

Oracle Call Interface™

Programmer's Guide, Volumes 1 & 2

Release 8.0

December 1997

Part No. A58234-01

Oracle Call Interface Programmer's Guide

Part No. A58234-01

Release 8.0

Copyright © 1997, Oracle Corporation. All rights reserved.

Primary Author: Phil Locke

Contributors: John Bellemore, John Boonleungtomnu, Sashi Chandrasekaran, Debashish Chatterjee, Ernest Chen, Calvin Cheng, Luxi Chidambaran, Diana Foch-Lorentz, Sreenivas Gollapudi, Brajesh Goyal, Radhakrishna Hari, Don Herkimer, Amit Jasuja, Sanjay Kaluskar, Kai Korot, Susan Kotsovolos, Srinath Krishnaswamy, Ramkumar Krishnan, Sanjeev Kumar, Thomas Kurian, Paul Lane, Shoaib Lari, Chon Lei, Nancy Liu, Valarie Moore, Tin Nguyen, Denise Oertel, Rosanne Park, Jacqui Pons, Den Raphaely, Anindo Roy, Tim Smith, Ekrem Soylemez, Ashwini Surpur, Sudheer Thakur, Alan Thiessen, Peter Vasterd, Randall Whitman, Joyo Wijaya, Allen Zhao

This book is dedicated to the memory of Denise Elizabeth Oertel.

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 property law. Reverse engineering of the Programs is prohibited.

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

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. No part of this document 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 this Program is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Programs delivered subject to the DOD FAR Supplement are 'commercial computer software' and use, duplication and disclosure of the Programs 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 52.227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle, SQL*Forms, SQL*Net, and SQL*Plus are registered trademarks of Oracle Corporation, Redwood Shores, California.

Oracle Call Interface, Oracle7, Oracle7 Server, Oracle8, Oracle Forms, PL/SQL, Pro*C, Pro*C/C++, Pro*COBOL, Net8, and Trusted Oracle are trademarks of Oracle Corporation, Redwood Shores, California.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Contents

Send Us Your Comments	xxvii
Preface.....	xxix
Purpose of this Guide	xxx
Audience.....	xxx
Feature Coverage and Availability	xxx
How to Use this Guide.....	xxxi
How this Guide Is Organized	xxxi
Conventions Used in this Guide	xxxii
Your Comments Are Welcome.....	xxxv
Part I Basic OCI Concepts	
1 Introduction and New Features	
The Oracle Call Interface	1-2
SQL Statements.....	1-4
Data Definition Language	1-4
Control Statements	1-5

Data Manipulation Language	1-5
Queries	1-5
PL/SQL	1-6
Embedded SQL	1-7
Special OCI/SQL Terms	1-8
Object Support in the OCI	1-8
Parts of the OCI	1-10
Release 8.0 New Features	1-10
Obsolescent and Obsolete OCI Calls	1-11
Compiling and Linking	1-11

2 OCI Programming Basics

Overview	2-2
OCI Program Structure	2-3
OCI Data Structures	2-5
Handles	2-6
Allocating and Freeing Handles	2-7
Environment Handle	2-8
Error Handle	2-8
Service Context and Associated Handles	2-8
Statement Handle, Bind Handle, and Define Handle	2-10
Describe Handle	2-10
Complex Object Retrieval Handle	2-10
Security Handle	2-10
Handle Attributes	2-11
User Memory Allocation	2-12
Descriptors and Locators	2-12
Snapshot Descriptor	2-13
LOB/FILE Datatype Locator	2-13
Parameter Descriptor	2-14
ROWID Descriptor	2-15
Complex Object Descriptor	2-15
Advanced Queueing Descriptors	2-15
User Memory Allocation	2-15
OCI Programming Steps	2-16

Initialization, Connection, and Session Creation.....	2-17
Initialize an OCI Process.....	2-17
Allocate Handles and Descriptors	2-18
Application Initialization, Connection, and Session Creation.....	2-18
Understanding Multiple Connections and Handles	2-21
A Connection Example	2-21
Processing SQL Statements	2-23
Commit or Rollback	2-23
Terminating the Application	2-24
Error Handling	2-25
Functions Returning Other Values	2-27
Additional Coding Guidelines	2-27
Parameter Types.....	2-27
Nulls	2-28
Indicator Variables	2-29
Canceling Calls.....	2-31
Positioned Updates and Deletes.....	2-31
Application Linking	2-32
Using PL/SQL in an OCI Program.....	2-32

3 Datatypes

Oracle Datatypes.....	3-2
Internal Datatype Codes.....	3-4
External Datatype Codes.....	3-4
Internal Datatypes	3-5
LONG, RAW, LONG RAW, VARCHAR2.....	3-5
Character Strings and Byte Arrays.....	3-6
External Datatypes.....	3-7
VARCHAR2	3-9
NUMBER	3-10
INTEGER	3-11
FLOAT.....	3-11
STRING	3-12
VARNUM	3-13
LONG	3-13

VARCHAR.....	3-13
ROWID	3-14
DATE	3-14
RAW.....	3-15
VARRAW	3-15
LONG RAW.....	3-15
UNSIGNED.....	3-16
LONG VARCHAR.....	3-16
LONG VARRAW	3-16
CHAR	3-16
CHARZ.....	3-17
MLSLABEL	3-18
New OCI 8.0 External Datatypes.....	3-18
NAMED DATA TYPE.....	3-18
REF	3-19
LOB	3-19
New C Datatype Mappings.....	3-21
Data Conversions.....	3-22
Typecodes	3-24
Relationship Between SQLT and OCI_TYPECODE Values.....	3-25
Definitions in oratypes.h	3-27

4 SQL Statement Processing

Overview.....	4-2
Processing SQL Statements.....	4-2
Preparing Statements	4-4
Using Prepared Statements on Multiple Servers	4-5
Binding.....	4-5
Executing Statements	4-6
Execution Snapshots.....	4-7
Execution Modes.....	4-7
Describing Select-List Items.....	4-8
Implicit Describe	4-9
Explicit Describe of Queries	4-10
Defining	4-11

Fetching Results	4-12
Fetching LOB Data	4-12
Setting Prefetch Count	4-12

5 Binding and Defining

Binding	5-2
Named Binds and Positional Binds	5-4
OCI Array Interface	5-4
Binding Placeholders in PL/SQL	5-5
Steps Used in Binding	5-6
PL/SQL Example	5-7
Advanced Binds	5-9
Advanced Bind Operations	5-9
Static Array Binds	5-10
Named Data Type Binds	5-10
Binding REFs	5-10
Binding LOBs	5-10
Binding in OCI_DATA_AT_EXEC Mode	5-11
Binding Ref Cursor Variables	5-12
Summary of Bind Information	5-12
Defining	5-13
Steps Used in Defining	5-14
Advanced Defines	5-15
Advanced Define Operations	5-16
Defining Named Data Type Output Variables	5-16
Defining REF Output Variables	5-16
Defining LOB Output Variables	5-16
Defining PL/SQL Output Variables	5-17
Defining For a Piecewise Fetch	5-17
Defining Arrays of Structures	5-17
Arrays of Structures	5-17
Skip Parameters	5-18
OCI Calls Used with Arrays of Structures	5-20
Arrays of Structures and Indicator Variables	5-20
DML with RETURNING Clause	5-21

Using DML with RETURNING Clause.....	5-21
Binding RETURNING...INTO variables.....	5-22
Error Handling.....	5-23
DML with RETURNING REF...INTO clause.....	5-23
Additional Notes About Callbacks.....	5-25
NCHAR and Character Conversion Issues	5-25
NCHAR Issues.....	5-25
OCI_ATTR_MAXDATA_SIZE Attribute.....	5-26
Character Count Attribute.....	5-26
PL/SQL REF CURSORS and Nested Tables	5-27

6 Describing Schema Metadata

Overview	6-2
Using OCIDescribeAny()	6-2
Restrictions.....	6-3
Note on Datatype Codes.....	6-4
Note on Describing Types	6-4
Note on OCI_ATTR_LIST_ARGUMENTS.....	6-5
Parameter Attributes.....	6-5
Table/View Attributes.....	6-7
Procedure/Function Attributes.....	6-7
Package Attributes.....	6-8
Type Attributes	6-9
Type Attribute Attributes.....	6-10
Type Method Attributes	6-12
Collection Attributes	6-13
Synonym Attributes	6-14
Sequence Attributes.....	6-15
Column Attributes.....	6-15
Argument/Result Attributes	6-17
List Attributes.....	6-19
Examples	6-20
Retrieving column data types for a table	6-20
Describing the stored procedure	6-21

Retrieving attributes of an object type.....	6-23
Retrieving the collection element's data type of a named collection type	6-25

7 OCI Programming Advanced Topics

Overview	7-2
Transactions	7-3
Levels of Transactional Complexity	7-3
Transaction Examples	7-9
Related Initialization Parameters	7-10
User Authentication and Password Management	7-11
Authentication	7-11
Password Management	7-12
Thread Safety	7-13
Advantages of OCI Thread Safety	7-13
Thread Safety and Three-Tier Architectures	7-13
Basic Concepts of Multi-threaded Development	7-14
Implementing Thread Safety with OCI 8.0	7-14
Run Time Data Allocation and Piecewise Operations	7-16
Providing INSERT or UPDATE Data at Run Time	7-18
Piecewise Operations With PL/SQL	7-20
Providing FETCH Information at Run Time	7-20
Additional Information About Piecewise Operations with No Callbacks.....	7-23
LOB and FILE Operations	7-24
LOBs and LOB Locators	7-24
FILES	7-26
Creating and Modifying Internal LOBs	7-26
Associating a FILE in a Table with an OS File.....	7-27
Writing to a LOB Attribute of an Object	7-27
Transient Objects with LOB Attributes	7-27
LOB Buffering	7-28
LOB/FILE Functions.....	7-28
Server Roundtrips for LOB Functions	7-31
LOB Read/Write Callbacks.....	7-31
The Callback Interface for Streaming	7-31
Reading LOBs using Callbacks.....	7-32

Writing LOBs using Callbacks	7-34
OCI Callbacks From External Procedures	7-35
Application Failover Callbacks	7-36
Failover Callback Overview	7-36
Failover Callback Structure and Parameters	7-36
Failover Callback Registration	7-37
Failover Callback Example	7-38
OCI and Advanced Queueing	7-40
OCI Advanced Queueing Functions	7-40
OCI Advanced Queueing Descriptors	7-40
Advanced Queueing in OCI vs. PL/SQL	7-41
Writing Oracle Security Services Applications	7-43

Part II OCI Object Concepts

8 OCI Object-Relational Programming

Chapter Overview	8-2
OCI Object Overview	8-3
Working with Objects in the OCI	8-4
Basic Object Program Structure	8-4
Persistent Objects, Transient Objects, and Values	8-5
Developing an OCI Object Application	8-8
Representing Objects in C Applications	8-8
Initializing Environment and Object Cache	8-10
Making Database Connections	8-10
Retrieving an Object Reference from the Server	8-11
Pinning an Object	8-12
Manipulating Object Attributes	8-13
Marking Objects and Flushing Changes	8-14
Fetching Embedded Objects	8-15
Object Meta-Attributes	8-17
Complex Object Retrieval	8-21
COR Prefetching	8-25
Pin Count and Unpinning	8-28
Nullness	8-28

Creating, Freeing, and Copying Objects	8-31
Object Reference and Type Reference	8-32
Error Handling in Object Applications	8-32

9 Object-Relational Datatypes

Overview	9-2
Mapping Oracle8 Datatypes to C	9-3
OCI Type Mapping Methodology.....	9-5
Manipulating C Datatypes With OCI	9-5
Precision of Oracle Number Operations	9-7
Date (OCIDate)	9-7
Date Conversion Functions.....	9-7
Date Assignment and Retrieval Functions	9-8
Date Arithmetic and Comparison Functions.....	9-8
Date Information Accessor Functions	9-8
Date Validity Checking Functions	9-8
Date Example	9-9
Number (OCINumber)	9-10
Number Arithmetic Functions.....	9-11
Number Conversion Functions	9-11
Exponential and Logarithmic Functions.....	9-12
Trigonometric Functions	9-12
Number Assignment and Comparison Functions.....	9-12
Number Example.....	9-13
Fixed or Variable-Length String (OCIStrng)	9-15
String Functions.....	9-15
String Example.....	9-15
Raw (OCIRaw)	9-16
Raw Functions.....	9-16
Raw Example.....	9-17
Collections (OCITable, OCIArray, OCIColl, OCIIter)	9-17
Generic Collection Functions.....	9-17
Collection Data Manipulation Functions.....	9-18
Collection Scanning Functions	9-19
Varray/Collection Iterator Example.....	9-19

Nested Table Manipulation Functions	9-20
REF (OCIRef)	9-22
REF Manipulation Functions	9-22
REF Example	9-22
Object Type Information Storage and Access	9-23
Descriptor Objects.....	9-23

10 Binding and Defining in Object Applications

Binding	10-2
Named Data Type Binds.....	10-2
Binding REFs	10-3
Additional Information for Named Data Type and REF Binds.....	10-3
Defining	10-4
Defining Named Data Type Output Variables.....	10-4
Defining REF Output Variables.....	10-4
Additional Information for Object and REF Defines, and PL/SQL OUT Binds	10-5
Binding And Defining Oracle8 C Datatypes	10-6
Bind and Define Examples	10-8
3 Salary Update Examples.....	10-10
SQLT_NTY Bind/Define Example	10-13
Bind Example	10-13
Define Example	10-14

11 Object Cache and Object Navigation

Chapter Overview	11-2
The Object Cache and Memory Management	11-2
Cache Consistency and Coherency	11-4
Object Cache Parameters	11-5
Object Cache Operations	11-6
Operations for Loading and Removing Object Copies	11-6
Operations for Making Changes to Object Copies	11-9
Operations for Synchronizing Object Copies with Server.....	11-10
Other Operations	11-12
Commit and Rollback in Object Applications	11-13
Object Duration	11-13

Memory Layout of an Instance.....	11-15
Object Navigation	11-16
Simple Object Navigation.....	11-16
OCI Navigational Functions	11-18
Pin/Unpin/Free Functions	11-18
Flush and Refresh Functions.....	11-19
Mark and Unmark Functions.....	11-19
Object Meta-Attribute Accessor Functions	11-19
Other Functions	11-20

12 Using the Object Type Translator

OTT Overview	12-2
Using the Object Type Translator	12-2
Creating Types in the Database.....	12-4
Invoking the OTT	12-5
The OTT Command Line	12-6
OTT	12-6
userid.....	12-6
intype.....	12-6
outtype	12-6
code.....	12-7
hfile	12-7
initfile.....	12-7
initfunc	12-8
The Intype File	12-8
OTT Datatype Mappings	12-9
Null Indicator Structs.....	12-15
The Outtype File	12-16
Using the OTT with OCI Applications	12-18
Accessing and Manipulating Objects with OCI.....	12-19
Calling the Initialization Function	12-20
Tasks of the Initialization Function.....	12-22
OTT Reference	12-22
OTT Command Line Syntax	12-23
OTT Parameters	12-24

Where OTT Parameters Can Appear	12-28
Structure of the Intype File	12-29
Nested #include File Generation	12-31
SCHEMA_NAMES Usage	12-33
Default Name Mapping	12-35
Restrictions.....	12-37

Part III OCI Reference

13 OCI Relational Functions

Introduction	13-2
OCI Quick Reference	13-3
Calling OCI Functions	13-6
Server Roundtrips for LOB Functions	13-6
The OCI Relational Functions	13-7
OCIAQDeq()	13-8
OCIAQEnq()	13-11
OCIAttrGet()	13-23
OCIAttrSet()	13-25
OCIBindArrayOfStruct()	13-28
OCIBindByName()	13-30
OCIBindByPos()	13-34
OCIBindDynamic().....	13-38
OCIBindObject()	13-42
OCIBreak()	13-45
OCIDefineArrayOfStruct().....	13-46
OCIDefineByPos().....	13-48
OCIDefineDynamic()	13-52
OCIDefineObject().....	13-55
OCIDescribeAny()	13-57
OCIDescriptorAlloc()	13-60
OCIDescriptorFree()	13-62
OCIEnvInit()	13-63
OCIErrorGet()	13-65
OCIHandleAlloc().....	13-68

OCIHandleFree()	13-70
OCIInitialize()	13-72
OCILdaToSvcCtx()	13-75
OCILobAppend()	13-76
OCILobAssign()	13-78
OCILobCharSetForm()	13-80
OCILobCharSetId()	13-81
OCILobCopy()	13-82
OCILobDisableBuffering()	13-84
OCILobEnableBuffering()	13-85
OCILobErase()	13-86
OCILobFileClose()	13-88
OCILobFileCloseAll()	13-89
OCILobFileExists()	13-90
OCILobFileGetName()	13-91
OCILobFileIsOpen()	13-93
OCILobFileOpen()	13-95
OCILobFileSetName()	13-96
OCILobFlushBuffer()	13-98
OCILobGetLength()	13-100
OCILobIsEqual()	13-102
OCILobLoadFromFile()	13-103
OCILobLocatorIsInit()	13-105
OCILobRead()	13-107
OCILobTrim()	13-111
OCILobWrite()	13-112
OCILogoff()	13-116
OCILogon()	13-117
OCIParmGet()	13-119
OCIParmSet()	13-121
OCIPasswordChange()	13-123
OCIServerAttach()	13-125
OCIServerDetach()	13-127
OCIServerVersion()	13-128
OCISessionBegin()	13-129

OCISessionEnd()	13-132
OCIStmtExecute()	13-134
OCIStmtFetch()	13-137
OCIStmtGetBindInfo()	13-139
OCIStmtGetPieceInfo()	13-141
OCIStmtPrepare()	13-143
OCIStmtSetPieceInfo()	13-145
OCISvcCtxToLda()	13-147
OCITransCommit()	13-149
OCITransDetach()	13-152
OCITransForget()	13-154
OCITransPrepare()	13-155
OCITransRollback()	13-156
OCITransStart()	13-157

14 OCI Navigation and Type Functions

Introduction	14-2
Object Types and Lifetimes	14-2
Terminology	14-4
Navigational Function Return Values	14-4
Navigational Function Error Codes	14-5
Server Roundtrips for Cache and Object Functions	14-7
OCI Navigational Functions Quick Reference	14-8
The OCI Navigational Functions	14-10
OCICacheFlush()	14-11
OCICacheFree()	14-13
OCICacheRefresh()	14-14
OCICacheUnmark()	14-16
OCICacheUnpin()	14-17
OCIObjectArrayPin()	14-18
OCIObjectCopy()	14-20
OCIObjectExists()	14-22
OCIObjectFlush()	14-23
OCIObjectFree()	14-24
OCIObjectGetAttr()	14-26

OCIObjectGetInd()	14-28
OCIObjectGetObjectRef()	14-29
OCIObjectGetProperty()	14-30
OCIObjectGetTypeRef()	14-34
OCIObjectIsDirty()	14-35
OCIObjectIsLocked()	14-36
OCIObjectLock()	14-37
OCIObjectMarkDelete()	14-38
OCIObjectMarkDeleteByRef()	14-40
OCIObjectMarkUpdate()	14-41
OCIObjectNew()	14-43
OCIObjectPin()	14-46
OCIObjectPinCountReset()	14-49
OCIObjectPinTable()	14-51
OCIObjectRefresh()	14-53
OCIObjectSetAttr()	14-55
OCIObjectUnmark()	14-57
OCIObjectUnmarkByRef()	14-58
OCIObjectUnpin()	14-59
OCITypeArrayByName()	14-61
OCITypeArrayByRef()	14-64
OCITypeByName()	14-66
OCITypeByRef()	14-69

15 OCI Datatype Mapping and Manipulation Functions

Introduction	15-2
Datatype Mapping and Manipulation Function Return Values	15-2
Functions Returning Other Values	15-3
Server Roundtrips for Datatype Mapping and Manipulation Functions	15-3
OCI Datatype Mapping Functions Quick Reference	15-4
OCICollAppend()	15-9
OCICollAssign()	15-11
OCICollAssignElem()	15-13
OCICollGetElem()	15-15
OCICollMax()	15-18

OCICollSize()	15-19
OCICollTrim()	15-21
OCIDateAddDays()	15-22
OCIDateAddMonths()	15-23
OCIDateAssign()	15-24
OCIDateCheck()	15-25
OCIDateCompare()	15-27
OCIDateDaysBetween()	15-28
OCIDateFromText()	15-29
OCIDateGetDate()	15-31
OCIDateGetTime()	15-32
OCIDateLastDay()	15-33
OCIDateNextDay()	15-34
OCIDateSetDate()	15-36
OCIDateSetTime()	15-37
OCIDateSysDate()	15-38
OCIDateToText()	15-39
OCIDateZoneToZone()	15-41
OCIIterCreate()	15-43
OCIIterDelete()	15-45
OCIIterGetCurrent()	15-46
OCIIterInit()	15-47
OCIIterNext()	15-48
OCIIterPrev()	15-50
OCINumberAbs()	15-52
OCINumberAdd()	15-53
OCINumberArcCos()	15-54
OCINumberArcSin()	15-55
OCINumberArcTan()	15-56
OCINumberArcTan2()	15-57
OCINumberAssign()	15-58
OCINumberCeil()	15-59
OCINumberCmp()	15-60
OCINumberCos()	15-61
OCINumberDiv()	15-62

OCINumberExp()	15-63
OCINumberFloor()	15-64
OCINumberFromInt()	15-65
OCINumberFromReal()	15-67
OCINumberFromText()	15-68
OCINumberHypCos()	15-70
OCINumberHypSin()	15-71
OCINumberHypTan()	15-72
OCINumberIntPower()	15-73
OCINumberIsZero()	15-74
OCINumberLn()	15-75
OCINumberLog()	15-76
OCINumberMod()	15-77
OCINumberMul()	15-78
OCINumberNeg()	15-79
OCINumberPower()	15-80
OCINumberRound()	15-81
OCINumberSetZero()	15-82
OCINumberSign()	15-83
OCINumberSin()	15-84
OCINumberSqrt()	15-85
OCINumberSub()	15-86
OCINumberTan()	15-87
OCINumberToInt()	15-88
OCINumberToReal()	15-90
OCINumberToText()	15-91
OCINumberTrunc()	15-93
OCIRawAllocSize()	15-94
OCIRawAssignBytes()	15-95
OCIRawAssignRaw()	15-96
OCIRawPtr()	15-97
OCIRawResize()	15-98
OCIRawSize()	15-99
OCIRefAssign()	15-100
OCIRefClear()	15-101

OCIRefFromHex()	15-102
OCIRefHexSize()	15-104
OCIRefIsEqual()	15-105
OCIRefIsNull()	15-106
OCIRefToHex()	15-107
OCIStrAllocSize()	15-109
OCIStrAssign()	15-110
OCIStrAssignText()	15-111
OCIStrPtr()	15-112
OCIStrResize()	15-113
OCIStrSize()	15-114
OCITableDelete()	15-115
OCITableExists()	15-116
OCITableFirst()	15-117
OCITableLast()	15-118
OCITableNext()	15-119
OCITablePrev()	15-121
OCITableSize()	15-123

16 OCI External Procedure Functions

Introduction	16-2
The OCI External Procedure Functions	16-3
OCIExtProcAllocCallMemory()	16-4
OCIExtProcRaiseExcp()	16-5
OCIExtProcRaiseExcpWithMsg()	16-6
OCIExtProcGetEnv()	16-8

Part IV Appendices

A Upgrading Release 7.x OCI Applications to Release 8.0

Compatibility and Upgrade Overview	A-2
Obsolescent OCI Routines	A-2
Obsolete OCI Routines	A-4
Compatibility	A-4

Upgrading	A-6
Application Linking Issues	A-7
Non-deferred linking	A-7
Single-task linking	A-9

B Handle and Descriptor Attributes

Conventions	B-2
Environment Handle Attributes	B-3
OCI_ATTR_CACHE_MAX_SIZE	B-3
OCI_ATTR_CACHE_OPT_SIZE	B-3
OCI_ATTR_OBJECT	B-3
OCI_ATTR_FNCODE	B-4
OCI_ATTR_PINOPTION	B-4
OCI_ATTR_ALLOC_DURATION	B-4
OCI_ATTR_PIN_DURATION	B-6
Service Context Handle Attributes	B-7
OCI_ATTR_SQLCODE	B-7
OCI_ATTR_ENV	B-7
OCI_ATTR_SERVER.....	B-7
OCI_ATTR_SESSION.....	B-9
OCI_ATTR_TRANS	B-9
OCI_ATTR_IN_V8_MODE	B-9
Server Handle Attributes	B-11
OCI_ATTR_ENV	B-11
OCI_ATTR_FNCODE	B-11
OCI_ATTR_EXTERNAL_NAME.....	B-11
OCI_ATTR_INTERNAL_NAME	B-12
OCI_ATTR_IN_V8_MODE	B-12
OCI_ATTR_FOCBK.....	B-12
User Session Handle Attributes	B-13
OCI_ATTR_USERNAME	B-13
OCI_ATTR_PASSWORD	B-13
Transaction Handle Attributes	B-14
OCI_ATTR_TRANS_NAME.....	B-14
OCI_ATTR_XID	B-14

Statement Handle Attributes	B-15
OCI_ATTR_FNCODE	B-15
OCI_ATTR_ROW_COUNT.....	B-15
OCI_ATTR_SQLFNCODE.....	B-15
OCI_ATTR_ENV	B-16
OCI_ATTR_STMT_TYPE.....	B-16
OCI_ATTR_ROWID	B-17
OCI_ATTR_PARAM_COUNT	B-17
OCI_ATTR_PREFETCH_ROWS.....	B-18
OCI_ATTR_PREFETCH_MEMORY	B-18
Bind Handle Attributes	B-19
OCI_ATTR_FNCODE	B-19
OCI_ATTR_CHAR_COUNT	B-19
OCI_ATTR_CHARSET_ID	B-19
OCI_ATTR_CHARSET_FORM.....	B-20
OCI_ATTR_MAXDATA_SIZE	B-20
OCI_ATTR_PDSCL	B-20
OCI_ATTR_PDFMT	B-21
OCI_ATTR_ROWS_RETURNED	B-21
Define Handle Attributes	B-22
OCI_ATTR_FNCODE	B-22
OCI_ATTR_CHAR_COUNT	B-22
OCI_ATTR_CHARSET_ID	B-22
OCI_ATTR_CHARSET_FORM.....	B-23
OCI_ATTR_PDSCL	B-23
OCI_ATTR_PDFMT	B-23
Describe Handle Attributes	B-24
OCI_ATTR_PARAM_COUNT	B-24
Parameter Descriptor Attributes	B-24
LOB Locator Attributes	B-25
OCI_ATTR_LOBEMPTY	B-25
Complex Object Attributes	B-26
Complex Object Retrieval Handle Attributes.....	B-26
OCI_ATTR_COMPLEXOBJECT_LEVEL	B-26
OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE	B-26

Complex Object Retrieval Descriptor Attributes	B-27
OCI_ATTR_COMPLEXOBJECTCOMP_TYPE	B-27
OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL.....	B-27
Advanced Queueing Descriptor Attributes.....	B-28
OCIAQEnqOptions Descriptor Attributes.....	B-28
OCI_ATTR_RELATIVE_MSGID.....	B-28
OCI_ATTR_SEQUENCE_DEVIATION.....	B-28
OCI_ATTR_VISIBILITY	B-29
OCIAQDeqOptions Descriptor Attributes	B-29
OCI_ATTR_CONSUMER_NAME.....	B-29
OCI_ATTR_CORRELATION	B-30
OCI_ATTR_DEQ_MODE.....	B-30
OCI_ATTR_DEQ_MSGID	B-31
OCI_ATTR_NAVIGATION.....	B-31
OCI_ATTR_VISIBILITY	B-32
OCI_ATTR_WAIT	B-32
OCIAQMsgProperties Descriptor Attributes.....	B-33
OCI_ATTR_ATTEMPTS.....	B-33
OCI_ATTR_CORRELATION	B-33
OCI_ATTR_DELAY	B-33
OCI_ATTR_ENQ_TIME.....	B-34
OCI_ATTR_EXCEPTION_QUEUE.....	B-34
OCI_ATTR_EXPIRATION.....	B-35
OCI_ATTR_MSG_STATE	B-35
OCI_ATTR_PRIORITY	B-36
OCI_ATTR_RECIPIENT_LIST	B-36
OCIAQAgent Descriptor Attributes.....	B-37
OCI_ATTR_AGENT_ADDRESS.....	B-37
OCI_ATTR_AGENT_NAME.....	B-37
OCI_ATTR_AGENT_PROTOCOL	B-37

C Oracle Reserved Words, Keywords and Namespaces

Oracle Reserved Words and Keywords	C-2
PL/SQL Reserved Words.....	C-10
Oracle Reserved Namespaces	C-11

D Code Examples

Example 1, SQL Processing	D-2
Example 2, Object Retrieval	D-11
Example 3, DML with RETURNING Clause.....	D-25
Example 4, Describing an Object	D-55
Example 5, CLOB/BLOB Operations.....	D-76
Example 6, LOB Buffering.....	D-96
Example 7, REF Pinning and Navigation	D-118

E OCI Function Server Roundtrips

Overview.....	E-2
LOB Function Roundtrips	E-2
Object and Cache Function Roundtrips	E-4
Describe Operation Roundtrips.....	E-5
Datatype Mapping and Manipulation Function Roundtrips.....	E-6
Other Local Functions	E-7

F Oracle8 OCI New Features

Introduction	F-2
Oracle8 OCI Enhancements	F-2
Encapsulated/Opaque Interfaces.....	F-2
Simplified User Authentication and Password Management.....	F-3
Extensions to Improve Application Performance and Scalability	F-4
Consistent Interface for Transaction Management.....	F-4
Oracle8 OCI Object Support.....	F-4
Runtime Environment for Objects.....	F-6
Type Management, Mapping and Manipulation Functions	F-6
Object Type Translator	F-7
OCI Support for Oracle Advanced Queueing	F-7

Benefits of the OCI's New Features..... F-8
 Comprehensive Support for Oracle8 Objects..... F-8
 Improved Application Performance..... F-8
 Greater Scalability F-9
 Simplified Migration of Existing Applications F-9
 Enhanced Application Extensibility F-10

Index

Send Us Your Comments

Oracle Call Interface Programmer's Guide, Release 8.0

Part No. A58234-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. 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 about this manual?

If you find any errors or have any other suggestions for improvement, please indicate 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)506-7228
- postal service:
Oracle Corporation
Oracle Server Documentation Manager
500 Oracle Parkway
Redwood Shores, CA 94065
USA

If you would like a reply, please give your name, address, and telephone number below.

Preface

The Oracle Call Interface (OCI) is an application programming interface (API) that allows applications written in C to interact with one or more Oracle servers. The OCI gives your programs the capability to perform the full range of database operations that are possible with an Oracle8 server, including SQL statement processing and object manipulation.

The Preface includes the following sections:

- Purpose of this Guide
- Audience
- Feature Coverage and Availability
- How to Use this Guide
- How this Guide Is Organized
- Conventions Used in this Guide
- Your Comments Are Welcome

Purpose of this Guide

This guide gives you a sound basis for developing applications using the OCI. The guide is divided into two volumes.

Volume I contains information about the following topics:

- the structure of an OCI application
- conversion of data between the server and variables in your OCI application
- object functions that provide navigational access to objects, type management, and data type mapping and manipulation

Volume II contains the following information:

- a description of every OCI function call, along with syntax information and parameter descriptions
- a listing of all OCI handle attributes
- a listing of Oracle reserved words, keywords, and reserved namespaces
- sample programs that illustrate the features of the OCI
- upgrading applications from earlier releases of the OCI to release 8.0
- server roundtrips required for most OCI calls

Audience

The *Oracle Call Interface Programmer's Guide* is intended for programmers developing new applications or converting existing applications to run in the Oracle environment. This comprehensive treatment of the OCI will also be valuable to systems analysts, project managers, and others interested in the development of database applications.

This guide assumes that you have a working knowledge of application programming using C. Readers should also be familiar with the use of Structured Query Language (SQL) to access information in relational database systems. In addition, some sections of this guide also assume a knowledge of the basic concepts of object-oriented programming.

For information about SQL, refer to the *Oracle8 SQL Reference* and the *Oracle8 Administrator's Guide*. For information about basic Oracle concepts, see *Oracle8 Concepts*. For information about the Oracle Precompilers, which enable you to embed SQL commands in a third-generation language (3GL) application, refer to the *Pro*C/C++ Precompiler Programmer's Guide* and the *Pro*COBOL Precompiler Programmer's Guide*.

Feature Coverage and Availability

The *Oracle Call Interface Programmer's Guide* contains information that describes the features and functionality of the Oracle8 and the Oracle8 Enterprise Edition products. Oracle8 and Oracle8 Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use object functionality, you must have the Enterprise Edition and the Objects Option.

For information about the differences between Oracle8 and the Oracle8 Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8 and the Oracle8 Enterprise Edition*.

How to Use this Guide

The *Oracle Call Interface Programmer's Guide* provides an introduction to the features of the OCI for both new OCI programmers and those programmers who have previously worked with earlier versions of the OCI.

VOLUME I

Part 1

Part 1 of this guide (Chapters 1 through 7) provides conceptual information about how to program with the OCI to access relational data in an Oracle database. This part describes the basics of OCI programming and builds the foundation for the discussion of object-relational features in Part 2.

Part 2

Part 2 of this guide (Chapters 8 through 12) describes OCI functionality for accessing object-relational data with the OCI. The chapters in this part describe how to retrieve and manipulate objects through an Oracle8 server.

VOLUME II

Part 3

Part 3 of this book (Chapters 13 through 16) lists all function calls in the Oracle8 OCI library.

Part 4

Part 4 of this book (Appendices A through F) provides additional information about OCI programming, along with complete code examples.

Where to Begin

Because of the many enhancements to the OCI for Release 8.0, both new and experienced users should read the conceptual material in Part 1. Although most basic concepts (e.g., binding, defining, etc.) have remained the same as in Release 7.3, those concepts have a new implementation in Release 8.0.

Readers familiar with the current version of the OCI and interested in its object capabilities may want to skim Part 1 and then begin reading the chapters in Part 2.

Readers looking for reference information (e.g., OCI function syntax, handle attribute descriptions) should refer to *Volume II*.

How this Guide Is Organized

The *Oracle Call Interface Programmer's Guide* contains four parts, split between two volumes. A brief summary of what you will find in each chapter and appendix follows:

VOLUME I

PART 1: OCI RELATIONAL APPLICATIONS

Chapter 1: Introduction and New Features

This chapter introduces you to the Oracle Call Interface and describes special terms and typographical conventions that are used in describing the interfaces. This chapter also mentions features new to the current release.

Chapter 2: OCI Programming Basics

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

Chapter 3: Datatypes

Understanding how data is converted between Oracle tables and variables in your host program is essential for using the OCI interfaces. This chapter discusses Oracle internal and external datatypes, and data conversions.

Chapter 4: SQL Statement Processing

This chapter discusses the steps involved in SQL statements using the Oracle8 OCI.

Chapter 5: Binding and Defining

This chapter discusses OCI bind and define operations in detail, including a discussion of advanced bind and define operations.

Chapter 6: Describing Schema Metadata

This chapter discusses how to use the *OCIDescribeAny()* call to obtain information about schema objects and their associated elements.

Chapter 7: OCI Programming Advanced Topics

This chapter covers more sophisticated OCI programming topics, including descriptions of transaction management, LOB support, advanced binding and defining, and other functionality.

PART 2: OCI OBJECT-RELATIONAL APPLICATIONS

Chapter 8: OCI Object-Relational Programming

This chapter provides an introduction to the concepts involved when using the OCI to access objects in an Oracle8 server. The chapter includes a discussion of basic object concepts and object pinning, and the basic structure of object-relational applications.

Chapter 9: Object-Relational Datatypes

This chapter outlines the object datatypes used in OCI programming.

Chapter 10: Binding and Defining in Object Applications

This chapter discusses the C mappings of user-defined datatypes in an Oracle8 database, and the functions that manipulate such data. Binding and defining using these C mappings is also covered.

Chapter 11: Object Cache and Object Navigation

This chapter provides an introduction to the concepts involved when using the OCI to access objects in an Oracle8 server. This chapter also discusses the Object Cache, and the use of the OCI navigational calls to manipulate objects retrieved from the server.

Chapter 12: Using the Object Type Translator

This chapter discusses the use of the Object Type Translator to convert database object definitions to C structure representations for use in OCI applications.

VOLUME II**PART 3: OCI REFERENCE****Chapter 13: OCI Relational Functions**

This chapter contains a list of the OCI relational functions, including syntax, comments, parameter descriptions, and other useful information.

Chapter 14: OCI Navigation and Type Functions

This chapter contains a list of the OCI navigational functions, including syntax, comments, parameter descriptions, and other useful information.

Chapter 15: OCI Datatype Mapping and Manipulation Functions

This chapter contains a list of the OCI datatype mapping and manipulation functions, including syntax, comments, parameter descriptions, and other useful information.

Chapter 16: OCI External Procedure Functions

This chapter discusses special OCI functions used by external procedures.

PART 4: APPENDICES**Appendix A: Upgrading Release 7.x OCI Applications to Release 8.0**

This appendix discusses the issues involved in upgrading existing OCI applications to use the new Oracle8 OCI. This includes lists of those OCI calls which are now obsolescent or obsolete.

Appendix B: Handle and Descriptor Attributes

This appendix describes the attributes of OCI application handles that can be set or read using OCI calls.

Appendix C: Oracle Reserved Words, Keywords and Namespaces

This appendix lists words that have a special meaning to Oracle, and namespaces reserved by Oracle products.

Appendix D: Code Examples

This appendix includes complete OCI application code examples.

Appendix E: OCI Function Server Roundtrips

This appendix includes tables which show the number of server roundtrips required by various OCI applications.

Appendix F: Oracle8 OCI New Features

This appendix provides detailed information about features and enhancements available in the Oracle8 OCI.

Conventions Used in this Guide

The following notational and text formatting conventions are used in this guide:

[]

Square brackets indicate that the enclosed item is optional. Do not type the brackets.

{ }

Braces enclose items of which only one is required.

|

A vertical bar separates items within braces, and may also be used to indicate that multiple values are passed to a function parameter.

...

In code fragments, an ellipsis means that code not relevant to the discussion has been omitted.

font change

SQL or C code examples are shown in monospaced font.

italics

Italics are used for OCI parameters, OCI routines names, file names, and data fields.

UPPERCASE

Uppercase is used for SQL keywords, like SELECT or UPDATE.

bold

Boldface type is used to identify the names of C datatypes, like **ub4**, **sword**, or **OCINumber**.

This guide uses special text formatting to draw the reader's attention to some information. A paragraph that is indented and begins with a bold text label may have special meaning. The following paragraphs describe the different types of information that are flagged this way.

Note: The "Note" flag indicates that the reader should pay particular attention to the information to avoid a common problem or increase understanding of a concept.

7.x Upgrade Note: An item marked with "7.x Upgrade Note" typically alerts the programmer to something that is done much differently in the release 8.0 OCI than in the 7.x OCIs.

Warning: An item marked as "Warning" indicates something that an OCI programmer must be careful to do or not do in order for an application to work correctly.

See Also: Text marked "See Also" points you to another section of this guide, or to other documentation, for additional information about the topic being discussed.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important feedback we receive.

You can send comments and suggestions about this manual to the following e-mail address:

infodev@us.oracle.com

If you prefer, you can send letters or faxes containing your comments to the following address:

Oracle8 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
Fax: (650) 506-7228

Part I

Basic OCI Concepts

This part of the guide contains chapters that describe basic OCI programming concepts:

- Chapter 1, “Introduction and New Features”, provides an introduction to the OCI and discusses features that are new to release 8.0.
- Chapter 2, “OCI Programming Basics”, discusses the basic concepts of OCI programming.
- Chapter 3, “Datatypes”, describes datatypes used in OCI applications and within the Oracle8 Server.
- Chapter 4, “SQL Statement Processing”, discusses how to process SQL statements using the Oracle8 OCI.
- Chapter 5, “Binding and Defining”, discusses bind and define operations in detail.
- Chapter 6, “Describing Schema Metadata”, discusses the *OCIDescribeAny()* function.
- Chapter 7, “OCI Programming Advanced Topics”, covers some advanced topics in OCI programming.

Introduction and New Features

This chapter introduces you to the Oracle Call Interface, Release 8.0. It gives you background information that you need to develop applications using the OCI. It also introduces special terms that are used in discussing the OCI.

This chapter also discusses the changes in the OCI since release 7.3.

The following topics are covered:

- The Oracle Call Interface
- SQL Statements
- Special OCI/SQL Terms
- Object Support in the OCI
- Parts of the OCI
- Release 8.0 New Features
- Obsolescent and Obsolete OCI Calls
- Compiling and Linking

The Oracle Call Interface

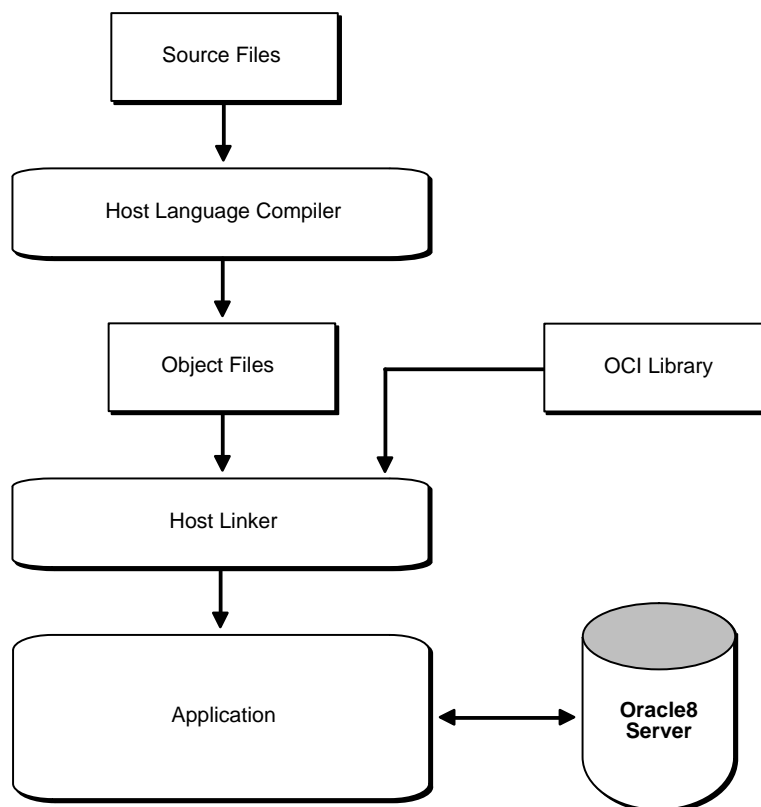
Structured Query Language (SQL) is a nonprocedural language. A program in a nonprocedural language specifies the set of data to be operated on, but does not specify precisely what operations will be performed, or how the operations are to be carried out. The nonprocedural nature of SQL makes it an easy language to learn and to 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.

However, most programming languages, such as C and C++ are procedural. 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 Oracle Call Interface (OCI) allows you to develop applications that combine the nonprocedural data access power of SQL with the procedural capabilities of C. The OCI supports all SQL data definition, data manipulation, query, and transaction control facilities that are available through an Oracle8 server.

You can also take advantage of PL/SQL, Oracle's procedural extension to SQL. Thus, the applications you develop can be more powerful and flexible than applications written in SQL alone. The OCI also provides facilities for accessing and manipulating objects in an Oracle8 server.

The OCI is an application programming interface (API) that allows you to manipulate data and schemas in an Oracle database. As Figure 1-1 shows, you compile and link an OCI program in the same way that you compile and link a nondatabase application. There is no need for a separate preprocessing or precompilation step.

Figure 1–1 The OCI Development Process

Note: On some platforms, it may be necessary to include other libraries, in addition to the OCI library, to properly link your OCI programs. Check your Oracle system-specific documentation for further information about extra libraries that may be required.

SQL Statements

One of the main tasks of an OCI 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 OCI application.

Oracle8 recognizes eight kinds of SQL statements:

- Data Definition Language
- Control Statements (3 types)
 - Transaction Control
 - Session Control
 - System Control
- Data Manipulation Language (DML)
- Queries
- PL/SQL
- Embedded SQL

Note: Queries are often classified as DML statements, but OCI applications process queries differently, so they are considered separately here.

Data Definition Language

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. For example:

```
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 Oracle8 server, as in the following series of statements which creates 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

OCI applications treat transaction control, session control, and system control statements like DML statements. See the *Oracle8 SQL Reference* for information about these types of statements.

Data Manipulation Language

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

- 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 using input (bind) variables. See the section “Binding” on page 4-5 for more information about input bind variables.

DML statements also allow you to work with objects in the Oracle8 server, 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 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 access data in tables, and they are often classified with DML statements. However, OCI applications process queries differently, so they are considered separately in this guide.

Queries can require the program to supply data to the database 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.

When processing a query, an OCI application also needs to define output variables to receive the returned results. In the above statement, you would need to define an output variable to receive any `name` values returned from the query.

See Also: See the section “Binding” on page 5-2 for more information about input bind variables.

See the section “Defining” on page 5-13 for information about defining output variables.

See Chapter 4, “SQL Statement Processing”, for detailed information about how SQL statements are processed in an OCI program.

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:

- one or more SQL statements
- variable declarations
- assignment statements
- procedural control statements (IF...THEN...ELSE statements and loops)
- exception handling

You can use PL/SQL blocks in your OCI program to

- call Oracle 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 Oracle8 server

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.

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

Keep in mind that the placeholders in this statement are not PL/SQL variables. They represent input values passed to Oracle when the statement is processed. These placeholders need to be bound to C language variables in your program.

See Also: See the *PL/SQL User's Guide and Reference* for information about coding PL/SQL blocks.

See the section “Binding Placeholders in PL/SQL” on page 5-5 for information about working with placeholders in PL/SQL.

Embedded SQL

The OCI processes SQL statements as text strings, which an application passes to Oracle on execution. The Oracle precompilers (Pro*C/C++, Pro*COBOL, Pro*FORTRAN) allow programmers to embed SQL statements directly into their application code. A separate precompilation step is then necessary to generate an executable application.

It is possible to mix OCI calls and embedded SQL in a precompiler program. Refer to the *Pro*COBOL Precompiler Programmer's Guide* for more information.

Special OCI/SQL Terms

This guide uses special terms to refer to the different parts of a SQL statement. For example, a SQL statement such as

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

contains the following 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 variable in the second part of the WHERE clause — :sales

When you develop your OCI application, you call routines that specify to the Oracle8 server the address (location) of input and output variables in your program. In this guide, specifying the address of a placeholder variable for data input is called a *bind operation*. Specifying the address of a variable to receive select-list items is called a *define operation*.

For PL/SQL, both input and output specifications are called bind operations.

These terms and operations are described in detail in Chapter 4.

Object Support in the OCI

With Release 8.0, the Oracle server 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 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. 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.

See Also: For a more detailed explanation of object types and objects, see *Oracle8 Concepts*.

The Oracle8 OCI includes functions that extend the capabilities of the OCI to handle objects in an Oracle8 server. Specifically, the following capabilities have been added to the OCI:

- support for execution of SQL statements that manipulate object data and schema information
- support for passing object references and instances as input variables in SQL statements.
- support for declaring object references and instances as variables to receive the output of SQL statements
- support for fetching object references and instances from a database
- support for describing the properties of SQL statements that return object instances and references
- support for describing PL/SQL procedures or functions with object parameters or results
- commit and rollback calls have been extended to synchronize object and relational functionality

Additional OCI calls are provided to support manipulation of objects after they have been accessed by way of SQL statements.

Note: For a more detailed description of enhancements and new features, please refer to Appendix F, “Oracle8 OCI New Features”.

Parts of the OCI

The OCI encompasses four main sets of functionality:

- OCI *relational functions*, for managing database access and processing SQL statements
- OCI *navigational functions*, for manipulating objects retrieved from an Oracle8 server
- OCI *datatype mapping and manipulation functions*, for manipulating data attributes of Oracle8 types
- OCI *external procedure functions*, which are used for writing C callbacks from PL/SQL

These terms are used throughout this guide.

Release 8.0 New Features

The Oracle8 OCI provides a wide range of new features and functions. All calls available in Release 7.3 are still supported, but they are not able to take full advantage of new Oracle8 features.

Note: For a more detailed description of enhancements and new features, please refer to Appendix F, “Oracle8 OCI New Features”.

Release 8.0 has the following new features and performance advantages:

- increased client-side processing and reduced server-side requirements
- implicit prefetching of SELECT statement result set rows
- API access to both objects and relational data
- reduction of the number of network round trips
- the ability to handle LOB columns
- a set of API calls for performing operations on LOBs and FILES
- improved national language support (NLS) capabilities
- a migration path for existing OCI applications, and some ability to mix old and new calls within a single application
- improved support for multithreaded environments
- additional functionality to provide navigational access to objects in an Oracle8 server

Each of these features is discussed in greater detail in later chapters of this guide.

Release 8.0 of the OCI contains an entirely new set of API calls that replace those used in earlier releases. Additionally, new calls are included to provide functionality not available in earlier releases.

See Also: See the section “Obsolescent OCI Routines” on page A-2 for information about new calls that supersede existing routines.

See Chapters 13, 14, 15, and 16 for complete listings of all OCI calls.

Obsolescent and Obsolete OCI Calls

Refer to Appendix A for lists of OCI calls that are now considered to be obsolescent or obsolete.

Compiling and Linking

Oracle Corporation supports most popular third-party compilers. The details of linking an OCI program vary from system to system. See your Oracle system-specific documentation and the installation guide for more information about compiling and linking an OCI application for your specific platform.

OCI Programming Basics

This chapter introduces you to the basic concepts involved in programming with the Oracle Call Interface.

This chapter covers the following topics:

- Overview
- OCI Program Structure
- OCI Data Structures
- Handles
- Descriptors and Locators
- OCI Programming Steps
- Initialization, Connection, and Session Creation
- Processing SQL Statements
- Commit or Rollback
- Terminating the Application
- Error Handling
- Additional Coding Guidelines
- Using PL/SQL in an OCI Program

Overview

This chapter provides an introduction to the concepts and procedures involved in developing an OCI application. After reading this chapter, you should have most of the tools necessary to understand and create a basic OCI application.

New users should pay particular attention to the information presented in this chapter, because it forms the basis for the rest of the material presented in this guide.

This information in this chapter is supplemented by information in later chapters. More specifically, after reading this chapter you may want to continue with any or all of the following:

- Chapter 3, for detailed information about OCI internal and external datatypes
- Chapter 4, for information about processing SQL statements
- Chapter 5, for more information about binding and defining
- Chapter 6, for information about the *OCIDescribe()* call.
- Chapter 7, for a discussion of advanced OCI concepts and techniques
- Chapters 8 through 12, for information about writing OCI applications that take advantage of the object capabilities of the Oracle8 server
- Chapter 13, for a complete listing of all of the OCI relational function calls, including descriptions, syntax, and parameters
- Appendix D, for code examples

This chapter is broken down into the following major sections:

- OCI Program Structure - covers the basic overall structure of an OCI application, including the major steps involved in creating one.
- OCI Data Structures - discusses handles, descriptors, and locators.
- OCI Programming Steps - discusses in detail each of the steps involved in coding an OCI application.
- Error Handling - covers error handling in OCI applications.
- Additional Coding Guidelines - provides additional useful information to keep in mind when coding an OCI application.
- Using PL/SQL in an OCI Program - discusses some important points to keep in mind when working with PL/SQL in an OCI application.

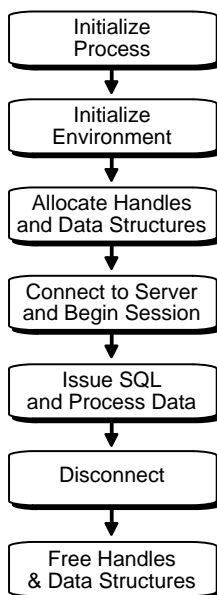
OCI Program Structure

The general goal of an OCI application is to connect to an Oracle server, engage in some sort of data exchange, and perform necessary data processing. While some flexibility exists in the order in which specific tasks can be performed, every OCI application needs to accomplish particular steps.

The OCI uses the following basic program structure:

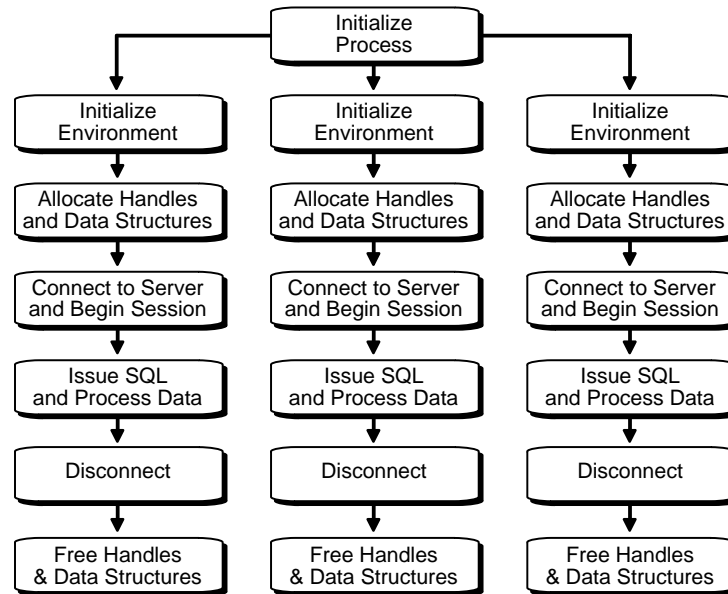
1. Initialize the OCI programming environment and processes.
2. Allocate necessary handles, and establish a server connection and a user session.
3. Issue SQL statements to the server, and perform necessary application data processing.
4. Free statements and handles not to be reused or reexecute prepared statements again, or prepare a new statement.
5. Terminate user session and server connection.

Figure 2-1 illustrates the flow of steps in an OCI application. Each step is described in more detail in the section “OCI Programming Steps” on page 2-16.

Figure 2–1 Basic OCI Program Flow

Keep in mind that the above diagram and the list of steps on page 2-3 present a simple generalization of OCI programming steps. Variations are possible, depending on the functionality of the program. OCI applications that include more sophisticated functionality (e.g., managing multiple transactions, using objects, etc.) will require additional steps.

Once the OCI process is initialized, an application may choose to create multiple environments, as illustrated in the following figure:

Figure 2–2 Multiple Environments Within an OCI Process

Note: It is possible to have more than one active connection and statement in an OCI application.

See Also: For information about accessing and manipulating objects, see Chapter 8.

OCI Data Structures

Handles and *descriptors* are opaque data structures which are defined in OCI applications and may be allocated directly, through specific allocate calls, or may be implicitly allocated by other OCI functions.

7.x Upgrade Note: Programmers who have previously written 7.x OCI applications will need to become familiar with these new data structures which are used by most OCI calls.

Handles and descriptors store information pertaining to data, connections, or application behavior. Handles are defined in more detail in the following section. Descriptors are discussed in the section “Descriptors and Locators” on page 2-12.

Handles

Almost all Oracle8 OCI calls include in their parameter list one or more *handles*. A handle is an opaque pointer to a storage area allocated by the OCI library. A handle may be used to store context or connection information, (e.g., an environment or service context handle), or it may store information about other OCI functions or data (e.g., an error or describe handle). Handles can make programming easier, because the library, rather than the application, maintains this data.

Most OCI applications will need to access the information stored in handles. The get and set attribute OCI calls, *OCIAttrGet()* and *OCIAttrSet()*, access this information.

See Also: For more information about using handle attributes, see the section “Handle Attributes” on page 2-11.

The following table lists the handles defined for the OCI. For each handle type, the C datatype and handle type constant (used to identify the handle type in OCI calls) are listed.

Table 2–1 *OCI Handle Types*

C Type	Description	Handle Type
OCIEnv	OCI environment handle	OCI_HTYPE_ENV
OCIError	OCI error handle	OCI_HTYPE_ERROR
OCISvcCtx	OCI service context handle	OCI_HTYPE_SVCCTX
OCIStmt	OCI statement handle	OCI_HTYPE_STMT
OCIBind	OCI bind handle	OCI_HTYPE_BIND
OCIDefine	OCI define handle	OCI_HTYPE_DEFINE
OCIDescribe	OCI describe handle	OCI_HTYPE_DESCRIBE
OCIServer	OCI server handle	OCI_HTYPE_SERVER
OCISession	OCI user session handle	OCI_HTYPE_SESSION
OCITrans	OCI transaction handle	OCI_HTYPE_TRANS
OCIComplexObject	OCI complex object retrieval (COR) handle	OCI_HTYPE_COMPLEXOBJECT
OCISecurity	OCI security service handle	OCI_HTYPE_SECURITY

Allocating and Freeing Handles

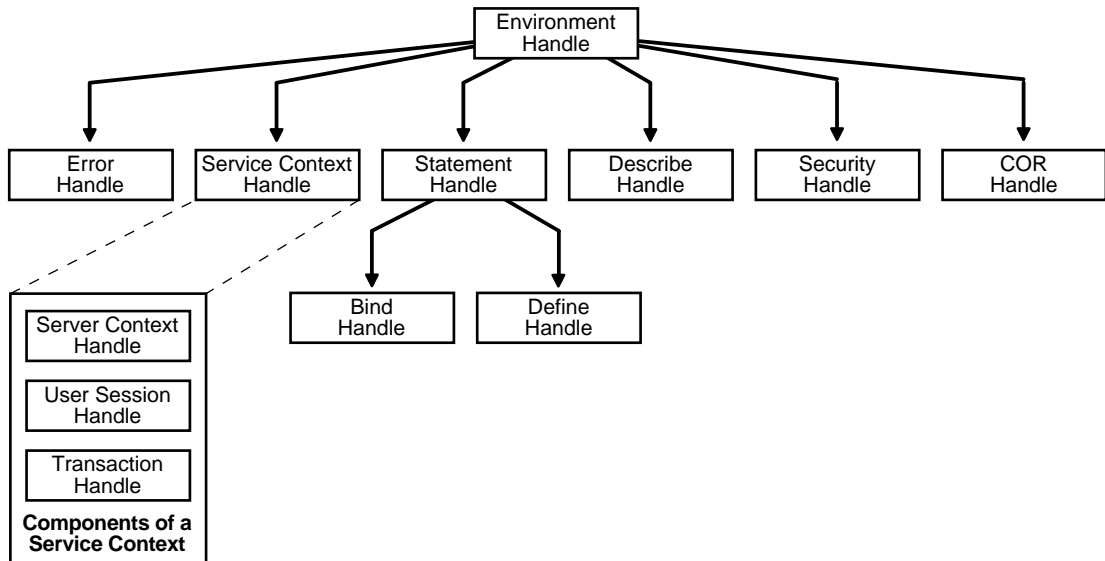
Your application allocates all handles (except the bind and define handles) with respect to particular environment handle. You pass the environment handle as one of the parameters to the handle allocation call. The allocated handles is then specific to that particular environment.

The bind and define handles are allocated with respect to a statement handle, and contain information about the statement represented by that handle.

Note: The bind and define handles are implicitly allocated by the OCI library, and do not require user allocation.

Figure 2–3 illustrates the relationship between the various types of handles.

Figure 2–3 Hierarchy of Handles:



All user-allocated handles, except the environment handle, must be allocated using the OCI handle allocation call, *OCIHandleAlloc()*. The environment handle is allocated and initialized with a call to *OCIEnvInit()*, which is required by all OCI applications.

An application must free all handles when they are no longer needed. The *OCIHandleFree()* function frees handles.

Note: When a parent handle is freed, all child handles associated with it are also freed, and may no longer be used. For example, when a statement handle is freed, any bind and define handles associated with it are also freed.

Handles obviate the need for global variables. Handles also make error reporting easier. An error handle is used to return errors and diagnostic information.

See Also: For sample code demonstrating the allocation and use of OCI handles, see the first example program in Appendix D.

The various handle types are described in more detail in the following sections.

Environment Handle

The *environment handle* defines a context in which all OCI functions are invoked. Each environment handle contains a memory cache, which allows for fast memory management in a threaded environment where each thread has its own environment. When multiple threads share a single environment, they may block on access to the cache.

The environment handle is passed as the *parenth* parameter to the *OCIHandleAlloc()* call to allocate all other handle types, except for the bind and define handles.

Error Handle

The *error handle* is passed as a parameter to most OCI calls. The error handle maintains information about errors that occur during an OCI operation. If an error occurs in a call, the error handle can be passed to *OCIErrorGet()* to obtain additional information about the error that occurred.

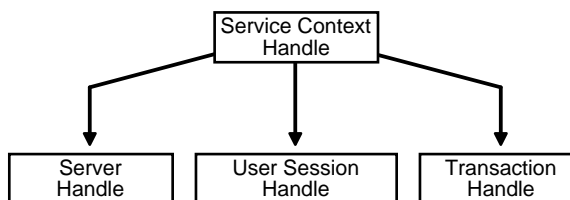
Allocating the error handle is one of the first steps in an OCI application.

Service Context and Associated Handles

A *service context handle* defines attributes that determine the operational context for OCI calls to a server. You must allocate and initialize the service context handle with *OCIHandleAlloc()* or *OCILogon()* before you can use it.

The service context contains three additional handles that represent a server connection, a user session, and a transaction, as illustrated in the following figure.

Figure 2–4 Components of a Service Context



- A *server handle* identifies a data source. It translates into a physical connection in a connection-oriented transport mechanism.
- A *user session handle* defines a user's roles and privileges (also known as the user's security domain), and the operational context on which the calls execute.
- A *transaction handle* defines the transaction in which the SQL operations are performed. The transaction context includes user session state information, including the fetch state and package instantiation, if any.

Breaking the service context down in this way provides scalability and enables programmers to create sophisticated three-tiered applications and transaction processing (TP) monitors to execute requests on behalf of multiple users on multiple application servers and different transaction contexts.

Applications maintaining only a single user session per database connection at any time can call *OCILogon()* to allocate the service context and its associated handles.

In applications requiring more complex session management, the service context must be explicitly allocated, and the server handle and user session handle must be explicitly set into the service context by calling *OCIServerAttach()* and *OCISessionBegin()*, respectively. An application may need to define a transaction explicitly, as well, or it may be able to work with the implicit transaction created when the application makes changes to the database.

See Also: For more information about transactions, see the section “Transactions” on page 7-3.

For more information about establishing a server connection and user session, see the sections “Initialization, Connection, and Session Creation” on page 2-17, and “User Authentication and Password Management” on page 7-11.

Statement Handle, Bind Handle, and Define Handle

A *statement handle* is the context that identifies a SQL or PL/SQL statement and its associated attributes.

Information about input variables is stored in *bind handles*. The OCI library allocates a bind handle for each placeholder bound with the *OCIBindByName()* or *OCIBindByPos()* function. The user does not need to allocate bind handles. They are implicitly allocated by the bind call.

Fetches data returned by a query is converted and stored according to the specifications of the *define handles*. The OCI library allocates a define handle for each output variable defined with *OCIDefineByPos()*. The user does not need to allocate define handles. They are implicitly allocated by the define call.

Describe Handle

The *describe handle* is used by the OCI describe call, *OCIDescribeAny()*. This call obtains information about schema objects in a database (e.g., functions, procedures). The call takes a describe handle as one of its parameters, along with information about the object being described. When the call completes, the describe handle is populated with information about the object. The OCI application can then obtain describe information through the attributes of parameter descriptors.

See Also: See Chapter 6, “Describing Schema Metadata”, for more information about using the *OCIDescribeAny()* function.

Complex Object Retrieval Handle

The *complex object retrieval (COR) handle* is used by some OCI applications that work with objects in an Oracle8 server. This handle contains *COR descriptors*, which provide instructions to the OCI about retrieving objects referenced by another object.

See Also: For information about complex object retrieval and the complex object retrieval handle, refer to “Complex Object Retrieval” on page 8-21.

Security Handle

For information about the security handle, and about using OCI calls to write Oracle Security Services applications, refer to the *Oracle Security Server Guide*.

Handle Attributes

All OCI handles have *attributes* associated with them. These attributes represent data stored in that handle. You can read handle attributes using the attribute get call, *OCIAttrGet()*, and you can change them with the attribute set call, *OCIAttrSet()*.

For example, the following statements set the username in the transaction handle by writing to the OCI_ATTR_USERNAME attribute:

```
text username[] = "scott";
err = OCIAttrSet ((dvoid*) mysessp, OCI_HTYPE_SESSION, (dvoid*) username,
                 (ub4) strlen(username), OCI_ATTR_USERNAME,
                 (OCIError *) myerrhp);
```

The next set of statements demonstrates the use of *OCIAttrGet()* to read the function code of the last OCI function processed on a handle (in this case a bind handle):

```
ub4 fcode = 0;
OCIBind *mybndp;
err = OCIAttrGet( (dvoid*) mybndp, OCI_HTYPE_BIND, (dvoid*) &fcode,
                 (ub4) 0, OCI_ATTR_FNCODE, (OCIError *) myerrhp);
```

Some OCI functions require that particular handle attributes be set before the function is called. For example, when *OCISessionBegin()* is called to establish a user's login session, the username and password must be set in the user session handle before the call is made.

Other OCI functions provide useful return data in handle attributes after the function completes. For example, when *OCIStmtExecute()* is called to execute a SQL query, describe information relating to the select-list items is returned in the statement handle.

For a list of all handle attributes, refer to Appendix B, "Handle and Descriptor Attributes".

See Also: See the description of *OCIAttrGet()* on page 13 - 11 for an example showing the username and password handle attributes being set.

User Memory Allocation

The *OCIEnvInit()* call, which initializes the environment handle, and the generic handle allocation (*OCIHandleAlloc()*) and descriptor/locator allocation (*OCIDescriptorAlloc()*) calls have an *xtrmem_sz* parameter in their parameter list. This parameter is used to specify an amount of user memory which should be allocated along with that handle.

Typically, an application uses this parameter to allocate an application-defined structure that has the same lifetime as the handle. This structure maybe used for application “bookkeeping” or storing context information.

Using the *xtrmem_sz* parameter means that the application does not need to explicitly allocate and deallocate memory as each handle is allocated and deallocated. The memory is allocated along with the handle, and freeing the handle frees up the user’s data structures as well.

Descriptors and Locators

OCI *descriptors* and *locators* are opaque data structures that maintain specific data-information. The OCI has six descriptor and locator types. The following table lists them, along with their C datatype, and the OCI type constant that allocates a descriptor of that type in a call to *OCIDescriptorAlloc()*. The *OCIDescriptorFree()* function frees descriptors and locators.

Table 2–2 Descriptor Types

C Type	Description	OCI Type Constant
OCISnapshot	snapshot descriptor	OCI_DTYPE_SNAP
OCILobLocator	LOB datatype locator	OCI_DTYPE_LOB
OCILobLocator	FILE datatype locator	OCI_DTYPE_FILE
OCIParam	read-only parameter descriptor	OCI_DTYPE_PARAM
OCIRowid	ROWID descriptor	OCI_DTYPE_ROWID
OCIComplexObjectComp	complex object descriptor	OCI_DTYPE_COMPLEXOBJECTCOMP
OCIAQEnqOptions	advanced queueing enqueue options	OCI_DTYPE_AQENQ_OPTIONS
OCIAQDeqOptions	advanced queueing dequeue options	OCI_DTYPE_AQDEQ_OPTIONS
OCIAQMsgProperties	advanced queueing message properties	OCI_DTYPE_AQMSG_PROPERTIES
OCIAQAgent	advanced queueing agent	OCI_DTYPE_AQAGENT

Note: Although there is a single C type for **OCILobLocator**, this locator is allocated with a different OCI type constant for internal and external LOBs. The section below on LOB locators discusses this difference.

The main purpose of each descriptor type is listed here, and each descriptor type is described in the following sections:

- **OCISnapshot** - used in statement execution
- **OCILOBLocator** - used for LOB (OCI_DTYPE_LOB) or FILE (OCI_DTYPE_FILE) calls
- **OCIPParam** - used in describe calls
- **OCIRowid** - used for binding or defining ROWID values
- **OCIComplexObjectComp** - used for complex object retrieval
- **OCIAQEnqOptions**, **OCIAQDeqOptions**, **OCIAQMsgProperties**, **OCIAQAgent** - used for advanced queueing

Snapshot Descriptor

The *snapshot descriptor* is an optional parameter to the `execute` call, `OCISstmtExecute()`. It indicates that a query is being executed against a particular database snapshot. A database snapshot represents the state of a database at a particular point in time.

You allocate a snapshot descriptor with a call to `OCIDescriptorAlloc()`, by passing `OCI_DTYPE_SNAP` as the *type* parameter.

See Also: For more information about `OCISstmtExecute()` and database snapshots, see the section “Execution Snapshots” on page 4-7.

LOB/FILE Datatype Locator

A LOB (large object) is an Oracle datatype that can hold up to 4 gigabytes of binary (BLOB) or character (CLOB) data. In the database, an opaque data structure called a *LOB locator* is stored in a LOB column of a database row, or in the place of a LOB attribute of an object. The locator serves as a pointer to the actual LOB value, which is stored in a separate location.

The OCI *LOB locator* is used to perform OCI operations against a LOB (BLOB or CLOB) or FILE (BFILE). OCI functions do not take actual LOB values as parameters; all OCI calls operate on the LOB locator. This descriptor—**OCILobLocator**—is also used for operations on FILES.

The LOB locator is allocated with a call to *OCIDescriptorAlloc()*, by passing `OCI_DTYPE_LOB` as the *type* parameter for BLOBs or CLOBs, and `OCI_DTYPE_FILE` for BFILEs.

Warning: The two LOB locator types are *not* interchangeable. When binding or defining a BLOB or CLOB, the application must take care that the locator is properly allocated using `OCI_DTYPE_LOB`. Similarly, when binding or defining a BFILE, the application must be sure to allocate the locator using `OCI_DTYPE_FILE`.

An OCI application can retrieve a LOB locator from the server by issuing a SQL statement containing a LOB column or attribute as an element in the select list. In this example, the application would first allocate the LOB locator and then use it to define an output variable.

Similarly, a LOB locator can be used as part of a bind operation to create an association between a LOB and a placeholder in a SQL statement.

The LOB locator datatype (**OCILobLocator**) is not a valid datatype when connected to an Oracle7 Server.

See Also: For more information about OCI LOB operations, see the section “LOB and FILE Operations” on page 7-24.

Parameter Descriptor

OCI applications use *parameter descriptors* to obtain information about select-list columns or schema objects. This information is obtained through a describe operation.

The parameter descriptor is the one descriptor type that is *not* allocated using *OCIDescriptorAlloc()*. You can obtain it only as an attribute of a describe, statement, or complex object retrieval handle by specifying the position of the parameter using an *OCIParamGet()* call.

See Also: See Chapter 6, “Describing Schema Metadata”, and “Describing Select-List Items” on page 4-8 for more information about obtaining and using parameter descriptors.

ROWID Descriptor

The ROWID descriptor is used by applications that need to retrieve and use Oracle ROWIDs. The size and structure of the ROWID has changed from Oracle7 to Oracle8, and is opaque to the user. To work with a ROWID using the Oracle8 OCI, an application can define a ROWID descriptor for a position in a SQL select-list, and retrieve a ROWID into the descriptor. This same descriptor can later be bound to an input variable in an INSERT statement or WHERE clause.

Complex Object Descriptor

For information about the complex object descriptor and its use, refer to “Complex Object Retrieval” on page 8-21.

Advanced Queueing Descriptors

For information about advanced queueing and its related descriptors, refer to “OCI and Advanced Queueing” on page 7-40.

User Memory Allocation

The *OCIDescriptorAlloc()* call has an *xtrmem_sz* parameter in its parameter list. This parameter is used to specify an amount of user memory which should be allocated along with a descriptor or locator.

Typically, an application uses this parameter to allocate an application-defined structure that has the same lifetime as the descriptor or locator. This structure maybe used for application “bookkeeping” or storing context information.

Using the *xtrmem_sz* parameter means that the application does not need to explicitly allocate and deallocate memory as each descriptor or locator is allocated and deallocated. The memory is allocated along with the descriptor or locator, and freeing the descriptor or locator (with *OCIDescriptorFree()*) frees up the user’s data structures as well.

The *OCIHandleAlloc()* call has a similar parameter for allocating user memory which will have the same lifetime as the handle.

The *OCIEnvInit()* call has a similar parameter for allocating user memory which will have the same lifetime as the environment handle.

OCI Programming Steps

Each of the steps that you perform in an OCI application is described in greater detail in the following sections. Some of the steps are optional. For example, you do not need to describe or define select-list items if the statement is not a query.

Note: For an example showing the use of OCI calls for processing SQL statements, see the first sample program in Appendix D.

The special case of dynamically providing data at run time is described in detail in the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

Special considerations for operations involving arrays of structures are described in the section “Arrays of Structures” on page 5-17.

Refer to the section “Error Handling” on page 2-25 for an outline of the steps involved in processing a SQL statement within an OCI program.

For information on using the OCI to write multithreaded applications, refer to “Thread Safety” on page 7-13.

For more information about types of SQL statements, refer to the section “SQL Statements” on page 1-4.

The following sections describe the steps that are required of a release 8.0 OCI application:

- Initialization, Connection, and Session Creation
- Processing SQL Statements
- Commit or Rollback
- Terminating the Application
- Error Handling

Application-specific processing will also occur in between any and all of the OCI function steps.

7.x Upgrade Note: OCI programmers should take note that OCI programs no longer require an explicit parse step. This means that 8.0 applications must issue an execute command for both DML and DDL statements.

Initialization, Connection, and Session Creation

This section describes how to initialize the Oracle8 OCI environment, establish a connection to a server, and authorize a user to perform actions against a database.

The three main steps in initializing the OCI environment are described in this section:

1. Initialize an OCI Process
2. Allocate Handles and Descriptors
3. Initialize the Application, Connection, and Authorization

Additionally, this section describes connection modes for OCI applications.

Initialize an OCI Process

The initialize process call, *OCIInitialize()*, must be invoked before any other OCI call. The *mode* parameter of this call specifies whether the application will run in a threaded environment (*mode* = *OCI_THREADED*), and whether or not it will use objects (*mode* = *OCI_OBJECT*). Initializing in object mode is necessary if the application will be binding and defining objects, or if the application will be using the OCI's object navigation calls.

The program may also choose to use neither of these features (*mode* = *OCI_DEFAULT*) or both, separating the options with a vertical bar (*mode* = (*OCI_THREADED* | *OCI_OBJECT*)).

The *OCIInitialize()* call can also specify user-defined memory management functions.

See Also: See the description of *OCIInitialize()* on page 13-72 for more information about the call.

For information about using the OCI to write multithreaded applications, refer to “Thread Safety” on page 7-13.

Allocate Handles and Descriptors

Oracle provides OCI functions to allocate and deallocate handles and descriptors. You must allocate handles using *OCIHandleAlloc()* before passing them into an OCI call, unless the OCI call allocates the handles for you (e.g. *OCIBindByPos()*).

You can allocate the following types of handles with *OCIHandleAlloc()*:

- error handle
- service context handle
- statement handle
- describe handle
- server handle
- user session handle
- transaction handle
- complex object retrieval handle

Depending on the functionality of your application, it will need to allocate some or all of these handles.

See Also: See the description of *OCIHandleAlloc()* on page 13-68 for more information about using this call.

Application Initialization, Connection, and Session Creation

Once *OCIInitialize()* has been called, an application must call *OCIEnvInit()* to initialize the OCI environment handle. Following this step, the application has two options for establishing a server connection and beginning a user session: Single User, Single Connection; or Multiple Sessions or Connections.

Option 1: Single User, Single Connection

This option is the simplified logon method.

If an application will maintain only a single user session per database connection at any time, the application can take advantage of the OCI's simplified logon procedure.

When an application calls *OCILogon()*, the OCI library initializes the service context handle that is passed to it and creates a connection to the specified server for the user whose username and password are passed to the function.

The following is an example of what a call to *OCILogon()* might look like:

```
OCILogon(envhp, errhp, &svchp, "scott", nameLen, "tiger",  
        passwdLen, "oracle8", dbnameLen)
```

The parameters to this call include the service context handle (which will be initialized), the username, the user's password, and the name of the database that will be used to establish the connection. The server and user session handles are also implicitly allocated by this function.

If an application uses this logon method, the service context, server, and user session handles will all be "read only", which means that the application cannot switch session or transaction by changing the appropriate attributes of the service context handle, using *OCIAttrSet()*.

An application that creates its session and authorization using *OCILogon()* should terminate them using *OCILogoff()*.

Option 2: Multiple Sessions or Connections

This option uses explicit attach and begin session calls.

If an application needs to maintain multiple user sessions on a database connection, the application requires a different set of calls to set up the sessions and connections. This includes specific calls to attach to the server and begin sessions:

- *OCIServerAttach()* creates an access path to a data source for OCI operations.
- *OCISessionBegin()* establishes a session for a user against a particular server. This call is required for the user to be able to execute any operation on the server.

These calls set up an operational environment that allows you to execute SQL and PL/SQL statements against a database. The database must be up and running before the calls are made, or else they will fail.

These calls are described in more detail in Chapter 13. Refer to Chapter 7, "OCI Programming Advanced Topics", for more information about maintaining multiple sessions, transactions, and connections.

Example

The following example demonstrates the use of the OCI initialization calls. In the example, a server context is created and set in the service handle. Then a user session handle is created and initialized using a database username and password. For the sake of simplicity, error checking is not included.

