related to `Blob` objects.

Methods of the `ResultSet` and `Statement` classes, such as `getBlob()` and `setBlob()`, enable you to access an SQL BLOB value.

To create a null `Blob` object, use the syntax:

```
Blob();
```

The only methods valid on a null `Blob` object are `setNull()`, `isNull()`, and `operator=()`.

To create an uninitialized `Blob` object, use the syntax:

```
Blob(const Connection *connectionp);
```

An uninitialized `Blob` object can be initialized by:

- The `setEmpty()` method. The BLOB can then be modified by inserting this BLOB into the table and then retrieving it using `SELECT` ... `FOR UPDATE`. The `write()` method will modify the BLOB; however, the modified data will be flushed to the table only when the transaction is committed. Note that an update is not required.
- Assigning an initialized `Blob` object to it.

To create a copy of a `Blob` object, use the syntax:

```
Blob(const Blob &srcBlob);
```

Summary of Blob Methods

Method	Summary
append() on page 8-14	Append a specified BLOB to the end of the current BLOB.
close() on page 8-15	Close a previously opened BLOB.
closeStream() on page 8-15	Close the `Stream` object obtained from the BLOB.
copy() on page 8-15	Copy a specified portion of a BFILE or BLOB into the current BLOB.
getChunkSize() on page 8-17	Return the chunk size of the BLOB.
getStream() on page 8-17	Return data from the BLOB as a `Stream` object.
isInitialized() on page 8-17	Test whether the `Blob` object is initialized
isNull() on page 8-18	Test whether the `Blob` object is atomically null.
isOpen() on page 8-18	Test whether the BLOB is open
length() on page 8-18	Return the number of bytes in the BLOB.
open() on page 8-18	Open the BLOB with read or read-write access.
operator=() on page 8-19	Assign a BLOB locator to the `Blob` object.
operator==() on page 8-19	Test whether two `Blob` objects are equal.
operator!= () on page 8-20	Test whether two `Blob` objects are not equal.
read() on page 8-20	Read a portion of the BLOB into a buffer.
setEmpty() on page 8-21	Set the `Blob` object to empty.
setNull() on page 8-21	Set the `Blob` object to atomically null.
trim() on page 8-21	Truncate the BLOB to a specified length.
write() on page 8-22	Write a buffer into an *unopened* BLOB.
writeChunk() on page 8-23	Write a buffer into an *open* BLOB.

append()

This method appends a BLOB to the end of the current BLOB.

Syntax

```
void append(const Blob &srcBlob);
```

Parameters

srcBlob
The BLOB to be appended to the current BLOB.

close()

This method closes a BLOB.

Syntax

```
void close();
```

closeStream()

This method closes the Stream object obtained from the BLOB.

Syntax

```
void closeStream(Stream *stream);
```

Parameters

stream
The `Stream` object to be closed.

copy()

This method copies a part or all of the BFILE or BLOB into the current BLOB.

Syntax
There are variants of syntax:

```
void copy(const Bfile &srcBfile,
```

```
    unsigned int numBytes,
    unsigned int dstOffset = 1,
    unsigned int srcOffset = 1);

void copy(const Blob &srcBlob,
    unsigned int numBytes,
    unsigned int dstOffset = 1,
    unsigned int srcOffset = 1);
```

Parameters

srcBfile

The BFILE from which the data is to be copied.

srcBlob

The BLOB from which the data is to be copied.

numBytes

The number of bytes to be copied from the source BFILE or BLOB.

Valid values are:

Numbers greater than 0.

dstOffset

The starting position at which to begin writing data into the current BLOB.

Valid values are:

Numbers greater than or equal to 1.

srcOffset

The starting position at which to begin reading data from the source BFILE or BLOB.

Valid values are:

Numbers greater than or equal to 1.

getChunkSize()

This method returns the chunk size of the BLOB.

When creating a table that contains a BLOB, the user can specify the chunking factor, which can be a multiple of Oracle blocks. This corresponds to the chunk size used by the LOB data layer when accessing or modifying the BLOB.

Syntax

```
unsigned int getChunkSize() const;
```

getStream()

This method returns a `Stream` object from the BLOB. If a stream is already open, it is disallowed to open another stream on `Blob` object, so the user must always close the stream before performing any `Blob` object operations.

Syntax

```
Stream* getStream(unsigned int offset = 1,
    unsigned int amount = 0);
```

Parameters

offset

The starting position at which to begin reading data from the BLOB.

Valid values are:

Numbers greater than or equal to 1.

amount

The total number of consecutive bytes to be read. If *`amount`* is 0, the data will be read from the *`offset`* value until the end of the BLOB.

isInitialized()

This method tests whether the `Blob` object is initialized. If the `Blob` object is initialized, then true is returned; otherwise, false is returned.

Syntax

```
bool isInitialized() const;
```

isNull()

This method tests whether the `Blob` object is atomically null. If the `Blob` object is atomically null, then true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

isOpen()

This method tests whether the BLOB is open. If the BLOB is open, then true is returned; otherwise, false is returned.

Syntax

```
bool isOpen() const;
```

length()

This method returns the number of bytes in the BLOB.

Syntax

```
unsigned int length() const;
```

open()

This method opens the BLOB in read-write or read-only mode.

Syntax

```
void open(LobOpenMode mode = OCCI_LOB_READWRITE);
```

Parameters

mode

The mode the BLOB is to be opened in.

Valid values are:

`OCCI_LOB_READWRITE`

`OCCI_LOB_READONLY`

operator=()

This method assigns a BLOB to the current BLOB. The source BLOB gets copied to the destination BLOB only when the destination BLOB gets stored in the table.

Syntax

```
Blob& operator=(const Blob &srcBlob);
```

Parameters

srcBlob

The BLOB to copy data from.

operator==()

This method compares two `Blob` objects for equality. Two `Blob` objects are equal if they both refer to the same BLOB. Two null `Blob` objects are not considered equal. If the `Blob` objects are equal, then true is returned; otherwise, false is returned.

Syntax

```
bool operator==(const Blob &srcBlob) const;
```

Parameters

srcBlob

The `Blob` object to be compared with the current `Blob` object.

operator!= ()

This method compares two `Blob` objects for inequality. Two `Blob` objects are equal if they both refer to the same BLOB. Two null `Blob` objects are not considered equal. If the `Blob` objects are not equal, then true is returned; otherwise, false is returned.

Syntax

```
bool operator!=(const Blob &srcBlob) const;
```

Parameters

srcBlob

The `Blob` object to be compared with the current `Blob` object.

read()

This method reads a part or all of the BLOB into a buffer. The actual number of bytes read is returned.

Syntax

```
unsigned int read(unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1) const;
```

Parameters

amt

The number of consecutive bytes to be read from the BLOB.

buffer

The buffer into which the BLOB data is to be read.

bufsize

The size of the buffer.

Valid values are:

Numbers greater than or equal to *amount*.

offset

The starting position at which to begin reading data from the BLOB. If *offset* is not specified, the data is written from the beginning of the BLOB.

Valid values are:

Numbers greater than or equal to 1.

setEmpty()

This method sets the `Blob` object to empty.

Syntax

```
void setEmpty();
```

setNull()

This method sets the `Blob` object to atomically null.

Syntax

```
void setNull();
```

trim()

This method truncates the BLOB to the new length specified.

Syntax

```
void trim(unsigned int newlen);
```

Parameters

newlen

The new length of the BLOB.

Valid values are:

Numbers less than or equal to the current length of the BLOB.

write()

This method writes data from a buffer into a BLOB. This method implicitly opens the BLOB, copies the buffer into the BLOB, and implicitly closes the BLOB. If the BLOB is already open, use `writeChunk()` instead. The actual number of bytes written is returned.

Syntax

```
unsigned int write(unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1);
```

Parameters

amt

The number of consecutive bytes to be written to the BLOB.

buffer

The buffer containing the data to be written to the BLOB.

bufsize

The size of the buffer containing the data to be written to the BLOB.

Valid values are:

Numbers greater than or equal to *`amt`*.

offset

The starting position at which to begin writing data into the BLOB. If `offset` is not specified, the data is written from the beginning of the BLOB.

Valid values are:

Numbers greater than or equal to 1.

writeChunk()

This method writes data from a buffer into a previously opened BLOB. The actual number of bytes written is returned.

Syntax

```
unsigned int writeChunk(unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1);
```

Parameters

amt

The number of consecutive bytes to be written to the BLOB.

buffer

The buffer containing the data to be written.

bufsize

The size of the buffer containing the data to be written.

Valid values are:

Numbers greater than or equal to *amt*.

offset

The starting position at which to begin writing data into the BLOB. If *offset* is not specified, the data is written from the beginning of the BLOB.

Valid values are:

Numbers greater than or equal to 1.

Bytes Class

Methods of the `Bytes` class enable you to perform specific tasks related to `Bytes` objects.

To create a `Bytes` object, use the syntax:

```
Bytes(Environment *env = NULL);
```

To create a `Bytes` object that contains a subarray of bytes from a character array, use the syntax:

```
Bytes(unsigned char *value,
    unsigned int count,
    unsigned int offset = 0,
    Environment *env = NULL);
```

To create a copy of a `Bytes` object, use the syntax:

```
Bytes(const Bytes &e);
```

Summary of Bytes Methods

Method	Summary
byteAt() on page 8-24	Return the byte at the specified position of the `Bytes` object.
getBytes() on page 8-25	Return a byte array from the `Bytes` object.
isNull() on page 8-26	Test whether the `Bytes` object is null.
length() on page 8-26	Return the number of bytes in the `Bytes` object.
setNull() on page 8-26	Set the `Bytes` object to null.

byteAt()

This method returns the byte at the specified position in the `Bytes` object.

Syntax

```
unsigned char byteAt(unsigned int index) const;
```

Parameters

index

The position of the byte to be returned from the `Bytes` object; the first byte of the `Bytes` object is at 0.

getBytes()

This method copies bytes from a `Bytes` object into the specified byte array.

Syntax

```
void getBytes(unsigned char *dst,
    unsigned int count,
    unsigned int srcBegin = 0,
    unsigned int dstBegin = 0) const;
```

Parameters

dst

The destination buffer into which data from the `Bytes` object is to be written.

count

The number of bytes to copy.

srcBegin

The starting position at which data is to be read from the `Bytes` object; the position of the first byte in the `Bytes` object is at 0.

dstBegin

The starting position at which data is to be written in the destination buffer; the position of the first byte in *`dst`* is at 0.

isNull()

This method tests whether the `Bytes` object is atomically null. If the `Bytes` object is atomically null, then true is returned; otherwise false is returned.

Syntax

```
bool isNull() const;
```

length()

This method returns the length of the `Bytes` object.

Syntax

```
unsigned int length() const;
```

setNull()

This method sets the `Bytes` object to atomically null.

Syntax

```
void setNull();
```

Clob Class

The Clob class defines the common properties of objects of type CLOB. A CLOB is a large character object stored as a column value in a row of a database table. A `Clob` object contains a logical pointer to a CLOB, not the CLOB itself.

Methods of the Clob class enable you to perform specific tasks related to `Clob` objects, including methods for getting the length of a SQL CLOB, for materializing a CLOB on the client, and for extracting a part of the CLOB.

Methods in the `ResultSet` and `Statement` classes, such as `getClob()` and `setClob()`, enable you to access an SQL CLOB value.

To create a null `Clob` object, use the syntax:

```
Clob();
```

The only methods valid on a null `Clob` object are `setNull()`, `isNull()`, and `operator=()`.

To create an uninitialized `Clob` object, use the syntax:

```
Clob(const Connection *connectionp);
```

An uninitialized `Clob` object can be initialized by:

- The `setEmpty()` method. The CLOB can then be modified by inserting this CLOB into the table and retrieving it using `SELECT` ... `FOR UPDATE`. The `write()` method will modify the CLOB; however, the modified data will be flushed to the table only when the transaction is committed. Note that an `insert` is not required.
- Assigning an initialized `Clob` object to it.

To create a copy of a `Clob` object, use the syntax:

```
Clob(const Clob &srcClob);
```

Summary of Clob Methods

Method	Summary
append() on page 8-29	Append a CLOB at the end of the current CLOB.
close() on page 8-29	Close a previously opened CLOB.
closeStream() on page 8-29	Close the `Stream` object obtained from the current CLOB.
copy() on page 8-29	Copy all or a portion of a CLOB or BFILE into the current CLOB.
getCharSetForm() on page 8-31	Return the character set form of the CLOB.
getCharSetId() on page 8-31	Return the character set ID of the CLOB.
getChunkSize() on page 8-31	Return the chunk size of the CLOB.
getStream() on page 8-31	Return data from the CLOB as a `Stream` object.
isInitialized() on page 8-32	Test whether the `Clob` object is initialized.
isNull() on page 8-32	Test whether the `Clob` object is atomically null.
isOpen() on page 8-33	Test whether the CLOB is open.
length() on page 8-33	Return the number of characters in the current CLOB.
open() on page 8-33	Open the CLOB with read or read-write access.
operator=() on page 8-33	Assign a CLOB locator to the current Clob object.
operator==() on page 8-34	Test whether two `Clob` objects are equal.
operator!=() on page 8-34	Test whether two `Clob` objects are not equal.
read() on page 8-35	Read a portion of the CLOB into a buffer.
setEmpty() on page 8-36	Set the `Clob` object to empty.
setNull() on page 8-36	Set the `Clob` object to atomically null.
trim() on page 8-36	Truncate the CLOB to a specified length.
write() on page 8-37	Write a buffer into an *unopened* CLOB.
writeChunk() on page 8-38	Write a buffer into an *open* CLOB.

append()

This method appends a CLOB to the end of the current CLOB.

Syntax

```
void append(const Clob &srcClob);
```

Parameters

srcClob

The CLOB to be appended to the current CLOB.

close()

This method closes a CLOB.

Syntax

```
void close();
```

closeStream()

This method closes the `Stream` object obtained from the CLOB.

Syntax

```
void closeStream(Stream *stream);
```

Parameters

stream

The `Stream` object to be closed.

copy()

This method copies a part or all of a BFILE or CLOB into the current CLOB.

Syntax

There are variants of syntax:

```
void copy(const Bfile &srcBfile,
    unsigned int numBytes,
    unsigned int dstOffset = 1,
    unsigned int srcOffset = 1);

void copy(const Clob &srcClob,
    unsigned int numBytes,
    unsigned int dstOffset = 1,
    unsigned int srcOffset = 1);
```

Parameters

srcBfile

The BFILE from which the data is to be copied.

srcClob

The CLOB from which the data is to be copied.

numBytes

The number of characters to be copied from the source BFILE or CLOB.

Valid values are:

Numbers greater than 0.

dstOffset

The starting position at which to begin writing data into the current CLOB.

Valid values are:

Numbers greater than or equal to 1.

srcOffset

The starting position at which to begin reading data from the source BFILE or CLOB.

Valid values are:

Numbers greater than or equal to 1.

getCharSetForm()

This method returns the character set form of the CLOB.

Syntax

```
CharSetForm getCharSetForm() const;
```

getCharSetId()

This method returns the character set ID of the CLOB.

Syntax

```
CharSet getCharSetId() const;
```

getChunkSize()

This method returns the chunk size of the CLOB.

When creating a table that contains a CLOB, the user can specify the chunking factor, which can be a multiple of Oracle blocks. This corresponds to the chunk size used by the LOB data layer when accessing and modifying the CLOB.

Syntax

```
unsigned int getChunkSize() const;
```

getStream()

This method returns a `Stream` object from the CLOB. If a stream is already open, it is disallowed to open another stream on `Clob` object, so the user must always close the stream before performing any `Clob` object operations.

Syntax

```
Stream* getStream(unsigned int offset = 1,
    unsigned int amount = 0,
    CharSet charsetId = DefaultCharSet,
    CharSetForm charsetForm = OCCI_SQLCS_IMPLICIT);
```

Parameter

offset

The starting position at which to begin reading data from the CLOB. If *`offset`* is not specified, the data is written from the beginning of the CLOB.

Valid values are:

Numbers greater than or equal to 1.

amount

The total number of consecutive characters to be read. If *`amount`* is 0, the data will be read from the *`offset`* value until the end of the CLOB.

charsetId

The character set ID of the CLOB.

charsetForm

The character set form of the CLOB.

isInitialized()

This method tests whether the `Clob` object is initialized. If the `Clob` object is initialized, true is returned; otherwise, false is returned.

Syntax

```
bool isInitialized() const;
```

isNull()

This method tests whether the `Clob` object is atomically null. If the `Clob` object is atomically null, true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

isOpen()

This method tests whether the CLOB is open. If the CLOB is open, true is returned; otherwise, false is returned.

Syntax

```
bool isOpen() const;
```

length()

This method returns the number of characters in the CLOB.

Syntax

```
unsigned int length() const;
```

open()

This method opens the CLOB in read-write or read-only mode.

Syntax

```
void open(LObOpenMode mode = OCCI_LOB_READWRITE);
```

Parameters

mode

The mode the CLOB is to be opened in.

Valid values are:

`OCCI_LOB_READWRITE`

`OCCI_LOB_READONLY`

operator=()

This method assigns a CLOB to the current CLOB. The source CLOB gets copied to the destination CLOB only when the destination CLOB gets stored in the table.

Syntax

```
Clob& operator=(const Clob &srcClob);
```

Parameters

srcClob

The CLOB to copy data from.

operator==()

This method compares two `Clob` objects for equality. Two `Clob` objects are equal if they both refer to the same CLOB. Two null `Clob` objects are not considered equal. If the `Blob` objects are equal, then true is returned; otherwise, false is returned.

Syntax

```
bool operator==(const Clob &srcClob) const;
```

Parameters

srcClob

The `Clob` object to be compared with the current `Clob` object.

operator!=()

This method compares two `Clob` objects for inequality. Two `Clob` objects are equal if they both refer to the same CLOB. Two null `Clob` objects are not considered equal. If the `Clob` objects are not equal, then true is returned; otherwise, false is returned.

Syntax

```
bool operator!=(const Clob &srcClob) const;
```

Parameters

srcClob

The `Clob` object to be compared with the current `Clob` object.

read()

This method reads a part or all of the CLOB into a buffer. The actual number of characters read is returned.

Syntax

```
unsigned int read(unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1,
    CharSet charsetId = DefaultCharSet,
    CharSetForm charsetForm = OCCI_SQLCS_IMPLICIT) const;
```

Parameters

amt
The number of consecutive characters to be read from the CLOB.

buffer
The buffer into which the CLOB data is to be read.

bufsize
The size of the buffer.

Valid values are:

Numbers greater than or equal to *amt*.

offset
The position at which to begin reading data from the CLOB. If *offset* is not specified, the data is read from the beginning of the CLOB.

Valid values are:

Numbers greater than or equal to 1.

charsetId
The character set ID of the CLOB.

charsetForm
The character set form of the CLOB.

setEmpty()

This method sets the `Clob` object to empty.

Syntax
```
void setEmpty();
```

setNull()

This method sets the `Clob` object to atomically null.

Syntax
```
void setNull();
```

trim()

This method truncates the CLOB to the new length specified.

Syntax
```
void trim(unsigned int newlen);
```

Parameters

newlen
The new length of the CLOB.

Valid values are:

Numbers less than or equal to the current length of the CLOB.

write()

This method writes data from a buffer into a CLOB. This method implicitly opens the CLOB, copies the buffer into the CLOB, and implicitly closes the CLOB. If the CLOB is already open, use `writeChunk()` instead. The actual number of characters written is returned.

Syntax

```
unsigned int write(unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1,
    CharSet charsetId = DefaultCharSet,
    CharSetForm charsetForm = OCCI_SQLCS_IMPLICIT);
```

Parameters

amt
The number of consecutive characters to be written to the CLOB.

buffer
The buffer containing the data to be written to the CLOB.

bufsize
The size of the buffer containing the data to be written to the CLOB.

Valid values are:

Numbers greater than or equal to *amt*.

offset
The position at which to begin writing data into the CLOB. If *offset* is not specified, the data is written from the beginning of the CLOB.

Valid values are:

Numbers greater than or equal to 1.

charsetId
The character set ID of the buffer data.

charsetForm

The character set form of the buffer data.

writeChunk()

This method writes data from a buffer into a previously opened CLOB. The actual number of characters written is returned.

Syntax

```
unsigned int writechunk(unsigned int amt,
    unsigned char *buffer,
    unsigned int bufsize,
    unsigned int offset = 1,
    CharSet charsetId = DefaultCharSet,
    CharSetForm charsetForm = OCCI_SQLCS_IMPLICIT);
```

Parameters

amt

The number of consecutive characters to be written to the CLOB.

buffer

The buffer containing the data to be written to the CLOB.

bufsize

The size of the buffer containing the data to be written to the CLOB.

Valid values are:

Numbers greater than or equal to *amt*.

offset

The position at which to begin writing data into the CLOB. If *offset* is not specified, the data is written from the beginning of the CLOB.

Valid values are:

Numbers greater than or equal to 1.

charsetId

The character set ID of the CLOB.

charsetForm

The character set form of the CLOB.

Connection Class

The **Connection** class represents a connection with a specific database. Within the context of a connection, SQL statements are executed and results are returned.

To create a connection, use the syntax:

```
Connection();
```

Summary of Connection Methods

Method	Description
changePassword() on page 8-41	Change the password for the current user.
commit() on page 8-41	Commit changes made since the previous commit or rollback and release any database locks held by the **session**.
createStatement() on page 8-41	Create a `Statement` object to execute SQL statements.
flushCache() on page 8-42	Flush the object cache associated with the connection.
getClientCharSet() on page 8-42	Return the default client character set.
getClientNCHARCharSet() on page 8-42	Return the default client NCHAR character set.
getMetaData() on page 8-42	Return the metadata for an object accessible from the connection.
getOCIServer() on page 8-43	Return the OCI server context associated with the connection.
getOCIServiceContext() on page 8-43	Return the OCI service context associated with the connection.
getOCISession() on page 8-43	Return the OCI session context associated with the connection
rollback() on page 8-43	Roll back all changes made since the previous commit or rollback and release any database locks held by the **session**.
terminateStatement() on page 8-44	Close a `Statement` object and free all resources associated with it.

changePassword()

This method changes the password of the user currently connected to the database.

Syntax

```
void changePassword(const string &user,
    const string &oldPassword,
    const string &newPassword);
```

Parameters

user

The user currently connected to the database.

oldPassword

The current password of the user.

newPassword

The new password of the user.

commit()

This method commits all changes made since the previous commit or rollback, and releases any database locks currently held by the session.

Syntax

```
void commit();
```

createStatement()

This method creates a `Statement` object with the SQL statement specified.

Syntax

```
Statement* createStatement(const string &sql ="");
```

Parameters

sql

flushCache()

This method flushes the object cache associated with the connection.

Syntax

```
void flushCache();
```

getClientCharSet()

This method returns the default client character set.

Syntax

```
CharSet getClientCharSet() const;
```

getClientNCHARCharSet()

This method returns the default client NCHAR character set.

Syntax

```
CharSet getClientNCHARCharSet() const;
```

getMetaData()

This method returns metadata for an object in the database.

Syntax

There are variants of syntax:

```
MetaData getMetaData(const string &object,
    MetaData::ParamType prmtyp = MetaData::PTYPE_UNK) const;
```

```
MetaData getMetaData(const RefAny &ref) const;
```

Parameters

object

prmtyp

ref

getOCIServer()

This method returns the OCI server context associated with the connection.

Syntax

```
OCIServer* getOCIServer() const;
```

getOCIServiceContext()

This method returns the OCI service context associated with the connection.

Syntax

```
OCISvcCtx* getOCIServiceContext() const;
```

getOCISession()

This method returns the OCI session context associated with the connection.

Syntax

```
OCISession* getOCISession() const;
```

rollback()

This method drops all changes made since the previous commit or rollback, and releases any database locks currently held by the session.

Syntax

```
void rollback();
```

terminateStatement()

This method closes a `Statement` object and frees all resources associated with it.

Syntax

```
void terminateStatement(Statement *statement);
```

Parameters

statement

The `Statement` to be closed.

ConnectionPool Class

The ConnectionPool class represents a pool of connections for a specific database.

To create a connection pool, use the syntax:

```
ConnectionPool();
```

Summary of ConnectionPool Methods

Method	Summary
createConnection() on page 8-46	Create a pooled connection.
createProxyConnection() on page 8-46	Create a proxy connection.
getBusyConnections() on page 8-47	Return the number of busy connections in the connection pool.
getIncrConnections() on page 8-47	Return the number of incremental connections in the connection pool.
getMaxConnections() on page 8-47	Return the maximum number of connections in the connection pool.
getMinConnections() on page 8-48	Return the minimum number of connections in the connection pool.
getOpenConnections() on page 8-48	Return the number of open connections in the connection pool.
getPoolName() on page 8-48	Return the name of the connection pool.
getTimeOut() on page 8-48	Return the time-out period for a connection in the connection pool.
setErrorOnBusy() on page 8-48	Specify that a SQLException is to be generated when all connections in the connection pool are busy and no further connections can be opened.
setPoolSize() on page 8-49	Set the minimum, maximum, and incremental number of pooled connections for the connection pool.
setTimeOut() on page 8-48	Set the time-out period, in seconds, for a connection in the connection pool.
terminateConnection() on page 8-50	Destroy the connection.

createConnection()

This method creates a pooled connection.

Syntax

```
Connection* createConnection(const string &userName,
    const string &password);
```

Parameters

userName

The name of the user to connect as.

password

The password of the user.

createProxyConnection()

This method creates a proxy connection from the connection pool.

Syntax

There are variants of syntax:

```
Connection* createProxyConnection(cont string &name,
    Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);
```

```
Connection* createProxyConnection(const string &name,
    string roles[],
    int numRoles,
    Connection::ProxyType proxyType = Connection::PROXY_DEFAULT);
```

Parameters

name

The user name to connect with.

roles

The roles to activate on the database server.

numRoles

The number of roles to activate on the database server.

proxyType

The type of proxy authentication to perform.

Valid values are:

PROXY_DEFAULT representing a database user name.

PROXY_EXTERNAL_AUTH representing an external user name.

getBusyConnections()

This method returns the number of busy connections in the connection pool.

Syntax

```
unsigned int getBusyConnections() const;
```

getIncrConnections()

This method returns the number of incremental connections in the connection pool.

Syntax

```
unsigned int getIncrConnections() const;
```

getMaxConnections()

This method returns the maximum number of connections in the connection pool.

Syntax

```
unsigned int getMaxConnections() const;
```

getMinConnections()

This method returns the minimum number of connections in the connection pool.

Syntax

```
unsigned int getMinConnections() const;
```

getOpenConnections()

This method returns the number of open connections in the connection pool.

Syntax

```
unsigned int getOpenConnections() const;
```

getPoolName()

This method returns the name of the connection pool.

Syntax

```
string getPoolName() const;
```

getTimeOut()

This method returns the time-out period of a connection in the connection pool.

Syntax

```
unsigned int getTimeOut() const;
```

setErrorOnBusy()

This method specifies that a SQLException is to be generated when all connections in the connection pool are busy and no further connections can be opened.

Syntax

```
void setErrorOnBusy();
```

setPoolSize()

This method sets the minimum, maximum, and incremental number of pooled connections for the connection pool.

Syntax

```
void setPoolSize(unsigned int minConn = 0,
    unsigned int maxConn = 1,
    unsigned int incrConn = 1);
```

Parameters

minConn

The minimum number of connections for the connection pool.

maxConn

The maximum number of connections for the connection pool.

incrConn

The incremental number of connections for the connection pool.

setTimeOut()

This method sets the time-out period for a connection in the connection pool. OCCI will terminate any connections related to this connection pool that have been idle for longer than the time-out period specified.

Syntax

```
void setTimeOut(unsigned int connTimeOut = 0);
```

Parameter

connTimeOut

The time-out period in number of seconds.

terminateConnection()

This method terminates the pooled connection or proxy connection.

Syntax

```
void terminateConnection(Connection *connection);
```

Parameter

connection

The pooled connection or proxy connection to terminate.

Date Class

The Date class specifies the abstraction for a SQL DATE data item. The Date class also adds formatting and parsing operations to support the OCCI escape syntax for date values.

Since SQL92 DATE is a subset of Oracle Date, this class can be used to support both.

To create a null `Date` object, use the syntax:

```
Date();
```

To create a copy of a `Date` object, use the syntax:

```
Date(const Date &a);
```

To create a `Date` object using integer parameters, where:

year	-4712 to 9999, except 0
month	1 to 12
day	1 to 31
minutes	0 to 59
seconds	0 to 59

use the syntax:

```
Date(const Environment *envp,
    int year = 1,
    unsigned int month = 1,
    unsigned int day = 1,
    unsigned int hour = 0,
    unsigned int minute = 0,
    unsigned int seconds = 0);
```

Objects from the Date class can be used as standalone class objects in client side numerical computations and also used to fetch from and set to the database.

The following code example demonstrates a Date column value being retrieved from the database, a bind using a Date object, and a computation using a standalone Date object:

```
/* Create a connection */
Environment *env = Environment::createEnvironment(Environment::DEFAULT);
Connection *conn = env->createConnection(user, passwd, db);

/* Create a statement and associate a DML statement to it */
string sqlStmt = "SELECT job-id, start_date from JOB_HISTORY
                            where end_date = :x";
Statement *stmt = conn->createStatement(sqlStmt);

/* Create a Date object and bind it to the statement */
Date edate(env, 2000, 9, 3, 23, 30, 30);
stmt->setDate(1, edate);
ResultSet *rset = stmt->executeQuery();

/* Fetch a date from the database */
while(rset->next())
{
    Date sd = rset->getDate(2);
    Date temp = sd;    /*assignment operator */
    /* Methods on Date */
    temp.getDate(year, month, day, hour, minute, second);
    temp.setMonths(2);
    IntervalDS inter = temp.daysBetween(sd);
    .
    .
    .
}
```

Summary of Date Methods

Method	Summary
addDays() on page 8-53	Return a `Date` object with *n* days added.
addMonths() on page 8-54	Return a `Date` object with *n* months added.
daysBetween() on page 8-54	Return the number of days between the current `Date` object and the date specified.
fromBytes() on page 8-54	Convert an external `Bytes` representation of a `Date` object to a `Date` object.