```

main()
{
    OCIEnv *myenvhp; /* the environment handle */
    OCIServer *mysrvhp; /* the server handle */
    OCIError *myerrhp; /* the error handle */
    OCISession *myusrhp; /* user session handle */

    (void) OCIInitialize (OCI_THREADED | OCI_OBJECT, (dvoid *)0,
        mymalloc, myrealloc, myfree);
    /* initialize the mode to be the threaded and object environment */

    (void) OCIEnvInit (&myenvhp, OCI_DEFAULT, 0, (dvoid **)0);

    (void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&mysrvhp,
        OCI_HTYPE_SVR, 0, (dvoid **) 0);

        /* allocate a server handle */

    (void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&myerrhp,
        OCI_HTYPE_ERROR, 0, (dvoid **) 0);

        /* allocate an error handle */

    (void) OCIServerAttach (mysrvhp, myerrhp, (text *)"inst1_alias",
        strlen ("inst1_alias"), OCI_DEFAULT);

        /* create a server context */

    (void) OCIAttrSet ((dvoid *)mysvchp, OCI_HTYPE_SVCCTX,
        (dvoid *)mysrvhp, (ub4) 0, OCI_ATTR_SERVER, myerrhp);

    /* set the server context in the service context */

    (void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&myusrhp,
        OCI_HTYPE_SESSION, 0, (dvoid **) 0);

        /* allocate a user session handle */

    (void) OCIAttrSet ((dvoid *)myusrhp, OCI_HTYPE_SESSION,
        (dvoid *)"scott", (ub4)strlen("scott"),
        OCI_ATTR_USERNAME, myerrhp);

        /* set username attribute in user session handle */

```

```
(void) OCIAttrSet ((dvoid *)myusrhp, OCI_HTYPE_SESSION,
    (dvoid *)"tiger", (ub4)strlen("tiger"),
    OCI_ATTR_PASSWORD, myerrhp);

/* set password attribute in user session handle */

(void) OCISessionBegin ((dvoid *) mysvchp, myerrhp, myusrhp,
    OCI_CRED_RDBMS, OCI_DEFAULT);

(void) OCIAttrSet ( (dvoid *)mysvchp, OCI_HTYPE_SVCCTX,
    (dvoid *)myusrhp, (ub4) 0, OCI_ATTR_SESSION, myerrhp);
/* set the user session in the service context */
```

Understanding Multiple Connections and Handles

This section presents one possible scenario for an application which is managing multiple user, multiple server connections, and multithreading. This example is intended to help the reader understand some of the issues involved in programming such an application.

A Connection Example

An application is supporting two users, User1 and User2. The application has completed the following steps:

- initialized the OCI process in OCI_THREADED mode with a call to `OCIInitialize()`
- allocated a single environment handle with `OCIEnvInit()`
- in two different threads, connected to two different databases, DB1 and DB2, residing on the same machine

User1 performs the following actions:

- Attaches to DB1
- Starts two new transactions, TX1 and TX2
- Prepares and executes a statement in each transaction at the same time in different threads (STMT1 in TX1, STMT2 in TX2).
- Commits TX1 and TX2
- Detaches from DB1

User2 performs the following actions:

- Attaches to DB2
- Starts two new transactions, TX3 and TX4
- Prepares and executes a statement in each transaction at the same time (STMT3 in TX3, STMT4 in TX4).
- Commits TX3 and TX4
- Detaches from DB2

The following questions and answers relate to the above scenario:

Q1. How many server handles are required?

A1. Even though DB1 and DB2 reside on the same server machine, 2 server handles are required. Each server handle represents a database connection, and is identified by its own connect string.

Q2. How many service context handles are required?

A2. Four service context handles are required. Each user is executing two transactions simultaneously, so each requires its own service context. 2 users x 2 transactions = 4 service context handles. If each user had executed the statements in the same transaction, each would require only a single service context.

Q3. How many user session handles are required?

A3. Four user session handles are required. Each user needs a user session handle on each server. If each user executed their statements serially, then two sessions would be sufficient.

Q4. How many transaction handles are required?

A4. Four transaction handles are required; one for each concurrent transaction. However, the application could also take advantage of the implicit transaction created when database changes are made, and avoid allocating transaction handles altogether.

Q5. Could the example use multiple environment handles?

A5. Yes. Since there are two databases involved, the application should use two environment handles so that accesses to each database can be completely concurrent.

Q6. If a single user in a single environment wants to execute four different statements on 4 transactions concurrently against the same database, how many server handles are required?

A6. Four server handles are required; one for each concurrent transaction. There can be at most a single outstanding call on any one server handle at a time.

Processing SQL Statements

For information about processing SQL statements, refer to Chapter 4, “SQL Statement Processing”.

Commit or Rollback

An application commits changes to the database by calling *OCITransCommit()*. This call takes a service context as one of its parameters. The transaction currently associated with the service context is the one whose changes are committed. This may be a transaction explicitly created by the application or the implicit transaction created when the application modifies the database.

Note: Using the `OCI_COMMIT_ON_SUCCESS` mode of the *OCIExecute()* call, the application can selectively commit transactions at the end of each statement execution.

If you want to roll back a transaction, use the *OCITransRollback()* call.

If an application disconnects from Oracle in some way other than a normal logoff (for example, losing a network connection), and *OCITransCommit()* has not been called, all active transactions are rolled back automatically.

See Also: For more information about implicit transactions and transaction processing, see the section “Service Context and Associated Handles” on page 2-8, and the section “Transactions” on page 7-3.

Terminating the Application

An OCI application should perform the following three steps before it terminates:

1. Delete the user session by calling *OCISessionEnd()* for each session.
2. Delete access to the data source(s) by calling *OCIServerDetach()* for each source.
3. Explicitly deallocate all handles by calling *OCIHandleFree()* for each handle, or
4. Delete the environment handle, which deallocates all other handles associated with it.

Note: When a parent OCI handle is freed, any child handles associated with it are freed automatically.

The calls to *OCIServerDetach()* and *OCISessionEnd()* are not mandatory. If the application terminates, and *OCITransCommit()* (transaction commit) has not been called, any pending transactions are automatically rolled back. For an example showing handles being freed at the end of an application, refer to the first sample program in Appendix D, “Code Examples”.

Note: If the application has used the simplified logon method provided by *OCILogon()*, then a call to *OCILogoff()* will terminate the session, disconnect from the server, and free the service context and associated handles. The application is still responsible for freeing other handles it has allocated.

Error Handling

OCI function calls have a set of return codes, listed below in Table 2–3, which indicate the success or failure of the call (e.g., `OCI_SUCCESS` or `OCI_ERROR`) or provide other information that may be required by the application (e.g., `OCI_NEED_DATA` or `OCI_STILL_EXECUTING`). Most OCI calls return one of these codes. For exceptions, see “Functions Returning Other Values” on page 2-27.

Table 2–3 OCI Return Codes

OCI Return Code	Description
<code>OCI_SUCCESS</code>	The function completed successfully.
<code>OCI_SUCCESS_WITH_INFO</code>	The function completed successfully; a call to <code>OCIErrorGet()</code> will return additional diagnostic information. This may include warnings.
<code>OCI_NO_DATA</code>	The function completed, and there is no further data.
<code>OCI_ERROR</code>	The function failed; a call to <code>OCIErrorGet()</code> will return additional information.
<code>OCI_INVALID_HANDLE</code>	An invalid handle was passed as a parameter. No further diagnostics are available.
<code>OCI_NEED_DATA</code>	The application must provide run-time data.

If the return code indicates that an error has occurred, the application can retrieve Oracle-specific error codes and messages by calling `OCIErrorGet()`. One of the parameters to `OCIErrorGet()` is the error handle passed to the call that caused the error.

Note: Multiple error records can be retrieved by calling `OCIErrorGet()` repeatedly until there are no more records (`OCI_NO_DATA` is returned). `OCIErrorGet()` returns at most a single diagnostic record at any time.

The following example code, taken from the first sample program in Appendix D, “Code Examples”, returns error information given an error handle and the return code from an OCI function call. If the return code is `OCI_ERROR`, the function prints out diagnostic information. `OCI_SUCCESS` results in no printout, and other return codes print the return code information.

```
STATICF void checkerr(errhp, status)
OCIError *errhp;
sword status;
```

```
{
    text errbuf[512];
    ub4 buflen;
    ub4 errcode;

    switch (status)
    {
    case OCI_SUCCESS:
        break;
    case OCI_SUCCESS_WITH_INFO:
        (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
        break;
    case OCI_NEED_DATA:
        (void) printf("Error - OCI_NEED_DATA\n");
        break;
    case OCI_NO_DATA:
        (void) printf("Error - OCI_NODATA\n");
        break;
    case OCI_ERROR:
        (void) OCIErrGet (errhp, (ub4) 1, (text *) NULL, &errcode,
                        errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
        (void) printf("Error - %s\n", errbuf);
        break;
    case OCI_INVALID_HANDLE:
        (void) printf("Error - OCI_INVALID_HANDLE\n");
        break;
    case OCI_STILL_EXECUTING:
        (void) printf("Error - OCI_STILL_EXECUTE\n");
        break;
    default:
        break;
    }
}
```

Functions Returning Other Values

Some functions return values other than the OCI error codes listed in Table 2–3. When using these function be sure to take into account that they return a value directly from the function call, rather than through an OUT parameter. More detailed information about each function and its return values is listed in *Volume II*.

- *OCICollMax()*
- *OCIRawPtr()*
- *OCIRawSize()*
- *OCIRefHexSize()*
- *OCIRefIsEqual()*
- *OCIRefIsNull()*
- *OCIStringPtr()*
- *OCIStringSize()*

Additional Coding Guidelines

This section explains some additional factors to keep in mind when coding applications using the Oracle Call Interface.

Parameter Types

OCI functions take a variety of different types of parameters, including integers, handles, and character strings. Special considerations must be taken into account for some types of parameters, as described in the following sections.

For more information about parameter datatypes and parameter passing conventions, refer to the introductory section in Chapter 13, “OCI Relational Functions”, which covers the function calls for the OCI.

Address Parameters

Address parameters pass the address of the variable to Oracle. You should be careful when developing in C, which normally passes scalar parameters by value, to make sure that the parameter is an address. In all cases, you should pass your pointers carefully.

Integer Parameters

Binary integer parameters are numbers whose size is system dependent. Short binary integer parameters are smaller numbers whose size is also system dependent. See your Oracle system-specific documentation for the size of these integers on your system.

Character String Parameters

Character strings are a special type of address parameter. This section describes additional rules that apply to character string address parameters.

Each OCI routine that allows a character string to be passed as a parameter also has a string length parameter. The length parameter should be set to the length of the string.

7.x Upgrade Note: Unlike earlier versions of the OCI, in release 8.0 you should not pass -1 for the string length parameter of a null-terminated string.

Nulls

You can insert a null into a database column in several ways. One method is to use a literal NULL in the text of an INSERT or UPDATE statement. For example, the SQL statement

```
INSERT INTO emp (ename, empno, deptno)
VALUES (NULL, 8010, 20)
```

makes the ENAME column null.

Another method is to use indicator variables in the OCI bind call. See the section “Indicator Variables” on page 2-29 for more information.

One other method to insert a NULL is to set the buffer length and maximum length parameters both to zero on a bind call.

Note: Following SQL92 requirements, Oracle8 returns an error if an attempt is made to fetch a null select-list item into a variable that does not have an associated indicator variable specified in the define call.

Indicator Variables

Each bind and define OCI call has a parameter that allows you to associate an indicator variable, or an array of indicator variables if you are using arrays, with a DML statement, PL/SQL statement, or query.

Host languages do not have the concept of null values; therefore you associate indicator variables with input variables to specify whether the associated placeholder is a NULL. When data is passed to Oracle, the values of these indicator variables determine whether or not a NULL is assigned to a database field.

For output variables, indicator variables determine whether the value returned from Oracle is a NULL or a truncated value. In the case of a NULL fetch (on *OCIStmtFetch()*) or a truncation (on *OCIStmtExecute()* or *OCIStmtFetch()*), the OCI call returns *OCI_SUCCESS_WITH_INFO*. The corresponding indicator variable is set to the appropriate value, as listed in the “Output” section below. If the application provided a return code variable in the corresponding *OCIDefineByPos()* call, the OCI assigns a value of ORA-01405 (for NULL fetch) or ORA-01406 (for truncation) to the return code variable.

The datatype of indicator variables is **sb2**. In the case of arrays of indicator variables, the individual array elements should be of type **sb2**.

Input

For input host variables, the OCI application can assign the following values to an indicator variable:

Input Indicator Value	Action Taken by Oracle
-1	Oracle assigns a NULL to the column, ignoring the value of the input variable.
>=0	Oracle assigns the value of the input variable to the column.

Output

On output, Oracle can assign the following values to an indicator variable:

Output Indicator Value	Meaning
-2	The length of the item is greater than the length of the output variable; the item has been truncated. Additionally, the original length is longer than the maximum data length that can be returned in the sb2 indicator variable.
-1	The selected value is null, and the value of the output variable is unchanged.
0	Oracle assigned an intact value to the host variable.
>0	The length of the item is greater than the length of the output variable; the item has been truncated. The positive value returned in the indicator variable is the actual length before truncation.

Indicator Variables for Named Data Types and REFs

Indicator variables for most new (release 8.0) datatypes function as described above. The only exception is `SQLT_NTY` (a named datatype). Data of type `SQLT_REF` uses a standard scalar indicator, just like other variable types. For data of type `SQLT_NTY`, the indicator variable must be a pointer to an indicator structure.

When database types are translated into C struct representations using the Object Type Translator (OTT), a null indicator structure is generated for each object type. This structure includes an atomic null indicator, plus indicators for each object attribute.

See Also: See the documentation for the OTT in Chapter 12, “Using the Object Type Translator”, and the section “Nullness” on page 8-28 of this manual for information about null indicator structures.

See the descriptions of `OCIBindByName()` and `OCIBindByPos()` in Chapter 13, and the sections “Additional Information for Named Data Type and REF Binds” on page 10-3, and “Additional Information for Named Data Type and REF Defines, and PL/SQL OUT Binds” on page 10-5, for more information about setting indicator parameters for named datatypes and REFs.

Canceling Calls

On most platforms, you can cancel a long-running or repeated OCI call. You do this by entering the operating system's interrupt character (usually CTRL-C) from the keyboard.

Note: This is not to be confused with cancelling a cursor, which is accomplished by calling *OCIStmtFetch()* with the *nrows* parameter set to zero.

When you cancel the long-running or repeated call using the operating system interrupt, the error code ORA-01013 ("user requested cancel of current operation") is returned.

Given a particular service context pointer or server context pointer, the *OCIBreak()* function performs an immediate (asynchronous) abort of any currently executing OCI function that is associated with the server. It is normally used to stop a long-running OCI call being processed on the server.

Positioned Updates and Deletes

You can use the binary ROWID associated with a SELECT...FOR UPDATE OF... statement in a later UPDATE or DELETE statement. The ROWID is retrieved by calling *OCIAttrGet()* on the statement handle to retrieve the handle's *OCI_ATTR_ROWID* attribute.

For example, for a SQL statement such as

```
SELECT ename FROM emp WHERE empno = 7499 FOR UPDATE OF sal
```

when the fetch is performed, the ROWID attribute in the handle contains the row identifier of the SELECTed row. You can retrieve the ROWID into a buffer in your program by calling *OCIAttrGet()* as follows:

```
OCIRowid *rowid; /* the rowid in opaque format */
/* allocate descriptor with OCIDescriptorAlloc() */
err = OCIAttrGet ((dvoid*) mystmt, OCI_HTYPE_STMT,
    (dvoid*) &rowid, (ub4 *) 0, OCI_ATTR_ROWID, (OCIError *) myerrhp);
```

You can then use the saved ROWID in a DELETE or UPDATE statement. For example, if *MY_ROWID* is the buffer in which the row identifier has been saved, you can later process a SQL statement such as

```
UPDATE emp SET sal = :1 WHERE rowid = :2
```

by binding the new salary to the :1 placeholder and MY_ROWID to the :2 placeholder. Be sure to use datatype code 104 (ROWID descriptor) when binding MY_ROWID to :2.

Application Linking

For information about application linking modes, including Oracle support for non-deferred linking and single task linking in various versions of the OCI, please refer to "Application Linking Issues" on page A-7.

Using PL/SQL in an OCI Program

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 (DML) statements. PL/SQL allows you to group a number of constructs into a single block and execute them as a unit. These constructs include:

- one or more SQL statements
- variable declarations
- assignment statements
- procedural control statements such as IF...THEN...ELSE statements and loops
- exception handling

You can use PL/SQL blocks in your OCI program to perform the following operations:

- call Oracle 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
- operate on objects in an Oracle8 server

Note: While the OCI can only directly process anonymous blocks, and not named packages or procedures, the user can always put the package or procedure call within an anonymous block and process that block.

Warning: When writing PL/SQL code, it is important to keep in mind that the parser treats everything that starts with "--" to a carriage return as a comment.

So if comments are indicated on each line by "--", the C compiler can concatenate all lines in a PL/SQL block into a single line without putting a carriage return "/n" for each line. In this particular case, the parser fails to extract the PL/SQL code of a line if the previous line ends with a comment. To avoid the problem, the programmer should put "/n" after each "--" comment to make sure the comment ends there.

See the *PL/SQL User's Guide and Reference* for information about coding PL/SQL blocks.

Datatypes

This chapter provides a reference to Oracle external datatypes used by OCI applications. It also provides a general discussion of Oracle datatypes, including special datatypes new to Release 8.0. The information in this chapter is useful for understanding the conversions between internal and external representations that occur when you transfer data between your program and Oracle.

For detailed information about Oracle internal datatypes, see the *Oracle8 SQL Reference*.

This chapter contains the following sections:

- Oracle Datatypes
- Internal Datatypes
- External Datatypes
- New OCI 8.0 External Datatypes
- Data Conversions
- Typecodes
- Definitions in `oratypes.h`

Oracle Datatypes

One of the main functions of an OCI program is to communicate with a database through an Oracle server. The OCI application may retrieve data from database tables through SQL SELECT queries, or it may modify existing data in tables through INSERTs, UPDATEs, or DELETEs.

Inside a database, values are stored in columns in tables. Internally, Oracle represents data in particular formats known as *internal datatypes*. Examples of internal datatypes include NUMBER, CHAR, and DATE.

In general, OCI applications do not work with internal datatype representations of data. OCI applications work with host language datatypes which are predefined by the language in which they are written. When data is transferred between an OCI client application and a database table, the OCI libraries convert the data between internal datatypes and *external datatypes*.

External datatypes are host language types that have been defined in the OCI header files. When an OCI application binds input variables, one of the bind parameters is an indication of the external datatype code (or *SQLT code*) of the variable. Similarly, when output variables are specified in a define call, the external representation of the retrieved data must be specified.

In some cases, external datatypes are similar to internal types. External types provide a convenience for the programmer by making it possible to work with host language types instead of proprietary data formats.

Note: Even though some external types are similar to internal types, an OCI application never binds to internal datatypes. They are discussed here because it can be useful to understand how internal types can map to external types.

The OCI is capable of performing a wide range of datatype conversions when transferring data between Oracle and an OCI application. There are more OCI external datatypes than Oracle internal datatypes. In some cases a single external type maps to an internal type; in other cases multiple external types map to an single internal type.

The many-to-one mappings for some datatypes provide flexibility for the OCI programmer. For example, if you are processing the SQL statement

```
SELECT sal FROM emp WHERE empno = :employee_number
```

and you want the salary to come back as character data, rather than in a binary floating-point format, specify an Oracle external string datatype, such as VARCHAR2 (code = 1) or CHAR (code = 96) for the *dtype* parameter in the

OCIDefineByPos() call for the `sal` column. You also need to declare a string variable in your program and specify its address in the *valuep* parameter.

If you want the salary information to be returned as a binary floating-point value, however, specify the `FLOAT` (code = 4) external datatype. You also need to define a variable of the appropriate type for the *valuep* parameter.

Oracle performs most data conversions transparently. The ability to specify almost any external datatype provides a lot of power for performing specialized tasks. For example, you can input and output `DATE` values in pure binary format, with no character conversion involved, by using the `DATE` external datatype (code = 12). See the description of the `DATE` external datatype on page 3 - 14 for more information.

To control data conversion, you must use the appropriate external datatype codes in the bind and define routines. You must tell Oracle where the input or output variables are in your OCI program and their datatypes and lengths.

The Oracle8 OCI also supports an additional set of OCI typecodes which are used by Oracle8's type management system to represent datatypes of object type attributes. There is a set of predefined constants which can be used to represent these typecodes. The constants each contain the prefix "`OCI_TYPECODE`".

In summary, the OCI programmer must be aware of the following different datatypes or data representations:

- Internal Oracle datatypes, which are used by table columns in an Oracle database. These also include datatypes used by PL/SQL which are not used by Oracle columns (e.g., indexed table, boolean, record). For more information, see "Internal Datatypes" on page 3-5 and "Internal Datatype Codes" on page 3-4.
- External OCI datatypes, which are used to specify host language representations of Oracle data. For more information, see "External Datatypes" on page 3-7, and "External Datatype Codes" on page 3-4.
- `OCI_TYPECODE` values, which are used to Oracle to represent type information for object type attributes. For more information, see "Typecodes" on page 3-24, and "Relationship Between SQLT and `OCI_TYPECODE` Values" on page 3-25.

Internal Datatype Codes

In some circumstances, an OCI application needs to know the internal representation of Oracle data. For example, you may need to know the datatype of a column in a dynamic SQL query so that you can define output variables to receive the fetched data. After executing the query, you can use a combination of the *OCIParamGet()* and *OCIAttrGet()* functions to obtain describe information about select-list items from the statement handle. You can get the same information from a describe handle without executing the statement by calling *OCIDescribeAny()*, and then the combination of *OCIParamGet()* and *OCIAttrGet()*.

Information about a column's internal datatype is conveyed to your application in the form of an internal datatype code. Once your application knows what type of data will be returned, it can make appropriate decisions about how to convert and format the output data. The Oracle internal datatype codes are listed in the section "Internal Datatypes" on page 3-5.

See Also: For detailed information about Oracle internal datatypes, see the *Oracle8 SQL Reference*. For information about describing select-list items in a query, see the section "Describing Select-List Items" on page 4-8.

External Datatype Codes

An external datatype code indicates to Oracle how a host variable represents data in your program. This determines how the data is converted when returned to output variables in your program, or how it is converted from input (bind) variables to Oracle column values. For example, if you want to convert a NUMBER in an Oracle column to a variable-length character array, you specify the VARCHAR2 external datatype code in the *OCIDefineByPos()* call that defines the output variable.

To convert a bind variable to a value in an Oracle column, specify the external datatype code that corresponds to the type of the bind variable. For example, if you want to input a character string such as '02-FEB-65' to a DATE column, specify the datatype as a character string and set the length parameter to nine.

It is always the programmer's responsibility to make sure that values are convertible. If you try to INSERT the string 'MY BIRTHDAY' into a DATE column, you will get an error when you execute the statement.

For a complete list of the external datatypes and datatype codes, see Table 3-2 on page 3-7.

Internal Datatypes

The following table lists the Oracle internal datatypes, along with each type's maximum internal length and datatype code.

Table 3–1 Internal Oracle Datatypes

Internal Oracle Datatype	Maximum Internal Length	Datatype Code
VARCHAR2	4000 bytes	1
NUMBER	21 bytes	2
LONG	2 ³¹ -1 bytes	8
ROWID	10 bytes	11
DATE	7 bytes	12
RAW	2000 bytes	23
LONG RAW	2 ³¹ -1 bytes	24
CHAR	2000 bytes	96
MLSLABEL	255 bytes	105
User-defined type (object type, VARRAY, Nested Table)	<N/A>	108
REF	<N/A>	111
CLOB	<N/A>	112
BLOB	<N/A>	113

For more information about any of these internal datatypes, see the *Oracle8 SQL Reference*. The following sections provide OCI-specific information about these datatypes.

LONG, RAW, LONG RAW, VARCHAR2

You can use the piecewise capabilities provided by *OCIBindByName()*, *OCIBindByPos()*, *OCIDefineByPos()*, *OCISstmtGetPieceInfo()* and *OCISstmtSetPieceInfo()* to perform inserts, updates or fetches involving column data of these types.

Character Strings and Byte Arrays

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

Note: LOBs and FILEs may also contain characters or binary data. They are handled differently than other types, so they are not included in this discussion. See the section “LOB and FILE Operations” on page 7-24 for more information about these data types.

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 bit-mapped graphics image. Character data can be transformed when passed through a gateway between networks. For example, character data passed between machines 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.

It is the responsibility of the database designer to choose the appropriate Oracle internal datatype for each column in the table. The OCI programmer must be aware of the many possible ways that character and byte-array data can be represented and converted between variables in the OCI program and Oracle tables.

When an array holds characters, the length parameter for the array in an OCI call is always passed in and returned in bytes, not characters.

External Datatypes

Table 3–2 lists datatype codes for external datatypes. For each datatype, the table lists the program variable types for C from or to which Oracle internal data is normally converted.

Table 3–2 External Datatypes and Codes

EXTERNAL DATATYPE			
NAME	CODE	TYPE OF PROGRAM VARIABLE	OCI DEFINED CONSTANT
VARCHAR2	1	char[n]	SQLT_CHR
NUMBER	2	unsigned char[21]	SQLT_NUM
8-bit signed INTEGER	3	signed char	SQLT_INT
16-bit signed INTEGER	3	signed short, signed int	SQLT_INT
32-bit signed INTEGER	3	signed int, signed long	SQLT_INT
FLOAT	4	float, double	SQLT_FLT
Null-terminated STRING	5	char[n+1]	SQLT_STR
VARNUM	6	char[22]	SQLT_VNU
LONG	8	char[n]	SQLT_LNG
VARCHAR	9	char[n+sizeof(short integer)]	SQLT_VCS
ROWID	11	char[n]	SQLT_RID (see note 1)
DATE	12	char[7]	SQLT_DAT
VARRAW	15	unsigned char[n+sizeof(short integer)]	SQLT_VBI
RAW	23	unsigned char[n]	SQLT_BIN
LONG RAW	24	unsigned char[n]	SQLT_LBI
UNSIGNED INT	68	unsigned	SQLT_UIN
LONG VARCHAR	94	char[n+sizeof(integer)]	SQLT_LVC
LONG VARRAW	95	unsigned char[n+sizeof(integer)]	SQLT_LVB
CHAR	96	char[n]	SQLT_AFC
CHARZ	97	char[n+1]	SQLT_AVC
ROWID descriptor	104	OCIRowid	SQLT_RDD
MLSLABEL	106	char[n]	SQLT_LAB

Table 3–2 External Datatypes and Codes (Cont.)

EXTERNAL DATATYPE			
NAME	CODE	TYPE OF PROGRAM VARIABLE	OCI DEFINED CONSTANT
NAMED DATA TYPE	108	struct	SQLT_NTY
REF	110	OCIRef	SQLT_REF
Character LOB	112	OCILobLocator (see note 3)	SQLT_CLOB
Binary LOB	113	OCILobLocator (see note 3)	SQLT_BLOB
Binary FILE	114	OCILobLocator	SQLT_FILE
OCI string type	155	OCIString	SQLT_VST (see note 2)
OCI date type	156	OCIDate	SQLT_ODT (see note 2)

Notes:

(1) This type is valid only for version 7.x OCI calls. Oracle8 OCI applications should use the ROWID descriptor (type 104).

(2) For more information on the use of these datatypes, refer to Chapter 9, “Object-Relational Datatypes”.

(3) In applications using datatype mappings generated by OTT, CLOBs may be mapped as OCIClobLocator, and BLOBs may be mapped as OCIBlobLocator. For more information, refer to Chapter 12, “Using the Object Type Translator”.

Note: Where the length is shown as *n*, it is a variable, and depends on the requirements of the program (or of the operating system in the case of ROWID).

Each of the external datatypes is described below. Datatypes that are new as of release 8.0 are described in the section “New OCI 8.0 External Datatypes” on page 3-18.

The following three types are internal to PL/SQL and cannot be returned as values by OCI:

- Boolean, SQLT_BOOL
- Indexed Table, SQLT_TAB
- Record, SQLT_REC

VARCHAR2

The VARCHAR2 datatype is a variable-length string of characters with a maximum length of 4000 bytes.

Note: If you are using Oracle8 objects, you can work with a special **OCIStr**ing external datatype using a set of predefined OCI functions. Refer to Chapter 9, “Object-Relational Datatypes” for more information about this datatype.

Input

The *value_sz* parameter determines the length in the *OCIBindByName()* or *OCIBindByPos()* call.

If the *value_sz* parameter is greater than zero, Oracle obtains the bind variable value by reading exactly that many bytes, starting at the buffer address in your program. Trailing blanks are stripped, and the resulting value is used in the SQL statement or PL/SQL block. If, in the case of an INSERT statement, the resulting value is longer than the defined length of the database column, the INSERT fails, and an error is returned.

Note: A trailing null is not stripped. Variables should be blank-padded but not null-terminated.

If the *value_sz* parameter is zero, Oracle treats the bind variable as a null, regardless of its actual content. Of course, a null must be allowed for the bind variable value in the SQL statement. If you try to insert a null into a column that has a NOT NULL integrity constraint, Oracle issues an error, and the row is not inserted.

When the Oracle internal (column) datatype is NUMBER, input from a character string that contains the character representation of a number is legal. Input character strings are converted to internal numeric format. If the VARCHAR2 string contains an illegal conversion character, Oracle returns an error and the value is not inserted into the database.

Output

Specify the desired length for the return value in the *value_sz* parameter of the *OCIDefineByPos()* call, or the *value_sz* parameter of *OCIBindByName()* or *OCIBindByPos()* for PL/SQL blocks. If zero is specified for the length, no data is returned.

If you omit the *rlenp* parameter of *OCIDefineByPos()*, returned values are blank-padded to the buffer length, and nulls are returned as a string of blank characters. If *rlenp* is included, returned values are not blank-padded. Instead, their actual lengths are returned in the *rlenp* parameter.

To check if a null is returned or if character truncation has occurred, include an indicator parameter in the *OCIDefineByPos()* call. Oracle sets the indicator parameter to -1 when a null is fetched and to the original column length when the returned value is truncated. Otherwise, it is set to zero. If you do not specify an indicator parameter and a null is selected, the fetch call returns the error code OCI_SUCCESS_WITH_INFO. Retrieving diagnostic information on the error will return ORA-1405.

See Also: For more information about indicator variables, see the section “Indicator Variables” on page 2-29.

You can also request output to a character string from an internal NUMBER datatype. Number conversion follows the conventions established by National Language Support for your system. For example, your system might be configured to recognize a comma rather than period as the decimal point.

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.

Note: If you are using objects in Oracle8, you can work with a special **OCINumber** datatype using a set of predefined OCI functions. Refer to Chapter 9, “Object-Relational Datatypes” for more information about this datatype.

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. The lower 7 bits represent the exponent, which is a base-100 digit with an offset of 65.

Each mantissa byte is a base-100 digit, in the range 1..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 2 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 Oracle NUMBER.

If you specify the datatype code 2 in the *dt*y parameter of an *OCIDefineByPos()* call, your program receives numeric data in this Oracle internal format. The output variable should be a 21-byte array to accommodate the largest possible number. Note that only the bytes that represent the number are returned. There is no blank padding or null termination. If you need to know the number of bytes returned, use the VARNUM external datatype instead of NUMBER. See the description of VARNUM on page 3-13 for examples of the Oracle internal number format.

INTEGER

The INTEGER datatype converts numbers. An external integer is a signed binary number; the size in bytes is system dependent. The host system architecture determines the order of the bytes in the variable. A length specification is required for input and output. If the number being returned from Oracle is not an integer, the fractional part is discarded, and no error or other indication is returned. If the number to be returned exceeds the capacity of a signed integer for the system, Oracle returns an "overflow on conversion" error.

FLOAT

The FLOAT datatype processes numbers that have fractional parts or that exceed the capacity of an integer. The number is represented in the host system's floating-point format. Normally the length is either four or eight bytes. The length specification is required for both input and output.

The internal format of an Oracle number is decimal, and most floating-point implementations are binary; therefore Oracle can represent numbers with greater precision than floating-point representations.

Note: You may receive a round-off error when converting between FLOAT and NUMBER. Thus, using a FLOAT as a bind variable in a query may return an ORA-1403 error. You can avoid this situation by converting the FLOAT into a STRING and then using datatype code 1 or 5 for the operation.

STRING

The null-terminated STRING format behaves like the VARCHAR2 format (datatype code 1), except that the string must contain a null terminator character. This datatype is most useful for C programs.

Input

The string length supplied in the *OCIBindByName()* or *OCIBindByPos()* call limits the scan for the null terminator. If the null terminator is not found within the length specified, Oracle issues the error

ORA-01480: trailing null missing from STR bind value

If the length is not specified in the bind call, the OCI uses an implied maximum string length of 4000.

The minimum string length is two bytes. If the first character is a null terminator and the length is specified as two, a null is inserted in the column, if permitted. Unlike types 1 and 96, a string containing all blanks is not treated as a null on input; it is inserted as is.

Note: Unlike earlier versions of the OCI, in release 8.0 you cannot pass -1 for the string length parameter of a null-terminated string.

Output

A null terminator is placed after the last character returned. If the string exceeds the field length specified, it is truncated and the last character position of the output variable contains the null terminator.

A null select-list item returns a null terminator character in the first character position. An ORA-01405 error is possible, as well.

VARNUM

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

Table 3 - 3 shows several examples of the VARNUM values returned for numbers in an Oracle table.

Table 3–3 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

LONG

The LONG datatype stores character strings longer than 4000 bytes. You can store up to two gigabytes ($2^{31}-1$ bytes) in a LONG column. Columns of this type are used only for storage and retrieval of long strings. They cannot be used in functions, expressions, or WHERE clauses. LONG column values are generally converted to and from character strings.

VARCHAR

The VARCHAR datatype stores character strings of varying length. The first two bytes contain the length of the character string, and the remaining bytes contain the string. The specified length of the string in a bind or a define call must include the two length bytes, so the largest VARCHAR string that can be received or sent is 65533 bytes long, not 65535. For converting longer strings, use the LONG VARCHAR external datatype.

ROWID

The ROWID datatype identifies a particular row in a database table. ROWID can be a select-list item in a query; for example:

```
SELECT rowid, ename, sal FROM emp FOR UPDATE OF sal
```

In this case, you use the returned ROWID in further INSERT, UPDATE, or DELETE statements. This can be the fastest way to access a particular row.

In the Oracle8 OCI, you access ROWIDs through the use of a ROWID descriptor, which you can use as a bind or define variable. See the sections “Descriptors and Locators” on page 2-12 and “Positioned Updates and Deletes” on page 2-31 for more information about the use of the ROWID descriptor.

DATE

The DATE datatype can update, insert, or retrieve a date value using the Oracle internal date binary format. A date in binary format contains seven bytes, as shown in Table 3-4.

Table 3-4 Format of the DATE Datatype

Byte	1	2	3	4	5	6	7
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example (for 30-NOV-1992, 3:17 PM)	119	192	11	30	16	18	1

The century and year bytes are in an excess-100 notation. Dates Before Common Era (BCE) are less than 100. The era begins on 01-JAN-4712 BCE, which is Julian day 1. For this date, the century byte is 53, and the year byte is 88. The hour, minute, and second bytes are in excess-1 notation. The hour byte ranges from 1 to 24, the minute and second bytes from 1 to 60. If no time was specified when the date was created, the time defaults to midnight (1, 1, 1).

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

Note: There is little need to use the Oracle external DATE datatype in ordinary database operations. It is much more convenient to convert DATEs into character format, because the program usually deals with data in a character format, such as ‘DD-MON-YY’.

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

Note: If you are using objects in Oracle8, you can work with a special **OCIDate** datatype using a set of predefined OCI functions. Refer to Chapter 9, “Object-Relational Datatypes” for more information about this datatype.

RAW

The RAW datatype is used for binary data or byte strings that are not to be interpreted by Oracle, for example, to store graphics character sequences. The maximum length of a RAW column is 2000 bytes. For more information, see the *Oracle8 SQL Reference*.

When RAW data in an Oracle table is converted to a character string in a program, the data is represented in hexadecimal character code. Each byte of the RAW data is returned as two characters that indicate the value of the byte, from '00' to 'FF'. If you want to input a character string in your program to a RAW column in an Oracle table, you must code the data in the character string using this hexadecimal code.

You can use the piecewise capabilities provided by *OCIDefineByPos()*, *OCIBindByName()*, *OCIBindByPos()*, *OCIStmtGetPieceInfo()*, and *OCIStmtSetPieceInfo()* to perform inserts, updates, or fetches involving RAW (or LONG RAW) columns.

Note: If you are using objects in Oracle8, you can work with a special **OCIRaw** datatype using a set of predefined OCI functions. Refer to Chapter 9, “Object-Relational Datatypes” for more information about this datatype.

VARRAW

The VARRAW datatype is similar to the RAW datatype. However, 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 long, not 65535. For converting longer strings, use the LONG VARRAW external datatype.

LONG RAW

The LONG RAW datatype is similar to the RAW datatype, except that it stores raw data with a length up to two gigabytes ($2^{31}-1$ bytes).

UNSIGNED

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

LONG VARCHAR

The LONG VARCHAR datatype stores data from and into an Oracle LONG column. The first four bytes of a LONG VARCHAR contain the length of the item. So, the maximum length of a stored item is $2^{31}-5$ bytes.

LONG VARRAW

The LONG VARRAW datatype is used to store data from and into an Oracle LONG RAW column. The length is contained in the first four bytes. The maximum length is $2^{31}-5$ bytes.

CHAR

The CHAR datatype is a string of characters, with a maximum length of 2000. CHAR strings are compared using blank-padded comparison semantics (see the *Oracle8 SQL Reference*).

Input

The length is determined by the *value_sz* parameter in the *OCIBindByName()* or *OCIBindByPos()* call.

Note: The entire contents of the buffer (*value_sz* chars) is passed to the database, including any trailing blanks or nulls.

If the *value_sz* parameter is zero, Oracle treats the bind variable as a null, regardless of its actual content. Of course, a null must be allowed for the bind variable value in the SQL statement. If you try to insert a null into a column that has a NOT NULL integrity constraint, Oracle issues an error and does not insert the row.

Negative values for the *value_sz* parameter are not allowed for CHARs.

When the Oracle internal (column) datatype is NUMBER, input from a character string that contains the character representation of a number is legal. Input

character strings are converted to internal numeric format. If the CHAR string contains an illegal conversion character, Oracle returns an error and does not insert the value. Number conversion follows the conventions established by National Language Support settings for your system. For example, your system might be configured to recognize a comma (,) rather than a period (.) as the decimal point.

Output

Specify the desired length for the return value in the *value_sz* parameter of the *OCIDefineByPos()* call. If zero is specified for the length, no data is returned.

If you omit the *rlenp* parameter of *OCIDefineByPos()*, returned values are blank padded to the buffer length, and nulls are returned as a string of blank characters. If *rlenp* is included, returned values are not blank padded. Instead, their actual lengths are returned in the *rlenp* parameter.

To check whether a null is returned or if character truncation has occurred, include an indicator parameter or array of indicator parameters in the *OCIDefineByPos()* call. An indicator parameter is set to -1 when a null is fetched and to the original column length when the returned value is truncated. Otherwise, it is set to zero. If you do not specify an indicator parameter and a null is selected, the fetch call returns an ORA-01405 error.

See Also: For more information about indicator variables, see “Indicator Variables” on page 2-29

You can also request output to a character string from an internal NUMBER datatype. Number conversion follows the conventions established by the National Language Support settings for your system. For example, your system might use a comma (,) rather than a period (.) as the decimal point.

CHARZ

The CHARZ external datatype 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.

On input, the length parameter must indicate the exact length, including the null terminator. For example, if an array in C is declared as

```
char my_num[] = "123.45";
```

then the length parameter when you bind `my_num` must be seven. Any other value would return an error for this example.

MLSLABEL

Trusted Oracle provides the MLSLABEL datatype, which stores Trusted Oracle's internal representation of labels generated by multilevel secure operating systems. Trusted Oracle uses labels to control database access.

You can define a column using the MLSLABEL datatype in Oracle8 for compatibility with Trusted Oracle applications, but the only valid value for the column in Oracle8 is NULL.

See the *Trusted Oracle Server Administrator's Guide* for more information about the MLSLABEL datatype and Trusted Oracle.

New OCI 8.0 External Datatypes

The following new external datatypes are being introduced with release 8.0. These datatypes are not supported when connect to an Oracle7 server.

Note: Both internal and external datatypes have Oracle-defined constant values (e.g., `SQLT_NTY`, `SQLT_REF`) corresponding to their datatype codes. Although the constants are not listed for all of the types in this chapter, they are used in this section when discussing new Oracle8 datatypes. The datatype constants are also used in other chapters of this guide when referring to these new types.

Note: Named data types and REFs are only available if you have purchased the Oracle8 Enterprise Edition.

NAMED DATA TYPE

Named data types are user-defined types which are specified with the `CREATE TYPE` command in SQL. Examples include object types, varrays, and nested tables. In the OCI, "named data type" refers to a host language representation of the type. The `SQLT_NTY` datatype code is used when binding or defining named data types.

In a C application, named data types are represented as C structs. These structs can be generated from types stored in the database by using the Object Type Translator. These types correspond to `OCI_TYPECODE_OBJECT`.

See Also: For more information about working with named data types in the OCI, refer to Part 2 of this guide.

For information about how named data types are represented as C structs, refer to Chapter 12, "Using the Object Type Translator".

REF

This is a reference to a named data type. The C language representation of a REF is a variable declared to be of type **OCIRef ***. The `SQLT_REF` datatype code is used when binding or defining REFs.

Access to REFs is only possible when an OCI application has been initialized in object mode. When REFs are retrieved from the server, they are stored in the client-side object cache.

To allocate a REF for use in your application, you should declare a variable to be a pointer to a REF, and then call *OCIObjectNew()*, passing `OCI_TYPECODE_REF` as the *typecode* parameter.

See Also: For more information about working with REFs in the OCI, refer to Part 2 of this guide.

LOB

A LOB (Large Object) stores binary or character data up to 4 gigabytes in length. Binary data is stored in a BLOB (Binary LOB), and character data is stored in a CLOB (Character LOB) or NCLOB (National Character LOB).

LOB values may or may not be stored inline with other row data in the database. In either case, LOBs have the full transactional support of the database server. A database table stores a *LOB locator* which points to the LOB value which may be in a different storage space.

When an OCI application issues a SQL query which includes a LOB column or attribute in its select-list, fetching the result(s) of the query returns the locator, rather than the actual LOB value. In the OCI, the LOB locator maps to a variable of type **OCILobLocator**.

See Also: For more information about descriptors, including the LOB locator, see the section “Descriptors and Locators” on page 2-12.

For more information about LOBs refer to the *Oracle8 SQL Reference* and the *Oracle8 Application Developer's Guide*.

The OCI functions for LOBs take a LOB locator as one of their arguments. The OCI functions assume that the locator has already been created, whether or not the LOB to which it points contains data.

Bind and define operations are performed on the LOB locator, which is allocated with the *OCIDescriptorAlloc()* function.

The locator is always fetched first using SQL or *OCIObjectPin()*, and then operations are performed using the locator. The OCI functions never take the actual LOB value as a parameter.

See Also: For more information about OCI LOB functions, see the section “LOB and FILE Operations” on page 7-24.

The datatype codes available for binding or defining LOBs are:

- `SQLT_BLOB` - a binary LOB data type.
- `SQLT_CLOB` - a character LOB data type.

The NCLOB is a special type of CLOB with the following requirements:

- To write into or read from an NCLOB, the user must set the character set form (*csfrm*) parameter to be `SQLCS_NCHAR`.
- The “amount” (*amtp*) parameter in calls involving CLOBs and NCLOBs is always interpreted in terms of characters, rather than bytes.

FILE

The FILE datatype provides access to file LOBs that are stored in file systems outside the Oracle8 database. Oracle8 currently supports access to binary files, or BFILEs.

A BFILE column or attribute stores a file LOB locator, which serves as a pointer to a binary file on the server’s file system. The locator maintains the directory alias and the filename.

Binary file LOBs do not participate in transactions. Rather, the underlying operating system provides file integrity and durability. The maximum file size supported is 4 gigabytes.

The database administrator must ensure that the file exists and that Oracle8 processes have operating system read permissions on the file.

The BFILE datatype allows read-only support of large binary files; you cannot modify a file through Oracle. Oracle8 provides APIs to access file data. The primary interfaces that you use to access file data are the PL/SQL DBMS_LOB package, and the OCI.

The datatype code available for binding or defining FILEs is:

- `SQLT_BFILE` - a binary FILE LOB data type (see the next section)

For more information about directory aliases, refer to the *Oracle8 Application Developer’s Guide*.

BLOB

The BLOB datatype stores unstructured binary large objects. BLOBs can be thought of as bitstreams with no character set semantics. BLOBs can store up to four gigabytes of binary data.

BLOBs have full transactional support; changes made through the PL/SQL DBMS_LOB package, or the OCI participate fully in the transaction. The BLOB value manipulations can be committed or rolled back. You cannot save a BLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

CLOB

The CLOB datatype stores single-byte character data. Varying-width character sets are not supported. CLOBs can store up to 4 gigabytes of character data.

CLOBs have full transactional support; changes made through the PL/SQL DBMS_LOB package or the OCI participate fully in the transaction. The CLOB value manipulations can be committed or rolled back. You cannot save a CLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

NCLOB An NCLOB is a national character version of a CLOB. It stores fixed-width, single- or multi-byte national character set character (NCHAR) data. Varying-width character sets are not supported. NCLOBs can store up to 4 gigabytes of character text data.

NCLOBs have full transactional support; changes made through the PL/SQL DBMS_LOB package, or the OCI participate fully in the transaction. NCLOB value manipulations can be committed or rolled back. You cannot save a NCLOB locator in a PL/SQL or OCI 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.

New C Datatype Mappings

The OCI now includes support for Oracle-defined C datatypes used to map user-defined datatypes and ADT attributes to C representations (e.g. **OCINumber**, **OCIArray**). The OCI provides a set of calls to operate on these datatypes, and to use these datatypes in bind and define operations, in conjunction with OCI external datatype codes. For information on using these Oracle-defined C datatypes, refer to Chapter 9, “Object-Relational Datatypes”.

Data Conversions

Table 3–5 shows the supported conversions from internal Oracle datatypes to external datatypes, and from external datatypes into internal column representations, for all datatypes available through release 7.3. Information about data conversions for data types new to release 8.0 is listed here:

- REFs stored in the database are converted to `SQLT_REF` on output.
- `SQLT_REF` is converted to the internal representation of REFs on input.
- Named Data Types stored in the database can be converted to `SQLT_NTY` (and represented by a C struct in the application) on output.
- `SQLT_NTY` (represented by a C struct in an application) is converted to the internal representation of the corresponding type on input.
- LOBs and BFILES are represented by descriptors in OCI applications, so there are no input or output conversions.
- For information about **OCString**, **OCINumber**, and other new Oracle8 datatypes, refer to Chapter 9, “Object-Relational Datatypes”, and Chapter 10, “Binding and Defining in Object Applications”.

Table 3–5 Data Conversions

EXTERNAL DATATYPES	INTERNAL DATATYPES								
	1 VARCHAR2	2 NUMBER	8 LONG	11 ROWID	12 DATE	23 RAW	24 LONG RAW	96 CHAR	105 MLSLABEL
1 VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I/O(3)		I/O(7)
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	I/O(7)
6 VARNUM	I/O(4)	I/O	I					I/O(4)	
7 DECIMAL	I/O(4)	I/O	I					I/O(4)	
8 LONG	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I/O(3, 5)	I/O	I/O(7)
9 VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I/O(3, 5)	I/O	I/O(7)
11 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)	

Table 3–5 Data Conversions (Cont.)

EXTERNAL DATATYPES	INTERNAL DATATYPES								
	1 VARCHAR2	2 NUMBER	8 LONG	11 ROWID	12 DATE	23 RAW	24 LONG RAW	96 CHAR	105 MLSLABEL
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	I/O(7)
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)	I/O	I/O(7)
97 CHARZ	I/O	I/O	I/O	I/O(1)	I/O(2)	I/O(3)	I(3)	I/O	I/O(7)
104 ROWID DESC.									
106 MLSLABEL									I/O(8)

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 hex format.
On output, column value is returned in hex format.
- (4) For output, column value must represent a valid number.
- (5) Length must be less than or equal to 2000.
- (6) On input, column value is stored in hex format.
On output, column value must be in hex format.
- (7) For input, host string must be a valid OS label in text format.
On output, column value is returned in OS label text format.
- (8) For character representation of MLSLABEL, use the TO_CHAR(mlscolumn) function.

Legend:

- I = Conversion valid for input only
- O = Conversion valid for output only
- I/O = Conversion valid for input or output

Typecodes

There is a unique typecode associated with each Oracle8 type, whether scalar, collection, reference, or object type. This typecode identifies the type, and is used by Oracle to manage information about object type attributes. This typecode system is designed to be generic and extensible, and is not tied to a direct one-to-one mapping to Oracle datatypes. Consider the following SQL statements:

```
CREATE TYPE my_type AS OBJECT
( attr1    NUMBER,
  attr2    INTEGER,
  attr3    SMALLINT)
CREATE TABLE my_table AS TABLE OF my_type;
```

These statements create an object type and an object table. When it is created, `my_table` will have three columns, all of which are of Oracle `NUMBER` type, because `SMALLINT` and `INTEGER` map internally to `NUMBER`. The internal representation of the attributes of `my_type`, however, maintains the distinction between the datatypes of the three attributes: `attr1` is `OCI_TYPECODE_NUMBER`, `attr2` is `OCI_TYPECODE_INTEGER`, and `attr3` is `OCI_TYPECODE_SMALLINT`. If an application describes `my_type`, these typecodes are returned.

OCITypeCode is the C datatype of the typecode. The typecode is used by some OCI functions, like `OCIObjectNew()` (where it helps determine what type of object is created). It is also returned as the value of some attributes when an object is described; e.g., querying the `OCI_ATTR_TYPECODE` attribute of a type returns an **OCITypeCode** value.

Table 3–6 lists the possible values for an **OCITypeCode**. There is a value corresponding to each Oracle8 datatype.

Table 3–6 *OCITypeCode Values*

Value	Datatype
<code>OCI_TYPECODE_REF</code>	<code>REF</code>
<code>OCI_TYPECODE_DATE</code>	<code>date</code>
<code>OCI_TYPECODE_REAL</code>	<code>single-precision real</code>
<code>OCI_TYPECODE_DOUBLE</code>	<code>double-precision real</code>
<code>OCI_TYPECODE_FLOAT</code>	<code>floating-point</code>
<code>OCI_TYPECODE_NUMBER</code>	<code>Oracle number</code>

Table 3–6 OCITypeCode Values (Cont.)

Value	Datatype
OCI_TYPECODE_DECIMAL	decimal
OCI_TYPECODE_OCTET	octet
OCI_TYPECODE_INTEGER	integer
OCI_TYPECODE_SMALLINT	smallint
OCI_TYPECODE_RAW	RAW
OCI_TYPECODE_VARCHAR2	variable string ANSI SQL, i.e., VARCHAR2
OCI_TYPECODE_VARCHAR	variable string Oracle SQL, i.e., VARCHAR
OCI_TYPECODE_CHAR	fixed-length string inside SQL, i.e. SQL CHAR
OCI_TYPECODE_VARRAY	variable-length array (varray)
OCI_TYPECODE_TABLE	multiset
OCI_TYPECODE_CLOB	character large object (CLOB)
OCI_TYPECODE_BLOB	binary large object (BLOB)
OCI_TYPECODE_BFILE	binary large object file (BFILE)
OCI_TYPECODE_OBJECT	named object type
OCI_TYPECODE_NAMEDCOLLECTION	Domain (named primitive type)

Relationship Between SQLT and OCI_TYPECODE Values

Oracle recognizes two different sets of datatype code values. One set is distinguished by the “SQLT_” prefix, the other by the “OCI_TYPECODE_” prefix.

The SQLT typecodes are used by OCI to specify a datatype in a bind or define operation. In this way, the SQL typecodes help to control data conversions between Oracle and OCI client applications. The OCI_TYPECODE types are used by Oracle8’s type system to reference or describe predefined types when manipulating or creating user-defined types.

In many cases there are direct mappings between SQLT and OCI_TYPECODE values. In other cases, however, there is not a direct one-to-one mapping. For example OCI_TYPECODE_SIGNED16, OCI_TYPECODE_SIGNED32, OCI_TYPECODE_INTEGER, OCI_TYPECODE_OCTET, and OCI_TYPECODE_SMALLINT are all mapped to the SQLT_INT type.

The following table illustrates the mappings between SQLT and OCI_TYPECODE types.

Table 3–7 OCI_TYPECODE to SQLT Mappings

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
BFILE	OCI_TYPECODE_BFILE	SQLT_BFILE
BLOB	OCI_TYPECODE_BLOB	SQLT_BLOB
CHAR	OCI_TYPECODE_CHAR (n)	SQLT_AFC(n) [note 1]
CLOB	OCI_TYPECODE_CLOB	SQLT_CLOB
COLLECTION	OCI_TYPECODE_NAMEDCOLLECTION	SQLT_NCO
DATE	OCI_TYPECODE_DATE	SQLT_DAT
FLOAT	OCI_TYPECODE_FLOAT (b)	SQLT_FLT (8) [note 2]
DECIMAL	OCI_TYPECODE_DECIMAL (p)	SQLT_NUM (p, 0) [note 3]
DOUBLE	OCI_TYPECODE_DOUBLE	SQLT_FLT (8)
INTEGER	OCI_TYPECODE_INTEGER	SQLT_INT (i) [note 4]
NUMBER	OCI_TYPECODE_NUMBER (p, s)	SQLT_NUM (p, s) [note 5]
OCTECT	OCI_TYPECODE_OCTECT	SQLT_INT (1)
POINTER	OCI_TYPECODE_PTR	<NONE>
RAW	OCI_TYPECODE_RAW	SQLT_LVB
REAL	OCI_TYPECODE_REAL	SQLT_FLT (4)
REF	OCI_TYPECODE_REF	SQLT_REF
OBJECT	OCI_TYPECODE_OBJECT	SQLT_NTY
SIGNED(8)	OCI_TYPECODE_SIGNED8	SQLT_INT (1)
SIGNED(16)	OCI_TYPECODE_SIGNED16	SQLT_INT (2)
SIGNED(32)	OCI_TYPECODE_SIGNED32	SQLT_INT (4)
SMALLINT	OCI_TYPECODE_SMALLINT	SQLT_INT (i) [note 4]
TABLE [note 6]	OCI_TYPECODE_TABLE	SQLT_TAB
UNSIGNED(8)	OCI_TYPECODE_UNSIGNED8	SQLT_UIN (1)
UNSIGNED(16)	OCI_TYPECODE_UNSIGNED16	SQLT_UIN (2)
UNSIGNED(32)	OCI_TYPECODE_UNSIGNED32	SQLT_UIN (4)
VARRAY [note 6]	OCI_TYPECODE_VARRAY	SQLT_NAR

Table 3–7 OCI_TYPECODE to SQLT Mappings (Cont.)

Oracle Type System Typename	Oracle Type System Type	Equivalent SQLT Type
VARCHAR	OCI_TYPECODE_VARCHAR (n)	SQLT_CHR (n) [note 1]
VARCHAR2	OCI_TYPECODE_VARCHAR2 (n)	SQLT_VCS (n) [note 1]

Notes:

1. n is the size of the string in bytes
2. These are floating point numbers, the precision is given in terms of binary digits. b is the precision of the number in binary digits.
3. This is equivalent to a NUMBER with no decimal places.
4. i is the size of the number in bytes, set as part of an OCI call.
5. p is the precision of the number in decimal digits; s is the scale of the number in decimal digits.
6. Can only be part of a named collection type.

Definitions in *oratypes.h*

Throughout this guide you will see references to datatypes like **ub2** or **sb4**, or to constants like **UB4MAXVAL**. These types are defined in the *oratypes.h* header file, an example of which is included here. The exact contents may vary according to the platform you are using.