Method	Summary
fromText() on page 8-55	Convert the date from a given input string with format and nls parameters specified.
getDate() on page 8-55()	Return the date and time components of the `Date` object.
getSystemDate() on page 8-56	Return a `Date` object containing the system date.
isNull() on page 8-57	Returns true if `Date` is null; otherwise returns false.
lastDay() on page 8-57	Returns a `Date` that is the last day of the month.
nextDay() on page 8-57	Returns a `Date` that is the date of the next day of the week.
operator=() on page 8-57	Assign the values of `a` to the lvalue.
operator==() on page 8-58	Returns true if `a` and `b` are the same false otherwise.
operator!=() on page 8-58	Returns true if `a` and `b` are unequal false otherwise.
operator>() on page 8-59	Returns true if `a` is past `b`, false otherwise.
operator>=() on page 8-59	Returns true if `a` is past `b` or equal to `b`, false otherwise.
operator<() on page 8-60	Returns true is `a` is before `b`, false otherwise.
operator<=() on page 8-60	Returns true is `a` is before `b`, or equal to b false otherwise
setDate() on page 8-61	Set the date from the date components input
setNull() on page 8-61	Sets the object state to null.
toBytes() on page 8-61	Converts the `Date` object into an external `Bytes` representation.
toText() on page 8-62	Get the `Date` object as a string
toZone() on page 8-62	Return a `Date` object converted from one time zone to another.

addDays()

This method adds a specified number of days to the `Date` object and returns the new date.

Syntax

```
Date addDays(int i) const;
```

Parameters

i

The number of days to be added to the current `Date` object.

addMonths()

This method adds a specified number of months to the `Date` object and returns the new date.

Syntax

```
Date addMonths(int i) const;
```

Parameters

i

The number of months to be added to the current `Date` object.

daysBetween()

This method returns the number of days between the current `Date` object and the date specified.

Syntax

```
IntervalDS daysBetween(const Date &d) const;
```

Parameters

d

The date to be used.

fromBytes()

This method converts a `Bytes` object....

Syntax

```
void fromBytes(const Bytes &byteStream,
```

```
    const Environment *envp = NULL);
```

Parameters

byteStream
`Date` in external format in the form of Bytes.

envp

fromText()

This method converts a string...

Syntax
```
void fromText(const string &datestr,
    const string &fmt = "",
    const string &nlsParam = "",
    const Environment *envp = NULL);
```

Parameters

datestr
The date string to be converted.

fmt
The format string.

nlsParam
A string specifying the nls parameters to be used.

envp
The environment from which the nls parameters are to be obtained.

getDate()

This method returns the date in the form of the date components year, month, day, hour, minute, seconds.

Syntax

```
void getDate(int &year,
    unsigned int &month,
    unsigned int &day,
    unsigned int &hour,
    unsigned int &min,
    unsigned int &sec) const;
```

Parameters

year
Storage for year

month
Storage for month

day
Storage for day

hour
Storage for hour

min
Storage for minute

seconds
Storage for sec

getSystemDate()

This method returns the system date.

Syntax

```
static Date getSystemDate(const Environment *envp);
```

Parameters

envpr

isNull()

This method tests whether the `Date` is null. If the `Date` is null, true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

lastDay()

This method returns a date representing the last day of the current month.

Syntax

```
Date lastDay() const;
```

nextDay()

This method returns a date representing the day after the day of the week specified.

Syntax

```
Date nextDay(const string &dow) const;
```

Parameters

dow

Day Of Week.

operator=()

This method assigns ...

Syntax

```
Date& operator=(const Date &d);
```

Parameters

d

A date.

operator==()

This method compares the dates specified. If the dates are equal, true is returned; otherwise, false is returned.

Syntax

```
bool operator==(const Date &a,
    const Date &b);
```

Parameters

a

A date.

b

Another date.

operator!=()

This method compares the dates specified. If the dates are not equal then true is returned; otherwise, false is returned.

Syntax

```
bool operator!=(const Date &a,
    const Date &b);
```

Parameters

a

A date.

b

Another date.

operator>()

This method compares the dates specified. If the first date is in the future relative to the second date then true is returned; otherwise, false is returned. If either date is null then false is returned. If the dates are not the same type then false is returned.

Syntax

```
bool operator>(const Date &a,
    const Date &b);
```

Parameters

a

A date.

b

Another date.

operator>=()

This method compares the dates specified. If the first date is in the future relative to the second date or the dates are equal then true is returned; otherwise, false is returned. If either date is null then false is returned. If the dates are not the same type then false is returned.

Syntax

```
bool operator>=(const Date &a,
    const Date &b);
```

Parameters

a

A date.

b

Another date.

operator<()

This method compares the dates specified. If the first date is in the past relative to the second date then true is returned; otherwise, false is returned. If either date is null then false is returned. If the dates are not the same type then false is returned.

Syntax

```
bool operator<(const Date &a,
    const Date &b);
```

Parameters

a

A date.

b

Another date.

operator<=()

This method compares the dates specified. If the first date is in the past relative to the second date or the dates are equal then true is returned; otherwise, false is returned. If either date is null then false is returned. If the dates are not the same type then false is returned.

Syntax

```
bool operator<=(const Date &a,
    const Date &b);
```

Parameters

a

A date.

b

Another date.

setDate()

This method sets the date to the values specified.

Syntax

```
void setDate(int year = 1,
    unsigned int month = 1,
    unsigned int day = 1,
    unsigned int hour = 0,
    unsigned int minute = 0,
    unsigned int seconds = 0);
```

Parameters

year

month

day

hour

minute

second

setNull()

This method set the date to atomically null.

Syntax

```
void setNull();
```

toBytes()

This method returns the date in `Bytes` representation.

Syntax

```
Bytes toBytes() const;
```

toText()

This method returns a string with the value of this date formatted using *fmt* and *nlsParam*.

Syntax

```
string toText(const string &fmt = "",
    const string &nlsParam = "") const;
```

Parameters

fmt

The fmt string.

nlsParam

A string specifying the nls parameters to be used.

toZone()

This method returns `Date` value converted from one time zone to another. Valid time zones are:

AST, ADT	Atlantic Standard or Daylight Time
BST, BDT	Bering Standard or Daylight Time
CST, CDT	Central Standard or Daylight Time
EST, EDT	Eastern Standard or Daylight Time
GMT	Greenwich Mean Time
HST, HDT	Alaska-Hawaii Standard Time or Daylight Time
MST, MDT	Mountain Standard or Daylight Time
NST	Newfoundland Standard Time
PST, PDT	Pacific Standard or Daylight Time
YST, YDT	Yukon Standard or Daylight Time

Syntax

```
Date toZone(const string &zone1,
```

```
    const string &zone2) const;
```

Parameters

zone1
Time zone to be converted from.

zone2
Time zone to be converted to.

Environment Class

The Environment class provides an OCCI environment to manage memory and other resources for OCCI objects.

The application can have multiple OCCI environments. Each environment would have its own heap and thread-safety mutexes and so forth.

To create an `Environment` object, use the syntax:

```
Environment()
enum Mode
{
    DEFAULT = OCI_DEFAULT,
    OBJECT = OCI_OBJECT,
    SHARED = OCI_SHARED,
    NO_USERCALLBACKS = OCI_NO_UCB,
    THREADED_MUTEXED = OCI_THREADED,
    THREADED_UNMUTEXED = OCI_THREADED | OCI_ENV_NO_MUTEX
  };
```

Summary of Environment Methods

Method	Summary
createConnection() on page 8-65	Create a connection to a database.
createConnectionPool() on page 8-65	Create a connection pool.
createEnvironment() on page 8-66	Create an environment and use the specified memory management functions.
getCurrentHeapSize () on page 8-67	Return the current amount of memory allocated to all objects in the current environment.
getMap() on page 8-67()	Return the Map for the current environment.
getOCIEnvironment() on page 8-68	Return the OCI environment associated with the current environment.
terminateConnection () on page 8-68	Close the connection and free all related resources.
terminateConnectionPool() on page 8-68	Close the connection pool and free all related resources.

terminateEnvironment() on page 8-68	Destroy the environment.

createConnection()

This method establishes a connection to the database specified.

Syntax

```
Connection * createConnection(const string &username,
    const string &password,
    const string &connectString);
```

Parameter

username

The name of the user to connect as.

password

The password of the user.

connectString

The database to connect to.

createConnectionPool()

This method creates a connection pool based on the parameters specified.

Syntax

```
ConnectionPool* createConnectionPool(const string &poolUserName,
    const string &poolPassword,
    const string &connectString = "",
    unsigned int minConn = 0,
    unsigned int maxConn = 1,
    unsigned int incrConn = 1);
```

Parameters

poolUserName

poolPassword

connectString

minConn

maxConn

incrConn

createEnvironment()

This method creates an `Environment` object. It is created with the specified memory management functions specified in the `setMemMgrFunctions()` method. If no memory manager functions are specified, then OCCI uses its own default functions. An `Environment` object must eventually be closed to free all the system resources it has acquired.

If the mode specified is THREADED_MUTEXED or THREADED_UN_MUTEXED, then all three memory management functions must be thread-safe.

Syntax

```
static Environment * createEnvironment(Mode mode = DEFAULT,
    void *ctxp = 0,
    void *(*malocfp)(void *ctxp, size_t size) = 0,
    void *(*ralocfp)(void *ctxp, void *memptr, size_t newsize) = 0,
    void (*mfreefp)(void *ctxp, void *memptr) = 0);
```

Parameters

mode

Valid values are:

DEFAULT not thread safe, not in object mode

THREADED_MUTEXED thread safe, mutexed internally by OCCI

THREADED_UN-MUTEXTED thread safe, client responsible for mutexing

OBJECT uses object features

ctxp
Context pointer for user-defined memory management function.

size_t
The size of the memory to be allocated by user-defined memory allocation function.

malocfp
User-defined memory allocation function.

ralocfp
User-defined memory reallocation function.

mfreefp
User-defined memory free function.

getCurrentHeapSize ()

This method returns the amount of memory currently allocated to all objects in this environment.

Syntax
```
unsigned int getCurrentHeapSize() const;
```

getMap()

This method returns a pointer to the map for this environment.

Syntax
```
Map *getMap() const;
```

getOCIEnvironment()

This method returns a pointer to the OCI environment associated with this environment.

Syntax

```
OCIEnv *getOCIEnvironment() const;
```

terminateConnection ()

This method closes the connection to the environment, and frees all related system resources.

Syntax

```
void terminateConnection(Connection *connection);
```

Parameters

connection

terminateConnectionPool()

This method closes the connections in the connection pool, and frees all related system resources.

Syntax

```
void terminateConnectionPool(ConnectionPool *poolp);
```

Parameters

poolp

terminateEnvironment()

This method closes the environment, and frees all related system resources.

Syntax

```
void terminateEnvironment(Environment *env);
```

Parameter

env

Environment to be closed.

IntervalDS Class

Leading field precision will be determined by number of decimal digits in day input. Fraction second precision will be determined by number of fraction digits on input.

```
IntervalDS(const Environment *env,
    int day = 0,
    int hour = 0,
    int minute = 0,
    int second = 0,
    int fs = 0);
```

day

0 to 999999999

hour

0 to 23

minute

0 to 59

second

0 to 59

fs

0 to 999999

Constructs an `IntervalDS` object from the given string.The string is interpreted using the nls parameters set in the environment.

```
IntervalDS(const Environment *env,
    const string &inpstr);
```

inpstr

Input string representing a year month interval of the form 'days hours:minutes:seconds' for example, '10 20:14:10.2'

Constructs a null `IntervalDS` object. A null `intervalDS` can be initialized by assignment or calling `fromText` method. Methods that can be called on null `intervalDS` objects are `setNull` and `isNull`.

```
IntervalDS();
```

Constructs an IntervalDS object as a copy of an Interval reference.

```
IntervalDS(const Interval &src);
```

The following code example demonstrates that the default constructor creates a null value, and how you can assign a non null value to a day-second interval and then perform operations on it:

```
Environment *env = Environment::createEnvironment();

//create a null day-second interval
IntervalDS ds
if(ds.isnull())
    cout << "\n ds is null";

//assign a non null value to ds
IntervalDS anotherDS(env, "10-30");
ds = anotherDS;

//now all operations are valid on DS...
int DAY = ds.getDay();
```

The following code example demonstrates how to create a null day-second interval, initialize the day-second interval by using the `fromText` method, add to the day-second interval by using the += operator, multiply by using the * operator, compare 2 day-second intervals, and convert a day-second interval to a string by using the `toText` method:

```
Environment *env = Environment::createEnvironment();

//create a null day-second interval
IntervalDS ds1

//initialize a null day-second interval by using the fromText method
ds1.fromText("20 10:20:30.9","",env);

IntervalDS addWith(env,2,1);
ds1 += addWith;    //call += operator
```

```
IntervalDS mulDs1=ds1 * Number(env,10);    //call * operator
if(ds1==mulDs1)    //call == operator
    .
    .
    .;
string strds=ds1.toText(2,4);    //2 is leading field precision
                                 //4 is the fractional field precision
```

Summary of IntervalDS Methods

Method	Summary
fromText() on page 8-73	Return an IntervalDS with the value represented by instring.
getDay() on page 8-73	Return day interval values.
getFracSec() on page 8-74	Return fractional second interval values.
getHour() on page 8-74	Return hour interval values.
getMinute() on page 8-74	Return minute interval values.
getSecond() on page 8-74	Return second interval values.
isNull() on page 8-74	Return true if IntervalDS is null, false otherwise.
operator*() on page 8-75	Return the product of two IntervalDS values.
operator*=() on page 8-75	Multiplication assignment.
operator=() on page 8-75	Simple assignment.
operator==() on page 8-76	Check if a and b are equal.
operator!=() on page 8-76	Check if a and b are not equal.
operator/() on page 8-77	Return an IntervalDS with value (a / b).
operator/=() on page 8-77	Division assignment.
operator>() on page 8-77	Check if a is greater than b
operator>=() on page 8-78	Check if a is greater than or equal to b.
operator<() on page 8-80	Check if a is less than b.
operator<=() on page 8-79	Check if a is less than or equal to b.
operator-() on page 8-79	Return an IntervalDS with value (a - b).

Method	Summary
operator-=() on page 8-80	Subtraction assignment.
operator+() on page 8-80	Return the sum of two `IntervalDS` values.
operator+=() on page 8-80	Addition assignment.
set() on page 8-81	Set day-second interval.
setNull() on page 8-81	Set day-second interval to null.
toText() on page 8-82	

fromText()

This method creates the interval from the string specified. The string is converted using the nls parameters, if specified, associated with the relevant environment.

If *nlsParam* is specified, this will determine the nls parameters to be used for the conversion. If *nlsParam* is not specified, the nls parameters are picked up from *env*. If *env* is null, the nls parameters are picked up from the environment associated with the instance, if any.

Syntax

```
void fromText(const string &inpstr,
    const string &nlsParam = "",
    const Environment *env = NULL);
```

Parameters

inpstr

Input string representing a day second interval of the form 'days hours:minutes:seconds' for example, '10 20:14:10.2'

nlsParam

env

getDay()

This method returns the day component of the interval.

Syntax

```
int getDay() const;
```

getFracSec()

This method returns the fractional second component of the interval.

Syntax

```
int getFracSec() const;
```

getHour()

This method returns the hour component of the interval.

Syntax

```
int getHour() const;
```

getMinute()

This method returns the minute component of this interval.

Syntax

```
int getMinute() const;
```

getSecond()

This method returns the seconds component of this interval.

Syntax

```
int getSecond() const;
```

isNull()

This method tests whether the interval is null. If the interval is null then true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

operator*()

This method returns ...

Syntax

```
const IntervalDS operator*(const IntervalDS &a,
    const Number &factor);
```

Parameters

a
A day second interval.

factor

operator*=()

This method assigns the product of `IntervalDS` and `a` to `IntervalDS`.

Syntax

```
IntervalDS& operator*=(const IntervalDS &a);
```

Parameters

a
A day second interval.

operator=()

This method assigns the specified value to the interval.

Syntax

```
IntervalDS& operator=(const IntervalDS &src);
```

Parameters

src

operator==()

This method compares the intervals specified. If the intervals are equal then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator==(const IntervalDS &a,
    const IntervalDS &b);
```

Parameters

a

An day second interval.

b

Another day second interval.

operator!=()

This method compares the intervals specified. If the intervals are not equal then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator!=(const IntervalDS &a,
    const IntervalDS &b);
```

Parameters

a

An day second interval.

b

Another day second interval.

operator/()

This method returns ... .

Syntax

```
const IntervalDS operator/(const IntervalDS &a,
    const Number &factor);
```

Parameters

a

A day second interval.

factor

operator/=()

This method assigns the quotient of `IntervalDS` and `a` to `IntervalDS`.

Syntax

```
IntervalDS& operator/=(const IntervalDS &a);
```

Parameters

a

A day second interval.

operator>()

This method compares the intervals specified. If the first interval is greater than the second interval then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator>(const IntervalDS &a,
    const IntervalDS &b);
```

Parameters

a

A day second interval.

b

Another day second interval.

operator>=()

This method compares the intervals specified. If the first interval is greater than or equal to the second interval then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator>=(const IntervalDS &a,
    const IntervalDS &b);
```

operator<()

This method compares the intervals specified. If the first interval is less than the second interval then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator<(const IntervalDS &a,
    const IntervalDS &b);
```

Parameters

a

A day second interval.

b

Another day second interval.

operator<=()

This method compares the intervals specified. If the first interval is less than or equal to the second interval then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator<=(const IntervalDS &a,
    const IntervalDS &b);
```

Parameters

a

A day second interval.

b

Another day second interval.

operator-()

This method returns the difference between the intervals `a` and `b`.

Syntax

```
const IntervalDS operator-(const IntervalDS &a,
    const IntervalDS &b);
```

Parameters

a

A day second interval.

b

Another day second interval.

operator-=()

This method assigns the difference between `IntervalDS` and `a` to `IntervalDS`.

Syntax

```
IntervalDS& operator-=(const IntervalDS &a);
```

Parameters

a

A day second interval.

operator+()

This method returns the sum of the intervals specified.

Syntax

```
const IntervalDS operator+(const IntervalDS &a,
    const IntervalDS &b);
```

Parameters

a

A day second interval.

b

Another day second interval.

operator+=()

This method assigns the sum of `IntervalDS` and `a` to `IntervalDS`.

Syntax

```
IntervalDS& operator+=(const IntervalDS &a);
```

Parameters

a
A day second interval.

set()

This method sets the interval to the values specified.

Syntax

```
void set(int day,
    int hour,
    int minute,
    int second,
    int fracsec);
```

Parameters

day
0 to 999999999

hour
0 to 23

minute
0 to 59

second
0 to 59

fracsec
0 to 999999

setNull()

This method sets the interval to null.

Syntax

```
void setNull();
```

toText()

This method ...

Syntax

```
string toText(unsigned int lfprec,
    unsigned int fsprec,
    const string &nlsParam = "") const;
```

Parameters

lfprec

fsprec

nlsParam

IntervalYM Class

`IntervalYM` supports the SQL92 datatype Year-Month Interval.

Leading field precision will be determined by number of decimal digits on input.

```
IntervalYM(const Environment *env,
    int year = 0,
    int month=0);
```

year

0 to 999999999

month

0 to 11

Constructs an `IntervalYM` object from the given string. The string is interpreted using the nls parameters set in the environment.

```
IntervalYM(const Environment *env,
    const string &inpstr);
```

inpstr

Input string representing a year month interval of the form 'year-month'

Constructs a null `IntervalYM` object. A null intervalYM can be initialized by assignment or calling `fromText` method. Methods that can be called on null `intervalYM` objects are `setNull` and `isNull`.

```
IntervalYM();
```

Constructs an `IntervalYM` object from *`src`*.

```
IntervalYM(const IntervalYM &src);
```

The following code example demonstrates that the default constructor creates a null value, and how you can assign a non null value to a year-month interval and then perform operations on it:

```
Environment *env = Environment::createEnvironment();

//create a null year-month interval
```

```
IntervalYM ym
if(ym.isnull())
    cout << "\n ym is null";

//assign a non null value to ym
IntervalYM anotherYM(env, "10-30");
ym = anotherYM;

//now all operations are valid on ym...
int yr = ym.getYear();
```

The following code example demonstrates how to get the year-month interval column from a result set, add to the year-month interval by using the += operator, multiply by using the * operator, compare 2 year-month intervals, and convert a year-month interval to a string by using the `toText` method:

```
//SELECT WARRANT_PERIOD from PRODUCT_INFORMATION
//obtain result set
resultset->next();

//get interval value from resultset
IntervalYM ym1 = resultset->getIntervalYM(1);

IntervalYM addWith(env, 10, 1);
ym1 += addWith;    //call += operator

IntervalYM mulYm1 = ym1 * Number(env, 10);    //call * operator

if(ym1<mulYm1)    //comparison
    .
    .
    .;
string strym = ym1.toText(3);    //3 is the leading field precision
```

Summary of IntervalYM Methods

Method	Summary
fromText() on page 8-85	Return an `IntervalYM` with the value represented by `instring`.
getMonth() on page 8-86	Return month interval value.

Method	Summary
getYear() on page 8-86	Return year interval value.
isNull() on page 8-86	Check if the interval is null.
operator*() on page 8-87	Return the product of two `IntervalYM` values.
operator*=() on page 8-87	Multiplication assignment.
operator=() on page 8-87	Simple assignment.
operator==() on page 8-88	Check if a and b are equal.
operator!=() on page 8-88	Check if a and b are not equal.
operator/() on page 8-89	Return an interval with value (a / b).
operator/=() on page 8-89	Division assignment.
operator>() on page 8-90	Check if a is greater than b.
operator>=() on page 8-90	Check if a is greater than or equal to b.
operator<() on page 8-91	Check if a is less than b.
operator<=() on page 8-91	Check if a is less than or equal to b.
operator-() on page 8-92	Return an interval with value (a - b).
operator-=() on page 8-92	Subtraction assignment.
operator+() on page 8-92	Return the sum of two `IntervalYM` values.
operator+=() on page 8-93	Addition assignment.
set() on page 8-93	
setNull() on page 8-93	
toText() on page 8-94	

fromText()

This method creates the interval from the string specified. The string is converted using the nls parameters, if specified, associated with the relevant environment.

initializes the interval to the values in `inpstr`. The string is interpreted using the nls parameters set in the environment.

If *`nlsParam`* is specified, this will determine the nls parameters to be used for the conversion. If *`nlsParam`* is not specified, the nls parameters are picked up from

env. If *env* is null, the nls parameters are picked up from the environment associated with the instance, if any.

Syntax

```
void fromText(const string &inpstr,
    const string &nlsParam = "",
    const Environment *env = NULL);
```

Parameters

inpstr

Input string representing a year month interval of the form ‘year-month’

nlsParam

env

getMonth()

This method returns the month component of the interval.

Syntax

```
int getMonth() const;
```

getYear()

This method returns the year component of the interval.

Syntax

```
int getYear() const;
```

isNull()

This method tests whether the interval is null. If the interval is null then true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

operator*()

This method returns ...

Syntax

```
const IntervalYM operator*(const IntervalYM &a,
    const Number &factor);
```

Parameters

a

A year month interval.

factor

operator*=()

This method assigns the product of `IntervalYM` and `a` to `IntervalYM`.

Syntax

```
IntervalYM& operator*=(const IntervalYM &a);
```

Parameters

a

A year month interval.

operator=()

This method assigns the specified value to the interval.

Syntax

```
const IntervalYM& operator=(const IntervalYM &src);
```

Parameters

src

A year month interval.

operator==()

This method compares the intervals specified. If the intervals are equal then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator==(const IntervalYM &a,
    const IntervalYM &b);
```

Parameters

a

A year month interval.

b

Another year month interval.

operator!=()

This method compares the intervals specified. If the intervals are not equal then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator!=(const IntervalYM &a,
    const IntervalYM &b);
```

Parameters

a

A year month interval.

b

Another year month interval.

operator/()

This method returns ...

Syntax

```
const IntervalYM operator/(const IntervalYM &a,
    const Number &factor);
```

Parameters

a

A year month interval.

b

Another year month interval.

operator/=()

This method assigns the quotient of `IntervalYM` and `a` to `IntervalYM`.

Syntax

```
IntervalYM& operator/=(const IntervalYM &a);
```

Parameters

a

A year month interval.

operator>()

This method compares the intervals specified. If the first interval is greater than the second interval then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator>(const IntervalYM &a,
    const IntervalYM &b);
```

Parameters

a

A year month interval.

b

Another year month interval.

operator>=()

This method compares the intervals specified. If the first interval is greater than or equal to the second interval then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator>=(const IntervalYM &a,
    const IntervalYM &b);
```

Parameters

a

A year month interval.

b

Another year month interval.

operator<()

This method compares the intervals specified. If the first interval is less than the second interval then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator<(const IntervalYM &a,
    const IntervalYM &b);
```

Parameters

a

A year month interval.

b

Another year month interval.

operator<=()

This method compares the intervals specified. If the first interval is less than or equal to the second interval then true is returned; otherwise, false is returned. If either interval is null then false is returned.

Syntax

```
bool operator<=(const IntervalYM &a,
    const IntervalYM &b);
```

Parameters

a

A year month interval.

b

Another year month interval.

operator-()

This method returns the difference between the intervals specified.

Syntax

```
const IntervalYM operator-(const IntervalYM &a,
    const IntervalYM &b);
```

Parameters

a

A year month interval.

b

Another year month interval.

operator-=()

This method assigns the difference between `IntervalYM` and `a` to `IntervalYM`.

Syntax

```
IntervalYM& operator-=(const IntervalYM &a);
```

Parameters

a

A year month interval.

operator+()

This method returns the sum of the intervals specified.

Syntax

```
const IntervalYM operator+(const IntervalYM &a,
    const IntervalYM &b);
```

Parameters

a
A year month interval.

b
Another year month interval.

operator+=()

This method assigns the sum of `IntervalYM` and `a` to `IntervalYM`.

Syntax

```
IntervalYM& operator+=(const IntervalYM &a);
```

set()

This method sets the interval to the values specified.

Syntax

```
void set(int year,
    int month);
```

Parameters

year
0 to 99999999

month
0 to 11

setNull()

This method sets the interval to null.

Syntax

```
void setNull();
```

toText()

This method ...

Syntax

```
string toText(unsigned int lfprec,
    const string &nlsParam = "") const;
```

Parameters

lfprec

nlsParam

Map Class

The `Map` class is used to store the mapping of the SQL structured type to C++ classes.

For each user defined type, the Object Type Translator (OTT) generates a C++ class declaration and implements the static methods `readSQL()` and `writeSQL()`. The `readSQL()` method is called when the object from the server appears in the application as a C++ class instance. The `writeSQL()` method is called to marshal the object in the application cache to server data when it is being written / flushed to the server. The `readSQL()` and `writeSQL()` methods generated by OTT are based upon the OCCI standard C++ mappings.

If you want to override the standard OTT generated mappings with customized mappings, you must implement a custom C++ class along with the `readSQL()` and `writeSQL()` methods for each SQL structured type you need to customize. In addition, you must add an entry for each such class into the `Map` member of the `Environment`.

To..., use the syntax:

```
Map();
```

Summary of Map Methods

Method	Summary
put() on page 8-95	Adds a map entry for the type to be customized.

put()

This method adds a map entry for the type to be customized.

This method adds a map entry for the type, type_name, that you want to customize. You must implement the type_name C++ class along with its static methods, `readSQL()` and `writeSQL()`.

You must then add this information into a map object, which should be registered with the connection if the user wants the standard mappings to overridden.This registration can be done by calling the this method after the environment is created passing the environment.

Syntax

```
void put(const string&, void *(*)(void *),
    void (*)(void *, void *));
```

MetaData Class

A `MetaData` object can be used to describe the types and properties of the columns in a `ResultSet` or the existing schema objects in the database. It also provides information about the database as a whole.

Listing of the parameter types for objects:

- PTYPE_TABLE
- PTYPE_VIEW
- PTYPE_PROC
- PTYPE_FUNC
- PTYPE_PKG
- PTYPE_TYPE
- PTYPE_TYPE_ATTR
- PTYPE_TYPE_COLL
- PTYPE_TYPE_METHOD
- PTYPE_SYN
- PTYPE_SEQ
- PTYPE_COL
- PTYPE_ARG
- PTYPE_TYPE_ARG
- PTYPE_TYPE_RESULT
- PTYPE_SCHEMA
- PTYPE_DATABASE
- PTYPE_UNK

Listing of attribute values:

- DURATION_SESSION
- DURATION_TRANS
- DURATION_NULL

- TYPEENCAP_PRIVATE
- TYPEENCAP_PUBLIC
- TYPEPARAM_IN
- TYPEPARAM_OUT
- TYPEPARAM_INOUT
- CURSOR_OPEN
- CURSOR_CLOSED
- CL_START
- CL_END
- SP_SUPPORTED
- SP_UNSUPPORTED
- NW_SUPPORTED
- NW_UNSUPPORTED
- AC_DDL
- NO_AC_DDL
- LOCK_IMMEDIATE
- LOCK_DELAYED

These are returned on executing a get method passing some attribute for which these are the results.

To..., use the syntax:

```
MetaData(const MetaData &omd);
```

omd

The source metadata object to be copied from.

Summary of MetaData Methods

Method	Summary
getAttributeCount() on page 8-99	Gets the count of the attribute as a `MetaData` object

Method	Summary
getAttributeId() on page 8-99	Gets the ID of the specified attribute
getAttributeType() on page 8-100	Gets the type of the specified attribute.
getBoolean() on page 8-100	Gets the value of the attribute as a C++ boolean.
getInt() on page 8-100	Gets the value of the attribute as a C++ int.
getMetaData() on page 8-101	Gets the value of the attribute as a `MetaData` object
getNumber() on page 8-101	Returns the specified attribute as a `Number` object.
getRef() on page 8-101	Gets the value of the attribute as a `Ref<T>`.
getString() on page 8-102	Gets the value of the attribute as a string.
getTimeStamp() on page 8-102	Gets the value of the attribute as a `Timestamp` object
getUInt() on page 8-102	Gets the value of the attribute as a C++ unsigned int.
getVector() on page 8-103	Gets the value of the attribute as an C++ vector.
operator=() on page 8-103	Assigns one metadata object to another.

getAttributeCount()

This method returns the number of attributes related to the metadata object.

Syntax

```
unsigned int getAttributeCount() const;
```

getAttributeId()

This method returns the attribute ID (`ATTR_NUM_COLS, ...`) of the attribute represented by the attribute number specified.

Syntax

```
AttrId getAttributeId(unsigned int attributenum) const;
```

Parameters

attributenum

The number of the attribute for which the attribute ID is to be returned.

getAttributeType()

This method returns the attribute type (NUMBER, INT, . . .) of the attribute represented by attribute number specified.

Syntax

```
Type getAttributeType(unsigned int attributenum) const;
```

Parameters

attributenum

The number of the attribute for which the attribute type is to be returned.

getBoolean()

This method returns the value of the attribute as a C++ boolean. If the value is a SQL null, the result is false.

Syntax

```
bool getBoolean(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID.

getInt()

This method returns the value of the attribute as a C++ int. If the value is SQL null, the result is 0.