```
#ifndef ORASTDDEF
# include <stddef.h>
# define ORASTDDEF
#endif
```

```
#ifndef ORALIMITS
# include <limits.h>
# define ORALIMITS
#endif
```

```
#ifndef SX_ORACLE
#define SX_ORACLE
#define SX
#define ORATYPES
```

```
#ifndef TRUE
# define TRUE 1
# define FALSE 0
#endif
```

```
#ifdef lint
```

```
# ifndef mips
#   define signed
# endif
#endif

#ifdef ENCORE_88K
# ifndef signed
#   define signed
# endif
#endif

#if defined(SYSV_386) || defined(SUN_OS)
# ifdef signed
#   undef signed
# endif
# define signed
#endif

#ifndef lint
typedef          int eword;
typedef unsigned int uword;
typedef  signed int sword;
#else
#define eword int
#define uword unsigned int
#define sword signed int
#endif

#define EWORDMAXVAL ((eword) INT_MAX)
#define EWORDMINVAL ((eword)      0)
#define UWORDMAXVAL ((uword)UINT_MAX)
#define UWORDMINVAL ((uword)      0)
#define SWORDMAXVAL ((sword) INT_MAX)
#define SWORDMINVAL ((sword) INT_MIN)
#define MINEWORDMAXVAL ((eword) 32767)
#define MAXEWORDMINVAL ((eword)      0)
#define MINUWORDMAXVAL ((uword) 65535)
#define MAXUWORDMINVAL ((uword)      0)
#define MINSWORDMAXVAL ((sword) 32767)
#define MAXSWORDMINVAL ((sword) -32767)

#ifndef lint
# ifdef mips
```



```

typedef    signed char  ebl;
# else
typedef          char  ebl;
# endif
typedef unsigned char  ubl;
typedef    signed char  sbl;
#else
#define ebl char
#define ubl unsigned char
#define sbl signed char
#endif

#define EB1MAXVAL ((ebl)SCHAR_MAX)
#define EB1MINVAL ((ebl)      0)
#if defined(mips)
# ifndef lint
#  define UB1MAXVAL (UCHAR_MAX)
# endif
#endif
#ifndef UB1MAXVAL
# ifdef SCO_UNIX
#  define UB1MAXVAL (UCHAR_MAX)
# else
#  define UB1MAXVAL ((ubl)UCHAR_MAX)
# endif
#endif
#define UB1MINVAL ((ubl)      0)
#define SB1MAXVAL ((sbl)SCHAR_MAX)
#define SB1MINVAL ((sbl)SCHAR_MIN)
#define MINEB1MAXVAL ((ebl) 127)
#define MAXEB1MINVAL ((ebl)   0)
#define MINUB1MAXVAL ((ubl) 255)
#define MAXUB1MINVAL ((ubl)   0)
#define MINSB1MAXVAL ((sbl) 127)
#define MAXSB1MINVAL ((sbl) -127)

#define UB1BITS          CHAR_BIT
#define UB1MASK          ((1 << ((uword)CHAR_BIT)) - 1)

typedef unsigned char OraText;

#ifndef LUSEMFC
# define text OraText
#endif

```

```
#ifndef lint
typedef          short    eb2;
typedef unsigned short    ub2;
typedef  signed short    sb2;
#else
#define eb2  short
#define ub2  unsigned short
#define sb2  signed short
#endif

#define EB2MAXVAL ((eb2) SHRT_MAX)
#define EB2MINVAL ((eb2)      0)
#define UB2MAXVAL ((ub2) USHRT_MAX)
#define UB2MINVAL ((ub2)      0)
#define SB2MAXVAL ((sb2) SHRT_MAX)
#define SB2MINVAL ((sb2) SHRT_MIN)
#define MINEB2MAXVAL ((eb2) 32767)
#define MAXEB2MINVAL ((eb2)      0)
#define MINUB2MAXVAL ((ub2) 65535)
#define MAXUB2MINVAL ((ub2)      0)
#define MINSB2MAXVAL ((sb2) 32767)
#define MAXSB2MINVAL ((sb2)-32767)

#if defined(A_OSF)

#ifndef lint
typedef          int    eb4;
typedef unsigned int    ub4;
typedef  signed int    sb4;
#else
#define eb4  int
#define ub4  unsigned int
#define sb4  signed int
#endif

#define EB4MAXVAL ((eb4) INT_MAX)
#define EB4MINVAL ((eb4)      0)
#define UB4MAXVAL ((ub4) UINT_MAX)
#define UB4MINVAL ((ub4)      0)
#define SB4MAXVAL ((sb4) INT_MAX)
#define SB4MINVAL ((sb4) INT_MIN)
#define MINEB4MAXVAL ((eb4) 2147483647)
#define MAXEB4MINVAL ((eb4)      0)
```

```
#define MINUB4MAXVAL ((ub4) 4294967295)
#define MAXUB4MINVAL ((ub4) 0)
#define MINSB4MAXVAL ((sb4) 2147483647)
#define MAXSB4MINVAL ((sb4)-2147483647)

#else

#ifndef lint
typedef      long  eb4;
typedef unsigned long  ub4;
typedef  signed long  sb4;
#else
#define eb4 long
#define ub4 unsigned long
#define sb4 signed long
#endif

#define EB4MAXVAL ((eb4) LONG_MAX)
#define EB4MINVAL ((eb4) 0)
#define UB4MAXVAL ((ub4) ULONG_MAX)
#define UB4MINVAL ((ub4) 0)
#define SB4MAXVAL ((sb4) LONG_MAX)
#define SB4MINVAL ((sb4) LONG_MIN)
#define MINEB4MAXVAL ((eb4) 2147483647)
#define MAXEB4MINVAL ((eb4) 0)
#define MINUB4MAXVAL ((ub4) 4294967295)
#define MAXUB4MINVAL ((ub4) 0)
#define MINSB4MAXVAL ((sb4) 2147483647)
#define MAXSB4MINVAL ((sb4)-2147483647)
#endif

#ifndef lint
typedef unsigned long  ubig_ora;
typedef  signed long  sbig_ora;
#else
#define ubig_ora unsigned long
#define sbig_ora signed long
#endif

#define UBIG_ORAMAXVAL ((ubig_ora) ULONG_MAX)
#define UBIG_ORAMINVAL ((ubig_ora) 0)
#define SBIG_ORAMAXVAL ((sbig_ora) LONG_MAX)
#define SBIG_ORAMINVAL ((sbig_ora) LONG_MIN)
```

```
#define MINUBIG_ORAMAXVAL ((ubig_ora) 4294967295)
#define MAXUBIG_ORAMINVAL ((ubig_ora) 0)
#define MINSBIG_ORAMAXVAL ((sbig_ora) 2147483647)
#define MAXSBIG_ORAMINVAL ((sbig_ora)-2147483647)

#define UBIGORABITS      (UB1BITS * sizeof(ubig_ora))

#define SLU8NATIVE
#define SLS8NATIVE

#ifdef SLU8NATIVE

#ifndef lint
typedef unsigned long long ub8;
#else
#define ub8 unsigned long long
#endif

#define UB8ZERO          ((ub8)0)

#define UB8MINVAL        ((ub8)0)
#define UB8MAXVAL        ((ub8)18446744073709551615)

#define MAXUB8MINVAL     ((ub8)0)
#define MINUB8MAXVAL     ((ub8)18446744073709551615)

#endif

#ifdef SLS8NATIVE

#ifndef lint
typedef signed long long sb8;
#else
#define sb8 signed long long
#endif

#define SB8ZERO          ((sb8)0)

#define SB8MINVAL        ((sb8)-9223372036854775808)
#define SB8MAXVAL        ((sb8) 9223372036854775807)
```

```
#define MAXSB8MINVAL ((sb8)-9223372036854775807)
#define MINSB8MAXVAL ((sb8) 9223372036854775807)

#endif

#undef CONST

#ifdef _olint
# define CONST const
#else
#if defined(PMAX) && defined(__STDC__)
#   define CONST const
#else
#   ifdef M88OPEN
#       define CONST const
#   else
#       if defined(SEQ_PSX) && defined(__STDC__)
#           define CONST const
#       else
#           ifdef A_OSF
#               if defined(__STDC__)
#                   define CONST const
#               else
#                   define CONST
#               endif
#           else
#               define CONST
#           endif
#       endif
#   endif
#endif
#endif
#endif

#ifdef lint
# define dvoid void
#else

#   ifdef UTS2
#       define dvoid char
#   else
#       define dvoid void
#   endif

#endif
```

```
#endif

typedef void (*lgenfp_t)( void );

#ifndef ORASYSTYPES
# include <sys/types.h>
# define ORASYSTYPES
#endif
#define boolean int

#ifdef sparc
# define SIZE_TMAXVAL SB4MAXVAL
#else
# define SIZE_TMAXVAL UB4MAXVAL
#endif

#define MINSIZE_TMAXVAL (size_t)65535

#endif
```

SQL Statement Processing

This chapter discusses the concepts and steps involved in processing SQL statements with the Oracle Call Interface.

The following topics are covered in this chapter:

- Overview
- Processing SQL Statements
- Preparing Statements
- Binding
- Executing Statements
- Describing Select-List Items
- Defining
- Fetching Results

Overview

Chapter 2 discussed the basic steps involved in any OCI application. This chapter presents a more detailed look at the specific tasks involved in processing SQL statements in an OCI program.

Processing SQL Statements

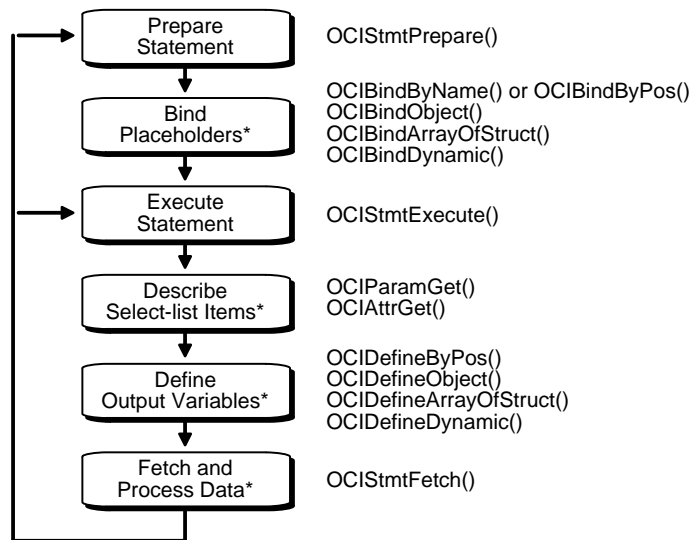
One of the most common tasks of an OCI program is to accept and process SQL statements. This section outlines the specific steps involved in processing SQL.

Once you have allocated the necessary handles and attached to a server, the basic steps in processing a SQL statement are the following, as illustrated in Figure 4–1:

1. **Prepare.** Define an application request using *OCIStmtPrepare()*.
2. **Bind.** For DML statements and queries with input variables, perform one or more bind calls using *OCIBindByPos()*, *OCIBindByName()*, *OCIBindObject()*, *OCIBindDynamic()* or *OCIBindArrayOfStruct()* to bind the address of each input variable (or PL/SQL output variable) or array to each placeholder in the statement.
3. **Execute.** Call *OCIStmtExecute()* to execute the statement. For DDL statements, no further steps are necessary.
4. **Describe.** Describe the select-list items, if necessary, using *OCIParamGet()* and *OCIAttrGet()*. This is an optional step; it is not required if the number of select-list items and the attributes of each item (such as its length and datatype) are known at compile time.
5. **Define.** For queries, perform one or more define calls to *OCIDefineByPos()*, *OCIDefineObject()*, *OCIDefineDynamic()*, or *OCIDefineArrayOfStruct()* to define an output variable for each select-list item in the SQL statement. Note that you do not use a define call to define the output variables in an anonymous PL/SQL block. You have done this when you have bound the data.
6. **Fetch.** For queries, call *OCIStmtFetch()* to fetch the results of the query.

Following these steps, the application can free allocated handles and then detach from the server, or it may process additional statements.

7.x Upgrade Note: OCI programs no longer require an explicit parse step. If a statement must be parsed, that step takes place on execute. This means that 8.0 applications must issue an execute command for both DML and DDL statements.

Figure 4–1 Steps In Processing SQL Statements

* These steps performed if necessary

For each of the steps in the diagram, the corresponding OCI function calls are listed. In some cases multiple calls may be required.

Each step above is described in detail in the following sections.

Note: Some variation in the order of steps is possible. For example, it is possible to do the define step before the execute if the datatypes and lengths of returned values are known at compile time. Also, as indicated by the asterisks (*), some steps may not be required by your application.

Additional steps beyond those listed above may be required if your application needs to do the following:

- initiate and manage multiple transactions
- manage multiple threads of execution
- perform piecewise inserts, updates, or fetches

These topics are described in Chapter 7.

Preparing Statements

SQL and PL/SQL statements need to be prepared for execution by using the statement prepare call and bind calls (if necessary). In this phase, the application specifies a SQL or PL/SQL statement and binds associated placeholders in the statement to data for execution. The client-side library allocates storage to maintain the statement prepared for execution.

An application requests a SQL or PL/SQL statement to be prepared for execution using the *OCIStmtPrepare()* call and passing it a previously allocated statement handle. This is a completely local call, requiring no round-trip to the server. No association is made at this point between the statement and a particular server.

Following the request call, an application can call *OCIAttrGet()* on the statement handle, passing *OCI_ATTR_STMT_TYPE* to the *attrtype* parameter, to determine what type of SQL statement was prepared. The possible attribute values, and corresponding statement types are listed in Table 4-1.

Table 4-1 *OCI_ATTR_STMT_TYPE Values and Statement Types*

Attribute Value	Statement Type
OCI_STMT_SELECT	SELECT statement
OCI_STMT_UPDATE	UPDATE statement
OCI_STMT_DELETE	DELETE statement
OCI_STMT_INSERT	INSERT statement
OCI_STMT_CREATE	CREATE statement
OCI_STMT_DROP	DROP statement
OCI_STMT_ALTER	ALTER statement
OCI_STMT_BEGIN	BEGIN... (PL/SQL)
OCI_STMT_DECLARE	DECLARE... (PL/SQL)

See Also: For more information on the specifics of using PL/SQL in an OCI application, see the section “Using PL/SQL in an OCI Program” on page 2-32.

The *OCIStmtPrepare()* call is described in more detail in Chapter 13, “OCI Relational Functions”.

Using Prepared Statements on Multiple Servers

A prepared application request can be executed on multiple servers at run time by reassociating the statement handle with the respective service context handles for the servers. All information cached about the current service context and statement handle association is lost when a new association is made.

For example, consider an application such as a network manager, which manages multiple servers. In many cases, it is likely that the same SELECT statement will need to be executed against multiple servers to retrieve information for display. The OCI allows the server manager application to prepare a SELECT statement once and execute it against multiple servers. It must fetch all of the required rows from each server prior to reassociating the prepared statement with the next server.

Note: If a prepared statement must be reexecuted frequently on the same server, it is efficient to prepare a new statement for another service context.

Binding

Most DML statements, and some queries (such as those with a WHERE clause), require a program to pass data to Oracle as part of a SQL or PL/SQL statement. Such data can be constant or literal data, known when your program is compiled. For example, the following SQL statement, which adds an employee to a database contains several literals, such as 'BESTRY' and 2365:

```
INSERT INTO emp VALUES
    (2365, 'BESTRY', 'PROGRAMMER', 2000, 20)
```

Hard coding a statement like this into an application would severely limit its usefulness. You would need to change the statement and recompile the program each time you add a new employee to the database. To make the program more flexible, you can write the program so that a user can supply input data at run time.

When you prepare a SQL statement or PL/SQL block that contains input data to be supplied at run time, placeholders in the SQL statement or PL/SQL block mark where data must be supplied. For example, the following SQL statement contains five placeholders, indicated by the leading colons (e.g., :ename), that show where input data must be supplied by the program.

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

You can use placeholders for input variables in any DELETE, INSERT, SELECT, or UPDATE statement, or PL/SQL block, in any position in the statement where you

can use an expression or a literal value. In PL/SQL, placeholders can also be used for output variables.

Note: Placeholders cannot be used to represent other Oracle objects such as tables. For example, the following is *not* a valid use of the `:emp` placeholder:

```
INSERT INTO :emp VALUES
(12345, 'OERTEL', 'WRITER', 50000, 30)
```

For each placeholder in the SQL statement or PL/SQL block, you must call an OCI routine that binds the address of a variable in your program to the placeholder. When the statement executes, Oracle gets the data that your program placed in the input, or bind, variables and passes it to the server with the SQL statement.

For detailed information about implementing bind operations, please refer to Chapter 5, “Binding and Defining”.

Executing Statements

An OCI application executes prepared statements individually using *OCIStmtExecute()*.

When an OCI application executes a query, it receives data from Oracle that matches the query specifications. Within the database, the data is stored in Oracle-defined formats. When the results are returned, an OCI application can request that data be converted to a particular host language format, and stored in a particular output variable or buffer.

For each item in the select-list of a query, the OCI application must define an output variable to receive the results of the query. The define step indicates the address of the buffer and the type of the data to be retrieved.

Note: If output variables are defined for a SELECT statement before a call to *OCIStmtExecute()*, the number of rows specified by the *iters* parameter are fetched directly into the defined output buffers and additional rows equivalent to the prefetch count are prefetched. If there are no additional rows, then the fetch is complete without calling *OCIStmtFetch()*.

For non-queries, the *iters* parameter of the *OCIStmtExecute()* call controls how many times the statement is executed during array operations. For example, if an array of 10 items is bound to a placeholder for an INSERT statement, and *iters* is set to 10, all 10 items will be inserted in a single execute call.

See Also: See the section “Defining” on page 4-11 for more information about defining output variables.

Execution Snapshots

The *OCISmtExecute()* call provides the ability to ensure that multiple service contexts operate on the same consistent snapshot of the database's committed data. This is achieved by taking the contents of the *snap_out* parameter of one *OCISmtExecute()* call and passing that value in the *snap_in* parameter of the next *OCISmtExecute()* call.

Note: Uncommitted data in one service context is *not* visible to another context, even when using the same snapshot.

The datatype of both the *snap_out* and *snap_in* parameter is **OCISnapshot**, an OCI snapshot descriptor. This descriptor is allocated with the *OCIDescAlloc()* function.

See Also: For more information about descriptors, see the section “Descriptors and Locators” on page 2-12.

It is not necessary to specify a snapshot when calling *OCISmtExecute()*. The following sample code shows a basic execution in which the snapshot parameters are passed as NULL.

```
checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (OCISnapshot *)NULL, (OCISnapshot *) NULL, OCI_DEFAULT))
```

Note: The *checkerr()* function evaluates the return code from an OCI application. The code for the function is listed in the section “Error Handling” on page 2-25.

Execution Modes

A user can specify one of three modes for the *OCISmtExecute()* call:

- **OCI_DEFAULT.** Calling *OCISmtExecute()* in this mode executes the statement. It also implicitly returns describe information about the select-list.
- **OCI_DESCRIBE_ONLY.** This mode is for users who wish to describe a query prior to execution. Calling *OCISmtExecute()* in this mode does not execute the statement, but it does return the select-list description.
- **OCI_COMMIT_ON_SUCCESS** - When a statement is executed in this mode, the current transaction is committed after execution, provided that execution completes successfully.

Describing Select-List Items

If your OCI application is processing a query, you may need to obtain more information about the items in the select-list. This is particularly true for dynamic queries whose contents are not known until run time. In this case, the program may need to obtain information about the datatypes and column lengths of the select-list items. This information is necessary to define output variables that will receive query results.

For example, a user might enter a query such as

```
SELECT * FROM employees
```

where the program has no prior information about the columns in the `employees` table.

In release 8.0, there are two types of describes available: implicit and explicit. An *implicit describe* is one which does not require any special calls to retrieve describe information from the server (although special calls *are* necessary to access the information). An *explicit describe* is one which requires the application to call a particular function to bring the describe information from the server.

An application may describe a select-list (query) either implicitly or explicitly. Other schema elements must be described explicitly.

An implicit describe allows an application to obtain select-list information as an attribute of the statement handle *after a statement has been executed* without making a specific describe call. It is called “implicit”, because no describe call is required. The describe information comes “free” with the execute.

Users may choose to describe a query explicitly prior to execution. To do this, specify `OCI_DESCRIBE_ONLY` as the mode of `OCIStmtExecute()`. Calling `OCIStmtExecute()` in this mode does not execute the statement, but it does return the select-list description. For performance reasons, however, it is recommended that applications take advantage of the implicit describe that comes “free” with a standard statement execution.

An explicit describe with the `OCIDescribeAny()` call obtains information about schema objects rather than select-lists.

In all cases, the specific information about columns and datatypes is retrieved by reading handle attributes.

See Also: For information about using `OCIDescribeAny()` to obtain meta-data pertaining to schema objects, refer to Chapter 6, “Describing Schema Metadata”.

Implicit Describe

After a SQL statement is executed, information about the select-list is available as an attribute of the statement handle. No explicit describe call is needed.

To retrieve information about select-list items from the statement handle, the application must call *OCIParmGet()* once for each position in the select-list to allocate a parameter descriptor for that position. Select-list positions are 1-based, meaning that the first item in the select-list is considered to be position number 1.

To retrieve information about multiple select-list items, an application can call *OCIParmGet()* with the *pos* parameter set to 1 the first time, and then iterate the value of *pos* and repeat the *OCIParmGet()* call until *OCI_NO_DATA* is returned. An application could also specify any position *n* to get a column at random.

Once a parameter descriptor has been allocated for a position in the select-list, the application can retrieve specific information by calling *OCIAttrGet()* on the parameter descriptor. Information available from the parameter descriptor includes the datatype and maximum size of the parameter.

The following sample code shows a loop that retrieves the column names and data types corresponding to a query following query execution. The query was associated with the statement handle by a prior call to *OCIStmtPrepare()*.

```

OCIParm      *mypard;
ub4          counter;
ub2          dtype;
text         *col_name;
ub4          col_name_len;
sb4          parm_status;

...

/* Request a parameter descriptor for position 1 in the select-list */
counter = 1;
parm_status = OCIParmGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
                        (ub4) counter);

/* Loop only if a descriptor was successfully retrieved for
   current position, starting at 1 */
while (parm_status==OCI_SUCCESS) {

/* Retrieve the data type attribute */
checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                        (dvoid*) &dtype, (ub4 *) 0, (ub4) OCI_ATTR_DATA_TYPE,
                        (OCIError *) errhp ));

```

```
/* Retrieve the column name attribute */
checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
    (dvoid**) &col_name, (ub4 *) &col_name_len, (ub4) OCI_ATTR_NAME,
    (OCIError *) errhp ));

printf("column=%s datatype=%d\n\n", col_name, dtype);
fflush(stdout);

/* increment counter and get next descriptor, if there is one */
counter++;
parm_status = OCIParmGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
    (ub4) counter);
}
```

Note: Error handling for the initial *OCIParmGet()* call is not included in this example. Ellipses (...) indicate portions of code that have been omitted for this example.

The *checkerr()* function is used for error handling. The complete listing can be found in the first sample application in Appendix D, “Code Examples”.

The calls to *OCIAttrGet()* and *OCIParmGet()* are local calls that do not require a network round trip, because all of the select-list information is cached on the client side after the statement is executed.

See Also: See the descriptions of *OCIParmGet()* and *OCIAttrGet()* in Chapter 13, “OCI Relational Functions”, for more information about these calls.

See the section “Parameter Attributes” on page 6-5 for a list of the specific attributes of the parameter descriptor which may be read by *OCIAttrGet()*.

Explicit Describe of Queries

Users may choose to describe a query explicitly prior to execution. To do this, specify *OCI_DESCRIBE_ONLY* as the mode of *OCISstmtExecute()*. Calling *OCISstmtExecute()* in this mode does not execute the statement, but it does return the select-list description.

Note: To maximize performance, it is recommended that applications execute the statement in default mode and use the implicit describe which accompanies the execution.

The following short example demonstrates the use of this mechanism to perform an explicit describe of a select-list to return information about the columns in the select-list. This pseudo-code shows how to retrieve column information (for example, data type).


```

/* initialize svchp, stmhp, errhp, rowoff, iters, snap_in, snap_out */
/* set the execution mode to OCI_DESCRIBE_ONLY. Note that setting the mode to
OCI_DEFAULT does an implicit describe of the statement in addition to executing
the statement */

OCIParam *colhd; /* column handle */
checkerr(errhp, OCISStmtExecute(svchp, stmhp, errhp, iters, rowoff,
                               snap_in, snap_out, OCI_DESCRIBE_ONLY);

/* Get the number of columns in the query */
checkerr(errhp, OCIAttrGet(stmhp, OCI_HTYPE_STMT, &numcols,
                           0, OCI_ATTR_PARAM_COUNT, errh));

/* go through the column list and retrieve the data type of each column. We
start from pos = 1 */
for (i = 1; i <= numcols; i++)
{
    /* get parameter for column i */
    checkerr(errhp, OCIParamGet(stmhp, OCI_HTYPE_STMT, errh, &colhd, i));

    /* get data-type of column i */
    checkerr(errhp, OCIAttrGet(colhd, OCI_DTYPE_PARAM,
                               &type[i-1], 0, OCI_ATTR_DATA_TYPE, errh));
}

```

Defining

Query statements return data from the database to your application. When processing a query, you must define an output variable or an array of output variables for each item in the select-list from which you want to retrieve data. The define step creates an association which determines where returned results are stored, and in what format.

For example, if your OCI statement processes the following statement:

```

SELECT name, ssn FROM employees
       WHERE empno = :empnum

```

you would normally need to define two output variables, one to receive the value returned from the name column, and one to receive the value returned from the ssn column.

For information about implementing define operations, please refer to Chapter 5, “Binding and Defining”.

Fetching Results

If an OCI application has processed a query, it is typically necessary to fetch the results with *OCIStmtFetch()* after the statement has been executed.

Fetches data is retrieved into output variables that have been specified by define operations.

Note: If output variables are defined for a SELECT statement before a call to *OCIStmtExecute()*, the number of rows specified by the *iters* parameter is fetched directly into the defined output buffers.

See Also: These statements fetch data associated with the sample code in the section “Steps Used in Defining” on page 5-14. Refer to that example for more information.

For information about defining output variables, see the section “Defining” on page 5-13.

Fetching LOB Data

If LOB columns or attributes are part of a select-list, LOB locators are returned as results of the query. The actual LOB value is not returned by the fetch. The application can perform further operations on these locators.

See Also: See the section “LOB and FILE Operations” on page 7-24 for more information about working with LOB locators in the OCI.

Setting Prefetch Count

In order to minimize server round trips and maximize the performance of applications, the OCI can prefetch result set rows when executing a query. The OCI programmer can customize this prefetching by setting the *OCI_ATTR_PREFETCH_ROWS* or *OCI_ATTR_PREFETCH_MEMORY* attribute of the statement handle using the *OCIAttrSet()* function.

OCI_ATTR_PREFETCH_ROWS sets the number of rows to be prefetched.

OCI_ATTR_PREFETCH_MEMORY sets the memory allocated for rows to be prefetched. The application then fetches as many rows as will fit into that much memory.

When both of these attributes are set, the OCI prefetches rows up to the *OCI_ATTR_PREFETCH_ROWS* limit unless the *OCI_ATTR_PREFETCH_MEMORY* limit is reached, in which case the OCI returns as many rows as will fit in a buffer of size *OCI_ATTR_PREFETCH_MEMORY*.

By default, prefetching is turned on, and the OCI fetches an extra row all the time. To turn prefetching off, set both the OCI_ATTR_PREFETCH_ROWS and OCI_ATTR_PREFETCH_MEMORY attributes 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.

See Also: For more information about these handle attributes, see the section "Statement Handle Attributes" on page B-15.

Binding and Defining

Chapter 2, “OCI Programming Basics”, introduced the concepts of binding and defining in OCI applications. This chapter revisits the basic concepts, and provides more detailed information about the different types of binds and defines you may use in OCI applications. The chapter includes short code examples to demonstrate the use of these different binds and defines.

Additionally, this chapter discusses the use of arrays of structures, as well as other issues involved in binding, defining, and character conversions.

Note: For information about binding and defining new Oracle8 datatypes for object applications, refer to Chapter 10.

This chapter includes the following sections:

- Binding
- Advanced Bind Operations
- Defining
- Advanced Define Operations
- Arrays of Structures
- DML with RETURNING Clause
- NCHAR and Character Conversion Issues
- PL/SQL REF CURSORS and Nested Tables

Binding

Most DML statements, and some queries (such as those with a WHERE clause), require a program to pass data to Oracle as part of a SQL or PL/SQL statement. Such data can be constant or literal data, known when your program is compiled. For example, the following SQL statement, which adds an employee to a database contains several literals, such as 'BESTRY' and 2365:

```
INSERT INTO emp VALUES
    (2365, 'BESTRY', 'PROGRAMMER', 2000, 20)
```

Hard coding a statement like this into an application would severely limit its usefulness. You would need to change the statement and recompile the program each time you add a new employee to the database. To make the program more flexible, you can write the program so that a user can supply input data at run time.

When you prepare a SQL statement or PL/SQL block that contains input data to be supplied at run time, placeholders in the SQL statement or PL/SQL block mark where data must be supplied. For example, the following SQL statement contains five placeholders, indicated by the leading colons (e.g., :ename), that show where input data must be supplied by the program.

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

You can use placeholders for input variables in any DELETE, INSERT, SELECT, or UPDATE statement, or PL/SQL block, in any position in the statement where you can use an expression or a literal value. In PL/SQL, placeholders can also be used for output variables.

Note: Placeholders cannot be used to name other Oracle objects such as tables or columns.

For each placeholder in the SQL statement or PL/SQL block, you must call an OCI routine that binds the address of a variable in your program to the placeholder. When the statement executes, Oracle gets the data that your program placed in the input, or bind, variables and passes it to the server with the SQL statement. Data does not have to be in a bind variable when you perform the bind step. At the bind step, you are only specifying the address, datatype, and length of the variable.

Note: If program variables do not contain data at bind time, make sure they contain valid data when you execute the SQL statement or PL/SQL block using *OCIStmtExecute()*.

For example, given the INSERT statement

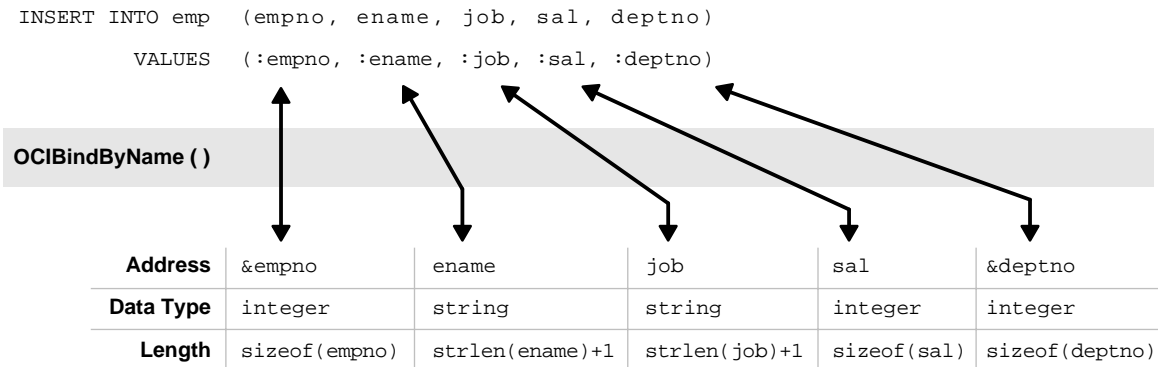
```
INSERT INTO emp VALUES
(:empno, :ename, :job, :sal, :deptno)
```

and the following variable declarations

```
text      *ename, *job
sword     empno, sal, deptno
```

the bind step makes an association between the placeholder name and the address of the program variables. The bind also indicates the datatype and length of the program variables, as illustrated in Figure 5–1. The code that implements this example is found in the section “Steps Used in Binding” on page 5-6.

Figure 5–1 Using *OCIBindByName()* to Associate Placeholders with Program Variables



If you change only the value of a bind variable, it is not necessary to rebind in order to execute the statement again. The bind is a bind by reference, so as long as the address of the bind variable and bind handle remain valid, you can reexecute a statement that references the variable without rebinding.

Note: At the interface level, all bind variables are considered at least IN and thus must be properly initialized (to zero if they are pure OUT bind variables).

For release 8.0, new datatypes have been implemented for named data types, REFs and LOBs, and they may be bound as placeholders in a SQL statement.

Note: For opaque data types (e.g., descriptors and locators) whose sizes are not known to the user, the address of the descriptor or locator pointer must be passed. Set the size parameter set to the size of the appropriate data structure (e.g., `sizeof(structure)`)

Named Binds and Positional Binds

The SQL statement in the previous section is an example of a *named bind*. Each placeholder in the statement has a name associated with it (e.g., 'ename' or 'sal'). When this statement is prepared and the placeholders are associated with values in the application, the association is made by the name of the placeholder using the *OCIBindByName()* call with the name of the placeholder passed in the *placeholder* parameter.

A second type of bind is known as a *positional bind*. In a positional bind, the placeholders are referred to by their position in the statement rather than their names. For binding purposes, an association is made between an input value and the position of the placeholder, using the *OCIBindByPos()* call.

The example from the previous section could also be used for a positional bind:

```
INSERT INTO emp VALUES
(:empno, :ename, :job, :sal, :deptno)
```

The five placeholders would then each be bound by calling *OCIBindByPos()* and passing the position number of the placeholder in the *position* parameter. For example, the `:empno` placeholder would be bound by calling *OCIBindByPos()* with a position of 1, `:ename` with a position of 2, and so on.

In the case of a duplicate bind, only a single bind call may be necessary. Consider the following SQL statement, which queries the database for those employees whose commission and salary are both greater than a given amount:

```
SELECT empno FROM emp
WHERE sal > :some_value
AND comm > :some_value
```

An OCI application could complete the binds for this statement with a single call to *OCIBindByName()* to bind the `:some_value` placeholder by name. In this case, the second placeholder inherits the bind information from the first placeholder.

OCI Array Interface

You can pass data to Oracle in various ways. You can execute a SQL statement repeatedly using the *OCIStmtExecute()* routine and supply different input values on each iteration. Alternatively, you can use the Oracle array interface and input many values with a single statement and a single call to *OCIStmtExecute()*. In this case you bind an array to an input placeholder, and the entire array can be passed at the same time, under the control of the *iters* parameter.

The array interface significantly reduces round-trips to Oracle when you need to update or insert a large volume of data. This reduction can lead to considerable performance gains in a busy client/server environment. For example, consider an application that needs to insert 10 rows into the database. Calling *OCIStmtExecute()* ten times with single values results in ten network round-trips to insert all the data. The same result is possible with a single call to *OCIStmtExecute()* using an input array, which involves only one network round-trip.

Note: When using the OCI array interface to perform inserts, row triggers in the database are fired as each row of the insert gets inserted.

Binding Placeholders in PL/SQL

You process a PL/SQL block by placing the block in a string variable, binding any variables, and executing the statement containing the block, just as you would with a single SQL statement.

When you bind placeholders in a PL/SQL block to program variables, you must use *OCIBindByName()* or *OCIBindByPos()* to perform the basic bind binds. You can use *OCIBindByName()* or *OCIBindByPos()* to bind host variables that are either scalars or arrays.

The following short PL/SQL block contains two placeholders, which represent IN parameters to a procedure that updates an employee's salary, given the employee number and the new salary amount:

```
char plsql_statement[] = "BEGIN\
                          RAISE_SALARY(:emp_number, :new_sal);\
                          END;" ;
```

These placeholders can be bound to input variables in the same way as placeholders in a SQL statement.

When processing PL/SQL statements, output variables are also associated with program variables using bind calls.

For example, in a PL/SQL block such as

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

you would use *OCIBindByName()* to bind variables in place of the `:emp_name`, `:salary`, and `:commission` output placeholders, and in place of the input placeholder `:emp_number`.

7.x Upgrade Note: In the Oracle7 OCI, it was sufficient for applications to initialize only IN-bind buffers. In Oracle8, all buffers, even pure OUT buffers, must be initialized by setting the buffer length to zero in the bind call, or by setting the corresponding indicator to -1.

See Also: For more information about binding PL/SQL placeholders see “Additional Information for Named Data Type and REF Binds” on page 10-3.

Steps Used in Binding

Binding placeholders is done in one or more steps. For a simple scalar or array bind, it is only necessary to specify an association between the placeholder and the data. This is done by using OCI bind by name (*OCIBindByName()*) or OCI bind by position (*OCIBindByPos()*) call.

Note: See the section “Named Binds and Positional Binds” on page 5-4 for information about the difference between these types of binds.

Once the bind is complete, the OCI library knows where to find the input data (or where to put PL/SQL output data) when the SQL statement is executed. As mentioned in the section “Binding” on page 5-2, program input data does not need to be in the program variable when it is bound to the placeholder, but the data must be there when the statement is executed.

The following code example shows handle allocation and binding for each of five placeholders in a SQL statement.

Note: The *checkerr()* function evaluates the return code from an OCI application. The code for the function is listed in the section “Error Handling” on page 2-25.

```
...
/* The SQL statement, associated with stmthp (the statement handle)
by calling OCIStmtPrepare() */
text *insert = (text *) "INSERT INTO emp(empno, ename, job, sal, deptno)\
VALUES (:empno, :ename, :job, :sal, :deptno)";
...

/* Bind the placeholders in the SQL statement, one per bind handle. */
checkerr(errhp, OCIBindByName(stmthp, &bndlp, errhp, (text *) ":ENAME",
    strlen(":ENAME"), (ub1 *) ename, enamelen+1, STRING_TYPE, (dvoid *) 0,
    (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT))
```

```

checkerr(errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":JOB",
    strlen(":JOB"), (ub1 *) job, joblen+1, STRING_TYPE, (dvoid *)
    &job_ind, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":SAL",
    strlen(":SAL"), (ub1 *) &sal, (sword) sizeof(sal), INT_TYPE,
    (dvoid *) &sal_ind, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0,
    OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd4p, errhp, (text *) ":DEPTNO",
    strlen(":DEPTNO"), (ub1 *) &deptno, (sword) sizeof(deptno), INT_TYPE,
    (dvoid *) 0, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd5p, errhp, (text *) ":EMPNO",
    strlen(":EMPNO"), (ub1 *) &empno, (sword) sizeof(empno), INT_TYPE,
    (dvoid *) 0, (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT))

```

PL/SQL Example

Perhaps the most common use for PL/SQL blocks in an OCI program is to call stored procedures or stored functions. For example, assume that there is a procedure called `RAISE_SALARY` stored in the database, and you want to call this procedure from an OCI program. You do this by embedding a call to that procedure in an anonymous PL/SQL block, then processing the PL/SQL block in the OCI program.

The following program fragment shows how to embed a stored procedure call in an OCI application. For the sake of brevity, only the relevant portions of the program are reproduced here.

The program passes an employee number and a salary increase as inputs to a stored procedure called `raise_salary`, which takes these parameters:

```
raise_salary (employee_num IN, sal_increase IN, new_salary OUT);
```

This procedure raises a given employee's salary by a given amount. The increased salary which results is returned in the stored procedure's OUT variable `new_salary`, and the program displays this value.

```

/* Define PL/SQL statement to be used in program. */
text *give_raise = (text *) "BEGIN\
    RAISE_SALARY(:emp_number,:sal_increase, :new_salary);\
END;";

OCIBind *bnd1p = NULL;           /* the first bind handle */
OCIBind *bnd2p = NULL;           /* the second bind handle */
OCIBind *bnd3p = NULL;           /* the third bind handle */

static void checkerr();

```

```
sb4 status;

main()
{
    sword    empno, raise, new_sal;
    dvoid    *tmp;
    OCISession *usrhp = (OCISession *)NULL;
    ...
    /* attach to database server, and perform necessary initializations
    and authorizations */
    ...
    /* allocate a statement handle */
    checkerr(usrhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
        OCI_HTYPE_STMT, 100, (dvoid **) &tmp));

    /* prepare the statement request, passing the PL/SQL text
    block as the statement to be prepared */
    checkerr(usrhp, OCISmtPrepare(stmthp, usrhp, (text *) give_raise, (ub4)
        strlen(give_raise), OCI_NTV_SYNTAX, OCI_DEFAULT));

    /* bind each of the placeholders to a program variable */
    checkerr(usrhp, OCIBindByName(stmthp, &bnd1p, usrhp, (text *) ":emp_number",
        -1, (ub1 *) &empno,
        (sword) sizeof(empno), SOLT_INT, (dvoid *) 0,
        (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

    checkerr(usrhp, OCIBindByName(stmthp, &bnd2p, usrhp, (text *) ":sal_increase",
        -1, (ub1 *) &raise,
        (sword) sizeof(raise), SOLT_INT, (dvoid *) 0,
        (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

    /* remember that PL/SQL OUT variable are bound, not defined */

    checkerr(usrhp, OCIBindByName(stmthp, &bnd3p, usrhp, (text *) ":new_salary",
        -1, (ub1 *) &new_sal,
        (sword) sizeof(new_sal), SOLT_INT, (dvoid *) 0,
        (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

    /* prompt the user for input values */
    printf("Enter the employee number: ");
    scanf("%d", &empno);
    /* flush the input buffer */
    myfflush();

    printf("Enter employee's raise: ");
```

```
scanf("%d", &raise);
    /* flush the input buffer */
myfflush();

    /* execute PL/SQL block*/
checkerr(errhp, OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
    (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT));

    /* display the new salary, following the raise */
printf("The new salary is %d\n", new_sal);
}
```

The following is one possible sample output from this program. Before execution, the salary of employee 7954 is 2000.

```
Enter the employee number: 7954
Enter employee's raise: 1000
```

The new salary is 3000

Advanced Binds

The previous section and example demonstrated how to perform a simple scalar bind. In that case, only a single bind call is necessary. In some cases, additional bind calls are necessary to define specific attributes for specific bind datatypes or execution modes. These more sophisticated bind operations are discussed in the following section.

Oracle8 also provides predefined C datatypes that map ADT attributes. Information about binding these datatypes (e.g., **OCIDate**, **OCINumber**) can be found in Chapter 10.

Advanced Bind Operations

The section “Binding” on page 4-5 discussed how a basic bind operation is performed to create an association between a placeholder in a SQL statement and a program variable using *OCIBindByName()* or *OCIBindByPos()*.

This section covers more advanced bind operations, including multi-step binds, and binds of named data types and REFs.

In certain cases, additional bind calls are necessary to define specific attributes for certain bind data types or certain execution modes.

The following sections describe these special cases, and the information about binding is summarized in Table 5-1 on page 5-12.

Static Array Binds

Static array bind attributes are set using the OCI array of structures bind call *OCIBindArrayOfStruct()*. This call is made following a call to *OCIBindByName()* or *OCIBindByPos()*.

Note: A static array bind does not refer to binding a column of type ARRAY of scalars or named data types, but a bind to a PL/SQL table or for multiple row operations in SQL (INSERTs/UPDATEs).

The *OCIBindArrayOfStruct()* call is also used to define the skip parameters needed if the application utilizes arrays of structures functionality.

See Also: For more information on using arrays of structures, see the section “Arrays of Structures” on page 5-17.

Named Data Type Binds

For information on binding named data types (objects), refer to “Named Data Type Binds” on page 10-2.

Binding REFs

For information on binding REFs, refer to “Binding REFs” on page 10-3.

Binding LOBs

When working with LOBs, the LOB locators, rather than the actual LOB value, are bound. The LOB value is written or read by passing a LOB locator to the PL/SQL DBMS_LOB package or OCI LOB functions.

Either a single locator or an array of locators can be bound in a single bind call. In each case, the application must pass *the address of a LOB locator* and not the locator itself.

For example, if an application has prepared a SQL statement like

```
INSERT INTO some_table VALUES (:one_lob)
```

where `:one_lob` is a bind variable corresponding to a LOB column, and has made the following declaration:

```
OCILOBLocator * one_lob;
```

then the following sequence of steps would be used to bind the placeholder, and execute the statement

```
/* initialize single locator */
one_lob = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
...
/* pass the address of the locator */
OCIBindByName(...,(dvoid *) &one_lob,...);
OCIStmtExecute(...,1,...)           /* 1 is the iters parameter */
```

Note: In these examples, most parameters are omitted for simplicity.

You could also do an array insert using the same SQL INSERT statement. In this case, the application would include the following code:

```
OCILobLocator * lob_array[10];
...
for (i=0; i<10, i++)
lob_array[i] = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
/* initialize array of locators */
...
OCIBindByName(...,(dvoid *) lob_array,...);
OCIStmtExecute(...,10,...);           /* 10 is the iters parameter */
```

Note that you must allocate descriptors with the *OCIDescriptorAlloc()* routine before they can be used. In the case of an array of locators, you must initialize each array element using *OCIDescriptorAlloc()*. Use *OCI_DTYPE_LOB* as the *type* parameter when allocating BLOBs, CLOBs, and NCLOBs. Use *OCI_DTYPE_FILE* when allocating BFILEs.

See Also: For more information about OCI LOB functions, refer to the section “LOB and FILE Operations” on page 7-24.

Binding in OCI_DATA_AT_EXEC Mode

If the *mode* parameter in a call to *OCIBindByName()* or *OCIBindByPos()* is set to *OCI_DATA_AT_EXEC*, an additional call to *OCIBindDynamic()* is necessary if the application will use the callback method for providing data at runtime. The call to *OCIBindDynamic()* sets up the callback routines, if necessary, for indicating the data or piece that is being provided.

If the *OCI_DATA_AT_EXEC* mode is chosen, but the standard OCI piecewise polling method will be used instead of callbacks, the call to *OCIBindDynamic()* is not necessary.

When binding RETURN clause variables, an application must use OCI_DATA_AT_EXEC mode, and it must provide callbacks.

See Also: For more information about piecewise operations, please refer to the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

Binding Ref Cursor Variables

Ref Cursors are bound to a statement handle with a bind datatype of SQLT_RSET. See “PL/SQL REF CURSORS and Nested Tables” on page 5-27

Summary of Bind Information

The following table summarizes the bind calls necessary for different types of binds. For each type, the table lists the bind datatype (passed in the *dt*y parameter of *OCIBindByName()* or *OCIBindByPos()*), and notes about the bind.

Table 5–1 Bind Information for Different Bind Types

Type of Bind	Bind Datatype	Notes
Scalar	any scalar datatype	Bind a single scalar using <i>OCIBindByName()</i> or <i>OCIBindByPos()</i> .
Array of Scalars	any scalar datatype	Bind an array of scalars using <i>OCIBindByName()</i> or <i>OCIBindByPos()</i> .
Named Data Type	SQLT_NTY	Two bind calls are required: <ul style="list-style-type: none">▪ <i>OCIBindByName()</i> or <i>OCIBindByPos()</i>▪ <i>OCIBindObject()</i>
REF	SQLT_REF	Two bind calls are required: <ul style="list-style-type: none">▪ <i>OCIBindByName()</i> or <i>OCIBindByPos()</i>▪ <i>OCIBindObject()</i>
LOB	SQLT_BLOB SQLT_CLOB	Allocate the LOB locator using <i>OCIDescriptorAlloc()</i> , and then bind its address (OCIlobLocator **) with <i>OCIBindByName()</i> or <i>OCIBindByPos()</i> , using one of the LOB datatypes.

Table 5–1 Bind Information for Different Bind Types (Cont.)

Type of Bind	Bind Datatype	Notes
Array of Structures or Static Arrays	varies	Two bind calls are required: <ul style="list-style-type: none"> ■ OCIBindByName() or OCIBindByPos() ■ OCIBindArrayOfStruct()
Piecewise Insert	varies	OCIBindByName() or OCIBindByPos() is required. The application may also need to call OCIBindDynamic() to register piecewise callbacks.
REF CURSOR variables	SQLT_RSET	Allocate a statement handle, OCISmt , and then bind its address (OCISmt **) using the SQLT_RSET datatype.

See Also: For more information about datatypes and datatype codes, see Chapter 3, “Datatypes”.

Defining

Query statements return data from the database to your application. When processing a query, you must define an output variable or an array of output variables for each item in the select-list from which you want to retrieve data. The define step creates an association that determines where returned results are stored, and in what format.

For example, if your OCI statement processes the following statement:

```
SELECT name, ssn FROM employees
WHERE empno = :empnum
```

you would normally need to define two output variables, one to receive the value returned from the `name` column, and one to receive the value returned from the `ssn` column.

For information about implementing define operations, please refer to Chapter 5, “Binding and Defining”.

Note: If you were only interested in retrieving values from the `name` column, you would not need to define an output variable for `ssn`.

If the SELECT statement being processed might return more than a single value for a query, the output variables you define may be arrays instead of scalar values.

Note: Depending on the application, the define step can take place before or after the execute. If the datatypes of select-list items are known when the

application is coded, the define can take place before the statement is executed. If your application is processing dynamic SQL statements—statements entered by the user at run time— or statements that do not have a clearly defined select-list, such as

```
SELECT * FROM employees
```

the application must execute the statement and retrieve describe information before defining output variables. See the section “Describing Select-List Items” on page 4-8 for more information.

The OCI processes the define call locally, on the client side. In addition to indicating the location of buffers where results should be stored, the define step also determines what type of data conversions, if any, will take place when data is returned to the application.

The *dt* parameter of the *OCIDefineByPos()* call specifies the datatype of the output variable. The OCI is capable of a wide range of data conversions when data is fetched into the output variable. For example, internal data in Oracle DATE format can be automatically converted to a string datatype on output.

See Also: For more information about datatypes and conversions, refer to Chapter 3, “Datatypes”.

Steps Used in Defining

Defining output variables is done in one or more steps. A basic define is accomplished with the OCI define by position call, *OCIDefineByPos()*. This step creates an association between a select-list item and an output variable. Additional define calls may be necessary for certain datatypes or fetch modes.

Once the define step is complete, the OCI library knows where to put retrieved data after fetching it from the database.

Note: You can make your define calls again to redefine the output variables without having to reprepare or reexecute the SQL statement.

The following example code shows a scalar output variable being defined following an execute and a describe.

```
/* The following statement was prepared, and associated with statement  
handle stmthp1.
```

```
SELECT dname FROM dept WHERE deptno = :dept_input
```

```
The input placeholder was bound earlier, and the data comes from the
```

```

user input below */

printf("Enter employee dept: ");
scanf("%d", &deptno);
myfflush();

/* Execute the statement. If OCISmtExecute() returns OCI_NO_DATA, meaning that
no data matches the query, then the department number is invalid. */
if ((status = OCISmtExecute(svchp, stmthp1, errhp, 1, 0, 0, 0,
OCI_DEFAULT))
    && (status != OCI_NO_DATA))
{
    checkerr(errhp, status);
    do_exit(EXIT_FAILURE);
}
if (status == OCI_NO_DATA) {
    printf("The dept you entered doesn't exist.\n");
    return 0;
}

/* The next two statements describe the select-list item, dept, and
return its length */
checkerr(errhp, OCIParmGet(stmthp1, errhp, &parmdp, (ub4) 1));
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
(dvoid*) &deptlen, (ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
(OCIError *) errhp ));

/* Use the retrieved length of dept to allocate an output buffer, and
then define the output variable. If the define call returns an error,
exit the application */
dept = (text *) malloc((int) deptlen + 1);
if (status = OCIDefineByPos(stmthp1, &defnp, errhp,
1, (ub1 *) dept, deptlen+1,
SQLT_STRING, (dvoid *) 0,
(ub2 *) 0, OCI_DEFAULT))
{
    checkerr(errhp, status);
    do_exit(EXIT_FAILURE);
}

```

For an explanation of the describe step, see the section “Describing Select-List Items” on page 4-8.

Advanced Defines

In some cases the define step requires more than just a call to *OCIDefineByPos()*. There are additional calls that define the attributes of an array fetch

(*OCIDefineArrayOfStruct()*) or a named data type fetch (*OCIDefineObject()*). For example, to fetch multiple rows with a column of named data types, all three calls must be invoked for the column; but to fetch multiple rows of scalar columns, *OCIDefineArrayOfStruct()* and *OCIDefineByPos()* are sufficient.

These more sophisticated define operations are covered in the section “Advanced Define Operations” on page 5-16.

Oracle8 also provides pre-defined C datatypes that map object type attributes. Information about defining these datatypes (e.g., **OCIDate**, **OCINumber**) can be found in Chapter 10.

Advanced Define Operations

The section “Defining” on page 4-11 discussed how a basic bind operation is performed to create an association between a SQL select-list item and an output buffer in an application.

This section covers more advanced defined operations, including multi-step defines, and defines of named data types and REFs.

In some cases the define step requires more than just a call to *OCIDefineByPos()*. There are additional calls that define the attributes of an array fetch (*OCIDefineArrayOfStruct()*) or a named data type fetch (*OCIDefineObject()*). For example, to fetch multiple rows with a column of named data types, all the three calls must be invoked for the column; but to fetch multiple rows of scalar columns only *OCIDefineArrayOfStruct()* and *OCIDefineByPos()* are sufficient.

The following sections discuss specific information pertaining to different types of defines.

Defining Named Data Type Output Variables

For information on defining named data type (object) output variables, refer to “Defining Named Data Type Output Variables” on page 10-4.

Defining REF Output Variables

For information on defining REF output variables, refer to “Defining REF Output Variables” on page 10-4.

Defining LOB Output Variables

For LOBs, the buffer pointer must be a locator of type **OCILOBLocator**, allocated by the *OCIDescriptorAlloc()* call. LOB locators, and not LOB values, are always

returned for a LOB column. LOB values can then be fetched using OCI LOB calls on the fetched locator.

Defining PL/SQL Output Variables

You do not use the define calls to define output variables for select-list items in a SQL SELECT statement in a PL/SQL block. You must use OCI bind calls instead.

See Also: See the section “Additional Information for Named Data Type and REF Defines, and PL/SQL OUT Binds” on page 10-5 for more information about defining PL/SQL output variables.

Defining For a Piecewise Fetch

When performing a piecewise fetch, an initial call to *OCIDefineByPos()* is required. An additional call to *OCIDefineDynamic()* is necessary if the application will use callbacks rather than the standard polling mechanism for fetching data.

See Also: See the section “Run Time Data Allocation and Piecewise Operations” on page 7-16 for more information.

Defining Arrays of Structures

When using arrays of structures, an initial call to *OCIDefineByPos()* is required. An additional call to *OCIDefineArrayOfStruct()* is necessary to set up additional parameters, including the skip parameter necessary for arrays of structures operations.

See Also: For more information, refer to the section “Arrays of Structures” on page 5-17.

Arrays of Structures

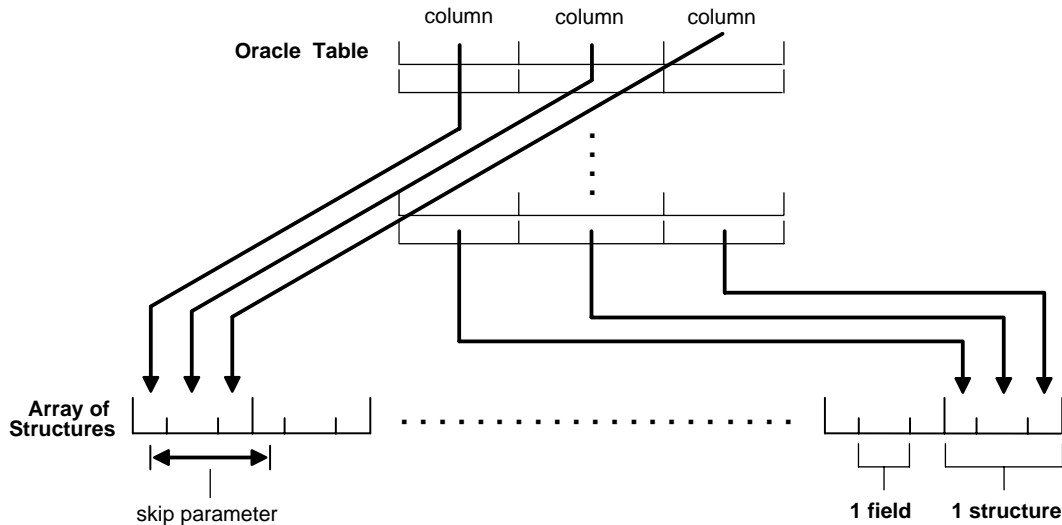
The “arrays of structures” functionality of the Oracle8 OCI can simplify the processing of multi-row, multi-column operations. The OCI programmer can create a structure of related scalar data items and then fetch values from the database into an array of these structures or insert values into the database from an array of these structures.

For example, an application may need to fetch multiple rows of data from three columns named NAME, AGE, and SALARY. The OCI application could include the definition of a structure containing separate fields to hold the NAME, AGE and SALARY data from one row in the database table. The application would then fetch data into an array of these structures.

In order to perform a multi-row, multi-column operation using an array of structures, the developer associates each column involved in the operation with a field in a structure. This association, which is part of the *OCIDefineArrayOfStruct()* and *OCIBindArrayOfStruct()* calls, specifies where fetched data will be stored, or where inserted or updated data will be found.

Figure 5–2 is a graphical representation of this process. In the figure, an application fetches various fields from a database row into a single structure in an array of structures. Each column being fetched corresponds to one of the fields in the structure.

Figure 5–2 Fetching Data Into an Array of Structures



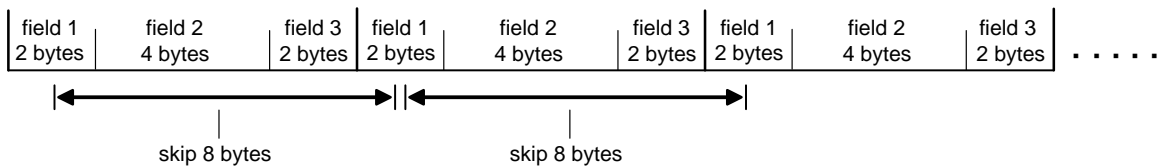
Skip Parameters

When you split column data across an array of structures, it is no longer contiguous. The single array of structures stores data as though it were composed of several interleaved arrays of scalars. Because of this fact, developers must specify a “skip parameter” for each field they are binding or defining. This skip parameter specifies the number of bytes that need to be skipped in the array of structures before the same field is encountered again. In general this will be equivalent to the byte size of one structure.

The figure below demonstrates how a skip parameter is determined. In this case the skip parameter is the sum of the sizes of the fields *field1*, *field2* and *field3*, which is 8 bytes. This equals the size of one structure.

Figure 5–3 Determining Skip Parameters.

Array of Structures



On some systems it may be necessary to set the skip parameter to be *sizeof*(one array element) rather than *sizeof*(struct). This is because some compilers may insert padding into a structure. For example, consider an array of C structures consisting of two fields, a **ub4** and a **ub1**.

```
struct demo {
    ub4 field1;
    ub1 field2;
};
struct demo demo_array[MAXSIZE];
```

Some compilers insert three bytes of padding after the **ub1** so that the **ub4** which begins the next structure in the array is properly aligned. In this case, the following statement may return an incorrect value:

```
skip_parameter = sizeof(struct demo);
```

On some systems this will produce a proper skip parameter of eight. On other systems, *skip_parameter* will be set to five bytes by this statement. In this case, use the following statement to get the correct value for the skip parameter:

```
skip_parameter = sizeof(demo_array[0]);
```

Skip Parameters for Standard Arrays

The ability to work with arrays of structures is an extension of the functionality for binding and defining arrays of program variables. Programmers can also work with standard arrays (as opposed to arrays of structures). When specifying a

standard array operation, the related skip will be equal to the size of the datatype of the array under consideration. For example, for an array declared as

```
text emp_names[4][20]
```

the skip parameter for the bind or define operation will be 20. Each data element in the array is then recognized as a separate unit, rather than being part of a structure.

OCI Calls Used with Arrays of Structures

Two OCI calls must be used when performing operations involving arrays of structures: *OCIBindArrayOfStruct()* (for binding fields in arrays of structures for input variables) and *OCIDefineArrayOfStruct()* (for defining arrays of structures for output variables).

Note: When binding or defining for arrays of structures, multiple calls are required. A call to *OCIBindByName()* or *OCIBindByPos()* must proceed a call to *OCIBindArrayOfStruct()*, and a call to *OCIDefineByPos()* must proceed a call to *OCIDefineArrayOfStruct()*.

See Also: See the descriptions of *OCIBindArrayOfStruct()* and *OCIDefineArrayOfStruct()* in Chapter 13 for syntax and parameter descriptions.

Arrays of Structures and Indicator Variables

The implementation of arrays of structures also supports the use of indicator variables and return codes. OCI application developers can declare parallel arrays of column-level indicator variables and return codes, corresponding to the arrays of information being fetched, inserted, or updated. These arrays can have their own skip parameters, which are specified during a call to *OCIBindArrayOfStruct()* or *OCIDefineArrayOfStruct()*.

You can set up arrays of structures of program values and indicator variables in many ways. For example, consider an application that fetches data from three database columns into an array of structures containing three fields. You can set up a corresponding array of indicator variable structures of three fields, each of which is a column-level indicator variable for one of the columns being fetched from the database.

Note: A one-to-one relationship between the fields in an indicator struct and the number of select-list items is not necessary.

See Also: See “Indicator Variables” on page 2-29 for more information about indicator variables.

DML with RETURNING Clause

The OCI supports the use of the RETURNING clause with SQL INSERT, UPDATE, and DELETE statements. This section outlines the rules an OCI application must follow to correctly implement DML statements with the RETURNING clause.

Note: For more information about the use of the RETURNING clause with INSERT, UPDATE, or DELETE statements, please refer to the descriptions of those commands in the *Oracle8 SQL Reference*.

For an complete code example, refer to “Example 3, DML with RETURNING Clause” on page D-25.

Using DML with RETURNING Clause

Using the RETURNING clause with a DML statement allows you to essentially combine two SQL statements into one, possibly saving you a server round-trip. This is accomplished by adding an extra clause to the traditional UPDATE, INSERT, and DELETE statements. The extra clause effectively adds a query to the DML statement.

In the OCI, the values are returned to the application through the use of OUT bind variables. The rules for binding these variables are described in the next section. In the following examples, the bind variables are indicated by the preceding colon (e.g., :out1). These examples assume the existence of a table called `table1`, which contains three columns: `col1`, `col2`, and `col3`.

For example, the following statement inserts new values into the database and then retrieves the column values of the affected row from the database, allowing your application to work with inserted rows.

```
INSERT INTO table1 VALUES (:1, :2, :3,)
      RETURNING col1, col2, col3
      INTO :out1, :out2, :out3
```

The next example uses the UPDATE statement. This statement updates the values of all columns whose `col1` value falls within a certain range, and then returns the affected rows to the application, allowing the application to see which rows were modified.

```
UPDATE table1 SET col1 = col1 + :1, col2 = :2, col3 = :3
      WHERE col1 >= :low AND col1 <= :high
      RETURNING col1, col2, col3
      INTO :out1, :out2, :out3
```

The following DELETE statement deletes the rows whose `col1` value falls within a certain range, and then returns the data from those rows so that the application can check them.

```
DELETE FROM table1 WHERE col1 >= :low AND col2 <= :high
    RETURNING col1, col2, col3
    INTO :out1, :out2, :out3
```

Note that in both the UPDATE and DELETE examples there is the possibility that the statement will affect multiple rows in the table. Additionally, a DML statement could be executed multiple times in a single *OCIExecute()* statement. Because of this possibility for multiple returning values, an OCI application may not know how much data will be returned at runtime. As a result, the variables corresponding to the RETURNING...INTO placeholders must be bound in OCI_DATA_AT_EXEC mode. It is an additional requirement that the application must define its own dynamic data handling callbacks (rather than using the OCI_DATA_AT_EXEC polling mechanism).

Note: Even if the application can be sure that it will only get a single value back in the RETURNING clause, it must still bind in OCI_DATA_AT_EXEC mode and use callbacks.

The returning clause can be particularly useful when working with LOBs. Normally, an application must insert an empty LOB locator into the database, and then SELECT it back out again to operate on it. Using the RETURNING clause, the application can combine these two steps into a single statement:

```
INSERT INTO some_table VALUES (:in_locator)
    RETURNING lob_column
    INTO :out_locator
```

Binding RETURNING...INTO variables

As mentioned in the previous section, an OCI application implements the placeholders in the RETURNING clause as pure OUT bind variables. An application must adhere to the following rules when working with these bind variables:

1. Bind RETURNING clause placeholders in OCI_DATA_AT_EXEC mode using *OCIBindByName()* or *OCIBindByPos()*, followed by a call to *OCIBindDynamic()* for each placeholder.

Note: The OCI only supports the callback mechanism for RETURNING clause binds. The polling mechanism is not supported.

2. When binding RETURNING clause placeholders, you must supply a valid out bind function as the *ocbfp* parameter of the *OCIBindDynamic()* call. This function must provide storage to hold the returned data.
3. The *icbfp* parameter of *OCIBindDynamic()* call should provide a “dummy” function which returns NULL values when called.
4. The *piecep* parameter of *OCIBindDynamic()* must be set to OCI_ONE_PIECE.
5. No duplicate binds are allowed in a DML statement with a RETURNING clause (i.e., no duplication between bind variables in the DML section and the RETURNING section of the statement).

Error Handling

The out bind function provided to *OCIBindDynamic()* must be prepared to receive partial results of a statement in the event of an error. For example, if the application has issued a DML statement which should be executed 10 times, and an error occurs during the fifth iteration, the server will still return the data from iterations 1 through 4. The callback function would still be called to receive data for the first four iterations.

DML with RETURNING REF...INTO clause

The RETURNING clause can also be used to return a REF to an object which is being inserted into or updated in the database. The following SQL statement shows how this could be used.

```
UPDATE EXTADDR E SET E.ZIP = '12345', E.STATE='AZ'
WHERE E.STATE = 'CA' AND E.ZIP='95117'
RETURNING REF(E), ZIP
INTO :addref, :zip
```

This statement updates several attributes of an object in an object table and then returns a REF to the object (along with the scalar ZIP code) in the RETURNING clause.

Binding the REF output variable in an OCI application requires three steps:

1. The initial bind information is set using *OCIBindByName()*
2. Additional bind information for the REF (including the TDO) is set with *OCIBindObject()*
3. A call to *OCIBindDynamic()*

The following pseudocode shows a function which performs the binds necessary for the above example.

```
sword bind_output(stmthp, bndhp, errhp)
OCIStmt *stmthp;
OCIBind *bndhp[];
OCIError *errhp;
{
    ub4 i;

    /* get TDO for BindObject call */
    if (OCITypeByName(envhp, errhp, svchp, (CONST text *) 0,
        (ub4) 0, (CONST text *) "ADDRESS_OBJECT",
        (ub4) strlen((CONST char *) "ADDRESS_OBJECT"),
        (CONST text *) 0, (ub4) 0,
        OCI_DURATION_SESSION, OCI_TYPEGET_HEADER, &addrtdo))
    {
        return OCI_ERROR;
    }

    /* initial bind call for both variables */
    if (OCIBindByName(stmthp, &bndhp[2], errhp,
        (text *) ":addref", (sb4) strlen((char *) ":addref"),
        (dvoid *) 0, (sb4) sizeof(OCIRef *), SOLT_REF,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
    || OCIBindByName(stmthp, &bndhp[3], errhp,
        (text *) ":zip", (sb4) strlen((char *) ":zip"),
        (dvoid *) 0, (sb4) MAXZIPLLEN, SOLT_CHR,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC))
    {
        return OCI_ERROR;
    }

    /* object bind for REF variable */
    if (OCIBindObject(bndhp[2], errhp, (OCIType *) addrtdo,
        (dvoid **) &addrref[0], (ub4 *) 0, (dvoid **) 0, (ub4 *) 0))
    {
        return OCI_ERROR;
    }

    for (i = 0; i < MAXCOLS; i++)
        pos[i] = i;

    /* dynamic binds for both RETURNING variables */
}
```

```

if (OCIBindDynamic(bndhp[2], errhp, (dvoid *) &pos[0], cbf_no_data,
                  (dvoid *) &pos[0], cbf_get_data)
|| OCIBindDynamic(bndhp[3], errhp, (dvoid *) &pos[1], cbf_no_data,
                  (dvoid *) &pos[1], cbf_get_data))
{
    return OCI_ERROR;
}

return OCI_SUCCESS;
}

```

Additional Notes About Callbacks

When a callback function is called, the `OCI_ATTR_ROWS_RETURNED` attribute of the bind handle tells the application the number of rows being returned in that particular iteration. Thus, when the callback is called the first time in a particular iteration (i.e., `index=0`), the user can allocate space for all the rows which will be returned for that bind variable. When the callback is called subsequently (with `index>0`) within the same iteration, the user can merely increment the buffer pointer to the correct memory within the allocated space to retrieve the data.

NCHAR and Character Conversion Issues

This section discusses issues involving NCHAR data and character conversions between the client and the server.

NCHAR Issues

Oracle8 provides support for NCHAR data in the database, and the Oracle8 OCI provides support for binding and defining NCHAR data. If a database column containing character data is defined to be an NCHAR column, then a bind or define involving that column must take into account special considerations for dealing with character set specifications.

These considerations are necessary in case the width of the client character set is different from that on the server, and also for proper character conversion between the client and server. During conversion of data between different character sets, the size of the data may grow or shrink as much as fourfold. Care must be taken to insure that buffers provided to hold the data are of sufficient size.

In some cases, it may also be easier for an application to deal with NCHAR data in terms of numbers of characters, rather than numbers of bytes (which is the usual case).

Each OCI bind and define handle has “form” (OCI_ATTR_CHRSETFORM) and “character set ID” (OCI_ATTR_CHRSETID) attributes associated with it. An application can set these attributes with the *OCIAttrSet()* call in order to specify the character set ID and form of the bind/define buffer.

The *form* attribute has two possible values:

- SQLCS_IMPLICIT - database character set ID
- SQLCS_NCHAR - NCHAR character set ID

The default value is SQLCS_IMPLICIT.

If the character set ID is not specified, then the default value of the database or NCHAR character set ID of the client is used, depending on the value of *form*. That is the value specified in the NLS_LANG and NLS_NCHAR environment variables.

If nothing is specified, then the default database character set ID of the client is assumed.

Note: No matter what values are assigned to the character set ID and form of the client-side bind buffer, the data is converted and inserted into the database according to the server’s database/NCHAR character set ID and form.

See Also: For more information about NCHAR data, refer to the *Oracle8 Reference*.

OCI_ATTR_MAXDATA_SIZE Attribute

Every bind handle has a OCI_ATTR_MAXDATA_SIZE attribute. This attribute specifies the number of bytes to be allocated on the server to accommodate the client-side bind data after any necessary character set conversions.

Note: Character set conversions performed when data is sent to the server may result in the data expanding or contracting, so its size on the client may not be the same as its size on the server.

An application will typically set OCI_ATTR_MAXDATA_SIZE to the maximum size of the column or the size of the PL/SQL variable, depending on how it is used. Oracle issues an error if OCI_ATTR_MAXDATA_SIZE is not a large enough value to accommodate the data after conversion, and the operation will fail.

Character Count Attribute

Bind and define handles have a character count attribute associate with them. An application can use this attribute to work with data in terms of numbers of characters, rather than numbers of bytes. If this attribute is set to a non-zero value,

it indicates that all calculations should be done in terms of characters instead of bytes, and any constraint sizes should be thought of in terms of characters rather than bytes.

This attribute can be set in addition to the `OCI_ATTR_MAXDATA_SIZE` attribute for bind handles. For example, if `OCI_ATTR_MAXDATA_SIZE` is set to 100, and `OCI_ATTR_CHAR_COUNT` is set to 0, this means that the maximum possible size of the data on the server after conversion is 100 bytes. However, if `OCI_ATTR_MAXDATA_SIZE` is set to 100, and `OCI_ATTR_CHAR_COUNT` is set to a non-zero value, then if the character set has 2 bytes/character, the maximum possible allocated size is 200 bytes (2 bytes/char * 100 chars).

Note: This attribute is valid only for fixed-width character set IDs. For variable-width character set IDs, these values are always treated as numbers of bytes, rather than numbers of characters.

For binds, the `OCI_ATTR_CHAR_COUNT` attribute sets the number of characters that an application wants to reserve on the server to store the data being bound. This overrides the `OCI_ATTR_MAXDATA_SIZE` attribute. For all datatypes that have a length prefix as part of their value (e.g., `VARCHAR2`), the length prefix is then considered to be the number of characters, rather than the number of bytes. In this case, indicator lengths and return codes are also in characters.

Note: Regardless of the value of the `OCI_ATTR_CHAR_COUNT` attribute, the buffer lengths specified in a bind or define call are always considered to be in terms of number of bytes. The actual length values sent and received by the user are also in characters in this case.

For defines, the `OCI_ATTR_CHAR_COUNT` attribute specifies the maximum number of characters of data the client application wants to receive. This constraint overrides the *maxlength* parameter specified in the *OCIDefineByPos()* call.

PL/SQL REF CURSORS and Nested Tables

The OCI provides the ability to bind and define PL/SQL REF CURSORS and nested tables. An application can use a statement handle to bind and define these types of variables. As an example, consider this PL/SQL block:

```
static const text *plsql_block = (text *)
    "begin \
      OPEN :cursor1 FOR SELECT empno, ename, job, mgr, sal, deptno \
        FROM emp_rc WHERE job=:job ORDER BY empno; \
      OPEN :cursor2 FOR SELECT * FROM dept_rc ORDER BY deptno; \
    end;";
```

An application would allocate a statement handle for binding, by calling *OCIHandleAlloc()*, and then bind the `:cursor1` placeholder to the statement handle, as in the following code, where `:cursor1` is bound to `stm2p`. Note that the handle allocation code is not included here.

```
err = OCIStmtPrepare (stm1p, errhp, (text *) nst_tab, strlen(nst_tab),
                    OCI_NTV_SYNTAX, OCI_DEFAULT);
...
err = OCIBindByName (stm1p, (OCIBind **) bndp, errhp,
                    (text *)":cursor1", (sb4)strlen((char *)":cursor1"),
                    (dvoid *)&stm2p, (sb4) 0,  SOLT_RSET, (dvoid *)0,
                    (ub2 *)0, (ub2 *)0, (ub4)0, (ub4 *)0, (ub4)OCI_DEFAULT);
```

In this code, `stm1p` is the statement handle for the PL/SQL block, while `stm2p` is the statement handle which is bound as a REF CURSOR for later data retrieval. A value of `SOLT_RSET` is passed for the *dtv* parameter.

As another example, consider the following:

```
static const text *nst_tab = (text *)
    "SELECT ename, CURSOR(SELECT dname, loc FROM dept_rc) \
    FROM emp_rc WHERE ename = 'LOCKE'";
```

In this case the second position is a nested table, which an OCI application can define as a statement handle as follows. Note that the handle allocation code is not included here.

```
err = OCIStmtPrepare (stm1p, errhp, (text *) nst_tab, strlen(nst_tab),
                    OCI_NTV_SYNTAX, OCI_DEFAULT);
...
err = OCIDefineByPos (stm1p, (OCIDefine **) dfn2p, errhp, (ub4)2,
                    (dvoid *)&stm2p,
                    (sb4)0, SOLT_RSET, (dvoid *)0, (ub2 *)0,
                    (ub2 *)0, (ub4)OCI_DEFAULT);
```

After execution, when you fetch a row into `stm2p` it then becomes a valid statement handle.

Note: If you have retrieved multiple ref cursors, you must take care when fetching them into `stm2p`. If you fetch the first one, you can then perform fetches on it to retrieve its data. However, once you fetch the second ref cursor into `stm2p`, you no longer have access to the data from the first ref cursor.

Describing Schema Metadata

This chapter discusses the use of the *OCIDescribeAny()* function to obtain information about schema elements.

The following topics are covered in this chapter:

- Overview
- Using *OCIDescribeAny()*
- Examples

Overview

This chapter deals with the use of the *OCIDescribeAny()* function to describe schema objects. For information about describing select-list items, refer to the section “Describing Select-List Items” on page 4-8.

For additional information about the *OCIDescribeAny()* call and its parameters, refer to the function description on page 13 - 57.

Using OCIDescribeAny()

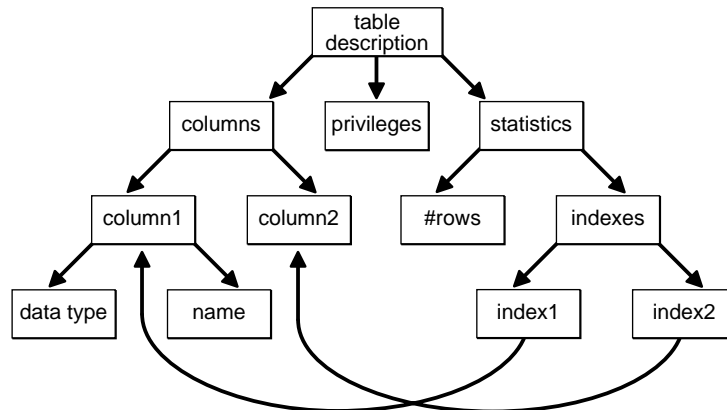
The *OCIDescribeAny()* function allows you to perform an explicit describe of one of the following schema objects:

- tables and views
- synonyms
- procedures
- functions
- packages
- sequences
- collections
- types

Information about other schema elements (procedure/function arguments, columns, type attributes, and type methods) is available through a describe of one of the above schema objects. For example, when an application describes a table, it can then retrieve information about that table’s columns.

The *OCIDescribeAny()* call requires a describe handle as one of its parameters. The describe handle must have been previously allocated with a call to *OCIHandleAlloc()*. After the call to *OCIDescribeAny()*, an application can retrieve information about the described object from the describe handle.

The information returned by *OCIDescribeAny()* is organized hierarchically like a tree. For example, the figure shows how description of a certain table might be organized:



The describe handle returned by *OCIDescribeAny()* points to such a tree of descriptions. Each node of the tree has attributes associated with the node and attributes (which are like recursive describe handles) that point to subtrees containing more information. If all the attributes are homogenous, as in case of elements of a list (e.g. column list), then we refer to them as *parameters*. In this document, we will use the terms *handle* and *parameter* interchangeably. The attributes associated with any node are returned by *OCIAttrGet()*, and the parameters are returned by *OCIParmGet()*.

For example, an *OCIAttrGet()* on the describe handle for the table can return a handle to the column-list information. An application can then use *OCIParmGet()* to retrieve the handle to the column description of a particular column in the column-list. The handle to the column descriptor can be passed to *OCIAttrGet()* to get further information about the column, such as the name and data type (as illustrated by following the left-hand side of the above figure).

No subsequent *OCIAttrGet()* or *OCIParmGet()* call requires extra round trips, as all the description is cached on the client side by *OCIDescribeAny()*.

Restrictions

The *OCIDescribeAny()* call limits information returned to the basic information and stops expanding a node if it amounts to another describe. For example, if a table column is of an object type, then the OCI does not return a subtree describing the type since this information can be obtained by another describe.

For similar reasons, the OCI also does not allow describes on columns, arguments, or fields of tables, views, functions, procedures, or types. Such information can be obtained by describing the top-level object containing it.

Note on Datatype Codes

For more information about typecodes (e.g., the OCI_TYPECODE values returned in the OCI_ATTR_TYPECODE attribute, and the SQLT typecodes returned in the OCI_ATTR_DATA_TYPE attribute), refer to the section “Typecodes” on page 3-24.

OCI_ATTR_TYPECODE returns typecodes which represent the types supplied by the user when a new type is created (using the CREATE TYPE statement). These typecodes are of the enumerated type **OCITypeCode**, and are represented by OCI_TYPECODE constants. Internal PL/SQL types (boolean, indexed table) are not supported.

OCI_ATTR_DATA_TYPE returns typecodes which represent the datatypes stored in database columns. These are similar to the describe values returned by previous versions of Oracle. These values are represented by SQLT constants (**ub2** values). BOOLEAN types return SQLT_BOL.

Note on Describing Types

In order to describe type objects, it is necessary to initialize the OCI process in object mode:

```
/* Initialize the OCI Process */
if (OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,
                 (dvoid * (*)(dvoid *, size_t)) 0,
                 (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                 (void (*)(dvoid *, dvoid *)) 0 ))
{ (void) printf("FAILED: OCIInitialize()\n");
  return OCI_ERROR; }
```

For more information on this function, refer to the description of *OCIInitialize()* on page 13-72.

Note on OCI_ATTR_LIST_ARGUMENTS

The OCI_ATTR_LIST_ARGUMENTS attribute for type methods represents “second-level” arguments for the method.

For example, given the following record `my_type` and the procedure `my_proc` which takes an argument of type `my_type`:

```
my_rec record(a number, b char)
my_proc (my_input my_rec)
```

the OCI_ATTR_LIST_ARGUMENTS attribute would apply to arguments `a` and `b` of the `my_type` record.

Parameter Attributes

A parameter is returned by *OCIParamGet()*. Parameters can describe different types of objects or information. Hence, parameters have attributes depending on the type of description they contain — these are the type-specific attributes. This section describes the attributes and handles that belong to different parameters.

The following table lists the attributes that belong to all parameters:

Table 6–1 Attributes Belonging to All Parameters

Attribute	Description	Attribute Datatype
OCI_ATTR_PTYPE	type of information described by the parameter. Possible values are: OCI_PTYPE_TABLE - table OCI_PTYPE_VIEW - view OCI_PTYPE_PROC - procedure OCI_PTYPE_FUNC - function OCI_PTYPE_PKG - package OCI_PTYPE_TYPE - type OCI_PTYPE_TYPE_ATTR - attribute of a type OCI_PTYPE_TYPE_COLL - collection type information OCI_PTYPE_TYPE_METHOD - a method of a type OCI_PTYPE_SYN - synonym OCI_PTYPE_SEQ - sequence OCI_PTYPE_COL - column of a table or view OCI_PTYPE_ARG - argument of a function or procedure OCI_PTYPE_TYPE_ARG - argument of a type method OCI_PTYPE_TYPE_RESULT - the results of a method OCI_PTYPE_LIST - column list for tables and views, argument list for functions and procedures, or subprogram list for packages.	ub1
OCI_ATTR_TIMESTAMP	the timestamp of the object this description is based on (in Oracle date format)	ub1 *
OCI_ATTR_NUM_ATTRS	the number of attributes	ub2
OCI_ATTR_NUM_PARAMS	the number of parameters	ub2

The subsections that follow list the attributes and handles specific to different types of parameters.

Table/View Attributes

When a parameter is for a table or view (type OCI_PTYPE_TABLE or OCI_PTYPE_VIEW), it has the following type specific attributes:

Table 6–2 Attributes Belonging to Tables or Views

Attribute	Description	Attribute Datatype
OCI_ATTR_OBJID	object id	ub4
OCI_ATTR_NUM_COLS	number of columns	ub2
OCI_ATTR_LIST_COLUMNS	column list (type OCI_PTYPE_LIST)	dvoid *

The following are additional attributes which belong to tables:

Table 6–3 Attributes Specific to Tables

Attribute	Description	Attribute Datatype
OCI_ATTR_RDBA	data block address of the segment header	ub4
OCI_ATTR_TABLESPACE	tablespace the table resides in	word
OCI_ATTR_CLUSTERED	whether the table is clustered	ub1
OCI_ATTR_PARTITIONED	whether the table is partitioned	ub1
OCI_ATTR_INDEX_ONLY	whether the table is index only	ub1

Procedure/Function Attributes

When a parameter is for a procedure or function (type OCI_PTYPE_PROC or OCI_PTYPE_FUNC), it has the following type specific attributes:

Table 6–4 Attribute Belonging to Procedures or Functions

Attribute	Description	Attribute Datatype
OCI_ATTR_LIST_ARGUMENTS	argument list. See “List Attributes” on page 6-19.	dvoid *

The following attributes are defined only for package subprograms:

Table 6–5 Attributes Specific to Package Subprograms

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	name of the procedure or function	text *
OCI_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.	ub2

Package Attributes

When a parameter is for a package (type OCI_PTYPE_PKG), it has the following type specific attributes:

Table 6–6 Attributes Belonging to Packages

Attribute	Description	Attribute Datatype
OCI_ATTR_LIST_SUBPROGRAMS	subprogram list. See “List Attributes” on page 6-19.	dvoid *

Type Attributes

When a parameter is for a type (type `OCI_PTYPE_TYPE`), it has the attributes listed in Table 6–7. These attributes are only valid if the application initialized the OCI process in `OCI_OBJECT` mode in a call to `OCIInitialize()`.

Table 6–7 Attributes Belonging to Types

Attribute	Description	Attribute Datatype
<code>OCI_ATTR_REF_TDO</code>	returns the in-memory REF of the type descriptor object for the type, if the column type is an object type. If space has not been reserved for the OCISRef, then it is allocated implicitly in the cache. The caller can then pin the TDO with <code>OCIObjectPin()</code> .	OCISRef *
<code>OCI_ATTR_TYPECODE</code>	typecode. See “Note on Datatype Codes” on page 6-4. Currently can be only <code>OCI_TYPECODE_OBJECT</code> or <code>OCI_TYPECODE_NAMEDCOLLECTION</code> .	OCITypeCode
<code>OCI_ATTR_COLLECTION_TYPECODE</code>	typecode of collection if type is collection; invalid otherwise. See “Note on Datatype Codes” on page 6-4. Currently can be only <code>OCI_TYPECODE_VARRAY</code> or <code>OCI_TYPECODE_TABLE</code> . Error is returned if this attribute is queried for non-collection type.	OCITypeCode
<code>OCI_ATTR_VERSION</code>	a null terminated string containing the user-assigned version	text *
<code>OCI_ATTR_IS_INCOMPLETE_TYPE</code>	is this an incomplete type?	ub1
<code>OCI_ATTR_IS_SYSTEM_TYPE</code>	is this a system type?	ub1
<code>OCI_ATTR_IS_PREDEFINED_TYPE</code>	is this a predefined type?	ub1
<code>OCI_ATTR_IS_TRANSIENT_TYPE</code>	is this a transient type?	ub1
<code>OCI_ATTR_IS_SYSTEM_GENERATED_TYPE</code>	is this a system-generated type?	ub1
<code>OCI_ATTR_HAS_NESTED_TABLE</code>	does this type contain a nested table attribute?	ub1
<code>OCI_ATTR_HAS_LOB</code>	does this type contain a LOB attribute?	ub1
<code>OCI_ATTR_HAS_FILE</code>	does this type contain a FILE attribute?	ub1

Table 6–7 Attributes Belonging to Types (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_COLLECTION_ELEMENT	handle to collection element. See “Collection Attributes” on page 6-13.	dvoid *
OCI_ATTR_NUM_TYPE_ATTRS	number of type attributes	ub4
OCI_ATTR_LIST_TYPE_ATTRS	list of type attributes. See “List Attributes” on page 6-19.	dvoid *
OCI_ATTR_NUM_TYPE_METHODS	number of type methods	ub4
OCI_ATTR_LIST_TYPE_METHODS	list of type methods. See “List Attributes” on page 6-19.	dvoid *
OCI_ATTR_MAP_METHOD	map method of type. See “Type Method Attributes” on page 6-12.	dvoid *
OCI_ATTR_ORDER_METHOD	order method of type. See “Type Method Attributes” on page 6-12.	dvoid *

Type Attribute Attributes

When a parameter is for an attribute of a type (type OCI_PTYPE_TYPE_ATTR), it has the attributes listed in Table 6–8.

Table 6–8 Attributes Belonging to Type Attributes

Attribute	Description	Attribute Datatype
OCI_ATTR_DATA_SIZE	the maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERS.	ub2
OCI_ATTR_TYPECODE	typecode. See “Note on Datatype Codes” on page 6-4.	OCITypeCode
OCI_ATTR_DATA_TYPE	the data type of the type attribute. See “Note on Datatype Codes” on page 6-4.	ub2
OCI_ATTR_NAME	a pointer to a string which is the type attribute name	text *

Table 6–8 Attributes Belonging to Type Attributes (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_PRECISION	the precision of numeric type attributes. If a describe returns a value of zero for precision or -127 for scale, this indicates that the item being described is uninitialized; i.e., it is NULL in the data dictionary.	ub1
OCI_ATTR_SCALE	the scale of numeric type attributes. If a describe returns a value of zero for precision or -127 for scale, this indicates that the item being described is uninitialized; i.e., it is NULL in the data dictionary.	sb1
OCI_ATTR_TYPE_NAME	a string which is the type name. The returned value will contain the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, the name of the named data type's type is returned. If the data type is SQLT_REF, the type name of the named data type pointed to by the REF is returned	text *
OCI_ATTR_SCHEMA_NAME	a string with the schema name under which the type has been created	text *
OCI_ATTR_REF_TDO	returns the in-memory REF of the TDO for the type, if the column type is an object type. If space has not been reserved for the OCIRef, then it is allocated implicitly in the cache. The caller can then pin the TDO with <i>OCIObjectPin()</i> .	OCIRef *
OCI_ATTR_CHARSET_ID	the character set id, if the type attribute is of a string/character type	ub2
OCI_ATTR_CHARSET_FORM	the character set form, if the type attribute is of a string/character type	ub1

Type Method Attributes

When a parameter is for a method of a type (type OCI_PTYPE_TYPE_METHOD), it has the attributes listed in Table 6–9.

Table 6–9 Attributes Belonging to Type Methods

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	name of method (procedure or function)	text *
OCI_ATTR_ENCAPSULATION	encapsulation level of the method (either OCI_TYPEENCAP_PRIVATE or OCI_TYPEENCAP_PUBLIC)	OCITYPEEncap
OCI_ATTR_LIST_ARGUMENTS	argument list. See “Note on OCI_ATTR_LIST_ARGUMENTS” on page 6-5, and “List Attributes” on page 6-19.	dvoid *
OCI_ATTR_IS_CONSTRUCTOR	is method a constructor?	ub1
OCI_ATTR_IS_DESTRUCTOR	is method a destructor?	ub1
OCI_ATTR_IS_OPERATOR	is method an operator?	ub1
OCI_ATTR_IS_SELFISH	is method selfish?	ub1
OCI_ATTR_IS_MAP	is method a map method?	ub1
OCI_ATTR_IS_ORDER	is method an order method?	ub1
OCI_ATTR_IS_RNDS	is “Read No Data State” set for method?	ub1
OCI_ATTR_IS_RNPS	is “Read No Process State” set for method?	ub1
OCI_ATTR_IS_WNDS	is “Write No Data State” set for method?	ub1
OCI_ATTR_IS_WNPS	is “Write No Process State” set for method?	ub1

As a reference, the following code shows the possible method flags which are used when determining the corresponding procedure/function attributes:

```

OCITYPEMethodFlag
{
    OCI_TYPEMETHOD_INLINE = 0x0001,           /* inline */
    OCI_TYPEMETHOD_CONSTANT = 0x0002,        /* constant */
    OCI_TYPEMETHOD_VIRTUAL = 0x0004,         /* virtual */
    OCI_TYPEMETHOD_CONSTRUCTOR = 0x0008,     /* constructor */
    OCI_TYPEMETHOD_DESTRUCTOR = 0x0010,     /* destructor */
    OCI_TYPEMETHOD_OPERATOR = 0x0020,       /* operator */
    OCI_TYPEMETHOD_SELFISH = 0x0040, /* selfish method (generic otherwise) */

```

```

OCI_TYPEMETHOD_MAP = 0x0080,          /* map (relative ordering) */
OCI_TYPEMETHOD_ORDER = 0x0100,        /* order (relative ordering) */
/* OCI_TYPEMETHOD_MAP and OCI_TYPEMETHOD_ORDER are mutually exclusive */

OCI_TYPEMETHOD_RNDS= 0x0200,          /* Read no Data State (default) */
OCI_TYPEMETHOD_WNDS= 0x0400,          /* Write no Data State */
OCI_TYPEMETHOD_RNPS= 0x0800,          /* Read no Process State */
OCI_TYPEMETHOD_WNPS= 0x1000          /* Write no Process State */ }

```

Collection Attributes

When a parameter is for a collection type (type OCI_PTYPE_COLL), it has the attributes listed in Table 6–10.

Table 6–10 Attributes Belonging to Collection Types

Attribute	Description	Attribute Datatype
OCI_ATTR_DATA_SIZE	the maximum size of the type attribute. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERS.	ub2
OCI_ATTR_TYPECODE	typecode. See “Note on Datatype Codes” on page 6-4.	OCITypeCode
OCI_ATTR_DATA_TYPE	the data type of the type attribute. See “Note on Datatype Codes” on page 6-4.	ub2
OCI_ATTR_NUM_ELEMENTS	the number of elements in an array. It is only valid for collections that are arrays	ub4
OCI_ATTR_NAME	a pointer to a string which is the type attribute name	text *
OCI_ATTR_PRECISION	the precision of numeric type attributes. If a describe returns a value of zero for precision or -127 for scale, this indicates that the item being described is uninitialized; i.e., it is NULL in the data dictionary.	ub1
OCI_ATTR_SCALE	the scale of numeric type attributes. If a describe returns a value of zero for precision or -127 for scale, this indicates that the item being described is uninitialized; i.e., it is NULL in the data dictionary.	sb1

Table 6–10 Attributes Belonging to Collection Types (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_TYPE_NAME	a string which is the type name. The returned value will contain the type name if the data type is <code>SQLT_NTY</code> or <code>SQLT_REF</code> . If the data type is <code>SQLT_NTY</code> , the name of the named data type's type is returned. If the data type is <code>SQLT_REF</code> , the type name of the named data type pointed to by the REF is returned	text *
OCI_ATTR_SCHEMA_NAME	a string with the schema name under which the type has been created	text *
OCI_ATTR_REF_TDO	returns the in-memory REF of the TDO for the type, if the column type is an object type. If space has not been reserved for the OCIRef, then it is allocated implicitly in the cache. The caller can then pin the TDO with <i>OCIObjectPin()</i> .	OCIRef *
OCI_ATTR_CHARSET_ID	the character set id, if the type attribute is of a string/character type	ub2
OCI_ATTR_CHARSET_FORM	the character set form, if the type attribute is of a string/character type	ub1

Synonym Attributes

When a parameter is for a synonym (type `OCI_PTYPE_SYN`), it has the attributes listed in Table 6–11.

Table 6–11 Attributes Belonging to Synonyms

Attribute	Description	Attribute Datatype
OCI_ATTR_OBJID	object id	ub4
OCI_ATTR_SCHEMA	a null-terminated string containing the schema name of the synonym translation	text *
OCI_ATTR_NAME	a null-terminated string containing the object name of the synonym translation	text *
OCI_ATTR_LINK	a null-terminated string containing the database link name of the synonym translation	text *

Sequence Attributes

When a parameter is for a sequence (type OCI_PTYPE_SEQ), it has the attributes listed in Table 6–12.

Table 6–12 Attributes Belonging to Sequences

Attribute	Description	Attribute Datatype
OCI_ATTR_OBJID	object id	ub4
OCI_ATTR_MIN	minimum value (in Oracle number format)	ub1 *
OCI_ATTR_MAX	maximum value (in Oracle number format)	ub1 *
OCI_ATTR_INCR	increment (in Oracle number format)	ub1 *
OCI_ATTR_CACHE	number of sequence numbers cached; zero if the sequence is not a cached sequence (in Oracle number format)	ub1 *
OCI_ATTR_ORDER	whether the sequence is ordered	ub1
OCI_ATTR_HW_MARK	high-water mark (in Oracle number format)	ub1 *

Column Attributes

When a parameter is for a column of a table or view (type OCI_PTYPE_COL), it has the attributes listed in Table 6–13.

Table 6–13 Attributes Belonging to Columns of Tables or Views

Attribute	Description	Attribute Datatype
OCI_ATTR_DATA_SIZE	the maximum size of the column. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERS.	ub2
OCI_ATTR_DATA_TYPE	the data type of the column. See “Note on Datatype Codes” on page 6-4.	ub2
OCI_ATTR_NAME	a pointer to a string which is the column name	text *

Table 6–13 Attributes Belonging to Columns of Tables or Views (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_PRECISION	the precision of numeric columns. If a describe returns a value of zero for precision or -127 for scale, this indicates that the item being described is uninitialized; i.e., it is NULL in the data dictionary.	ub1
OCI_ATTR_SCALE	the scale of numeric columns. If a describe returns a value of zero for precision or -127 for scale, this indicates that the item being described is uninitialized; i.e., it is NULL in the data dictionary.	sb1
OCI_ATTR_IS_NULL	returns 0 if null values are not permitted for the column	ub1
OCI_ATTR_TYPE_NAME	returns a string which is the type name. The returned value will contain the type name if the data type is <code>SQLT_NTY</code> or <code>SQLT_REF</code> . If the data type is <code>SQLT_NTY</code> , the name of the named data type's type is returned. If the data type is <code>SQLT_REF</code> , the type name of the named data type pointed to by the REF is returned	text *
OCI_ATTR_SCHEMA_NAME	returns a string with the schema name under which the type has been created	text *
OCI_ATTR_REF_TDO	the REF of the TDO for the type, if the column type is an object type	OCISRef *
OCI_ATTR_CHARSET_ID	the character set id, if the column is of a string/character type	ub2
OCI_ATTR_CHARSET_FORM	the character set form, if the column is of a string/character type	ub1

Argument/Result Attributes

When a parameter is for an argument of a procedure/function (type OCI_PTYPE_ARG), for a type method argument (type OCI_PTYPE_TYPE_ARG) or for method results (type OCI_PTYPE_TYPE_RESULT), it has the attributes listed in Table 6–14.

Table 6–14 Attributes Belonging to Arguments/Results

Attribute	Description	Attribute Datatype
OCI_ATTR_NAME	returns a pointer to a string which is the argument name	text *
OCI_ATTR_POSITION	the position of the argument in the argument list. Always returns zero.	ub2
OCI_ATTR_TYPECODE	typecode. See “Note on Datatype Codes” on page 6-4.	OCITypeCode
OCI_ATTR_DATA_TYPE	the data type of the argument. See “Note on Datatype Codes” on page 6-4.	ub2
OCI_ATTR_DATA_SIZE	the size of the data type of the argument. This length is returned in bytes and not characters for strings and raws. It returns 22 for NUMBERS.	ub2
OCI_ATTR_PRECISION	the precision of numeric arguments. If a describe returns a value of zero for precision or -127 for scale, this indicates that the item being described is uninitialized; i.e., it is NULL in the data dictionary.	ub1
OCI_ATTR_SCALE	the scale of numeric arguments. If a describe returns a value of zero for precision or -127 for scale, this indicates that the item being described is uninitialized; i.e., it is NULL in the data dictionary.	sb1
OCI_ATTR_LEVEL	the data type levels. This attribute always returns zero.	ub2
OCI_ATTR_HAS_DEFAULT	indicates whether an argument has a default	ub1
OCI_ATTR_LIST_ARGUMENTS	the list of arguments at the next level (when the argument is of a record or table type).	dvoid *

Table 6–14 Attributes Belonging to Arguments/Results (Cont.)

Attribute	Description	Attribute Datatype
OCI_ATTR_IOMODE	indicates the argument mode: 0 is IN (OCI_TYPEPARAM_IN), 1 is OUT (OCI_TYPEPARAM_OUT), 2 is IN/OUT (OCI_TYPEPARAM_INOUT)	OCITypeParamMode
OCI_ATTR_RADIX	returns a radix (if number type)	ub1
OCI_ATTR_IS_NULL	returns 0 if null values are not permitted for the column	ub1
OCI_ATTR_TYPE_NAME	returns a string which is the type name, or the package name in the case of package local types. The returned value will contain the type name if the data type is SQLT_NTY or SQLT_REF. If the data type is SQLT_NTY, the name of the named data type's type is returned. If the data type is SQLT_REF, the type name of the named data type pointed to by the REF is returned.	text *
OCI_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	text *
OCI_ATTR_SUB_NAME	for SQLT_NTY or SQLT_REF, returns a string with the type name, in the case of package local types	text *
OCI_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.	text *
OCI_ATTR_REF_TDO	returns the REF of the TDO for the type, if the argument type is an object	OCIRef *
OCI_ATTR_CHARSET_ID	returns the character set ID if the argument is of a string/character type	ub2
OCI_ATTR_CHARSET_FORM	returns the character set form if the argument is of a string/character type	ub1

List Attributes

When a parameter is for a list of columns, arguments, or subprograms (type `OCI_PTYPE_LIST`), it has the following type specific attributes and handles (parameters):

- The list has an `OCI_ATTR_LIST_TYPE` attribute which designates the list type. The possible values are:
 - `OCI_LTYPE_COL` - column list
 - `OCI_LTYPE_ARG_PROC` - procedure argument list
 - `OCI_LTYPE_ARG_FUNC` - function argument list
 - `OCI_LTYPE_SUBPRG` - subprogram list
 - `OCI_LTYPE_TYPE_ATTR` - type attribute list
 - `OCI_LTYPE_TYPE_METHOD` - type method list
 - `OCI_LTYPE_TYPE_ARG_PROC` - type method without result argument list
 - `OCI_LTYPE_TYPE_ARG_FUNC` - type method without result argument list
- The list has an `OCI_ATTR_NUM_PARAMS` attribute, which tells the number of elements in the list.
- The list has 1..`OCI_ATTR_NUM_PARAMS` parameters for each of the columns, arguments, or subprograms in the list (type `OCI_PTYPE_COL`, `OCI_PTYPE_ARG`, `OCI_PTYPE_PROC`, or `OCI_PTYPE_FUNC`). In the case of a function argument list, position 0 has a parameter for the return value (type `OCI_PTYPE_ARG`).

Examples

The following examples demonstrate the use of *OCIDescribeAny()* for describing different types of schema objects. For a more detailed code sample, refer to “Example 4, Describing an Object” on page D-55.

Retrieving column data types for a table

This example illustrates the use of an explicit describe. Let us take an example application, which needs to retrieve the column datatypes for a table. The following pseudo-code shows how an application would be able to use the describe interface:

```
text objptr[] = <table-name>;
ub4 objp_len = strlen(<table_name>);
OCIParm *parmh;          /* parameter handle */
OCIParm *collsthd;       /* handle to list of columns */
OCIParm *colhd;          /* column handle */

/* get the describe handle for the table */
if (OCIDescribeAny(svch, errh, objptr, objp_len, OCI_OTYPE_NAME, 0,
    OCI_PTYPE_TABLE, &dschp))
    return error;
/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
    errh))
    return error;
/* The type information of the object, in this case, OCI_PTYPE_TABLE,
is obtained from the parameter descriptor returned by the OCIAttrGet */
/* get the number of columns in the table */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &numcols, 0,
    OCI_ATTR_NUM_COLS, errh))
    return error;
/* get the handle to the column list of the table */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &collsthd, 0,
    OCI_ATTR_LIST_COLUMNS, errh)==OCI_NO_DATA)
    return error;
/* go through the column list and retrieve the data-type of each column,
and then recursively describe column types. */

for (i = 1; i <= numcols; i++)
{
    /* get parameter for column i */
    if (OCIParmGet(collsthd, OCI_DTYPE_PARAM, errh, &colhd, i))
        return error;
```

```

    /* for example, get data type for ith column */
    if (OCIAttrGet(colhd, OCI_DTYPE_PARAM, &datatype[i-1], 0,
        OCI_ATTR_DATA_TYPE, errh))
        return error;
}

```

Describing the stored procedure

Let us consider a stored procedure or a function. The difference between a procedure and a function is that the latter has a return type at position 0 in the argument list, while the former has no argument associated with position 0 in the argument list. The steps required to describe type methods (also divided into functions and procedures) are identical to that of regular PL/SQL functions and procedures. Note that procedures/functions can take default types of objects as arguments. Let us consider the following procedure:

```
P1 (arg1 emp.sal%type, arg2 emp%rowtype)
```

Furthermore, let us assume that each row in emp table has two columns name (VARCHAR2(20)), and sal (NUMBER). Thus, in the argument list for P1, we have two arguments, arg1 and arg2, at positions 1 and 2 respectively at level 0, and arguments name and sal at positions 1 and 2 respectively at level 1. Description of P1 returns the number of arguments as two while returning the higher level (> 0) arguments as attributes of the 0 zero level arguments.

The following pseudocode elucidates the description of P1.

```

text objptr[] = "P1";      /* procedure name */
ub4 objp_len = strlen("P1");
OCIPParam *parmh;          /* parameter handle */
OCIPParam *arglst;         /* list of args */
OCIPParam *arg;            /* argument handle */
ub2 numargs, pos, level;
text *name;
ub4 namelen;

/* get the describe handle for the table */
if (OCIDescribeAny(svch, errh, objptr, objp_len, OCI_OTYPE_NAME, 0,
    OCI_PTYPE_PROC, &dschp))
    return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
    errh))
    return error;

```

```
/* Get the number of arguments and the arg list */
if (OCIAttrGet (pamh, OCI_DTYPE_PARAM, &arglst,
0, OCI_ATTR_LIST_ARGUMENTS, errh))
    return error;
if (OCIAttrGet (pamh, OCI_DTYPE_PARAM, &numargs, 0,
    OCI_ATTR_NUM_PARAMS, errh))
    return error;

/* For a procedure, we begin with i = 1; for a
function, we begin with i = 0. */

for (i = 1; i < numargs; i++) {
    OCIParamGet (arglst, OCI_DTYPE_PARAM, errh, &arg, i);
    OCIAttrGet (arg, OCI_DTYPE_PARAM, &name, &namelen, OCI_ATTR_NAME,
        errh);
    ...
    /* to print the attributes of the argument of type record
    (arguments at the next level), traverse the argument list */

    OCIAttrGet (arg, OCI_DTYPE_PARAM, &arglst1, 0,
    OCI_ATTR_LIST_ARGUMENTS, errh);

    /* check if the current argument is a record. For arg1 in P1
    arglst1 is NULL. */

    if (arglst1) {
        OCIAttrGet (arg, OCI_DTYPE_PARAM, &numargs1, 0, OCI_ATTR_NUM_PARAMS,
            errh);

        /* Note that for both functions and procedures, the next higher level
        arguments start from index 1. For arg2 in P1, the number of arguments at
        the level 1 would be 2 */

        for (i = 1; i < numargs1, i++) {
            OCIParamGet (arglst1, OCI_DTYPE_PARAM, errh, &arg1, i);
            OCIAttrGet (arg1, OCI_DTYPE_PARAM, &name1, &namelen1,
                OCI_ATTR_NAME, errh);
            ...
        }
    }
}
```

Retrieving attributes of an object type

This example illustrates the use of an explicit describe on a named object type. We illustrate how you can describe an object by its name or by its object reference (**OCIRef**). The following pseudo-code attempts to retrieve the data type value each of the object type's attribute. It is very similar to the first example on page 6 - 20.

```

text type_name[] = <type_name>;
ub4 type_name_len = strlen(<type_name>);
OCIRef *type_ref = <type_ref>;
un4 numattrs;
OCIDescribe *dschp;      /* describe handle */
OCIParam *parmh;         /* parameter handle */
OCIParam *attrlsthd;      /* handle to list of attrs */
OCIParam *attrhd;        /* attribute handle */

/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
                  (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
    return error;

/* get the describe handle for the type */
if (describe_by_name)
    if (OCIDescribeAny(svch, errh, (dvoid*)type_name, type_name_len,
                      OCI_OTYPE_NAME, 0, OCI_PTYPE_TYPE, dschp))
        return error;
else
    if (OCIDescribeAny(svch, errh, (dvoid*)type_ref, 0, OCI_OTYPE_REF,
                      0, OCI_PTYPE_TYPE, dschp))
        return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
              errh))
    return error;

/* The type information of the object, in this case, OCI_PTYPE_TYPE, is
obtained from the parameter descriptor returned by the OCIAttrGet */

/* get the number of attributes in the type */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &numattrs, 0,
              OCI_ATTR_NUM_TYPE_ATTRS, errh))
    return error;

/* get the handle to the attribute list of the type */

```

```
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, (dvoid *)&attrlsthd, 0,
    OCI_ATTR_LIST_TYPE_ATTRS, errh)==OCI_NO_DATA)
    return error;

/* go through the attribute list and retrieve the data-type of each attribute,
and then recursively describe attribute types. */

for (i = 1; i <= numattrs; i++)
{
    /* get parameter for attribute i */
    if (OCIParamGet(attrlsthd, OCI_DTYPE_PARAM, errh, &attrhd, i))
        return error;

    /* for example, get data type and typecode for attribute; note that
OCI_ATTR_DATA_TYPE returns the SQLT code, while OCI_ATTR_TYPECODE returns the
Oracle Type System typecode. */
    if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,&datatype[i-1], 0,
        OCI_ATTR_DATA_TYPE,errh))
        return error;
    /* for example, get data type for attribute*/
    if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,&typecode[i-1], 0,
        OCI_ATTR_TYPECODE, errh))
        return error;

    /* if attribute is an object type, recursively describe it */
    if (typecode[i-1] == OCI_TYPECODE_OBJECT)
    {
        OCIRef *attr_type_ref;
        OCIDescribe *nested_dschp;

        /* allocate describe handle */
        if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
            (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
            return error;

        if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,
            &attr_type_ref, 0, OCI_ATTR_REF_TDO,errh))
            return error;
        OCIDescribeAny(svch, errh, (dvoid*)attr_type_ref, 0,
            OCI_OTYPE_REF, 0, OCI_PTYPE_TYPE, nested_dschp);
        /* go on describing the type... */
    }
}
```


Retrieving the collection element's data type of a named collection type

This example illustrates the use of an explicit describe on a named collection type. We illustrate how you can describe an object by its name or by its object reference (**OCIRef**). The following pseudo-code attempts to retrieve the data type value each of the object type's attribute. It is very similar to the first example on page 6 - 20.

```
text type_name[] = <type_name>;
ub4 type_name_len = strlen(<type_name>);
OCIRef *type_ref = <type_ref>;
un4 numattrs;
OCIDescribe *dschp;      /* describe handle */
OCIParam *parmh;         /* parameter handle */
OCIParam *attrlsthd;      /* handle to list of attrs */
OCIParam *attrhd;        /* attribute handle */

/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
                  (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
    return error;

/* get the describe handle for the type */
if (describe_by_name)
    if (OCIDescribeAny(svch, errh, (dvoid*)type_name, type_name_len,
                      OCI_OTYPE_NAME, 0, OCI_PTYPE_TYPE, dschp))
        return error;
else
    if (OCIDescribeAny(svch, errh, (dvoid*)type_ref, 0, OCI_OTYPE_REF, 0,
                      OCI_PTYPE_TYPE, &dschp))
        return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
              errh))
    return error;

/* get the Oracle Type System type code of the type to determine that this is a
collection type */
if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM, &typecode, 0, OCI_ATTR_TYPECODE,
              errh))
    return error;

/* if typecode is OCI_TYPECODE_NAMEDCOLLECTION,
   proceed to describe collection element */
if (typecode == OCI_TYPECODE_NAMEDCOLLECTION)
```

```
{
    /* get the collection's type: ie, OCI_TYPECODE_VARRAY or OCI_TYPECODE_TABLE */

    if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, (dvoid *)&collection_typecode, 0,
        OCI_ATTR_COLLECTION_TYPECODE, errh))
        return error;

    /* get the collection element; you MUST use this to further retrieve
    information about the collection's element */
    if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &collection_element_parmh, 0,
        OCI_ATTR_COLLECTION_ELEMENT, errh))
        return error;

    /* get the number of elements if collection is a VARRAY; not valid for nested
    tables */
    if (collection_typecode == OCI_TYPECODE_VARRAY)
        if (OCIAttrGet(collection_element_parmh, OCI_DTYPE_PARAM,
            (dvoid *)&num_elements, 0, OCI_ATTR_NUM_ELEMENTS, errh))
            return error;

    /* now use the collection_element parameter handle to retrieve information
    about the collection element */
    if (OCIAttrGet(collection_element_parmh, OCI_DTYPE_PARAM,
        (dvoid *)&element_typecode, 0, OCI_ATTR_TYPECODE, errh))
        return error;