Syntax

```
int getInt(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID .

getMetaData()

This method returns a `MetaData` instance holding the attribute value. A metadata attribute value can be retrieved as a `MetaData` instance. This method can only be called on attributes of the metadata type.

Syntax

```
MetaData getMetaData(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID.

getNumber()

This method returns the value of the attribute as a `Number` object. If the value is a SQL null, the result is null.

Syntax

```
Number getNumber(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID.

getRef()

This method returns the value of the attribute as a `RefAny`. If the value is SQL null, the result is null.

Syntax

```
RefAny getRef(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID.

getString()

This method returns the value of the attribute as a string. If the value is SQL null, the result is null.

Syntax

```
string getString(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID.

getTimeStamp()

This method returns the value of the attribute as a `Timestamp` object. If the value is a SQL null, the result is null.

Syntax

```
Timestamp getTimestamp(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID.

getUInt()

This method returns the value of the attribute as a C++ unsigned int. If the value is a SQL null, the result is 0.

Syntax

```
unsigned int getUInt(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID.

getVector()

This method returns a C++ vector containing the attribute value. A collection attribute value can be retrieved as a C++ vector instance. This method can only be called on attributes of a list type.

Syntax

```
vector<MetaData> getVector(MetaData::AttrId attrid) const;
```

Parameters

attrid

The attribute ID.

operator=()

Syntax

```
void operator=(const MetaData &omd);
```

Parameters

omd

Number Class

The `Number` class handles limited-precision signed base 100 numbers. A `Number` guarantees 38 decimal digits of precision. All positive numbers in the range displayed here can be represented to a full 38-digit precision:

```
10^-130
```

and

```
9.99999999999999999999999999999999999999*10^125
```

The range of representable negative numbers is symmetrical.

The number zero can be represented exactly. Also, Oracle numbers have representations for positive and negative infinity. These are generally used to indicate overflow.

The internal storage type is opaque and private. Scale is not preserved when `Number` instances are created.

`Number` does not support the concept of NaN and is not IEEE-754-85 compliant. `Number` does support +Infinity and -Infinity.

There are several variants of syntax:

Default constructor.

```
Number();
```

```
Number(const Number &srcNum);
```

Translates a native long double into a `Number`. The `Number` is created using the precision of the platform-specific constant LDBL_DIG.

```
Number(long double val);
```

Translates a native double into a `Number`. The `Number` is created using the precision of the platform-specific constant DBL_DIG.

```
Number(double val);
```

Translates a native float into a `Number`. The `Number` is created using the precision of the platform-specific constant FLT_DIG.

```
Number(float val);
```

Translates a native long into a `Number`.

```
Number(long val);
```

Translates a native int into a `Number`.

```
Number(int val);
```

Translates a native short into a `Number`.

```
Number(short val);
```

Translates a native char into a `Number`.

```
Number(char val);
```

Translates an native unsigned long into a `Number`.

```
Number(unsigned long val);
```

Translates a native unsigned int into a `Number`.

```
Number(unsigned int val);
```

Translates a native unsigned short into a `Number`.

```
Number(unsigned short val);
```

Translates the unsigned character array into a `Number`.

```
Number(unsigned char val);
```

Objects from the `Number` class can be used as standalone class objects in client side numerical computations. They can also be used to fetch from and set to the database.

The following code example demonstrates a `Number` column value being retrieved from the database, a bind using a `Number` object, and a comparison using a standalone `Number` object:

```
/* Create a connection */
Environment *env = Environment::createEnvironment(Environment::DEFAULT);
Connection *conn = env->createConnection(user, passwd, db);

/* Create a statement and associate a select clause with it */
string sqlStmt = "SELECT department_id FROM DEPARTMENTS";
```

```
Statement *stmt = conn->createStatement(sqlStmt);

/* Execute the statement to get a result set */
ResultSet *rset = stmt->executeQuery();
while(rset->next())
{
    Number deptId = rset->getNumber(1);
    /* Display the department id with the format string 9,999 */
    cout << "Department Id" << deptId.toText(env, "9,999");

    /* Use the number obtained as a bind value in the following query */
    stmt->setSQL("SELECT * FROM EMPLOYEES WHERE department_id = :x");
    stmt->setNumber(1, deptId);
    ResultSet *rset2 = stmt->executeQuery();
    .
    .
    .
}
/* Using a Number object as a standalone and the operations on them */

/* Create a number to a double value */
double value = 2345.123;
Number nu1 (value);

/* Some common Number methods */
Number abs = nu1.abs();    /* absolute value */
Number sqrt = nu1.squareroot();    /* square root */

/* Cast operators can be used */
long lnum = (long) nu1;

/* Unary increment/decrement prefix/postfix notation */
nu1++;
--nu1;

/* Arithmetic operations */
Number nu2(nu1);

/* Assignment operators */
Number nu3;
nu3 = nu2;
nu2 = nu2 + 5.89;
Number nu4;
nu4 = nu1 + nu2;
```

```
/* Comparison operators */
if(nu1>nu2)
.
.
.
else if(nu1 == nu2)
.
.
.
```

Summary of Number Methods

Method	Summary
abs() on page 8-110	Return a `Number` whose value is the absolute value of the passed value.
arcCos() on page 8-110	Return a `Number` with value arcCosine of the passed value.
arcSin() on page 8-110	Return a `Number` with value arcSine of the passed value.
arcTan() on page 8-110	Return a `Number` with value arcTangent of the passed value.
arcTan2() on page 8-111	Return a `Number` with value `atan2(y,x)` where the passed value is y and n is associated with x.
ceil() on page 8-111	Return the smallest integral value not less than the value of the passed value.
cos() on page 8-111	Return a `Number` with value cosine of the passed value.
exp() on page 8-111	Return a `Number` with value `e` raised to the power.
floor() on page 8-112	Return the largest integral value not greater than the value of the passed value.
fromBytes() on page 8-112	Return a Number derived from a `Bytes` object.
fromText() on page 8-112	Return a `Number` derived from the input string `numstr` and the format string `fmt`.
hypCos() on page 8-113	Return a `Number` with value hyperbolic cosine of the passed value.
hypSin() on page 8-113	Return a `Number` with value hyperbolic sine of the passed value.

Method	Summary
hypTan() on page 8-113	Return a Number with value hyperbolic tangent of the passed value.
intPower() on page 8-113	Return a Number with the passed value raised to the n power.
isNull() on page 8-114	Check if Number is null.
ln() on page 8-114	Return a Number with value natural logarithm of the passed value.
log() on page 8-114	Return a Number with value logarithm the passed value with base n.
operator++() on page 8-114	Increment the internal value of Number by 1.
operator++() on page 8-115	Increment the internal value of Number by integer specified.
operator--() on page 8-115	Decrement the internal value of Number by 1.
operator--() on page 8-115	Decrement the internal value of Number by integer specified.
operator*() on page 8-121	Return the product of two Numbers.
operator/() on page 8-116	Return the quotient of two Numbers.
operator%() on page 8-116	Return the modulo of two Numbers.
operator+() on page 8-117	Return the sum of two Numbers.
operator-() on page 8-117	Return the negated value of Number .
operator-() on page 8-117	Return the difference between two Numbers.
operator<() on page 8-118	Check if a is less than b.
operator<=() on page 8-118	Check if a is less than or equal to b.
operator>() on page 8-119	Check if a is greater than b.
operator>=() on page 8-119	Check if a is greater or equal to b.
operator==() on page 8-120	Check if a and b are equal.
operator!=() on page 8-120	Check if a and b are not equal.
operator=() on page 8-121	Simple assignment.
operator*=() on page 8-121	Multiplication assignment.
operator/=() on page 8-122	Division assignment.

Method	Summary
operator%=() on page 8-122	Modulo assignment.
operator+=() on page 8-122	Addition assignment.
operator-=() on page 8-123	Subtraction assignment.
operator char() on page 8-123	Return `Number` converted to native char.
operator double() on page 8-123	Return `Number` converted to a native double.
operator float() on page 8-123	Return `Number` converted to a native float.
operator int() on page 8-124	Return `Number` converted to native integer.
operator long() on page 8-124	Return `Number` converted to native long.
operator long double() on page 8-124	Return `Number` converted to a native long double.
operator short() on page 8-124	Return `Number` converted to native short integer.
operator unsigned char() on page 8-125	Return `Number` converted to an unsigned native char.
operator unsigned int() on page 8-125	Return `Number` converted to an unsigned native integer.
operator unsigned long() on page 8-125	Return `Number` converted to an unsigned native long.
operator unsigned short() on page 8-125	Return `Number` converted to an unsigned native short integer.
power() on page 8-126	Return `Number` raised to the *n* power.
prec() on page 8-126	Return `Number` rounded to *n* digits of precision.
round() on page 8-126	Return `Number` rounded to decimal place *n*. Negative values are allowed.
setNull() on page 8-127	Set `Number` to null.
shift() on page 8-127	Return a `Number` that is equivalent to the passed value `* 10^n`, where `n` may be positive or negative.
sign() on page 8-127	Return the sign of the value of the passed value: -1 for the passed value < 0, 0 for the passed value == 0, and 1 for the passed value > 0.
sin() on page 8-127	Return a `Number` with value sine of the passed value.
sqareroot() on page 8-128	Return the square root of the passed value.

Method	Summary
tan() on page 8-128	Returns a Number with value tangent of the passed value.
toBytes() on page 8-128	Return a Bytes object representing the Number.
toText() on page 8-128	Return the value of the passed value as a string formatted based on the format *fmt*.
trunc() on page 8-129	Return a Number with the value truncated at n decimal place(s). Negative values are allowed.

abs()

This method returns the absolute value of the Number object.

Syntax

```
const Number abs() const;
```

arcCos()

This method returns the arccosine of the Number object.

Syntax

```
const Numberconst Number arcCos() const;
```

arcSin()

This method returns the arcsine of the Number object.

Syntax

```
const Number arcSin() const;
```

arcTan()

This method returns the arctangent of the Number object.

Syntax

```
const Number arcTan() const;
```

arcTan2()

This method returns the arctangent of the `Number` object divided by the value of the parameter specified.

Syntax

```
const Number arcTan2(const Number &val) const
```

Parameters

val

ceil()

This method returns the smallest integer that is greater than or equal to the `Number` object.

Syntax

```
const Number ceil() const;
```

cos()

This method returns the cosine of the `Number` object.

Syntax

```
const Number cos() const;
```

exp()

This method returns the exponential of the `Number` object.

Syntax

```
const Number exp() const;
```

floor()

This method returns the largest integer that is less than or equal to the `Number` object.

Syntax

```
const Number floor() const;
```

fromBytes()

This method returns a `Number` object represented by the byte string specified.

Syntax

```
void fromBytes(const Bytes &s);
```

Parameters

s

A byte string.

fromText()

This method returns a `Number` object derived from a string value.

Syntax

```
void fromText(const Environment *envp,
    const string &number,
    const string &fmt,
    const string &nlsParam = "");
```

Parameters

envp

The OCCI environment.

number

The number string to be converted to a `Number` object.

fmt

Format string.

nlsParam

The nls parameters string. If *nlsParam* is specified, this determines the nls parameters to be used for the conversion. If *nlsParam* is not specified, the nls parameters are picked up from *envp*.

hypCos()

This method returns the hypercosine of the `Number` object.

Syntax

```
const Number hypCos() const;
```

hypSin()

This method returns the hypersine of the `Number` object.

Syntax

```
const Number hypSin() const;
```

hypTan()

This method returns the hypertangent of the `Number` object.

Syntax

```
const Number hypTan() const;
```

intPower()

This method returns a `Number` whose value is (this^val).

Syntax

```
const Number intPower(int val) const;
```

Parameters

val

isNull()

This method tests whether the `Number` object is null. If the `Number` object is null, then true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

ln()

This method returns the natural logorithm of the `Number` object.

Syntax

```
const Number ln() const;
```

log()

This method returns the logorithm of the `Number` object with the base provided by the parameter specified.

Syntax

```
const Number log(const Number &val) const;
```

Parameters

val

The base to be used in the logorithm calculation.

operator++()

Unary `operator++()`. This method returns the `Number` object incremented by 1.

Syntax

```
Number& operator++();
```

operator++()

This method returns the `Number` object incremented by the integer specified.

Syntax

```
const Number operator++(int);
```

Parameters

int

operator--()

Unary `operator--()`. This method returns the `Number` object decremented by 1.

Syntax

```
Number& operator--();
```

operator--()

This method returns the `Number` object decremented by the integer specified.

Syntax

```
const Number operator--(int);
```

Parameters

int

operator*()

This method returns the product of the parameters specified.

Syntax

```
Number operator*(const Number &a,
    const Number &b);
```

Parameters

a

A number.

b

Another number.

operator/()

This method returns the quotient of the parameters specified.

Syntax

```
Number operator/(const Number &dividend,
    const Number &divisor);
```

Parameters

dividend

The number to be divided.

divisor

The number to divide by.

operator%()

This method returns the remainder of the division of the parameters specified.

Syntax

```
Number operator%(const Number &a,
    const Number &b);
```

Parameters

a
A number interval.

b
Another number interval.

operator+()

This method returns the sum of the parameters specified.

Syntax

```
Number operator+(const Number &a,
    const Number &b);
```

Parameters

a
A number.

b
Another number.

operator-()

Unary `operator-()`. This method returns the negated value of the `Number` object.

Syntax

```
const Number operator-();
```

operator-()

This method returns the difference between the parameters specified.

Syntax

```
Number operator-(const Number &subtrahend,
    const Number &subtractor);
```

Parameters

subtrahend
The number to be reduced.

subtractor
The number to be subtracted.

operator<()

This method checks whether the first parameter specified is less than the second parameter specified. If the first parameter is less than the second parameter, then true is returned; otherwise, false is returned. If either parameter is equal to infinity, then false is returned.

Syntax

```
bool operator<(const Number &a,
    const Number &b);
```

Parameters

a
A parameter of type `Number`.

b
Another parameter of type `Number`.

operator<=()

This method checks whether the first parameter specified is less than or equal to the second parameter specified. If the first parameter is less than or equal to the second parameter, then true is returned; otherwise, false is returned.If either `parameter` is equal to infinity, then false is returned.

Syntax

```
bool operator<=(const Number &a,
    const Number &b);
```

Parameters

a

A parameter of type `Number`.

b

Another parameter of type `Number`.

operator>()

This method checks whether the first parameter specified is greater than the second parameter specified. If the first parameter is greater than the second parameter, then true is returned; otherwise, false is returned. If either parameter is equal to infinity, then false is returned.

Syntax

```
bool operator>(const Number &a,
    const Number &b);
```

Parameters

a

A parameter of type `Number`.

b

Another parameter of type `Number`.

operator>=()

This method checks whether the first parameter specified is greater than or equal to the second parameter specified. If the first parameter is greater than or equal to the second parameter, then true is returned; otherwise, false is returned.If either parameter is equal to infinity, then false is returned.

Syntax

```
bool operator>=(const Number &a,
    const Number &b);
```

Parameters

a

A parameter of type `Number`.

b

Another parameter of type `Number`.

operator==()

This method checks whether the parameters specified are equal. If the parameters are equal, then true is returned; otherwise, false is returned. If either parameter is equal to +infinity or -infinity, then false is returned.

Syntax

```
bool operator==(const Number &a,
    const Number &b);
```

Parameters

a

A parameter of type `Number`.

b

Another parameter of type `Number`.

operator!=()

This method checks whether the first parameter specified is equal to the second parameter specified. If the parameters are not equal, true is returned; otherwise, false is returned.

Syntax

```
bool operator!=(const Number &a,
    const Number &b);
```

Parameters

a

A parameter of type `Number`.

b

Another parameter of type `Number`.

operator=()

This method assigns the value of the parameter specified to the `Number` object.

Syntax

```
Number& operator=(const Number &a);
```

Parameters

a

A parameter of type `Number`.

operator*=()

This method multiplies the `Number` object by the parameter specified, and assigns the product to the `Number` object.

Syntax

```
Number& operator*=(const Number &a);
```

Parameters

a

A parameter of type `Number`.

operator/=()

This method divides the `Number` object by the parameter specified, and assigns the quotient to the `Number` object.

Syntax

```
Number& operator/=(const Number &a);
```

Parameters

a

A parameter of type `Number`.

operator%=()

This method divides the `Number` object by the parameter specified, and assigns the remainder to the `Number` object.

Syntax

```
Number& operator%=(const Number &a);
```

Parameters

a

A parameter of type `Number`.

operator+=()

This method adds the Number object and the parameter specified, and assigns the sum to the `Number` object.

Syntax

```
Number& operator+=(const Number &a);
```

Parameters

a

A parameter of type `Number`.

operator-=()

This method subtracts the parameter specified from the `Number` object, and assigns the difference to the `Number` object.

Syntax

```
Number& operator-=(const Number &a);
```

Parameters

a

A parameter of type `Number`.

operator char()

This method returns the value of the Number object converted to a native char.

Syntax

```
operator char() const;
```

operator double()

This method returns the value of the `Number` object converted to a native double.

Syntax

```
operator double() const;
```

operator float()

This method returns the value of the `Number` object converted to a native float.

Syntax

```
operator float() const;
```

operator int()

This method returns the value of the `Number` object converted to a native int.

Syntax

```
operator int()const;
```

operator long()

This method returns the value of the `Number` object converted to a native long.

Syntax

```
operator long() const;
```

operator long double()

This method returns the value of the `Number` object converted to a native long double.

Syntax

```
operator long double() const;
```

operator short()

This method returns the value of the `Number` object converted to a native short integer.

Syntax

```
operator short() const;
```

operator unsigned char()

This method returns the value of the `Number` object converted to a native unsigned char.

Syntax

```
operator unsigned char() const;
```

operator unsigned int()

This method returns the value of the `Number` object converted to a native unsigned integer.

Syntax

```
operator unsigned int() const;
```

operator unsigned long()

This method returns the value of the `Number` object converted to a native unsigned long.

Syntax

```
operator unsigned long() const;
```

operator unsigned short()

This method returns the value of the `Number` object converted to a native unsigned short integer.

Syntax

```
operator unsigned short() const;
```

power()

This method returns the value of the `Number` object raised to the power provided by the parameter specified.

Syntax

```
const Number power(const Number &val) const;
```

Parameters

val

prec()

This method returns the value of the `Number` object rounded to the digits of precision provided by the parameter specified.

Syntax

```
const Number prec(int digits) const;
```

Parameters

digits

The number of digits of precision.

round()

This method returns the value of the `Number` object rounded to the decimal place provided by the parameter specified.

Syntax

```
const Number round(int decplace) const;
```

Parameters

decplace

The number of digits to the right of the decimal point.

setNull()

This method sets the value of the `Number` object to null.

Syntax

```
void setNull();
```

shift()

This method returns the `Number` object multiplied by 10 to the power provided by the parameter specified.

Syntax

```
const Number shift(int val) const;
```

Parameters

val

An integer value

sign()

This method returns the sign of the value of the `Number` object. If the `Number` object is negative, then -1 is returned. If the `Number` object is equal to 0, then 0 is returned. If the `Number` object is positive, then 1 is returned.

Syntax

```
const int sign() const;
```

sin()

This method returns the sin of the `Number` object.

Syntax

```
const Number sin();
```

sqareroot()

This method returns the square root of the `Number` object.

Syntax

```
const Number squareroot() const;
```

tan()

This method returns the tangent of the `Number` object.

Syntax

```
const Number tan() const;
```

toBytes()

This method converts the `Number` object into a `Bytes` object. The bytes representation is assumed to be in length excluded format, that is, the `Byte.length()` method gives the length of valid bytes and the 0th byte is the exponent byte.

Syntax

```
Bytes toBytes() const;
```

toText()

This method converts the `Number` object to a formatted string based on the parameters specified.

> **See Also:** "*Oracle9i SQL Reference Manual* for information on TO_CHAR.

Syntax

```
string toText(const Environment *envp,
    const string &fmt,
    const string &nlsParam = "") const;
```

Parameters

envp

The OCCI environment.

fmt

The format string.

nlsParam

The nls parameters string. If *nlsParam* is specified, this determines the nls parameters to be used for the conversion. If *nlsParam* is not specified, the nls parameters are picked up from *envp*.

trunc()

This method returns the `Number` object truncated at the number of decimal places provided by the parameter specified.

Syntax

```
const Number trunc(int decplace) const;
```

Parameters

decplace

The number of places to the right of the decimal place at which the value is to be truncated.

PObject Class

OCCI provides object navigational calls that enable applications to perform any of the following on objects:

- Creating, accessing, locking, deleting, copying, and flushing objects
- Getting references to the objects

This class enables the type definer to specify when a class is capable of having persistent or transient instances. Instances of classes derived from `PObject` are either persistent or transient. A class (called "A") that is persistent-capable inherits from the `PObject` class:

```
class A : PObject { ... }
```

Some of the methods provided, such as `lock()` and `refresh()`, are applicable only for persistent instances, not for transient instances.

To create a null `Pobject`, use the syntax:

```
PObject();
```

The only methods valid on a null `Pobject` are `setNull()`, `isNull`, and `operator=()`.

To create a copy of a `Pobject`, use the syntax:

```
PObject(const PObject& obj);
```

Summary of PObject Methods

Method	Summary
flush() on page 8-131	Flushes a modified persistent object to the database server.
getConnection() on page 8-131	Return the connection from which the `PObject` object was instantiated.
getRef() on page 8-131	Return a reference to a given persistent object.
isLocked() on page 8-132	Test whether the persistent object is locked.
isNull() on page 8-132	Test whether the object is null.

Method	Summary
lock() on page 8-132	Lock a persistent object on the database server. The default mode is to wait for the lock if not available.
markDelete() on page 8-133	Mark a persistent object as deleted.
markModified() on page 8-133	Mark a persistent object as modified or dirty.
operator=() on page 8-133	Assignment operator.
operator delete() on page 8-133	Remove the persistent object from the application cache only.
operator new() on page 8-134	Creates a new persistent / transient instance.
pin() on page 8-135	Pins an object.
setNull() on page 8-135	Sets the object value to null.
unmark() on page 8-135	Unmarks an object as dirty.
unpin() on page 8-135	Unpins an object. In the default mode, the pin count of the object is decremented by one.

flush()

This method flushes a modified persistent object to the database server.

Syntax

```
void flush();
```

getConnection()

This method returns the connection from which the persistent object was instantiated.

Syntax

```
const Connection *getConnection() const;
```

getRef()

This method returns a reference to the persistent object.

Syntax

```
RefAny getRef() const;
```

isLocked()

This method test whether the persistent object is locked. If the persistent object is locked, then true is returned; otherwise, false is returned.

Syntax

```
bool isLocked() const;
```

isNull()

This method tests whether the persistent object is null. If the persistent object is null, then true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

lock()

This method locks a persistent object on the database server.

Syntax

```
void lock(PObject::LockOption lock_option);
```

Parameters

lock_option

Specifies whether the lock operation should wait if the object is already locked by another user. The default value is `OCCI_LOCK_WAIT`, meaning the operation will wait.

Valid values are:

`OCCI_LOCK_WAIT`

`OCCI_LOCK_NOWAIT`

markDelete()

This method marks a persistent object as deleted.

Syntax

```
void markDelete();
```

markModified()

This method marks a persistent object as modified or dirty.

Syntax

```
void mark_Modified();
```

operator=()

This method assigns the value of a persistent object this `PObject` object. The nature (transient or persistent) of the object is maintained. Null information is copied from the source instance.

Syntax

```
PObject& operator=(const PObject& obj);
```

Parameters

obj

The object to copy from.

operator delete()

This method is used to delete a persistent or transient object. The delete operator on a persistent object removes the object from the application cache only. To delete the object from the database server, invoke the `markDelete()` method.

Syntax

```
void operator delete(void *obj,
    size_t size);
```

Parameters

obj

The object instance to be deleted.

size

operator new()

This method is used to create a new object. A persistent object is created if the connection and table name are provided. Otherwise, a transient object is created.

Syntax

There are variants of syntax:

```
void *operator new(size_t size);

void *operator new(size_t size,
    const Connection *x,
    const string& tablename,
    const char *type_name);
```

Parameters

size

x

The connection to the database in which the persistent object is to be created.

tablename

The name of the table in the database server.

type_name

The SQL type name corresponding to this C++ class. The format is `<schemaname>.<typename>`.

pin()

This method pins the object and increments the pin count by one. As long as the object is pinned, it will not be freed by the cache even if there are no references to this object instance.

Syntax

```
void pin();
```

setNull()

This method sets the object value to null.

Syntax

```
void setNull();
```

unmark()

This method unmarks a persistent object as modified or deleted.

Syntax

```
void unmark();
```

unpin()

This method unpins a persistent object. In the default mode, the pin count of the object is decremented by one. When this method is invoked with `OCCI_PINCOUNT_RESET`, the pin count of the object is reset.

If the pin count is reset, this method invalidates all the references (`Ref`) pointing to this object. The cache sets the object eligible to be freed, if necessary, reclaiming memory.

Syntax

```
void unpin(UnpinOption mode=OCCI_PINCOUNT_DECR);
```

Parameters

mode

Specifies whether the pin count should be decremented or reset to 0.

Valid values are:

`OCCI_PINCOUNT_RESET`

`OCCI_PINCOUNT_DECR`

Ref Class

The mapping in the C++ programming language of an SQL REF value, which is a reference to an SQL structured type value in the database.

Each REF value has a unique identifier of the object it refers to. An SQL REF value may be used in place of the SQL structured type it references; it may be used as either a column value in a table or an attribute value in a structured type.

Because an SQL REF value is a logical pointer to an SQL structured type, a Ref object is by default also a logical pointer; thus, retrieving an SQL REF value as a Ref object does not materialize the attributes of the structured type on the client.

A Ref object can be saved to persistent storage and is de-referenced through `operator*` or `operator->` or `ptr()` methods. T must be a class derived from `PObject`. In the following sections, T* and `PObject*` are used interchangeably.

To create a null `Ref` object, use the syntax:

```
Ref();
```

The only methods valid on a null `Ref` object are `isNull`, and `operator=()`.

To create a copy of a `Ref` object, use the syntax:

```
Ref(const Ref<T> &src);
```

Summary of Ref Methods

Method	Summary
clear() on page 8-138	Clears the reference.
getConnection() on page 8-138	Returns the connection this ref was created from.
getRef() on page 8-139	Returns the `Ref`.
isNull() on page 8-139	This method checks if the `Ref` is null.
markDelete() on page 8-139	Marks the referred object as deleted.
operator->() on page 8-139	De-reference the `Ref` and pins the object if necessary.
operator*() on page 8-140	This operator de-references the `Ref` and pins / fetches the object if necessary.
operator==() on page 8-140	Checks if the `Ref` and the pointer refer to the same object.
operator!=() on page 8-140	Checks if the `Ref` and the pointer refer to different objects.
operator=() on page 8-141	Assignment operator.
ptr() on page 8-141	De-references the `Ref` and pins / fetches the object if necessary.
setPrefetch() on page 8-141	Specifies type and depth of the object attributes to be followed for prefetching.
setLock() on page 8-142	Sets the lock option for the object referred from this.
unmarkDelete() on page 8-143	Unmarks for delete the object referred by this.

clear()

This method clears the `Ref` object.

Syntax

```
void clear();
```

getConnection()

This method returns the connection from which the `Ref` object was instantiated.

Syntax

```
const Connection *getConnection() const;
```

getRef()

This method returns the OCI Ref from the `Ref` object.

Syntax

```
OCIRef *getRef() const;
```

isNull()

This method test whether the `Ref` object is null. If the `Ref` object is null, then true is returned; otherwise, false is returned.

Syntax

```
bool isNull() const;
```

markDelete()

This method marks the referenced object as deleted.

Syntax

```
void markDelete();
```

operator->()

This method dereferences the `Ref` object and pins or fetches the referenced object if necessary. This might result in prefetching a graph of objects if prefetch attributes of the referenced object are set.

Syntax

There are variants of syntax:

```
T * operator->();
```

```
const T * operator->() const;
```

operator*()

This method dereferences the `Ref` object and pins or fetches the referenced object if necessary. This might result in prefetching a graph of objects if prefetch attributes of the referenced object are set.

The object does not need to be deleted. Destructor would be automatically called when it goes out of scope.

Syntax

There are variants of syntax:

```
T & operator *();
```

```
const T & operator*() const;
```

operator==()

This method tests whether two `Ref` objects are referencing the same object. If the `Ref` objects are referencing the same object, then true is returned; otherwise, false is returned.

Syntax

```
bool operator == (const Ref<T> &ref) const;
```

Parameters

ref

The `Ref` object of the object to be compared.

operator!=()

This method tests whether two `Ref` objects are referencing the same object. If the `Ref` objects are not referencing the same object, then true is returned; otherwise, false is returned.

Syntax

```
bool operator!= (const Ref<T> &ref) const;
```

Parameters

ref

The `Ref` object of the object to be compared.

operator=()

Assigns the ref or the object to a ref. For the first case, the refs are assigned and for the second case, the ref is constructed from the object and then assigned.

Syntax

There are variants of syntax:

```
Ref<T>& operator=(const Ref<T> &src);
```

```
Ref<T>& operator=(const T *obj);
```

Parameters

src

obj

ptr()

This operator dereferences the Ref and pins/fetches the object if necessary. This might result in prefetching a graph of objects if prefetch attributes of the Ref are set.

Syntax

There are variants of syntax:

```
T * ptr();
```

```
const T * ptr() const;
```

setPrefetch()

Sets the prefetching options for the complex object retrieval.

This method specifies depth up to which all objects reachable from this object through `Ref`s (transitive closure) should be prefetched. If only selected attribute types are to be prefetched, then `setPrefetch(type_name, depth)` should be used.

This method specifies which Ref attributes of the object it refers to should be followed for prefetching of the objects (complex object retrieval) and how many levels deep those links should be followed.

Syntax
There are variants of syntax:

```
void setPrefetch(const string &typeName,
    unsigned int depth);


void setPrefetch(unsigned int depth);
```

Parameters

typeName
Type of the Ref attribute to be prefetched.

depth
Depth level to which the links should be followed.

setLock()

This method specifies how the object should be locked when dereferenced.

Syntax
```
void setLock(LockOptions);
```

Parameters

LockOptions

unmarkDelete()

This method unmarks the referred object as dirty.

Syntax

```
void unmarkDelete();
```

RefAny Class

The `RefAny` class is designed to support a reference to any type. Its primary purpose is to handle generic references and allow conversions of `Ref` in the type hierarchy. A `RefAny` object can be used as an intermediary between any two types, `Ref<x>` and `Ref<y>`, where *x* and *y* are different types.

A `Ref<T>` can always be converted to a `RefAny`; there is a method to perform the conversion in the `Ref<T>` template. Each `Ref<T>` has a constructor and assignment operator that takes a reference to `RefAny`.