    /* do the same to describe additional collection element information; this is
    very similar to describing type attributes */
}
```

OCI Programming Advanced Topics

The following topics are covered in this chapter:

- Overview
- Transactions
- User Authentication and Password Management
- Thread Safety
- Run Time Data Allocation and Piecewise Operations
- LOB and FILE Operations
- OCI Callbacks From External Procedures
- Application Failover Callbacks
- OCI and Advanced Queueing
- Writing Oracle Security Services Applications

Note: for information on using the OCI to manipulate objects in an Oracle8 server, see Chapter 8, “OCI Object-Relational Programming”.

Overview

Chapter 2 introduced the basic concepts of OCI programming. This chapter is designed to introduce more advanced concepts, including the following:

Transactions Chapter 2 described how a simple transaction can be committed or rolled back. This section talks about different levels of transaction complexity, including global transactions, and the operations that are possible through OCI calls.

User Authentication and Password Management Chapter 2 talked about the *OCISessionBegin()* call as part of OCI initialization. This section describes additional options available with *OCISessionBegin()*. It also describes user authentication and password management using the *OCIPasswordChange()* call.

Thread Safety This section describes OCI support for thread safety and multithreaded application development.

Run Time Data Allocation and Piecewise Operations Inserting, updating, and fetching data in a piecewise fashion is described in this section.

LOB and FILE Operations This section describes OCI functions available for operating on LOBs and FILES.

OCI Callbacks From External Procedures This section contains a pointer to information about writing external subroutines.

Application Failover Callbacks This section discusses how to write and use application failover callback functions.

OCI and Advanced Queueing This section covers the OCI functions related to Oracle8's Advanced Queueing feature.

Writing Oracle Security Services Applications This section contains a pointer to information on writing Oracle Security Services Applications.

Transactions

Release 8.0 of the Oracle Call Interface provides a set of API calls to support operations on both local and global transactions. These calls include object support, so that if an OCI application is running in object mode, the commit and rollback calls will synchronize the object cache with the state of the transaction.

The functions listed below perform transaction operations. Each call takes a service context handle that should be initialized with the proper server context and user session handle. The transaction handle is the third element of the service context; it stores specific information related to a transaction. When a SQL statement is prepared, it is associated with a particular service context. When the statement is executed, its effects (query, fetch, insert) become part of the transaction that is currently associated with the service context.

- *OCITransStart()* - marks the start of a transaction
- *OCITransDetach()* - detaches a transaction
- *OCITransCommit()* - commits a transaction
- *OCITransRollback()* - rolls back a transaction
- *OCITransPrepare()* - prepares a transaction to be committed in a distributed processing environment
- *OCITransForget()* - causes the server to forget a heuristically completed global transaction.

Depending on the level of transactional complexity in your application, you may need all or only a few of these calls. The following section discusses this in more detail.

See Also: For more specific information about these calls, refer to the function descriptions in Chapter 10.

Levels of Transactional Complexity

The OCI supports three levels of transaction complexity. Each level is described in one of the following sections.

1. Simple Local Transactions
2. Serializable or Read-Only Local Transactions
3. Global Transactions

Simple Local Transactions

Many applications work with only simple local transactions. In these applications, an implicit transaction is created when the application makes database changes. The only transaction-specific calls needed by such applications are:

- *OCITransCommit()* - to commit the transaction
- *OCITransRollback()* - to roll back the transaction

As soon as one transaction has been committed or rolled back, the next modification to the database creates a new implicit transaction for the application.

Only one implicit transaction can be active at any time on a service context. Attributes of the implicit transaction are opaque to the user.

If an application creates multiple authorizations, each one can have an implicit transaction associated with it.

For sample code showing the use of simple local transactions, refer to the example on page 13-150.

Serializable or Read-Only Local Transactions

Applications requiring serializable or read-only transactions require an additional OCI call beyond those needed by applications operating on simple local transactions. To initiate a serializable or read-only transactions, the application must create the transaction by calling *OCITransStart()* to start the transaction.

The call to *OCITransStart()* should specify `OCI_TRANS_SERIALIZABLE` or `OCI_TRANS_READONLY`, as appropriate, for the *flags* parameter. If no flag is specified, the default value is `OCI_TRANS_READWRITE` for a standard read-write transaction.

Specifying the read-only option in the *OCITransStart()* call saves the application from performing a server round-trip to execute a `SET TRANSACTION READ ONLY` statement.

Global Transactions

Global transactions are necessary only in more sophisticated transaction-processing applications.

Note: Users not operating in distributed or global transaction environments may skip this section.

This section provides some background about global transactions, and then gives specific information about using OCI calls to process global transactions.

Transaction Identifiers Three-tiered applications such as transaction processing (TP) monitors create and manage global transactions. They supply a *global transaction identifier* (XID), which a server then associates with a local transaction.

A global transaction has one or more *branches*. Each branch is identified by an XID. The XID consists of a *global transaction identifier* (gtrid) and a *branch qualifier* (bqual). This structure is based on the standard XA specification.

For example, the following is the structure for one possible XID of 1234:

Component	Value
gtrid	12
bqual	34
gtrid+bqual=XID	1234

See Also: For more information about transaction identifiers, refer to the *Oracle8 Distributed Database Systems* manual.

The transaction identifier used by OCI transaction calls is set in the OCI_ATTR_XID attribute of the transaction handle, using *OCIAttrSet()*. Alternately, the transaction can be identified by a name set in the OCI_ATTR_TRANS_NAME attribute.

Transaction Branches Within a single global transaction, Oracle8 supports both tightly coupled and loosely coupled relationships between a pair of branches.

- Tightly coupled branches are different branches that share the same local transaction. In this case, the *gtrid* references a unique local transaction, and multiple branches point to that same transaction. The owner of the transaction is the branch that was created first.
- Loosely coupled branches are different branches that use different local transactions. In this case the *gtrid* and *bqual* together map to a unique local transaction. Each branch points to a different transaction.

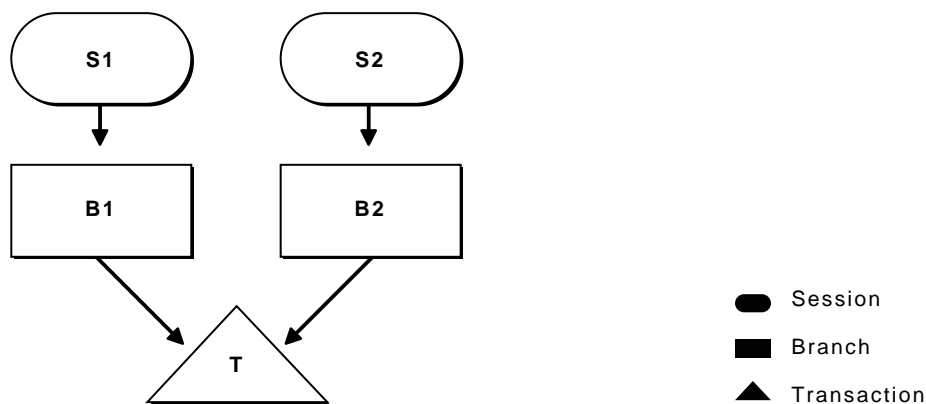
The *flags* parameter of *OCITransStart()* allows applications to pass OCI_TRANS_TIGHT or OCI_TRANS_LOOSE to specify the type of coupling.

In the Oracle8 OCI, a session corresponds to a user session, created with *OCISessionBegin()*.

The following figure illustrates tightly coupled branches within an application. In the figure, S1 and S2, are sessions, B1 and B2 are branches, and T is a transaction. In this first example, the XIDs of the two branches would share the same *gtrid*,

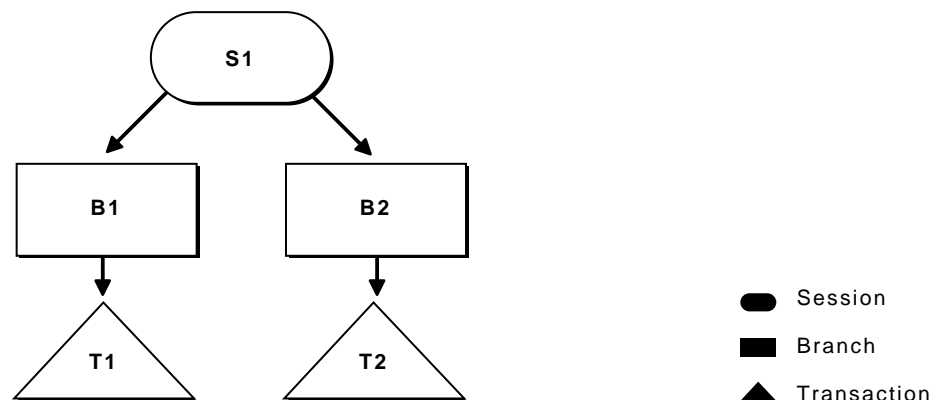
because they are operating on the same transaction, but they would have a different *bqual*, because they are separate branches

Figure 7-1 Multiple Tightly Coupled Branches



It is also possible for a single session to operate on different branches. In this case, illustrated in the next figure, *grid* component of the XIDs would be different, because they are separate global transactions

Figure 7-2 Session Operating on Multiple Branches

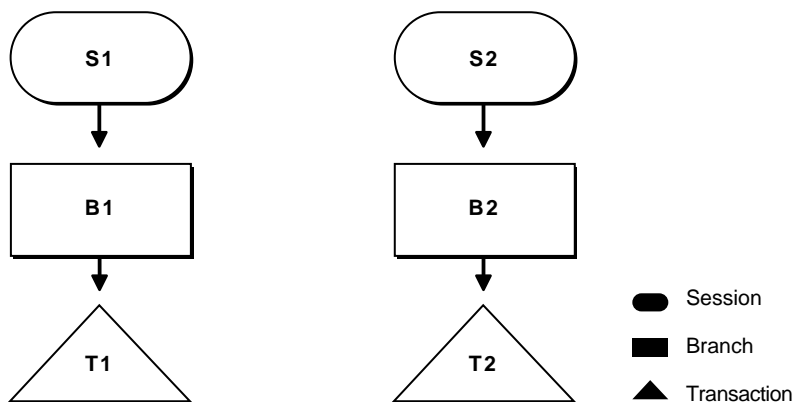


For sample code demonstrating this scenario, refer to the example on page 13-158.

It is possible for a single session to operate on multiple branches that share the same transaction, but this scenario does not have much practical value. Sample code demonstrating this scenario can be found in the example on page 13-161.

The following figure illustrates loosely coupled branches:

Figure 7-3 Loosely Coupled Branches



Branch States Transaction branches are classified into two states: *active branches* and *inactive branches*.

A branch is active if a server process is executing requests on the branch. A branch is inactive if no server processes are executing requests in the branch. In this case no session is the parent of the branch, and the branch becomes owned by the PMON process in the server.

Detaching and Resuming Branches A branch becomes inactive when an OCI application detaches it, using the `OCITransDetach()` call. The branch can be made active again by resuming it with a call to `OCITransStart()` with the *flags* parameter set to `OCI_TRANS_RESUME`.

When an application detaches a branch with `OCITransDetach()`, it utilizes the value specified in the *timeout* parameter of the `OCITransStart()` call that created the branch. The *timeout* specifies the number of seconds the transaction can remain dormant as a child of PMON before being deleted.

When an application wants to resume a branch, it calls `OCITransStart()`, specifying the XID of the branch as an attribute of the transaction handle, `OCI_TRANS_RESUME` for the *flags* parameter, and a different *timeout* parameter. This *timeout* value for this call specifies the length of time that the session will wait for the branch to become available if it is currently in use by another process. If no other processes are accessing the branch, it can be resumed immediately.

Note: A transaction can be resumed by a different process than the one that detached it, as long as that process has the same authorization as the one that detached the transaction.

Setting Client Database Name The server handle has `OCI_ATTR_EXTERNAL_NAME` and `OCI_ATTR_INTERNAL_NAME` attributes associated with it. These attributes set the client database name that will be recorded when performing global transactions. The name can be used by the DBA to track transactions that may be pending in a prepared state due to failures.

Warning: An OCI application should set these attributes, using `OCIAttrSet()`, before logging on and using global transactions.

One-Phase Versus Two-Phase Commit Global transactions may be committed in one or two phases. The simplest situation is when a single transaction is operating against a single database. In this case, the application can perform a one-phase commit of the transaction, by calling `OCITransCommit()`, because the default value of the call is for one-phase commit.

The situation is more complicated if the application is processing transactions against multiple databases or multiple Oracle servers. In this case, a two-phase commit is necessary. A two-phase commit consists of these steps:

1. **Prepare** - The application issues a prepare call, `OCITransPrepare()` against each transaction. The transaction returns a value indicating whether it is able to commit its current work (`OCI_SUCCESS`) or not (`OCI_ERROR`).
2. **Commit** - If each prepare call returns a value of `OCI_SUCCESS`, the application can issue a commit call, `OCITransCommit()` to each transaction. The *flags* parameter of the commit call must be explicitly set to `OCI_TRANS_TWOPHASE` for the appropriate behavior. The default for this call is for a one-phase commit.

Note: The prepare call can also return `OCI_SUCCESS_WITH_INFO` if a transaction needs to indicate that it is read-only, so that a commit is neither appropriate nor necessary.

An additional call, `OCITransForget()` indicates that a database should forget a heuristically completed transaction. This call is for situations in which a problem has occurred that requires that a two-phase commit be aborted. When a server receives a `OCITransForget()` call, it “forgets” all information about the transaction.

See Also: For more information about two-phase commit, refer to the *Oracle8 Distributed Database Systems* manual.

Transaction Examples

This section provides examples of how to use the transaction OCI calls. The following tables provide series of OCI calls and other actions, along with their resulting behavior. For the sake of simplicity, not all parameters to these calls are listed; rather, the flow of calls which is being demonstrated.

The “OCI Action” column indicates what the OCI application is doing, or what call it is making. The “XID” column lists the transaction identifier, when necessary. The “Flags” column lists the value(s) passed in the *flags* parameter. The “Result” column describes the result of the call.

Update Successfully, One-Phase Commit

Step	OCI Action	XID	Flags	Result
1	OCITransStart	1234	OCI_TRANS_NEW	Starts new read-write transaction
2	SQL UPDATE			Update rows
3	OCITransCommit			Commit succeeds

Start a Transaction, Detach, Resume, Prepare, Two-Phase Commit

Step	OCI Action	XID	Flags	Result
1	OCITransStart	1234	OCI_TRANS_NEW	Starts new read-only transaction
2	SQL UPDATE			Update rows
3	OCITransDetach			Transaction is detached
4	OCITransStart	1234	OCI_TRANS_RESUME	Transaction is resumed
5	SQL UPDATE			
6	OCITransPrepare			Transaction prepared for two-phase commit
7	OCITransCommit		OCI_TRANS_TWOPHASE	Transaction is committed.

Note: In step 4, above, the transaction could have been resumed by a different process, as long as it had the same authorization.

Read-Only Update Fails				
Step	OCI Action	XID	Flags	Result
1	OCITransStart	1234	OCI_TRANS_NEW OCI_TRANS_READONLY	Starts new read-only transaction
2	SQL UPDATE			Update fails, because transaction is read-only
3	OCITransCommit			Commit has no effect

Start a Read-Only Transaction, Select and Commit				
Step	OCI Action	XID	Flags	Result
1	OCITransStart	1234	OCI_TRANS_NEW OCI_TRANS_READONLY	Starts new read-only transaction
2	SQL SELECT			Query database
3	OCITransCommit			No effect — transaction is read-only, no changes made

Related Initialization Parameters

- Two initialization parameters relate to the use of global transaction branches and migratable open connections:
- TRANSACTIONS - This parameter specifies the maximum number of global transaction branches in the entire system. In contrast, MAX_TRANSACTION_BRANCHES specifies the number of branches on a single global transaction.
 - OPEN_LINKS_PER_INSTANCE - This parameter specifies the maximum number of migratable open connections. Migratable open connections are used by global transactions so that connections are cached after a transaction is committed. This is different from the OPEN_LINKS parameter, which is the number of connections from a section (and is not applicable to applications that use global transactions).

User Authentication and Password Management

Beginning with release 8.0, the OCI provides the ability to authenticate and maintain multiple users in an OCI application. There is also a new OCI call which allows the application to update a user's password. This is particularly helpful if an expired password message is returned by an authentication attempt.

Authentication

The *OCISessionBegin()* call is used to authenticate a user against the server set in the service context handle.

For Oracle8, *OCISessionBegin()* must be called for any given server handle before requests can be made against it. Also, *OCISessionBegin()* only supports authenticating the user for access to the Oracle server specified by the server handle in the service context. In other words, after *OCIServerAttach()* is called to initialize a server handle, *OCISessionBegin()* must be called to authenticate the user for that given server.

When *OCISessionBegin()* is called for the first time for a given server handle, the user session may not be created in migratable (OCI_MIGRATE) mode.

After *OCISessionBegin()* has been called for a server handle, the application may call *OCISessionBegin()* again to initialize another user session handle with different (or the same) credentials and different (or the same) operation modes. If an application wants to authenticate a user in OCI_MIGRATE mode, the service handle must already be associated with a non-migratable user handle. The user ID of that user handle becomes the ownership ID of the migratable user session. Every migratable session must have a non-migratable parent session.

If the OCI_MIGRATE mode is not specified, then the user session context can only ever be used with the same server handle set in *svchp*. If OCI_MIGRATE mode is specified, then the user authentication may be set with different server handles. However, the user session context may only be used with server handles which resolve to the same database instance. Security checking is done during session switching. A process or circuit is allowed to switch to a migratable session only if the ownership ID of the session matches the user ID of a non-migratable session currently connected to that same process or circuit, unless it is the creator of the session.

OCI_SYSDBA, OCI_SYSOPER, and OCI_PRELIM_AUTH may only be used with a primary user session context.

To provide credentials for a call to *OCISessionBegin()*, one of two methods are supported. The first is to provide a valid username and password pair for database

authentication in the user session handle passed to *OCISessionBegin()*. This involves using *OCIAttrSet()* to set the `OCI_ATTR_USERNAME` and `OCI_ATTR_PASSWORD` attributes on the user session handle. Then *OCISessionBegin()* is called with `OCI_CRED_RDBMS`.

Note: When the user session handle is terminated using *OCISessionEnd()*, the username and password attributes remain unchanged and thus can be re-used in a future call to *OCISessionBegin()*. Otherwise, they must be reset to new values before the next *OCISessionBegin()* call.

The second type of credentials supported are external credentials. No attributes need to be set on the user session handle before calling *OCISessionBegin()*. The credential type is `OCI_CRED_EXT`. This is equivalent to the Oracle7 'connect /' syntax. If values have been set for `OCI_ATTR_USERNAME` and `OCI_ATTR_PASSWORD`, then these are ignored if `OCI_CRED_EXT` is used.

Password Management

The release 8.0 OCI provides the *OCIPasswordChange()* to allow an OCI application to modify a user's database password as necessary. This is particularly useful if a call to *OCISessionBegin()* returns an error message or warning indicating that a user's password has expired.

Applications can also use *OCIPasswordChange()* to establish a user authentication context, as well as to change password, if appropriate flags are set. If *OCIPasswordChange()* is called with an uninitialized service context, it establishes a service context and authenticates the user's account using the old password, and then changes the password to the new password. If the `OCI_AUTH` flag is set, it leaves the user session initialized. Otherwise, the user session is cleared.

If the service context passed to *OCIPasswordChange()* is already initialized, then *OCIPasswordChange()* authenticates the given account using the old password and changes the password to the new password. In this case, no matter how the flag is set, the user session remains initialized.

Thread Safety

The thread safety feature of the Oracle8 server and OCI libraries allows developers to use the OCI in a multithreaded environment. With thread safety, OCI code can be reentrant, with multiple threads of a user program making OCI calls without side effects from one thread to another.

Note: Thread safety is not available on every platform. Check your Oracle system-specific documentation for more information.

The following sections describe how you can use the OCI to develop multithreaded applications.

Advantages of OCI Thread Safety

The implementation of thread safety in the Oracle Call Interface provides the following benefits and advantages:

- Multiple threads of execution can make OCI calls with the same result as successive calls made by a single thread.
- When multiple threads make OCI calls, there are no side effects between threads.
- Users who do not write multithreaded programs do not pay a performance penalty for using thread-safe OCI calls.
- Use of multiple threads can improve program performance. Gains may be seen 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 multithreaded program, a typical use of multithreaded 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 this scenario is an Oracle database. The applications server (agent) is very well suited to being a multithreaded application server, with each thread serving a client application. In an Oracle environment this application server is an OCI or precompiler program.

Basic Concepts of Multi-threaded Development

Threads are lightweight processes that exist within a larger process. Threads 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 OCI 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 the Oracle8 OCI, mutexes are granted on a per-environment-handle basis.

Implementing Thread Safety with OCI 8.0

In order to take advantage of thread safety in the Oracle8 OCI, an application must be running on a thread-safe platform. Then the application must tell the OCI layer that the application is running in multithreaded mode, by specifying `OCI_THREADED` for the *mode* parameter of the opening call to *OCIInitialize()*, which must be the first OCI function called in the application.

Note: Applications running on non-thread-safe platforms should not pass a value of `OCI_THREADED` to *OCIInitialize()*.

If an application is single-threaded, whether or not the platform is thread safe, the application should pass a value of `OCI_DEFAULT` to *OCIInitialize()*. Single-threaded applications which run in `OCI_THREADED` mode may incur performance hits.

If a multi-threaded application is running on a thread-safe platform, the OCI library will manage mutexing for the application on a per-environment-handle basis. If the application programmer desires, this application can override this feature and maintain its own mutexing scheme. This is done by specifying a value of `OCI_NO_MUTEX` to the *OCIEnvInit()* call.

The following three scenarios are possible, depending on how many connections exist per environment handle, and how many threads will be spawned per connection.

1. If an application has multiple environment handles, but each only has one thread (one session exists per environment handle), no mutexing is required.

2. If an application (running in OCI_THREADED mode) maintains multiple environment handles, each of which has one connection which can spawn multiple threads, the programmer has the following options:
 - Pass a value of OCI_NO_MUTEX for the mode of *OCIEnvInit()*. In this case the application must mutex OCI calls made on the same environment handle by itself. This has the advantage that the mutexing scheme can be optimized based on the application design. The programmer must also insure that only one OCI call is in process on the environment handle connection at any given time.
 - Pass a value of OCI_DEFAULT to *OCIEnvInit()*. In this case, the OCI library automatically gets a mutex on every OCI call on the environment handle.
3. If an application (running in OCI_THREADED mode) maintains one or more environment handles, each of which has multiple connections, it also has the following options:
 - Pass a value of OCI_NO_MUTEX for the mode of *OCIEnvInit()*. In this case the application must mutex OCI calls by made on the same environment handle itself. This has the advantage that the mutexing scheme can be optimized based on the application design. The programmer must also insure that only one OCI call is in process on the environment handle connection at any given time.
 - Pass a value of OCI_DEFAULT to *OCIEnvInit()*. In this case, the OCI library automatically gets a mutex on every OCI call on the same environment handle.

In this case, however, the programmer should be aware that if the application has two calls on the same environment handle, and one call operating on the server is mutexed, application performance can degrade if the mutexed call is long-running, thus tying up the server connection.

Mixing 7.x and 8.0 OCI calls

If an application is mixing 8.0 and 7.x OCI calls, and the application has been initialized as thread safe (with the appropriate 8.0 calls), it is not necessary to call *opinit()* to achieve thread safety. The application will get 7.x behavior on any subsequent 7.x function calls.

Run Time Data Allocation and Piecewise Operations

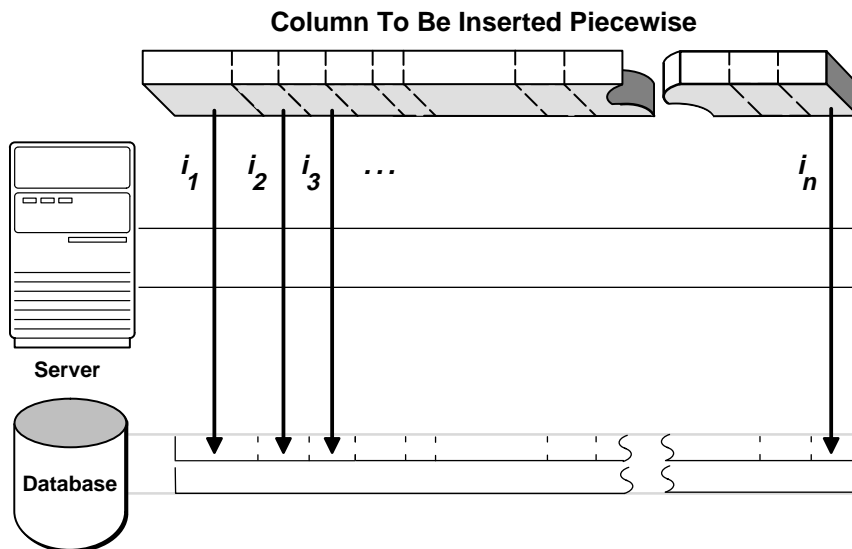
You can use the OCI to perform piecewise inserts and updates, and fetches of data. You can also use the OCI to provide data dynamically in the case of array inserts or updates, instead of providing a static array of bind values. You can insert or retrieve a very large column as a series of chunks of smaller size, minimizing client-side memory requirements.

The size of individual pieces is determined at run time by the application. Each piece may be of the same size as other pieces, or it may be of a different size.

The OCI's piecewise functionality can be particularly useful when you are performing operations on extremely large blocks of string or binary data (for example, operations involving database columns that store LOB, LONG or LONG RAW data). See the section “Valid Datatypes for Piecewise Operations” on page 7-17 for information about which datatypes are valid for piecewise operations.

Figure 2 - 8 shows a single long column being inserted piecewise into a database table through a series of insert operations (i_1 , i_2 , i_3 ... i_n). In this example the inserted pieces are of varying sizes.

Figure 7-4 Piecewise Insert of a LONG Column



You can perform piecewise operations in two ways:

- Use calls provided in the OCI library to execute piecewise operations under a polling paradigm, as in release 7.3.
- Employ user-defined callback functions to provide the necessary information and data blocks.

When you set the *mode* parameter of an *OCIBindByPos()* or *OCIBindByName()* call to *OCI_DATA_AT_EXEC*, this indicates that an OCI application will be providing data for an INSERT or UPDATE dynamically at run time.

Similarly, when you set the *mode* parameter of an *OCIDefineByPos()* call to *OCI_DYNAMIC_FETCH*, this indicates that an application will dynamically provide allocation space for receiving data at the time of the fetch.

In each case, you can provide the run-time information for the INSERT, UPDATE, or FETCH in one of two ways: through callback functions, or by using piecewise operations. If callbacks are desired, an additional bind or define call is necessary to register the callbacks.

The following sections give specific information about run-time data allocation and piecewise operations for inserts, updates, and fetches.

Note: In addition to SQL statements, piecewise operations are also valid for PL/SQL blocks.

Valid Datatypes for Piecewise Operations

Only some datatypes can be manipulated in pieces. OCI applications can perform piecewise fetches, inserts, or updates of the following data types:

- VARCHAR2
- STRING
- LONG
- LONG RAW

Some LOB/FILE operations also provide piecewise semantics for reading or writing data. See the descriptions of *OCILobWrite()* on page 13-112 and *OCILobRead()* on page 13-107 for more information about these operations.

Another way of using this feature for *all* datatypes is to provide data dynamically for array inserts or updates. Note, however, that the callbacks should always specify *OCI_ONE_PIECE* for the *piecep* parameter of the callback for datatypes that do not support piecewise operations.

Providing INSERT or UPDATE Data at Run Time

When you specify the `OCI_DATA_AT_EXEC` mode in a call to `OCIBindByPos()` or `OCIBindByName()`, the `value_sz` parameter defines the total size of the data that can be provided at run time. The application must be ready to provide to the OCI library the run-time IN data buffers on demand as many times as is necessary to complete the operation. When the allocated buffers are not required any more, they should be freed by the client.

Run-time data is provided in one of the two ways:

- You can define a callback using the `OCIBindDynamic()` function which when called at run time returns a piece or the whole data.
- If no callbacks are defined, the call to `OCISmtExecute()` to process the SQL statement returns the `OCI_NEED_DATA` error code. The client application then provides the IN/OUT data buffer or piece using the `OCISmtSetPieceInfo()` call. `OCISmtGetPieceInfo()` provides information about which bind and which piece are being used.

Performing a Piecewise Insert

Once the OCI environment has been initialized, and a database connection and session have been established, a piecewise insert begins with calls to prepare a SQL or PL/SQL statement and to bind input values. Piecewise operations using standard OCI calls, rather than user-defined callbacks, do not require a call to `OCIBindDynamic()`.

Note: Additional bind variables in the statement that are not part of piecewise operations may require additional bind calls, depending on their datatypes.

Following the statement preparation and bind, the application performs a series of calls to `OCISmtExecute()`, `OCISmtGetPieceInfo()` and `OCISmtSetPieceInfo()` to complete the piecewise operation. Each call to `OCISmtExecute()` returns a value that determines what action should be performed next. In general, the application retrieves a value indicating that the next piece needs to be inserted, populates a buffer with that piece, and then executes an insert. When the last piece has been inserted, the operation is complete.

Keep in mind that the insert buffer can be of arbitrary size and is provided at run time. In addition, each inserted piece does not need to be of the same size. The size of each piece to be inserted is established by each `OCISmtSetPieceInfo()` call.

Note: If the same piece size is used for all inserts, and the size of the data being inserted is not evenly divisible by the piece size, the final inserted piece will be smaller than the pieces that preceded it. For example, if a data value 10,050,036

bytes long is inserted in chunks of 500 bytes each, the last remaining piece will be only 36 bytes. The programmer must account for this by indicating the smaller size in the final *OCISmtSetPieceInfo()* call.

The following steps outline the procedure involved in performing a piecewise insert. The procedure is illustrated in on the following page.

Step 1. Initialize the OCI environment, allocate the necessary handles, connect to a server, authorize a user, and prepare a statement request. These steps are described in the section “OCI Programming Steps” on page 2-16.

Step 2. Bind a placeholder using *OCIBindByName()* or *OCIBindByPos()*. At this point you do not need to specify the actual size of the pieces you will use, but you must provide the total size of the data that can be provided at run time.

7.x Upgrade Note: The context pointer that was formerly part of the *obindps()* and *ogetpi()* routines does not exist in release 8.0. Clients wishing to provide their own context can use the callback method.

Step 3. Call *OCISmtExecute()* for the first time. At this point no data is actually inserted, and the OCI_NEED_DATA error code is returned to the application.

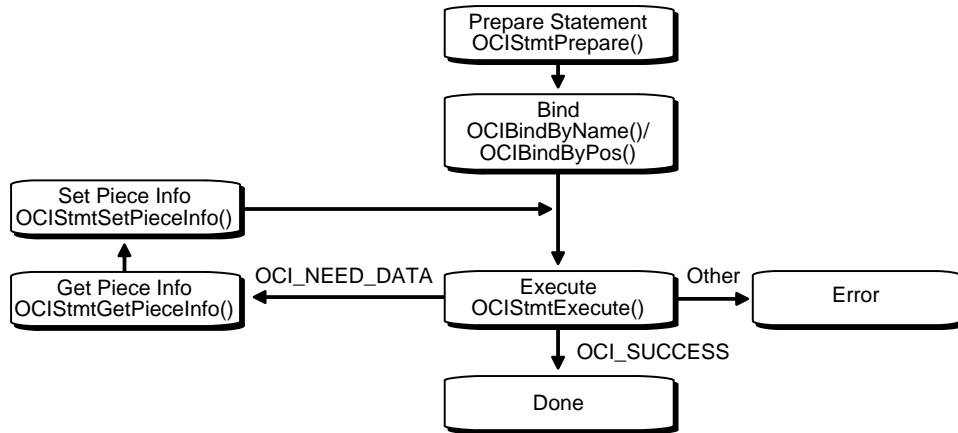
If any other value is returned, it indicates that an error occurred.

Step 4. Call *OCISmtGetPieceInfo()* to retrieve information about the piece that needs to be inserted. The parameters of *OCISmtGetPieceInfo()* include a pointer that returns a value indicating whether the required piece is the first piece (OCI_FIRST_PIECE) or a subsequent piece (OCI_NEXT_PIECE).

Step 5. The application populates a buffer with the piece of data to be inserted and calls *OCISmtSetPieceInfo()*. The parameters passed to *OCISmtSetPieceInfo()* include a pointer to the piece, a pointer to the length of the piece, and a value indicating whether this is the first piece (OCI_FIRST_PIECE), an intermediate piece (OCI_NEXT_PIECE) or the last piece (OCI_LAST_PIECE).

Step 6. Call *OCISmtExecute()* again. If OCI_LAST_PIECE was indicated in Step 5 and *OCISmtExecute()* returns OCI_SUCCESS, all pieces were inserted successfully. If *OCISmtExecute()* returns OCI_NEED_DATA, go back to Step 3 for the next insert. If *OCISmtExecute()* returns any other value, an error occurred.

The piecewise operation is complete when the final piece has been successfully inserted. This is indicated by the OCI_SUCCESS return value from the final *OCISmtExecute()* call.

Figure 7-5 Steps for Performing Piecewise Insert

Piecewise updates are performed in a similar manner. In a piecewise update operation the insert buffer is populated with the data that is being updated, and *OCIStmtExecute()* is called to execute the update.

Note: For additional important information about piecewise operations, see the section “Additional Information About Piecewise Operations with No Callbacks” on page 7-23.

Piecewise Operations With PL/SQL

An OCI application can perform piecewise operations with PL/SQL for IN, OUT, and IN/OUT bind variables in a method similar to that outlined above. Keep in mind that all placeholders in PL/SQL statements are bound, rather than defined. The call to *OCIBindDynamic()* specifies the appropriate callbacks for OUT or IN/OUT parameters.

Providing FETCH Information at Run Time

When a call is made to *OCIDefineByPos()* with the *mode* parameter set to *OCI_DYNAMIC_FETCH*, an application can specify information about the data buffer at the time of fetch. The user also may need to call *OCIDefineDynamic()* to set up the callback function that will be invoked to get information about the user’s data buffer.

Run-time data is provided in one of the two ways:

- You can define a callback using the *OCIDefineDynamic()* call. The *value_sz* parameter defines the maximum size of the data that will be provided at run

time. When the client library needs a buffer to return the fetched data, the callback will be invoked to provide a run-time buffer into which a piece or the whole data will be returned.

- If no callbacks are defined, the `OCI_NEED_DATA` error code is returned and the OUT data buffer or piece can then be provided by the client application using `OCISstmtSetPieceInfo()` call. The `OCISstmtGetPieceInfo()` call provides Information about which define and which piece are involved.

See Also: For information about which datatypes are valid for piecewise operations, refer to the section “Valid Datatypes for Piecewise Operations” on page 7-17.

Performing a Piecewise Fetch

Once the OCI environment has been initialized, and a database connection and session have been established, a piecewise fetch begins with calls to prepare a SQL or PL/SQL statement and to define output variables. Piecewise operations using standard OCI calls, rather than user-defined callbacks, do not require a call to `OCIDefineDynamic()`.

Following the statement preparation and define, the application performs a series of calls to `OCISstmtFetch()`, `OCISstmtGetPieceInfo()`, and `OCISstmtSetPieceInfo()` to complete the piecewise operation. Each call to `OCISstmtFetch()` returns a value that determines what action should be performed next. In general, the application retrieves a value indicating that the next piece needs to be fetched, and then fetches that piece into a buffer. When the last piece has been fetched, the operation is complete.

Keep in mind that the fetch buffer can be of arbitrary size. In addition, each fetched piece does not need to be of the same size. The only requirement is that the size of the final fetch must be exactly the size of the last remaining piece. The size of each piece to be fetched is established by each `OCISstmtSetPieceInfo()` call.

The following steps outline the method for fetching a row piecewise.

Step 1. Initialize the OCI environment, allocate necessary handles, connect to a database, authorize a user, prepare a statement, and execute the statement. These steps are described on page 2-16.

Step 2. Define an output variable using `OCIDefineByPos()`, with *mode* set to `OCI_DYNAMIC_FETCH`. At this point you do not need to specify the actual size of the pieces you will use, but you must provide the total size of the data that will be fetched at run time.

7.x Upgrade Note: The context pointer that was part of the *odefinps()* and *ogetpi()* routines does not exist in release 8.0. Clients wishing to provide their own context can use the callback method.

Step 3. Call *OCISstmtFetch()* for the first time. At this point no data is actually retrieved, and the OCI_NEED_DATA error code is returned to the application.

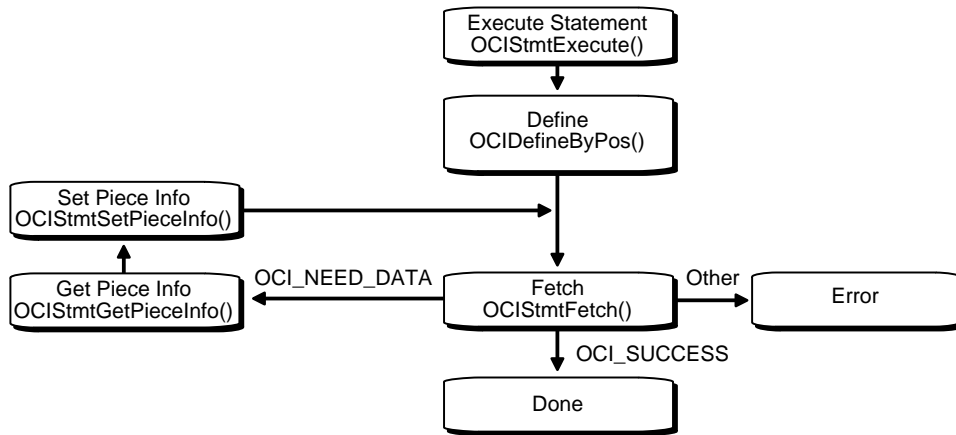
If any other value is returned, an error occurred.

Step 4. Call *OCISstmtGetPieceInfo()* to obtain information about the piece to be fetched. The *piecep* parameter indicates whether it is the first piece (OCI_FIRST_PIECE), a subsequent piece (OCI_NEXT_PIECE), or the last piece (OCI_LAST_PIECE).

Step 5. Call *OCISstmtSetPieceInfo()* to specify the buffer into which you wish to fetch the piece.

Step 6. Call *OCISstmtFetch()* again to retrieve the actual piece. If *OCISstmtFetch()* returns OCI_SUCCESS, all the pieces have been fetched successfully. If *OCISstmtFetch()* returns OCI_NEED_DATA, return to Step 4 to process the next piece. If any other value is returned, an error occurred.

The piecewise fetch is complete when the final *OCISstmtFetch()* call returns a value of OCI_SUCCESS.

Figure 7–6 Steps for Performing Piecewise Fetch

Additional Information About Piecewise Operations with No Callbacks

In both the piecewise fetch and insert, it is important to understand the sequence of calls necessary for the operation to complete successfully. In particular, keep in mind that for a piecewise insert you must call *OCIStmtExecute()* one time more than the number of pieces to be inserted (if callbacks are not used). This is because the first time *OCIStmtExecute()* is called, it merely returns a value indicating that the first piece to be inserted is required. As a result, if you are inserting *n* pieces, you must call *OCIStmtExecute()* a total of *n+1* times.

Similarly, when performing a piecewise fetch, you must call *OCIStmtFetch()* once more than the number of pieces to be fetched.

Users who are binding to PL/SQL tables can retrieve a pointer to the current index of the table during the *OCIStmtGetPieceInfo()* calls.

LOB and FILE Operations

The Oracle8 OCI includes a set of functions for performing operations on large objects (LOBs) in a database. *Internal* LOBs (BLOBs, CLOBs, NCLOBs) are stored in the database tablespaces in a way that optimizes space and provides efficient access. These LOBs have the full transactional support of the database server. External LOBs (FILEs) are large data objects stored in the server's operating system files outside the database tablespaces.

The maximum length of a LOB/FILE is 4 gigabytes.

FILE functionality is read-only. Oracle8 currently supports only binary files (BFILEs).

See Also: For code samples showing the use of LOB operations, refer to “Example 5, CLOB/BLOB Operations” on page D-76, and “Example 6, LOB Buffering” on page D-96.

Customers who are interested in using the `dbms_lob` package to work with LOBs should refer to the *Oracle8 Application Developer's Guide*

LOBs and LOB Locators

A database table stores a *LOB locator* which points to the LOB data. When an OCI application issues a SQL query that includes a LOB column in its select-list, fetching the result(s) of the query returns the locator, rather than the actual LOB value. In the OCI, the LOB locator maps to the datatype **OCILobLocator**.

Note: The LOB value can be stored inline in a database table if it is less than approximately 4,000 bytes.

Internal LOBs have copy semantics. Thus, if a LOB in one row is copied to a LOB in another row, the actual LOB value is copied, and a new LOB locator is created for the copied LOB.

The OCI functions for LOBs take LOB locators as their arguments. The OCI functions assume that the LOB to which the locator points has already been created, whether or not the LOB contains some value.

An application first fetches the locator using SQL, and then performs further operations using the locator. The OCI functions never take the actual LOB value as a parameter. It is good practice to use a locator in a LOB modification call if and only if its snapshot is recent enough that it sees the current value of the LOB data, since it is the current value that gets modified.

You allocate memory for an internal LOB locator with a call to *OCIDescriptorAlloc()* by passing `OCI_DTYPE_LOB` as the descriptor type. To allocate memory for an external LOB (FILE) locator, pass `OCI_DTYPE_FILE`.

Once you have allocated the LOB locator memory, you must initialize it before passing it to any OCI LOB routines. You can accomplish this by any of the following methods:

1. SELECTing the LOB from the database (which contains a valid LOB locator) into the LOB locator you have just allocated.
2. Using the locator in the RETURNING clause of a SQL INSERT or UPDATE statement.
3. Assigning a different, already initialized LOB locator to the newly allocated LOB locator.

You can also initialize a LOB locator to empty by calling *OCIAttrSet()* on the locator's `OCI_ATTR_LOBEMPTY` attribute. A locator initialized in this way may only be used to create an empty LOB in the database. Thus, it can only be used in the VALUES clause of a SQL INSERT statement, or as the source of the SET clause of a SQL UPDATE statement.

Warning: Locators for LOB and FILE operations are not interchangeable. Locators for LOB operations must be allocated as type `OCI_DTYPE_LOB`, and locators for FILE operations must be allocated as type `OCI_DTYPE_FILE`. An internal LOB locator may not be assigned to an external LOB (FILE) locator, and vice versa.

See Also: For more information about locators, including the LOB locator, see the section “Descriptors and Locators” on page 2-12.

For sample code showing the use of OCI LOB calls, refer to Example 3 in Appendix B, and the description of *OCILobWrite()* on page 13-112.

For more information about LOBs, locators, and read-consistent LOBs, see the *Oracle8 Application Developer's Guide*.

FILES

A FILE locator may be considered to be a pointer to a file on the server's file system. Oracle does not provide any transactional semantics on FILES, and Oracle8 currently supports only read-only operations on binary FILES (BFILES).

Since operations on both internal LOBs and FILES are similar, all OCI LOB/FILE functions expect a LOB locator as an input to all operations. The only difference is in the way the FILE locator is allocated. When allocating a locator for FILES, you must pass `OCI_DTYPE_FILE` as the descriptor type in the *OCIDescriptorAlloc()* call.

Warning: Locators for LOB and FILE operations are not interchangeable. Locators for LOB operations must be allocated as type `OCI_DTYPE_LOB`, and locators for FILE operations must be allocated as type `OCI_DTYPE_FILE`. An internal LOB locator may not be assigned to an external LOB (FILE) locator, and vice versa.

See Also: For information about associating a BFILE with an OS file, see the section "Associating a FILE in a Table with an OS File" on page 7-27.

Creating and Modifying Internal LOBs

You create a new internal LOB by initializing a new LOB locator using *OCIDescriptorAlloc()*, calling *OCIAttrSet()* to set it to empty (using the `OCI_ATTR_LOBEMPTY` attribute), and then binding the locator to a placeholder in an INSERT statement. Doing so inserts the empty locator into a table with a LOB column or attribute. You can then `SELECT...FOR UPDATE` this row to get the locator, and then write to it using one of the OCI LOB functions.

Note: Whenever you want to modify a 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 relock the row (by reissuing the `SELECT...FOR UPDATE`, for example), because the commit closes the transaction.

Note: LOB reads and writes are not allowed from within a trigger.

See Also: For information about binding LOB locators to placeholders, and using them in INSERT statements, refer to the section "Binding LOBs" on page 5-10.

Associating a FILE in a Table with an OS File

The `BFILENAME()` function can be used in an `INSERT` statement to associate an external server-side (OS) file with a `BFILE` column/attribute in a table. Using `BFILENAME()` in an `UPDATE` statement associates the `BFILE` column or attribute with a different OS file.

See Also: For more information about the `BFILENAME()` function, please refer to the *Oracle8 Application Developer's Guide*.

Writing to a LOB Attribute of an Object

It is possible to use the OCI to create a new persistent object with a LOB attribute and write to that LOB attribute. The application would follow these steps:

1. Call `OCIObjectNew()` to create a persistent object with a LOB attribute.
2. Mark the object as dirty.
3. Flush the object, thereby inserting a row into the table
4. Repin the latest version of the object (or refresh the object), thereby retrieving the object from the database and acquiring a valid locator for the LOB
5. Call `OCILobWrite()` using the LOB locator in the object to write the data.

For more information about object operations, such as marking, flushing, and refreshing, refer to Chapter 8, “OCI Object-Relational Programming”.

Transient Objects with LOB Attributes

An application can call `OCIObjectNew()` and create a transient object with an internal LOB (BLOB, CLOB, NCLOB) attribute. However, the user cannot perform any operations (e.g., read or write) on the LOB attribute because transient LOBs are not currently supported. Calling `OCIObjectNew()` to create a transient internal LOB type will 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 `OCIObjectNew()` to create a transient `FILE` and use that `FILE` to read from the server's file.

LOB Buffering

The Oracle8 OCI provides several calls for controlling LOB buffering for small reads and writes of internal LOB values:

- *OCILobEnableBuffering()*
- *OCILobDisableBuffering()*
- *OCILobFlushBuffer()*

These functions provide performance improvements by allowing applications using internal LOBs (BLOB, CLOB, NCLOB) to buffer small reads and writes of LOBs in client-side buffers. This reduces the number of network roundtrips and LOB versions, thereby improving LOB performance significantly for small reads and writes.

See Also: For more information on LOB buffering, refer to the chapter on LOBs in the *Oracle8 Application Developer's Guide*, and the LOB buffering code example in Appendix D of this guide.

For a code sample showing the use of LOB buffering, refer to “Example 6, LOB Buffering” on page D-96.

LOB/FILE Functions

The functions in Table 7–1 are available to operate on LOBs and FILES. More detailed information about each function is found in Chapter 13.

These LOB/FILE calls are not valid when an application is connected to an Oracle7 Server.

Note: In all LOB operations that involve offsets into the data, the offset begins at 1. BLOB and BFILE offsets and amounts are in terms of bytes. CLOB and NCLOB offsets and amounts are in terms of characters.

See Also: For more information about FILES, refer to the description of BFILES in the *Oracle8 Application Developer's Guide*.

Table 7–1 OCI LOB and FILE Functions

Function	Restrictions	Purpose
<i>OCILobAppend()</i>	Internal LOBs only	<p>This function appends data from one internal LOB onto another internal LOB. The source and the destination LOBs must already exist. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB.</p> <p>It is an error to extend the destination LOB beyond the maximum length allowed (4 gigabytes) or to try to append from a NULL LOB.</p>
<i>OCILobAssign()</i>		Assigns one LOB/FILE locator to another.
<i>OCILobCharSetForm()</i>		Gets the character set form of a CLOB/NCLOB.
<i>OCILobCharSetId()</i>		Gets the character set ID of a CLOB/NCLOB.
<i>OCILobCopy()</i>	Internal LOBs only	<p>This function copies a portion of an internal LOB into another internal LOB. The source and destination LOBs must already exist. If data already exists at the destination's start position, it is overwritten with the source data.</p> <p>If the destination's start position is beyond the end of the current value, zero-byte fillers (BLOBs) or spaces (CLOBs/NCLOBs) are placed in the LOB from the end of the destination value to the beginning of the newly written data from the source. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is an error to extend the destination LOB beyond the maximum length allowed (4 gigabytes).</p> <p>LOB copy operations must be performed on LOBs of the same type; i.e., one CLOB can be copied to another CLOB, and one BLOB can be copied to another BLOB, but a CLOB cannot be copied to a BLOB, and vice versa.</p>
<i>OCILobDisableBuffering()</i>	Internal LOBs only	Disables LOB buffering for a given internal locator.
<i>OCILobEnableBuffering()</i>	Internal LOBs only	Enables LOB buffering for a given internal locator.

Table 7–1 OCI LOB and FILE Functions (Cont.)

Function	Restrictions	Purpose
<i>OCILobErase()</i>	Internal LOBs only	<p>Erases a specified portion of the internal LOB value starting at a specified offset. The actual number of characters/bytes erased is returned. The actual number of characters/bytes and the requested number of characters/bytes will differ if the end of the LOB data is reached before erasing the requested number of characters/bytes.</p> <p>If the LOB is NULL, this routine shows that 0 characters/bytes were erased.</p>
<i>OCILobFileClose()</i> , <i>OCILobFileCloseAll()</i>		Closes a previously opened FILE, or all open FILEs. It is an error if this function is called for an internal LOB. No error is returned if the FILE exists but is not opened.
<i>OCILobFileExists()</i>		Tests to see if a FILE exists on the server.
<i>OCILobFileNameGet()</i>		Gets the name and the directory alias of a FILE.
<i>OCILobFileIsOpen()</i>		Tests to see if a FILE has been opened with the input locator.
<i>OCILobFileOpen()</i>		Opens a FILE. The FILE can be opened for read-only access. It is an error if this call is made on an internal LOB.
<i>OCILobFileNameSet()</i>		Sets the name and the directory alias of a FILE.
<i>OCILobFlushBuffer()</i>	Internal LOBs only	Flushes the LOB buffer.
<i>OCILobGetLength()</i>		This function gets the length of a LOB/FILE. If the LOB/FILE is NULL, the length is undefined. Empty internal LOBs have a length of zero.
<i>OCILobIsEqual()</i>		Tests to see if two LOB/FILE locators are equal. Two locators are equal if and only if they both refer to the same LOB/FILE value.
<i>OCILobLoadFromFile()</i>		Populates all or part of a LOB with data from a FILE.
<i>OCILobLocatorIsInit()</i>		Tests to see if a LOB/FILE locator is initialized.
<i>OCILobRead()</i>		This function reads a portion of the LOB/FILE value into a buffer. It is an error to try to read from a NULL LOB/FILE.
<i>OCILobTrim()</i>	Internal LOBs only	This function truncates a LOB, trimming the LOB value to a specified smaller length.
<i>OCILobWrite()</i>	Internal LOBs only	This function writes data from a buffer into an internal LOB. If data already exists in the LOB, it is overwritten with the data stored in the buffer.

Server Roundtrips for LOB Functions

For a table showing the number of server roundtrips required for individual OCI LOB functions, refer to Appendix E, “OCI Function Server Roundtrips”.

LOB Read/Write Callbacks

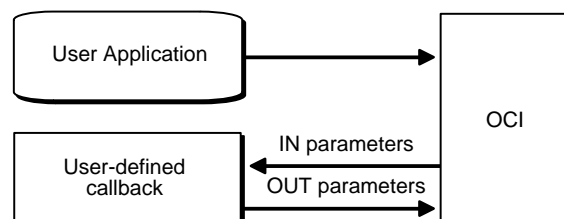
The OCI LOB read and write functions provide the ability to define callback functions which can be used to provide data to be written or handle data that was read. This allows the client application to perform optional processing on the data. One example usage of this would be to use the callbacks to implement a compression algorithm for writing the data and a decompression algorithm for reading it.

Note: The LOB read/write streaming callbacks provides a fast method for using reading/writing large amounts of LOB data.

The following sections describe the use of callbacks in more detail.

The Callback Interface for Streaming

Your application can use user-defined read and write callback functions to insert data into or retrieve data from a LOB. This provides an alternative to the polling method for streaming data into a LOB and retrieving data from a LOB. The user-defined callbacks have a specific prototype which is described below. These functions are implemented by the user and registered with OCI through the *OCILobRead()* and *OCILobWrite()* calls. The callback functions are called by OCI whenever required.



Reading LOBs using Callbacks

The user-defined read callback function is registered through the *OCILobRead()* function. The callback function should have the following prototype:

```
<CallbackFunctionName> ( dvoid *ctxp, CONST dvoid *bufp, ub4 len, ub1 piece)
```

The first parameter, *ctxp*, is the context of the callback that is passed to OCI in the *OCILobRead()* function call. When the callback function is called, the information provided by the user in *ctxp* is passed back to the user (the OCI does not use this information on the way IN). The *bufp* parameter is the pointer to the storage where the LOB data is returned and *bufl* is the length of this buffer. It tells the user how much data has been read into the buffer provided by the user.

If the buffer length provided by the user in the original *OCILobRead()* call is insufficient to store all the data returned by the server, then the user-defined callback is called. In this case the *piece* parameter indicates to the user whether the information returned in the buffer in the first, next or last piece.

The following is a code fragment of a typical way to implement read callback functions.

Assume here that *lobl* is a valid locator that has been previously selected, *svchp* is a valid service handle and *errhp* is a valid error handle.

```
...
ub4   offset = 1;
ub4   loblen = 0;
ub1   bufp[MAXBUFLen];
ub4   amtp = 0;

sword retval;

amtp = 4294967295;          /* 4 gigabytes */

if (retval = OCILobRead(svchp, errhp, lobl, &amtp, offset, (dvoid *) bufp,
    (ub4) MAXBUFLen, (dvoid *) bufp, cbk_read_lob,
    (ub2) 0, (ub1) SQLCS_IMPLICIT))
{
    (void) printf("ERROR: OCILobRead() LOB.\n");
    report_error();
}

...
sb4 cbk_read_lob(ctxp, bufxp, lenp, piece)
dvoid *ctxp;
CONST dvoid *bufxp;
```

```
ub4 lenp;
ub1 piece;

{
static ub4 piece_count = 0;

piece_count++;

switch (piece)
{
    case OCI_LAST_PIECE:

        /* process buffer bufxp */
        --- buffer processing code goes here ---

        (void) printf("callback read the %d th piece\n\n", piece_count);

        piece_count = 0;

        break;

    case OCI_FIRST_PIECE:
    case OCI_NEXT_PIECE:

        /* process buffer bufxp */
        --- buffer processing code goes here ---

        (void) printf("callback read the %d th piece\n", piece_count);

        break;

    default:

        (void) printf("callback read error: unkown piece = %d.\n", piece);

        return OCI_ERROR;
    }
    return OCI_CONTINUE;
}
```

In the above example the user defined function *cbk_read_lob* is repeatedly called until all the LOB data has been read by the user.

Writing LOBs using Callbacks

Similar to read callbacks, the user-defined write callback function is registered through the *OCILobWrite()* function. The callback function should have the following prototype:

```
<CallbackFunctionName> ( dvoid *ctxp, dvoid *bufp, ub4 *len, ub1 *piece)
```

The first parameter, *ctxp*, is the context of the callback that is passed to OCI in the *OCILobWrite()* function call. The information provided by the user in *ctxp*, is passed back to the user when the callback function is called by the OCI (the OCI does not use this information on the way IN). The *bufp* parameter is the pointer to a storage area that contains the LOB data to be inserted, and *bufl* is the length of this storage area. The user provides this pointer in the call to *OCILobWrite()*. After inserting the data provided in the call to *OCILobWrite()* if there is more to write, then the user defined callback is called. In the callback the user should provide the data to insert in the storage indicated by *bufp* and also specify the length in *bufl*. The user should also indicate whether it is the next (OCI_NEXT_PIECE) or the last (OCI_LAST_PIECE) piece using the *piece* parameter. Note that the user is completely responsible for the storage pointer the application provides and should make sure that it does not write more than the allocated size of the storage.

The following is a code fragment of a typical way to implement write callback functions.

Assume here that *lobl* is a valid locator that has been locked for updating, *svchp* is a valid service handle and *errhp* is a valid error handle

```
...

ub4   offset = 1;
ub1   bufp[MAXBUFLen];
ub4   amtp = MAXBUFLen * 20;
ub4   nbytes = MAXBUFLen;

/* Fill bufp with some data */

-- code to fill bufp with data goes here. nbytes should reflect the size and
should be less than or equal to MAXBUFLen --

if (retval = OCILobWrite(svchp, errhp, lobl, &amtp, offset, (dvoid*)
    bufp, (ub4)nbytes, OCI_FIRST_PIECE, (dvoid *)0, cbk_write_lob,
    (ub2) 0, (ub1) SQLCS_IMPLICIT))
{
```

```

        (void) printf("ERROR: OCILobWrite().\n");
        report_error();
        return;
    }
    ...

sb4 cbk_write_lob(ctxp, bufxp, lenp, piece)
dvoid *ctxp;
dvoid *bufxp;
ub4 *lenp;
ubl *piece;

{
    /* Fill bufxp with data */

    -- code to fill bufxp with data goes here. *lenp should reflect the size
    and should be less than or equal to MAXBUFLen --

    if (this is the last data buffer)

        *piecep = OCI_LAST_PIECE;

    else

        *piecep = OCI_NEXT_PIECE;;

    return OCI_CONTINUE;
}

```

In the above example, the user defined function *cbk_write_lob* is repeatedly called until the user indicates that the application is providing the last piece using the *piecep* parameter.

OCI Callbacks From External Procedures

There are four OCI functions that can be used as callbacks from external procedures. These functions are listed in Chapter 16, “OCI External Procedure Functions”.

For information about writing C subroutines that can be called from PL/SQL code, including a list of which OCI calls can be used, and some example code, refer to the *PL/SQL User's Guide and Reference*.

Application Failover Callbacks

Application failover callbacks can be used in the event of the failure of one database instance, and failover to another instance. Because of the delay which can occur during failover, the application developer may want to inform the user that failover is in progress, and request that the user stand by. Additionally, the session on the initial instance may have received some ALTER SESSION commands. These will not be automatically replayed on the second instance. Consequently, the developer may wish to replay these ALTER SESSION commands on the second instance.

Note: To use application failover you must be using the Oracle8 Enterprise Edition with the Parallel Server Option.

See Also: For more detailed information about application failover, refer to the *Oracle8 Parallel Server Concepts and Administration* manual.

Failover Callback Overview

To address the problems described above, the application developer can register a failover callback function. In the event of failover, the callback function is invoked several times during the course of reestablishing the user's session.

The first call to the callback function occurs when Oracle first detects an instance connection loss. This callback is intended to allow the application to inform the user of an upcoming delay. If failover is successful, a second call to the callback function occurs when the connection is reestablished and usable. At this time the client may wish to replay ALTER SESSION commands and inform the user that failover has happened. If failover is unsuccessful, then the callback is called to inform the application that failover will not take place. Additionally, the callback is called each time a user handle besides the primary handle is reauthenticated on the new connection. Since each user handle represents a server-side session, the client may wish to replay ALTER SESSION commands for that session.

Failover Callback Structure and Parameters

The basic structure of a user-defined application failover callback function is as follows:

```
sb4 callback_fn ( dvoid      * svchp,  
                  dvoid      * envhp,  
                  dvoid      * fo_ctx,  
                  ub4         fo_type,  
                  ub4         fo_event );
```

Each of the parameters is described below, and an example is provided in the section “Failover Callback Example” on page 7-38.

svchp The first parameter, *svchp*, is the service context handle. It is of type **dvoid ***.

envhp The second parameter, *envhp*, is the OCI environment handle. It is of type **dvoid ***.

fo_ctx The third parameter, *fo_ctx*, is a client context. It is a pointer to memory specified by the client. In this area the client can keep any necessary state or context. It is passed as a **dvoid ***.

fo_type The fourth parameter, *fo_type*, is the failover type. This lets the callback know what type of failover the client has requested. The usual values are:

- OCI_FO_SESSION, which indicates that the user has requested only session failover, and
- OCI_FO_SELECT, which indicates that the user has requested select failover as well.

fo_event The last parameter is the failover event. This indicates to the callback why it is being called. It has several possible values:

- OCI_FO_BEGIN indicates that failover has detected a lost connection and failover is starting.
- OCI_FO_END indicates successful completion of failover.
- OCI_FO_ABORT indicates that failover was unsuccessful.
- OCI_FO_REAUTH indicates that a user handle has been reauthenticated. To find out which, the application should check the OCI_ATTR_SESSION attribute of the service context handle (which is the first parameter).

Failover Callback Registration

For the failover callback to be used, it must be registered on the server context handle. This registration is done by creating a callback definition structure and setting the OCI_ATTR_FOCBK attribute of the server handle to this structure. The callback definition structure must be of type **OCIFocbkStruct**. It has two fields: *callback_function*, which contains the address of the function to call, and *fo_ctx* which contains the address of the client context.

An example of callback registration is included as part of the example in the next section.

Failover Callback Example

The following code shows an example of a simple user-defined callback function definition and registration.

Part 1, Failover Callback Definition

```
sb4 callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event )
dvoid * svchp;
dvoid * envhp;
dvoid *fo_ctx;
ub4 fo_type;
ub4 fo_event;
{
    switch (fo_event)
    {
        case OCI_FO_BEGIN:
        {
            printf(" Failing Over ... Please stand by \n");
            printf(" Failover type was found to be %s \n",
                ((fo_type==OCI_FO_SESSION) ? "SESSION"
                :(fo_type==OCI_FO_SELECT) ? "SELECT"
                : "UNKNOWN!"));
            printf(" Failover Context is :%s\n",
                (fo_ctx?(char *)fo_ctx:"NULL POINTER!"));
            break;
        }
        case OCI_FO_ABORT:
        {
            printf(" Failover aborted. Failover will not take place.\n");
            break;
        }
        case OCI_FO_END:
        {
            printf(" Failover ended ...resuming services\n");
            break;
        }
        case OCI_FO_REAUTH:
        {
            printf(" Failed over user. Resuming services\n");
            break;
        }
    }
}
```



```

default:
{
    printf("Bad Failover Event: %d.\n", fo_event);
    return -20000;    /* error -should not have happened */
}
}
return 0;
}

```

Part 2, Failover Callback Registration

```

int register_callback(svrh, errh)
dvoid *svrh; /* the server handle */
OCIError *errh; /* the error handle */
{
    OCIFocbkStruct failover;          /* failover callback structure */

    /* allocate memory for context */
    if (!(failover.fo_ctx = (dvoid *)malloc(strlen("my context."))))
        return(1);

    /* initialize the context. */
    strcpy((char *)failover.context_function, "my context.");

    failover.callback_function = &callback_fn;

    /* do the registration */
    if (OCIAttrSet(svrh, (ub4) OCI_HTYPE_SRV,
                  (dvoid *) &failover, (ub4) 0,
                  (ub4) OCI_ATTR_FOCBK, errh) != OCI_SUCCESS)
        return(2);

    /* successful conclusion */
    return (0);
}

```

OCI and Advanced Queueing

The OCI provides an interface to Oracle8's Advanced Queueing feature. Oracle AQ provides message queuing as an integrated part of the Oracle server. Oracle AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database*. By providing an integrated solution Oracle AQ frees application developers to devote their efforts to their specific business logic rather than having to construct a messaging infrastructure.

Note: In order to use advanced queueing, you must be using the Oracle8 Enterprise Edition. To use AQ with queues of datatypes other than **RAW**, you must also have purchased the Objects Option.

See Also: For detailed information about AQ, including concepts, features, and examples, refer to the chapter on Advanced Queueing in the *Oracle8 Application Developer's Guide*.

For example code demonstrating the use of the OCI with AQ, refer to the description of *OCIAQEnq()* on page 13-11.

OCI Advanced Queueing Functions

The OCI library includes two functions related to advanced queueing:

- *OCIAQEnq()*
- *OCIAQDeq()*

Chapter 13, "OCI Relational Functions", contains complete descriptions of these functions and their parameters.

OCI Advanced Queueing Descriptors

The following descriptors are used by OCI AQ operations:

- **OCIAQEnqOptions** - equivalent to `dbms_aq.enqueue_options_t`
- **OCIAQDeqOptions** - equivalent to `dbms_aq.dequeue_options_t`
- **OCIAQMsgProperties** - equivalent to `dbms_aq.message_properties_t`
- **OCIAQAgent** - equivalent to `sys.aq$_agent`

You can allocate these descriptors with respect to the service handle using the standard *OCIDescriptorAlloc()* call. The following code shows examples of this:

```
OCIDescriptorAlloc(svch, &enqueue_options, OCI_DTYPE_AQENQ_OPTIONS, 0, 0 );
OCIDescriptorAlloc(svch, &dequeue_options, OCI_DTYPE_AQDEQ_OPTIONS, 0, 0 );
```

```
OCIDescriptorAlloc(svch, &message_properties, OCI_DTYPE_AQMSG_PROPERTIES, 0, 0);
OCIDescriptorAlloc(svch, &agent, OCI_DTYPE_AQAGENT, 0, 0 );
```

As with other OCI descriptors, the structure of these descriptors is opaque to the user. Each descriptor has a variety of attributes which can be set and/or read. These attributes are described in more detail in “Advanced Queueing Descriptor Attributes” on page B-28.

Advanced Queueing in OCI vs. PL/SQL

The following tables compare functions, parameters, and options for OCI AQ functions and descriptors, and PL/SQL AQ functions in the `dbms_aq` package.

PL/SQL Function	OCI Function
DBMS_AQ.ENQUEUE	OCIAQEnq()
DBMS_AQ.DEQUEUE	OCIAQDeq()

DBMS_AQ.ENQUEUE Parameter	OCIAQEnq() Parameter
queue_name	queue_name
enqueue_options	enqueue_options
message_properties	message_properties
payload	payload
msgid	msgid
Note: OCIAQEnq() also requires the following additional parameters: <i>svch</i> , <i>errh</i> , <i>payload_tdo</i> , <i>payload_ind</i> , and <i>flags</i>	

DBMS_AQ.DEQUEUE Parameter	OCIAQDeq() Parameter
queue_name	queue_name
dequeue_options	dequeue_options
message_properties	message_properties
payload	payload
msgid	msgid
Note: OCIAQDeq() also requires the following additional parameters: <i>svch</i> , <i>errh</i> , <i>payload_tdo</i> , <i>payload_ind</i> , and <i>flags</i>	

PL/SQL Agent Parameter	OCIAQAgent Attribute
name	OCI_ATTR_AGENT_NAME
address	OCI_ATTR_AGENT_ADDRESS
protocol	OCI_ATTR_AGENT_PROTOCOL

PL/SQL Message Property	OCIAQMsgProperties Attribute
priority	OCI_ATTR_PRIORITY
delay	OCI_ATTR_DELAY
expiration	OCI_ATTR_EXPIRATION
correlation	OCI_ATTR_CORRELATION
attempts	OCI_ATTR_ATTEMPTS
recipient_list	OCI_ATTR_RECIPIENT_LIST
exception_queue	OCI_ATTR_EXCEPTION_QUEUE
enqueue_time	OCI_ATTR_ENQ_TIME
state	OCI_ATTR_MSG_STATE

PL/SQL Enqueue Option	OCIAQEnqOptions Attribute
visibility	OCI_ATTR_VISIBILITY
relative_msgid	OCI_ATTR_RELATIVE_MSGID
sequence_deviation	OCI_ATTR_SEQUENCE_DEVIATION

PL/SQL Dequeue Option	OCIAQDeqOptions Attribute
consumer_name	OCI_ATTR_CONSUMER_NAME
dequeue_mode	OCI_ATTR_DEQ_MODE
navigation	OCI_ATTR_NAVIGATION
visibility	OCI_ATTR_VISIBILITY
wait	OCI_ATTR_WAIT
msgid	OCI_ATTR_DEQ_MSGID
correlation	OCI_ATTR_CORRELATION

Writing Oracle Security Services Applications

For information about writing C applications using the Oracle Security Services Toolkit, refer to the *Oracle Security Server Guide*.

Part II

OCI Object Concepts

This part of the book contains chapters that describe the use of Oracle8 objects with the OCI:

- Chapter 8, “OCI Object-Relational Programming”, provides an introduction to object concepts and object-relational programming with the OCI.
- Chapter 9, “Object-Relational Datatypes”, discusses object datatypes and how you can represent database objects as C structures. This chapter also describes OCI functions that map and manipulate datatypes.
- Chapter 10, “Binding and Defining in Object Applications”, covers binding and defining object-relational datatypes.
- Chapter 11, “Object Cache and Object Navigation”, describes the object cache and how to navigate between objects.
- Chapter 12, “Using the Object Type Translator”, discusses how the OTT is used to convert database type definitions into host language representations.

Note: The functionality described in this part of the book is only available if you have purchased the Oracle8 Enterprise Edition with the Objects Option.

OCI Object-Relational Programming

This chapter introduces the OCI's facility for working with objects in an Oracle8 server. It also discusses the OCI's object navigational function calls.

This chapter includes the following sections:

- Chapter Overview
- OCI Object Overview
- Working with Objects in the OCI
- Developing an OCI Object Application

Note: The functionality described in this chapter is only available if you have purchased the Oracle8 Enterprise Edition with the Objects Option.

Chapter Overview

This chapter is divided into three sections covering the basic concepts involved in writing OCI applications to manipulate Oracle8 objects. The chapter also covers the OCI navigational function calls.

The following specific sections are included:

- **OCI Object Overview** presents a brief introduction to the OCI facilities for working with objects.
- **Working with Objects in the OCI** describes the basic structure of an OCI object application and the different types of objects with which the OCI works. This section provides a foundation upon which the rest of the chapter builds.
- **Developing an OCI Object Application** discusses each of the main elements of an OCI object application in more detail. Simple examples illustrate the most important points.

The next four chapters contain additional information about using the OCI to work with objects:

- Chapter 9, “Object-Relational Datatypes”, discusses the datatypes used by OCI object-relational applications. This information supplements that found in Chapter 3, “Datatypes”. This chapter also includes a discussion of the OCI datatype mapping and manipulation functions.
- Chapter 10, “Binding and Defining in Object Applications”, discusses information about bind and define operations specific to object-relational datatypes. This information supplements that in Chapter 2, “OCI Programming Basics”, and Chapter 5, “Binding and Defining”.
- Chapter 11, “Object Cache and Object Navigation”, discusses the object cache and object navigation. This chapter includes a discussion of the OCI navigational functions.
- Chapter 12, “Using the Object Type Translator” discusses the Object Type Translator.

Complete descriptions of all of the OCI object-relational functions are contained in Chapter 14, “OCI Navigation and Type Functions”, and Chapter 15, “OCI Datatype Mapping and Manipulation Functions”. Additionally, some object functionality is included in those functions contained in Chapter 13, “OCI Relational Functions”.

OCI Object Overview

The Oracle Call Interface (OCI) provides functions for managing database access and processing SQL statements. These functions are described in detail in Part 1 of this book. The SQL capabilities of the OCI relational interface allow an application to access objects from an Oracle8 server through SQL statements.

Note: The Oracle8 OCI libraries are supported only for C.

The OCI allows applications to access any of the datatypes found in the Oracle8 server, including scalar values, collections, and instances of any object type. This includes all of the following:

- objects
- variable-length arrays (VARARRAYs)
- nested tables (multisets)
- references (REFs)
- LOBs

In order to take full advantage of Oracle8 server object capabilities, most applications need to do more than just access objects. Once the object has been retrieved, the application must navigate through references from that object to other objects. The OCI provides the capability to do this.

Through the OCI's object *navigational calls*, an application can perform any of the following functions on Oracle8 objects:

- creating, accessing, locking, deleting, copying, and flushing objects
- getting references to the objects and their meta-objects
- dynamically getting and setting values of objects' attributes

The OCI navigational calls are discussed in more detail later in this chapter.

The OCI also provides the ability to access type information stored in an Oracle8 database. The *OCIDescribeAny()* function enables an application to access most information relating to types stored in the database, including information about methods, attributes, and type meta-data.

OCIDescribeAny() is discussed in Chapter 6, "Describing Schema Metadata".

Applications interacting with Oracle8 objects need a way to represent those objects in a host language format. Oracle8 provides a utility called the Object Type Translator (OTT), which can convert type definitions in the database to C struct

declarations. The declarations are stored in a header file that can be included in an OCI application.

When type definitions are represented in C, the types of attributes are mapped to special C variable types that are new to Oracle8. The OCI includes a set of *datatype mapping and manipulation functions* that enable an application to manipulate these datatypes, and thus manipulate the attributes of objects. These functions are discussed in more detail in Chapter 9, “Object-Relational Datatypes”.

The terminology for objects can occasionally become confusing. In the remainder of this chapter, the terms *object* and *instance* both refer to an object that is either stored in the database or is present in the object cache.

Working with Objects in the OCI

Many of the programming principles that govern a relational OCI application (as discussed in Chapters 2 through 6) are the same for an object-relational application. An object-relational application uses the standard OCI calls to establish database connections and process SQL statements. The difference is that the SQL statements issued retrieve object references (or objects by value), which can then be manipulated with the OCI's object functions.

Basic Object Program Structure

The basic structure of an OCI application that uses objects is essentially the same as that for a relational OCI application, as described in the section “OCI Program Structure” on page 2-3. That paradigm is reproduced here, with extra information covering basic object functionality.

1. Initialize the OCI programming environment.

Note: You *must* initialize the environment in object mode.

Your application will most likely also need to include C struct representations of database objects in a header file. These structs can be created by the programmer, or, more easily, they can be generated by the Object Type Translator (OTT), as described in Chapter 12, “Using the Object Type Translator”.

2. Allocate necessary handles, and establish a connection to a server.
3. Prepare a SQL statement for execution. This is a local (client-side) step, which may include binding placeholders and defining output variables. In an object-relational application, this SQL statement should return a reference (REF) to an object.

Note: It is also possible to fetch an entire object, rather than just a reference (REF). If you SELECT a referenceable object, rather than pinning it, you get that object “by value”. Alternately, you can select a non-referenceable object, as described in “Fetching Embedded Objects” on page 8-15

4. Associate the prepared statement with a database server, and execute the statement.
5. Fetch returned results.

In an object-relational application, this step entails retrieving the REF, and then pinning the object to which it refers. Once the object is pinned, your application will do some or all of the following:

- Manipulate the attributes of the object and mark it as “dirty”
 - Follow a REF to another object or series of objects
 - Access type and attribute information
 - Navigate a complex object retrieval graph
 - Flush modified objects to the server
6. Commit the transaction. This step implicitly flushes all modified objects to the server and commits the changes.
 7. Free statements and handles not to be reused or reexecute prepared statements again.

All of these steps are discussed in more detail in the remainder of this chapter.

See Also: For information about using the OCI to connect to a server, process SQL statements, and allocate handles, see Chapter 2 and the description of the OCI relational functions in Chapter 13.

For information about the OTT, refer to the section “Representing Objects in C Applications” on page 8-8, and Chapter 12, “Using the Object Type Translator”.

Persistent Objects, Transient Objects, and Values

Instances of an Oracle8 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 referenceable by way of an object identifier.

Note: The terms *object* and *instance* are used interchangeably in this manual.

See Also: For more information about objects, refer to the *Oracle8 Concepts* manual.

Persistent Objects

A persistent object is an object which is stored in an Oracle8 database. It may be fetched into the object cache and modified by an OCI 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:

- Standalone instances are stored in rows of a object table, and each one has a unique object identifier. An OCI application can retrieve a REF to a standalone instance, pin the object and navigate from the pinned object to other related objects.

Standalone object 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 its REF.

- Embedded instances are not stored as rows in a object table. They are embedded within other structures. Examples of embedded objects are objects which are attributes of another object, or instances which exist in an object column of a database table. Embedded instances do not have object identifiers, and OCI applications cannot get REFs to embedded instances.

Embedded objects may also be referred to as *non-referenceable 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.

Example 1, Standalone Objects

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

Objects which are stored in the object table `person_tab` are standalone instances. They have object identifiers and are referenceable. They can be pinned in an OCI application.

Example 2, Embedded Objects

```
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 are not referenceable. This means they cannot be pinned in an OCI 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. It may have an object identifier, and it has a lifetime which is determined by the application when the instance is created. The application can also delete a transient object at any time.

Transient objects are often created by the application using the `OCIObjectNew()` function to store temporary values for computation.

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

See Also: See the section “Creating, Freeing, and Copying Objects” on page 8-31 for more information about using `OCIObjectNew()`.

Values

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

- a scalar value which is stored in a non-object column of a database table. An OCI application can fetch values from a database by issuing SQL statements.
- an embedded or non-referenceable object.

The context should make it clear which meaning is intended.

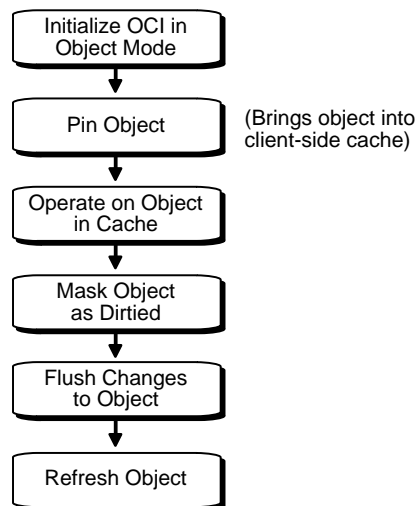
Note: It is possible to `SELECT` a referenceable object into the object cache, rather than pinning it, in which case you fetch the object “by value” instead of fetching its REF.

Developing an OCI Object Application

This section discusses the steps involved in developing a basic OCI object application. Each step mentioned in the section “Basic Object Program Structure” on page 8-4 is described here in more detail.

The following figure shows a simple program logic flow for how an application might work with objects. For simplicity, some required steps are omitted. Each step in this diagram is discussed in the following sections.

Figure 8–1 Basic Object Operational Flow



Representing Objects in C Applications

Before an OCI application can work with object types, those types must exist in the database. Typically, you create types with SQL DDL statements (e.g., CREATE TYPE).

When the Oracle8 server processes the type definition DDL commands, it stores the type definitions in the data dictionary as type descriptor objects (TDOs).

When your application retrieves instances of object types from the database, it needs to have a client-side representation of the objects. In a C program, the representation of an object type is a `struct`. In an OCI object application, you may also include a null indicator structure corresponding to each object type structure.

Oracle8 provides a utility called the Object Type Translator (OTT), which generates C struct representations of database object types for you. For example, if you have a type in your database declared as

```
CREATE TYPE emp_t AS OBJECT
( name      VARCHAR2(30),
  empno     NUMBER,
  deptno    NUMBER,
  hiredate  DATE,
  salary    NUMBER);
```

the OTT produces the following C struct and corresponding null indicator struct:

```
struct emp_t
{
    OCIStrng      * name;
    OCINumber     empno;
    OCINumber     deptno;
    OCIDate       hiredate;
    OCINumber     salary;
};
typedef struct emp_t emp_t

struct emp_t_ind
{
    OCIIInd       _atomic;
    OCIIInd       name;
    OCIIInd       empno;
    OCIIInd       deptno;
    OCIIInd       hiredate;
    OCIIInd       salary;
};
typedef struct emp_t_ind emp_t_ind;
```

The variable types used in the struct declarations are special types employed by the OCI object calls. A subset of OCI functions manipulate data of these types. These functions are mentioned later in this chapter, and are discussed in more detail in Chapter 9, “Object-Relational Datatypes”.

These struct declarations are automatically written to a .h file whose name is determined by the OTT input parameters. You can include this header file in the code files for an application to provide access to objects.

See Also: For more information about the OTT, see Chapter 12, “Using the Object Type Translator”.

For more information on the use of the NULL indicator struct, see the section “Nullness” on page 8-28.

Initializing Environment and Object Cache

If your OCI application will be accessing and manipulating objects, it is essential that you specify a value of `OCI_OBJECT` for the *mode* parameter of the `OCIInitialize()` call, which is the first OCI call in any OCI application. Specifying this value for *mode* indicates to the OCI libraries that your application will be working with objects. This notification has the following important effects:

- it establishes the *object run-time environment*
- it sets up the *object cache*

If the *mode* parameter of `OCIInitialize()` is not set to `OCI_OBJECT`, any attempt to use an object-related function will result in an error.

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

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

See Also: The object cache is mentioned throughout this chapter. For a detailed explanation of the object cache, see Chapter 11, “Object Cache and Object Navigation”.

Making Database Connections

Once the OCI environment has been properly initialized, the application can connect to a server. This is accomplished through the standard OCI connect calls described in “OCI Programming Steps” on page 2-16. When using these calls, no additional considerations need to be made because this application will be accessing objects.

There is only one object cache allocated per OCI environment. All objects retrieved or created via different connections within the environment use the same physical object cache.

Retrieving an Object Reference from the Server

In order to work with objects, your application must first retrieve one or more objects from the server. You accomplish this by issuing a SQL statement that returns 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. See the section “Fetching Embedded Objects” on page 8-15 for more information.

In the following example, the application declares a text block that stores a SQL statement designed to retrieve a REF to a single employee object from a object table of employees (`emp_tab`) in the database, given a particular employee number which is passed as an input variable (`:emp_num`) at run time:

```
text *selemp = (text *) "SELECT REF(e)
                        FROM emp_tab e
                        WHERE empno = :emp_num";
```

Your application should prepare and process this statement in the same way that it would handle any relational SQL statement, as described in Chapter 2:

- Prepare an application request, using *OCISstmtPrepare()*.
- Bind the host input variable using the appropriate bind call(s).
- Declare and prepare an output variable to receive the employee object reference. Here you would use an employee object reference, like the one declared in “Representing Objects in C Applications” on page 8-8:

```
OCISRef *emp1_ref = (OCISRef *) 0;
/* reference to an employee object */
```

When defining the output variable, set the *dtv* datatype parameter for the define call to `SQLT_REF`, the datatype constant for REF.

- Execute the statement with *OCISstmtExecute()*.
- Fetch the resulting REF into `emp1_ref`, using *OCISstmtFetch()*.

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

See Also: For general information about preparing and executing SQL statements, see the section “OCI Programming Steps” on page 2-16. For specific information about binding and defining REF variables, refer to the sections “Advanced Bind Operations” on page 5-9 and “Advanced Define Operations” on page 5-16.

For a code example showing REF retrieval and pinning, see “Example 7, REF Pinning and Navigation” on page D-118.

Pinning an Object

Upon completion of the fetch step, your application has a REF, or pointer, 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 instance into the object cache, and enables you to access and modify the instance’s attributes and follow references from that object to other objects, if necessary. Your application also controls when modified objects are written back to the 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 “Complex Object Retrieval” on page 8-21.

An application pins an object by calling the function *OCIObjectPin()*. The parameters for this function allow you to specify the *pin option*, *pin duration*, and *lock option* for the object.

The following sample code illustrates a pin operation for the employee reference we retrieved in the previous section:

```
if (OCIObjectPin(env, err, &empl_ref, (OCIComplexObject *) 0,
    OCI_PIN_ANY,
    OCI_DURATION_TRANS,
    OCI_LOCK_X, &empl) != OCI_SUCCESS)
    process_error(err);
```

In this example, *process_error()* represents an error-handling function. If the call to *OCIObjectPin()* returns anything but *OCI_SUCCESS*, the error-handling function is called. The parameters of the *OCIObjectPin()* function are as follows:

- *env* is the OCI environment handle.
- *err* is the OCI error handle.
- *empl_ref* is the reference that was retrieved through SQL.
- *(OCIComplexObject *) 0* indicates that this pin operation is not utilizing complex object retrieval.
- *OCI_PIN_ANY* is the pin option. See “Pinning an Object Copy” on page 11-6 for more information.
- *OCI_DURATION_TRANS* is the pin duration. See “Object Duration” on page 11-13 for more information.

- `OCI_LOCK_X` is the lock option. See “Locking Objects For Update” on page 11-12 for more information.
- `emp1` is an out parameter, which returns a pointer to the pinned object.

Now that the object has been pinned, the OCI application can modify that object. In this simple example, the object contains no references to other objects. For an example of navigation from one instance to another, see the section “Simple Object Navigation” on page 11-16.

Array Pin

Given an array of references, an OCI application can pin an array of objects by calling `OCIObjectArrayPin()`. The references may point to objects of different types.

Manipulating Object Attributes

Once an object has been pinned, an OCI application can modify its attributes. The OCI provides a set of function for working with datatypes of object type structs, known as the OCI datatype mapping and manipulation functions.

Note: Changes made to objects pinned in the object cache affect only those object copies (instances), and *not* the original object in the database. In order for changes made by the application to reach the database, those changes must be flushed/committed to the server. See “Marking Objects and Flushing Changes” on page 8-14 for more information.

For example, assume that the employee object in the previous section was pinned so that the employee’s salary could be increased. Assume also that at this company, yearly salary increases are prorated for employees who have been at the company for less than 180 days.

So for this example we will need to access the employee’s hire date and check whether it is more or less than 180 days prior to the current date. Based on that calculation, the employee’s salary is increased by either \$5000 (for more than 180 days) or \$3000 (for less than 180 days). The sample code on the following page demonstrates this process.

Note that the datatype mapping and manipulation functions work with a specific set of datatypes; you must convert other types, like `int`, to the appropriate OCI types before using them in calculations.

```
/* assume that sysdate has been fetched into sys_date, a string. */
/* emp1 and emp1_ref are the same as in previous sections. */
/* err is the OCI error handle. */
/* NOTE: error handling code is not included in this example. */
```

```

sb4 num_days;          /* the number of days between today and hiredate */
OCIDate curr_date;     /* holds the current date for calculations */
int raise;             /* holds the employee's raise amount before calculations */
OCINumber raise_num;   /* holds employee's raise for calculations */
OCINumber new_sal;     /* holds the employee's new salary */

/* convert date string to an OCIDate */
OCIDateFromText(err, (text *) sys_date, (ub4) strlen(sys_date), (text *)
                NULL, (ub1) 0, (text *) NULL, (ub4) 0, &curr_date);

/* get number of days between hire date and today */
OCIDateDaysBetween(err, &curr_date, &empl->hiredate, &num_days);

/* calculate raise based on number of days since hiredate */
if num_days > 180
    raise = 5000
else
    raise = 3000;

/* convert raise value to an OCINumber */
OCINumberFromInt(err, (dvoid *)&raise, (uword)sizeof(raise),
                OCI_NUMBER_SIGNED, &raise_num);

/* add raise amount to salary */
OCINumberAdd(err, &raise_num, &empl->salary, &new_sal);
OCINumberAssign(err, &new_sal, &empl->salary);

```

This example points out how values must be converted to OCI datatypes (e.g., **OCIDate**, **OCINumber**) before being passed as parameters to the OCI datatype mapping and manipulation functions.

See Also: For more information about the OCI datatypes and the datatype mapping and manipulation functions, refer to Chapter 9, “Object-Relational Datatypes”.

Marking Objects and Flushing Changes

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

The first step is to indicate that the object has been modified. This is done with the *OCIObjectMarkUpdate()* function. This function marks the object as *dirty* (modified).

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

- Flush a single dirty object by calling *OCIObjectFlush()*.
- Flush the entire cache using *OCICacheFlush()*. In this case the OCI traverses the dirty list maintained by the cache and flushes the dirty objects to the server.
- Call *OCITransCommit()* to commit a transaction. Doing so also traverses the dirty list and flushes objects to the server.

The flush operations work only on persistent objects in the cache. Transient objects are never flushed to the server.

Flushing an object to the server can activate triggers in the database. In fact, on some occasions an application may want to explicitly flush objects just to fire triggers on the server side.

See Also: For more information about *OCITransCommit()* see the section “Transactions” on page 7-3.

For information about transient and persistent objects, see the section “Creating, Freeing, and Copying Objects” on page 8-31.

For information about seeing and checking object meta-attributes (like “dirty”), see the section “Object Meta-Attributes” on page 8-17.

Fetching Embedded Objects

If your application needs to fetch an embedded object instance—an object stored in a column of a regular table, rather than an object table—you cannot use the REF retrieval mechanism described in the section “Retrieving an Object Reference from the Server” on page 8-11. Embedded instances do not have object identifiers, so it is not possible to get a REF 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 OCI 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.

Your application should prepare and process this statement in the same way that it would handle any relational SQL statement, as described in Chapter 2:

- Prepare an application request, using *OCIStmtPrepare()*.
- Bind the input variable using the appropriate bind call(s).
- Define an output variable to receive the `address` instance. You use a C struct representation of the object type that was generated by the OTT, as described in the section “Representing Objects in C Applications” on page 8-8:

```
addr1      *address; /* variable of the address struct type */
```

When defining the output variable, set the *dt* datatype parameter for the define call to `SQLT_NTY`, the datatype constant for named data types.

- Execute the statement with *OCIStmtExecute()*
- Fetch the resulting instance into `addr1`, using *OCIStmtFetch()*.

Following this, you can access the attributes of the instance, as described in the section “Manipulating Object Attributes” on page 8-13, or pass the instance as an input parameter for another SQL statement.

Note: Changes made to an embedded instance can be made persistent only by executing a SQL UPDATE statement.

See Also: For more information about preparing and executing SQL statements, see the section “OCI Programming Steps” on page 2-16.

Object Meta-Attributes

An object's *meta-attributes* serve as flags which can provide information to an application, or to the object cache, about the status of an object. For example, one of the meta-attributes of an object indicates whether or not it has been flushed to the server. These can help an application control the behavior of instances.

Persistent and transient object instances have different sets of meta-attributes. The meta-attributes for persistent objects are further broken down into *persistent meta-attributes* and *transient meta-attributes*. Transient meta-attributes exist only when an instance is in memory. Persistent meta-attributes also apply to objects stored in the server.

Persistent Object Meta-Attributes

The following table shows the meta-attributes for *standalone* persistent objects.

Persistent Meta-Attributes	Meaning
existent	does the object exist?
nullness	null information of the instance
locked	has the object been locked?
dirty	has the object been marked as "dirtied"?
Transient Meta-Attributes	
pinned	is the object pinned?
allocation duration	see "Object Duration" on page 11-13
pin duration	see "Object Duration" on page 11-13

Note: Embedded persistent objects only have the *nullness* and *allocation duration* attributes, which are transient.

The OCI provides the *OCIObjectGetProperty()* function, which allows an application to check the status of a variety of attributes of an object.

The syntax of the function is:

```
sword OCIObjectGetProperty ( OCIEnv          *envh,  
                             OCIError       *errh,  
                             CONST dvoid    *obj,  
                             OCIObjectPropId propertyId,  
                             dvoid          *property,  
                             ub4           *size );
```

The *propertyId* and *property* parameters are used to retrieve information about any of a variety of properties or attributes

The different property ids and the corresponding type of *property* argument are given below. For more information, see *OCIObjectGetProperty()* on page 14-30.

OCI_OBJECTPROP_LIFETIME

This identifies whether the given object is a persistent object or a transient object or a value instance. The *property* argument must be a pointer to a variable of type **OCIObjectLifetime**. Possible values include:

- OCI_OBJECT_PERSISTENT
- OCI_OBJECT_TRANSIENT
- OCI_OBJECT_VALUE

OCI_OBJECTPROP_SCHEMA

This returns the schema name of the table in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the schema name an error is returned, the error message will communicate the required size. Upon success, the size of the returned schema name in bytes is returned via *size*. The *property* argument must be an array of type **text** and *size* should be set to size of array in bytes by the caller.

OCI_OBJECTPROP_TABLE

This returns the table name in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the table name an error is returned, the error message will communicate the required size. Upon success, the size of the returned table name in bytes is returned via *size*. The *property* argument must be an array of type **text** and *size* should be set to size of array in bytes by the caller.

OCI_OBJECTPROP_PIN_DURATION

This returns the pin duration of the object. An error is returned if the given object points to a value instance. The *property* argument must be a pointer to a variable of type **OCIDuration**. Valid values include:

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

For more information about durations, see “Object Duration” on page 11-13.

OCI_OBJECTPROP_ALLOC_DURATION

This returns the allocation duration of the object. The *property* argument must be a pointer to a variable of type **OCIDuration**. Valid values include:

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

For more information about durations, see “Object Duration” on page 11-13.

OCI_OBJECTPROP_LOCK

This returns the lock status of the object. The possible lock status is enumerated by **OCILockOpt**. An error is returned if the given object points to a transient or value instance. The *property* argument must be a pointer to a variable of type **OCILockOpt**. Note, the lock status of an object can also be retrieved by calling *OCIObjectIsLocked()*.

OCI_OBJECTPROP_MARKSTATUS

This returns the dirty status and indicates whether the object is a new object, updated object or deleted object. An error is returned if the given object points to a transient or value instance. The *property* argument must be of type **OCIObjectMarkStatus**. Valid values include:

- OCI_OBJECT_NEW
- OCI_OBJECT_DELETED
- OCI_OBJECT_UPDATED

The following macros are available to test the object mark status:

- OCI_OBJECT_IS_UPDATED(flag)
- OCI_OBJECT_IS_DELETED(flag)
- OCI_OBJECT_IS_NEW(flag)
- OCI_OBJECT_IS_DIRTY(flag)

OCI_OBJECTPROP_VIEW

This identifies whether the specified object is a view object or not. If the property value returned is TRUE, it indicates the object is a view otherwise it is not. An error is returned if the given object points to a transient or value instance. The *property* argument must be of type boolean.