```
RefAny();
RefAny(const Connection *sessptr, const OCIRef *Ref);
RefAny(const RefAny& src);
```

Summary of RefAny Methods

Method	Summary
clear() on page 8-138	Clear the reference.
getConnection() on page 8-138	Return the connection this ref was created from.
getRef() on page 8-139	Return the `Ref`.
isNull() on page 8-139	Check if the `RefAny` object is null.
markDelete() on page 8-139	Mark the object as deleted.
operator=() on page 8-141	Assignment operator.
operator==() on page 8-140	Check if equal.
operator!=() on page 8-140	Check if not equal.
unmarkDelete() on page 8-143	Unmark the object as deleted.

clear()

This method clears the reference.

Syntax

```
void clear();
```

getConnection()

Returns the connection from which this ref was instantiated.

Syntax

```
const Connection * getConnection() const;
```

getRef()

Returns the underlying OCIRef *

Syntax

```
OCIRef* getRef() const;
```

isNull()

Returns true if the object pointed to by this ref is null else false.

Syntax

```
bool isNull() const;
```

markDelete()

This method marks the referred object as deleted.

Syntax

```
void markDelete();
```

operator=()

Assigns the ref or the object to a ref. For the first case, the refs are assigned and for the second case, the ref is constructed from the object and then assigned.

Syntax

```
RefAny& operator=(const RefAny& src);
```

Parameters

src

operator==()

Compares this ref with the Ref/RefAny object and returns true if both the refs are referring to the same object in the cache, otherwise it returns false.

Syntax

```
bool operator== (const RefAny &refAnyR) const;
```

Parameters

refAnyR

operator!=()

Compares this ref with the Ref/RefAny object and returns true if both the refs are not referring to the same object in the cache, otherwise it returns false.

Syntax

```
bool operator!= (const RefAny &refAnyR) const;
```

Parameters

refAnyR

unmarkDelete()

This method unmarks the referred object as dirty.

Syntax

```
void unmarkDelete();
```

ResultSet Class

A `ResultSet` provides access to a table of data generated by executing a `Statement`. Table rows are retrieved in sequence. Within a row, column values can be accessed in any order.

A `ResultSet` maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The `next` method moves the cursor to the next row.

The get ... () methods retrieve column values for the current row. You can retrieve values either using the index number of the column or the name of the column. In general, using the column index is more efficient. Columns are numbered beginning at 1.

For the get ... () methods, OCCI attempts to convert the underlying data to the specified C++ type and returns a C++ value.

SQL types are mapped to C++ types with the `ResultSet::get ... ()` methods.

The number, types and properties of a ResultSet's columns are provided by the MetaData object returned by the getColumnListMetaData method.

```
enum Status
{
    END_OF_FETCH = 0,
    DATA_AVAILABLE,
    STREAM_DATA_AVAILABLE
};
```

ResultSet()

This is the `ResultSet` constructor.

Syntax

```
ResultSet()
```

Summary of RefAny Methods

Method	Description
cancel() on page 8-150	Cancel the `ResultSet`.
closeStream() on page 8-150	Close the specified `Stream`.
getBfile() on page 8-150	Return the value of a column in the current row as a `Bfile`.
getBlob() on page 8-151	Return the value of a column in the current row as a `Blob` object.
getBytes() on page 8-151	Return the value of a column in the current row as a `Bytes` array.
getCharSet() on page 8-152	Return the character set in which data would be fetched.
getClob() on page 8-152	Return the value of a column in the current row as a `Clob` object.
getColumnListMetaData() on page 8-152	Return the describe information of the result set columns as a `MetaData` object.
getCurrentStreamColumn() on page 8-153	Return the column index of the current readable `Stream`.
getCurrentStreamRow() on page 8-153	Return the current row of the `ResultSet` being processed.
getCursor() on page 8-153	Return the nested cursor as a `ResultSet`.
getDate() on page 8-154	Return the value of a column in the current row as a `Date` object.
getDouble() on page 8-154	Return the value of a column in the current row as a C++ double.
getFloat() on page 8-154	Return the value of a column in the current row as a C++ float.
getInt() on page 8-155	Return the value of a column in the current row as a C++ int.
getIntervalDS() on page 8-155	Return the value of a column in the current row as a `IntervalDS`.
getIntervalYM() on page 8-155	Return the value of a column in the current row as a `IntervalYM`.

Method	Description
getMaxColumnSize() on page 8-156	Return the maximum amount of data to read from a column.
getNumArrayRows() on page 8-156	Return the actual number of rows fetched in the last array fetch when next(int numRows) returned `END_OF_DATA`.
getNumber() on page 8-156	Return the value of a column in the current row as a `Number` object.
getObject() on page 8-157	Return the value of a column in the current row as a `PObject`.
getRef() on page 8-157	Return the value of a column in the current row as a `Ref`.
getRowid() on page 8-158	Return the current ROWID for a SELECT FOR UPDATE statement.
getRowPosition() on page 8-158	Return the Rowid of the current row position.
getStatement() on page 8-158	Return the `Statement` of the `ResultSet`.
getStream() on page 8-158	Return the value of a column in the current row as a `Stream`.
getString() on page 8-159	Return the value of a column in the current row as a string.
getTimestamp() on page 8-159	Return the value of a column in the current row as a `Timestamp` object.
getUInt() on page 8-159	Return the value of a column in the current row as a C++ unsigned int
getVector() on page 8-160	Return the specified parameter as a vector.
isNull() on page 8-162	Check whether the value is null.
isTruncated() on page 8-162	Check whether truncation has occurred.
next() on page 8-163	Make the next row the current row in a `ResultSet`.
preTruncationLength() on page 8-164	
setBinaryStreamMode() on page 8-164	Specify that a column is to be returned as a binary stream.
setCharacterStreamMode() on page 8-164	Specify that a column is to be returned as a character stream.

Method	Description
setCharSet() on page 8-165	Specify the character set in which the data is to be returned.
setDataBuffer() on page 8-165	Specify the data buffer into which data is to be read.
setErrorOnNull() on page 8-166	Enable/disable exception when null value is read.
setErrorOnTruncate() on page 8-167	Enable/disable exception when truncation occurs.
setMaxColumnSize() on page 8-167	Specify the maximum amount of data to read from a column.
status() on page 8-168	Return the current status of the `ResultSet`.

cancel()

This method cancels the result set.

Syntax

```
void cancel();
```

closeStream()

This method closes the stream specified by the parameter *`stream`*.

Syntax

```
void closeStream(Stream *stream);
```

Parameters

stream

The stream to be closed.

getBfile()

This method returns the value of a column in the current row as a `Bfile`. Returns the column value; if the value is SQL null, the result is false.

Syntax

```
Bfile getBfile(unsigned int colIndex);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getBlob()

Get the value of a column in the current row as an `Blob`. Returns the column value; if the value is SQL null, the result is false.

Syntax

```
Blob getBlob(unsigned int colIndex);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getBytes()

Get the value of a column in the current row as a `Bytes` array. The bytes represent the raw values returned by the server. Returns the column value; if the value is SQL null, the result is null array

Syntax

```
Bytes getBytes(unsigned int colIndex);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getCharSet()

Get the character set in which data would be fetched.

Syntax

```
CharSet getCharSet(unsigned int colIndex) const;
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getClob()

Get the value of a column in the current row as a Clob. Returns the column value; if the value is SQL null, the result is false.

Syntax

```
Clob getClob(unsigned int colIndex);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getColumnListMetaData()

The number, types and properties of a ResultSet's columns are provided by the getMetaData method. Returns the description of a ResultSet's columns. This method will return the value of the given column as a PObject. The type of the C++ object will be the C++ PObject type corresponding to the column's SQL type registered with Environment's map. This method is used to materialize data of SQL user-defined types.

Syntax

```
vector<MetaData> getColumnListMetaData() const;
```

getCurrentStreamColumn()

If the result set has any input `Stream` parameters, this method returns the column index of the current input `Stream` that must be read. If no output `Stream` needs to be read, or there are no input `Stream` columns in the result set, this method returns 0. Returns the column index of the current input `Stream` column that must be read.

Syntax

```
unsigned int getCurrentStreamColumn() const;
```

getCurrentStreamRow()

If the result has any input `Stream`s, this method returns the current row of the result set that is being processed by OCCI. If this method is called after all the rows in the set of array of rows have been processed, it returns 0. Returns the row number of the current row that is being processed. The first row is numbered 1 and so on.

Syntax

```
unsigned int getCurrentStreamRow() const;
```

getCursor()

Get the nested cursor as an `ResultSet`. Data can be fetched from this result set. A nested cursor results from a nested query with a CURSOR(SELECT ...) clause.

```
SELECT ename, CURSOR(SELECT  dname, loc FROM dept) FROM emp WHERE ename =
'JONES'
```

Note that if there are multiple `REF CURSOR`s being returned, data from each cursor must be completely fetched before retrieving the next `REF CURSOR` and starting fetch on it. Returns A `ResultSet` for the nested cursor.

Syntax

```
ResultSet * getCursor(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getDate()

Get the value of a column in the current row as a `Date` object. Returns the column value; if the value is SQL null, the result is null

Syntax

```
Date getDate(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getDouble()

Gets the value of a column in the current row as a C++ double. Returns the column value; if the value is SQL null, the result is 0

Syntax

```
double getDouble(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getFloat()

Get the value of a column in the current row as a C++ float. Returns the column value; if the value is SQL null, the result is 0.

Syntax

```
float getFloat(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getInt()

Get the value of a column in the current row as a C++ int. Returns the column value; if the value is SQL null, the result is 0.

Syntax
```
int getInt(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getIntervalDS()

Get the value of a column in the current row as a `IntervalDS` object. Returns the column value; if the value is SQL null, the result is null.

Syntax
```
IntervalDS getIntervalDS(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getIntervalYM()

Get the value of a column in the current row as a `IntervalYM` object. Returns the column value; if the value is SQL null, the result is null

Syntax
```
IntervalYM getIntervalYM(unsigned int colIndex);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getMaxColumnSize()

Get the maximum amount of data to read for a column.

Syntax

```
unsigned int getMaxColumnSize(unsigned int colIndex) const;
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getNumArrayRows()

Returns the actual number of rows fetched in the last array fetch when next(int numRows) returned `END_OF_DATA`. Returns the actual number of rows fetched in the final array fetch

Syntax

```
unsigned int getNumArrayRows() const;
```

getNumber()

Get the value of a column in the current row as a `Number` object. Returns the column value; if the value is SQL null, the result is null

Syntax

```
Number getNumber(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getObject()

Returns a pointer to a `PObject` holding the column value.

Syntax
```
PObject * getObject(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getRef()

Get the value of a column in the current row as a `RefAny`. Retrieving a `Ref` value does not materialize the data to which `Ref` refers. Also the `Ref` value remains valid while the session or connection on which it is created is open. Returns a `RefAny` holding the column value.

Syntax
```
RefAny getRef(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getRowid()

Get the current rowid for a `SELECT` ... `FOR UPDATE` statement. The rowid can be bound to a prepared `DELETE` statement and so on. Returns Current rowid for a `SELECT` ... `FOR UPDATE` statement.

Syntax

```
Bytes getRowid(unsigned int colIndex);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getRowPosition()

Get the Rowid of the current row position.

Syntax

```
Bytes getRowPosition() const
```

getStatement()

This method returns the `Statement` of the `ResultSet`.

Syntax

```
const Statement* getStatement() const;
```

getStream()

This method returns the value of a column in the current row as a `Stream`.

Syntax

```
Stream * getStream(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getString()

Get the value of a column in the current row as a string. Returns the column value; if the value is SQL null, the result is an empty string.

Syntax

```
string getString(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getTimestamp()

Get the value of a column in the current row as a Timestamp object. Returns the column value; if the value is SQL null, the result is null.

Syntax

```
Timestamp getTimestamp(unsigned int colIndex);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

getUInt()

Get the value of a column in the current row as a C++ int. Returns the column value; if the value is SQL null, the result is 0.

Syntax

```
unsigned int getUInt(unsigned int colIndex);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

getVector()

This method returns the attribute in the current position as a vector of objects. It retrieves the column in the specified position as a vector of `RefAny`. The attribute at the current position should be a collection type (`varray` or nested table). The SQL type of the elements in the collection should be compatible with objects.

Syntax

There are variants of syntax:

```
void getVector(ResultSet *rs,
    unsigned int index,
    vector<int> &vect);


void getVector(ResultSet *rs,
    unsigned int index,
    vector<string> &vect);


void getVector(ResultSet *rs,
    unsigned int index,
    vector<T *> &vect);


void getVector(ResultSet *rs,
    unsigned int,
    vector<unsigned int> &vect);


void getVector(ResultSet *rs,
    unsigned int,
    vector<float> &vect);


void getVector(ResultSet *rs,
```

```
    unsigned int,
    vector<double> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<Date> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<Timestamp> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<RefAny> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<Blob> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<Clob> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<Bfile> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<Number> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<IntervalDS> &vect);

void getVector(ResultSet *rs,
    unsigned int,
    vector<IntervalYM> &vect);
```

```
void getVector(ResultSet *rs,
    unsigned int,
    vector<Ref<T>> &vect);
```

Parameters

rs
The result set.

vect
The reference to the vector of objects (OUT parameter).

isNull()

A column may have the value of SQL null; `wasNull()` reports whether the last column read had this special value. Note that you must first call get*xxx* on a column to try to read its value and then call `wasNull()` to find if the value was the SQL null. Returns true if last column read was SQL null.

Syntax

```
bool isNull(unsigned int colIndex) const;
```

Parameters

colIndex
The first column is 1, the second is 2, ...

isTruncated()

This method checks whether the value of the parameter is truncated. If the value of the parameter is truncated, then true is returned; otherwise, false is returned.

Syntax

```
bool isTruncated(unsigned int paramIndex) const;
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

next()

A `ResultSet` is initially positioned before its first row; the first call to next makes the first row the current row; the second call makes the second row the current row, and so on. If a read-able stream from the previous row is open, it is implicitly closed. The `ResultSet`'s warning chain is cleared when a new row is read.

For non-streamed mode, next() always returns `RESULT_SET_AVAILABLE` or `END_OF_DATA`. Data is available for `get`*`xxx`* method when the `RESULT_SET_AVAILABLE` status is returned. When this version of `next()` is used, array fetches are done for data being fetched with the `setDataBuffer()` interface. This means that `get`*`xxx`*`()` methods should not be called. The `numRows` amount of data for each column would materialize in the buffers specified with the `setDataBuffer()` interface. With array fetches, stream input is allowed, so `get`*`xxx`*`Stream()` methods can also be called (once for each column).

Returns one of following:

- `DATA_AVAILABLE` — call `get`*`xxx`*`()` or read data from buffers specified with `setDataBuffer()`
- `END_OF_FETCH` — no more data available. This is the last set of rows for array fetches. This value is defined to be 0.
- `STREAM_DATA_AVAILABLE` — call the `getCurrentStreamColumn` method and read stream

Syntax

```
Status next(unsigned int numRows =1);
```

Parameters

numRows
Number of rows to fetch for array fetches

preTruncationLength()

Syntax

```
int preTruncationLength(unsigned int paramIndex) const;
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

setBinaryStreamMode()

Defines that a column is to be returned as a binary stream by the `getStream` method.

Syntax

```
void setBinaryStreamMode(unsigned int colIndex,
    unsigned int size);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

size

setCharacterStreamMode()

Defines that a column is to be returned as a character stream by the `getStream` method.

Syntax

```
void setCharacterStreamMode(unsigned int colIndex,
    unsigned int size);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

size

setCharSet()

Overrides the default character set for the specified column. Data is converted from the database character set to the specified character set for this column.

Syntax

```
void setCharSet(unsigned int colIndex,
    CharSet charSet);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

charSet
Desired character set.

setDataBuffer()

Specify a data buffer where data would be fetched. The *buffer* parameter is a pointer to a user allocated data buffer. The current length of data must be specified in the **length* parameter. The amount of data should not exceed the *size* parameter. Finally, *type* is the data type of the data. Only non OCCI and non C++ specific types can be used i.e STL string, OCCI classes like `Bytes` and `Date` cannot be used.

If `setDataBuffer()` is used to fetch data for array fetches, it should be called only once for each result set. Data for each row is assumed to be at `buffer + (i - 1)*size` location where *i* is the row number. Similarly the length of the data would be assumed to be at `*(length + (i - 1))`.

Syntax

```
void setDataBuffer(unsigned int colIndex,
    void *buffer,
    Type type,
    sb4 size = 0,
    ub2 *length = NULL,
    sb2 *ind = NULL,
    ub2 *rc = NULL);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

buffer
Pointer to user-allocated buffer; if array fetches are done, it should have `numRows` * size bytes in it

type
Type of the data that is provided (or retrieved) in the buffer

size
Size of the data buffer; for array fetches, it is the size of each element of the data items

length
Pointer to the length of data in the buffer; for array fetches, it should be an array of length data for each buffer element; the size of the array should be equal to `arrayLength`

ind

rc

setErrorOnNull()

This method enables/disables exceptions for reading of null values on `colIndex` column of the result set.

Syntax

```
void setErrorOnNull(unsigned int colIndex,
    bool causeException);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

causeExcetpion
Enable exceptions if true. Disable if false

setErrorOnTruncate()

This method enables/disables exceptions when truncation occurs.

Syntax

```
void setErrorOnTruncate(unsigned int paramIndex,
    bool causeException);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

causeException
Enable exceptions if true. Disable if false

setMaxColumnSize()

Set the maximum amount of data to read for a column

Syntax

```
void setMaxColumnSize(unsigned int colIndex,
    unsigned int max);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

max

status()

Returns the current status of the result set. The status method can be called repeatedly to find out the status of the result. Data is available for `get`*`xxx`* method when the `RESULT_SET_AVAILABLE` status is returned. Returns one of following:

- `DATA_AVAILABLE` — call `get`*`xxx`*`()` or read data from buffers specified with the `setDataBuffer` method
- `STREAM_DATA_AVAILABLE` — call `getCurrentStream()` and read stream
- `END_OF_FETCH`

Syntax

```
Status status() const;
```

SQLException Class

The `SQLException` class provides information on generated errors, their codes and associated messages.

SQLException()

This is the `SQLException` constructor.

Syntax

There are variants of syntax:

```
SQLException();
SQLException(const SQLException &e);
```

Summary of SQLException Methods

Method	Summary
getErrorCode() on page 8-169	Return the database error code.
getMessage() on page 8-169	Return the error message string for this exception.
setErrorCtx() on page 8-170	Set the error context.

getErrorCode()

Gets the database error code.

Syntax

```
int getErrorCode() const;
```

getMessage()

Returns the error message string of this `SQLException` if it was created with an error message string. Returns null if the `SQLException` was created with no error message.

Syntax

```
string getMessage() const;
```

setErrorCtx()

Sets the pointer to the error context.

Syntax

```
void setErrorCtx(void *ctx);
```

Parameters

ctx

The pointer to the error context.

Statement Class

A `Statement` object is used for executing SQL statements. The statement may be a query returning result set or a non-query statement returning an update count.

Non-query SQL can be insert, update, or delete statements. Non-query SQL statements can also be DDL statements (such as create, grant, and so on) or stored procedure calls.

A query, insert / update / delete, or stored procedure call statements may have IN bind parameters. A DML returning insert / update / delete statement or stored procedure call may have OUT bind parameters. Finally, a stored procedure call statement may have bind parameters that are both IN and OUT, referred to as IN/OUT parameters.

The `Statement` class methods are divided into three categories:

- `Statement` methods applicable to all statements
- Methods applicable to prepared statements with IN bind parameters
- Methods applicable to callable statements and DML returning statements with OUT bind parameters.

To..., use the syntax:

```
Statement()
enum Status
  {
    UNPREPARED,
    PREPARED,
    RESULT_SET_AVAILABLE,
    UPDATE_COUNT_AVAILABLE,
    NEEDS_STREAM_DATA,
    STREAM_DATA_AVAILABLE
  };
```

Summary of Statement Methods

Method	Description
addIteration() on page 8-175	Add an iteration for execution.

Method	Description
closeResultSet() on page 8-175	Immediately releases a result set's database and OCCI resources instead of waiting for automatic release.
closeStream() on page 8-176	Close the stream specified by the parameter *stream*.
execute() on page 8-176	Execute the SQL statement.
executeArrayUpdate() on page 8-177	Execute insert/update/delete statements which use only the setDataBuffer() or stream interface for bind parameters.
executeQuery() on page 8-179	Execute a SQL statement that returns a single ResultSet.
executeUpdate() on page 8-179	Execute a SQL statement that does not return a ResultSet.
getAutoCommit() on page 8-179	Return the current auto-commit state.
getBfile() on page 8-180	Return the value of a BFILE as a Bfile object.
getBlob() on page 8-180	Return the value of a BLOB as a Blob object.
getBytes() on page 8-180	Return the value of a SQL BINARY or VARBINARY parameter as Bytes.
getCharSet() on page 8-181	Return the character set that is in effect for the specified parameter.
getClob() on page 8-181	Return the value of a CLOB as a Clob object.
getConnection() on page 8-181	
getCurrentIteration() on page 8-182	Return the iteration number of the current iteration that is being processed.
getCurrentStreamIteration() on page 8-182	Return the current iteration for which stream data is to be read or written.
getCurrentStreamParam() on page 8-182	Return the parameter index of the current output Stream that must be read or written.
getCursor() on page 8-182	Return the REF CURSOR value of an OUT parameter as a ResultSet.
getDatabaseNCHARParam() on page 8-183	Return whether data is in NCHAR character set.
getDate() on page 8-183	Return the value of a parameter as a Date object
getDouble() on page 8-184	Return the value of a parameter as a C++ double.
getFloat() on page 8-184	Return the value of a parameter as a C++ float.

Method	Description
getInt() on page 8-184	Return the value of a parameter as a C++ int.
getIntervalDS() on page 8-185	Return the value of a parameter as a IntervalDS object.
getIntervalYM() on page 8-185	Return the value of a parameter as a IntervalYM object.
getMaxIterations() on page 8-186	Return the current limit on maximum number of iterations.
getMaxParamSize() on page 8-186	Return the current max parameter size limit.
getNumber() on page 8-186	Return the value of a parameter as a Number object.
getObject() on page 8-187	Return the value of a parameter as a PObject.
getOCIStatement() on page 8-187	Return the OCI statement handle associated with the Statement.
getRef() on page 8-187	Return the value of a REF parameter as RefAny
getResultSet() on page 8-188	Return the current result as a ResultSet.
getRowid() on page 8-188	Return the rowid param value as a Bytes object.
getSQL() on page 8-188	Return the current SQL string associated with the Statement object.
getStream() on page 8-188	Return the value of the parameter as a stream.
getString() on page 8-189	Return the value of the parameter as a string.
getTimestamp() on page 8-189	Return the value of the parameter as a Timestamp object
getUInt() on page 8-189	Return the value of the parameter as a C++ unsigned int
getUpdateCount() on page 8-190	Return the current result as an update count for non-query statements.
getVector() on page 8-160	Return the specified parameter as a vector.
isNull() on page 8-192	Check whether the parameter is null.
isTruncated() on page 8-193	Check whether the value is truncated.
preTruncationLength() on page 8-193	
registerOutParam() on page 8-193	Register the type and max size of the OUT parameter.

Method	Description
setAutoCommit() on page 8-194	Specify auto commit mode.
setBfile() on page 8-195	Set a parameter to a Bfile value.
setBinaryStreamMode() on page 8-195	Specify that a column is to be returned as a binary stream.
setBlob() on page 8-196	Set a parameter to a Blob value.
setBytes() on page 8-196	Set a parameter to a Bytes array.
setCharacterStreamMode() on page 8-197	Specify that a column is to be returned as a character stream.
setCharSet() on page 8-197	Specify the character set for the specified parameter.
setClob() on page 8-197	Set a parameter to a Clob value.
setDate() on page 8-198	Set a parameter to a Date value.
setDatabaseNCHARParam() on page 8-198	Set to true if the data is to be in the NCHAR character set of the database; set to false to restore the default.
setDataBuffer() on page 8-199	Specify a data buffer where data would be available for reading or writing.
setDataBufferArray() on page 8-200	Specify an array of data buffers where data would be available for reading or writing.
setDouble() on page 8-202	Set a parameter to a C++ double value.
setErrorOnNull() on page 8-203	Enable/disable exceptions for reading of null values.
setErrorOnTruncate() on page 8-203	Enable/disable exception when truncation occurs.
setFloat() on page 8-204	Set a parameter to a C++ float value.
setInt() on page 8-204	Set a parameter to a C++ int value.
setIntervalDS() on page 8-204	Set a parameter to a IntervalDS value.
setIntervalYM() on page 8-205	Set a parameter to a IntervalYM value.
setMaxIterations() on page 8-205	Set the maximum number of invocations that will be made for the DML statement.
setMaxParamSize() on page 8-206	Set the maximum amount of data that can sent or returned from the parameter.
setNull() on page 8-206	Set a parameter to SQL null.
setNumber() on page 8-207	Set a parameter to a Number value.

Method	Description
setObject() on page 8-207	Set the value of a parameter using an object.
setPrefetchMemorySize() on page 8-208	Set the amount of memory that will be used internally by OCCI to store data fetched during each round trip to the server.
setPrefetchRowCount() on page 8-208	Set the number of rows that will be fetched internally by OCCI during each round trip to the server.
setRef() on page 8-209	Set a parameter to a `RefAny` value.
setRowid() on page 8-209	Set a row id bytes array for a bind position.
setSQL() on page 8-210	Associate a new SQL string with a `Statement` object.
setString() on page 8-210	Set a parameter to an string value.
setTimestamp() on page 8-210	Set a parameter to a `Timestamp` value.
setUInt() on page 8-211	Set a parameter to a C++ unsigned int value.
setVector() on page 8-211	Set a parameter to a vector of unsigned int.
status() on page 8-214	Return the current status of the statement. This is useful when there is streamed data to be written.

addIteration()

After specifying set parameters, an iteration is added for execution.

Syntax

```
void addIteration();
```

closeResultSet()

In many cases, it is desirable to immediately release a result set's database and OCCI resources instead of waiting for this to happen when it is automatically closed; the `closeResultSet` method provides this immediate release.

Syntax

```
void closeResultSet(ResultSet *resultSet);
```

Parameters

resultSet

closeStream()

Closes the stream specified by the parameter `stream`.

Syntax

```
void closeStream(Stream *stream);
```

Parameters

stream

The stream to be closed.

execute()

Executes a SQL statement that may return either a result set or an update count. The statement may have read-able streams which may have to be written, in which case the results of the execution may not be readily available. The returned value is one of the following:

- `UNPREPARED`
- `PREPARED`
- `RESULT_SET_AVAILABLE`
- `UPDATE_COUNT_AVAILABLE`
- `NEEDS_STREAM_DATA`
- `STREAM_DATA_AVAILABLE`

If `RESULT_SET_AVAILABLE` is returned, the `getResultSet()` method must be called to get the result set.

If `UPDATE_COUNT_AVAILABLE` is returned, the `getUpdateCount` method must be called to find out the update count.

If `NEEDS_STREAM_DATA` is returned, output `Streams` must be written for the streamed `IN` bind parameters. If there is more than one streamed parameter, call the `getCurrentStreamParam` method to find out the bind parameter needing the

stream. If the statement is executed iteratively, call `getCurrentIteration` to find out the iteration for which stream needs to be written.

If `STREAM_DATA_AVAILABLE` is returned, input `Streams` must be read for the streamed `OUT` bind parameters. If there is more than one streamed parameter, call the `getCurrentStreamParam` method to find out the bind parameter needing the stream. If the statement is executed iteratively, call `getCurrentIteration` to find out the iteration for which stream needs to be read.

If only one `OUT` value is returned for each invocation of the DML returning statement, iterative executes can be performed for DML returning statements. If output streams are used for `OUT` bind variables, they must be completely read in order. The `getCurrentStreamParam` method would indicate which stream needs to be read. Similarly, `getCurrentIteration` would indicate the iteration for which data is available.

Returns

- `RESULT_SET_AVAILABLE`-- call `getResultSet()`
- `UPDATE_COUNT_AVAILABLE` -- call `getUpdateCount()`
- `NEEDS_STREAM_DATA` -- call `getCurrentStream()` and `getCurrentIteration()`, and `write` (or `read`) stream

Syntax

```
Status execute(const string &sql = "");
```

Parameters

sql

executeArrayUpdate()

Executes insert/update/delete statements which use only the setDataBuffer() or stream interface for bind parameters. The bind parameters must be arrays of size arrayLength parameter. The statement may have read-able streams which may have to be written. The returned value is one of the following:

- `UPDATE_COUNT_AVAILABLE`
- `NEEDS_STREAM_DATA`
- `STREAM_DATA_AVAILABLE`

- PREPARED
- UNPREPARED

If UPDATE_COUNT_AVAILABLE is returned, getUpdateCount() must be called to find out the update count.

If NEEDS_STREAM_DATA is returned, output Streams must be written for the streamed bind parameters. If there is more than one streamed parameter, getCurrentStreamParam() can be called to find out the bind parameter needing the stream. The getCurrentIteration() can be called to find out the iteration for which stream needs to be written.

If STREAM_DATA_AVAIALBE is returned, input Streams must be read for the streamed OUT bind parameters. If there is more than one streamed parameter, getCurrentStreamParam() can be called to find out the bind parameter needing the stream. If the statement is executed iteratively, getCurrentIteration() can be called to find out the iteration for which stream needs to be read.

If only one OUT value is returned for each invocation of the DML returning statement, array executes can be done for DML returning statements also. If output streams are used for OUT bind variables, they must be completely read in order. The getCurrentStreamParam() method would indicate which stream needs to be read. Similarly, getCurrentIteration() would indicate the iteration for which data is available.

Note that you cannot perform array executes for queries or callable statements.

Syntax

```
Status executeArrayUpdate(unsigned int arrayLength);
```

Parameters

arrayLength

The number of elements provided in each buffer of bind variables. The statement is executed this many times with each array element used for each iteration. Returns:

- UPDATE_COUNT_AVAILABLE -- call getUpdateCount()
- NEEDS_STREAM_DATA -- call getCurrentStream() and getCurrentIteration(), and write (or read) stream

executeQuery()

Execute a SQL statement that returns a `ResultSet`. Should not be called for a statement which is not a query, has streamed parameters. Returns a `ResultSet` that contains the data produced by the query

Syntax

```
ResultSet * executeQuery(const string &sql = "");
```

Parameters

sql

executeUpdate()

Executes a non-query statement such as a SQL `INSERT`, `UPDATE`, `DELETE` statement, a DDL statement such as `CREATE/ALTER` and so on, or a stored procedure call. Returns either the row count for `INSERT`, `UPDATE` or `DELETE` or 0 for SQL statements that return nothing

Syntax

```
unsigned int executeUpdate(const string &sql = "");
```

Parameters

sql

getAutoCommit()

Get the current auto-commit state. Returns Current state of auto-commit mode.

Syntax

```
bool getAutoCommit() const;
```

getBfile()

Get the value of a `BFILE` parameter as a `Bfile` object. Returns the parameter value.

Syntax

```
Bfile getBfile(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getBlob()

Get the value of a `BLOB` parameter as a `Blob`. Returns the parameter value

Syntax

```
Blob getBlob(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getBytes()

Get the value of a SQL BINARY or `VARBINARY` parameter as `Bytes`. Returns the parameter value; if the value is SQL null, the result is null.

Syntax

```
Bytes getBytes(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getCharSet()

Returns the character set that is in effect for the specified parameter.

Syntax

```
CharSet getCharSet(unsigned int paramIndex) const;
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getClob()

Get the value of a `CLOB` parameter as a `Clob`. Returns the parameter value.

Syntax

```
Clob getClob(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getConnection()

Syntax

```
const Connection* getConnection() const;
```

getCurrentIteration()

If the prepared statement has any output `Streams`, this method returns the current iteration of the statement that is being processed by OCCI. If this method is called after all the invocations in the set of iterations has been processed, it returns 0. Returns the iteration number of the current iteration that is being processed. The first iteration is numbered 1 and so on. If the statement has finished execution, a 0 is returned.

Syntax

```
unsigned int getCurrentIteration() const;
```

getCurrentStreamIteration()

Returns the current param stream for which data is available

Syntax

```
unsigned int getCurrentStreamIteration() const;
```

getCurrentStreamParam()

If the prepared statement has any output `Stream` parameters, this method returns the parameter index of the current output `Stream` that must be written. If no output `Stream` needs to be written, or there are no output `Stream` parameters in the prepared statement, this method returns 0.

Returns the parameter index of the current output `Stream` parameter that must be written.

Syntax

```
unsigned int getCurrentStreamParam() const;
```

getCursor()

Get the `REF CURSOR` value of an `OUT` parameter as a `ResultSet`. Data can be fetched from this result set. The `OUT` parameter must be registered as `CURSOR` with the `OCCICallableStatement.registerOutParameter(int paramIndex,`

CURSOR) method. Note that if there are multiple REF CURSORs being returned due to a batched call, data from each cursor must be completely fetched before retrieving the next REF CURSOR and starting fetch on it. Returns A ResultSet for the OUT parameter value.

Syntax

```
ResultSet * getCursor(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getDatabaseNCHARParam()

Get whether this data is for an NCHAR parameter. Returns whether data is in NCHAR character set. Returns true if NCHAR parameter; false otherwise

Syntax

```
bool getDatabaseNCHARParam(unsigned int paramIndex) const;
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getDate()

Get the value of a SQL DATE parameter as a Date object. Returns the parameter value; if the value is SQL null, the result is null.

Syntax

```
Date getDate(unsigned int paramIndex) const;
```

Parameters

paramIndex

the first parameter is 1, the second is 2, ...

getDouble()

Get the value of a DOUBLE parameter as a C++ double. Returns the parameter value; if the value is SQL null, the result is 0.

Syntax

```
double getDouble(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getFloat()

Get the value of a FLOAT parameter as a C++ float. Returns the parameter value; if the value is SQL null, the result is 0.

Syntax

```
float getFloat(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getInt()

Get the value of an INTEGER parameter as a C++ int. Returns the parameter value; if the value is SQL null, the result is 0

Syntax

```
unsigned int getInt(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getIntervalDS()

Get the value of a parameter as a `IntervalDS` object.

Syntax

```
IntervalDS getIntervalDS(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getIntervalYM()

Get the value of a parameter as a `IntervalYM` object.

Syntax

```
IntervalYM getIntervalYM(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getMaxIterations()

Gets the current limit on maximum number of iterations. Default is 1. Returns the current maximum number of iterations.

Syntax

```
unsigned int getMaxIterations() const;
```

getMaxParamSize()

The `maxParamSize` limit (in bytes) is the maximum amount of data sent or returned for any parameter value; it only applies to character and binary types. If the limit is exceeded, the excess data is silently discarded. Returns the current max parameter size limit

Syntax

```
unsigned int getMaxParamSize(unsigned int paramIndex) const;
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getNumber()

Get the value of a `NUMERIC` parameter as a `Number` object. Returns the parameter value; if the value is SQL nullnull, the result is null.

Syntax

```
Number getNumber(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getObject()

Get the value of a parameter as a `PObject`. This method returns an `PObject` whose type corresponds to the SQL type that was registered for this parameter using `registerOutParameter`. Note that this method may be used to read database-specific, abstract data types. Returns A `PObject` holding the `OUT` parameter value.

Syntax

```
PObject * getObject(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getOCIStatement()

Get the OCI statement handle associated with the `Statement`. Returns the OCI statement handle associated with the `Statement`

Syntax

```
OCIStmt * getOCIStatement() const;
```

getRef()

Get the value of a `REF` parameter as `RefAny`. Returns the parameter value.

Syntax

```
RefAny getRef(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getResultSet()

Returns the current result as a `ResultSet`.

Syntax

```
ResultSet * getResultSet();
```

getRowid()

Get the `rowid` param value as a `Bytes`

Syntax

```
Bytes getRowid(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getSQL()

Returns the current SQL string associated with the Statement object.

Syntax

```
string getSQL() const;
```

getStream()

Syntax

```
Stream * getStream(unsigned int paramIndex);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

getString()

Get the value of a `CHAR`, `VARCHAR`, or `LONGVARCHAR` parameter as an string. Returns the parameter value; if the value is SQL null, the result is empty string.

Syntax
```
string getString(unsigned int paramIndex);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

getTimestamp()

Get the value of a SQL TIMESTAMP parameter as a Timestamp object. Returns the parameter value; if the value is SQL null, the result is null

Syntax
```
Timestamp getTimestamp(unsigned int paramIndex);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

getUInt()

Get the value of a `BIGINT` parameter as a C++ unsigned int. Returns the parameter value; if the value is SQL null, the result is 0

Syntax

```
unsigned int getUInt(unsigned int paramIndex);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

getUpdateCount()

Returns the current result as an update count.

Syntax

```
unsigned int getUpdateCount() const;
```

getVector()

This method returns the attribute in the current position as a vector of objects. It retrieves the column in the specified position as a vector of `RefAny`. The attribute at the current position should be a collection type (`varray` or nested table). The SQL type of the elements in the collection should be compatible with objects.

Syntax

There are variant of syntax:

```
void getVector(Statement *stmt,
    unsigned int index,
    vector<int> &vect;
```

```
void getVector(Statement *stmt,
    unsigned int index,
    vector<string> &vect;
```

```
void getVector(Statement *stmt,
    unsigned int index,
    vector<T *> &vect;
```

```
void getVector(Statement *stmt,
```

```
    unsigned int,
    vector<unsigned int> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<float> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<double> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<Date> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<Timestamp> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<RefAny> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<Blob> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<Clob> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<Bfile> &vect;

void getVector(Statement *stmt,
    unsigned int,
    vector<Number> &vect;
```

```
void getVector(Statement *stmt,
    unsigned int,
    vector<IntervalDS> &vect;
```

```
void getVector(Statement *stmt,
    unsigned int,
    vector<IntervalYM> &vect;
```

```
void getVector(Statement *stmt,
    unsigned int,
    vector<Ref<T>> &vect;
```

Parameters

rs

The result set.

vect

The reference to the vector of objects (OUT parameter).

isNull()

An `OUT` parameter may have the value of SQL null; `wasNull` reports whether the last value read has this special value. Note that you must first call `getXXX` on a parameter to read its value and then call wasNull() to see if the value was SQL null. Returns `true` if the last parameter read was SQL null

Syntax

```
bool isNull(unsigned int paramIndex ) const;
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

isTruncated()

This method checks whether the value of the parameter is truncated. If the value of the parameter is truncated, then true is returned; otherwise, false is returned.

Syntax

```
bool isTruncated(unsigned int paramIndex) const;
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

preTruncationLength()

Syntax

```
int preTruncationLength(unsigned int paramIndex) const;
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

registerOutParam()

This method registers the `type` of each `out` paramater of a PL/SQL stored procedure. Before executing a PL/SQL stored procedure, you must explicitly call this method to register the `type` of each `out` parameter. This method should be called for `out` parameters only. Use the `set`*`xxx`* method for in/out parameters.

When reading the value of an `out` parameter, you must use the `get`*`xxx`* method that corresponds to the parameter's registered SQL type. For example, use `getInt` or `getNumber` when `OCCIINT` or `OCCINumber` is the type specified.

If a PL/SQL stored procedure has an `out` paramater of `type ROWID`, the `type` specified in this method should be `OCCISTRING`. The value of the `out` parameter can then be retrieved by calling the `getString()` method.

If a PL/SQL stored procedure has an in/out parameter of type `ROWID`, call the methods `setString()` and `getString()` to set the type and retrieve the value of the in/out parameter.

Syntax

```
void registerOutParam(unsigned int paramIndex,
    Type type,
    unsigned int maxSize = 0,
    const string &sqltype = "");
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

type

SQL type code defined by *`type`*; only datatypes corresponding to OCCI data types such as `Date`, `Bytes`, and so on.

maxSize

The maximum size of the retrieved value. For datatypes of `OCCIBYTES` and `OCCISTRING`, *`maxSize`* should be greater than 0.

sqltype

The name of the type in the data base (used for types which have been created with `CREATE TYPE`)

setAutoCommit()

A `Statement` can be in auto-commit mode. In this case any statement executed is also automatically committed. By default, the auto-commit mode is turned-off.

Syntax

```
void setAutoCommit(bool autoCommit);
```

Parameters

autoCommit
True enables auto-commit; false disables auto-commit.

setBfile()

Set a parameter to a `Bfile` value.

Syntax

```
void setBfile(unsigned int paramIndex,
    const Bfile &x);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

x
The parameter value.

setBinaryStreamMode()

Defines that a column is to be returned as a binary stream by the `getStream` method.

Syntax

```
void setBinaryStreamMode(unsigned int colIndex,
    unsigned int size);
```

Parameters

colIndex
The first column is 1, the second is 2, ...

size

setBlob()

Set a parameter to a `Blob` value.

Syntax

```
void setBlob(unsigned int paramIndex,
    const Blob &x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setBytes()

Set a parameter to a `Bytes` array.

Syntax

```
void setBytes(unsigned int paramIndex,
    const Bytes &x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setCharacterStreamMode()

Defines that a column is to be returned as a character stream by the `getStream` method.

Syntax

```
void setCharacterStreamMode(unsigned int colIndex,
    unsigned int size);
```

Parameters

colIndex

The first column is 1, the second is 2, ...

size

setCharSet()

Overrides the default character set for the specified parameter.   Data is assumed to be in the specified character set and is converted to database character set. For OUT binds of `OCICallableStatements`, this specifies the character set to which database characters are converted to.

Syntax

```
void setCharSet(unsigned int paramIndex,
    CharSet charSet);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

charSet

Selected character set.

setClob()

Set a parameter to a `Clob` value.

Syntax

```
void setClob(unsigned int paramIndex,
    const Clob &x);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

x
The parameter value.

setDate()

Set a parameter to a `Date` value.

Syntax

```
void setDate(unsigned int paramIndex,
    const Date &x);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

x
The parameter value.

setDatabaseNCHARParam()

If the parameter is going to be inserted in a column that contains data in the database's `NCHAR` character set, then OCCI must be informed by passing a true value. A false can be passed to restore the dafault.Returns returns the character set that is in effect for the specified parameter.

Syntax

```
void setDatabaseNCHARParam(unsigned int paramIndex,
    bool isNCHAR);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

isNCHAR

True if this parameter contains data in Database's NCHAR character set; false otherwise.

setDataBuffer()

Specify a data buffer where data would be available. Also, used for OUT bind parameters of callable statements (and DML returning OUT binds in future).

The *buffer* parameter is a pointer to a user allocated data buffer. The current length of data must be specified in the **length* parameter. The amount of data should not exceed the *size* parameter. Finally, *type* is the data type of the data.

Note that not all types can be supplied in the buffer. For example, all OCCI allocated types (such as Bytes, Date and so on.) cannot be provided by the setDataBuffer interface. Similarly, C++ Standard Library strings cannot be provided with the setDataBuffer interface either. The type can be any of OCI data types such VARCHAR2, CSTRING, CHARZ and so on.

If setDataBuffer() is used to specify data for iterative or array executes, it should be called only once in the first iteration only. For subsequent iterations, OCCI would assume that data is at buffer + i*size location where i is the iteration number. Similarly the length of the data would be assumed to be at *(length + i).

Syntax

```
void setDataBuffer(unsigned int paramIndex,
    void *buffer,
    Type type,
    sb4 size,
    ub2 *length,
    sb2 *ind = NULL,
    ub2 *rc= NULL);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

buffer

Pointer to user-allocated buffer; if iterative or array executes are done, it should have numIterations * size bytes in it.

type

Type of the data that is provided (or retrieved) in the buffer

size

Size of the data buffer; for iterative and array executes, it is the size of each element of the data items

length

Pointer to the length of data in the buffer; for iterative and array executes, it should be an array of length data for each buffer element; the size of the array should be equal to arrayLength.

ind

Indicator. For iterative and array executes, an indicator for every buffer element

rc

Return code — for iterative and array executes, a return code for every buffer element

setDataBufferArray()

Specify an array of data buffers where data would be available for reading or writing. Used for `IN`, `OUT`, and `IN/OUT` bind parameters for stored procedures which read/write array parameters.

A stored procedure can have an array of values for `IN`, `IN/OUT`, or `OUT` parameters. In this case, the parameter must be specified using the `setDataBufferArray()` method. The array is specified just as for the

`setDataBuffer()` method for iterative or array executes, but the number of elements in the array is determined by `*arrayLength` parameter.

For `OUT` and `IN/OUT` parameters, the maximum number of elements in the array is specified by the `arraySize` parameter. Note that for iterative prepared statements, the number of elements in the array is determined by the number of iterations, and for array executes the number of elements in the array is determined by the `arrayLength` parameter of the `executeArrayUpdate()` method. However, for array parameters of stored procedures, the number of elements in the array must be specified in the `*arrayLength` parameter of the `setDataBufferArray()` method because each parameter may have a different size array.

This is different from prepared statements where for iterative and array executes, the number of elements in the array for each parameter is the same and is determined by the number of iterations of the statement, but a callable statement is executed only once, and each of its parameter can be a varying length array with possibly a different length.

Note that for `OUT` and `IN/OUT` binds, the number of elements returned in the array is returned in `*arrayLength` as well. The client must make sure that it has allocated `elementSize * arraySize` bytes for the *`buffer`*.

Syntax

```
void setDataBufferArray(unsigned int paramIndex,
    void *buffer,
    Type type,
    ub4 arraySize,
    ub4 *arrayLength,
    sb4 elementSize,
    ub2 *elementLength,
    sb2 *ind = NULL,
    ub2 *rc = NULL);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

buffer
Pointer to user-allocated buffer. It should have `size * arraySize` bytes in it

type
Type of the data that is provided (or retrieved) in the buffer

arraySize
Maximum number of elements in the array

arrayLength
Pointer to number of current elements in the array

elementSize
Size of the data buffer for each element

elementLength
Pointer to an array of lengths. `elementLength[i]` has the current length of the ith element of the array

id

rc

setDouble()

Set a parameter to a C++ double value.

Syntax

```
void setDouble(unsigned int paramIndex,
    double x);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

x
The parameter value.

setErrorOnNull()

Enables/disables exceptions for reading of null values on paramIndex parameter of the statement. If exceptions are enabled, calling a `get`*`xxx`* on *`paramIndex`* parameter would result in an `SQLException` if the parameter value is null. This call can also be used to disable exceptions.

Syntax

```
void setErrorOnNUll(unsigned int paramIndex,
    bool causeException);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

causeExcetpion

Enable exceptions if true, disable if false

setErrorOnTruncate()

This method enables/disables exceptions when truncation occurs.

Syntax

```
void setErrorOnTruncate(unsigned int paramIndex,
    bool causeException);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

causeException

Enable exceptions if true. Disable if false

setFloat()

Set a parameter to a C++ float value.

Syntax

```
void setFloat(unsigned int paramIndex,
    float x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setInt()

Set a parameter to a C++ int value.

Syntax

```
void setInt(unsigned int paramIndex,
    int x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setIntervalDS()

Set a parameter to a `IntervalDS` value.

Syntax

```
void setIntervalDS(unsigned int paramIndex,
```

```
    const IntervalDS &x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setIntervalYM()

Set a parameter to a `Interval` value.

Syntax

```
void setIntervalYM(unsigned int paramIndex,
    const IntervalYM &x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setMaxIterations()

Sets the maximum number of invocations that will be made for the DML statement. This must be called before any parameters are set on the prepared statement. The larger the iterations, the larger the numbers of parameters sent to the server in one round trip. However, a large number causes more memory to be reserved for all the parameters. Note that this is just the maximum limit. Actual number of iterations depends on the number of `addIterations()` that are done.

Syntax

```
void setMaxIterations(unsigned int maxIterations);
```

Parameters

maxIterations

Maximum number of iterations allowed on this statement.

setMaxParamSize()

This method sets the maximum amount of data to be sent or received for the specified parameter. It only applies to character and binary data. If the maximum amount is exceeded, the excess data is discarded. This method can be very useful when working with a `LONG` column. It can be used to truncate the `LONG` column by reading or writing it into a string or `Bytes` data type.

If the `setString` or `setBytes` method has been called to bind a value to an IN/OUT parameter of a pl/sql procedure, and the size of the OUT value is expected to be greater than the size of the IN value, then `setMaxParamSize` should be called.

Syntax

```
void setMaxParamSize(unsigned int paramIndex,
    unsigned int maxSize);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

maxSize

The new max parameter size limit (> 0) .

setNull()

Set a parameter to SQL null. Note that you must specify the parameter's SQL type.

Syntax

```
void setNull(unsigned int paramIndex,
    Type type);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

type

SQL type code defined by Type

setNumber()

Set a parameter to a `Number` value.

Syntax

```
void setNumber(unsigned int paramIndex,
    const Number &x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setObject()

Set the value of a parameter using an object; use the C++.lang equivalent objects for integral values. The OCCI specification specifies a standard mapping from C++ `Object` types to SQL types. The given parameter C++ object will be converted to the corresponding SQL type before being sent to the database.

Syntax

```
void setObject(unsigned int paramIndex,
    PObject * x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The object containing the input parameter value.

setPrefetchMemorySize()

Set the amount of memory that will be used internally by OCCI to store data fetched during each round trip to the server. A value of 0 means that the amount of data fetched during the round trip is constrained by the `FetchRowCount` parameter. If both parameters are nonzero, the smaller of the two is used.

Syntax

```
void setPrefetchMemorySize(unsigned int bytes);
```

Parameters

bytes

Number of bytes to use for storing data fetched during each round trip to the server.

setPrefetchRowCount()

Set the number of rows that will be fetched internally by OCCI during each round trip to the server. A value of 0 means that the amount of data fetched during the round trip is constrained by the `FetchMemorySize` parameter. If both parameters are nonzero, the smaller of the two is used. If both of these parameters are zero, row count internally defaults to 1 row and that is the value returned from the `getFetchRowCount` method below.

Syntax

```
void setPrefetchRowCount(unsigned int rowCount);
```

Parameters

rowCount
Number of rows to fetch for each round trip to the server.

setRef()

Set a parameter to a `RefAny` value.

Syntax

```
void setRef(unsigned int paramIndex,
    RefAny &x);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

x
The parameter value.

setRowid()

Set a `Rowid` bytes array for a bind position.

Syntax

```
void setRowid(unsigned int paramIndex,
    const Bytes &x);
```

Parameters

paramIndex
The first parameter is 1, the second is 2, ...

x
The parameter value.

setSQL()

A new SQL string can be associated with a `Statement` object by this call. Resources associated with the previous SQL statement are freed. In particular, a previously obtained result set is invalidated. If an empty sql string, "", was used when the `Statement` was created, a `setSQL` method with the proper SQL string must be done prior to execution.

Syntax

```
void setSQL(const string &sql);
```

Parameters

sql

Any SQL statement.

setString()

Set a parameter to an string value.

Syntax

```
void setString(unsigned int paramIndex,
    const string &x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setTimestamp()

Set a parameter to a `Timestamp` value.

Syntax

```
void setTimestamp(unsigned int paramIndex,
    const Timestamp &x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setUInt()

Set a parameter to a C++ unsigned int value.

Syntax

```
void setUInt(unsigned int paramIndex,
    unsigned int x);
```

Parameters

paramIndex

The first parameter is 1, the second is 2, ...

x

The parameter value.

setVector()

This method sets a parameter to a vector of unsigned int.

Syntax

There are variants of syntax:

```
void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<int> &vect,
```

```
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<unsigned int> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<double> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<float> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<string> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<RefAny> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<Blob> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<Clob> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<Bfile> &vect,
    string sqltype);
```

```
void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<Timestamp> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<IntervalDS> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<IntervalYM> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<Date> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<T *> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<Number> &vect,
    string sqltype);

void setVector(Statement *stmt,
    unsigned int paramIndex,
    vector<Ref<T>> &vect,
    string sqltype);
```

Parameters

stmt

paramIndex

vect

vecind

sqltype

status()

Returns the current status of the statement. Useful when there is streamed data to be written (or read). Other methods such as `getCurrentStreamParam` and `getCurrentIteration` can be called to find out the streamed parameter that needs to be written and the current iteration number for an iterative or array execute.

The `status` method can be called repeatedly to find out the status of the execution. Returns one of following:

- `RESULT_SET_AVAILABLE`-- call `getResultSet()`
- `UPDATE_COUNT_AVAILABLE` -- call `getUpdateCount()`
- `NEEDS_STREAM_DATA` -- call `getCurrentStream()` and `getCurrentIteration()`, and write stream
- `STREAM_DATA_AVAILABLE` -- call `getCurrentStream()` and `getCurrentIteration()`, and read stream
- `PREPARED`
- `UNPREPARED`

Syntax

```
Status status() const;
```

Stream Class

You use a Stream to read or write streamed data (usually LONG).

- A read-able stream is used to obtain streamed data from a result set or OUT bind variable from a stored procedure call. A read-able stream must be read completely until the end of data is reached or it should be closed to discard any unwanted data.
- A write-able stream is used to provide streamed data (usually LONG) to parameterized statements including callable statements.

```
Stream()
enum Status
   {
      READY_FOR_READ,
      READY_FOR_WRITE,
      INACTIVE
   };
```

Summary of Stream Methods

Method	Summary
readBuffer() on page 8-215	Reads the stream and returns the amount of data read from the Stream object.
readLastBuffer() on page 8-216	Reads last buffer from Stream.
writeBuffer() on page 8-216	Writes data from buffer to the stream.
writeLastBuffer() on page 8-217	Writes the last data from buffer to the stream.
status() on page 8-217	Returns the current status of the stream.

readBuffer()

Reads data from `Stream`. The size parameter specifies the maximum number of byte characters to read. Returns the amount of data read from the `Stream` object. -1 means end of data on the stream

Syntax

```
int readBuffer(char *buffer,
    unsigned int size);
```

Parameters

buffer

Pointer to data buffer; must be allocated and freed by caller

size

Size of the buffer.

readLastBuffer()

This method reads the last buffer from the `Stream`. It can also be called top discard unread data.

The size parameter specifies the maximum number of byte characters to read. Returns the amount of data read from the `Stream` object. -1 means end of data on the stream.

Syntax

```
int readLastBuffer(char *buffer,
    unsigned int size);
```

Parameters

buffer

Pointer to data buffer; must be allocated and freed by caller.

size

Specifies the maximum number of bytes to be read.

writeBuffer()

Writes data from buffer to the stream. The amount of data written is determined by size.

Syntax

```
void writeBuffer(char *buffer,
    unsigned int size);
```

Parameters

buffer

Pointer to data buffer.

size

Number of char's in the buffer.

writeLastBuffer()

This method writes the last data buffer to the stream. It can also be called to write the last chunk of data.

The amount of data written is determined by size.

Syntax

```
void writeLastBuffer(char *buffer,
    unsigned int size);
```

Parameters

buffer

Pointer to data buffer.

size

Specifies the number of bytes to be written.

status()

Returns the current status of the streams, which can be one of the following:

- READY_FOR_READ
- READY_FOR_WRITE

- INACTIVE

Syntax

```
Status status() const;
```

Timestamp Class

This class conforms to the SQL92 TIMESTAMP and TIMESTAMPTZ types.

OCCI expects the SQL data type corresponding to the `Timestamp` class to be of type TIMESTAMP WITH TIME ZONE.

```
Timestamp(const Environment *env,
    int year = 1,
    unsigned int month = 1,
    unsigned int day = 1,
    unsigned int hour = 0,
    unsigned int min = 0,
    unsigned int sec = 0,
    unsigned int fs = 0,
    int tzhour = 0,
```

Returns a null `Timestamp` object. A null timestamp can be initialized by assignment or calling the `fromText` method. Methods that can be called on null timestamp objects are `setNull`, `isNull` and `operator=()`.

```
Timestamp();
```

Assigns the values found in `a`.