Additional Attribute Functions

The OCI also provides routines which allow an application to set or check some of these attributes directly or indirectly, as shown in the following table:

Meta-Attribute	Set With	Check With
nullness	<none>	OCIObjectGetInd()
existence	<none>	OCIObjectExists()
locked	OCIObjectLock()	OCIObjectIsLocked()
dirty	OCIObjectMark()	OCIObjectIsDirty()

Transient Object Meta-Attributes

Transient objects have no persistent attributes, and the following transient attributes:

Transient Meta-Attributes	Meaning
existent	does the object exist?
pinned	is the object being accessed by the application?
dirty	has the object been marked as “dirtied”?
nullness	null information of the instance
allocation duration	see “Object Duration” on page 11-13
pin duration	see “Object Duration” on page 11-13

Complex Object Retrieval

In the examples earlier in this chapter, only a single instance at a time was fetched or pinned. In these cases, each pin operation involved a separate 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. The applications process objects by starting at some initial set of objects, and then using the references in these initial objects to traverse the remaining objects. In a client-server setting, each of these traversals could result in costly network roundtrips to fetch objects.

Application performance when dealing with objects may be increased through the use of *complex object retrieval (COR)*. This is a prefetching mechanism in which an application specifies a criteria for retrieving a set of linked objects in a single operation.

Note: As described below, this does not mean that these prefetched objects are all 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.

The use of COR does not add functionality; it only improves performance. Therefore, its use is optional.

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 `VARARRAY` 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:

1. a REF to the desired root object.
2. 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 above, the application must specify the following:

1. the REF to the root purchase order object
2. one or more pairs of type and depth information for `cust`, `related_orders`, or `line_items`

An application fetching a purchase order will very likely need access to the customer information for that order. Using simple navigation, this would require two server accesses to retrieve the two objects. Through complex object retrieval, the customer can be prefetched when the application pins the purchase order. In this case, the complex object would consist of the purchase order object and the customer object it references.

In the above example, the application would specify the `purchase_order` REF, and would indicate that the `cust` REF attribute should be followed to a depth level of 1:

1. REF(PO object)
2. {(customer, 1)}

If the application wanted to prefetch the `purchase_order` object and all objects in the object graph it contains, the application would specify that both the `cust` and `related_orders` should be followed to the maximum depth level possible.

1. REF(PO object)
2. {(customer, 1), (purchase_order, UB4MAXVAL)}

where `UB4MAXVAL` specifies that all objects of the specified type reachable through references from the root object should be prefetched.

If an application wanted to fetch a PO and all the associated line items, it would specify:

1. REF(PO object)
2. {(line_item, 1)}

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 above two examples, the application could also specify (PO object REF, UB4MAXVAL) 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 server round trip.

Prefetching 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 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.

Implementing Complex Object Retrieval in the OCI

Complex Object Retrieval (COR) allows an application to prefetch a complex object while fetching the root object. The complex object specifications are passed to the same *OCIObjectPin()* function used for simple objects.

An application specifies the parameters for complex object retrieval using a *complex object retrieval handle*. This handle is of type **OCIComplexObject** and is allocated in the same way as other OCI handles.

The complex object retrieval handle contains a list of *complex object retrieval descriptors*. The descriptors are of type **OCIComplexObjectComp**, and are allocated in the same way as other OCI descriptors.

Each COR descriptor contains a type REF and a depth level. The type REF specifies a type of reference to be followed while constructing the complex object. The depth level indicates how far a particular type of reference should be followed. Specify an integer value, or the constant `UB4MAXVAL` for the maximum possible depth level.

The application can also specify the depth level in the COR handle without creating COR descriptors for type and depth parameters. In this case, all REFs are followed to the depth specified in the COR handle. The COR handle can also be used to specify whether a collection attribute should be fetched separately on demand (out-of-line) as opposed to the default case of fetching it along with the containing object (inline).

The application uses *OCIAttrSet()* to set the attributes of a COR handle. The attributes are:

`OCI_ATTR_COMPLEXOBJECT_LEVEL` - the depth level

`OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE` - fetch collection attribute in an object type out-of-line

The application allocates the COR descriptor using *OCIDescriptorAlloc()* and then can set the following attributes:

`OCI_ATTR_COMPLEXOBJECTCOMP_TYPE` - the type REF

`OCI_ATTR_COMPLEXOBJECTCOMP_LEVEL` - the depth level for references of the above type

Once these attributes are set, the application calls *OCIParmSet()* to put the descriptor into a complex object retrieval handle. The handle has an `OCI_ATTR_PARAM_COUNT` attribute which specifies the number of descriptors on the handle. This attribute can be read with *OCIAttrGet()*.

Once the handle has been populated, it can be passed to the *OCIObjectPin()* call to pin the root object and prefetch the remainder of the complex object.

The complex object retrieval handles and descriptors must be freed explicitly when they are no longer needed.

See Also: For more information about handles and descriptors, see “Handles” on page 2-6 and “Descriptors and Locators” on page 2-12.

COR Prefetching

The application specifies a complex object while fetching the root object. The prefetched objects are obtained by doing a breadth-first traversal of the graph(s) of objects rooted at a given root object(s). The traversal stops when all required objects have been prefetched, or when the total size of all the prefetched objects exceeds the *prefetch limit*.

COR interface

The interface for fetching complex objects is the OCI pin interface. The application can pass an initialized COR handle to *OCIObjectPin()* (or an array of handles to *OCIObjectArrayPin()*) to fetch the root object and the prefetched objects specified in the COR handle.

```

sword OCIObjectPin ( OCIEnv          *env,
                    OCIError        *err,
                    OCIRef          *object_ref,
                    OCIComplexObject *corhdl,
                    OCIPinOpt       pin_option,
                    OCIDuration     pin_duration,
                    OCILockOpt      lock_option,
                    dvoid           **object );

sword OCIObjectArrayPin ( OCIEnv          *env,
                        OCIError        *err,
                        OCIRef          **ref_array,
                        ub4             array_size,
                        OCIComplexObject **cor_array,
                        ub4             cor_array_size,
                        OCIPinOpt       pin_option,
                        OCIDuration     pin_duration,
                        OCILockOpt      lock,
                        dvoid           **obj_array,
                        ub4             *pos );

```

Keep the following points in mind when using COR:

1. A null COR handle argument defaults to pinning just the root object.
2. A COR handle with type of the root object and a depth level of 0 fetches only the root object and is thus equivalent to a null COR handle.
3. The lock options apply only to the root object.

Note: In order to specify lock options for prefetched objects, the application can visit all the objects in a complex object, create an array of REFs, and lock the entire complex object in another round trip using the array interface (*OCIObjectArrayPin()*).

Example of COR

The following example illustrates how an application program can be modified to use complex object retrieval.

Consider an application that displays a purchase order and the line items associated with it. The code in boldface accomplishes this. The rest of the code uses complex object retrieval for prefetching and thus enhances the application's performance.

```
OCIEnv *envhp;
OCIError *errhp;
OCIRef *liref;
OCIRef *poref;
OCIIter *itr;
boolean eoc;
purchase_order *po = (purchase_order *)0;
line_item *li = (line_item *)0;
OCISvcCtx *svchp;
OCIComplexObject *corhp;
OCIComplexObjectComp *cordp;
OCIType *litdo;
ub4 level = 0;

/* get COR Handle */
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &corhp, (ub4)
               OCI_HTYPE_COMPLEXOBJECT, 0, (dvoid **)0);

/* get COR descriptor for type line_item */
OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &cordp, (ub4)
                  OCI_DTYPE_COMPLEXOBJECTCOMP, 0, (dvoid **) 0);

/* get type of line_item to set in COR descriptor */
OCITypeByName(envhp, errhp, svchp, (const text *) 0, (ub4) 0,
              const text *) "LINE_ITEM", (ub4) strlen((const char *)
              "LINE_ITEM"), OCI_DURATION_SESSION, &litdo);

/* set line_item type in COR descriptor */
OCIAttrSet((dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP,
           dvoid *) litdo, (ub4) sizeof(dvoid *), (ub4)
```

```

        OCI_ATTR_COMPLEXOBJECTCOMP_TYPE, (OCIError *) errhp);
level = 1;

/* set depth level for line_item_varray in COR descriptor */
OCIAttrSet( (dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP,
            (dvoid *) &level, (ub4) sizeof(ub4), (ub4)
            OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL, (OCIError *) errhp);

/* put COR descriptor in COR handle */
OCIParamSet(corhp, OCI_HTYPE_COMPLEXOBJECT, &errhp, cordp,
            OCI_DTYPE_COMPLEXOBJECTCOMP, 1);

/* pin the purchase order */
OCIObjectPin(envhp, errhp, poref, corhp, OCI_PIN_LATEST,
            OCI_REFRESH_LOADED, OCI_DURATION_SESSION,
            OCI_LOCK_NONE, (ub2) 1, (dvoid **)&po)

/* free COR descriptor and COR handle */
OCIDescriptorFree((dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP);
OCIHandleFree((dvoid *) corhp, (ub4) OCI_HTYPE_COMPLEXOBJECT);

/* iterate and print line items for this purchase order */
OCIIterCreate(envhp, errhp, po.line_items, &itr);

/* get first line item */
OCIIterNext(envhp, errhp, itr, &liref, (dvoid **)0, &eoc);
while (!eoc)          /* not end of collection */
{
    /* pin line item */
    OCIObjectPin(envhp, errhp, liref, (dvoid *)0, OCI_PIN_RECENT,
                OCI_REFRESH_LOADED, OCI_DURATION_SESSION,
                OCI_LOCK_NONE, (ub2) 1, (dvoid **)&li);
    display_line_item(li);

    /* get next line item */
    OCIIterNext(envhp, errhp, itr, &liref, (dvoid **)0, &eoc);
}

```

Pin Count and Unpinning

Each object in the object cache has a *pin count* associated with it. The pin count essentially indicates the number of code modules that are concurrently accessing the object. The pin count is set to 1 when an object is pinned into the cache for the first time. Objects prefetched with complex object retrieval enter the object cache with a pin count of zero.

It is possible to pin an already-pinned object. Doing so increases the pin count by one. When a process finishes using an object, it should *unpin* it, using `OCIObjectUnpin()`. This call decrements the pin count by one.

When the pin count of an object reaches zero, that object is eligible to be aged out of the cache if necessary, freeing up the memory space occupied by the object.

The pin count of an object can be set to zero explicitly by calling `OCIObjectPinCountReset()`.

An application can unpin all objects in the cache related to a specific connection, by calling `OCICacheUnpin()`.

See Also: See the section “Freeing an Object Copy” on page 11-8 for more information about the conditions under which objects with zero pin count are removed from the cache.

For information about explicitly flushing an object or the entire cache, see the section “Marking Objects and Flushing Changes” on page 8-14.

See the section “Freeing an Object Copy” on page 11-8 for more information about objects being aged out of the cache.

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 instance 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 instance still exists, its value is just not known. It may be thought of as an existing object with no data.

When working with objects in the OCI, an application can define a *null indicator structure* for each object type used by the application. In most cases, doing so

simply requires including the null indicator structure generated by the OTT along with the struct declaration. When the OTT output header file is included, the null indicator struct becomes available to your application.

For each type, the null indicator structure includes an atomic null indicator (whose type is **OCIInd**), and a null indicator for each attribute of the instance. If the type has an object attribute, the null indicator structure includes that attribute's null indicator structure. The following example shows the C representations of types with their corresponding null indicator structures.

```
struct address
{
    OCINumber    no;
    OCIStrng     *street;
    OCIStrng     *state;
    OCIStrng     *zip;
};
typedef struct address address;

struct address_ind
{
    OCIInd       _atomic;
    OCIInd       no;
    OCIInd       street;
    OCIInd       state;
    OCIInd       zip;
};
typedef struct address_ind address_ind;

struct person
{
    OCIStrng     *fname;
    OCIStrng     *lname;
    OCINumber     age;
    OCIDate       birthday;
    OCIArray      *dependentsAge;
    OCITable      *prevAddr;
    OCIRaw        *comment1;
    OCILobLocator *comment2;
    address       addr;
    OCIStrng     *spouse;
};
typedef struct person person;
```

```
struct person_ind
{
    OCInd      _atomic;
    OCInd      fname;
    OCInd      lname;
    OCInd      age;
    OCInd      birthday;
    OCInd      dependentsAge;
    OCInd      prevAddr;
    OCInd      comment1;
    OCInd      comment2;
    address_ind addr;
    OCInd      spouse;
};
typedef struct person_ind person_ind;
```

Note: The `dependentsAge` field of `person_ind` indicates whether the entire varray (`dependentsAge` field of `person`) is atomically null or not. Null information of individual elements of `dependentsAge` can be retrieved through the *elemind* parameter of a call to `OCICollGetElem()`. Similarly, the `prevAddr` field of `person_ind` indicates whether the entire nested table (`prevAddr` field of `person`) is atomically null or not. Null information of individual elements of `prevAddr` can be retrieved through the *elemind* parameter of a call to `OCICollGetElem()`.

For an object type instance, the first field of the null-indicator structure is the atomic null indicator, and the remaining fields are the attribute null indicators whose layout resembles the layout of the object type instance's attributes.

Checking the value of the atomic null indicator allows an application to test whether an instance is atomically NULL. Checking any of the others allows an application to test the NULL status of that attribute, as in the following code sample:

```
person_ind *my_person_ind
if ( my_person_ind -> _atomic = OCI_IND_NULL)
{
    /* instance is atomically null */
}
if ( my_person_ind -> fname = OCI_IND_NULL)
{
    /* fname attribute is NULL */
}
```

In the above example, the value of the atomic null indicator, or one of the attribute null indicators, is compared to the predefined value `OCI_IND_NULL` to test its nullness. The following predefined values are available for such a comparison:

- `OCI_IND_NOTNULL`, indicating that the value is not NULL
- `OCI_IND_NULL`, indicating that the value is NULL
- `OCI_IND_BADNULL`, indicates that an enclosing object (or parent object) is NULL. This is used by PL/SQL, and may also be referred to as an `INVALID_NULL`. For example if a type instance is NULL, then its attributes are `INVALID_NULLs`.

Use the *OCIObjectGetInd()* function to allocate storage for and retrieve the null indicator structure of an object.

See Also: For more information about OTT-generated null indicator structs, refer to Chapter 12.

Creating, Freeing, and Copying Objects

An OCI application can create any object using *OCIObjectNew()*. To create a persistent object, the application must specify the object table where the new object will reside. This value can be retrieved by calling *OCIObjectPinTable()*, and it is passed in the *table* parameter. To create a transient object, the application needs to pass only the type descriptor object (retrieved by calling *OCITypeByName()*) for the type of object being created.

OCIObjectNew() can also be used to create instances of scalars (e.g., REF, LOB, string, raw, number, and date) and collections (e.g., varray and nested table) by passing the appropriate value for the *typecode* parameter.

Use *OCIObjectFree()* to free memory allocated through *OCIObjectNew()*. Freeing an object deallocates all the memory allocated for the object, including the associated null indicator structure. This procedure deletes an object before its lifetime expires. An application can also use *OCIObjectMarkDelete()* to delete a persistent object.

An application can copy one instance to another instance of the same type using *OCIObjectCopy()*.

See Also: See the descriptions of these functions in Chapter 14 for more information.

Object Reference and Type Reference

The object extensions to the OCI provide the application with the flexibility to access the contents of objects using their pointers or their references. The OCI provides the function *OCIObjectGetObjectRef()* to return a reference to an object given the object's pointer.

For applications that also want to access the type information of objects, the OCI provides the function *OCIObjectGetProperty()* to return a reference to an object's type descriptor object (TDO), given a pointer to the object.

Error Handling in Object Applications

Error handling in OCI applications is the same, whether or not the application uses objects. For more information about function return codes and error messages, see the section "Error Handling" on page 2-25.

Object-Relational Datatypes

The OCI datatype mapping and manipulation functions provide OCI programs with the ability to manipulate instances of Oracle predefined datatypes in a C application. This chapter discusses those functions, and also includes information about how object types are stored in the database. For information about bind and define operations using the Oracle8 C datatypes, refer to Chapter 10, “Binding and Defining in Object Applications”.

The following topics are covered in this chapter:

- Overview
- Mapping Oracle8 Datatypes to C
- Manipulating C Datatypes With OCI
- Date (OCIDate)
- Number (OCINumber)
- Fixed or Variable-Length String (OCIString)
- Raw (OCIRaw)
- Collections (OCITable, OCIArray, OCIColl, OCIIter)
- REF (OCIRef)
- Object Type Information Storage and Access

Note: The functionality described in this chapter is only available if you have purchased the Oracle8 Enterprise Edition with the Objects Option.

Overview

The OCI datatype mapping and manipulation functions provide the ability to manipulate instances of predefined Oracle8 C datatypes. These datatypes are used to represent the attributes of user-defined datatypes, including object types in Oracle8.

Each group of functions within the OCI is distinguished by a particular naming convention. The datatype mapping and manipulation functions, for example, can be easily recognized because the function names start with the prefix “OCI”, followed by the name of a datatype, as in *OCIDateFromText()* and *OCIRawSize()*. As will be explained later, the names can be further broken down into function groups that operate on a particular type of data.

Additionally, the predefined Oracle8 C types on which these functions operate are also distinguished by names which begin with the prefix “OCI”, as in **OCIDate** or **OCIString**.

The datatype mapping and manipulation functions are used when an application needs to manipulate, bind, or define attributes of objects that are stored in an Oracle8 database, or which have been retrieved by a SQL query. Retrieved objects are stored in the client-side object cache, as was described in Chapter 6.

This chapter describes the purpose and structure of each of the datatypes that can be manipulated by the OCI datatype mapping and manipulation functions. It also summarizes the different function groups, and gives lists of available functions and their purposes.

This chapter also provides information about how to use these datatypes in bind and define operations within an OCI application.

These functions are valid only when an OCI application is running in object mode. For information about initializing the OCI in object mode, and creating an OCI application that accesses and manipulates objects, refer to the section “Initializing Environment and Object Cache” on page 8-10.

For detailed information about object types, attributes, and collection datatypes, refer to *Oracle8 Concepts*.

Mapping Oracle8 Datatypes to C

Oracle8 provides a rich set of predefined datatypes with which you can create tables and specify user-defined datatypes (including object types). Object types extend the functionality of Oracle8 by allowing you to create datatypes that precisely model the types of data with which they work. This can provide increased efficiency and ease-of-use for programmers who are accessing the data.

Database tables and object types are based upon the datatypes supplied by Oracle. These tables and types are created with SQL statements and stored using a specific set of Oracle internal datatypes, like VARCHAR2 or NUMBER. For example, the following SQL statements create a user-defined `address` datatype and an object table to store instances of that type:

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

The new `address` type could also be used to create a regular table with an object column:

```
CREATE TABLE employees
(name      varchar2(30),
birthday  date,
home_addr address);
```

An OCI application can manipulate information in the `name` and `birthday` columns of the `employees` table using straightforward bind and define operations in association with SQL statements. Accessing information stored as attributes of objects requires some extra steps.

The OCI application first needs a way to represent the objects in a C-language format. This is accomplished by using the Object Type Translator (OTT) to generate C struct representations of user-defined types. The elements of these structs have datatypes that represent C language mappings of Oracle8 datatypes. The following table lists the available Oracle types you can use as object attribute types and their C mappings:

Table 9–1 C Language Mappings of Object Type Attributes

Attribute Type	C Mapping
VARCHAR2(N)	OCIStrng *
VARCHAR(N)	OCIStrng *
CHAR(N), CHARACTER(N)	OCIStrng *
NUMBER, NUMBER(N), NUMBER(N,N)	OCINumber
NUMERIC, NUMERIC(N), NUMERIC(N,N)	OCINumber
REAL	OCINumber
INT, INTEGER, SMALLINT	OCINumber
FLOAT, FLOAT(N), DOUBLE PRECISION	OCINumber
DEC, DEC(N), DEC(N,N)	OCINumber
DECIMAL, DECIMAL(N), DECIMAL(N,N)	OCINumber
DATE	OCIDate
BLOB	OCILobLocator * or OCIBlobLocator *
CLOB	OCILobLocator * or OCIClobLocator *
BFILE	OCIBFileLocator*
REF	OCISRef *
RAW(N)	OCIRaw *
VARRAY	OCIArray *
Nested Table	OCITable *

An additional C type, **OCIInd**, is used to represent null indicator information corresponding to attributes of object types.

See Also: For more information and examples regarding the use of the OTT, refer to Chapter 12.

OCI Type Mapping Methodology

Oracle followed a distinct design philosophy when specifying the mappings of Oracle predefined types. The current system has the following benefits and advantages:

- The actual representation of datatypes like **OCINumber** is opaque to client applications, and the datatypes are manipulated with a set of predefined functions. This allows for the internal representation to change to accommodate future enhancements without breaking user code.
- The implementation is consistent with object-oriented paradigms in which class implementation is hidden and only the required operations are exposed.
- This implementation can have advantages for programmers. Consider a C program that wants to manipulate Oracle number variables without losing the accuracy provided by Oracle numbers. To do this in Oracle7, you would have had to issue a “SELECT...FROM DUAL” statement. In Oracle8, this is accomplished by invoking the *OCINumber*()* functions.

Manipulating C Datatypes With OCI

In an OCI application, the manipulation of data may be as simple as adding together two integer variables and storing the result in a third variable:

```
integer    int_1, int_2, sum;
...
/* some initialization occurs */
...
sum = int_1 + int_2;
```

The C language provides a set of predefined operations on simple types like **integer**. However, the C datatypes listed in Table 9–1 are not simple C primitives. Types like **OCIString** and **OCINumber** are actually structs with a specific Oracle-defined internal structure. It is not possible to simply add together two **OCINumbers** and store the value in the third.

The following is **not valid**:

```
OCINumber  num_1, num_2, sum;
...
/* some initialization occurs */
...
sum = num_1 + num_2;           /* NOT A VALID OPERATION */
```

The OCI datatype mapping and manipulation functions are provided to enable you to perform operations on these new datatypes. For example, the above addition of **OCINumbers** could be accomplished as follows, using the *OCINumberAdd()* function:

```
OCINumber    num_1, num_2, sum;
...
/* some initialization occurs */
...
OCINumberAdd(errhnp, &num_1, &num_2, &sum): /* errhnp is error handle */
```

The OCI provides functions to operate on each of the new datatypes. The names of the functions provide information about the datatype on which they operate. The first three letters, “OCI”, indicate that the function is part of the OCI. The next part of the name indicates the datatype on which the function operates. The following table shows the various function prefixes, along with example function names and the datatype on which those functions operate:

Function Prefix	Example	Operates On
OCIDate	OCIDateDaysBetween()	OCIDate
OCINumber	OCINumberAdd()	OCINumber
OCISString	OCISStringSize()	OCISString *
OCISRef	OCISRefAssign()	OCISRef *
OCISRaw	OCISRawResize()	OCISRaw *
OCIColl	OCICollGetElem()	OCIColl, OCIIter, OCITable, OCIArray
OCIIter	OCIIterInit()	OCIIter
OCITable	OCITableLast()	OCITable *

The structure of each of the datatypes is described later in this chapter, along with a list of the functions that manipulate that type.

Precision of Oracle Number Operations

Oracle numbers have a precision of 38 decimal digits. All Oracle number operations are accurate to the full precision, with the following exceptions:

- Inverse trigonometric functions are accurate to 28 decimal digits.
- Other transcendental functions, including trigonometric functions, are accurate to approximately 37 decimal digits.
- Conversions to and from native floating-point types have the precision of the relevant floating-point type, not to exceed 38 decimal digits.

Date (OCIDate)

The Oracle date format is mapped in C by the **OCIDate** type, which is an opaque C struct. Elements of the struct represent the year, month, day, hour, minute, and second of the date. The specific elements can be set and retrieved using the appropriate OCI functions.

The **OCIDate** datatype can be bound or defined directly using the external typecode `SQLT_ODT` in the bind or define call.

The OCI date manipulation functions are listed in the following tables, which are organized according to functionality. Unless otherwise specified, the term “date” in these tables refers to a value of type **OCIDate**.

See Also: The prototypes and descriptions for all the functions are provided in Chapter 15, “OCI Datatype Mapping and Manipulation Functions”.

Date Conversion Functions

The following functions perform date conversion.

Function	Purpose
<code>OCIDateToText()</code>	convert date to string
<code>OCIDateFromText()</code>	convert text string to date
<code>OCIDateZoneToZone()</code>	convert date from one time zone to another

Date Assignment and Retrieval Functions

The following functions retrieve and assign date elements.

Function	Purpose
OCIDateAssign()	OCIDate assignment
OCIDateGetDate()	get the date portion of an OCIDate
OCIDateSetDate()	set the date portion of an OCIDate
OCIDateGetTime()	get the time portion of an OCIDate
OCIDateSetTime()	set the time portion of an OCIDate

Date Arithmetic and Comparison Functions

The following functions perform date arithmetic and comparison.

Function	Purpose
OCIDateAddDays()	add days
OCIDateAddMonths()	add months
OCIDateCompare()	compare dates
OCIDateDaysBetween()	calculate the number of days between two dates

Date Information Accessor Functions

The following functions access date information.

Function	Purpose
OCIDateLastDay()	the last day of the month
OCIDateNextDay()	the first named day after a given date
OCIDateSysDate()	the system date

Date Validity Checking Functions

The following function checks date validity.

Function	Purpose
OCIDateCheck()	check whether a given date is valid

Date Example

The following code provides examples of how to manipulate an attribute of type **OCIDate** using OCI calls.

```
#define FMT "DAY, MONTH DD, YYYY"
#define LANG "American"
struct person
{
    OCIDate start_date;
};
typedef struct person person;

OCIError *err;
person *tim;
sword status; /* error status */
uword invalid;
OCIDate last_day, next_day;
text buf[100], last_day_buf[100], next_day_buf[100];
ub4 buflen = sizeof(buf);

/* For this example, assume the OCIEnv and OCIError have been
 * initialized as described in Chapter 2. */
/* Pin tim person object in the object cache. See Chapter 6 for
 * information about pinning. For this example, assume that
 * tim is pointing to the pinned object. */
/* set the start date of tim */
OCIDateSetTime(&tim->start_date,8,0,0);
OCIDateSetDate(&tim->start_date,1990,10,5)

/* check if the date is valid */
if (OCIDateCheck(err, &tim->start_date, &invalid) != OCI_SUCCESS)
/* error handling code */

if (invalid)
/* error handling code */

/* get the last day of start_date's month */
if (OCIDateLastDay(err, &tim->start_date, &last_day) != OCI_SUCCESS)
/* error handling code */

/* get date of next named day */
if (OCIDateNextDay(err, &tim->start_date, "Wednesday",    strlen("Wednesday"),
&next_day) != OCI_SUCCESS)
/* error handling code */
```

```
/* convert dates to strings and print the information out */
/* first convert the date itself*/
buflen = sizeof(buf);
if (OCIDateToText(err, &tim->start_date, FMT, sizeof(FMT)-1, LANG,
    sizeof(LANG)-1, &buflen, buf) != OCI_SUCCESS)
/* error handling code */

/* now the last day of the month */
buflen = sizeof(last_day_buf);
if (OCIDateToText(err, &last_day, FMT, sizeof(FMT)-1, LANG,    sizeof(LANG)-1,
    &buflen, last_day_buf) != OCI_SUCCESS)
/* error handling code */

/* now the first Wednesday after this date */
buflen = sizeof(next_day_out);
if (OCIDateToText(err, &next_day, FMT, sizeof(FMT)-1, LANG,
    sizeof(LANG)-1, &buflen, next_day_buf) != OCI_SUCCESS)
/* error handling code */

/* print out the info */
printf("For: %s\n", buf);
printf("The last day of the month is: %s\n", last_day_buf);
printf("The next Wednesday is: %s\n", next_day_buf);
```

The output will be:

```
For: Monday, May 13, 1996
The last day of the month is: Friday, May 31
The next Wednesday is: Wednesday, May 15
```

Number (OCINumber)

The **OCINumber** datatype is an opaque structure used to represent Oracle numeric datatypes (NUMBER, FLOAT, DECIMAL, and so forth).

This type can be bound and defined using the external typecode `SQLT_VNU` in the bind or define call.

The **OCINumber** manipulation functions are listed in the following tables, which are organized according to functionality. Unless otherwise specified, the term “number” in these tables refers to a value of type **OCINumber**.

See Also: The prototypes and descriptions for all the functions are provided in Chapter 15, “OCI Datatype Mapping and Manipulation Functions”.

Number Arithmetic Functions

The following functions perform arithmetic operations.

Function	Purpose
OCINumberAbs()	get the absolute value of a number
OCINumberAdd()	add two numbers together
OCINumberCeil()	get the ceiling value of a number
OCINumberDiv()	divide one number by another
OCINumberFloor()	get the floor value of a number
OCINumberMod()	get the modulus from the division of two numbers
OCINumberMul()	multiply two numbers together
OCINumberNeg()	negate a number
OCINumberRound()	round a number to a specified decimal place
OCINumberSign()	get the sign of a number
OCINumberSqrt()	get the square root of a number
OCINumberSub()	subtract one number from another
OCINumberTrunc()	truncate a number to a specified decimal place
OCINumberSign()	returns the sign of a given number

Number Conversion Functions

The following functions perform conversions between numbers and reals, integers, and strings.

Function	Purpose
OCINumberToInt()	convert number to integer
OCINumberFromInt()	convert integer to number
OCINumberToReal()	convert number to real
OCINumberFromReal()	convert real to number
OCINumberToText()	convert number to string
OCINumberFromText()	convert string to number

Exponential and Logarithmic Functions

The following functions perform exponential and logarithmic operations.

Function	Purpose
OCINumberPower()	take a number base to a given number exponent
OCINumberExp()	take the exponent with base e
OCINumberLog()	take the logarithm of a given base
OCINumberLn()	take the natural logarithm (base e)
OCINumberIntPower()	take a number base to a given integer power

Trigonometric Functions

The following functions perform trigonometric operations on numbers.

Function	Purpose
OCINumberArcCos()	calculate arc cosine
OCINumberArcSin()	calculate arc sine
OCINumberArcTan() / OCINumberArcTan2()	calculate arc tangent / of two numbers
OCINumberCos()	calculate cosine
OCINumberHypCos()	calculate cosine hyperbolic
OCINumberSin()	calculate sine
OCINumberHypSin()	calculate sine hyperbolic
OCINumberTan()	calculate tangent
OCINumberHypTan()	calculate tangent hyperbolic

Number Assignment and Comparison Functions

The following functions perform assign and compare operations on numbers.

Function	Purpose
OCINumberAssign()	assign one number to another
OCINumberCmp()	compare two numbers
OCINumberIsZero()	test if equal to zero
OCINumberSetZero()	initialize number to zero

Number Example

The following example shows how to manipulate an attribute of type **OCINumber**.

```

struct person
{
    OCINumber sal;
};

typedef struct person person;
OCIError *err;
person* steve;
person* scott;
person* jason;
OCINumber *stevesal;
OCINumber *scottsal;
OCINumber *debsal;
sword status;
int inum;
double dnum;
OCINumber ornum;
char buffer[21];
ub4 buflen;
sword result;

/* For this example, assume OCIEnv and OCIError are initialized. */
/* For this example, assume that steve, scott and jason are pointing to
   person objects which have been pinned in the object cache. */
stevesal = &steve->sal;
scottsal = &scott->sal;
debsal = &jason->sal;

/* initialize steve's salary to be $12,000 */
OCINumberInit(err, stevesal);
inum = 12000;
status = OCINumberFromInt(err, &inum, sizeof(inum), OCI_NUMBER_SIGNED,
    stevesal);
if (status != OCI_SUCCESS) /* handle error from OCINumberFromInt */;

/* initialize scott's salary to be same as steve */
OCINumberAssign(err, stevesal, scottsal);

/* initialize jason's salary to be 20% more than steve's */
dnum = 1.2;
status = OCINumberFromReal(err, &dnum, DBL_DIG, &ornum);
if (status != OCI_SUCCESS) /* handle error from OCINumberFromReal */;
status = OCINumberMul(err, stevesal, &ornum, debsal);

```

```

if (status != OCI_SUCCESS) /* handle error from OCINumberMul */;

/* give scott a 50% raise */
dnum = 1.5;
status = OCINumberFromReal(err, &dnum, DBL_DIG, &ornum);
if (status != OCI_SUCCESS) /* handle error from OCINumberFromReal */;
status = OCINumberMul(err, scottsal, &ornum, scottsal);
if (status != OCI_SUCCESS) /* handle error from OCINumberMul */;

/* double steve's salary */
status = OCINumberAdd(err, stevesal, stevesal, stevesal);
if (status != OCI_SUCCESS) /* handle error from OCINumberAdd */;

/* get steve's salary in integer */
status = OCINumberToInt(err, stevesal, sizeof(inum), OCI_NUMBER_SIGNED,
    &inum);
if (status != OCI_SUCCESS) /* handle error from OCINumberToInt */;

/* inum is set to 24000 */
/* get jason's salary in double */
status = OCINumberToReal(err, debsal, sizeof(dnum), &dnum);
if (status != OCI_SUCCESS) /* handle error from OCINumberToReal */;

/* dnum is set to 14400 */
/* print scott's salary as DEM0001`8000.00 */
buflen = sizeof(buffer);
status = OCINumberToText(err, scottsal, "C0999G9999D99", 13,
    "NLS_NUMERIC_CHARACTERS='.'`' NLS_ISO_CURRENCY='Germany'",
    54, &buflen, buffer);
if (status != OCI_SUCCESS) /* handle error from OCINumberToText */;
printf("scott's salary = %s\n", buffer);

/* compare steve and scott's salaries */
status = OCINumberCmp(err, stevesal, scottsal, &result);
if (status != OCI_SUCCESS) /* handle error from OCINumberCmp */;

/* result is positive */
/* read jason's new salary from string */
status = OCINumberFromText(err, "48`000.00", 9, "99G999D99", 9,
    "NLS_NUMERIC_CHARACTERS='.'`'", 27, debsal);
if (status != OCI_SUCCESS) /* handle error from OCINumberFromText */;
/* jason's salary is now 48000.00 */

```

Fixed or Variable-Length String (OCIString)

Fixed or variable-length string data is represented to C programs as an **OCIString ***.

The length of the string does not include the null character.

For binding and defining variables of type **OCIString *** use the external typecode **SQLT_VST**.

See Also: The prototypes and descriptions for all the functions are provided in Chapter 15, “OCI Datatype Mapping and Manipulation Functions”.

String Functions

The following functions allow the C programmer to manipulate an instance of a string.

Function	Purpose
OCIStringAssign()	assign one string to another
OCIStringAssignText()	assign text string to string
OCIStringAllocSize()	get allocated size of string memory in bytes
OCIStringPtr()	get pointer to string part of string
OCIStringSize()	get string size
OCIStringResize()	resize string memory

String Example

This example assigns a text string to a string, then gets a pointer to the string part of the string, as well as the string size, and prints it out.

Note the double indirection used in passing the *vstring1* parameter in *OCIStringAssignText()*.

```
OCIEnv      *envhp;
OCIError    *errhp;
OCIString    *vstring1 = (OCIString *)0;
OCIString    *vstring2 = (OCIString *)0;
text        c_string[20];
text        *text_ptr;
sword       status;

strcpy(c_string, "hello world");
/* Assign a text string to an OCIString */
```

```
status = OCIStrAssignText(envhp, errhp, c_string,
                          (ub4)strlen(c_string), &vstring1);
/* Memory for vstring1 is allocated as part of string assignment */

status = OCIStrAssignText(envhp, errhp, "hello again",
                          (ub4)strlen("This is a longer string."), &vstring1);
/* vstring1 is automatically resized to store the longer string */

/* Get a pointer to the string part of vstring1 */
text_ptr = OCIStrPtr(envhp, vstring1);
/* text_ptr now points to "hello world" */
printf("%s\n", text_ptr);
```

Raw (OCIRaw)

Variable-length raw data is represented in C using the **OCIRaw *** datatype.

For binding and defining variables of type **OCIRaw ***, use the external typecode **SQLT_LVB**.

See Also: The prototypes and descriptions for all the functions are provided in Chapter 15, “OCI Datatype Mapping and Manipulation Functions”.

Raw Functions

The following functions perform OCIRaw operations.

Function	Purpose
OCIRawAssignBytes()	assign raw data (ub1 *) to OCIRaw *
OCIRawAssignRaw()	assign one OCIRaw * to another
OCIRawAllocSize()	get the allocated size of raw memory in bytes
OCIRawPtr()	get pointer to raw data
OCIRawSize()	get size of raw data
OCIRawResize()	resize memory of variable-length raw data

Raw Example

In this example, a raw data block is set up and a pointer to its data is obtained.

Note the double indirection in the call to *OCIRawAssignBytes()*.

```
OCIEnv      *envhp;
OCIError    *errhp;
sword       status;
ub1         data_block[10000];
ub4         data_block_len = 10000;
OCIRaw      *rawl;
ub1 *rawl_pointer;

/* Set up the RAW */
/* assume 'data_block' has been initialized */
status = OCIRawAssignBytes(envhp, errhp, data_block, data_block_len, &rawl);

/* Get a pointer to the data part of the RAW */
rawl_pointer = OCIRawPtr(envhp, rawl);
```

Collections (OCITable, OCIArray, OCIColl, OCIIter)

Oracle8 provides two types of collections: variable-length arrays (varrays) and nested tables. In C applications, varrays are represented as **OCIArray ***, and nested tables are represented as **OCITable ***. Both of these datatypes (along with OCIColl and OCIIter, described later) are opaque structures.

A variety of generic collection functions enable you to manipulate collection data. You can use these functions on both varrays and nested tables. In addition, there is a set of functions specific to nested tables; see “Nested Table Manipulation Functions” on page 9-20.

You can allocate an instance of a varray or nested table using *OCIObjectNew()* and free it using *OCIObjectFree()*.

See Also: The prototypes and descriptions for all the functions are provided in Chapter 15, “OCI Datatype Mapping and Manipulation Functions”.

Generic Collection Functions

Oracle8 provides two types of collections: variable-length arrays (varrays) and nested tables. Both varrays and nested tables can be viewed as sub-types of a generic collection type.

In C, a generic collection is represented as **OCIColl ***, a varray is represented as **OCIArray ***, and a nested table as **OCITable ***. Oracle provides a set of functions to operated on generic collections (such as **OCIColl ***). These functions start with the prefix *OCIColl*, as in *OCICollGetElem()*. The *OCIColl*()* functions can also be called to operate on varrays and nested tables.

The generic collection functions are grouped into two main categories:

- manipulating varray or nested table data
- scanning through a collection with a collection iterator

The generic collection functions represent a complete set of functions for manipulating varrays. Additional functions are provided to operate specifically on nested tables. They are identified by the prefix *OCITable*, as in *OCITableExists()*. These are described in the section “Nested Table Manipulation Functions” on page 9-20.

Note: Indexes passed to collection functions are zero-based.

Collection Data Manipulation Functions

The following generic functions manipulate collection data:

Function	Purpose
OCICollAppend()	append an element
OCICollAssignElem()	assign element at given index
OCICollAssign()	assign one collection to another
OCICollGetElem()	get pointer to an element given its index
OCICollMax()	get upper bound of collection
OCICollSize()	get current size of collection
OCICollTrim()	trim n elements from the end of the collection

Collection Scanning Functions

The following generic functions enable you to scan collections with a collection iterator. The iterator is of type **OCIIter**, and is created by first calling *OCIIterCreate()*.

Function	Purpose
OCIIterCreate()	create an iterator for scanning collection
OCIIterDelete()	delete iterator
OCIIterGetCurrent()	get pointer to current element pointed by iterator
OCIIterInit()	initialize iterator to scan the given collection
OCIIterNext()	get pointer to next element
OCIIterPrev()	get pointer to previous element

Varray/Collection Iterator Example

This example creates and uses a collection iterator to scan through a varray.

```
OCIEnv      *envhp;
OCIError    *errhp;
text        *text_ptr;
sword       status;
OCIArray    *clients;
OCIString   *client_elem;
OCIIter     *iterator;
boolean     eoc;
dvoid       *elem;
OCIInd      *elemind;

/* Assume envhp, errhp have been initialized */
/* Assume clients points to a varray */

/* Print the elements of clients */
/* To do this, create an iterator to scan the varray */
status = OCIIterCreate(envhp, errhp, clients, &iterator);

/* Get the first element of the clients varray */
printf("Clients' list:\n");
status = OCIIterNext(envhp, errhp, iterator, &elem,
                    (dvoid **) &elemind, &eoc);

while (!eoc && (status == OCI_SUCCESS))
```

```
{
    client_elem = *(OCIStrng)**elem;
                                /* client_elem points to the string */

    /*
       the element pointer type returned by OCIIterNext() via 'elem' is
       the same as that of OCICollGetElem(). Refer to OCICollGetElem() for
       details.  */

    /*
       client_elem points to an OCIStrng descriptor, so to print it out,
       get a pointer to where the text begins
    */
    text_ptr = OCIStrngPtr(envhp, client_elem);

    /*
       text_ptr now points to the text part of the client OCIStrng, which is a
       NULL-terminated string
    */
    printf(" %s\n", text_ptr);
    status = OCIIterNext(envhp, errhp, iterator, &elem,
                        (dvoid **)&elemind, &eoc);
}

if (status != OCI_SUCCESS)
{
    /* handle error */
}

/* destroy the iterator */
status = OCIIterDelete(envhp, errhp, &iterator);
```

Nested Table Manipulation Functions

As its name implies, one table may be *nested* or contained within another, as a variable, attribute, parameter or column. Nested tables may have elements deleted, by means of the *OCITableDelete()* function.

For example, suppose a table is created with 10 elements, and *OCITableDelete()* is used to delete elements at index 0 through 4 and 9. The first existing element is now element 5, and the last existing element is element 8.

As noted above, the generic collection functions may be used to map to and manipulate nested tables. In addition, the following functions are specific to nested tables. They should not be used on varrays.

Function	Purpose
OCITableDelete()	delete an element at a given index
OCITableExists()	test whether an element exists at a given index
OCITableFirst()	return index for first existing element of table
OCITableLast()	return index for last existing element of table
OCITableNext()	return index for next existing element of table
OCITablePrev()	return index for previous existing element of table
OCITableSize()	return table size, not including deleted elements

Nested Table Element Ordering

When a nested table is fetched into the object cache, its elements are given a transient ordering, numbered from zero to the number of elements, minus 1. For example, a table with 40 elements would be numbered from 0 to 39.

You can use these position ordinals to fetch and assign the values of elements (for example, fetch to element *i*, or assign to element *j*, where *i* and *j* are valid position ordinals for the given table).

When the table is copied back to the database, its transient ordering is lost. Delete operations may be performed against elements of the table. Delete operations create transient “holes”; that is, they do not change the position ordinals of the remaining table elements.

REF (OCIRef)

In Oracle8, a REF (reference) is an identifier to an object. It is an opaque structure that uniquely locates the object. An object may point to another object by way of a REF.

In C applications, the REF is represented by **OCIRef ***.

See Also: The prototypes and descriptions for all the functions are provided in Chapter 15, “OCI Datatype Mapping and Manipulation Functions”.

REF Manipulation Functions

The following functions perform REF operations.

Function	Purpose
OCIRefToHex()	convert REF to a hexadecimal string
OCIRefAssign()	assign one REF to another
OCIRefClear()	clear or nullify a REF
OCIRefIsEqual()	compare two REFs for equality
OCIRefFromHex()	convert hexadecimal string to a REF
OCIRefIsNull()	test whether a REF is NULL
OCIRefHexSize()	return size of hex string representation of REF

REF Example

This example tests two REFs for NULL, compares them for equality, and assigns one REF to another.

Note the double indirection in the call to *OCIRefAssign()*.

```
OCIEnv      *envhp;
OCIError    *errhp;
sword       status;
boolean      refs_equal;
OCIRef      *ref1, ref2;

/* assume refs have been initialized to point to valid objects */
/*Compare two REFs for equality */
refs_equal = OCIRefIsEqual(envhp, ref1, ref2);
printf("After first OCIRefIsEqual:\n");
if(refs_equal)
```

```
printf("REFs equal\n");
else
    printf("REFs not equal\n");

/*Assign ref1 to ref2 */
status = OCIRefAssign (envhp, errhp, ref1, &ref2);
if(status != OCI_SUCCESS)
/*error handling*/

/*Compare the two REFs again for equality */
refs_equal = OCIRefIsEqual(envhp, ref1, ref2);
printf("After second OCIRefIsEqual:\n");
if(refs_equal)
    printf("REFs equal\n");
else
    printf("REFs not equal\n");
```

Object Type Information Storage and Access

Descriptor Objects

When a given type is created with the CREATE TYPE statement, it is stored in the server and associated with a type descriptor object (TDO). In addition, the database stores descriptor objects for each data attribute of the type, each method of the type, each parameter of each method, and the results returned by methods. The following table lists the OCI datatypes associated with each type of descriptor object.

Information Type		OCI Datatype
Type		OCIType
Type Attributes	Collection Elements	OCITypeElem
Method Parameters	Method Results	
Method		OCITypeMethod

Several OCI functions (including *OCIBindObject()* and *OCIObjectNew()*) require a TDO as an input parameter. An application can obtain the TDO by calling *OCITypeByName()*, which gets the type’s TDO in an **OCIType** variable. Once you obtain the TDO, you can pass it, as necessary to other calls.

Binding and Defining in Object Applications

The concepts of binding and defining were introduced and discussed in Chapter 2, “OCI Programming Basics” and in Chapter 5, “Binding and Defining”. This chapter provides additional information necessary for users who are developing object applications. This includes information about binding and defining object datatypes, as well as additional datatypes which have been introduced to support objects.

This chapter assumes that readers are familiar with the basics of binding and defining described in the earlier chapters.

This chapter includes the following sections:

- Binding
- Defining
- Binding And Defining Oracle8 C Datatypes

Note: The functionality described in this chapter is only available if you have purchased the Oracle8 Enterprise Edition with the Objects Option.

Binding

This section provides information on binding named data types (e.g., objects, collections) and REFs.

Named Data Type Binds

For a named data type (object type or collection) bind, a second bind call is necessary (following *OCIBindByName()* or *OCIBindByPos()*). The OCI Bind Object Type call, *OCIBindObject()*, sets up additional attributes specific to the object type bind. An OCI application uses this call when fetching data from a table which has a column with an object datatype.

The *OCIBindObject()* call takes, among other parameters, a Type Descriptor Object (TDO) for the named data type. The TDO, of datatype **OCITYPE** is created and stored in the database when a named data type is created. It contains information about the type and its attributes. An application can obtain a TDO by calling *OCITYPEByName()*.

The *OCIBindObject()* call also sets up the indicator variable or structure for the named data type bind.

When binding a named data type, use the `SQLT_NTY` datatype constant to indicate the datatype of program variable being bound. `SQLT_NTY` indicates that a C struct representing the named data type is being bound. A pointer to this structure is passed to the bind call.

It is possible that working with named data types may require the use of three bind calls in some circumstances. For example, to bind a static array of named data types to a PL/SQL table, three calls must be invoked: *OCIBindByName()*, *OCIBindArrayOfStruct()*, and *OCIBindObject()*.

See Also: For information about using these data types to fetch an embedded object from the database, refer to the section “Fetching Embedded Objects” on page 8-15.

For additional important information, see the section “Additional Information for Named Data Type and REF Binds” on page 10-3

For more information about descriptor objects, see “Descriptor Objects” on page 9-23.

Binding REFs

As with named data types, binding REFs is a two-step process. First, call *OCIBindByName()* or *OCIBindByPos()*, and then call *OCIBindObject()*.

REFs are bound using the `SQLT_REF` datatype. When `SQLT_REF` is used, then the program variable being bound must be of type **OCIRef ***.

See Also: For information about binding and pinning REFs to objects, see “Retrieving an Object Reference from the Server” on page 8-11.

For additional important information, see the section “Additional Information for Named Data Type and REF Binds” on page 10-3.

Additional Information for Named Data Type and REF Binds

This section presents some additional important information to keep in mind when working with named data type and REF defines. It includes pointers about memory allocation and indicator variable usage.

- If the datatype being bound is `SQLT_NTY`, the indicator struct parameter of the *OCIBindObject()* call (**dvoid ** indpp**) is used, and the scalar indicator is completely ignored.
- If the datatype is `SQLT_REF`, the scalar indicator is used, and the indicator struct parameter of *OCIBindObject()* is completely ignored.
- The use of indicator structures is optional. The user can pass a NULL pointer in the *indpp* parameter for the *OCIBindObject()* call. During the bind, this means that the object is not atomically NULL and none of its attributes are NULL.
- The indicator struct size pointer, *indsp*, and program variable size pointer, *pgvsp*, in the *OCIBindObject()* call is optional. Users can pass NULL if these parameters are not needed.

Information Regarding Array Binds

For doing array binds of named data types or REFs, for array inserts or fetches, the user needs to pass in an array of pointers to buffers (pre-allocated or otherwise) of the appropriate type. Similarly, an array of scalar indicators (for `SQLT_REF` types) or an array of pointers to indicator structs (for `SQLT_NTY` types) needs to be passed.

See Also: For more information about `SQLT_NTY`, see the section “New OCI 8.0 External Datatypes” on page 3-18.

Defining

This section provides information on defining named data types (e.g., objects, collections) and REFs.

Defining Named Data Type Output Variables

For a named data type (object type, nested table, varray) define, two define calls are necessary. The application should first call *OCIDefineByPos()*, specifying *SQLT_NTY* in the *dtype* parameter. Following *OCIDefineByPos()*, the application must call *OCIDefineObject()*. In this case, the data buffer pointer in *OCIDefineByPos()* is ignored and additional attributes pertaining to a named data type define are set up using the OCI Define Object attributes call, *OCIDefineObject()*.

There *SQLT_NTY* datatype constant is specified for a named datatype define. In this case, the application fetches the result data into a host-language representation of the named data type. In most cases, this will be a C struct generated by the Object Type Translator.

When making an *OCIDefineObject()* call, a pointer to the address of the C struct (preallocated or otherwise) must be provided. The object may have been created with *OCIObjectNew()*, allocated in the cache, or with user-allocated memory.

Note: Please refer to the section “Additional Information for Named Data Type and REF Defines, and PL/SQL OUT Binds” on page 10-5 for more important information about defining named data types.

Defining REF Output Variables

As with named data types, defining for a REF output variable is a two-step process. The first step is a call to *OCIDefineByPos()*, and the second is a call to *OCIDefineObject()*. Also as with named data types, the *SQLT_REF* datatype constant is passed to the *dtype* parameter of *OCIDefineByPos()*.

SQLT_REF indicates that the application will be fetching the result data into a variable of type **OCIRef ***. This REF can then be used as part of object pinning and navigation, as described in Chapter 6.

Note: Please refer to the section “Additional Information for Named Data Type and REF Defines, and PL/SQL OUT Binds” on page 10-5 for more important information about defining REFs.

Additional Information for Named Data Type and REF Defines, and PL/SQL OUT Binds

This section presents some additional important information to keep in mind when working with named data type and REF defines. It includes pointers about memory allocation and indicator variable usage.

A PL/SQL OUT bind refers to binding a placeholder to an output variable in a PL/SQL block. Unlike a SQL statement, where output buffers are set up with define calls, in a PL/SQL block, output buffers are set up with bind calls. Refer to the section “Binding Placeholders in PL/SQL” on page 5-5 for more information.

- If the datatype being defined is `SQLT_NTY`, the indicator struct parameter of the `OCIDefineObject()` call (**dvoid ** indpp**) is used, and the scalar indicator is completely ignored.
- If the datatype is `SQLT_REF`, the scalar indicator is used, and the indicator struct parameter of `OCIDefineObject()` is completely ignored.
- The use of indicator structures is optional. The user can pass a NULL pointer in the *indpp* parameter for the `OCIDefineObject()` call. During a fetch or PL/SQL OUT bind, this means that the user is not interested in any NULLness information.
- In a SQL define or PL/SQL OUT bind, if the user passes in preallocated memory for either the output variable or the indicator, then that preallocated memory is used to store result data, and all secondary memory (out-of-line memory), if any, will get deallocated. The pre-allocated memory can either come from the cache (the result of an `OCIObjectNew()` call), or from the client's private memory space.

Note: If a client application wants to allocate memory from its own private memory space, instead of the cache, it must insure that there is no secondary out-of-line memory in the object.

- In a SQL define or PL/SQL OUT bind, if the user passes in a NULL address for the output variable or the indicator, memory for the variable or the indicator will be implicitly allocated by OCI.
- If an output object of type `SQLT_NTY` is atomically NULL (in a SQL define or PL/SQL OUT bind), only the NULL indicator struct will get allocated (implicitly if necessary) and populated accordingly to indicate the atomic NULLness of the object. The top-level object, itself, will not get implicitly allocated.
- An application can free indicators by calling `OCIObjectFree()`. If there is a top-level object (as in the case of a non-atomically NULL object), then the indicator

is freed when the top-level object is freed with *OCIObjectFree()*. If the object is atomically null, then there is no top-level object, so the indicator must be freed separately.

- The indicator struct size pointer, *indsp*, and program variable size pointer, *pgvsp*, in the *OCIDefineObject()* call is optional. Users can pass NULL if these parameters are not needed.

Information About Array Defines

For doing array defines of named data types or REFs, the user needs to pass in an array of pointers to buffers (pre-allocated or otherwise) of the appropriate type. Similarly, an array of scalar indicators (for *SQLT_REF* types) or an array of pointers to indicator structs (for *SQLT_NTY* types) needs to be passed.

Binding And Defining Oracle8 C Datatypes

Previous chapters of this book have discussed OCI bind and define operations. “Binding” on page 4-5 discussed the basics of OCI bind operations, while “Defining” on page 4-11 discusses the basics of OCI define operations. Information specific to binding and defining named data types and REFs is found in Chapter 5, “Binding and Defining”.

The sections covering basic bind and define functionality showed how an application could use a scalar variable or array of scalars as an input (bind) value in a SQL statement, or as an output (define) buffer for a query.

The sections covering named data types and REFs showed how to bind or define an object or reference. Chapter 8 expanded on this to talk about pinning object references, object navigation, and fetching embedded instances.

The purpose of this section is to cover binding and defining of individual attribute values, using the datatype mappings explained in this chapter.

Variables of one of the types defined in this chapter (e.g., **OCINumber**, **OCIString**) can typically be declared in an application and used directly in an OCI bind or define operation as long as the appropriate datatype code is specified. The following table lists the datatypes that can be used for binds and defines, along with their C mapping, and the OCI external datatype which must be specified in the *dtv* (datatype code) parameter of the bind or define call.

Table 10–1 Datatype Mappings for Binds and Defines

Datatype	C Mapping	OCI External Datatype and Code
Oracle number	OCINumber	VARNUM (SOLT_VNU)
Oracle date	OCIDate	SOLT_ODT
VARCHAR2	OCIStrng *	SOLT_VST (see Note 1 below)
RAW	OCIRaw *	SOLT_LVB (see Note 1 below)
CHAR	OCIStrng *	SOLT_VST
OBJECT	struct *	Named Data Type (SOLT_NTY)
REF	OCISref *	REF (SOLT_REF)
VARRAY	OCISarray *	Named Data Type (SOLT_NTY)
Nested Table	OCIStable *	Named Data Type (SOLT_NTY)

Note 1: Before fetching data into a define variable of type **OCIStrng ***, the size of the string must first be set using the *OCIStrngResize()* routine. This may require a describe operation to obtain the length of the select-list data. Similarly, an **OCIRaw *** must be first sized with *OCIRawResize()*.

The following section presents examples of how to use C-mapped datatypes in an OCI application.

See Also: For a discussion of OCI external datatypes, and a list of datatype codes, refer to Chapter 3, “Datatypes”.

Bind and Define Examples

The examples in this section demonstrate how variables of type **OCINumber** can be used in OCI bind and define operations.

Note: The examples in this section are intended to demonstrate the flow of calls used to perform certain OCI tasks. An expanded pseudocode is used for the examples in this section. Actual function names are used, but for the sake of simplicity not all parameters and typecasts are filled in. Additionally, other necessary OCI calls, like handle allocations, have been omitted.

Assume, for this example, that the following **person** object type was created:

```
CREATE TYPE person AS OBJECT
(name      varchar2(30),
salary    number);
```

This type is then used to create an **employees** table which has a column of type **person**.

```
CREATE TABLE employees
(emp_id    number,
job_title  varchar2(30),
emp        person);
```

OTT generates the following C struct and null indicator struct for **person**:

```
struct person
{
    OCIStrng * name;
    OCINumber salary;};
typedef struct person person;

struct person_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd salary;};
typedef struct person_ind person_ind;
```

Assume that the **employees** table has been populated with values, and an OCI application has declared a **person** variable:

```
person *my_person;
```

and fetched an object into that variable through a **SELECT** statement, like

```
text *mystmt = (text *) "SELECT person FROM employees
                        WHERE emp.name='ANDREA'";
```


This would require defining `my_person` to be the output variable for this statement, using appropriate OCI define calls for named datatypes, as described in the section “Advanced Define Operations” on page 5-16. Executing the statement would retrieve the person object named ‘ANDREA’ into the `my_person` variable.

Once the object is retrieved into `my_person`, the OCI application now has access to the attributes of `my_person`, including the name and the salary.

The application could go on to update another employee’s salary to be the same as Andrea’s, as in

```
text *updstmt = (text *) "UPDATE employees SET emp.salary = :newsal
                           WHERE emp.name = 'MONGO' "
```

Andrea’s salary (stored in `my_person->salary`) would be bound to the placeholder `:newsal`, specifying an external datatype of VARNUM (datatype code=6) in the bind operation:

```
OCIBindByName(...,":newsal",...,&my_person->salary,...,6,...);
OCISstmtExecute(...,updstmt,...)
```

Executing the statement updates Mongo’s salary in the database to be equal to Andrea’s, as stored in `my_person`.

Conversely, the application could update Andrea’s salary to be the same as Mongo’s, by querying the database for Mongo’s salary, and then making the necessary salary assignment:

```
text *selstmt = (text *) "SELECT emp.salary FROM employees
                           WHERE emp.name = 'MONGO' "

OCINumber mongo_sal;
...
OCIDefineByPos(...,1,...,&mongo_sal,...,6,...);
OCISstmtExecute(...,selstmt,...);
OCINumberAssign(...,&mongo_sal, &my_person->salary);
```

In this case, the application declares an output variable of type **OCINumber** and uses it in the define step. In this case we define an output variable for position 1, and use the appropriate datatype code (6 for VARNUM).

The salary value is fetched into the `mongo_sal` **OCINumber**, and the appropriate OCI function, `OCINumberAssign()`, is used to assign the new salary to the copy of the Andrea object currently in the cache. To modify the data in the database, the change must be flushed to the server.

3 Salary Update Examples

The examples in the previous section should give some idea of the flexibility which the new Oracle8 datatypes provide for bind and define operations. The goal of this section is to show how the same operation can be performed in several different ways. The goal is to give you some idea of the variety of ways in which these datatypes can be used in OCI applications.

The examples in this section are intended to demonstrate the flow of calls used to perform certain OCI tasks. An expanded pseudocode is used for the examples in this section. Actual function names are used, but for the sake of simplicity not all parameters and typecasts are filled in. Additionally, other necessary OCI calls, like handle allocations, have been omitted.

The Scenario

The scenario for these examples is as follows:

1. An employee named 'BRUCE' exists in the `employees` database for a hospital (see `person` type and `employees` table creation statements in previous section).
2. Bruce's current job title is 'RADIOLOGIST'.
3. Bruce is being promoted to 'RADIOLOGY_CHIEF', and along with the promotion comes a salary increase.
4. Hospital salaries are in whole dollar values, are set according to job title, and stored in a table called `salaries`, defined as follows:

```
CREATE TABLE salaries
(job_title    varchar2(20),
 salary      integer);
```

5. Bruce's salary needs to be updated to reflect his promotion.

Accomplishing the above task requires that the application retrieve the salary corresponding to 'RADIOLOGY_CHIEF' from the `salaries` table, and update Bruce's salary. A separate step would write his new title and the modified object back to the database.

Assuming that a variable of type `person` has been declared

```
person * my_person;
```

and the object corresponding to Bruce has been fetched into it, the following sections present three different ways in which the salary update could be performed.

Method 1 - fetch, convert, assign

This example uses the following method:

1. Do a traditional OCI define using an integer variable to retrieve the new salary from the database.
2. Convert the integer to an **OCINumber**.
3. Assign the new salary to Bruce.

```
#define INT_TYPE 3          /* datatype code for sword integer define */

text *getsal = (text *) "SELECT salary FROM salaries
                        WHERE job_title='RADIOLOGY_CHIEF'

sword    new_sal;
OCINumber orl_new_sal;
...
OCIDefineByPos(...,1,...,new_sal,...,INT_TYPE,...);
                        /* define int output */
OCISstmtExecute(...,getsal,...);
                        /* get new salary as int */
OCINumberFromInt(...,new_sal,...,&orl_new_sal);
                        /* convert salary to OCINumber */
OCINumberAssign(...,&orl_new_sal, &my_person->salary);
                        /* assign new salary */
```

Method 2 - fetch, assign

This method eliminates one of the steps in Method 1:

1. Define an output variable of type **OCINumber**, so that no conversion is necessary after the value is retrieved.
2. Assign the new salary to Bruce

```
#define VARNUM_TYPE 6      /* datatype code for defining VARNUM */

text *getsal = (text *) "SELECT salary FROM salaries
                        WHERE job_title='RADIOLOGY_CHIEF'

OCINumber orl_new_sal;
...
OCIDefineByPos(...,1,...,orl_new_sal,...,VARNUM_TYPE,...);
                        /* define OCINumber output */
OCISstmtExecute(...,getsal,...);      /* get new salary as OCINumber */
OCINumberAssign(...,&orl_new_sal, &my_person->salary);
                        /* assign new salary */
```

Method 3 - direct fetch

This method accomplishes the entire operation with a single define and fetch. No intervening output variable is used, and the value retrieved from the database is fetched directly into the salary attribute of the object stored in the cache.

1. Since Bruce is pinned in the object cache, use the location of his salary attribute as the define variable, and execute/fetch directly into it.

```
#define VARNUM_TYPE 6          /* datatype code for defining VARNUM */

text *getsal = (text *) "SELECT salary FROM salaries
                        WHERE job_title='RADIOLOGY_CHIEF'

...
OCIDefineByPos(...,1,...,&my_person->salary,...,VARNUM_TYPE,...);
/* define bruce's salary in cache as output variable */
OCIStmtExecute(...,getsal,...);
/* execute and fetch directly */
```

Summary and Notes

As the previous three examples show, the Oracle8 C datatypes provide flexibility for binding and defining. In these examples an integer can be fetched, and then converted to an **OCINumber** for manipulation; an **OCINumber** could be used as intermediate variable to store the results of a query; or data can be fetched directly into a desired **OCINumber** attribute of an object.

Note: In all of these examples it is important to keep in mind that in the Oracle8 OCI, if an output variable is defined before the execution of a query, the resulting data will be prefetched directly into the output buffer.

In the above examples, extra steps would be necessary to insure that changes are written to the database permanently. This may involve SQL UPDATE calls and OCI transaction commit calls.

These examples all dealt with define operations, but a similar situation applies for binding.

Similarly, although these examples dealt exclusively with the **OCINumber** type, a similar variety of operations are possible for the other Oracle8 C types described in the remainder of this chapter.

SQLT_NTY Bind/Define Example

The following code fragments demonstrate the use of SQLT_NTY bind and define calls, including *OCIBindObject()* and *OCIDefineObject()*. In each example, a previously defined SQL statement is being processed.

Bind Example

```

/*
** This example performs a SQL insert statement
*/
STATICF void insert(envhp, svchp, stmthp, errhp, insstmt, nrows)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIError *errhp;
text *insstmt;
ub2 nrows;
{
    orttdo *addr_tdo = NULLP(orttdo);
    address addr;
    null_address naddr;
    address *addr = &addr;
    null_address *naddr = &naddr;
    sword custno = 300;
    OCIBind *bndl, *bnd2p;
    ub2 i;

    /* define the application request */
    checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) insstmt,
        (ub4) strlen((char *)insstmt),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    /* bind the input variable */
    checkerr(errhp, OCIBindByName(stmthp, &bndl, errhp, (text *) ":custno",
        (sb4) -1, (dvoid *) &custno,
        (sb4) sizeof(sword), SQLT_INT,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0, (ub4) 0, (ub4 *) 0,
        (ub4) OCI_DEFAULT));

    checkerr(errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":addr",
        (sb4) -1, (dvoid *) 0,
        (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT));
}

```

```

checkerr(errhp, OCITypeByName(envhpx, errhp, svchpx, (const text *)
    SCHEMA, (ub4) strlen((char *)SCHEMA), (const text *)
    "ADDRESS_VALUE", (ub4) strlen((char *)"ADDRESS_VALUE"),
    OCI_DURATION_SESSION, &addr_tdo));

if(!addr_tdo)
{
    DISCARD printf("Null tdo returned\n");
    goto done_insert;
}

checkerr(errhp, OCIBindObject(bnd2p, errhp, addr_tdo, (dvoid **) &addr,
    (ub4 *) 0, (dvoid **) &naddr, (ub4 *) 0));

```

Define Example

```

/*
** This example executes a SELECT statement from a table which includes
** an object.
*/

STATICF void selectval(envhp, svchp, stmthp, errhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIError *errhp;
{
    orttdo *addr_tdo = NULLP(orttdo);
    OCIDefine *defn1p, *defn2p;
    address *addr = (address *)NULL;
    sword custno =0;
    sb4 status;

    /* define the application request */
    checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) selvalstmt,
        (ub4) strlen((char *)selvalstmt),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    /* define the output variable */
    checkerr(errhp, OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *)
        &custno, (sb4) sizeof(sword), SQLT_INT, (dvoid *) 0, (ub2 *)0,
        (ub2 *)0, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCIDefineByPos(stmthp, &defn2p, errhp, (ub4) 2, (dvoid *)
        0, (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,

```

```
        (ub2 *)0, (ub4) OCI_DEFAULT));

checkerr(errhp, OCITypeByName(envhpx, errhp, svchpx, (const text *)
    SCHEMA, (ub4) strlen((char *)SCHEMA), (const text *)
    "ADDRESS_VALUE", (ub4) strlen((char *)"ADDRESS_VALUE"),OROOTSES,
    &addr_tdo));

if(!addr_tdo)
{
    printf("NULL tdo returned\n");
    goto done_selectval;
}

checkerr(errhp, OCIDefineObject(defn2p, errhp, addr_tdo, (dvoid **)
    &addr, (ub4 *) 0, (dvoid **) 0, (ub4 *) 0));

checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
    (OCISnapshot *) NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));
```

Object Cache and Object Navigation

This chapter introduces the OCI's facility for working with objects in an Oracle8 server. It also discusses the OCI's object navigational function calls.

This chapter includes the following sections:

- Chapter Overview
- The Object Cache and Memory Management
- Object Navigation
- OCI Navigational Functions

Note: The functionality described in this chapter is only available if you have purchased the Oracle8 Enterprise Edition with the Objects Option.

Chapter Overview

This chapter is broken down into several main sections that cover the basic concepts involved in writing OCI applications to manipulate Oracle8 objects. The chapter also covers the OCI navigational function calls.