```
Timestamp(const Timestamp &src);
```

The following code example demonstrates that the default constructor creates a null value, and how you can assign a non null value to a timestamp and perform operations on it:

```
Environment *env = Environment::createEnvironment();

//create a null timestamp
Timestamp ts;
if(ts.isnull())
    cout << "\n ts is Null";

//assign a non null value to ts
Timestamp notNullTs(env, 2000, 8, 17, 12, 0, 0, 0, 5, 30);
ts = notNullTs;
```

```
//now all operations are valid on ts...
int yr;
unsigned int mth, day;
ts.getDate(yr, mth, day);
```

The following code example demonstrates how to use the `fromText` method to initialize a null timestamp:

```
Environment *env = Environment::createEnvironment();

Timestamp ts1;
ts1.fromText("01:16:17.12 04/03/1825", "hh:mi:ssxff dd/mm/yyyy", "", env);
```

The following code example demonstrates how to get the timestamp column from a result set, check whether the timestamp is null, get the timestamp value in string format, and determine the difference between 2 timestamps:

```
Timestamp reft(env, 2001, 1, 1);
ResultSet *rs=stmt->executeQuery("select order_date from orders
                                  where customer_id=1");
rs->next();

//retrieve the timestamp column from result set
Timestamp ts=rs->getTimestamp(1);

//check timestamp for null
if(!ts.isNull())
{
    //get the timestamp value in string format
    string tsstr=ts.toText("dd/mm/yyyy hh:mi:ss [tzh:tzm]",0);

    if(reft<ts //compare timestamps
    {
        IntervalDS ds=reft.subDS(ts); //get difference between timestamps
    }
}
```

Summary of Timestamp Methods

Method	Summary
fromText() on page 8-222	Sets the time stamp from the values provided by the string.
getDate() on page 8-222	Gets the date from the `Timestamp` object.
getTime() on page 8-223	Gets the time from the `TimeStamp` object.
getTimeZoneOffset() on page 8-223	Returns the time zone hour and minute offset value.
intervalAdd() on page 8-224	Returns a `Timestamp` object with value (this + interval).
intervalSub() on page 8-224	Returns a `Timestamp` object with value (this - interval).
isNull() on page 8-225	Check if `Timestamp` is null.
operator=() on page 8-225	Simple assignment.
operator==() on page 8-225	Check if a and b are equal.
operator!=() on page 8-226	Check if a and b are not equal.
operator>() on page 8-226	Check if a is greater than b.
operator>=() on page 8-227	Check if a is greater than or equal to b.
operator<() on page 8-227	Check if a is less than b.
operator<=() on page 8-228	Check if a is less than or equal to b.
setDate() on page 8-228	Sets the year, month, day components contained for this timestamp.
setNull() on page 8-229	Sets the value of Timestamp to null
setTime() on page 8-229	Sets the day, hour, minute, second and fractional second components for this timestamp.
setTimeZoneOffset() on page 8-229	Sets the hour and minute offset for timezone.
subDS() on page 8-230	Returns a `IntervalDS` representing this - `val`.
subYM() on page 8-230	Returns a `IntervalYM` representing this - `val`.
toText() on page 8-230	

fromText()

If *nlsParam* is specified, this will determine the nls parameters to be used for the conversion. If *nlsParam* is not specified, the nls parameters are picked up from *env*. If *env* is null, the nls parameters are picked up from the environment associated with the instance, if any.

Syntax

```
void fromText(const string &timestmpStr,
    const string &fmt,
    const string &nlsParam = "",
    const Environment *env = NULL);
```

Parameters

timestmpStr

fmt

nlsParam

env

getDate()

Return the year, day, and month values of the `Timestamp`.

Syntax

```
void getDate(int &year,
    unsigned int &month,
    unsigned int &day) const;
```

Parameters

year

month

day

getTime()

Return the hour, minute, second, and fractional second (fs) values.

Syntax

```
void getTime(unsigned int &hour,
    unsigned int &minute,
    unsigned int &second,
    unsigned int &fs) const;
```

Parameters

hour

minute

second

fs

getTimeZoneOffset()

Returns the time zone offset from GMT in hours and minutes.

Syntax

```
void getTimeZoneOffset(int &hour,
    int &minute) const;
```

Parameters

hour

minute

intervalAdd()

Returns a `Timestamp` with the value of this + `val`.

Syntax
There are variants of syntax:

```
Timestamp intervalAdd(const IntervalDS& val) const;
```

```
Timestamp intervalAdd(const IntervalYM& val) const;
```

Parameters

val

intervalSub()

Returns a `Timestamp` with the value of this - `val`.

Syntax
There are variants of syntax:

```
Timestamp intervalSub(const IntervalDS& val) const;
```

```
Timestamp intervalSub(const IntervalYM& val) const;
```

Parameters

val

isNull()

Returns true if `Timestamp` is null or false otherwise.

Syntax

```
bool isNull() const;
```

operator=()

Assigns the values found in `src` to the `Timestamp` object.

Syntax

```
Timestamp & operator=(const Timestamp &src);
```

Parameters

src

operator==()

Returns true if a is the same as `b`, false otherwise. If either `a` or `b` is null then false is returned.

Syntax

```
bool operator==(const Timestamp &a,
    const Timestamp &b);
```

Parameters

a

A timestamp.

b

Another timestamp.

operator!=()

This method compares the timestamps specified. If the timestamps are not equal then true is returned; otherwise, false is returned. If either timestamp is null then false is returned.

Syntax

```
bool operator!=(const Timestamp &a,
    const Timestamp &b);
```

Parameters

a

A timestamp.

b

Another timestamp.

operator>()

Returns true if a is after `b`, false otherwise. If either `a` or `b` is null then false is returned.

Syntax

```
bool operator>(const Timestamp &a,
    const Timestamp &b);
```

Parameters

a

A timestamp.

b

Another timestamp.

operator>=()

This method compares the timestamps specified. If the first timestamp is greater than or equal to the second timestamp then true is returned; otherwise, false is returned. If either timestamp is null then false is returned.

Syntax

```
bool operator>=(const Timestamp &a,
    const Timestamp &b);
```

Parameters

a

A timestamp.

b

Another timestamp.

operator<()

Returns true if `a` is before `b`, false otherwise. If either `a` or `b` is null then false is returned.

Syntax

```
bool operator<(const Timestamp &a,
    const Timestamp &b);
```

Parameters

a

A timestamp.

b

Another timestamp.

operator<=()

This method compares the timestamps specified. If the first timestamp is less than or equal to the second timestamp then true is returned; otherwise, false is returned. If either timestamp is null then false is returned.

Syntax

```
bool operator<=(const Timestamp &a,
    const Timestamp &b);
```

Parameters

a

A timestamp.

b

Another timestamp.

setDate()

Sets the year, month, day components contained for this timestamp

Syntax

```
void setDate(int year,
    unsigned int month,
    unsigned int day);
```

Parameters

year

month

day

setNull()

Syntax

```
void setNull();
```

setTime()

Sets the day, hour, minute, second and fractional second components for this timestamp.

Syntax

```
void setTime(unsigned int hour,
    unsigned int minute,
    unsigned int second,
    unsigned int fs);
```

Parameters

hour

minute

second

fs

setTimeZoneOffset()

Sets the hour and minute offset for timezone.

Syntax

```
void setTimeZoneOffset(int hour,
    int minute);
```

Parameters

hour

minute

subDS()

Subtract dt from this (this - dt) and return the difference as a `IntervalDS`.

Syntax

```
IntervalDS subDS(const Timestamp& val) const;
```

Parameters

val

subYM()

Subtract dt from this (this - dt) and return the difference as a `IntervalYM`.

Syntax

```
IntervalYM subYM(const Timestamp& val) const;
```

Parameters

val

toText()

If `nlsParam` is specified, this will determine the nls parameters to be used for the conversion. If `nlsParam` is not specified, the nls parameters are picked up from `envp`. If `envp` is null, the nls parameters are picked up from the environment associated with the instance, if any.

Syntax

```
string toText(const string &fmt,
```

```
    unsigned int fsprec,
    const string &nlsParam = "") const;
```

Parameters

fmt

fsprec

nlsParam

Part III

Appendix

This part contains one appendix:

- Appendix A, "OCCI Demonstration Programs"

A

OCCI Demonstration Programs

Oracle provides code examples illustrating the use of OCCI calls. These programs are provided for demonstration purposes, and are not guaranteed to run on all platforms.

The demonstration programs (demos) are available with your Oracle installation. The location, names, and availability of the programs may vary on different platforms. On a UNIX workstation, the programs are installed in the `ORACLE_HOME/rdbms/demo` directory.

The `$ORACLE_HOME/rdbms/demo` directory contains not only demos but the file named `demo_rdbms.mk` that must be used as a template for building your own OCCI applications or external procedures. Building a new `demo_rdbms.mk` file consists of customizing the `demo_rdbms.mk` file by adding your own macros to the link line. However, Oracle requires that you keep the macros provided in the `demo_rdbms.mk` file, as it will result in simplified maintenance of your own `demo_rdbms.mk` files.

When a specific header or SQL file is required by the application, these files are also included. Review the information in the comments at the beginning of the demonstration programs for setups and hints on running the programs.

Prior to running any of these demos, the SQL file `occidemo.sql` must be run using the user name and password `SCOTT` and `TIGER` respectively.

Table A–1 lists the important demonstration programs and the OCCI features that they illustrate.

Table A–1 OCCI Demonstration Programs

Program Name	Features Illustrated
`occiblob.cpp`	Demonstrates how to read and write BLOBs

Table A–1 OCCI Demonstration Programs (Cont.)

Program Name	Features Illustrated
occiclob.cpp	Demonstrates how to read and write CLOBs
occicoll.cpp	Demonstrates how to perform simple insert, delete, and update operations on a table column of type Nested Table
occidesc.cpp	Demonstrates how to obtain metadata about a table, procedure, and object
occidml.cpp	Demonstrates how to perform insert, select, update, and delete operations of a table row by using OCCI
occiinh.cpp	Demonstrates object inheritance by using insert, select, update, and delete operations on a table row of subtype table
occiobj.cpp	Demonstrates how to perform insert, select, update, and delete operations on a table row containing an object as one of its columns
occipobj.cpp	Demonstrates how to perform insert, select, and update operations on persistent objects, as well as how to pin, unpin, mark for deletion, and flush an object
occipool.cpp	Demonstrates how to use the connection pool interface of OCCI
occiproc.cpp	Demonstrates how to invoke PL/SQL procedures with bind parameters
occistre.cpp	Demonstrates how to use OCCI ResultSet streams

OCCI Demonstration Programs

This section lists each of the OCCI demonstration program files, in addition to the demo make file.

demo_rdbms.mk

The following code is for the make file that generates the demonstration programs:

```
#
# Example for building demo OCI programs:
#
# 1. All OCI demos (including extdemo2 and extdemo4):
#
#    make -f demo_rdbms.mk demos
#
```

```
# 2. A single OCI demo:
#
#    make -f demo_rdbms.mk build EXE=demo OBJS="demo.o ..."
#    e.g. make -f demo_rdbms.mk build EXE=oci02 OBJS=oci02.o
#
# 3. A single OCI demo with static libraries:
#
#    make -f demo_rdbms.mk build_static EXE=demo OBJS="demo.o ..."
#    e.g. make -f demo_rdbms.mk build_static EXE=oci02 OBJS=oci02.o
#
# 4. To re-generate shared library:
#
#    make -f demo_rdbms.mk generate_sharedlib
#
# 5. All OCCI demos
#
#    make -f demo_rdbms.mk occidemos
#
# 6. A single OCCI demo:
#
#    make -f demo_rdbms.mk <demoname>
#    e.g. make -f demo_rdbms.mk occidml
#    OR
#    make -f demo_rdbms.mk buildocci EXE=demoname OBJS="demoname.o ..."
#    e.g. make -f demo_rdbms.mk buildocci EXE=occidml OBJS=occidml.o
#
#
# Example for building demo DIRECT PATH API programs:
#
# 1. All DIRECT PATH API demos:
#
#    make -f demo_rdbms.mk demos_dp
#
# 2. A single DIRECT PATH API demo:
#
#    make -f demo_rdbms.mk build_dp EXE=demo OBJS="demo.o ..."
#    e.g. make -f demo_rdbms.mk build_dp EXE=cdemodp_lip OBJS=cdemodp_lip.o
#
#
# Example for building external procedures demo programs:
#
# 1. All external procedure demos:
#
# 2. A single external procedure demo whose 3GL routines do not use the
#    "with context" argument:
```

```
#
#    make -f demo_rdbms.mk extproc_no_context SHARED_LIBNAME=libname
#                                             OBJS="demo.o ..."
#    e.g. make -f demo_rdbms.mk extproc_no_context SHARED_LIBNAME=epdemo.so
#                                             OBJS="epdemo1.o epdemo2.o"
#
# 3. A single external procedure demo where one or more 3GL routines use the
#    "with context" argument:
#
#    make -f demo_rdbms.mk extproc_with_context SHARED_LIBNAME=libname
#                                             OBJS="demo.o ..."
#    e.g. make -f demo_rdbms.mk extproc_with_context SHARED_LIBNAME=epdemo.so
#                                             OBJS="epdemo1.o epdemo2.o"
#    e.g. make -f demo_rdbms.mk extproc_with_context
#                        SHARED_LIBNAME=extdemo2.so OBJS="extdemo2.o"
#    e.g. or For EXTDEMO2 DEMO ONLY: make -f demo_rdbms.mk demos
#
# 4. To link C++ demos:
#
#    make -f demo_rdbms.mk c++demos
#
#
# NOTE: 1. ORACLE_HOME must be either:
#                  . set in the user's environment
#                  . passed in on the command line
#                  . defined in a modified version of this makefile
#
#       2. If the target platform support shared libraries (e.g. Solaris)
#          look in the platform specific documentation for information
#          about environment variables that need to be properly
#          defined (e.g. LD_LIBRARY_PATH in Solaris).
#

include $(ORACLE_HOME)/rdbms/lib/env_rdbms.mk

# flag for linking with non-deferred option (default is deferred mode)
NONDEFER=false

DEMO_DIR=$(ORACLE_HOME)/rdbms/demo
DEMO_MAKEFILE = $(DEMO_DIR)/demo_rdbms.mk

DEMOS = cdemo1 cdemo2 cdemo3 cdemo4 cdemo5 cdemo81 cdemo82 \
        cdemobj cdemolb cdemodsc cdemocor cdemolb2 cdemolbs \
        cdemodr1 cdemodr2 cdemodr3 cdemodsa obndra \
        cdemoext cdemothr cdemofil cdemofor \
```

```
        oci02 oci03 oci04 oci05 oci06 oci07 oci08 oci09 oci10 \
        oci11 oci12 oci13 oci14 oci15 oci16 oci17 oci18 oci19 oci20 \
        oci21 oci22 oci23 oci24 oci25 readpipe cdemosyev \
        ociaqdemo00 ociaqdemo01 ociaqdemo02 cdemoucb nchdemo1

DEMOS_DP = cdemodp_lip

C++DEMOS = cdemo6
OCCIDEMOS = occiblob occiclob occicoll occidesc occidml occipool occiproc \
            occistre
OCCIOTTDEMOS = occiobj occiinh occipobj
OTTUSR = scott
OTTPWD = tiger

.SUFFIXES: .o .cob .for .c .pc .cc .cpp

demos: $(DEMOS) extdemo2 extdemo4

demos_dp: $(DEMOS_DP)

generate_sharedlib:
        $(SILENT)$(ECHO) "Building client shared library ..."
        $(SILENT)$(ECHO) "Calling script $$ORACLE_HOME/bin/genclntsh ..."
        $(GENCLNTSH)
        $(SILENT)$(ECHO) "The library is $$ORACLE_HOME/lib/libclntsh.so... DONE"

BUILD=build
$(DEMOS):
        $(MAKE) -f $(DEMO_MAKEFILE) $(BUILD) EXE=$@ OBJS=$@.o

$(DEMOS_DP): cdemodp.c cdemodp0.h cdemodp.h
        $(MAKE) -f $(DEMO_MAKEFILE) build_dp EXE=$@ OBJS=$@.o

c++demos: $(C++DEMOS)

$(C++DEMOS):
        $(MAKE) -f $(DEMO_MAKEFILE) buildc++ EXE=$@ OBJS=$@.o

buildc++: $(OBJS)
        $(MAKECPLPLDEMO)

occidemos:      $(OCCIDEMOS) $(OCCIOTTDEMOS)

$(OCCIDEMOS):
        $(MAKE) -f $(DEMO_MAKEFILE) buildocci EXE=$@ OBJS=$@.o
```

```
$(OCCIOTTDEMOS):
        $(MAKE) -f $(DEMO_MAKEFILE) ott OTTFILE=$@
        $(MAKE) -f $(DEMO_MAKEFILE) buildocci EXE=$@ OBJS="$@.o $@o.o $@m.o"

buildocci: $(OBJS)
        $(MAKEOCCISHAREDDEMO)

buildocci_static: $(OBJS)
        $(MAKEOCCISTATICDEMO)

ott:
        $(ORACLE_HOME)/bin/ott \
                userid=$(OTTUSR)/$(OTTPWD) \
                intype=$(OTTFILE).typ \
                outtype=$(OTTFILE)out.type \
                code=cpp \
                hfile=$(OTTFILE).h \
                cppfile=$(OTTFILE)o.cpp \
                attraccess=private

# Pro*C rules
# SQL Precompiler macros

pc1:
        $(PCC2C)

.pc.c:
        $(MAKE) -f $(DEMO_MAKEFILE) PCCSRC=$* I_SYM=include= pc1

.pc.o:
        $(MAKE) -f $(DEMO_MAKEFILE) PCCSRC=$* I_SYM=include= pc1
        $(PCCC2O)

.cc.o:
        $(CCC2O)

.cpp.o:
        $(CCC2O)

build: $(LIBCLNTSH) $(OBJS)
        $(BUILDEXE)

extdemo2:
        $(MAKE) -f $(DEMO_MAKEFILE) extproc_with_context
```

```
SHARED_LIBNAME=extdemo2.so OBJS="extdemo2.o"

extdemo4:
        $(MAKE) -f $(DEMO_MAKEFILE) extproc_with_context
SHARED_LIBNAME=extdemo4.so OBJS="extdemo4.o"

.c.o:
        $(C2O)

build_dp: $(LIBCLNTSH) $(OBJS) cdemodp.o
        $(DPTARGET)

build_static: $(OBJS)
        $(O2STATIC)

# extproc_no_context and extproc_with_context are the current names of these
# targets.  The old names, extproc_nocallback and extproc_callback are
# preserved for backward compatibility.

extproc_no_context extproc_nocallback: $(OBJS)
        $(BUILDLIB_NO_CONTEXT)

extproc_with_context extproc_callback: $(OBJS) $(LIBCLNTSH)
        $(BUILDLIB_WITH_CONTEXT)

clean:
        $(RM) -f $(DEMOS) extdemo2 extdemo4 *.o *.so
        $(RM) -f $(OCCIDEMOS)  $(OCCIOTTDEMOS) occi*.h occi*m.cpp occi*o.cpp \
            occi*.type
```

occiblob.cpp

The following code example demonstrates how to read and write aBLOB:

```
#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

/**
 * The demo sample has starts from startDemo method.  This method is called
 * by main.  startDemo calls other methods, the supporting methods for
 * startDemo are,
```

```
 * insertRows    - insert the rows into the table
 * deleteRows    - delete the rows inserted
 * insertBlob    - Inserts a blob and an empty_blob
 * populateBlob  - populates a given blob
 * dumpBlob      - prints the blob as an integer stream
 */

class  demoBlob
{
  private:
  string username;
  string password;
  string url;

  void insertRows (Connection *conn)
   throw (SQLException)
  {
    Statement *stmt = conn->createStatement ("INSERT INTO
print_media(product_id,ad_id,ad_composite,ad_sourcetext) VALUES
(6666,11001,'10001','SHE')");
    stmt->executeUpdate();
    stmt->setSQL ("INSERT INTO
print_media(product_id,ad_id,ad_composite,ad_sourcetext) VALUES
(7777,11001,'1010','HEM')");
    stmt->executeUpdate();
    conn->commit();
    conn->terminateStatement (stmt);

  }

  void deleteRows (Connection *conn)
   throw (SQLException)
  {
    Statement *stmt = conn->createStatement ("DELETE print_media WHERE
product_id = 6666 AND ad_id=11001");
    stmt->executeUpdate();
    stmt->setSQL ("DELETE print_media WHERE product_id = 7777 AND ad_id=11001");
    stmt->executeUpdate();
    conn->commit();
    conn->terminateStatement (stmt);

  }

  /**
   * populating the blob;
```

```
 */
void populateBlob (Blob &blob, int size)
 throw (SQLException)
{
  Stream *outstream = blob.getStream (1,0);
  char *buffer = new char[size];

  memset (buffer, (char)10, size);
  outstream->writeBuffer (buffer, size);
  char *c = (char *)"";
  outstream->writeLastBuffer (c,0);
  delete (buffer);
  blob.closeStream (outstream);
}

/**
 * printing the blob data as integer stream
 */
void dumpBlob (Blob &blob, int size)
 throw (SQLException)
{
  Stream *instream = blob.getStream (1,0);
  char *buffer = new char[size];
  memset (buffer, NULL, size);

  instream->readBuffer (buffer, size);
  cout << "dumping blob: ";
  for (int i = 0; i < size; ++i)
    cout << (int) buffer[i];
  cout << endl;

  delete (buffer);
  blob.closeStream (instream);
}



/**
 * public methods
 */
public:
demoBlob ()
{
  /**
   * default values of username & password
```

```
      */
      username = "SCOTT";
      password = "TIGER";
      url = "";
    }

    void setUsername (string u)
    {
      username = u;
    }

    void setPassword (string p)
    {
      password = p;
    }

    void setUrl (string u)
    {
      url = u;
    }

    void runSample ()
     throw (SQLException)
    {
      Environment *env = Environment::createEnvironment (
      Environment::DEFAULT);
      try
      {
        Connection *conn = env->createConnection (username, password, url);
        Statement *stmt1;
        insertRows (conn);
        /**
         * Reading a populated blob & printing its property.
         */
        string sqlQuery = "SELECT  ad_composite FROM print_media WHERE
product_id=6666";
        Statement *stmt = conn->createStatement (sqlQuery);

        ResultSet *rset = stmt->executeQuery ();
        while (rset->next ())
        {
          Blob blob = rset->getBlob (1);
          cout << "Opening the blob in Read only mode" << endl;
          blob.open (OCCI_LOB_READONLY);
          int blobLength=blob.length ();
```

```
        cout << "Length of the blob is: " << blobLength << endl;
        dumpBlob (blob, blobLength);
        blob.close ();
      }
      stmt->closeResultSet (rset);

      /**
       * Reading a populated blob & printing its property.
       */
      stmt->setSQL ("SELECT ad_composite FROM print_media WHERE product_id =7777
FOR UPDATE");
      rset = stmt->executeQuery ();
      while (rset->next ())
      {
        Blob blob = rset->getBlob (1);
        cout << "Opening the blob in read write mode" << endl;
        blob.open (OCCI_LOB_READWRITE);
        cout << "Populating the blob" << endl;
        populateBlob (blob, 20);
        int blobLength=blob.length ();
        cout << "Length of the blob is: " << blobLength << endl;
        dumpBlob (blob, blobLength);
        blob.close ();
      }
      stmt->closeResultSet (rset);
      deleteRows (conn);
      conn->terminateStatement (stmt);
      env->terminateConnection (conn);
    }
    catch (SQLException ea)
    {
      cout << ea.what();
    }
    Environment::terminateEnvironment (env);
  }

};//end of class  demoBlob

int main (void)
{
  demoBlob *b = new demoBlob ();
  b->setUsername ("SCOTT");
  b->setPassword ("TIGER");
  b->runSample ();
}
```

occiclob.cpp

The following code example demonstrates how to read and write a CLOB:

```
#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

/**
 * The demo sample has starts from startDemo method.  This method is called
 * by main.  startDemo calls other methods, the supporting methods for
 * startDemo are,
 * insertRows    -  inserts the rows into the table1
 * deleteRows    -  delete the rows inserted
 * insertClob    - Inserts a clob and an empty_clob
 * populateClob  - populates a given clob
 * dumpClob      - prints the clob as an integer stream
 */

class  demoClob
{
  private:
  string username;
  string password;
  string url;

  void insertRows (Connection *conn)
   throw (SQLException)
  {
    Statement *stmt = conn->createStatement ("INSERT INTO
print_media(product_id,ad_id,ad_composite,ad_sourcetext) VALUES
(3333,11001,'10001','SHE')");
    stmt->executeUpdate();
    stmt->setSQL ("INSERT INTO
print_media(product_id,ad_id,ad_composite,ad_sourcetext) VALUES
(4444,11001,'1010','HEM')");
    stmt->executeUpdate();
    conn->commit();
    conn->terminateStatement (stmt);

  }
```

```
  void deleteRows (Connection *conn)
   throw (SQLException)
  {
    Statement *stmt = conn->createStatement ("DELETE print_media WHERE
product_id=3333 AND ad_id=11001");
    stmt->executeUpdate();
    stmt->setSQL("DELETE print_media WHERE product_id=4444 AND ad_id=11001");
    stmt->executeUpdate();
    conn->commit();
    conn->terminateStatement (stmt);

  }

  /**
   * populating the clob;
   */
  void populateClob (Clob &clob, int size)
   throw (SQLException)
  {
    Stream *outstream = clob.getStream (1,0);
    char *buffer = new char[size];

    memset (buffer,'H', size);
    outstream->writeBuffer (buffer, size);
    char *c = (char *)"";
    outstream->writeLastBuffer (c,0);
    delete (buffer);
    clob.closeStream (outstream);
  }

  /**
   * printing the clob data as integer stream
   */
  void dumpClob (Clob &clob, int size)
   throw (SQLException)
  {
    Stream *instream = clob.getStream (1,0);
    char *buffer = new char[size];
    memset (buffer, NULL, size);

    instream->readBuffer (buffer, size);
    cout << "dumping clob: ";
    for (int i = 0; i < size; ++i)
      cout << (char) buffer[i];
    cout << endl;
```

```
  delete (buffer);
  clob.closeStream (instream);
}



/**
 * public methods
 */
public:
demoClob ()
{
  /**
   * default values of username & password
   */
   username = "SCOTT";
   password = "TIGER";
   url = "";
}

void setUsername (string u)
{
  username = u;
}

void setPassword (string p)
{
  password = p;
}

void setUrl (string u)
{
  url = u;
}

void runSample ()
 throw (SQLException)
{
  Environment *env = Environment::createEnvironment (
  Environment::DEFAULT);
  try
  {
    Connection *conn = env->createConnection (username, password, url);
    Statement *stmt1;
```

```
      insertRows (conn);
      /**
       * Reading a populated clob & printing its property.
       */
      string sqlQuery = "SELECT  ad_sourcetext FROM print_media WHERE
product_id=3333";
      Statement *stmt = conn->createStatement (sqlQuery);

      ResultSet *rset = stmt->executeQuery ();
      while (rset->next ())
      {
        Clob clob = rset->getClob (1);
        cout << "Opening the clob in Read only mode" << endl;
        clob.open (OCCI_LOB_READONLY);
        int clobLength=clob.length ();
        cout << "Length of the clob is: " << clobLength << endl;
        dumpClob (clob, clobLength);
        clob.close ();
      }
      stmt->closeResultSet (rset);

      /**
       * Reading a populated clob & printing its property.
       */
      stmt->setSQL ("SELECT ad_sourcetext FROM print_media WHERE product_id
=4444 FOR UPDATE");
      rset = stmt->executeQuery ();
      while (rset->next ())
      {
        Clob clob = rset->getClob (1);
        cout << "Opening the clob in read write mode" << endl;
        clob.open (OCCI_LOB_READWRITE);
        cout << "Populating the clob" << endl;
        populateClob (clob, 20);
        int clobLength=clob.length ();
        cout << "Length of the clob is: " << clobLength << endl;
        dumpClob (clob, clobLength);
        clob.close ();
      }
      stmt->closeResultSet (rset);
      conn->terminateStatement (stmt);
      deleteRows(conn);
      env->terminateConnection (conn);
    }
    catch (SQLException ea)
```

```
    {
      cout << ea.what();
    }
    Environment::terminateEnvironment (env);
  }

};//end of class  demoClob

int main (void)
{
  demoClob *b = new demoClob ();
  b->setUsername ("SCOTT");
  b->setPassword ("TIGER");
  b->runSample ();
}
```

occicoll.cpp

The following code example demonstrates how to perform simple insert, delete, and update operations on a table column of type `Nested Table`:

```
/**
*occicoll - To exhibit simple insert, delete & update operations"
*     " on table having a Nested Table column
*
*Description
*  Create a program which has insert,delete and update on a
*  table having a Nested table column.
*  Perform all these operations using OCCI interface.
*/

#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

typedef vector<string> journal;

class occicoll
{
  private:

  Environment *env;
  Connection *conn;
```

```
  Statement *stmt;
  string tableName;
  string typeName;

  public:

  occicoll (string user, string passwd, string db)
  {
    env = Environment::createEnvironment (Environment::OBJECT);
    conn = env->createConnection (user, passwd, db);
    initRows();
  }

  ~occicoll ()
  {
    env->terminateConnection (conn);
    Environment::terminateEnvironment (env);
  }

  void setTableName (string s)
  {
    tableName = s;
  }

  void initRows ()
  {
    try{
    Statement *st1 = conn->createStatement ("DELETE FROM journal_tab");
    st1->executeUpdate ();
    st1->setSQL("INSERT INTO journal_tab (jid, jname) VALUES (22, journal
('NATION', 'TIMES'))");
    st1->executeUpdate ();
    st1->setSQL("INSERT INTO journal_tab (jid, jname) VALUES (33, journal
('CRICKET', 'ALIVE'))");
    st1->executeUpdate ();
    conn->commit();
    conn->terminateStatement (stmt);
    }catch(SQLException ex)
    {
     cout<<ex.what();
    }
  }
  /**
   * Insertion of a row
   */
```

```
void insertRow ()
{
  int c1 = 11;
  journal c2;

  c2.push_back ("LIFE");
  c2.push_back ("TODAY");
  c2.push_back ("INVESTOR");

  cout << "Inserting row with jid = " << 11 <<
    " and journal_tab (LIFE, TODAY, INVESTOR )" << endl;
  try{
  stmt = conn->createStatement (
   "INSERT INTO journal_tab (jid, jname) VALUES (:x, :y)");
  stmt->setInt (1, c1);
  setVector (stmt, 2, c2, "JOURNAL");
  stmt->executeUpdate ();
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for insertRow"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
  }
  cout << "Insertion - Successful" << endl;
  conn->terminateStatement (stmt);

}

// Displaying all the rows of the table
void displayAllRows ()
{
  cout << "Displaying all the rows of the table" << endl;
  stmt = conn->createStatement (
    "SELECT jid, jname FROM journal_tab");

  journal c2;

  ResultSet *rs = stmt->executeQuery();
  try{
  while (rs->next())
  {
    cout << "jid: " << rs->getInt(1) << endl;
    cout << "jname: ";
    getVector (rs, 2, c2);
    for (int i = 0; i < c2.size(); ++i)
```

```
      cout << c2[i] << " ";
    cout << endl;
  }
 }catch(SQLException ex)
  {
   cout<<"Exception thrown for displayRow"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
  }
  stmt->closeResultSet (rs);
  conn->terminateStatement (stmt);

} // End of displayAllRows()

// Deleting a row in a nested table
void deleteRow (int c1, string str)
{
  cout << "Deleting a row in a nested table of strings" << endl;
  stmt = conn->createStatement (
    "SELECT jname FROM journal_tab WHERE jid = :x");
  journal c2;
  stmt->setInt (1, c1);

  ResultSet *rs = stmt->executeQuery();
  try{
  if (rs->next())
  {
    getVector (rs, 1, c2);
    c2.erase (find (c2.begin(), c2.end(), str));
  }
  stmt->setSQL ("UPDATE journal_tab SET jname = :x WHERE jid = :y");
  stmt->setInt (2, c1);
  setVector (stmt, 1, c2, "JOURNAL");
  stmt->executeUpdate ();
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for delete row"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
  }

  cout << "Deletion - Successful" << endl;
  conn->commit();
  stmt->closeResultSet (rs);
  conn->terminateStatement (stmt);
```

```
  } // End of deleteRow (int, string)

  // Updating a row of the nested table of strings
  void updateRow (int c1, string str)
  {
    cout << "Updating a row of the nested table of strings" << endl;
    stmt = conn->createStatement (
      "SELECT jname FROM journal_tab WHERE jid = :x");

    journal c2;

    stmt->setInt (1, c1);
    ResultSet *rs = stmt->executeQuery();
    try{
    if (rs->next())
    {
      getVector (rs, 1, c2);
      c2[0] = str;
    }
    stmt->setSQL ("UPDATE journal_tab SET jname = :x WHERE jid = :y");
    stmt->setInt (2, c1);
    setVector (stmt, 1, c2, "JOURNAL");
    stmt->executeUpdate ();
    }catch(SQLException ex)
    {
     cout<<"Exception thrown for updateRow"<<endl;
     cout<<"Error number: "<<  ex.getErrorCode() << endl;
     cout<<ex.getMessage() << endl;
    }

    cout << "Updation - Successful" << endl;
    conn->commit();
    stmt->closeResultSet (rs);
    conn->terminateStatement (stmt);
  } // End of UpdateRow (int, string)

};//end of class occicoll

int main (void)
{
  string user = "SCOTT";
  string passwd = "TIGER";
  string db = "";

  try
```

```
  {
    cout << "occicoll - Exhibiting simple insert, delete & update operations"
     " on table having a Nested Table column" << endl;
    occicoll *demo = new occicoll (user, passwd, db);

    cout << "Displaying all rows before the operations" << endl;
    demo->displayAllRows ();

    demo->insertRow ();

    demo->deleteRow (11, "TODAY");

    demo->updateRow (33, "New_String");

    cout << "Displaying all rows after all the operations" << endl;
    demo->displayAllRows ();

    delete (demo);
    cout << "occicoll - done" << endl;
  }catch (SQLException ea)
  {
    cerr << "Error running the demo: " << ea.getMessage () << endl;
  }
}
```

occidesc.cpp

The following code example demonstrates how to obtain metadata about a table, procedure, and object:

```
/**
*  occidesc - Describing the various objects of the database.
*
*  DESCRIPTION :
*    This program describes the objects of the database, like, table, object
*    and procedure.
*
*/

#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

class occidesc
```

```
{
  private:

  Environment *env;
  Connection *conn;
  public :
  /**
   * Constructor for the occidesc demo program.
   */
  occidesc (string user, string passwd, string db) throw (SQLException)
  {
    env = Environment::createEnvironment (Environment::OBJECT);
    conn = env->createConnection (user, passwd, db);
  }// end of constructor occidesc (string, string, string )

  /**
   * Destructor for the occidesc demo program.
   */
  ~occidesc () throw (SQLException)
  {
    env->terminateConnection (conn);
    Environment::terminateEnvironment (env);
  }  // end of ~occidesc ()

  // Describing a subtype
  void describe_type()
  {
    cout << "Describing the object - PERSON_OBJ " << endl;
    MetaData metaData = conn->getMetaData ((char *)"PERSON_OBJ");
    int mdTyp = metaData.getInt(MetaData::ATTR_PTYPE);
    if (mdTyp == MetaData::PTYPE_TYPE)
    {
      cout << "PERSON_OBJ is a type" << endl;
    }
    int typcode = metaData.getInt(MetaData::ATTR_TYPECODE);
    if (typcode == OCCI_TYPECODE_OBJECT)
       cout << "PERSON_OBJ is an object type" << endl;
    else
       cout << "PERSON_OBJ is not an object type" << endl;
    int numtypeattrs = metaData.getInt(MetaData::ATTR_NUM_TYPE_ATTRS);
    cout << "Object has " << numtypeattrs << " attributes" << endl;
    try
    {
      cout << "Object id: " << metaData.getUInt (MetaData::ATTR_OBJ_ID)
        << endl;
```

```
}
catch (SQLException ex)
{
  cout << ex.getMessage() << endl;
}
cout << "Object Name: " <<
  metaData.getString (MetaData::ATTR_OBJ_NAME) << endl;
cout << "Schema Name: " <<
  (metaData.getString(MetaData::ATTR_OBJ_SCHEMA)) << endl;
cout << "Attribute version: " <<
  (metaData.getString(MetaData::ATTR_VERSION)) << endl;
if (metaData.getBoolean(MetaData::ATTR_IS_INCOMPLETE_TYPE))
  cout << "Incomplete type" << endl;
else
  cout << "Not Incomplete type" << endl;
if (metaData.getBoolean(MetaData::ATTR_IS_SYSTEM_TYPE))
  cout << "System type" << endl;
else
  cout << "Not System type" << endl;
if (metaData.getBoolean(MetaData::ATTR_IS_PREDEFINED_TYPE))
  cout << "Predefined Type" << endl;
else
  cout << "Not Predefined Type" << endl;
if (metaData.getBoolean(MetaData::ATTR_IS_TRANSIENT_TYPE))
  cout << "Transient Type" << endl;
else
  cout << "Not Transient Type" << endl;
if (metaData.getBoolean(MetaData::ATTR_IS_SYSTEM_GENERATED_TYPE))
  cout << "System-generated type" << endl;
else
  cout << "Not System-generated type" << endl;
if (metaData.getBoolean(MetaData::ATTR_HAS_NESTED_TABLE))
  cout << "Has nested table" << endl;
else
  cout << "Does not have nested table" << endl;
if (metaData.getBoolean(MetaData::ATTR_HAS_LOB))
  cout << "Has LOB" << endl;
else
  cout << "Does not have LOB" << endl;
if (metaData.getBoolean(MetaData::ATTR_HAS_FILE))
  cout << "Has BFILE" << endl;
else
  cout << "Does not have BFILE" << endl;
if (metaData.getBoolean(MetaData::ATTR_IS_INVOKER_RIGHTS))
  cout << "Object is Invoker rights" << endl;
```

```
  else
    cout << "Object is Not Invoker rights" << endl;
  RefAny ref = metaData.getRef (MetaData::ATTR_REF_TDO);
  MetaData md1 = conn->getMetaData (ref);
  vector<MetaData> v1 =
    md1.getVector (MetaData::ATTR_LIST_TYPE_ATTRS);

  for (int i = 0; i < v1.size (); ++i)
  {
    MetaData md2 = (MetaData)v1[i];
    cout << "Column Name :" <<
      (md2.getString(MetaData::ATTR_NAME)) << endl;
    cout << " Data Type :" <<
      (printType (md2.getInt(MetaData::ATTR_DATA_TYPE))) << endl;
    cout << " Size :" << md2.getInt(MetaData::ATTR_DATA_SIZE) << endl;
    cout << " Precision :" << md2.getInt(MetaData::ATTR_PRECISION) << endl;
    cout << " Scale :" << md2.getInt(MetaData::ATTR_SCALE) << endl << endl;
  }

  cout << "describe_type - done" << endl;
} // end of describe_type()

// Describing a table
void describe_table ()
{
  cout << "Describing the table - AUTHOR_TAB" << endl;
  vector<MetaData> v1;
  MetaData metaData = conn->getMetaData("AUTHOR_TAB");
  cout << "Object name:" <<
    (metaData.getString(MetaData::ATTR_OBJ_NAME)) << endl;
  cout << "Schema:" <<
    (metaData.getString(MetaData::ATTR_OBJ_SCHEMA)) << endl;
  if (metaData.getInt(MetaData::ATTR_PTYPE) ==
    MetaData::PTYPE_TABLE)
  {
    cout << "AUTHOR_TAB is a table" << endl;
  }
  else
    cout << "AUTHOR_TAB is not a table" << endl;
  if (metaData.getBoolean(MetaData::ATTR_PARTITIONED))
    cout << "Table is partitioned" << endl;
  else
    cout << "Table is not partitioned" << endl;
  if (metaData.getBoolean(MetaData::ATTR_IS_TEMPORARY))
    cout << "Table is temporary" << endl;
```

```
else
  cout << "Table is not temporary" << endl;
if (metaData.getBoolean(MetaData::ATTR_IS_TYPED))
  cout << "Table is typed" << endl;
else
  cout << "Table is not typed" << endl;
if (metaData.getBoolean(MetaData::ATTR_CLUSTERED))
  cout << "Table is clustered" << endl;
else
  cout << "Table is not clustered" << endl;
if (metaData.getBoolean(MetaData::ATTR_INDEX_ONLY))
  cout << "Table is Index-only" << endl;
else
  cout << "Table is not Index-only" << endl;
cout << "Duration:";
switch (metaData.getInt(MetaData::ATTR_DURATION))
{
  case MetaData::DURATION_SESSION : cout << "Connection" << endl;
                                            break;
  case MetaData::DURATION_TRANS   : cout << "Transaction" << endl;
                                            break;
  case MetaData::DURATION_NULL  : cout << "Table not temporary" << endl;
                                            break;
}
try
{
  cout << "Data Block Address:" <<
    metaData.getUInt (MetaData::ATTR_RDBA) << endl;
}
catch (SQLException ex)
{
  cout << ex.getMessage() << endl;
}
try
{
  cout << "Tablespace:" <<
    metaData.getInt (MetaData::ATTR_TABLESPACE) << endl;
}
catch (SQLException ex)
{
  cout << ex.getMessage() << endl;
}
try
{
  cout << "Object Id:" <<
```

```
      metaData.getUInt(MetaData::ATTR_OBJID) << endl;
  }
  catch (SQLException ex)
  {
    cout << ex.getMessage() << endl;
  }

  int columnCount = metaData.getInt(MetaData::ATTR_NUM_COLS);
  cout << "Number of Columns : " << columnCount << endl;

  v1 =  metaData.getVector(MetaData::ATTR_LIST_COLUMNS);
  for(int i=0; i < v1.size(); i++)
  {
    MetaData md = v1[i];
    cout << " Column Name :" <<
      (md.getString(MetaData::ATTR_NAME)) << endl;
    cout << " Data Type :" <<
      (printType (md.getInt(MetaData::ATTR_DATA_TYPE))) << endl;
    cout << " Size :" << md.getInt(MetaData::ATTR_DATA_SIZE) << endl;
    cout << " Precision :" << md.getInt(MetaData::ATTR_PRECISION) << endl;
    cout << " Scale :" << md.getInt(MetaData::ATTR_SCALE) << endl;
    bool isnull = md.getBoolean(MetaData::ATTR_IS_NULL);
    if (isnull)
      cout << " Allows null" << endl;
    else
      cout << " Does not allow null" << endl;
  }
  cout << "describe_table - done" << endl;
}  // end of describe_table ()

// Describing a procedure
void describe_proc ()
{
  cout << "Describing the procedure - DEMO_PROC" << endl;
  MetaData metaData = conn->getMetaData("DEMO_PROC");
  vector<MetaData> v1 =
    metaData.getVector ( MetaData::ATTR_LIST_ARGUMENTS );
  cout << "The number of arguments are:" << v1.size() << endl;
  cout << "Object Name :" <<
    (metaData.getString(MetaData::ATTR_OBJ_NAME)) << endl;
  cout << "Schema Name :" <<
    (metaData.getString(MetaData::ATTR_OBJ_SCHEMA)) << endl;
  if (metaData.getInt(MetaData::ATTR_PTYPE) == MetaData::
    PTYPE_PROC)
  {
```

```
  cout << "DEMO_PROC is a procedure" << endl;
}
else
{
  if (metaData.getInt(MetaData::ATTR_PTYPE) == MetaData::
    PTYPE_FUNC)
  {
    cout << "DEMO_PROC is a function" << endl;
  }
}
try
{
cout << "Object Id:" <<
  metaData.getUInt(MetaData::ATTR_OBJ_ID) << endl;
}
catch (SQLException ex)
{
  cout << ex.getMessage() << endl;
}
try
{
  cout << "Name :" <<
    (metaData.getString(MetaData::ATTR_NAME)) << endl;
}
catch (SQLException ex)
{
  cout << ex.getMessage() << endl;
}

if (metaData.getBoolean(MetaData::ATTR_IS_INVOKER_RIGHTS))
  cout << "It is Invoker-rights" << endl;
else
  cout << "It is not Invoker-rights" << endl;
cout << "Overload Id:" <<
  metaData.getInt(MetaData::ATTR_OVERLOAD_ID) << endl;

for(int i=0; i < v1.size(); i++)
{
  MetaData md = v1[i];
  cout << "Column Name :" <<
    (md.getString(MetaData::ATTR_NAME)) << endl;
  cout << "DataType :" <<
    (printType (md.getInt(MetaData::ATTR_DATA_TYPE)))
    << endl;
  cout << "Argument Mode:";
```

```
    int mode = md.getInt (MetaData::ATTR_IOMODE);
    if (mode == 0)
      cout << "IN" << endl;
    if (mode == 1)
      cout << "OUT" << endl;
    if (mode == 2)
      cout << "IN/OUT" << endl;
    cout << "Size :" <<
      md.getInt(MetaData::ATTR_DATA_SIZE) << endl;
    cout << "Precision :" <<
      md.getInt(MetaData::ATTR_PRECISION) << endl;
    cout << "Scale :" <<
      md.getInt(MetaData::ATTR_SCALE) << endl;
    int isNull = md.getInt ( MetaData::ATTR_IS_NULL);
    if (isNull != 0)
      cout << "Allows null," << endl;
    else
      cout << "Does not allow null," << endl;

    int hasDef = md.getInt ( MetaData::ATTR_HAS_DEFAULT);
    if (hasDef != 0)
      cout << "Has Default" << endl;
    else
      cout << "Does not have Default" << endl;
  }
  cout << "test1 - done" << endl;
}

// Method which prints the data type
string printType (int type)
{
  switch (type)
  {
    case OCCI_SQLT_CHR : return "VARCHAR2";
                         break;
    case OCCI_SQLT_NUM : return "NUMBER";
                         break;
    case OCCIINT : return "INTEGER";
                         break;
    case OCCIFLOAT : return "FLOAT";
                         break;
    case OCCI_SQLT_STR : return "STRING";
                         break;
    case OCCI_SQLT_VNU : return "VARNUM";
                         break;
```

```
      case OCCI_SQLT_LNG : return "LONG";
                           break;
      case OCCI_SQLT_VCS : return "VARCHAR";
                           break;
      case OCCI_SQLT_RID : return "ROWID";
                           break;
      case OCCI_SQLT_DAT : return "DATE";
                           break;
      case OCCI_SQLT_VBI : return "VARRAW";
                           break;
      case OCCI_SQLT_BIN : return "RAW";
                           break;
      case OCCI_SQLT_LBI : return "LONG RAW";
                           break;
      case OCCIUNSIGNED_INT : return "UNSIGNED INT";
                           break;
      case OCCI_SQLT_LVC : return "LONG VARCHAR";
                           break;
      case OCCI_SQLT_LVB : return "LONG VARRAW";
                           break;
      case OCCI_SQLT_AFC : return "CHAR";
                           break;
      case OCCI_SQLT_AVC : return "CHARZ";
                           break;
      case OCCI_SQLT_RDD : return "ROWID";
                           break;
      case OCCI_SQLT_NTY : return "NAMED DATA TYPE";
                           break;
      case OCCI_SQLT_REF : return "REF";
                           break;
      case OCCI_SQLT_CLOB: return "CLOB";
                           break;
      case OCCI_SQLT_BLOB: return "BLOB";
                           break;
      case OCCI_SQLT_FILE: return "BFILE";
                           break;
    }
  } // End of printType (int)


}; // end of class occidesc

int main (void)
{
  string user = "SCOTT";
```

```
  string passwd = "TIGER";
  string db = "";

  cout << "occidesc - Describing the various objects of the database" << endl;

  occidesc *demo = new occidesc (user, passwd, db);
  demo->describe_table();
  demo->describe_type();
  demo->describe_proc();
  delete demo;

}// end of main ()
```

occidml.cpp

The following code example demonstrates how to perform insert, select, update, and delete operations of a table row by using OCCI:

```
/**
 * occidml - To exhibit the insertion, Selection, updating and deletion of
 *           a row through OCCI.
 *
 * Description
 *   Create a program which has insert, select, update & delete as operations.
 *   Perform all these operations using OCCI interface.
 */

#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

class  occidml
{
  private:

  Environment *env;
  Connection *conn;
  Statement *stmt;
  public:

  occidml (string user, string passwd, string db)
  {
    env = Environment::createEnvironment (Environment::DEFAULT);
```

```
  conn = env->createConnection (user, passwd, db);
}

~occidml ()
{
  env->terminateConnection (conn);
  Environment::terminateEnvironment (env);
}

/**
 * Insertion of a row with dynamic binding, PreparedStatement functionality.
 */
void insertBind (int c1, string c2)
{
  string sqlStmt = "INSERT INTO author_tab VALUES (:x, :y)";
  stmt=conn->createStatement (sqlStmt);
  try{
  stmt->setInt (1, c1);
  stmt->setString (2, c2);
  stmt->executeUpdate ();
  cout << "insert - Success" << endl;
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for insertBind"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
  }

  conn->terminateStatement (stmt);
}

/**
 * Inserting a row into the table.
 */
void insertRow ()
{
  string sqlStmt = "INSERT INTO author_tab VALUES (111, 'ASHOK')";
  stmt = conn->createStatement (sqlStmt);
  try{
  stmt->executeUpdate ();
  cout << "insert - Success" << endl;
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for insertRow"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
```

```
    cout<<ex.getMessage() << endl;
   }

  conn->terminateStatement (stmt);
}

/**
 * updating a row
 */
void updateRow (int c1, string c2)
{
  string sqlStmt =
    "UPDATE author_tab SET author_name = :x WHERE author_id = :y";
  stmt = conn->createStatement (sqlStmt);
  try{
  stmt->setString (1, c2);
  stmt->setInt (2, c1);
  stmt->executeUpdate ();
  cout << "update - Success" << endl;
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for updateRow"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
  }

  conn->terminateStatement (stmt);
}


/**
 * deletion of a row
 */
void deleteRow (int c1, string c2)
{
  string sqlStmt =
    "DELETE FROM author_tab WHERE author_id= :x AND author_name = :y";
  stmt = conn->createStatement (sqlStmt);
  try{
  stmt->setInt (1, c1);
  stmt->setString (2, c2);
  stmt->executeUpdate ();
  cout << "delete - Success" << endl;
  }catch(SQLException ex)
  {
```

```
     cout<<"Exception thrown for deleteRow"<<endl;
     cout<<"Error number: "<<  ex.getErrorCode() << endl;
     cout<<ex.getMessage() << endl;
    }

    conn->terminateStatement (stmt);
  }

  /**
   * displaying all the rows in the table
   */
  void displayAllRows ()
  {
    string sqlStmt = "SELECT author_id, author_name FROM author_tab";
    stmt = conn->createStatement (sqlStmt);
    ResultSet *rset = stmt->executeQuery ();
    try{
    while (rset->next ())
    {
      cout << "author_id: " << rset->getInt (1) << "  author_name: "
        << rset->getString (2) << endl;
    }
    }catch(SQLException ex)
    {
     cout<<"Exception thrown for displayAllRows"<<endl;
     cout<<"Error number: "<<  ex.getErrorCode() << endl;
     cout<<ex.getMessage() << endl;
    }

    stmt->closeResultSet (rset);
    conn->terminateStatement (stmt);
  }

}; // end of class  occidml


int main (void)
{
  string user = "SCOTT";
  string passwd = "TIGER";
  string db = "";

  cout << "occidml - Exhibiting simple insert, delete & update operations"
    << endl;
  occidml *demo = new occidml (user, passwd, db);
```

```
  cout << "Displaying all records before any operation" << endl;
  demo->displayAllRows ();

  cout << "Inserting a record into the table author_tab "
    << endl;
  demo->insertRow ();

  cout << "Displaying the records after insert " << endl;
  demo->displayAllRows ();

  cout << "Inserting a records into the table author_tab using dynamic bind"
    << endl;
  demo->insertBind (222, "ANAND");

  cout << "Displaying the records after insert using dynamic bind" << endl;
  demo->displayAllRows ();

  cout << "deleting a row with author_id as 222 from author_tab table" << endl;
  demo->deleteRow (222, "ANAND");

  cout << "updating a row with author_id as 444 from author_tab table" << endl;
  demo->updateRow (444, "ADAM");

  cout << "displaying all rows after all the operations" << endl;
  demo->displayAllRows ();

  delete (demo);
  cout << "occidml - done" << endl;
}
```

occiinh.typ

```
CASE=LOWER
MAPFILE=occiinhm.cpp
TYPE FOREIGN_STUDENT as foreign_student
```

occiinh.cpp

The following code example demonstrates object inheritance by using insert, select, update, and delete operations on a table row of subtype table:

```
/**
```

```
 * occiinh.cpp - To exhibit the insertion, selection, updating and deletion
 *                 of a row of a table of derived object.
 *
 * Description
 *   Create a program which has insert, select, update & delete as operations
 *   of a object.  Perform all these operations using OCCI interface.
 *   Hierarchy
 *   person_typ <---- student <----- parttime_stud <----- foreign_student
**/

#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

#include "occiinhm.h"

/* Add on your methods in this class*/
class foreign_student_obj : public foreign_student
{
   /* New methods can be added here */
};

class occiinh
{
  private:

  Environment *env;
  Connection *con;

  // This method will return the Ref
  RefAny getRefObj(string sqlString)
  {
    Statement *stmt = con->createStatement (sqlString);
    ResultSet *rs;
    try
    {
       rs = stmt->executeQuery ();
       if ( rs->next() )
       {
          RefAny ref1 = rs->getRef (1);
          stmt->closeResultSet (rs);
          con->terminateStatement (stmt);
          return ref1;
       }
```

```
  }
  catch(SQLException ex)
  {
    cout << "Error in fetching ref" << endl;
  }
  stmt->closeResultSet (rs);
  con->terminateStatement (stmt);
}


public:

occiinh (string user, string passwd, string db)
 throw (SQLException)
{
  env = Environment::createEnvironment (Environment::OBJECT);
  occiinhm(env);
  con = env->createConnection (user, passwd, db);
}// end of constructor occiinh (string, string, string)

~occiinh ()
 throw (SQLException)
{
  env->terminateConnection (con);
  Environment::terminateEnvironment (env);
}// end of destructor

/**
 * Insertion of a row
 */
void insertRow ()
 throw (SQLException)
{
  cout << "Inserting a record (joe)" << endl;
  string sqlStmt =
  "INSERT INTO foreign_student_tab VALUES(:a)";
  Statement *stmt = con->createStatement (sqlStmt);
  string fs_name = "joe";
  Number fs_ssn (4);
  Date fs_dob(env, 2000, 5, 11, 16, 05, 0);
  Number fs_stud_id (400);
  Ref< person_typ > fs_teammate = getRefObj(
     "SELECT REF(a) FROM person_tab a where name='john'");
  Number fs_course_id(4000);
  Ref< student > fs_partner = getRefObj(
```

```
    "SELECT REF(a) FROM student_tab a");
  string fs_country = "india";
  Ref< parttime_stud > fs_leader = getRefObj(
    "SELECT REF(a) FROM parttime_stud_tab a");
  foreign_student_obj fsobj;
  foreign_student_obj *fs_obj=&fsobj;
  fs_obj->setname(fs_name);
  fs_obj->setssn(fs_ssn);
  fs_obj->setdob(fs_dob);
  fs_obj->setstud_id(fs_stud_id);
  fs_obj->setteammate(fs_teammate);
  fs_obj->setcourse_id(fs_course_id);
  fs_obj->setpartner(fs_partner);
  fs_obj->setcountry(fs_country);
  fs_obj->setleader(fs_leader);
  stmt->setObject(1, fs_obj);
  stmt->executeUpdate();

  con->terminateStatement (stmt);
  cout << "Insertion Successful" << endl;
}// end of insertRow ();


/**
 * updating a row
 */
void updateRow ()
 throw (SQLException)
{
  cout << "Upadating record (Changing name,teammate and course_id)" << endl;
  string sqlStmt =
   "UPDATE foreign_student_tab SET name=:x, teammate=:y, course_id=:z";
  Statement *stmt = con->createStatement (sqlStmt);
  string fs_name = "jeffree";
  Ref< person_typ > fs_teammate = getRefObj(
     "SELECT REF(a) FROM person_tab a where name='jill'");
  Number fs_course_id(5000);
  stmt->setString(1, fs_name);
  stmt->setRef(2,fs_teammate);
  stmt->setInt(3, fs_course_id);
  stmt->executeUpdate ();
  con->commit();
  con->terminateStatement (stmt);
  cout << "Updation Successful" << endl;
}// end of updateRow (int, string);
```

```
/**
 * deletion of a row
 */
void deleteRow ()
 throw (SQLException)
{
  cout << "Deletion of jeffree record " << endl;
  string sqlStmt = "DELETE FROM foreign_student_tab where name=:x";
  Statement *stmt = con->createStatement (sqlStmt);
  string fs_name = "jeffree";
  stmt->setString(1,fs_name);
  stmt->executeUpdate();
  con->commit();
  con->terminateStatement (stmt);
  cout << "Deletion Successful" << endl;
}// end of deleteRow (int, string);

/**
 * displaying all the rows in the table
 */
void displayAllRows ()
 throw (SQLException)
{
  int count=0;
  string sqlStmt = "SELECT REF(a) FROM foreign_student_tab a";
  Statement *stmt = con->createStatement (sqlStmt);
  ResultSet *resultSet = stmt->executeQuery ();

  while (resultSet->next ())
  {
     count++;
     RefAny fs_refany = resultSet->getRef(1);
     Ref <foreign_student_obj> fs_ref(fs_refany);
     fs_ref.setPrefetch(4);
     string fmt = "DD-MON-YYYY";
     string nlsParam = "NLS_DATE_LANGUAGE = American";
     Date fs_dob = fs_ref->getdob();
     string date1 = fs_dob.toText (fmt, nlsParam);
     cout << "Foreign Student Information" << endl;
     cout << "Name      : " << fs_ref->getname();
     cout << "  SSN       : " << (int)fs_ref->getssn();
     cout << "  DOB       : " << date1 << endl;
     cout << "Stud id   : " << (int)fs_ref->getstud_id() ;
```

```
    cout << "  Course id : " << (int)fs_ref->getcourse_id();
    cout << "  Country   : " << fs_ref->getcountry() <<endl;
    Ref <person_typ> fs_teammate = (Ref <person_typ>)
      fs_ref->getteammate();
    cout << "Teammate's Information " << endl;
    cout << "Name      : " << fs_teammate->getname();
    cout << "  SSN       : " << (int)fs_teammate->getssn();
    fs_dob = fs_teammate->getdob();
    date1 = fs_dob.toText(fmt, nlsParam);
    cout << "  DOB       : " << date1 << endl << endl;
    /* Leader */
    Ref< parttime_stud > fs_leader = (Ref < parttime_stud >)
       fs_ref->getleader();
    /* Leader's Partner */
    Ref < student > fs_partner = (Ref <student> )
       fs_leader->getpartner();
   /* Leader's Partenr's teammate */
     fs_teammate = (Ref <person_typ>) fs_partner->getteammate();

    cout << "Leader Information " << endl;
    cout << "Name      : " << fs_leader->getname();
    cout << "  SSN       : " << (int)fs_leader->getssn();
    fs_dob = fs_leader->getdob();
    date1 = fs_dob.toText(fmt, nlsParam);
    cout << "  DOB       : " << date1 << endl;
    cout << "Stud id   : " << (int)fs_leader->getstud_id();
    cout << "  Course id : " << (int)fs_leader->getcourse_id() << endl;

    cout << "Leader's Partner's Information " << endl;
    cout << "Name      : " << fs_partner->getname() ;
    cout << "  SSN       : " << (int)fs_partner->getssn();
    fs_dob = fs_partner->getdob();
    date1 = fs_dob.toText(fmt, nlsParam);
    cout << "  DOB       : " << date1 ;
    cout << "  Stud id   : " << (int)fs_partner->getstud_id() << endl;


    cout << "Leader's Partner's Teammate's Information " << endl;
    cout << "Name      : " << fs_teammate->getname();
    cout << "  SSN       : " << (int)fs_teammate->getssn();
    fs_dob = fs_teammate->getdob();
    date1 = fs_dob.toText(fmt, nlsParam);
    cout << "  DOB       : " << date1 << endl << endl;

}//end of while (resultSet->next ());
```

```
    if (count <=0)
       cout << "No record found " << endl;
    stmt->closeResultSet (resultSet);
    con->terminateStatement (stmt);
  }// end of updateRow (string);

}; // end of class occiinh


int main (void)
{
  string user = "scott";
  string passwd = "tiger";
  string db = "";

  try
  {
    cout << "occiinh - Exhibiting simple insert, delete & update operations"
     " on Oracle objects" << endl;
    occiinh *demo = new occiinh (user, passwd, db);

    cout << "displaying all rows before operations" << endl;
    demo->displayAllRows ();

    demo->insertRow ();
    cout << "displaying all rows after insertions" << endl;
    demo->displayAllRows ();

    demo->updateRow ();
    cout << "displaying all rows after updations" << endl;
    demo->displayAllRows ();

    demo->deleteRow ();
    cout << "displaying all rows after deletions" << endl;
    demo->displayAllRows ();


    delete (demo);
    cout << "occiinh - done" << endl;
  }catch (SQLException ea)
  {
    cerr << "Error running the demo: " << ea.getMessage () << endl;
  }
}// end of int main (void);
```

occiobj.typ

```
CASE=SAME
MAPFILE=occiobjm.cpp
TYPE address as address
```

occiobj.cpp

The following code example demonstrates how to perform insert, select, update, and delete operations on a table row containing an object as one of its columns:

```
/**
 * occiobj.cpp - To exhibit the insertion, selection, updating and deletion
 *               of a row containing object as one of the column.
 *
 * Description
 *   Create a program which has insert, select, update & delete as operations
 *   of a object.  Perform all these operations using OCCI interface.
 */

#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

#include "occiobjm.h"

class occiobj
{
  private:

  Environment *env;
  Connection *con;
  Statement *stmt;
  public:

  occiobj (string user, string passwd, string db)
  {
    env = Environment::createEnvironment (Environment::OBJECT);
    occiobjm (env);
    con = env->createConnection (user, passwd, db);
  }

  ~occiobj ()
```

```
{
  env->terminateConnection (con);
  Environment::terminateEnvironment (env);
}

/**
 * Insertion of a row
 */
void insertRow (int c1, int a1, string a2)
{
  cout << "Inserting record - Publisher id :" << c1 <<
    ", Publisher address :" << a1 << ", " << a2 <<endl;
  string sqlStmt = "INSERT INTO publisher_tab VALUES (:x, :y)";
  try{
  stmt = con->createStatement (sqlStmt);
  stmt->setInt (1, c1);
  address *o = new address ();
  o->setStreet_no (Number ( a1));
  o->setCity (a2);
  stmt->setObject (2, o);
  stmt->executeUpdate ();
  cout << "Insert - Success" << endl;
  delete (o);
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for insertRow"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
  }

  con->terminateStatement (stmt);
}


/**
 * updating a row
 */
void updateRow (int c1, int a1, string a2)
{
  cout << "Upadating record with publisher id :"<< c1 << endl;
  string sqlStmt =
    "UPDATE publisher_tab SET publisher_add= :x WHERE publisher_id = :y";
  try{
  stmt = con->createStatement (sqlStmt);
  address *o = new address ();
```

```
  o->setStreet_no (Number ( a1));
  o->setCity (a2);
  stmt->setObject (1, o);
  stmt->setInt (2, c1);
  stmt->executeUpdate ();
  cout << "Update - Success" << endl;
  delete (o);
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for updateRow"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
  }
  con->terminateStatement (stmt);
}