The following specific sections are included:

- **The Object Cache and Memory Management** - This section discusses OCI object programming in more detail, including more sophisticated options.
- **Object Navigation** - This section discusses basic object navigation using the Oracle8 OCI.
- **OCI Navigational Functions** - This section introduces the OCI functions that enable an application to navigate through a graph of objects.

Complete descriptions of the OCI navigational functions can be found in Chapter 14, “OCI Navigation and Type Functions”.

The Object Cache and Memory Management

The object cache is a client-side memory buffer that provides lookup and memory management support for objects. It stores and tracks object instances that have been fetched by an OCI application.

When objects are fetched by the application through a SQL SELECT, or through an OCI pin operation, a copy of the object is stored in the object cache. Objects that are fetched directly through a SELECT statement are fetched *by value*, and they are non-referenceable objects which cannot be pinned. Only referenceable objects may be pinned.

If an object is being pinned, and an appropriate version already exists in the cache, it does not need to be fetched from the server.

Every client program that uses the Oracle8 OCI to dereference REFs to retrieve objects utilizes the object cache. A client-side object cache is allocated for every OCI environment handle initialized in object mode. Multiple threads of a process can share the same client-side cache by sharing the same OCI environment handle.

Exactly one copy of each referenceable object exists in the cache per connection. Dereferencing a REF many times or dereferencing several equivalent REFs returns the same copy of the object.

If you modify a copy of an object in the cache, you must flush the changes to the server before they are visible to other processes. Objects that are no longer needed

can be unpinned or freed; they can then be swapped out of the cache, freeing the memory space they occupied.

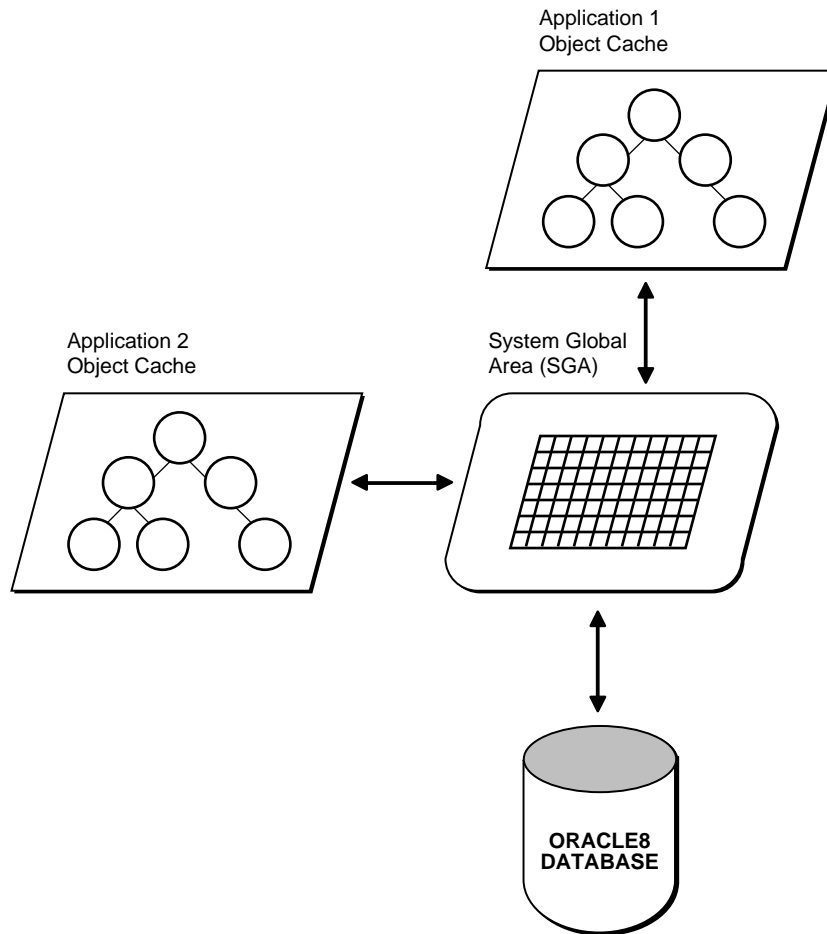
The object cache maintains the association between all object copies in the cache and their corresponding objects in the database.

The cache does not manage the contents of object copies; it does not automatically refresh object copies. The application must ensure the correctness and consistency of the contents of object copies. For example, if the application marks an object copy for insert, update, or delete, then aborts the transaction, the cache simply unmarks the object copy but does not purge or invalidate the copy. The application must pin “recent” or “latest”, or refresh the object copy in the next transaction. If it pins “any”, it may get the same object copy with its uncommitted changes from the previous aborted transaction.

See Also: For more information about pin options, see “Pinning an Object Copy” on page 11-6.

The object cache is created when the OCI environment is initialized in object mode, using *OCIInitialize()*. Each application processes running against the same server has its own object cache, as shown in Figure 11-1.

Figure 11–1 The Object Cache



The object cache tracks the objects that are currently in memory, maintains references to the objects, manages automatic object swapping, and tracks object meta-attributes.

Cache Consistency and Coherency

The object cache does not automatically maintain value coherency or consistency between object copies and their corresponding objects in the database. In other words, if an application makes changes to an object copy, the changes are not automatically applied to the corresponding object in the database, and vice versa.

The cache provides operations such as flushing a modified object copy to the database and refreshing a stale object copy with the latest value from the database to enable the program to maintain some coherency.

Note: Oracle8 does not support automatic cache coherency with the server's buffer cache or database. Automatic cache coherency refers to the mechanism by which the object cache refreshes local object copies when the corresponding objects have been modified in the server's buffer cache, and the object cache flushes the changes made to local object copies to the buffer cache before any direct access of corresponding objects in the server. Direct access includes using SQL, triggers, or stored procedures to read or modify objects in the server.

Object Cache Parameters

The object cache has two important parameters associated with it, which are attributes of the environment handle:

- OCI_ATTR_CACHE_MAX_SIZE, the maximum cache size
- OCI_ATTR_CACHE_OPT_SIZE, the optimal cache size

These parameters refer to levels of cache memory usage, and they help to determine when the cache automatically ages out eligible objects to free up memory.

If the memory occupied by the objects currently in the cache reaches or exceeds the high watermark, the cache automatically begins to free unmarked objects which have a pin count of zero. The cache continues freeing such objects until memory usage in the cache reaches the optimal size, or until it runs out of objects eligible for freeing.

OCI_ATTR_CACHE_MAX_SIZE is specified as a percentage of OCI_ATTR_CACHE_OPT_SIZE. The maximum object cache size (in bytes) is computed by incrementing OCI_ATTR_CACHE_OPT_SIZE by OCI_ATTR_CACHE_MAX_SIZE percentage:

```
maximum_cache_size = optimal_size + optimal_size * max_size_percentage / 100
```

or

```
maximum_cache_size = OCI_ATTR_CACHE_OPT_SIZE + OCI_ATTR_CACHE_OPT_SIZE *  
OCI_ATTR_CACHE_MAX_SIZE / 100
```

The default value for OCI_ATTR_CACHE_MAX_SIZE is 10%.

The default value for OCI_ATTR_CACHE_OPT_SIZE is 200k bytes.

See the section "Environment Handle Attributes" on page B-3 for more information.

Object Cache Operations

This section describes the most important functions the object cache provides to operate on object copies. All of the OCI's navigational and cache/object management functions are listed in the section "OCI Navigational Functions" on page 11-18.

Pinning and unpinning Pinning an object copy allows the application to access it in the cache by dereferencing the REF to it.

Unpinning an object indicates to the cache that the object currently is not being used. Objects should be unpinned when they are no longer needed to make them eligible for implicit freeing by the cache, thus freeing up memory.

Freeing Freeing an object copy removes it from the cache and frees its memory.

Marking and unmarking Marking an object notifies the cache that an object copy has been updated in the cache and the corresponding object must be updated in the server when the object copy is flushed.

Unmarking an object removes the indication that the object has been updated.

Flushing Flushing an object writes local changes made to marked object copies in the cache to the corresponding objects in the server. When this happens, the copies in the object cache are unmarked.

Refreshing Refreshing an object copy in the cache replaces it with the latest value of the corresponding object in the server.

Note: Pointers to top-level object memory are valid after a refresh. Pointers to secondary-level memory (e.g., string text pointers, collections, etc.) may become invalid after a refresh.

Operations for Loading and Removing Object Copies

Pin, unpin, and free functions are discussed in this section.

Pinning an Object Copy

When an application needs to dereference a REF in the object cache, it calls *OCIObjectPin()*. This call dereferences the REF and pins the object copy in the cache. As long as the object copy is pinned, it is guaranteed to be accessible by the application. Another variation of *OCIObjectPin()* is *OCIObjectPinArray()* which takes an array of REFs, dereferences the REFs, and pins the object copies. Both

OCIObjectPin() and *OCIObjectPinArray()* take a pin option, “any”, “recent”, or “latest”. The datatype of the pin option is **OCIPinOpt**.

- If the “any” (OCI_PIN_ANY) option is specified, the object cache immediately returns the object copy that is already in the cache, if there is one. If no copy is in the cache, the object cache loads the latest object copy from the database and then returns the object copy. The “any” option is appropriate for read-only, informational, fact, or meta objects (such as products, salesmen, vendors, regions, parts, offices, etc.). These objects usually don't change often, and even if they change, the change does not affect the application.
- If the “latest” (OCI_PIN_LATEST) option is specified, the object cache loads into the cache the latest object copy from the database and returns that copy unless the object copy is locked in the cache, in which case the marked object copy is returned immediately. If the object is already in the cache and not locked, the latest object copy is loaded and overwrites the existing one. The “latest” option is appropriate for operational objects (such as purchase orders, bugs, line items, bank accounts, stock quotes, etc.); these objects usually change often, and the program cares to access these objects at their latest possible state.
- If the “recent” (OCI_PIN_RECENT) option is specified, there are two possibilities:
 - a. If, in the same transaction, the object copy has been previously pinned using the “latest” or “recent” option, the “recent” option becomes equivalent to the “any” option. Otherwise,
 - b. the “recent” option becomes equivalent to the “latest” option.

When the program pins an object, the program also specifies one of two possible values for the pin duration: “session” or “transaction”. The datatype of the duration is **OCIDuration**.

- If the pin duration is “session” (OCI_DURATION_SESSION), the object copy remains pinned until the end of session (i.e., end of connection) or until it is unpinned explicitly by the program (by calling *OCIObjectUnpin()*).
- If the pin duration is “transaction” (OCI_DURATION_TRANS), the object copy remains pinned until the end of transaction or until it is unpinned explicitly.

When loading an object copy into the cache from the database, the cache effectively executes

```
SELECT VALUE(t) FROM t WHERE REF(t) = :r
```

where τ is the object table storing the object, and r is the REF, and the fetched value becomes the value of the object copy in the cache.

Since the object cache effectively executes a separate `SELECT` statement to load each object copy into the cache, in a read-committed transaction, object copies are not guaranteed to be read-consistent with each other.

In a serializable transaction, object copies (pinned “recent” or “latest”) are read-consistent with each other because the `SELECT` statements to load these object copies are executed based on the same database snapshot.

The object cache model is orthogonal to or independent of the Oracle transaction model. The behavior of the object cache does not change based on the transaction model, even though the objects that are retrieved from the server through the object cache can be different when running the same program under different transaction models (e.g., read committed versus serializable).

Unpinning an Object Copy

An object copy can be unpinned when it is no longer used by the program. It then becomes available to be freed. An object copy must be both completely unpinned and unmarked in order to become eligible to be implicitly freed by the cache when the cache begins to run out of memory. To be completely unpinned, an object copy that has been pinned N times must be unpinned N times.

An unpinned but marked object copy is not eligible for implicit freeing until the object copy is flushed or explicitly unmarked by the user. However, the object cache implicitly frees object copies only when it begins to run out of memory, so an unpinned object copy need not necessarily be freed. If it has not been implicitly freed and is pinned again (with the any or recent options), the program gets the same object copy.

An application calls `OCIObjectUnpin()` or `OCIObjectPinCountReset()` to unpin an object copy. In addition, a program can call `OCICacheUnpin()` to completely unpin all object copies in the cache for a specific connection.

Freeing an Object Copy

Freeing an object copy removes it from the object cache and frees up its memory. The cache supports two methods for freeing up memory:

1. Explicit freeing - A program explicitly frees or removes an object copy from the cache by calling `OCIObjectFree()` which takes an option to (forcefully) free either a marked or pinned object copy. The program can also call `OCICacheFree()` to free all object copies in the cache.

2. Implicit freeing - Should the cache begin to run out of memory, it implicitly frees object copies that are both unpinned and unmarked. Unpinned objects that are marked are eligible for implicitly freeing only when the object copy is flushed or unmarked. For more information, see the section “Object Cache Parameters” on page 11-5.

For memory management reasons, it is important that applications unpin objects when they are no longer needed. This makes these objects available for aging out of the cache, and makes it easier for the cache to free memory when necessary.

The OCI does not provide a function to free unreferenced objects in the client-side cache.

Operations for Making Changes to Object Copies

Functions for marking and unmarking object copies are discussed in this section.

Marking an Object Copy

An object copy can be created, updated, and deleted locally in the cache. If the object copy is created in the cache (by calling *OCIObjectNew()*), the object copy is marked for insert by the object cache, so that the object will be inserted in the server when the object copy is flushed.

If the object copy is updated in the cache, the user has to notify the object cache by marking the object copy for update (by calling *OCIObjectMarkUpdate()*). When the object copy is flushed, the corresponding object in the server is updated with the value in the object copy.

If the object copy is deleted, the object copy is marked for delete in the object cache (by calling *OCIObjectMarkDelete()*). When the object copy is flushed, the corresponding object in the server is deleted. The memory of the marked object copy is not freed until it is flushed and unpinned. When pinning an object marked for delete, the program receives an error, as if the program is dereferencing a dangling reference.

When a user makes multiple changes to an object copy, it is the final results of these changes which are applied to the object in the server when the copy is flushed. For example, if the user updates and deletes an object copy, the object in the server is simply deleted when the object copy is flushed. Similarly, if an attribute of an object copy is updated multiple times, it is the final value of this attribute which is updated in the server when the object copy is flushed.

The program can mark an object copy as updated or deleted only if the object copy has been loaded into the object cache.

Unmarking an Object Copy

A marked object copy can be unmarked in the object cache. By unmarking a marked object copy, the changes that are made to the object copy are not flushed to the server. The object cache does not undo the local changes that are already made to the object copy.

A program calls *OCIObjectUnmark()* to unmark an object. In addition, a program can call *OCICacheUnmark()* to unmark all object copies in the cache for a specific connection.

Operations for Synchronizing Object Copies with Server

Cache/server synchronization operations (flushing, refreshing) are discussed in this section.

Flushing Changes to Server

The local changes made to a marked object copy in the cache are written to the server when the object copy is flushed. The program can call *OCIObjectFlush()* to flush a single object copy or *OCICacheFlush()* to flush all marked object copies in the cache or a list of selected marked object copies. *OCICacheFlush()* flushes objects associated with a specific service context.

After flushing an object copy, the object copy is unmarked. (Note that the object is locked in the server after it is flushed; the object copy is therefore marked as locked in the cache.)

Note: The *OCICacheFlush()* operation incurs only a single server roundtrip even if multiple objects are being flushed.

If an application wishes to flush only dirty objects of a certain type, this functionality is available through the callback function which is an optional argument to the *OCICacheFlush()* call. The application can define a callback which returns only the desired objects. In this case the operation still incurs only a single server roundtrip for the flush.

Refreshing an Object Copy

When refreshed, an object copy is reloaded with the latest value of the corresponding object in the server. The latest value may contain changes made by other committed transactions and changes made directly (not through the object cache) in the server by the transaction. The program can change objects directly in the server using SQL DML, triggers, or stored procedures.

To refresh a marked object copy, the program must first unmark the object copy. An unpinned object copy is simply freed when it is refreshed (i.e., when the whole cache is refreshed).

The program can call *OCIObjectRefresh()* to refresh a single object copy or *OCICacheRefresh()* to refresh all object copies in the cache, all object copies that are loaded in a transaction (i.e., object copies that are pinned recent or pinned latest), or a list of selected object copies.

When an object is flushed to the server, triggers can be fired to modify more objects in the server. The same objects (modified by the triggers) in the object cache become out-of-date, and must be refreshed before they can be locked or flushed.

The various meta-attribute flags and durations of an object are modified as described in Table 11–1 after being refreshed:

Table 11–1 Object Attributes After Refresh

Object Attribute	Status After Refresh
existent	set to appropriate value
pinned	unchanged
flushed	reset
allocation duration	unchanged
pin duration	unchanged

During refresh, the object cache loads the new data into the top-level memory of an object copy, thus reusing the top level memory. The top-level memory of an object copy contains the in-line attributes of the object. On the other hand, the memory for the out-of-line attributes of an object copy may be freed and relocated, since the out-of-line attributes can vary in size.

See Also: See the section “Memory Layout of an Instance” on page 11-15 for more information about object memory.

Other Operations

Other pertinent OCI functions are discussed in this section.

Locking Objects For Update

The program can optionally call *OCIObjectLock()* to lock an object for update. This call instructs the object cache to get a row lock on the object in the database. This is similar to executing

```
SELECT NULL FROM t WHERE REF(t) = :r FOR UPDATE
```

where *t* is the object table storing the object to be locked and *r* is the REF identifying the object. The object copy is marked locked in the object cache after *OCIObjectLock()* is called.

To lock a graph or set of objects, several *OCIObjectLock()* calls are required, one per object, or the array pin *OCIObjectArrayPin()* call can be used for better performance.

By locking an object, the application is guaranteed that the object in the cache is up-to-date. No other transaction can modify the object while the application has it locked.

At the end of a transaction, all locks are released automatically by the server. The locked indicator in the object copy is reset.

Implementing Optimistic Locking

It is possible to implement optimistic locking in an OCI application if you run your transactions at the serializable level.

The Oracle8 OCI supports calls that allow you to dereference and pin objects in the object cache without locking them, modify them in the cache (again without locking them), and then flush them (the dirtied objects) to the database.

During the flush, if a dirty object has been modified by another committed transaction since the beginning of your transaction, a non-serializable transaction error is returned. If none of the dirty objects has been modified by any other any other transaction since the beginning of your transaction, then the changes are written to the database successfully.

Note: *OCITransCommit()* first flushes dirty objects into the database before committing a transaction.

The above mechanism effectively implements an optimistic locking model.

Commit and Rollback in Object Applications

When a transaction is committed (*OCITransCommit()*), all marked objects are flushed to the server. If an object copy is pinned with a transaction duration, the object copy is unpinned.

When a transaction is rolled back, all marked objects are unmarked. If an object copy is pinned with a transaction duration, the object copy is unpinned.

Object Duration

In order to maintain free space in memory, the object cache attempts to reuse objects' memory whenever possible. The object cache reuses an object's memory when the object's lifetime (*allocation duration*) expires or when the object's *pin duration* expires. The allocation duration is set when an object is created with *OCIObjectNew()*, and the pin duration is set when an object is pinned with *OCIObjectPin()*. The datatype of the duration value is **OCIDuration**.

Note: The pin duration for an object cannot be longer than the object's allocation duration.

When an object reaches the end of its allocation duration, it is automatically deleted and its memory can be reused. The pin duration indicates when an object's memory can be reused, and memory is reused when the cache is full.

The OCI supports two predefined durations:

1. transaction (OCI_DURATION_TRANS)
2. session (OCI_DURATION_SESSION)

The *transaction duration* expires when the containing transaction ends (commits or aborts). The *session duration* expires when the containing session/connection ends.

The application can explicitly unpin an object using *OCIObjectUnpin*. To minimize explicit unpinning of individual objects, the application can unpin all objects currently pinned in the object cache using the function *OCICacheUnpin*. By default, all objects are unpinned at the end of the pin duration.

Durations Example

Table 11–2 illustrates the use of the different durations in an application. Four objects are created or pinned in this application over the course of one connection and three transactions. The first column indicates the action performed by the database, and the second column indicates the function which performs the action.

The remaining columns indicate the states of the various objects at each point in the application.

For example, Object 1 comes into existence at T2 when it is created with a connection duration, and it exists until T19 when the connection is terminated. Object 2 is pinned at T7 with a transaction duration, after being fetched at T6, and it remains pinned until T9 when the transaction is committed.

Table 11–2 Example of Allocation and Pin Durations

Time	Application Action	Function	Object 1	Object 2	Object 3	Object 4
T ₁	Establish connection					
T ₂	Create object 1 - allocation duration = connection	OCIObjectNew()	exists			
T ₅	Start Transaction1	OCITransStart()	exists			
T ₆	SQL - fetch REF to object 2		exists			
T ₇	Pin object 2 - pin duration = transaction	OCIObjectPin()	exists	pinned		
T ₈	Process application data		exists	pinned		
T ₉	Commit Transaction1	OCITransCommit()	exists	unpinned		
T ₁₀	Start Transaction2	OCITransStart()	exists			
T ₁₁	Create object 3 - allocation duration = transaction	OCIObjectNew()	exists		exists	
T ₁₂	SQL - fetch REF to object 4		exists		exists	
T ₁₃	Pin object 4 - pin duration = connection	OCIObjectPin()	exists		exists	pinned
T ₁₄	Commit Transaction2	OCITransCommit()	exists		deleted	pinned
T ₁₅	Terminate session1	OCIDurationEnd()	exists			pinned
T ₁₆	Start Transaction3	OCITransStart()	exists			pinned
T ₁₇	Process application data		exists			pinned
T ₁₈	Commit Transaction3	OCITransCommit()	exists			pinned
T ₁₉	Terminate connection		deleted			unpinned

See Also: See the descriptions of *OCIObjectNew()* and *OCIObjectPin()* in Chapter 14 for specific information about parameter values which can be passed to these functions.

See the section “Creating, Freeing, and Copying Objects” on page 8-31 for information about freeing up an object’s memory before its allocation duration has expired.

Memory Layout of an Instance

An instance in memory is composed of a top-level memory chunk of the instance, a top-level memory of the null indicator structure and optionally, a number of secondary memory chunks. Consider a DEPARTMENT row type,

```
CREATE TYPE department AS OBJECT
( dep_name      varchar2(20),
  budget        number,
  manager       person,           /* person is an object type */
  employees     person_array );  /* varray of person objects */
```

and its C representation

```
struct department
{
  OCIStrng * dep_name;
  OCINumber budget;
  struct person manager;
  OCIArray * employees;
};
typedef struct department department;
```

Each instance of DEPARTMENT has a top-level memory chunk which contains the top-level attributes such as *dep_name*, *budget*, *manager* and *employees*. The attributes *dep_name* and *employees* are themselves actually pointers to the additional memory (the secondary memory chunks). The secondary memory is used to contain the actual data for the embedded instances (e.g. *employees* varray and *dep_name* string).

The top-level memory of the null indicator structure contains the null statuses of the attributes in the top level memory chunk of the instance. From the above example, the top level memory of the null structure contains the null statuses of the attributes *dep_name*, *budget*, *manager* and the atomic nullness of *employees*.

Object Navigation

This section discusses how OCI applications can navigate through graphs of objects in the object cache.

Simple Object Navigation

In the example in the previous sections, the object retrieved by the application was a simple object, whose attributes were all scalar values. If an application retrieves an object with an attribute which is a REF to another object, the application can use OCI calls to traverse the *object graph* and access the referenced instance.

As an example, consider the following declaration for a new type in the database:

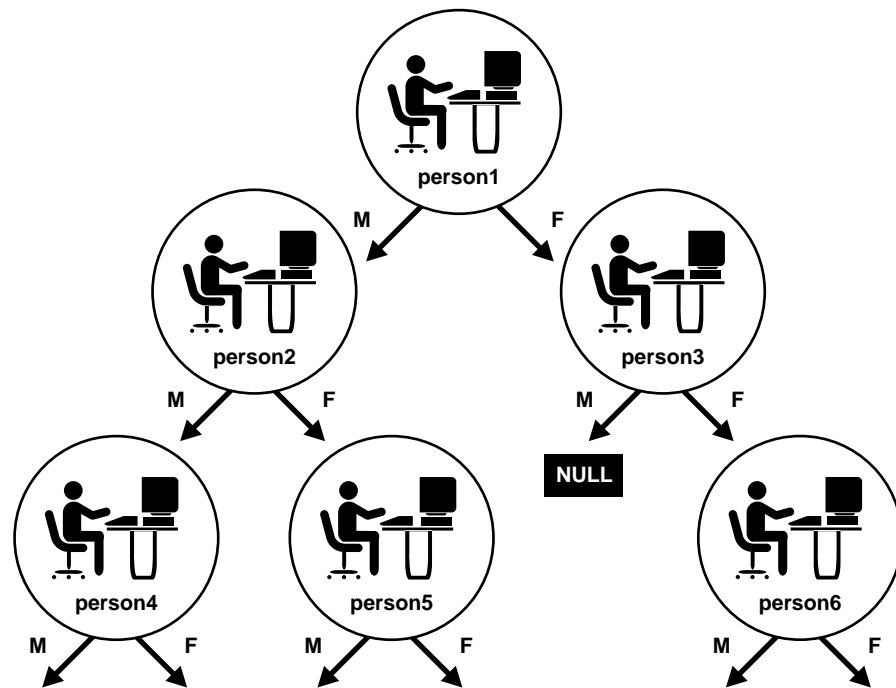
```
CREATE TYPE person_t AS OBJECT
(   name          VARCHAR2(30),
    mother         REF person_t,
    father         REF person_t);
```

An object table of `person_t` objects is created with the following statement:

```
CREATE TABLE person_table OF person_t;
```

Instances of the `person_t` type can now be stored in the typed table. Each instance of `person_t` includes references to two other objects, which would also be stored in the table. A NULL reference could represent a parent about whom information is not available.

An object graph is a graphical representation of the REF links between object instances. For example, on the following page depicts an object graph of `person_t` instances, showing the links from one object to another. The circles represent objects, and the arrows represent references to other objects.

Figure 11–2 Object Graph of *person_t* Instances

In this case, each object has links to two other instances of the same object. This need not always be the case. Objects may have links to other object types. Other types of graphs are also possible. For example, if a set of objects is implemented as a linked list, the object graph could be viewed as a simple chain, where each object references the previous and/or next objects in the linked list.

You can use the methods described earlier in this chapter to retrieve a reference to a *person_t* instance and then pin that instance. The OCI provides functionality which allows you to traverse the object graph by following a reference from one object to another.

As an example, assume that an application fetches the *person1* instance in the above graph and pins it as *pers_1*. Once that has been done, the application can access the mother instance of *person1* and pin it into *pers_2* through a second pin operation:

```
OCIObjectPin(env, err, pers_1->mother, OCI_PIN_ANY, OCI_DURATION_TRANS,
OCI_LOCK_X, (OCIComplexObject *) 0, &pers_2);
```

In this case, an OCI fetch operation is not required to retrieve the second instance.

The application could then pin the father instance of `person1`, or it could operate on the reference links of `person2`.

Note: Attempting to pin a NULL or dangling REF results in an error on the `OCIObjectPin()` call.

OCI Navigational Functions

This section provides a brief summary of the available OCI navigational functions. The functions are grouped according to their general functionality. More detailed descriptions of each of these functions can be found in Chapter 14, “OCI Navigation and Type Functions”.

The use of these functions is described in the earlier sections of this chapter.

The navigational functions follow a naming scheme which uses different prefixes for different types of functionality:

OCICache*() - these functions are Cache operations

OCIObject*() - these functions are individual Object operations

Pin/Unpin/Free Functions

The following functions are available to pin, unpin, or free objects:

Function	Purpose
<code>OCIObjectPin()</code>	Pin an object
<code>OCIObjectUnpin()</code>	Unpin an object
<code>OCIObjectPinCountReset()</code>	Unpin an object to zero pin count
<code>OCICacheUnpin()</code>	Unpin persistent objects in cache or connection
<code>OCIObjectArrayPin()</code>	Pin an array of references
<code>OCIObjectPinTable()</code>	Pin a table object with a given duration
<code>OCICacheFree()</code>	Free all instances in the cache
<code>OCIObjectFree()</code>	Free and unpin a standalone instance

Flush and Refresh Functions

The following functions are available to flush modified objects to the server:

Function	Purpose
OCICacheFlush()	Flush modified persistent objects in cache to server
OCIObjectFlush()	Flush a modified persistent object to the server
OCICacheRefresh()	Refresh pinned persistent objects in the cache
OCIObjectRefresh()	Refresh a single persistent object

Mark and Unmark Functions

The following functions allow an application to mark or unmark an object by modifying one of its meta-attributes:

Function	Purpose
OCIObjectMarkDelByRef()	Mark an object deleted given a REF
OCIObjectMarkUpd()	Mark an object as updated/dirty
OCIObjectMarkDel()	Mark an object deleted / delete a value instance
OCICacheUnmark()	Unmarks all objects in the cache
OCIObjectUnmark()	Marks a given object as updated
OCIObjectUnmarkByRef()	Marks an object as updated, given a REF

Object Meta-Attribute Accessor Functions

The following functions allow an application to access the meta-attributes of an object:

Function	Purpose
OCIObjectExists()	Get existence status of an instance
OCIObjectFlushStatus()	Get the flush status of an instance
OCIObjectGetInd()	Get null structure of an instance
OCIObjectIsDirtied()	Has an object been marked as updated?
OCIObjectIsLocked()	Is an object locked?

Other Functions

The following functions provide additional object functionality for OCI applications:

Function	Purpose
OCIObjectCopy()	Copy one instance to another
OCIObjectGetObjectRef()	Return reference to a given object
OCIObjectGetTypeRef()	Get a reference to a TDO of an instance
OCIObjectLock()	Lock a persistent object
OCIObjectNew()	Create a new instance

Using the Object Type Translator

This chapter discusses the Object Type Translator (OTT), which is used to map database object types and named collection types to C structs for use in OCI and Pro*C/C++ applications. The chapter includes the following sections:

- OTT Overview
- Using the Object Type Translator
- Using the OTT with OCI Applications
- OTT Reference

Note: For information specific to Pro*C/C++, please refer to the *Pro*COBOL Precompiler Programmer's Guide*.

Note: The functionality described in this chapter is only available if you have purchased the Oracle8 Enterprise Edition with the Objects Option.

OTT Overview

OTT (The Object Type Translator) is a new product released with Oracle8. It assists in the development of C language applications which make use of user-defined types in an Oracle8 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 OCI or Pro*C/C++ programmer can access objects stored in the tables.

An application which accesses object data needs to be able to represent the data in a host language format. This is accomplished by representing object types as C structs. It would be possible for a programmer to code struct declarations by hand to represent database object types, but this can be very time-consuming and error-prone if many types are involved. The OTT simplifies this step by automatically generating appropriate struct declarations. For Pro*C/C++, the application only needs to include the header file generated by the OTT. In OCI, the application also needs to call an initialization function generated by the OTT.

In addition to creating structs which represent stored datatypes, the OTT also generates parallel indicator structs which indicate whether an object type or its fields are null.

Using the Object Type Translator

The Object Type Translator (OTT) converts database definitions of object types and named collection types into C struct declarations which can be included in an OCI or Pro*C/C++ application.

You must explicitly invoke the OTT to translate database types to C representations. You must also 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.

On most operating systems, the OTT 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* and an optional *implementation file*. The following is an example of a command which invokes the OTT:

```
ott userid=scott/tiger intype=demo.in.typ outtype=demo.out.typ code=c hfile=demo.h
```

This command causes the OTT to connect to the database with username 'scott' and password 'tiger', and translate database types to C structs, based on instructions in the *intype file* (*demo.in.typ*). The resulting structs 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 `demo.in.typ` file:

```
CASE=LOWER
TYPE employee
```

Sample `demoout.typ` file:

```
CASE = LOWER
TYPE EMPLOYEE AS employee
  VERSION = "$8.0"
  HFILE = demo.h
```

In this example, the `demo.in.typ` file contains the type to be translated, preceded by `TYPE` (e.g., `TYPE employee`). The structure of the `outtype` file is similar to the `intype` file, with the addition of information obtained by the OTT.

Once the OTT has completed the translation, the header file contains a C struct representation of each type specified in the `intype` file, and a null indicator struct corresponding to each type. For example, if the `employee` type listed in the `intype` file was defined as

```
CREATE TYPE employee AS OBJECT
(
  name      VARCHAR2(30),
  empno     NUMBER,
  deptno    NUMBER,
  hiredate  DATE,
  salary    NUMBER
);
```

the header file generated by the OTT (`demo.h`) includes, among other items, the following declarations:

```
struct employee
{
  OCIStr * name;
  OCINumber empno;
  OCINumber deptno;
  OCIDate hiredate;
  OCINumber salary;
};
```

```
typedef struct emp_type emp_type;

struct employee_ind
{
    OCInd _atomic;
    OCInd name;
    OCInd empno;
    OCInd deptno;
    OCInd hiredate;
    OCInd salary;
};
typedef struct employee_ind employee_ind;
```

Note: Parameters in the intype file control the way generated structs are named. In this example, the struct name `employee` matches the database type name `employee`. The struct name is in lower case because of the line `CASE=lower` in the intype file.

The datatypes which appear in the struct declarations (e.g., `OCString`, `OCInd`) are special datatypes which are new to Oracle8. For more information about these types, see the section “OTT Datatype Mappings” on page 12-9.

The following sections describe these aspects of using the OTT:

- Creating Types in the Database
- Invoking the OTT
- The OTT Command Line
- The Intype File
- OTT Datatype Mappings
- Null Indicator Structs
- The Outtype File

The remaining sections of the chapter discuss the use of the OTT with OCI, followed by a reference section which describes command line syntax, parameters, intype file structure, nested `#include` file generation, schema names usage, default name mapping, and restrictions.

Creating Types in the Database

The first step in using the OTT 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.

See Also: For information about the CREATE TYPE statement, refer to the *Oracle8 SQL Reference*.

Invoking the OTT

The next step is to invoke the OTT. OTT parameters can be specified on the command line, or in a file called a configuration file. Certain parameters can also be specified in the INTYPE file.

If a parameter is specified in more than one place, its value on the command line will take precedence over its value in the INTYPE file, which takes precedence over its value in a user-defined configuration file, which takes precedence over its value in the default configuration file.

Command Line

Parameters (also called options) set on the command line override any set elsewhere. See the next section, "The OTT Command Line", for more information.

Configuration File

A configuration file is a text file that contains OTT parameters. Each non-blank line in the file contains one parameter, with its associated value or values. If more than one parameter is put on a line, only the first one will be used. No whitespace may occur on any non-blank line of a configuration file.

A configuration file may 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 system-specific. For example, on Solaris, the file specification is *\$ORACLE_HOME/precomp/admin/ottcfg.cfg*. See your platform-specific documentation for further information.

INTYPE File

The INTYPE file gives a list of types for the OTT to translate.

The parameters CASE, HFILE, INITFUNC, and INITFILE can appear in the INTYPE file. See "The Intype File" on page 12-8 for more information.

The OTT Command Line

On most platforms, the OTT is invoked on the command line. You can specify the input and output files, and the database connection information, among other things. Consult your platform-specific documentation to see how to invoke the OTT on your platform.

Example 1 The following is an example of an OTT invocation from the command line:

```
ott userid=bren/bigkitty intype=demoin.typ outtype=demoout.typ code=c hfile=demo.h
```

Note: No spaces are permitted around the equals sign (=).

The following sections describe the elements of the command line used in this example.

For a detailed discussion of the various OTT command line options, please refer to the section “OTT Reference” on page 12-22.

OTT

Causes the OTT to be invoked. It must be the first item on the command line.

userid

Specifies the database connection information which the OTT will use.

In Example 1, the OTT will attempt to connect with username ‘bren’ and password ‘bigkitty’.

intype

Specifies the name of the intype file which will be used.

In Example 1, the name of the intype file is specified as `demoin.typ`.

outtype

Specifies the name of the outtype file. When the OTT generates the C header file, it also writes information about the translated types into the outtype file. This file contains an entry for each of the types which is translated, including its version string, and the header file to which its C representation was written.

In “Example 1” on page 12-6, the name of the outtype file is specified as `demoout.typ`.

Note: If the file specified by the `outtype` keyword already exists, it is overwritten when the OTT runs. If the name of the `outtype` file is the same as the name of the `intype` file, the `outtype` information overwrites the `intype` file.

code

Specifies the target language for the translation. The following options are available:

- C (equivalent to ANSI_C)
- ANSI_C (for ANSI C)
- KR_C (for Kernighan & Ritchie C)

There is currently no default option, so this parameter is required.

Struct declarations are identical in both C dialects. The style in which the initialization function defined in the `INITFILE` file is defined depends on whether `KR_C` is used. If the `INITFILE` option is not used, all three options are equivalent.

hfile

Specifies the name of the C header file to which the generated structs should be written.

In “Example 1” on page 12-6, the generated structs will be stored in a file called `demo.h`.

Note: If the file specified by the `hfile` keyword already exists, it will be overwritten when the OTT runs, with one exception: if the contents of the file as generated by the OTT are identical to the previous contents of the file, the OTT will not actually write to the file. This preserves the modification time of the file so that UNIX `make` and similar facilities on other platforms do not perform unnecessary recompilations.

initfile

Specifies the use of the C source file into which the type initialization function is to be written.

Note: If the file specified by the `initfile` keyword already exists, it will be overwritten when the OTT runs, with one exception: if the contents of the file as generated by the OTT are identical to the previous contents of the file, the OTT will not actually write to the file. This preserves the modification time of the file so that UNIX `make` and similar facilities on other platforms do not perform unnecessary recompilations.

initfunc

Specifies the name of the initialization function to be defined in the initfile.

If this parameter is not used and an initialization function is generated, the name of the initialization function will be the same as the base name of the initfile.

The Intype File

When running the OTT, the INTYPE file tells the OTT which database types should be translated, and it can also control the naming of the generated structs. The intype file can be a user-created file, or it may be the outtype file of a previous invocation of the OTT. If the INTYPE parameter is not used, all types in the schema to which the OTT connects are translated.

The following is a simple 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
```

The first line, with the CASE keyword, indicates that generated C identifiers should be in lower case. However, this CASE option 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 would be named in lower case.

See Also: See the description of “case” on page 12-27 for further information regarding the CASE option.

The lines which 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.

The TRANSLATE...AS keywords specify that the name of an object attribute should be changed when the type is translated into a C struct. 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 struct. In this case, the purchase_order database type is translated into a struct called p_o.

If AS is not used to translate a type or attribute name, the database name of the type or attribute will be used as the C identifier name, except that the CASE option will be observed, and any characters that cannot be mapped to a legal C identifier character will be replaced by an underscore. Reasons for translating a type or attribute name include:

- 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 may happen, for example, if the program uses two types with the same name from different schemas.
- the programmer prefers a different name

The OTT may need to translate additional types which are not listed in the intype file. This is because the OTT analyzes the types in the intype file for type dependencies before performing the translation, and 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, the OTT would still translate ADDRESS because it is required to define the "Person" type.

A normal case-insensitive SQL identifier can be spelled in any combination of upper and lower case 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, e.g., 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, e.g., TYPE "CASE". When a name is quoted for this reason, the quoted name must be in upper case if the SQL identifier was created in a case-insensitive manner, e.g., CREATE TYPE Case. If an OTT-reserved word is used to refer to the name of a SQL identifier but is not quoted, the OTT will report a syntax error in the INTYPE file.

See Also: For a more detailed specification of the structure of the intype file and the available options, refer to the section "Structure of the Intype File" on page 12-29.

OTT Datatype Mappings

When the OTT generates a C struct from a database type, the struct contains one element corresponding to each attribute of the object type. The datatypes of the attributes are mapped to types which are used in Oracle8's object data types. The

datatypes found in Oracle8 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 in Oracle8 includes standard types which are familiar to most programmers, including number and character types. It also includes new datatypes being introduced with Oracle8 (e.g., BLOB, CLOB).

Oracle8 also includes a set of predefined types which are used to represent object type attributes in C structs. As an example, consider the following object type definition, and its corresponding OTT-generated struct declarations:

```
CREATE TYPE employee AS OBJECT
(   name          VARCHAR2(30),
    empno         NUMBER,
    deptno        NUMBER,
    hiredate      DATE,
    salary$       NUMBER);
```

The OTT output, assuming CASE=LOWER and no explicit mappings of type or attribute names, is:

```
struct employee
{
    OCIStrng * name;
    OCINumber empno;
    OCINumber department;
    OCIDate   hiredate;
    OCINumber salary_;
};
typedef struct emp_type emp_type;
struct employee_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd department;
    OCIInd hiredate;
    OCIInd salary_;
}
typedef struct employee_ind employee_ind;
```

The indicator struct (struct employee_ind) is explained in the section, “Null Indicator Structs” on page 12-15.

The datatypes in the struct declarations—**OCIStrng**, **OCINumber**, **OCIDate**, **OCIInd**—are new C mappings of object types being introduced with Oracle8. They are used here to map the datatypes of the object type attributes. The *number*

datatype of the `empno` attribute, maps to the new **OCINumber** datatype, for example. These new datatypes can also be used as the types of bind and define variables.

Mapping Object Datatypes to C

This section describes the mappings of Oracle8 object attribute types to C types generated by the OTT. The following section “OTT Type Mapping Example” on page 12-12 includes examples of many of these different mappings. The following table lists the mappings from types which can be used as attributes to object datatypes which are generated by the OTT.

Table 12–1 Object Datatype Mappings for Object Type Attributes

Object Attribute Types	C Mapping
VARCHAR2(N)	OCIStrng *
VARCHAR(N)	OCIStrng *
CHAR(N), CHARACTER(N)	OCIStrng *
NUMBER, NUMBER(N), NUMBER(N,N)	OCINumber
NUMERIC, NUMERIC(N), NUMERIC(N,N)	OCINumber
REAL	OCINumber
INT, INTEGER, SMALLINT	OCINumber
FLOAT, FLOAT(N), DOUBLE PRECISION	OCINumber
DEC, DEC(N), DEC(N,N)	OCINumber
DECIMAL, DECIMAL(N), DECIMAL(N,N)	OCINumber
DATE	OCIDate
BLOB	OCIBlobLocator *
CLOB	OCIClobLocator *
BFILE	OCIBfileLocator *
Nested Object Type	C name of the nested object type
REF	declared using typedef; equivalent to OCIStrng * See the following example.
RAW(N)	OCIRaw *

The next table shows the mappings of named collection types to Oracle8 object datatypes generated by the OTT:

Table 12–2 Object Datatype Mappings for Collection Types

Named Collection Type	C Mapping
VARRAY	declared using typedef; equivalent to OCIArray * See the following example.
NESTED TABLE	declared using typedef; equivalent to OCITable * See the following example.

Note: For REF, VARRAY, and NESTED TABLE types, the OTT generates a typedef. The type declared in the typedef is then used as the type of the data member in the struct declaration. For examples, see the next section, “OTT Type Mapping Example”.

If an object type includes an attribute of a REF or collection type, a typedef for the REF or collection type is first generated. Then the struct declaration corresponding to the object type is generated. The struct 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, the OTT first generates the nested type. It then maps the object type attribute to a nested struct of the type of the nested object type.

The Oracle8 C datatypes to which the OTT maps non-object database attribute types are structures, which, except for **OCIDate**, are opaque.

OTT Type Mapping Example

The following example is presented to demonstrate the various type mappings created by the OTT.

Given the following database types

```
CREATE TYPE my_varray AS VARRAY(5) of integer;

CREATE TYPE object_type AS OBJECT
(object_name  VARCHAR2(20));

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));

```

and an intype file which includes

```

CASE = LOWER
TYPE many_types

```

the OTT would generate the following C structs:

Note: Comments are provided here to help explain the structs. These comments are not part of actual OTT output.

```

#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE

#ifndef OCI_ORACLE
#include <oci.h>
#endif

typedef OCISRef many_types_ref;
typedef OCISRef object_type_ref;
typedef OCIArray my_varray;           /* part of many_types */
typedef OCISTable my_table;          /* part of many_types */
typedef OCISRef other_type_ref;
struct object_type                    /* part of many_types */
{
    OCISString * 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
{
    OCIStr *      the_varchar;
    OCIStr *      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. As described in the section “The OTT Command Line” on page 12-6, the OTT 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 ref in an object type attribute. For example, although the `many_types` type contains the attribute `another_ref REF other_type`, a declaration of struct `other_type` was not generated.

This example also illustrates how typedefs are used to declare VARRAY, NESTED TABLE, and REF types.

The typedefs occur near the beginning:

```
typedef OCISRef many_types_ref;
typedef OCISRef object_type_ref;
typedef OCISArray my_varray;
typedef OCISTable my_table;
typedef OCISRef other_type_ref;
```

In the struct `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;
    ...
}
```

Null Indicator Structs

Each time the OTT generates a C struct to represent a database object type, it also generates a corresponding null indicator struct. When an object type is selected into a C struct, null indicator information may be selected into a parallel struct.

For example, the following null indicator struct was generated in the example in the previous section:

```
struct many_types_ind
{
    OCISInd _atomic;
    OCISInd the_varchar;
    OCISInd the_char;
    OCISInd the_blob;
    OCISInd the_clob;
```

```

struct object_type_ind the_object;
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;

```

The layout of the null struct is important. The first element in the struct (`_atomic`) is the *atomic null indicator*. This value indicates the null status for the object type as a whole. The atomic null indicator is followed by an indicator element corresponding to each element in the OTT-generated struct representing the object type.

Notice that when an object type contains another object type as part of its definition (in the above example it is the `object_type` attribute), the indicator entry for that attribute is the null indicator struct (`object_type_ind`) corresponding to the nested object type.

VARRAYs and NESTED TABLEs contain the null information for their elements.

The datatype for all other elements of a null indicator struct is *OCIInd*.

See Also: For more information about atomic nullness, refer to the section “Nullness” on page 8-28.

The Outtype File

The outtype file is named on the OTT command line. When the OTT generates the C header file, it also writes the results of the translation into the outtype file. This file contains an entry for each of the types which is translated, including its version string, and the header file to which its C representation was written.

The outtype file from one OTT run can be used as the intype file for a subsequent OTT invocation.

For example, given the 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
```

the user has chosen to specify the case for OTT-generated C identifiers, and has provided a list of types which should be translated. In two of these types, naming conventions are specified.

The following is an example of what the outtype file might look like after running the OTT:

```
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 which were not included in the intype specification. For example, if the intype file only specified that the `person` type was to be translated

```
CASE = LOWER
TYPE PERSON
```

and the definition of the `person` type includes an attribute of type `address`, then the outtype file will include entries for both `PERSON` and `ADDRESS`. The `person` type cannot be translated completely without first translating `address`.

As described in the section “The OTT Command Line” on page 12-6, the OTT analyzes the types in the intype file for type dependencies before performing the translation, and translates other types as necessary.

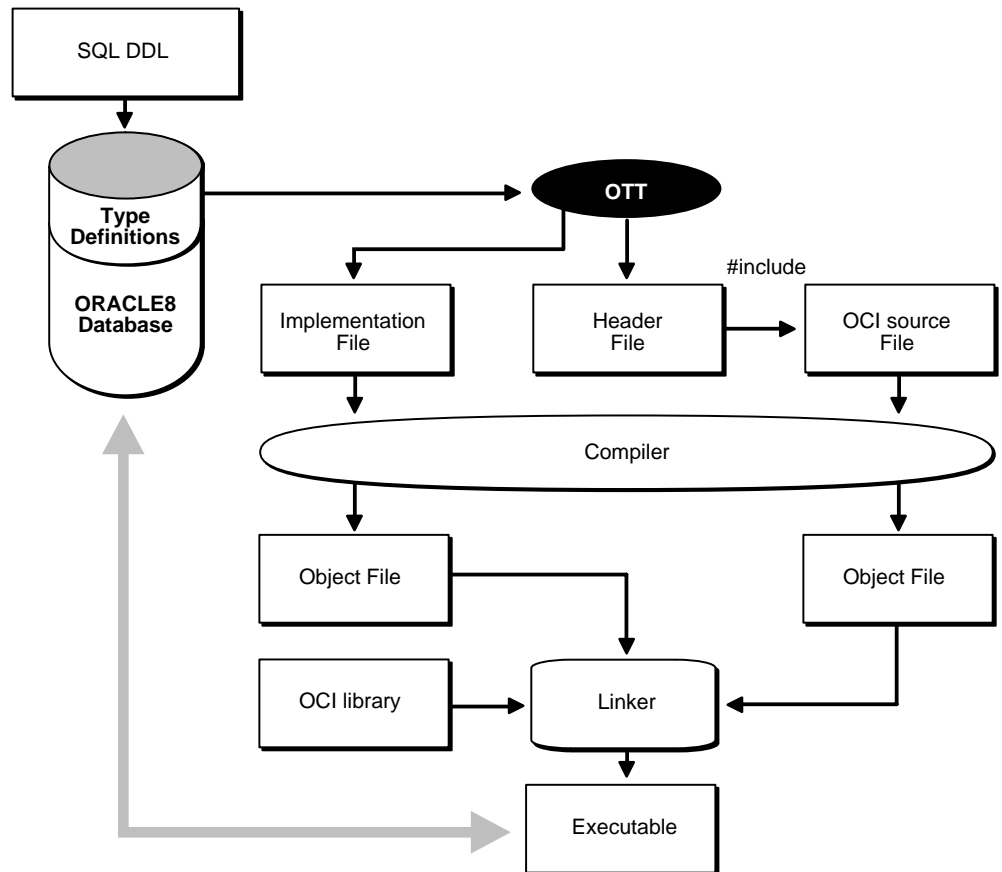
Using the OTT with OCI Applications

C header and implementation files which have been generated by the OTT can be used by an OCI application that accesses objects in an Oracle8 server. The header file is incorporated into the OCI code with an `#include` statement.

Once the header file has been included, the OCI application can access and manipulate object data in the host language format.

shows the steps involved in using the OTT with the OCI:

1. SQL is used to create type definitions in the database.
2. The OTT generates a header file containing C representations of object types and named collection types. It also generates an implementation file, as named with the `INITFILE` option.
3. The application is written. User-written code in the OCI application declares and calls the `INITFUNC` function.
4. The header file is included in an OCI source code file.
5. The OCI application, including the implementation file generated by the OTT, is compiled and linked with the OCI libraries.
6. The OCI executable is run against the Oracle8 server.

Figure 12–1 Using the OTT with OCI

Accessing and Manipulating Objects with OCI

Within the application, the OCI program can perform bind and define operations using program variables declared to be of types which appear in the OTT-generated header file.

For example, an application might fetch a REF to an object using a SQL SELECT statement and then pin that object using the appropriate OCI function. Once the object has been pinned, its attribute data can be accessed and manipulated with other OCI functions.

OCI includes a set of datatype mapping and manipulation functions which are specifically designed to work on attributes of object types and named collection types.

The following are examples of the available functions:

- *OCIStringSize()* gets the size of an **OCIString** string.
- *OCINumberAdd()* adds two **OCINumber** numbers together.
- *OCILobIsEqual()* compares two LOB locators for equality.
- *OCIRawPtr()* gets a pointer to an **OCIRaw** raw datatype.
- *OCICollAppend()* appends an element to a collection type (**OCIArray** or **OCITable**).
- *OCITableFirst()* returns the index for the first existing element of a nested table (**OCITable**).
- *OCIRefIsNull()* tests if a REF (**OCIRef**) is null

These functions are described in detail in other chapters of this guide.

Calling the Initialization Function

The OTT generates a C initialization function if requested. The initialization function tells the environment, for each object type used in the program, which version of the type is used. You may specify a name for the initialization function when invoking the OTT with the **INITFUNC** option, or may allow the OTT to select a default name based on the name of the implementation file (**INITFILE**) containing the function.

The initialization function takes two arguments, an environment handle pointer and an error handle pointer. There is typically a single initialization function, but this is not required. If a program has several separately compiled pieces requiring different types, you may want to execute the OTT separately for each piece requiring, for each piece, one initialization file, containing an initialization function.

After an environment handle is created by an explicit OCI object call, for example, by calling *OCIEnvInit()*, you must also explicitly call the initialization functions. All the initialization functions must be called for each explicitly created environment handle. This gives each handle access to all the Oracle8 datatypes used in the entire program.

If an environment handle is implicitly created via embedded SQL statements, such as **EXEC SQL CONTEXT USE** and **EXEC SQL CONNECT**, the handle is initialized

implicitly, and the initialization functions need not be called. This is only relevant when Pro*C/C++ is being combined with OCI applications.

The following example shows an initialization function.

Given an intype file, `ex2c.typ`, containing

```
TYPE BREN.PERSON
TYPE BREN.ADDRESS
```

and the command line

```
ott userid=bren/bigkitty intype=ex2c outtype=ex2co hfile=ex2ch.h
initfile=ex2cv.c
```

the OTT generates the following to the file `ex2cv.c`:

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword ex2cv(OCIEnv *env, OCIError *err)
{
    sword status = OCITypeVTInit(env, err);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
            "BREN", 5,
            "PERSON", 6,
            "$8.0", 4);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
            "BREN", 5,
            "ADDRESS", 7,
            "$8.0", 4);
    return status;
}
```

The function `ex2cv` creates the type version table and inserts the types `BREN.PERSON` and `BREN.ADDRESS`.

If a program explicitly creates an environment handle, all the initialization functions must be generated, compiled, and linked, because they must be called for each explicitly created handle. If a program does not explicitly create any environment handles, initialization functions are not required.

A program that uses an OTT-generated header file must also use the initialization function generated at the same time. More precisely, if a header file generated by

the OTT is included in a compilation that generates code that is linked into program P, and an environment handle is explicitly created somewhere in program P, the implementation file generated by the same invocation of the OTT must also be compiled and linked into program P. Doing this correctly is the user's responsibility.

Tasks of the Initialization Function

The C initialization function supplies version information about the types processed by the OTT. It adds to the type-version table the name and version identifier of every OTT-processed object datatype.

The type-version table is used by Oracle's type manager to determine which version of a type a particular program uses. Different initialization functions generated by the OTT at different times may add some of the same types to the type version table. When a type is added more than once, Oracle ensures the same version of the type is registered each time.

It is the OCI programmer's responsibility to declare a function prototype for the initialization function, and to call the function.

Note: In the current release of Oracle8, each type has only one version. Initialization of the type version table is required only for compatibility with future releases of Oracle8.

OTT Reference

Behavior of the OTT is controlled by parameters which can appear 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 Parameters
- Where OTT Parameters Can Appear
- Structure of the Intype File
- Nested #include File Generation
- SCHEMA_NAMES Usage
- Default Name Mapping
- Restrictions

The following conventions are used in this chapter to describe OTT syntax:

- Angle brackets (<...>) enclose strings to be supplied by the user.
- Strings in UPPERCASE are entered as shown, except that case is not significant.
- OTT keywords are listed in a lower-case monospaced font in examples and headings, but are printed in upper-case in text to make them more distinctive.
- Square brackets [...] enclose optional items.
- An ellipsis (...) immediately following an item (or items enclosed in brackets) means that the item can be repeated any number of times.
- Punctuation symbols other than those described above are entered as shown. These include '.', '@', etc.

OTT Command Line Syntax

The OTT command-line interface is used when explicitly invoking the OTT to translate database types into C structs. This is always required when developing OCI applications that use objects.

An OTT command line statement consists of the keyword OTT, followed by a list of OTT parameters.

The parameters which can appear on an OTT command line statement are as follows:

```
userid=<username>/<password>[@<db_name>]
[intype=<in_filename>]
outtype=<out_filename>
code=<C | ANSI_C | KR_C>
[hfile=<filename>]
[errtype=<filename>]
[config=<filename>]
[initfile=<filename>]
[initfunc=<filename>]
[case=<SAME | LOWER | UPPER | OPPOSITE>]
[schema_name=<ALWAYS | IF_NEEDED | FROM_INTYPE>]
```

Note: Generally, the order of the parameters following the OTT command does not matter, and only the OUTTYPE and CODE parameters are always required.

The HFILE parameter is almost always used. If omitted, HFILE must be specified individually for each type in the INTYPE file. If the OTT determines that a type not listed in the INTYPE file must be translated, 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, 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:

```
OTT userid=marc/cayman 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 sections.

OTT Parameters

Enter parameters on the OTT command line using the following format:

parameter=value

where *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 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, HFILE, INITFUNC, and INITFILE can appear in the INTYPE file.

userid

The USERID parameter specifies the Oracle username, password, and optional database name (Net8 database specification string). If the database name is omitted, the default database is assumed. The syntax of this parameter is:

```
userid=<username/password[@db_name]>
```

If this is the first parameter, "USERID=" may be omitted as shown here:

```
OTT username/password...
```

The USERID parameter is optional. If omitted, the OTT automatically attempts to connect to the default database as user OPSS`username`, where `username` is the user's operating system user name.

intype

The INTYPE parameter specifies the name of the file from which to read the list of object type specifications. The OTT translates each type in the list.

The syntax for this parameter is

`intype=<filename>`

"INTYPE=" may be omitted if USERID and INTYPE are the first two parameters, in that order, and "USERID=" is omitted. If INTYPE is not specified, all types in the user's schema will be translated.

OTT *username/password filename...*

The INTYPE file can be thought of as a makefile for type declarations. It lists the types for which C struct declarations are needed. The format of the INTYPE file is described in section "Structure of the Intype File" on page 12-29.

If the file name on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as "TYP" or ".typ" will be added.

outtype

The name of a file into which the OTT will write type information for all the object datatypes it processes. This 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.

`outtype=<filename>`

If the INTYPE and OUTTYPE parameters refer to the same file, 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.

OUTTYPE must be specified.

If the file name on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as "TYP" or ".typ" will be added.

code

This is the desired host language for OTT output, which may be specified as CODE=C, CODE=KR_C, or CODE=ANSI_C. "CODE=C" is equivalent to "CODE=ANSI_C".

```
CODE= C | KR_C | ANSI_C
```

There is no default value for this parameter; it must be supplied.

initfile

The INITFILE parameter specifies the name of the file where the OTT-generated initialization file is to be written. The initialization function will not be generated if this parameter is omitted.

For Pro*C/C++ programs, the INITFILE is not necessary, because the SQLLIB run-time library performs the necessary initializations. An OCI program user must compile and link the INITFILE file(s), and must call the initialization function(s) when an environment handle is created.

If the file name of an INITFILE on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as ".C" or ".c" will be added.

```
initfile=<filename>
```

initfunc

The INITFUNC parameter is only used in OCI programs. It specifies the name of the initialization function generated by the OTT. If this parameter is omitted, the name of the initialization function is derived from the name of the INITFILE.

```
initfunc=<filename>
```

hfile

The name of the include (.h) file to be generated by the OTT for the declarations of types that are mentioned in the INTYPE file but whose include files are not specified there. This parameter is required unless the include 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 file name of an HFILE on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as ".H" or ".h" will be added.

```
hfile=<filename>
```

config

The CONFIG parameter specifies the name of the OTT configuration file, which lists commonly used parameter specifications. Parameter specifications are also read from a system configuration file in a platform-dependent location. All remaining parameter specifications must appear on the command line, or in the INTYPE file.

`config=<filename>`

Note: A CONFIG parameter is not allowed in the CONFIG file.

errtype

If this parameter is supplied, a listing of the INTYPE file is written to the ERRTYPE file, along with all informational and error messages. Informational and error messages are sent to the standard output whether or not ERRTYPE 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 which caused the error.

If the file name of an ERRTYPE on the command line or in the INTYPE file does not include an extension, a platform-specific extension such as "TLS" or ".tls" will be added.

`errtype=<filename>`

case

This parameter affects the case of certain C identifiers generated by the OTT. The possible 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 identifiers. If CASE=LOWER, all uppercase letters are converted to lowercase. If CASE=UPPER, all lowercase letters are converted to uppercase. If CASE=OPPOSITE, all uppercase letters are converted to lower-case, and vice-versa.

`CASE=[SAME | LOWER | UPPER | OPPOSITE]`

This option 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.

Note: The case of the C struct identifier for a type specifically mentioned in the INTYPE is the same as its case in the INTYPE file. For example, if the INTYPE file includes the following line

```
TYPE Worker
```

then the OTT generates

```
struct Worker {...};
```

On the other hand, if the INTYPE file were written as

```
TYPE wOrKeR
```

the OTT generates

```
struct wOrKeR {...};
```

following the case of the INTYPE file.

Case-insensitive SQL identifiers not mentioned in the INTYPE file will appear in upper case if CASE=SAME, and in lower case if CASE=OPPOSITE. A SQL identifier is case-insensitive if it was not quoted when it was declared.

schema_names

This option 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 contains information about the types processed by the OTT, including the type names.

See “SCHEMA_NAMES Usage” on page 12-33 for further information.

Where OTT Parameters Can Appear

OTT parameters can appear 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 is invoked as follows:

```
OTT username/password <parameters>
```

If one of the parameters on the command line is

```
config=<filename>
```

additional parameters are read from the configuration file <filename>.

In addition, parameters are also read from a default configuration file in a platform-dependent location. This file must exist, but can be empty. Parameters in a configuration file must appear one per line, with no whitespace on the line.

If the OTT is executed without any arguments, an on-line parameter reference is displayed.

The types for the OTT to translate are named in the file specified by the INTYPE parameter. The parameters CASE, INITFILE, INITFUNC, and HFILE may also appear in the INTYPE file. OUTTYPE files generated by the OTT include the CASE parameter, and include the INITFILE, and INITFUNC parameters if an initialization file was generated. The OUTTYPE file specifies the HFILE individually for each type.

The case of the OTT command is platform-dependent.

Structure of the Intype File

The intype and outtype files list the types translated by the OTT, and provide all the information needed to determine how a type or attribute name is translated to a legal C identifier. These files contain one or more type specifications. These files also may contain specifications of the following options:

- CASE
- HFILE
- INITFILE
- INITFUNC

If the CASE, INITFILE, or INITFUNC options are present, they must precede any type specifications. If these options appear both on the command line and in the intype file, the value on the command line is used.

For an example of a simple user-defined intype file, and of the full outtype file that the OTT generates from it, see “The Outtype File” on page 12-16.

Intype File Type Specifications

A type specification in the INTYPE names an object datatype that is to be translated. A type specification in the OUTTYPE file names an object datatype that has been translated,

```
TYPE PERSON AS PERSON
  VERSION = "$8.0"
  HFILE = demo.h
```

The structure of a type specification is as follows:

```
TYPE <type_name> [AS <type_identifier>]
[VERSION [=] <version_string>]
[HFILE [=] <hfile_name>]
```

```
[TRANSLATE{<member_name> [AS <identifier>]}...]
```

The syntax of `type_name` is:
[<schema_name>.<type_name>]

where *schema_name* is the name of the schema which owns the given object datatype, and *type_name* is the name of the type. The default schema is that of the user running the OTT. The default database is the local database.

The components of a type specification are described below.

- `<type_name>` is the name of an Oracle8 object datatype.
- `<type_identifier>` is the C identifier used to represent the type. If omitted, the default name mapping algorithm will be used.
- `<version_string>` is the version string of the type which was used when the code was generated by a previous invocation of the OTT. The version string is generated by the OTT and written to the OUTTYPE file, which may later be used as the INTYPE file when the OTT is later executed. The version string does not affect the OTT's operation, but will eventually be used to select which version of the object datatype should be used in the running program.
- `<type_identifier>` is the C identifier used to represent the type. If omitted, the default type mapping algorithm will be used. For further information, see "Default Name Mapping" on page 12-35.
- `<member_name>` is the name of an attribute (data member) which is to be translated to the following `<identifier>`.
- `<identifier>` is the C identifier used to represent the attribute in the user program. Identifiers may be specified in this way for any number of attributes. The default name mapping algorithm will be used for the attributes that are not mentioned.
- `<hfile_name>` is the name of the header file in which the declarations of the corresponding struct or class appears or will appear. If `<hfile name>` is omitted, the file named by the command-line HFILE parameter will be used if a declaration is generated.

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.

If a type that is not mentioned explicitly is required by types declared in exactly one file, the translation of the required type is written to the same file(s) 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, the translation of the required type is written to the global HFILE file.

Nested #include File Generation

Every HFILE generated by the OTT #includes other necessary files, and #defines a symbol constructed from the name of the file, which may be used to determine if the HFILE 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);
```

where the intype file contains:

```
CASE=lower
type px1
  hfile tott95a.h
type px3
  hfile tott95b.h
```

If we invoke the OTT with

```
ott scott/tiger tott95i.typ outtype=tott95o.typ code=c
```

then it will generate the two following header files.

File tott95b.h is:

```
#ifndef TOT95B_ORACLE
#define TOT95B_ORACLE
#endif
#include <oci.h>
typedef OCIRef px3_ref;
struct px3
{
```

```
    struct px1 coll;  
};  
typedef struct px3 px3;  
struct px3_ind  
{  
    OCInd _atomic;  
    struct px1_ind coll;  
};  
typedef struct px3_ind px3_ind;  
#endif
```

File tott95a.h is:

```
#ifndef TOT95A_ORACLE  
#define TOT95A_ORACLE  
#ifndef OCI_ORACLE  
#include <oci.h>  
#endif  
typedef OCIRef px1_ref;  
struct px1  
{  
    OCINumber coll;  
    OCINumber col2;  
}  
typedef struct px1 px1;  
struct px1_ind  
{  
    OCInd _atomic;  
    OCInd coll;  
    OCInd col2;  
}  
typedef struct px1_ind px1_ind;  
#endif
```

In this file, the symbol TOT95B_ORACLE is defined first so that the programmer may conditionally include *tott95b.h* without having to worry whether *tott95b.h* depends on the include file using the following construct:

```
#ifndef TOT95B_ORACLE  
#include "tott95b.h"  
#endif
```

Using this technique, the programmer may include *tott95b.h* from some file, say *foo.h*, without having to know whether some other file included by *foo.h* also includes *tott95b.h*.

After the definition of the symbol TOTT95B_ORACLE, the file *oci.h* is `#included`. Every HFILE generated by the OTT 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 uses angle brackets in a `#include`.

Next, the file *tott95a.h* is included. This file is included because it contains the declaration of "struct px1", which *tott95b.h* requires. When the user's INTYPE file requests that type declarations be written to more than one file, the OTT determines which other files each HFILE must include, and will generate the necessary `#includes`.

Note that the OTT uses quotes in this `#include`. When a program including *tott95b.h* is compiled, the search for *tott95a.h* will begin where the source program was found, and will thereafter follow an implementation-defined search rule. If *tott95a.h* cannot be found in this way, a complete file name (e.g., a UNIX absolute pathname beginning with `/`) 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 is connected is qualified with a schema name in the OUTTYPE file.

The name of a type from a schema other than 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 settings:

- `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 has to be generated because of type dependencies will be written

with a schema name only if the first type encountered by the OTT that depends on it was written with a schema name. However, a type that is not in the default schema to which the OTT is connected will always be written with an explicit schema name.

The OUTTYPE file generated by the OTT is an input parameter to Pro*C/C++. From the point of view of Pro*C/C++, it is the Pro*C/C++ INTYPE file. This file matches database type names to C struct names. This information is used at run-time to make sure that the correct database type is selected into the struct. If a type appears with a schema name in the OUTTYPE file (Pro*C/C++ INTYPE file), the type will be found in the named schema during program execution. If the type appears without a schema name, the type will be found in the default schema to which the program connects, which may be different from the default schema the OTT used.

An Example If SCHEMA_NAMES is set to FROM_INTYPE, and the INTYPE file reads:

```
TYPE Person
TYPE david.Dept
TYPE eric.Company
```

then the Pro*C/C++ application that uses the OTT-generated structs will use the types sam.Company, david.Dept, and Person. Using Person without a schema name refers to the Person type in the schema to which the application is connected.

If the OTT and the application both connect to schema david, the application will use the same type (david.Person) that the OTT used. If the OTT connected to schema david but the application connects to schema jana, the application will use the type jana.Person. This behavior is appropriate only if the same "CREATE TYPE Person" statement has been executed in schema david and schema jana.