/**
 * deletion of a row
 */
void deleteRow (int c1, int a1, string a2)
{
  cout << "Deletion of record where publisher id :" << c1 <<endl;
  string sqlStmt =
  "DELETE FROM publisher_tab WHERE publisher_id= :x AND publisher_add = :y";
  try{
  stmt = con->createStatement (sqlStmt);
  stmt->setInt (1, c1);

  address *o = new address ();
  o->setStreet_no (Number ( a1));
  o->setCity (a2);
  stmt->setObject (2, o);
  stmt->executeUpdate ();
  cout << "Delete - Success" << endl;
  delete (o);
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for deleteRow"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
  }

  con->terminateStatement (stmt);
}
```

```
  /**
   * displaying all the rows in the table
   */
  void displayAllRows ()
  {
    string sqlStmt = "SELECT publisher_id, publisher_add FROM publisher_tab";
    try{
    stmt = con->createStatement (sqlStmt);
    ResultSet *rset = stmt->executeQuery ();

    while (rset->next ())
    {
      cout << "publisher id: " << rset->getInt (1)
           << " publisher address: address (" ;
      address *o =  (address *)rset->getObject (2);
      cout << (int)o->getStreet_no () << ", " << o->getCity () << ")" << endl;
    }

    stmt->closeResultSet (rset);
    }catch(SQLException ex)
    {
     cout<<"Exception thrown for displayAllRows"<<endl;
     cout<<"Error number: "<<  ex.getErrorCode() << endl;
     cout<<ex.getMessage() << endl;
    }

    con->terminateStatement (stmt);
  }

};//end of class occiobj;


int main (void)
{
  string user = "SCOTT";
  string passwd = "TIGER";
  string db = "";

  try
  {
    cout << "occiobj - Exhibiting simple insert, delete & update operations"
     " on Oracle objects" << endl;
    occiobj *demo = new occiobj (user, passwd, db);
```

```
    cout << "displaying all rows before operations" << endl;
    demo->displayAllRows ();

    demo->insertRow (12, 122, "MIKE");

    demo->deleteRow (11, 121, "ANNA");

    demo->updateRow (23, 123, "KNUTH");

    cout << "displaying all rows after all operations" << endl;
    demo->displayAllRows ();

    delete (demo);
    cout << "occiobj - done" << endl;
  }catch (SQLException ea)
  {
    cerr << "Error running the demo: " << ea.getMessage () << endl;
  }
}
```

occipobj.typ

```
CASE=SAME
MAPFILE=occipobjm.cpp
TYPE address as  address
```

occipobj.cpp

The following code example demonstrates how to perform insert, select, and update operations on persistent objects, as well as how to pin, unpin, mark for deletion, and flush a persistent object:

```
/**
 * occipobj.cpp - Manipulation (Insertion, selection & updating) of
 *                persistant objects, along with pinning, unpinning, marking
 *                for deletion & flushing.
 *
 * Description
 *   Create a program which has insert, select, update & delete as operations
 *   of a persistant object.  Along with the these the operations on Ref. are
 *   pinning, unpinning, marked for deletion & flushing.
 *
 */
```

```
#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

#include "occipobjm.h"

class address_obj : public address
{
  public:
   address_obj()
   {
   }

   address_obj(Number sno,string cty)
   {
     setStreet_no(sno);
     setCity(cty);
   }
};

class occipobj
{
  private:

  Environment *env;
  Connection *conn;
  Statement *stmt;
  string tableName;
  string typeName;

  public:

  occipobj (string user, string passwd, string db)
  {
    env = Environment::createEnvironment (Environment::OBJECT);
    occipobjm (env);
    conn = env->createConnection (user, passwd, db);
  }

  ~occipobj ()
  {
    env->terminateConnection (conn);
    Environment::terminateEnvironment (env);
```

```
}

void setTableName (string s)
{
  tableName = s;
}

/**
 * Insertion of a row
 */
void insertRow (int a1, string a2)
{
  cout << "Inserting row ADDRESS (" << a1 << ", " << a2 << ")" << endl;
  Number n1(a1);
  address_obj *o = new (conn, tableName) address_obj(n1, a2);
  conn->commit ();
  cout << "Insertion - Successful" << endl;
}

/**
 * updating a row
 */
void updateRow (int b1, int a1, string a2)
{
  cout << "Updating a row with attribute a1 = " << b1 << endl;
  stmt = conn->createStatement
    ("SELECT REF(a) FROM address_tab a WHERE street_no = :x FOR UPDATE");
  stmt->setInt (1, b1);
  ResultSet *rs = stmt->executeQuery ();
  try{
  if ( rs->next() )
  {
    RefAny rany = rs->getRef (1);
    Ref <address_obj > r1(rany);
    address_obj *o = r1.ptr();
    o->markModified ();
    o->setStreet_no (Number (a1));
    o->setCity (a2);
    o->flush ();
  }
  }catch(SQLException ex)
  {
   cout<<"Exception thrown updateRow"<<endl;
   cout<<"Error number: "<<  ex.getErrorCode() << endl;
   cout<<ex.getMessage() << endl;
```

```
    }

    conn->commit ();
    conn->terminateStatement (stmt);
    cout << "Updation - Successful" << endl;
  }


  /**
   * deletion of a row
   */
  void deleteRow (int a1, string a2)
  {
    cout << "Deleting a row with object ADDRESS (" << a1 << ", " << a2
     << ")" << endl;
    stmt = conn->createStatement
    ("SELECT REF(a) FROM address_tab a WHERE street_no = :x AND city = :y FOR
UPDATE");
    stmt->setInt (1, a1);
    stmt->setString (2, a2);
    ResultSet *rs = stmt->executeQuery ();
    try{
    if ( rs->next() )
    {
      RefAny rany = rs->getRef (1);
      Ref<address_obj > r1(rany);
      address_obj *o = r1.ptr();
      o->markDelete ();
    }
    }catch(SQLException ex)
    {
     cout<<"Exception thrown for deleteRow"<<endl;
     cout<<"Error number: "<<  ex.getErrorCode() << endl;
     cout<<ex.getMessage() << endl;
    }

    conn->commit ();
    conn->terminateStatement (stmt);
    cout << "Deletion - Successful" << endl;
  }

  /**
   * displaying all the rows in the table
   */
  void displayAllRows ()
```

```
  {
    string sqlStmt = "SELECT REF (a) FROM address_tab a";
    stmt = conn->createStatement (sqlStmt);
    ResultSet *rset = stmt->executeQuery ();
    try{
    while (rset->next ())
    {
      RefAny rany = rset->getRef (1);
      Ref<address_obj > r1(rany);
      address_obj *o = r1.ptr();
      cout << "ADDRESS(" << (int)o->getStreet_no () << ", " << o->getCity () <<
")" << endl;
    }
    }catch(SQLException ex)
    {
     cout<<"Exception thrown for displayAllRows"<<endl;
     cout<<"Error number: "<<  ex.getErrorCode() << endl;
     cout<<ex.getMessage() << endl;
    }

    stmt->closeResultSet (rset);
    conn->terminateStatement (stmt);
  }

}; // end of class occipobj


int main (void)
{
  string user = "SCOTT";
  string passwd = "TIGER";
  string db = "";

  try
  {
    cout << "occipobj - Exhibiting simple insert, delete & update operations"
     " on persistent objects" << endl;
    occipobj *demo = new occipobj (user, passwd, db);

    cout << "Displaying all rows before the opeations" << endl;
    demo->displayAllRows ();

    demo->setTableName ("ADDRESS_TAB");

    demo->insertRow (21, "KRISHNA");
```

```
    demo->deleteRow (22, "BOSTON");

    demo->updateRow (33, 123, "BHUMI");

    cout << "Displaying all rows after all the operations" << endl;
    demo->displayAllRows ();

    delete (demo);
    cout << "occipobj - done" << endl;
  }catch (SQLException ea)
  {
    cerr << "Error running the demo: " << ea.getMessage () << endl;
  }
}
```

occipool.cpp

The following code example demonstrates how to use the connection pool interface of OCCI:

```
/**
*  occipool - Demontrating the Connection Pool interface of OCCI.
*
*  DESCRIPTION :
*    This program demonstates the creating and using of connection pool in the
*    database and fetching records of a table.
*
*/

#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

class occipool
{
  private:

  Environment *env;
  Connection *con;
  Statement *stmt;
  public :
  /**
```

```
 * Constructor for the occipool test case.
 */
occipool ()
{
  env = Environment::createEnvironment (Environment::DEFAULT);
}// end of constructor occipool ()

/**
 * Destructor for the occipool test case.
 */
~occipool ()
{
  Environment::terminateEnvironment (env);
}  // end of ~occipool ()

/**
 * The testing logic of the test case.
 */
dvoid select ()
{
  cout << "occipool - Selecting records using ConnectionPool interface" <<
  endl;
  const string poolUserName = "SCOTT";
  const string poolPassword = "TIGER";
  const string connectString = "";
  const string username = "SCOTT";
  const string passWord = "TIGER";
  unsigned int maxConn = 5;
  unsigned int minConn = 3;
  unsigned int incrConn = 2;
  ConnectionPool *connPool = env->createConnectionPool
    (poolUserName, poolPassword, connectString, minConn, maxConn, incrConn);
  try{
  if (connPool)
    cout << "SUCCESS - createConnectionPool" << endl;
  else
    cout << "FAILURE - createConnectionPool" << endl;
  con = connPool->createConnection (username, passWord);
  if (con)
    cout << "SUCCESS - createConnection" << endl;
  else
    cout << "FAILURE - createConnection" << endl;
  }catch(SQLException ex)
  {
   cout<<"Exception thrown for createConnectionPool"<<endl;
```

```
     cout<<"Error number: "<<  ex.getErrorCode() << endl;
     cout<<ex.getMessage() << endl;
    }

    cout << "retrieving the data" << endl;
    stmt = con->createStatement
      ("SELECT author_id, author_name FROM author_tab");
    ResultSet *rset = stmt->executeQuery();
    while (rset->next())
    {
      cout << "author_id:" << rset->getInt (1) << endl;
      cout << "author_name:" << rset->getString (2) << endl;
    }
    stmt->closeResultSet (rset);
    con->terminateStatement (stmt);
    connPool->terminateConnection (con);
    env->terminateConnectionPool (connPool);

    cout << "occipool - done" << endl;
  } // end of test (Connection *)

}; // end of class occipool

int main (void)
{
  string user = "SCOTT";
  string passwd = "TIGER";
  string db = "";

  occipool *demo = new occipool ();

  demo->select();
  delete demo;

}// end of main ()
```

occiproc.cpp

The following code example demonstrates how to invoke PL/SQL procedures with bind parameters:

```
/**
*  occiproc - Demonstrating the invoking of a PL/SQL function and procedure
*  using OCCI.
```

```
*
*  DESCRIPTION :
*    This program demonstrates the invoking a PL/SQL function and procedure
*    having IN, IN/OUT and OUT parameters.
*
*/

#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

class occiproc
{
  private:

  Environment *env;
  Connection *con;

  public :
  /**
   * Constructor for the occiproc demo program.
   */
  occiproc (string user, string passwd, string db) throw (SQLException)
  {
    env = Environment::createEnvironment (Environment::DEFAULT);
    con = env->createConnection (user, passwd, db);
  }// end of constructor occiproc (string, string, string )

  /**
   * Destructor for the occiproc demo program.
   */
  ~occiproc () throw (SQLException)
  {
    env->terminateConnection (con);
    Environment::terminateEnvironment (env);
  }  // end of ~occiproc ()

  // Function to call a PL/SQL procedure
  void callproc ()
  {
    cout << "callproc - invoking a PL/SQL procedure having IN, OUT and IN/OUT ";
    cout << "parameters" << endl;
    Statement *stmt = con->createStatement
      ("BEGIN demo_proc(:v1, :v2, :v3); END;");
```

```
    cout << "Executing the block :" << stmt->getSQL() << endl;
    stmt->setInt (1, 10);
    stmt->setString (2, "IN");
    stmt->registerOutParam (2, OCCISTRING, 30, "");
    stmt->registerOutParam (3, OCCISTRING, 30, "");
    int updateCount = stmt->executeUpdate ();
    cout << "Update Count:" << updateCount << endl;

    string c1 = stmt->getString (2);
    string c2 = stmt->getString (3);
    cout << "Printing the INOUT & OUT parameters:" << endl;
    cout << "Col2:" << c1 << " Col3:" << c2 << endl;

    con->terminateStatement (stmt);
    cout << "occiproc - done" << endl;
  } // end of callproc ()

  // Function to call a PL/SQL function
  void callfun ()
  { cout << "callfun - invoking a PL/SQL function having IN, OUT and IN/OUT ";
    cout << "parameters" << endl;
    Statement *stmt = con->createStatement
      ("BEGIN :a := demo_fun(:v1, :v2, :v3); END;");
    cout << "Executing the block :" << stmt->getSQL() << endl;
    stmt->setInt (2, 10);
    stmt->setString (3, "IN");
    stmt->registerOutParam (1, OCCISTRING, 30, "");
    stmt->registerOutParam (3, OCCISTRING, 30, "");
    stmt->registerOutParam (4, OCCISTRING, 30, "");
    int updateCount = stmt->executeUpdate ();
    cout << "Update Count : " << updateCount << endl;

    string c1 = stmt->getString (1);
    string c2 = stmt->getString (3);
    string c3 = stmt->getString (4);

    cout << "Printing the INOUT & OUT parameters :" << endl;
    cout << "Col2:" << c2 << " Col3:" << c3 << endl;
    cout << "Printing the return value of the function :";
    cout << c1 << endl;

    con->terminateStatement (stmt);
    cout << "occifun - done" << endl;
  } // end of callfun ()
}; // end of class occiproc
```

```
int main (void)
{
  string user = "SCOTT";
  string passwd = "TIGER";
  string db = "";

  cout << "occiproc - invoking a PL/SQL function and procedure having ";
  cout << "parameters" << endl;

  occiproc *demo = new occiproc (user, passwd, db);
  demo->callproc();
  demo->callfun();
  delete demo;

}// end of main ()
```

occistre.cpp

The following code example demonstrates how to use OCCI `ResultSet` streams:

```
/**
 * occistrm - Demonstrating the usage of streams for VARCHAR2 data
 *
 * Description
 *   This demo program selects VARCHAR2 data using stream operations.
 */

#include <iostream.h>
#include <occi.h>
using namespace oracle::occi;
using namespace std;

class occistrm
{
  private:

  Environment *env;
  Connection *conn;

  public:

  occistrm (string user, string passwd, string db)
   throw (SQLException)
```

```
  {
    env = Environment::createEnvironment (Environment::DEFAULT);
    conn = env->createConnection (user, passwd, db);
  }// end of constructor occistrm (string, string, string)

  ~occistrm ()
   throw (SQLException)
  {
    env->terminateConnection (conn);
    Environment::terminateEnvironment (env);
  }// end of destructor

  /**
   * displaying all the rows in the table
   */
  void displayAllRows ()
  {
    Statement *stmt = conn->createStatement (
      "SELECT summary FROM book WHERE bookid = 11");
    stmt->execute ();
    ResultSet *rs = stmt->getResultSet ();
    rs->setCharacterStreamMode(1, 4000);
    char buffer[500];
    int length = 0;
    unsigned int size = 500;

    while (rs->next ())
    {
      Stream *stream = rs->getStream (1);
      while( (length=stream->readBuffer(buffer, size))!=-1)
      {
        cout << "Read " << length << " bytes from stream" << endl;
      }
    }
    stmt->closeResultSet (rs);
    conn->terminateStatement (stmt);
  }// end of updateRow (string);

}; // end of class occistrm


int main (void)
{
  string user = "SCOTT";
  string passwd = "TIGER";
```

```
  string db = "";

  cout << "occistrm - Exhibiting usage of streams for VARCHAR2 data"
    << endl;
  occistrm *demo = new occistrm (user, passwd, db);

  demo->displayAllRows ();

  delete (demo);
  cout << "occistrm - done" << endl;
}// end of int main (void);
```

Index

A

application-provided serialization, 2-26
associative access
  overview, 3-11
atomic nullness, 3-21
attributes, 1-9
automatic serialization, 2-25

B

Bfile class, 8-5
  methods, 8-6
BFILEs
  datatype, 5-3
  external datatype, 4-8
bind operations
  in bind operations, 1-9
  out bind operations, 1-9
Blob class, 8-13
  methods, 8-14
BLOBs
  datatype, 5-2
  external datatype, 4-8
Bytes class, 8-24
  methods, 8-24

C

callable statements, 2-10
  with arrays as parameters, 2-11
CASE OTT parameter, 7-91
CHAR
  external datatype, 4-8
classes
  Bfile class, 8-5
  Blob class, 8-13
  Bytes class, 8-24
  Clob class, 8-27
  Connection class, 8-40
  ConnectionPool, 8-45
  Date class, 8-51
  Environment class, 8-64
  IntervalDS class, 8-70
  IntervalYM class, 8-83
  Map class, 8-95
  Metadata class, 8-97
  Number class, 8-104
  PObject class, 8-130
  Ref class, 8-137
  RefAny class, 8-144
  ResultSet, 2-14
  ResultSet class, 8-147
  SQLException class, 8-169
  Statement class, 8-171
  Stream class, 8-215
  summary, 8-2
  Timestamp class, 8-219
Clob class, 8-27
  methods, 8-28
CLOBs
  datatype, 5-2
  external datatype, 4-9
code
  example programs, A-1
  list of demonstration programs, A-1
CODE OTT parameter, 7-92
collections

working with, 3-19
committing a transaction, 2-20
complex object retrieval
complex object, 3-16
depth level, 3-16
implementing, 3-17
overview, 3-16
prefetch limit, 3-16
root object, 3-16
complex objects, 3-16
prefetching, 3-19
retrieving, 3-17
CONFIG OTT parameter, 7-92
configuration files
and the OTT utility, 7-10
connecting to a database, 2-2
Connection class, 8-40
methods, 8-40
connection pool
createConnectionPool method, 2-4
creating, 2-3, 2-4, 2-5
ConnectionPool class, 8-45
methods, 8-45
control statements, 1-6
copy semantics
internal LOBs, 5-2

D

data conversions
DATE datatype, 4-22
internal datatypes, 4-20
INTERVAL datatypes, 4-22
LOB datatype, 4-22
TIMESTAMP datatypes, 4-22
data definition language (DDL) statements, 1-5
data manipulation language (DML) statements, 1-6
database
connecting to, 2-2
datatypes, 4-1
LOBs
external LOBs, 5-3
internal LOBs, 5-2
OTT mappings, 7-16
overview, 4-2
types
external datatypes, 4-2, 4-5
internal datatypes, 4-2, 4-3
DATE
external datatype, 4-9
data conversion, 4-22
Date class, 8-51
methods, 8-52
DDL statements
executing, 2-6
depth level, 3-16
DML statements, 1-6
executing, 2-6

E

embedded objects, 3-3
creating, 3-4
fetching, 3-20
prefetching, 3-20
Environment class, 8-64
methods, 8-64
error handling, 2-20
ERRTYPE OTT parameter, 7-93
executing SQL queries, 2-13
executing statements dynamically, 2-16
external datatypes, 4-5, 4-8
BFILE, 4-8
BLOB, 4-8
CHAR, 4-8
CHARZ, 4-9
CLOB, 4-9
DATE, 4-9
FLOAT, 4-11
INTEGER, 4-11
INTERVAL DAY TO SECOND, 4-11
INTERVAL YEAR TO MONTH, 4-12
LONG, 4-12
LONG RAW, 4-13
LONG VARCHAR, 4-13
LONG VARRAW, 4-13
NCLOB, 4-13
NUMBER, 4-13
OCCI BFILE, 4-14
OCCI BLOB, 4-14

OCCI BYTES, 4-14
OCCI CLOB, 4-15
OCCI DATE, 4-15
OCCI INTERVALDS, 4-15
OCCI INTERVALYM, 4-15
OCCI NUMBER, 4-15
OCCI POBJECT, 4-15
OCCI REF, 4-16
OCCI REFANY, 4-16
OCCI STRING, 4-16
OCCI TIMESTAMP, 4-16
OCCI VECTOR, 4-16
RAW, 4-16
REF, 4-17
ROWID, 4-17
STRING, 4-17
TIMESTAMP, 4-17
TIMESTAMP WITH LOCAL TIME ZONE, 4-18
TIMESTAMP WITH TIME ZONE, 4-18
UNSIGNED INT, 4-18
VARCHAR, 4-19
VARCHAR2, 4-19
VARNUM, 4-19
VARRAW, 4-19
external LOBs
BFILE, 5-3

F

FLOAT
external datatype, 4-11
freeing objects, 3-22

H

HFILE OTT parameter, 7-93

I

INITFILE OTT parameter, 7-94
INITFUNC OTT parameter, 7-94
INTEGER
external datatype, 4-11
internal datatypes, 4-3
CHAR, 4-4
LONG, 4-4
LONG RAW, 4-4
RAW, 4-4
VARCHAR2, 4-4
internal LOBs
BLOB, 5-2
CLOB, 5-2
NCLOB, 5-2
INTERVAL DAY TO SECOND
external datatype, 4-11
INTERVAL YEAR TO MONTH
external datatype, 4-12
IntervalDS class, 8-70
methods, 8-72
IntervalYM class, 8-83
methods, 8-84
INTYPE file
structure of, 7-98
INTYPE OTT parameter, 7-94

L

LOB locators
external LOBs, 5-4
internal LOBs, 5-4
LOBs
classes, 5-4, 5-5
closing, 5-8
copy semantics
internal LOBs, 5-2
creating, 5-7
datatypes
BFILE, 5-3
BLOB, 5-2
CLOB, 5-2
NCLOB, 5-2
external datatype
data conversion, 4-22
improving read and write performance, 5-14
using getChunckSize method, 5-15
using writeChunk method, 5-15
LOB locators, 5-3, 5-4
LOB value, 5-3
inline storage, 5-4
methods, 5-4, 5-5

nonstreamed read, 5-10
nonstreamed write, 5-12
opening, 5-8
overview, 5-2
reading, 5-10
reference semantics
external LOBs, 5-3
restrictions, 5-9
streamed read, 5-13
streamed write, 5-14
types
external LOBs, 5-3
internal LOBs, 5-2
updating, 5-15
writing, 5-10
LONG
external datatype, 4-12
LONG RAW
external datatype, 4-13
LONG VARCHAR
external datatype, 4-13

M

manipulating object attributes, 3-15
Map class, 8-95
methods, 8-95
metadata
argument and result attributes, 6-21
attribute groupings, 6-4
argument and result attributes, 6-4
collection attributes, 6-4
column attributes, 6-4
database attributes, 6-5
list attributes, 6-4
package attributes, 6-4
parameter attributes, 6-4
procedure, function, and subprogram attributes, 6-4
schema attributes, 6-5
sequence attributes, 6-4
synonym attributes, 6-4
table and view attributes, 6-4
type attribute attibutes, 6-4
type attributes, 6-4
type method attributes, 6-4
attributes, 6-9
code example, 6-5
collection attributes, 6-17
column attributes, 6-20
database attributes, 6-24
describing database objects, 6-3
list attributes, 6-23
overview, 6-2
package attributes, 6-13
parameter attributes, 6-10
procedure, function, and subprogram attributes, 6-12
schema attributes, 6-24
sequence attributes, 6-19
synonym attributes, 6-19
table and view attributes, 6-11
type attribute attributes, 6-15
type attributes, 6-13
type methods attributes, 6-16
MetaData class, 8-97
methods, 8-98
methods, 1-9
Bfile methods, 8-6
Blob methods, 8-14
Bytes methods, 8-24
Clob methods, 8-28
Connection methods, 8-40
ConnectionPool methods, 8-45
createConnection method, 2-3
createConnectionPool method, 2-4
createEnvironment method, 2-3
createProxyConnection method, 2-5, 2-6
createStatement method, 2-6
Date methods, 8-52
Environment class, 8-64
execute method, 2-6
executeArrayUpdate method, 2-7, 2-28
executeQuery method, 2-6
executeUpdate method, 2-6
IntervalDS methods, 8-72
IntervalYM methods, 8-84
Map methods, 8-95
MetaData methods, 8-98
Number methods, 8-107

PObject methods, 8-130
Ref methods, 8-138
RefAny methods, 8-144, 8-148
setDataBuffer method, 2-27
SQLException methods, 8-169
Statement methods, 8-171
Stream methods, 8-215
terminateConnection method, 2-3
terminateEnvironment, 2-3
terminateStatement method, 2-8
Timestamp methods, 8-221

N

navigational access
overview, 3-13
NCLOBs
datatype, 5-2
external datatype, 4-13
NEEDS_STREAM_DATA status, 2-17, 2-19
nonreferenceable objects, 3-3
nonstreamed reads
LOBs, 5-10
nonstreamed writes
LOBs, 5-12
nullness, 3-21
NUMBER
external datatype, 4-13
Number class, 8-104
methods, 8-107

O

object cache, 3-10, 3-11
flushing, 3-11
object mode, 3-9
object programming
overview, 3-2
using OCCI, 3-1
object references
using, 3-22
*see also*REF
Object Type Translator utility
see OTT utility
object types, 1-9
objects
access using SQL, 3-12
attributes, 1-9
dirty, 3-15
flushing, 3-15
freeing, 3-22
in OCCI, 3-2
inserting, 3-12
manipulating attributes, 3-15
marking, 3-15
methods, 1-9
modifying, 3-12
object types, 1-9
pinned, 3-14
pinning, 3-10, 3-14
recording database changes, 3-15
with LOB attributes, 5-16
OCCI
advantages, 1-2
benefits, 1-2
functionality, 1-4
object mode, 3-9
overview, 1-2
special SQL terms, 1-8
OCCI classes
Bfile class, 8-5
Blob class, 8-13
Bytes class, 8-24
Clob class, 8-27
Connection class, 8-40
ConnectionPool class, 8-45
Data class, 8-51
Environment class, 8-64
IntervalDS class, 8-70
IntervalYM class, 8-83
Map class, 8-95
MetaData class, 8-97
Number class, 8-104
PObject class, 8-130
Ref class, 8-137
RefAny class, 8-144
ResultSet class, 8-147
SQLException class, 8-169
Statement class, 8-171
Stream class, 8-215

summary, 8-2
Timestamp class, 8-219
OCCI environment
connection pool, 2-3
creating, 2-2
openning a connection, 2-3
scope, 2-2
terminating, 2-2
OCCI program
example of, 3-26
OCCI program development, 3-7
operational flow, 3-8
program structure, 3-7
OCCI types
data conversion, 4-2
optimizing performance, 2-15, 2-27
setting prefetch count, 2-15
OTT parameter TRANSITIVE, 7-96
OTT parameters
CASE, 7-91
CODE, 7-92
CONFIG, 7-92
ERRTYPE, 7-93
HFILE, 7-93
INITFILE, 7-94
INITFUNC, 7-94
INTYPE, 7-94
OUTTYPE, 7-95
SCHEMA_NAMES, 7-96
USERID, 7-97
where they appear, 7-97
OTT utility
command line, 7-12
command line syntax, 7-88
creating types in the database, 7-10
default name mapping, 7-105
description, 1-11
parameters, 7-90 to 7-96
restriction, 7-107
using, 7-2
out bind variables, 1-7
OUTTYPE OTT parameter, 7-95

P

parameterized statements, 2-9
performance
optimizing, 2-27
array fetch using next method, 2-29
executeArrayUpdate method, 2-28
setDataBuffer method, 2-27
persistent objects, 3-2, 3-3
standalone objects, 3-3
types
embedded objects, 3-3
nonreferenceable objects, 3-3
referenceable objects, 3-3
standalone objects, 3-3
with LOB attributes, 5-16
pinning objects, 3-10, 3-14
PL/SQL
out bind variables, 1-7
overview, 1-7
PObject class, 8-130
methods, 8-130
prefetch count
set, 2-15
prefetch limit, 3-16
PREPARED status, 2-17
proxy connections, 2-5
using createProxyConnection method, 2-5

Q

queries, 1-6
how to specify, 2-15

R

RAW
external datatype, 4-16
REF
external datatype, 4-17
retrieving a reference to an object, 3-13
Ref class, 8-137
methods, 8-138
RefAny class, 8-144
methods, 8-144, 8-148
reference semantics

- external LOBs, 5-3
referenceable objects, 3-3
relational programming
- using OCCI, 2-1
RESULT_SET_AVAILABLE status, 2-17, 2-18
ResultSet class, 2-14, 8-147
root object, 3-16
ROWID
- external datatype, 4-17

S

SCHEMA_NAMES OTT parameter, 7-96
- usage, 7-103
shared server environments
- application-provided serialization, 2-26
- automatic serialization, 2-25
- concurrency, 2-26
- thread safety, 2-23, 2-24
  - implementing, 2-24
- using, 2-23
SQL statements
- control statements, 1-6
- DDL statements, 1-5
- DML statements, 1-6
- processing of, 1-5
- queries, 1-6
- types
  - callable statements, 2-8, 2-10
  - parameterized statements, 2-8, 2-9
  - standard statements, 2-8, 2-9
SQLException class, 8-169
- methods, 8-169
standalone objects, 3-3
- creating, 3-3
standard statements, 2-9
Statement class, 8-171
- methods, 8-171
statement handles
- creating, 2-6
- reusing, 2-7
- terminating, 2-8
status
- NEEDS_STREAM_DATA, 2-17, 2-19
- PREPARED, 2-17
- RESULT_SET_AVAILABLE, 2-17, 2-18
- STREAM_DATA_AVAILABLE, 2-17, 2-19
- UNPREPARED, 2-17
- UPDATE_COUNT_AVAILABLE, 2-17, 2-18
Stream class, 8-215
- methods, 8-215
STREAM_DATA_AVAILABLE status, 2-17, 2-19
streamed reads, 2-12
- LOBs, 5-13
streamed writes, 2-12
- LOBs, 5-14
STRING
- external datatype, 4-17
substitutability, 3-24

T

thread safety, 2-23, 2-24
- implementing, 2-24
TIMESTAMP
- external datatype, 4-17
Timestamp class
- methods, 8-221
TIMESTAMP WITH LOCAL TIME ZONE
- external datatype, 4-18
TIMESTAMP WITH TIME ZONE
- external datatype, 4-18
transient objects, 3-2, 3-4
- creating, 3-4
- with LOB attributes, 5-17
TRANSITIVE OTT parameter, 7-23, 7-96
type inheritance, 3-22, 3-25

U

UNPREPARED status, 2-17
UNSIGNED INT
- external datatype, 4-18
UPDATE_COUNT_AVAILABLE status, 2-17, 2-18
USERID OTT parameter, 7-97

V

values
- in context of this document, 3-5

in object applications, 3-5
VARCHAR
external datatype, 4-19
VARCHAR2
external datatype, 4-19
VARNUM
external datatype, 4-19
VARRAW
external datatype, 4-13, 4-19