On the other hand, the application will use type david.Dept regardless of to which schema the application is connected. If this is the behavior you want, be sure to include schema names with your type names in the INTYPE file.

In some cases, the OTT 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 );

```

Now suppose that the OTT connects to schema david, SCHEMA_NAMES=FROM_INTYPE is specified, and the user's INTYPE files include either

```
TYPE Person
```

or

```
TYPE david.Person
```

but do not mention the type david.Address, which is used as a nested object type in type david.Person. If "TYPE david.Person" appeared in the INTYPE file, "TYPE david.Person" and "TYPE david.Address" will appear in the OUTTYPE file. If "Type Person" appeared in the INTYPE file, "TYPE Person" and "TYPE Address" will appear in the OUTTYPE file.

If the david.Address type is embedded in several types translated by the OTT, but is not explicitly mentioned in the INTYPE file, the decision of whether to use a schema name is made the first time the OTT encounters the embedded david.Address type. If, for some reason, the user wants type david.Address to have a schema name but does not want type Person to have one, the user should explicitly request

```
TYPE      david.Address
```

in the INTYPE FILE.

The main point is that in the usual case in which each type is declared in a single schema, it is safest for the user to qualify all type names with schema names in the INTYPE file.

Default Name Mapping

When the OTT creates a C identifier name for an object type or attribute, it translates the name from the database character set to a legal C identifier. First, the name is translated from the database character set to the character set used by the OTT. Next, if a translation of the resulting name is supplied in the INTYPE file, that translation is used. Otherwise, the OTT translates the name character-by-character

to the compiler character set, applying the CASE option. This process is described in more detail below:

When the OTT 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. In order for the OTT 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 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, the OTT character set must include 7-bit ASCII as a subset, and if the compiler character set is 7-bit EBCDIC, the OTT character set must include 7-bit EBCDIC as a subset. The user specifies the character set that the OTT uses by setting the `NLS_LANG` environment variable, or by some other platform-specific mechanism.

Once the OTT has read the name of a database entity, it translates the name from the character set used by the OTT to the compiler's character set. If a translation of the name appears in the `INTYPE` file, the OTT uses that translation.

Otherwise, the OTT attempts to translate the name as follows:

1. First, if the OTT character set is a multi-byte character set, all multi-byte characters in the name that have single-byte equivalents are converted to those single-byte equivalents.
2. Next, 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. Finally, the case of letters is set according to the CASE option in effect, and any character that is not legal in a C identifier, or that has no translation in the compiler character set, is replaced by an underscore. If at least one character is replaced by an underscore, the OTT gives a warning message. If all the characters in a name are replaced by underscores, the OTT 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 identifiers are not altered.

Name translation may, for example, translate accented single-byte characters such as “ö” with an umlaut or “à” with an accent grave to “o” or “a”, and may translate a multi-byte letter to its single-byte equivalent. Name translation will typically fail

if the name contains multi-byte characters that lack single-byte equivalents. In this case, the user must specify name translations in the INTYPE file.

The OTT 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.

Restrictions

The following restrictions exist which affect use of the OTT.

File Name Comparison

Currently, the OTT determines if two files are the same by comparing the file names provided by the user on the command line or in the INTYPE file. But one potential problem can occur when the OTT needs to know if two file names 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/elias/foo1.h`, the OTT should generate one `#include` if the two files are the same, and two `#includes` if the files are different. In practice, though, it would conclude that the two files are different, and would generate two `#includes`, as follows:

```
#ifndef FOOL_ORACLE
#include "foo1.h"
#endif
#ifndef FOOL_ORACLE
#include "/private/elias/foo1.h"
#endif
```

If `foo1.h` and `/private/elias/foo1.h` are different files, only the first one will be included. If `foo1.h` and `/private/elias/foo1.h` are the same file, a redundant `#include` will be written.

Therefore, if a file is mentioned several times on the command line or in the INTYPE file, each mention of the file should use exactly the same file name.

Part III

OCI Reference

This part of the book contains the OCI function reference chapters:

- Chapter 13, “OCI Relational Functions”
- Chapter 14, “OCI Navigation and Type Functions”
- Chapter 15, “OCI Datatype Mapping and Manipulation Functions”
- Chapter 16, “OCI External Procedure Functions”

OCI Relational Functions

This chapter describes the Oracle8 OCI relational functions for C. It includes information about calling OCI functions in your application, along with detailed descriptions of each function call.

This chapter contains the following sections:

- Introduction
- OCI Quick Reference
- The OCI Relational Functions
- Calling OCI Functions

Introduction

This chapter describes the OCI relational function calls. This chapter covers those functions in the basic OCI. The function calls for manipulating objects are described in the next three chapters.

For information about return codes and error handling, refer to the section “Error Handling” on page 2-25.

OCI Quick Reference

This table directs you to the location of a given OCI call in this chapter. The following list includes all OCI relational and type information accessor functions, grouped by functional category.

Table 13–1 OCI Quick Reference

Function	Purpose	Page
	CONNECT / INITIALIZE / AUTHORIZE	
OCIInitialize()	Initialize OCI process environment	13 - 72
OCIEnvInit()	Initialize an environment handle	13 - 63
OCIServerAttach()	Attach to a server; initialize server context handle	13 - 125
OCIServerDetach()	Detach from a server; uninitialize server context handle	13 - 127
OCISessionBegin()	Authenticate a user	13 - 129
OCISessionEnd()	Terminate a user session	13 - 132
OCILogon()	Simplified single-session logon	13 - 117
OCILogoff()	Simplified single-session logoff	13 - 116
	HANDLES / DESCRIPTORS	
OCIAttrGet()	Get the attributes of a handle	13 - 11
OCIAttrSet()	Set an attribute of a handle or descriptor	13 - 25
OCIDescriptorAlloc()	Allocate and initialize a descriptor or LOB locator	13 - 60
OCIDescriptorFree()	Free a previously allocated descriptor	13 - 62
OCIHandleAlloc()	Allocate and initialize a handle	13 - 68
OCIHandleFree()	Free a previously allocated handle	13 - 70
OCIParamGet()	Get a parameter descriptor	13 - 119
OCIParamSet()	Set parameter descriptor in COR handle	13 - 121
	TRANSACTION MANAGEMENT	
OCITransCommit()	Commit a transaction on a service context	13 - 149
OCITransDetach()	Detach a transaction from a service context	13 - 152
OCITransRollback()	Roll back a transaction	13 - 156
OCITransStart()	Start a transaction on a service context	13 - 157
OCITransPrepare()	Prepare a global transaction for commit	13 - 155

Table 13–1 OCI Quick Reference (Cont.)

Function	Purpose	Page
OCITransForget()	Forget a prepared global transaction	13 - 154
BIND		
OCIBindDynamic()	Set additional attributes after bind with OCI_DATA_AT_EXEC mode	13 - 38
OCIBindByName	Bind by name	13 - 30
OCIBindByPos()	Bind by position	13 - 11
OCIBindObject()	Set additional attributes for bind of named data type	13 - 42
OCIBindArrayOfStruct()	Set skip parameters for static array bind	13 - 28
OCISstmtGetBindInfo()	Get bind and indicator variable names and handles	13 - 139
DEFINE		
OCIDefineArrayOfStruct()	Set additional attributes for static array define	13 - 46
OCIDefineDynamic()	Set additional attributes for define in OCI_DYNAMIC_FETCH mode	13 - 52
OCIDefineByPos()	Define an output variable association	13 - 48
OCIDefineObject()	Set additional attributes for define of named data type	13 - 55
DESCRIBE		
OCIDescribeAny()	Describe existing schema objects	13 - 57
PREPARE/EXECUTE/FETCH		
OCISstmtPrepare()	Establish an application request	13 - 143
OCISstmtExecute()	Send statements to server for execution	13 - 134
OCISstmtFetch()	Fetch rows from a query	13 - 137
LOB/FILE OPERATIONS		
OCILobFileClose()	Close a previously opened FILE	13 - 88
OCILobFileCloseAll()	Close all previously opened files	13 - 89
OCILobFileOpen()	Open a FILE	13 - 95
OCILobAppend()	Append to a LOB	13 - 76
OCILobCopy()	Copy a LOB	13 - 82
OCILobErase()	Erase a portion of a LOB	13 - 86
OCILobGetLength()	Get length of a LOB or FILE	13 - 100
OCILobRead()	Read a portion of a LOB or FILE	13 - 107

Table 13–1 OCI Quick Reference (Cont.)

Function	Purpose	Page
OCILobTrim()	Truncate a LOB	13 - 111
OCILobWrite()	Write into a LOB	13 - 112
OCILobAssign()	Assign one LOB locator to another	13 - 78
OCILobIsEqual()	Compare two LOB locators for Equality	13 - 102
OCILobFileGetName()	Get directory alias and file NaMe from the LOB locator	13 - 91
OCILobFileIsOpen()	Check if file on server is open via this locator	13 - 93
OCILobFileSetName()	Set directory alias and file name in the LOB locator	13 - 96
OCILobLocatorIsInit()	Check to see if a LOB locator is initialized	13 - 105
OCILobCharSetID	Get character set ID from LOB locator	13 - 81
OCILobCharSetForm()	Get character set form from LOB locator	13 - 80
OCILobFileExists()	Check if a file exists on the server	13 - 90
OCILobLoadFromFile()	Load a LOB from a FILE	13 - 103
OCILobDisableBuffering()	Turn LOB buffering off	13 - 84
OCILobEnableBuffering()	Turn LOB buffering on	13 - 85
OCILobFlushBuffer()	Flush the LOB buffer	13 - 98
MISCELLANEOUS		
OCIBreak()	Perform an immediate asynchronous break	13 - 45
OCIServerVersion()	Get the Oracle version string	13 - 128
OCIPasswordChange()	Change password	13 - 123
OCIErrorGet()	Return error message and Oracle error	13 - 65
OCISstmtGetPieceInfo()	Get piece information for piecewise operations	13 - 141
OCISstmtSetPieceInfo()	Set piece information for piecewise operations	13 - 145
OCILdaToSvcCtx()	Toggle Lda_Def to service context handle	13 - 75
OCISvcCtxToLda()	Toggle service context handle to Lda_Def	13 - 147
OCIAQEnq()	Advanced queueing enqueue	13 - 11
OCIAQDeq()	Advanced queueing dequeue	13 - 8

Calling OCI Functions

Unlike earlier versions of the OCI, in release 8.0 you cannot pass -1 for the string length parameter of a null-terminated string.

When you pass string lengths as parameters, do not include the NULL terminator byte in the length. The OCI does not expect strings to be NULL-terminated.

Server Roundtrips for LOB Functions

For a table showing the number of server roundtrips required for individual OCI LOB functions, refer to Appendix E, “OCI Function Server Roundtrips”.

The OCI Relational Functions

The remainder of this chapter specifies the release 8.0 OCI relational functions for C. For each function, the following information is listed:

Purpose

A brief description of the action performed by the function.

Syntax

A code snippet showing the syntax for calling the function, including the ordering and types of the parameters.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below.

Mode	Description
IN	A parameter that passes data to the OCI
OUT	A parameter that receives data from the OCI on this call
IN/OUT	A parameter that passes data on the call and receives data on the return from this or a subsequent call.

Comments

More detailed information about the function (if available). This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Example

A complete or partial code example demonstrating the use of the function call being described. Not all function descriptions include an example.

Related Functions

A list of related function calls.

OCIAQDeq()

Purpose

This call is used for an advanced queueing dequeue.

Syntax

```
sword OCIAQDeq ( OCISvcCtx          *svch,
                  OCIError           *errh,
                  text                *queue_name,
                  OCIAQDeqOptions    *dequeue_options,
                  OCIAQMsgProperties *message_properties,
                  OCIType             *payload_tdo,
                  dvoid               **payload,
                  dvoid               **payload_ind,
                  OCIRaw              **msgid,
                  ub4                 flags );
```

Parameters

svch (IN)

OCI service context.

errh (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

queue_name (IN)

The target queue for the dequeue operation.

dequeue_options (IN)

The options for the dequeue operation; stored in an **OCIAQDeqOptions** descriptor.

message_properties (OUT)

The message properties for the message; stored in an **OCIAQMsgProperties** descriptor.

payload_tdo (IN)

The TDO (type descriptor object) of an object type. For a raw queue, this parameter should point to the TDO of SYS.RAW.

payload (IN/OUT)

A pointer to a pointer to a program variable buffer that is an instance of an object type. For a raw queue, this parameter should point to an instance of **OCIRaw**.

Memory for the payload is dynamically allocated in the object cache. The application can optionally call *OCIObjectFree()* to deallocate the payload instance when it is no longer needed. If the pointer to the program variable buffer (**payload*) is passed as NULL, the buffer is implicitly allocated in the cache.

The application may choose to pass NULL for *payload* the first time *OCIAQDeq()* is called, and let the OCI allocate the memory for the payload. It can then use a pointer to that previously allocated memory in subsequent calls to *OCIAQDeq()*.

The OCI provides functions which allow the user to set attributes of the payload, such as its text. For information about setting these attributes, refer to “Manipulating Object Attributes” on page 8-13.

payload_ind (IN/OUT)

A pointer to a pointer to the program variable buffer containing the parallel indicator structure for the object type.

The memory allocation rules for *payload_ind* are the same as those for *payload*, above.

msgid (OUT)

The message ID.

flags (IN)

Not currently used; pass as OCI_DEFAULT.

Comments

This function is used to perform an Advanced Queueing dequeue operation using the OCI.

Users must have the *aq_user_role* or privileges to execute the *dbms_aq* package in order to use this call.

The OCI environment must be initialized in object mode (using *OCIInitialize()*) to use this call.

For more information about OCI and Advanced Queueing, refer to “OCI and Advanced Queueing” on page 7-40.

For additional information about Advanced Queueing, refer to *Oracle8 Application Developer's Guide*.

To obtain a TDO for the payload, use *OCTypeByName()*, or *OCTypeByRef()*.

Examples

For examples demonstrating the use of *OCIAQDeq()*, refer to the description of *OCIAQEnq()* on page 13-11.

Related Functions

OCIAQEnq(), *OCIInitialize()*

OCIAQEnq()

Purpose

This call is used for an advanced queueing enqueue.

Syntax

```

sword OCIAQEnq ( OCISvcCtx          *svch,
                  OCIError           *errh,
                  text               *queue_name,
                  OCIAQEnqOptions    *enqueue_options,
                  OCIAQMsgProperties *message_properties,
                  OCIType            *payload_tdo,
                  dvoid              **payload,
                  dvoid              **payload_ind,
                  OCIRaw             **msgid,
                  ub4                flags );

```

Parameters

svch (IN)

OCI service context.

errh (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

queue_name (IN)

The target queue for the enqueue operation.

enqueue_options (IN)

The options for the enqueue operation; stored in an **OCIAQEnqOptions** descriptor.

message_properties (IN)

The message properties for the message; stored in an **OCIAQMsgProperties** descriptor.

payload_tdo (IN)

The TDO (type descriptor object) of an object type. For a raw queue, this parameter should point to the TDO of SYS.RAW.

payload (IN)

A pointer to a pointer to an instance of an object type. For a raw queue, this parameter should point to an instance of **OCIRaw**.

The OCI provides functions which allow the user to set attributes of the payload, such as its text. For information about setting these attributes, refer to “Manipulating Object Attributes” on page 8-13.

payload_ind (IN)

A pointer to a pointer to the program variable buffer containing the parallel indicator structure for the object type.

msgid (OUT)

The message ID.

flags (IN)

Not currently used; pass as OCI_DEFAULT.

Comments

This function is used to perform an Advanced Queueing enqueue operation using the OCI.

Users must have the aq_user_role or privileges to execute the dbms_aq package in order to use this call.

The OCI environment must be initialized in object mode (using *OCIInitialize()*) to use this call.

For more information about OCI and Advanced Queueing, refer to “OCI and Advanced Queueing” on page 7-40.

For additional information about Advanced Queueing, refer to *Oracle8 Application Developer's Guide*.

To obtain a TDO for the payload, use *OCITypeByName()*, or *OCITypeByRef()*.

Examples

The following four examples demonstrate the use of *OCIAQEnq()* and *OCIAQDeq()* in several different situations.

These examples assume that the database is set up as illustrated in the section “Oracle Advanced Queueing By Example” in the advanced queueing chapter of the *Oracle8 Application Developer's Guide*.

Example 1

Enqueue and dequeue of a payload object.

```

struct message
{
    OCIStrng    *subject;
    OCIStrng    *data;
};
typedef struct message message;

struct null_message
{
    OCIIInd     null_adt;
    OCIIInd     null_subject;
    OCIIInd     null_data;
};
typedef struct null_message null_message;

int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    dvoid *tmp;
    OCIType *mesg_tdo = (OCIType *) 0;
    message msg;
    null_message nmsg;
    message *mesg = &msg;
    null_message *nmesg = &nmsg;
    message *deqmesg = (message *)0;
    null_message *ndeqmesg = (null_message *)0;

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    52, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                    52, (dvoid **) &tmp);

```

```

OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
            (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

/* obtain TDO of message_type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
              (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
              (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
mesg->subject = (OCIStrng *)0;
mesg->data = (OCIStrng *)0;
OCIStrngAssignText(envhp, errhp, (CONST text *)"NORMAL MESSAGE",
                  strlen("NORMAL MESSAGE"), &mesg->subject);
OCIStrngAssignText(envhp, errhp, (CONST text *)"OCI ENQUEUE",
                  strlen("OCI ENQUEUE"), &mesg->data);
nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

/* enqueue into the msg_queue */
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
         mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* dequeue from the msg_queue */
OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
         mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
printf("Subject: %s\n", OCIStrngPtr(envhp, deqmesg->subject));
printf("Text: %s\n", OCIStrngPtr(envhp, deqmesg->data));
OCITransCommit(svchp, errhp, (ub4) 0);
}

```

Example 2

Enqueue and dequeue using correlation IDs.

```

struct message
{
    OCIStrng    *subject;
    OCIStrng    *data;
};

```

```

typedef struct message message;

struct null_message
{
    OCInd    null_adt;
    OCInd    null_subject;
    OCInd    null_data;
};
typedef struct null_message null_message;

int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    dvoid *tmp;
    OCIType *mesg_tdo = (OCIType *) 0;
    message msg;
    null_message nmsg;
    message *mesg = &msg;
    null_message *nmesg = &nmsg;
    message *degmesg = (message *)0;
    null_message *ndegmesg = (null_message *)0;
    OCIRaw*firstmsg = (OCIRaw *)0;
    OCIAQMsgProperties *msgprop = (OCIAQMsgProperties *)0;
    OCIAQDeqOptions *degopt = (OCIAQDeqOptions *)0;
    text correlation1[30], correlation2[30];

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );
    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    52, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                    52, (dvoid **) &tmp);

    OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                    52, (dvoid **) &tmp);

```

```

OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
            (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

/* allocate message properties descriptor */
OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
                   OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)0);
strcpy(correlation1, "1st message");
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)&correlation1,
           strlen(correlation1), OCI_ATTR_CORRELATION, errhp);

/* obtain TDO of message_type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
              (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
              (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
mesg->subject = (OCIStrng *)0;
mesg->data = (OCIStrng *)0;
OCIStrngAssignText(envhp, errhp, (CONST text *)"NORMAL ENQUEUE1",
                  strlen("NORMAL ENQUEUE1"), &mesg->subject);
OCIStrngAssignText(envhp, errhp, (CONST text *)"OCI ENQUEUE",
                  strlen("OCI ENQUEUE"), &mesg->data);
nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

/* enqueue into the msg_queue, store the message id into firstmsg */
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, msgprop,
         mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, &firstmsg, 0);

/* enqueue into the msg_queue with a different correlation id */
strcpy(correlation2, "2nd message");
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)&correlation2,
           strlen(correlation2), OCI_ATTR_CORRELATION, errhp);
OCIStrngAssignText(envhp, errhp, (CONST text *)"NORMAL ENQUEUE2",
                  strlen("NORMAL ENQUEUE2"), &mesg->subject);
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, msgprop,
         mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

OCITransCommit(svchp, errhp, (ub4) 0);

/* first dequeue by correlation id "2nd message" */
/* allocate dequeue options descriptor and set the correlation option */
OCIDescriptorAlloc(envhp, (dvoid **)&deqopt,
                   OCI_DTYPE_AQDEQ_OPTIONS, 0, (dvoid **)0);

```

```

OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)correlation2,
           strlen(correlation2), OCI_ATTR_CORRELATION, errhp);

/* dequeue from the msg_queue */
OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", deqopt, 0,
         msg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
OCITransCommit(svchp, errhp, (ub4) 0);

/* second dequeue by message id */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&firstmsg,
OCIRawSize(envhp, firstmsg), OCI_ATTR_DEQ_MSGID, errhp);
/* clear correlation id option */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
           (dvoid *)correlation2, 0, OCI_ATTR_CORRELATION, errhp);

/* dequeue from the msg_queue */
OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", deqopt, 0,
         msg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
OCITransCommit(svchp, errhp, (ub4) 0);
}

```

Example 3

Enqueue and dequeue of a raw queue.

```

int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    dvoid *tmp;
    OCIType *msg_tdo = (OCIType *) 0;
    char msg_text[100];
    OCIRaw *mesg = (OCIRaw *)0;
    OCIRaw*deqmesg = (OCIRaw *)0;
    OCIInd ind = 0;
    dvoid *indptr = (dvoid *)&ind;
    inti;

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                 (dvoid * (*)()) 0, (void (*)()) 0);

```

```

OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                52, (dvoid **) &tmp);

OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                52, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                52, (dvoid **) &tmp);

OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
            (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

/* obtain the TDO of the RAW data type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"SYS", strlen("SYS"),
              (CONST text *)"RAW", strlen("RAW"),
              (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
strcpy(msg_text, "Enqueue to a RAW queue");
OCIRawAssignBytes(envhp, errhp, msg_text, strlen(msg_text), &mesg);

/* enqueue the message into raw_msg_queue */
OCIAQEnq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
         mesg_tdo, (dvoid **)&mesg, (dvoid **)&indp, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* dequeue the same message into C variable deqmesg */
OCIAQDeq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
         mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&indp, 0, 0);
for (i = 0; i < OCIRawSize(envhp, deqmesg); i++)
    printf("%c", *(OCIRawPtr(envhp, deqmesg) + i));
OCITransCommit(svchp, errhp, (ub4) 0);
}

```

Example 4

Enqueue and dequeue using OCIAQAgent.

```

struct message
{
    OCIStrng    *subject;
    OCIStrng    *data;
};
typedef struct message message;

struct null_message
{
    OCIIInd     null_adt;
    OCIIInd     null_subject;
    OCIIInd     null_data;
};
typedef struct null_message null_message;

int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIErrr *errhp;
    OCISvcCtx *svchp;
    dvoid *tmp;
    OCITYPE *mesg_tdo = (OCITYPE *) 0;
    message msg;
    null_message rmsg;
    message *mesg = &msg;
    null_message *rmesg = &rmsg;
    message *deqmesg = (message *)0;
    null_message *ndeqmesg = (null_message *)0;
    OCIAQMsgProperties *msgprop = (OCIAQMsgProperties *)0;
    OCIAQAgent *agents[2];
    OCIAQDeqOptions *deqopt = (OCIAQDeqOptions *)0;
    ub4wait = OCI_DEQ_NO_WAIT;
    ub4 navigation = OCI_DEQ_FIRST_MSG;

    OCIIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                   52, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

```

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                52, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                52, (dvoid **) &tmp);

OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);

OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
            (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

/* obtain TDO of message_type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
              (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
              (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
mesg->subject = (OCIStrng *)0;
mesg->data = (OCIStrng *)0;
OCIStrngAssignText(envhp, errhp,
                  (CONST text *)"MESSAGE 1", strlen("MESSAGE 1"),
                  &mesg->subject);
OCIStrngAssignText(envhp, errhp,
                  (CONST text *)"mesg for queue subscribers",
                  strlen("mesg for queue subscribers"), &mesg->data);
nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

/* enqueue MESSAGE 1 for subscribers to the queue i.e. for RED and GREEN */
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, 0,
          mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

/* enqueue MESSAGE 2 for specified recipients i.e. for RED and BLUE */
/* prepare message payload */
OCIStrngAssignText(envhp, errhp,
                  (CONST text *)"MESSAGE 2", strlen("MESSAGE 2"),
                  &mesg->subject);
OCIStrngAssignText(envhp, errhp,
                  (CONST text *)"mesg for two recipients",
                  strlen("mesg for two recipients"), &mesg->data);
```



```

/* allocate AQ message properties and agent descriptors */
OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
    OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)0);
OCIDescriptorAlloc(envhp, (dvoid **)&agents[0],
    OCI_DTYPE_AQAGENT, 0, (dvoid **)0);
OCIDescriptorAlloc(envhp, (dvoid **)&agents[1],
    OCI_DTYPE_AQAGENT, 0, (dvoid **)0);

/* prepare the recipient list, RED and BLUE */
OCIAttrSet(agents[0], OCI_DTYPE_AQAGENT, "RED", strlen("RED"),
    OCI_ATTR_AGENT_NAME, errhp);
OCIAttrSet(agents[1], OCI_DTYPE_AQAGENT, "BLUE", strlen("BLUE"),
    OCI_ATTR_AGENT_NAME, errhp);
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)agents, 2,
    OCI_ATTR_RECIPIENT_LIST, errhp);

OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, msgprop,
    mesg_tdo, (dvoid **)&mesg, (dvoid **)&mresg, 0, 0);

OCITransCommit(svchp, errhp, (ub4) 0);

/* now dequeue the messages using different consumer names */
/* allocate dequeue options descriptor to set the dequeue options */
OCIDescriptorAlloc(envhp, (dvoid **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
    (dvoid **)0);

/* set wait parameter to NO_WAIT so that the dequeue returns immediately */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&wait, 0,
    OCI_ATTR_WAIT, errhp);

/* set navigation to FIRST_MESSAGE so that the dequeue resets the position */
/* after a new consumer_name is set in the dequeue options */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&navigation, 0,
    OCI_ATTR_NAVIGATION, errhp);

/* dequeue from the msg_queue_multiple as consumer BLUE */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"BLUE", strlen("BLUE"),
    OCI_ATTR_CONSUMER_NAME, errhp);
while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
    mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
    == OCI_SUCCESS)
{
    printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
    printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
}

```

```

OCITransCommit(svchp, errhp, (ub4) 0);

/* dequeue from the msg_queue_multiple as consumer RED */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"RED", strlen("RED"),
           OCI_ATTR_CONSUMER_NAME, errhp);
while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
               msg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
{
    printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
    printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
}
OCITransCommit(svchp, errhp, (ub4) 0);

/* dequeue from the msg_queue_multiple as consumer GREEN */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"GREEN", strlen("GREEN"),
           OCI_ATTR_CONSUMER_NAME, errhp);
while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
               msg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
{
    printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
    printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
}
OCITransCommit(svchp, errhp, (ub4) 0);
}

```

Related Functions

OCIAQDeq(), OCIInitialize()

OCIAttrGet()

Purpose

This call is used to get a particular attribute of a handle.

Syntax

```
sword OCIAttrGet ( CONST dvoid      *trgthndlp,  
                  ub4              trghndltyp,  
                  dvoid            *attributep,  
                  ub4              *sizep,  
                  ub4              attrtype,  
                  OCIError         *errhp );
```

Parameters

trgthndlp (IN)

Pointer to a handle type.

trghndltyp (IN)

The handle type.

attributep (OUT)

Pointer to the storage for an attribute value. The attribute value is filled in.

sizep (OUT)

The size of storage for the attribute value. This can be passed in as NULL for parameters whose size is well known. For **text*** parameters, a pointer to a **ub4** must be passed in to get the length of the string.

attrtype (IN)

The type of attribute being retrieved.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

Comments

This call is used to get a particular attribute of a handle.

See Appendix B, “Handle and Descriptor Attributes”, for a list of handle types and their readable attributes.

Related Functions

OCIAttrSet()

OCIAttrSet()

Purpose

This call is used to set a particular attribute of a handle or a descriptor.

Syntax

```
sword OCIAttrSet ( dvoid      *trgthndlp,  
                  ub4        trghndltyp,  
                  dvoid      *attributep,  
                  ub4        size,  
                  ub4        attrtype,  
                  OCIError    *errhp );
```

Parameters

trgthndlp (IN/OUT)

Pointer to a handle type whose attribute gets modified.

trghndltyp (IN/OUT)

The handle type.

attributep (IN)

Pointer to an attribute value. The attribute value is copied into the target handle. If the attribute value is a pointer, then only the pointer is copied, not the contents of the pointer.

size (IN)

The size of an attribute value. This can be passed in as 0 for most attributes as the size is already known by the OCI library. For **text*** attributes, a **ub4** must be passed in set to the length of the string.

attrtype (IN)

The type of attribute being set.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

Comments

This call is used to set a particular attribute of a handle or a descriptor.

See Appendix B, “Handle and Descriptor Attributes”, for a list of handle types and their writable attributes.

Example

The following code sample demonstrates *OCIAttrSet()* being used several times near the beginning of an application.

```
int main()
{
    OCIEnv *envhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCISmt *stmthp;
    OCISession *usrhp;

    OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT, (dvoid *)0,
        (dvoid * (*)()) 0, (dvoid * (*)()) 0, (void (*)()) 0 );
    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
        0, (dvoid **) &tmp);
    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 0, (dvoid **) &tmp );
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4)
        OCI_HTYPE_ERROR, 0, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4)
        OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
    OCIErrorAttach( svchp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp,
        (ub4) OCI_HTYPE_SVCCTX, , (dvoid **) &tmp);

    /* set attribute server context in the service context */
    OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) svchp,
        (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

    /* allocate a user session handle */
    OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp,
        (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);
    OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"sherry",
        (ub4)strlen("sherry"), OCI_ATTR_USERNAME, errhp);
    OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"penfield",
        (ub4)strlen("penfield"), OCI_ATTR_PASSWORD, errhp);
```

```
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,  
    OCI_DEFAULT));  
OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)usrhp,  
    (ub4)0, OCI_ATTR_SESSION, errhp);
```

Related Functions

OCIAQEnq()

OCIBindArrayOfStruct()

Purpose

This call sets up the skip parameters for a static array bind.

Syntax

```
sword OCIBindArrayOfStruct ( OCIBind      *bindp,  
                             OCIError    *errhp,  
                             ub4          pvskip,  
                             ub4          indskip,  
                             ub4          alskip,  
                             ub4          rcskip );
```

Parameters

bindp (IN/OUT)

The handle to a bind structure.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

pvskip (IN)

Skip parameter for the next data value.

indskip (IN)

Skip parameter for the next indicator value or structure.

alskip (IN)

Skip parameter for the next actual length value.

rcskip (IN)

Skip parameter for the next column-level return code value.

Comments

This call sets up the skip parameters necessary for a static array bind.

This call follows a call to *OCIBindByName()* or *OCIBindByPos()*. The bind handle returned by that initial bind call is used as a parameter for the *OCIBindArrayOfStruct()* call.

For information about skip parameters, see the section “Arrays of Structures” on page 5-17.

Related Functions

OCIBindByName(), *OCIBindByPos()*

OCIBindByName()

Purpose

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.

Syntax

```
sword OCIBindByName ( OCISstmt      *stmtp,
                      OCIBind       **bindpp,
                      OCIError       *errhp,
                      CONST text     *placeholder,
                      sb4            placeh_len,
                      dvoid          *valuep,
                      sb4            value_sz,
                      ub2            dty,
                      dvoid          *indp,
                      ub2            *alenp,
                      ub2            *rcodep,
                      ub4            maxarr_len,
                      ub4            *curelep,
                      ub4            mode );
```

Parameters

stmtp (IN/OUT)

The statement handle to the SQL or PL/SQL statement being processed.

bindpp (IN/OUT)

An address of a bind handle which is implicitly allocated by this call. The bind handle maintains all the bind information for this particular input value. The handle is freed implicitly when the statement handle is deallocated. On input, the value of the pointer must be NULL or a valid bind handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

placeholder (IN)

The placeholder attributes are specified by name if *OCIBindByName()* is being called.

placeh_len (IN)

The length of the placeholder name specified in *placeholder*.

valuep (IN/OUT)

An address of a data value or an array of data values of the type specified in the *dt* parameter. An array of data values can be specified for mapping into a PL/SQL table or for providing data for SQL multiple-row operations. When an array of bind values is provided, this is called an array bind in OCI terms.

For SQLT_NTY or SQLT_REF binds, the *valuep* parameter is ignored. The pointers to OUT buffers are set in the *pgvpp* parameter initialized by *OCIBindObject()*.

value_sz (IN)

The size of a data value. In the case of an array bind, this is the maximum size of any element possible with the actual sizes being specified in the *alenp* parameter.

For descriptors, locators, or REFs, whose size is unknown to client applications use the size of the structure you are passing in; e.g., sizeof (**OCILobLocator ***).

dt (IN)

The data type of the value(s) being bound. Named data types (SQLT_NTY) and REFs (SQLT_REF) are valid only if the application has been initialized in object mode. For named data types, or REFs, additional calls must be made with the bind handle to set up the datatype-specific attributes.

indp (IN/OUT)

Pointer to an indicator variable or array. For all data types except SQLT_NTY, this is a pointer to **sb2** or an array of **sb2s**.

For SQLT_NTY, this pointer is ignored and the actual pointer to the indicator structure or an array of indicator structures is initialized in a subsequent call *OCIBindObject()*. This parameter is ignored for dynamic binds.

See the section “Indicator Variables” on page 2-29 for more information about indicator variables.

alenp (IN/OUT)

Pointer to array of actual lengths of array elements. Each element in *alenp* is the length of the data in the corresponding element in the bind value array before and after the execute. This parameter is ignored for dynamic binds.

rcodep (OUT)

Pointer to array of column level return codes. This parameter is ignored for dynamic binds.

maxarr_len (IN)

The maximum possible number of elements of type *dt* in a PL/SQL binds. This parameter is not required for non-PL/SQL binds. If *maxarr_len* is non-zero, then either *OCIBindDynamic()* or *OCIBindArrayOfStruct()* can be invoked to set up additional bind attributes.

curelep(IN/OUT)

A pointer to the actual number of elements. This parameter is only required for PL/SQL binds.

mode (IN)

The valid modes for this parameter are:

OCI_DEFAULT - This is default mode.

OCI_DATA_AT_EXEC - When this mode is selected, the *value_sz* parameter defines the maximum size of the data that can be ever provided at runtime. The application must be ready to provide the OCI library runtime IN data buffers at any time and any number of times. Runtime data is provided in one of the two ways:

- callbacks using a user-defined function which must be registered with a subsequent call to *OCIBindDynamic()*.
- a polling mechanism using calls supplied by the OCI. This mode is assumed if no callbacks are defined.

For more information about using the OCI_DATA_AT_EXEC mode, see the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

When the allocated buffers are not required any more, they should be freed by the client.

Comments

This call is used to perform a basic bind operation. The bind creates an association between the address of a program variable and a placeholder in a SQL statement or PL/SQL block. The bind call also specifies the type of data which is being bound, and may also indicate the method by which data will be provided at runtime.

This function also implicitly allocates the bind handle indicated by the *bindpp* parameter. If a non-NULL pointer is passed in ***bindpp*, the OCI assumes that this points to a valid handle that has been previously allocated with a call to *OCIHandleAlloc()* or *OCIBindByName()*.

Data in an OCI application can be bound to placeholders statically or dynamically. Binding is *static* when all the IN bind data and the OUT bind buffers are well-defined just before the execute. Binding is *dynamic* when the IN bind data and the OUT bind buffers are provided by the application on demand at execute time to the client library. Dynamic binding is indicated by setting the *mode* parameter of this call to `OCI_DATA_AT_EXEC`.

See Also: For more information about dynamic binding, see the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

Both `OCIBindByName()` and `OCIBindByPos()` take as a parameter a bind handle, which is implicitly allocated by the bind call. A separate bind handle is allocated for each placeholder the application is binding.

Additional bind calls may be required to specify particular attributes necessary when binding certain data types or handling input data in certain ways:

- If arrays of structures are being utilized, `OCIBindArrayOfStruct()` must be called to set up the necessary skip parameters.
- If data is being provided dynamically at runtime, and the application will be using user-defined callback functions, `OCIBindDynamic()` must be called to register the callbacks.
- If a named data type is being bound, `OCIBindObject()` must be called to specify additional necessary information.
- If a statement with RETURNING clause is used, a call to `OCIBindDynamic()` must follow this call.

Related Functions

`OCIBindDynamic()`, `OCIBindObject()`, `OCIBindArrayOfStruct()`

OCIBindByPos()

Purpose

Creates an association between a program variable and a placeholder in a SQL statement or PL/SQL block.

Syntax

```
sword OCIBindByPos ( OCISmt      *stmtp,  
                    OCIBind     **bindpp,  
                    OCIError     *errhp,  
                    ub4          position,  
                    dvoid        *valuep,  
                    sb4          value_sz,  
                    ub2          dty,  
                    dvoid        *indp,  
                    ub2          *alenp,  
                    ub2          *rcodep,  
                    ub4          maxarr_len,  
                    ub4          *curelep,  
                    ub4          mode );
```

Parameters

stmtp (IN/OUT)

The statement handle to the SQL or PL/SQL statement being processed.

bindpp (IN/OUT)

An address of a bind handle which is implicitly allocated by this call. The bind handle maintains all the bind information for this particular input value. The handle is freed implicitly when the statement handle is deallocated. On input, the value of the pointer must be NULL or a valid bind handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

position (IN)

The placeholder attributes are specified by position if *OCIBindByPos()* is being called.

valuep (IN/OUT)

An address of a data value or an array of data values of the type specified in the *dt*y parameter. An array of data values can be specified for mapping into a PL/SQL table or for providing data for SQL multiple-row operations. When an array of bind values is provided, this is called an array bind in OCI terms.

For SQLT_NTY or SQLT_REF binds, the *valuep* parameter is ignored. The pointers to OUT buffers are set in the *pgvpp* parameter initialized by *OCIBindObject()*.

value_sz (IN)

The size of a data value. In the case of an array bind, this is the maximum size of any element possible with the actual sizes being specified in the *alenp* parameter.

For descriptors, locators, or REFs, whose size is unknown to client applications use the size of the structure you are passing in; e.g., sizeof (**OCILobLocator** *).

dty (IN)

The data type of the value(s) being bound. Named data types (SQLT_NTY) and REFs (SQLT_REF) are valid only if the application has been initialized in object mode. For named data types, or REFs, additional calls must be made with the bind handle to set up the datatype-specific attributes.

indp (IN/OUT)

Pointer to an indicator variable or array. For all data types, this is a pointer to **sb2** or an array of **sb2**s. The only exception is SQLT_NTY, when this pointer is ignored and the actual pointer to the indicator structure or an array of indicator structures is initialized by *OCIBindObject()*. Ignored for dynamic binds.

See the section “Indicator Variables” on page 2-29 for more information about indicator variables.

alenp (IN/OUT)

Pointer to array of actual lengths of array elements. Each element in *alenp* is the length of the data in the corresponding element in the bind value array before and after the execute. This parameter is ignored for dynamic binds.

rcodep (OUT)

Pointer to array of column level return codes. This parameter is ignored for dynamic binds.

maxarr_len (IN)

The maximum possible number of elements of type *dt*y in a PL/SQL binds. This parameter is not required for non-PL/SQL binds. If *maxarr_len* is non-zero, then

either *OCIBindDynamic()* or *OCIBindArrayOfStruct()* can be invoked to set up additional bind attributes.

curelep(IN/OUT)

A pointer to the actual number of elements. This parameter is only required for PL/SQL binds.

mode (IN)

The valid modes for this parameter are:

OCI_DEFAULT - This is default mode.

OCI_DATA_AT_EXEC - When this mode is selected, the *value_sz* parameter defines the maximum size of the data that can be ever provided at runtime. The application must be ready to provide the OCI library runtime IN data buffers at any time and any number of times. Runtime data is provided in one of the two ways:

- callbacks using a user-defined function which must be registered with a subsequent call to *OCIBindDynamic()*.
- a polling mechanism using calls supplied by the OCI. This mode is assumed if no callbacks are defined.

For more information about using the OCI_DATA_AT_EXEC mode, see the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

When the allocated buffers are not required any more, they should be freed by the client.

Comments

This call is used to perform a basic bind operation. The bind creates an association between the address of a program variable and a placeholder in a SQL statement or PL/SQL block. The bind call also specifies the type of data which is being bound, and may also indicate the method by which data will be provided at runtime.

This function also implicitly allocates the bind handle indicated by the *bindpp* parameter. If a non-NULL pointer is passed in ***bindpp*, the OCI assumes that this points to a valid handle that has been previously allocated with a call to *OCIHandleAlloc()* or *OCIBindByPos()*.

Data in an OCI application can be bound to placeholders statically or dynamically. Binding is *static* when all the IN bind data and the OUT bind buffers are well-defined just before the execute. Binding is *dynamic* when the IN bind data and the OUT bind buffers are provided by the application on demand at execute time to the

client library. Dynamic binding is indicated by setting the *mode* parameter of this call to OCI_DATA_AT_EXEC.

See Also: For more information about dynamic binding, see the section “Run Time Data Allocation and Piecewise Operations” on page 7-16

Both *OCIBindByName()* and *OCIBindByPos()* take as a parameter a bind handle, which is implicitly allocated by the bind call. A separate bind handle is allocated for each placeholder the application is binding.

Additional bind calls may be required to specify particular attributes necessary when binding certain data types or handling input data in certain ways:

- If arrays of structures are being utilized, *OCIBindArrayOfStruct()* must be called to set up the necessary skip parameters.
- If data is being provided dynamically at runtime, and the application will be using user-defined callback functions, *OCIBindDynamic()* must be called to register the callbacks.
- If a named data type is being bound, *OCIBindObject()* must be called to specify additional necessary information.
- If a statement with RETURNING clause is used, a call to *OCIBindDynamic()* must follow this call.

Related Functions

OCIBindDynamic(), *OCIBindObject()*, *OCIBindArrayOfStruct()*

OCIBindDynamic()

Purpose

This call is used to register user callbacks for dynamic data allocation.

Syntax

```
sword OCIBindDynamic ( OCIBind      *bindp,
                      OCIError      *errhp,
                      dvoid          *ictxp,
                      OCICallbackInBind (icbfp) (/*_
                      dvoid          *ictxp,
                      OCIBind      *bindp,
                      ub4           iter,
                      ub4           index,
                      dvoid         **bufpp,
                      ub4           *alenp,
                      ub1           *piecep,
                      dvoid         **indpp */),
                      dvoid          *octxp,
                      OCICallbackOutBind (ocbfp) (/*_
                      dvoid          *octxp,
                      OCIBind      *bindp,
                      ub4           iter,
                      ub4           index,
                      dvoid         **bufpp,
                      ub4           *alenpp,
                      ub1           *piecep,
                      dvoid         **indpp,
                      ub2           **rcodepp _ */) );
```

Parameters

bindp (IN/OUT)

A bind handle returned by a call to *OCIBindByName()* or *OCIBindByPos()*.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

ictxp (IN)

The context pointer required by the call back function *icbfp*.

icbfp (IN)

The callback function which returns a pointer to the IN bind value or piece at run time. The callback takes in the following parameters:

ictxp (IN/OUT)

The context pointer for this callback function.

bindp (IN)

The bind handle passed in to uniquely identify this bind variable.

iter (IN)

0-based execute iteration value.

index (IN)

Index of the current array, for an array bind in PL/SQL. For SQL it is the row index. The value is 0-based and not greater than *curelep* parameter of the bind call.

bufpp (OUT)

The pointer to the buffer or storage. For descriptors, **bufpp* contains a pointer to the descriptor. For example if you define

```
OCILOBLocator    *lobp;
```

then you would set **bufpp* to *lobp* not **lobp*.

For REFs, pass the address of the ref; i.e., pass *&my_ref* for **bufpp*.

alenp (OUT)

A pointer to a storage for OCI to fill in the size of the bind value/piece after it has been read. For descriptors, pass the size of the pointer to the descriptor; e.g., `sizeof(OCILOBLocator *)`.

piecep (OUT)

Which piece of the bind value. This can be one of the following values OCI_ONE_PIECE, OCI_FIRST_PIECE, OCI_NEXT_PIECE and OCI_LAST_PIECE. For datatypes that do not support piecewise operations, you must pass OCI_ONE_PIECE or an error will be generated.

indp (OUT)

Contains the indicator value. This is a pointer to either an **sb2** value or a pointer to an indicator structure for binding named data types.

octxp (IN)

The context pointer required by the callback function *ocbfp*.

ocbfp (IN)

The callback function which returns a pointer to the OUT bind value or piece at run time. The callback takes in the following parameters:

octxp (IN/OUT)

The context pointer for this call back function.

bindp (IN)

The bind handle passed in to uniquely identify this bind variable.

iter (IN)

0-based execute iteration value.

index (IN)

For PL/SQL index of the current array, for an array bind. For SQL, the index is the row number in the current iteration. It is 0-based, and must not be greater than *curelep* parameter of the bind call.

bufpp (OUT)

A pointer to a buffer to write the bind value/piece.

alenpp (IN/OUT)

A pointer to a storage for OCI to fill in the size of the bind value/piece after it has been read.

piecep (IN/OUT)

Returns a piece value from the callback (application) to Oracle, as follows:

- **IN** - The value can be OCI_ONE_PIECE or OCI_NEXT_PIECE.
- **OUT** - Depends on the IN value:
 - If IN value is OCI_ONE_PIECE, then OUT value can be OCI_ONE_PIECE or OCI_FIRST_PIECE
 - If IN value is OCI_NEXT_PIECE then OUT value can be OCI_NEXT_PIECE or OCI_LAST_PIECE

indpp (OUT)

Returns a pointer to contain the indicator value which either an **sb2** value or a pointer to an indicator structure for named data types.

rancodepp (OUT)

Returns a pointer to contains the return code.

Comments

This call is used to register user-defined callback functions for providing or receiving data if OCI_DATA_AT_EXEC mode was specified in a previous call to *OCIBindByName()* or *OCIBindByPos()*.

The callback function pointers must return OCI_CONTINUE if it the call is successful. Any return code other than OCI_CONTINUE signals that the client wishes to abort processing immediately.

For more information about the OCI_DATA_AT_EXEC mode, see the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

When passing the address of a storage area, make sure that the storage area will exist even after the application returns from the callback. This means that you should not allocate such storage on the stack.

Related Functions

OCIBindByName(), *OCIBindByPos()*

OCIBindObject()

Purpose

This function sets up additional attributes which are required for a named data type (object) bind.

Syntax

```
sword OCIBindObject ( OCIBind      *bindp,  
                     OCIError     *errhp,  
                     CONST OCIType *type,  
                     dvoid        **pgvpp,  
                     ub4          *pvszsp,  
                     dvoid        **indpp,  
                     ub4          *indszp, );
```

Parameters

bindp (IN/OUT)

The bind handle returned by the call to *OCIBindByName()* or *OCIBindByPos()*.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

type (IN)

Points to the TDO which describes the type of the program variable being bound. Retrieved by calling *OCITypeByName()*. Optional for REFs in SQL, but required for REFs in PL/SQL.

pgvpp (IN/OUT)

Address of the program variable buffer. For an array, *pgvpp* points to an array of addresses. When the bind variable is also an OUT variable, the OUT Named Data Type value or REF is allocated in the Object Cache, and a REF is returned.

pgvpp is ignored if the OCI_DATA_AT_EXEC mode is set. Then the Named Data Type buffers are requested at runtime. For static array binds, skip factors may be specified using the *OCIBindArrayOfStruct()* call. The skip factors are used to compute the address of the next pointer to the value, the indicator structure and their sizes.

pvszsp (OUT) [optional]

Points to the size of the program variable. The size of the named data type is not required on input. For an array, *pvszsp* is an array of **ub4s**. On return, for OUT bind variables, this points to size(s) of the Named Data Types and REFs received. *pvszsp* is ignored if the OCI_DATA_AT_EXEC mode is set. Then the size of the buffer is taken at runtime.

indpp (IN/OUT)[optional]

Address of the program variable buffer containing the parallel indicator structure. For an array, points to an array of pointers. When the bind variable is also an OUT bind variable, memory is allocated in the object cache, to store the OUT indicator values. At the end of the execute when all OUT values have been received, *indpp* points to the pointer(s) to these newly allocated indicator structure(s). Required only for SQLT_NTY binds.

indpp is ignored if the OCI_DATA_AT_EXEC mode is set. Then the indicator is requested at runtime.

indszp (IN/OUT)

Points to the size of the IN indicator structure program variable. For an array, it is an array of **sb2s**. On return for OUT bind variables, this points to size(s) of the received OUT indicator structures.

indszp is ignored if the OCI_DATA_AT_EXEC mode is set. Then the indicator size is requested at runtime.

Comments

This function sets up additional attributes which binding a named data type or a REF. An error will be returned if this function is called when the OCI environment has been initialized in non-object mode.

This call takes as a parameter a type descriptor object (TDO) of datatype **OCIType** for the named data type being defined. The TDO can be retrieved with a call to *OCITypeByName()*.

If the OCI_DATA_AT_EXEC mode was specified in *OCIBindByName()* or *OCIBindByPos()*, the pointers to the IN buffers are obtained either using the callback *icbfp* registered in the *OCIBindDynamic()* call or by the *OCISmtSetPieceInfo()* call. The buffers are dynamically allocated for the OUT data and the pointers to these buffers are returned either by calling *ocbfp()* registered by the *OCIBindDynamic()* or by setting the pointer to the buffer in the buffer passed in by *OCISmtSetPieceInfo()* called when *OCISmtExecute()* returned

OCI_NEED_DATA. The memory of these client library-allocated buffers must be freed when not in use anymore by using the *OCIObjectFree()* call.

Related Functions

OCIBindByName(), *OCIBindByPos()*

OCIBreak()

Purpose

This call performs an immediate (asynchronous) abort of any currently executing OCI function that is associated with a server.

Syntax

```
sword OCIBreak ( dvoid      *hndlp,  
                 OCIError   *errhp );
```

Parameters

hndlp (IN/OUT)

The service context handle or the server context handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

Comments

This call performs an immediate (asynchronous) abort of any currently executing OCI function that is associated with a server. It is normally used to stop a long-running OCI call being processed on the server.

This call can take either the service context handle or the server context handle as a parameter to identify the function to be aborted.

Related Functions

OCIDefineArrayOfStruct()

Purpose

This call specifies additional attributes necessary for a static array define.

Syntax

```
sword OCIDefineArrayOfStruct ( OCIDefine    *defnp,  
                               OCIError     *errhp,  
                               ub4          pvskip,  
                               ub4          indskip,  
                               ub4          rlskip,  
                               ub4          rcskip );
```

Parameters

defnp (IN/OUT)

The handle to the define structure which was returned by a call to *OCIDefineByPos()*.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

pvskip (IN)

Skip parameter for the next data value.

indskip (IN)

Skip parameter for the next indicator location.

rlskip (IN)

Skip parameter for the next return length value.

rcskip (IN)

Skip parameter for the next return code.

Comments

This call specifies additional attributes necessary for an array define, used in an array of structures (multi-row, multi-column) fetch. This call follows a call to *OCIDefineByPos()*.

For more information about skip parameters, see the section “Skip Parameters” on page 5-18.

If the application is binding an array of structures involving objects, it must call *OCIDefineObject()* first, and then call *OCIDefineArrayOfStruct()*.

Related Functions

OCIDefineByPos(), *OCIDefineObject()*

OCIDefineByPos()

Purpose

Associates an item in a select-list with the type and output data buffer.

Syntax

```
sword OCIDefineByPos ( OCISstmt      *stmtp,
                      OCIDefine     **defnpp,
                      OCIError      *errhp,
                      ub4            position,
                      dvoid          *valuep,
                      sb4            value_sz,
                      ub2            dtc,
                      dvoid          *indp,
                      ub2            *rlenp,
                      ub2            *rcodep,
                      ub4            mode );
```

Parameters

stmtp (IN/OUT)

A handle to the requested SQL query operation.

defnpp (IN/OUT)

A pointer to a pointer to a define handle. If this parameter is passed as NULL, this call implicitly allocates the define handle. In the case of a redefine, a non-NULL handle can be passed in this parameter. This handle is used to store the define information for this column.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

position (IN)

The position of this value in the select list. Positions are 1-based and are numbered from left to right. For example, in the SELECT statement

```
SELECT empno, ssn, mgrno FROM employees;
```

empno is at position 1, *ssn* is at position 2, and *mgrno* is at position 3.

valuep (IN/OUT)

A pointer to a buffer or an array of buffers of the type specified in the *dt*y parameter. A number of buffers can be specified when results for more than one row are desired in a single fetch call.

value_sz (IN)

The size of each *valuep* buffer in bytes. If the data is stored internally in VARCHAR2 format, the number of characters desired, if different from the buffer size in bytes, may be additionally specified by the using *OCIAttrSet()*.

In an NLS conversion environment, a truncation error will be generated if the number of bytes specified is insufficient to handle the number of characters desired.

dt (IN)

The data type. Named data type (SQT_NTY) and REF (SQT_REF) are valid only if the environment has been initialized with in object mode. For a listing of datatype codes and values, refer to Chapter 3, “Datatypes”.

indp (IN)

pointer to an indicator variable or array. For scalar data types, pointer to **sb2** or an array of **sb2**s. Ignored for SQT_NTY defines. For SQT_NTY defines, a pointer to a named data type indicator structure or an array of named data type indicator structures is associated by a subsequent *OCIDefineObject()* call.

See the section “Indicator Variables” on page 2-29 for more information about indicator variables.

rlemp (IN/OUT)

Pointer to array of length of data fetched. Each element in *rlemp* is the length of the data in the corresponding element in the row after the fetch.

rcodep (OUT)

Pointer to array of column-level return codes

mode (IN)

The valid modes are:

- OCI_DEFAULT - This is the default mode.
- OCI_DYNAMIC_FETCH - For applications requiring dynamically allocated data at the time of fetch, this mode must be used. The user may additionally call *OCIDefineDynamic()* to set up a callback function that will be invoked to receive the dynamically allocated buffers and. The *valuep* and *value_sz* parameters are ignored in this mode.

Comments

This call defines an output buffer which will receive data retrieved from Oracle. The define is a local step which is necessary when a SELECT statement returns data to your OCI application.

This call also implicitly allocates the define handle for the select-list item. If a non-NULL pointer is passed in **defnpp*, the OCI assumes that this points to a valid handle that has been previously allocated with a call to *OCIHandleAlloc()* or *OCIDefineByPos()*. This would be true in the case of an application which is redefining a handle to a different addresses so it can reuse the same define handle for multiple fetches.

Defining attributes of a column for a fetch is done in one or more calls. The first call is to *OCIDefineByPos()*, which defines the minimal attributes required to specify the fetch.

Following the call to *OCIDefineByPos()* additional define calls may be necessary for certain data types or fetch modes:

- A call to *OCIDefineArrayOfStruct()* is necessary to set up skip parameters for an array fetch of multiple columns.
- A call to *OCIDefineObject()* is necessary to set up the appropriate attributes of a named data type (i.e., object or collection) or REF fetch. In this case the data buffer pointer in *OCIDefineByPos()* is ignored.
- Both *OCIDefineArrayOfStruct()* and *OCIDefineObject()* must be called after *OCIDefineByPos()* in order to fetch multiple rows with a column of named data types.

For a LOB define, the buffer pointer must be a pointer to a lob locator of type **OCILobLocator**, allocated by the *OCIDescriptorAlloc()* call. LOB locators, and not LOB values, are always returned for a LOB column. LOB values can then be fetched using OCI LOB calls on the fetched locator. This same mechanism is true for all descriptor datatypes.

For NCHAR (fixed and varying length), the buffer pointer must point to an array of bytes sufficient for holding the required NCHAR characters.

Nested table columns are defined and fetched like any other named data type.

When defining an array of descriptors or locators, you should pass in an array of pointers to descriptors or locators.

When doing an array define for character columns, you should pass in an array of character buffers.

If the *mode* parameter of this call is set to OCI_DYNAMIC_FETCH, the client application can fetch data dynamically at runtime. Runtime data can be provided in one of two ways:

- callbacks using a user-defined function which must be registered with a subsequent call to *OCIDefineDynamic()*. When the client library needs a buffer to return the fetched data, the callback will be invoked and the runtime buffers provided will return a piece or the whole data.
- a polling mechanism using calls supplied by the OCI. This mode is assumed if no callbacks are defined. In this case, the fetch call returns the OCI_NEED_DATA error code, and a piecewise polling method is used to provide the data.

See Also: For more information about using the OCI_DYNAMIC_FETCH mode, see the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

For more information about defines, see “Defining” on page 5-13.

Related Functions

OCIDefineArrayOfStruct(), *OCIDefineDynamic()*, *OCIDefineObject()*

OCIDefineDynamic()

Purpose

This call is used to set the additional attributes required if the OCI_DYNAMIC_FETCH mode was selected in *OCIDefineByPos()*.

Syntax

```
sword OCIDefineDynamic ( OCIDefine      *defnp,  
                        OCIError       *errhp,  
                        dvoid          *octxp,  
                        OCICallbackDefine (ocbfp) (/*_  
                        dvoid          *octxp,  
                        OCIDefine      *defnp,  
                        ub4            iter,  
                        dvoid          **bufpp,  
                        ub4            **alenpp,  
                        ub1            *piecep,  
                        dvoid          **indpp,  
                        ub2            **rcodep _*/) );
```

Parameters

defnp (IN/OUT)

The handle to a define structure returned by a call to *OCIDefineByPos()*.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

octxp (IN)

Points to a context for the callback function.

ocbfp (IN)

Points to a callback function. This is invoked at runtime to get a pointer to the buffer into which the fetched data or a piece of it will be retrieved. The callback also specifies the indicator, the return code and the lengths of the data piece and indicator.

Warning: When working with callback parameters, it is important to keep in mind what is meant by IN and OUT for the parameter mode. Normally, in an

OCI function, an IN parameter refers to data being passed to Oracle, and an OUT parameter refers to data coming back from Oracle. In the case of callbacks, this is reversed. IN means data is coming from Oracle into the callback, and OUT means data is coming out of the callback and going to Oracle.

The callback parameters are listed below:

octxp (IN/OUT)

A context pointer passed as an argument to all the callback functions.

defnp (IN)

The define handle.

iter (IN)

Which row of this current fetch; 0-based.

bufpp (OUT)

Returns to Oracle a pointer to a buffer to store the column value, i.e., **bufpp* points to some appropriate storage for the column value.

alenpp (IN/OUT)

Used by the application to set the size of the storage it is providing in **bufpp*. After data is fetched into the buffer, *alenpp* indicates the actual size of the data.

piecep (IN/OUT)

Returns a piece value from the callback (application) to Oracle, as follows:

- **IN** - The value can be OCI_ONE_PIECE or OCI_NEXT_PIECE.
- **OUT** - Depends on the IN value:
 - If IN value is OCI_ONE_PIECE, then OUT value can be OCI_ONE_PIECE or OCI_FIRST_PIECE
 - If IN value is OCI_NEXT_PIECE then OUT value can be OCI_NEXT_PIECE or OCI_LAST_PIECE

indpp (IN)

Indicator variable pointer

rcodep (IN)

Return code variable pointer

Comments

This call is used to set the additional attributes required if the OCI_DYNAMIC_FETCH mode has been selected in a call to *OCIDefineByPos()*.

If OCI_DYNAMIC_FETCH mode was selected, and the call to *OCIDefineDynamic()* is skipped, then the application can fetch data piecewise using OCI calls (*OCIStmtGetPieceInfo()* and *OCIStmtSetPieceInfo()*).

For more information about OCI_DYNAMIC_FETCH mode, see the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

Related Functions

OCIDefineByPos()

OCIDefineObject()

Purpose

Sets up additional attributes necessary for a Named Data Type or REF define.

Syntax

```
sword OCIDefineObject ( OCIDefine      *defnp,  
                        OCIError       *errhp,  
                        CONST OCIType   *type,  
                        dvoid          **pgvpp,  
                        ub4             *pvszsp,  
                        dvoid          **indpp,  
                        ub4             *indszp );
```

Parameters

defnp (IN/OUT)

A define handle previously allocated in a call to *OCIDefineByPos()*.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

type (IN) [optional]

Points to the Type Descriptor Object (TDO) which describes the type of the program variable. Only used for program variables of type *SQLT_NTY*. This parameter is optional, and may be passed as *NULL* if it is not being used.

pgvpp (IN/OUT)

Points to a pointer to a program variable buffer. For an array, *pgvpp* points to an array of pointers. Memory for the fetched named data type instance(s) is dynamically allocated in the object cache. At the end of the fetch when all the values have been received, *pgvpp* points to the pointer(s) to these newly allocated named data type instance(s). The application must call *OCIObjectFree()* to deallocate the named data type instance(s) when they are no longer needed.

Note: If the application wants the buffer to be implicitly allocated in the cache, **pgvpp* should be passed in as *NULL*.

pvszsp (IN/OUT)

Points to the size of the program variable. For an array, it is an array of **ub4s**.

indpp (IN/OUT)

Points to a pointer to the program variable buffer containing the parallel indicator structure. For an array, points to an array of pointers. Memory is allocated to store the indicator structures in the object cache. At the end of the fetch when all values have been received, *indpp* points to the pointer(s) to these newly allocated indicator structure(s).

indszp (IN/OUT)

Points to the size(s) of the indicator structure program variable. For an array, it is an array of **ub4s**.

Comments

This function follows a call to *OCIDefineByPos()* to set initial define information. This call sets up additional attributes necessary for a Named Data Type define. An error will be returned if this function is called when the OCI environment has been initialized in non-Object mode.

This call takes as a parameter a type descriptor object (TDO) of datatype **OCIType** for the named data type being defined. The TDO can be retrieved with a call to *OCIDescribeAny()*.

See Also: See the description of *OCIInitialize()* on page 13 - 72 for more information about initializing the OCI process environment.

Related Functions

OCIDefineByPos()

OCIDescribeAny()

Purpose

Describes existing schema objects.

Syntax

```
sword OCIDescribeAny ( OCISvcCtx      *svchp,
                      OCIError      *errhp,
                      dvoid          *objptr,
                      ub4            objnm_len,
                      ub1            objptr_typ,
                      ub1            info_level,
                      ub1            objtyp,
                      OCIDescribe   *dschp );
```

Parameters

svchp (IN)

A service context handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

objptr (IN)

This parameter can be either

1. a string containing the name of the schema object to be described
2. a pointer to a REF to the TDO (for a type)
3. a pointer to a TDO (for a type).

These cases are distinguished by passing the appropriate value for *objptr_typ*. This parameter must be non-NULL.

In case 1, the string containing the object name should be in the format <schema-name>.<object-name>. No database links are allowed.

The object name is interpreted by the following SQL rules:

- If <schema-name> is NULL, the name refers to the object (of type table / view / procedure / function / package / type / synonym / sequence) with name

described by <object-name> in the schema of the current user. When connected to an Oracle7 Server, the only valid types are procedure and function.

- If <schema-name> is non-NULL, the name refers to the object with name described by <object-name>, in the schema with name described by <schema-name>.

objnm_len (IN)

The length of the name string pointed to by *objptr*. Must be non-zero if a name is passed. Can be zero if *objptr* is a pointer to a TDO or its REF.

objptr_type (IN)

The type of object passed in *objptr*. Valid values are:

- OCI_OTYPE_NAME, if *objptr* points to the name of a schema object
- OCI_OTYPE_REF, if *objptr* is a pointer to a REF to a TDO
- OCI_OTYPE_PTR, if *objptr* is a pointer to a TDO

info_level (IN)

Reserved for future extensions. Pass OCI_DEFAULT.

objtyp (IN/OUT)

The type of schema object being described. Valid values are:

- OCI_PTYPE_TABLE, for tables
- OCI_PTYPE_VIEW, for views
- OCI_PTYPE_PROC, for procedures
- OCI_PTYPE_FUNC, for functions
- OCI_PTYPE_PKG, for packages
- OCI_PTYPE_TYPE, for types
- OCI_PTYPE_SYN, for synonyms
- OCI_PTYPE_SEQ, for sequences
- OCI_PTYPE_UNK, for unknown schema objects

A value for this argument must be specified. If OCI_PTYPE_UNK is specified, then the description of an object with the specified name in the current schema is returned, if such an object exists, along with the actual type of the object.

dschp (IN/OUT)

A describe handle that is populated with describe information about the object after the call. Must be non-NULL.

Comments

This is a generic describe call that describes existing schema objects: tables, views, synonyms, procedures, functions, packages, sequences, and types. This call populates the describe handle with the object-specific attributes which can be obtained through an *OCIAttrGet()* call.

An *OCIParmGet()* on the describe handle returns a parameter descriptor for a specified position. Parameter positions begin with 1. Calling *OCIAttrGet()* on the parameter descriptor returns the specific attributes of a stored procedure or function parameter or a table column descriptor as the case may be.

These subsequent calls do not need an extra round trip to the server because the entire schema object description is cached on the client side by *OCIDescribeAny()*. Calling *OCIAttrGet()* on the describe handle can also return the total number of positions.

See Chapter 6, “Describing Schema Metadata”, for more information about describe operations.

Related Functions

OCIAQEnq(), *OCIParmGet()*

OCIDescriptorAlloc()

Purpose

Allocates storage to hold descriptors or LOB locators.

Syntax

```
sword OCIDescriptorAlloc ( CONST dvoid    *parenth,  
                           dvoid          **descpp,  
                           ub4            type,  
                           size_t         xtramen_sz,  
                           dvoid          **usrmemp );
```

Parameters

parenth (IN)

An environment handle.

descpp (OUT)

Returns a descriptor or LOB locator of desired type.

type (IN)

Specifies the type of descriptor or LOB locator to be allocated:

- OCI_DTYPE_SNAP - specifies generation of snapshot descriptor of C type **OCISnapshot**
- OCI_DTYPE_LOB - specifies generation of a LOB value type locator (for a BLOB or CLOB) of C type **OCILobLocator**
- OCI_DTYPE_FILE - specifies generation of a FILE value type locator of C type **OCILobLocator**.
- OCI_DTYPE_ROWID - specifies generation of a ROWID descriptor of C type **OCIRowid**.
- OCI_DTYPE_COMPLEXOBJECTCOMP - specifies generation of a complex object retrieval descriptor of C type **OCIComplexObjectComp**.
- OCI_DTYPE_AQENQ_OPTIONS - specifies generation of an advanced queueing enqueue options descriptor of C type **OCIAQEnqOptions**.
- OCI_DTYPE_AQDEQ_OPTIONS - specifies generation of an advanced queueing dequeue options descriptor of C type **OCIAQDeqOptions**.

- `OCI_DTYPE_AQMSG_PROPERTIES` - specifies generation of an advanced queueing message properties descriptor of C type **`OCIAQMsgProperties`**.
- `OCI_DTYPE_AQAGENT` - specifies generation of an advanced queueing agent descriptor of C type **`OCIAQAgent`**.

`xtrmem_sz` (IN)

Specifies an amount of user memory to be allocated for use by the application for the lifetime of the descriptor.

`usrmempp` (OUT)

Returns a pointer to the user memory of size *xtrmem_sz* allocated by the call for the user for the lifetime of the descriptor.

Comments

Returns a pointer to an allocated and initialized descriptor, corresponding to the type specified in *type*. A non-NULL descriptor or LOB locator is returned on success. No diagnostics are available on error.

This call returns `OCI_SUCCESS` if successful, or `OCI_INVALID_HANDLE` if an out-of-memory error occurs.

For more information about the *xtrmem_sz* parameter and user memory allocation, refer to “User Memory Allocation” on page 2-12.

Related Functions

OCIDescriptorFree()

OCIDescriptorFree()

Purpose

Deallocates a previously allocated descriptor.

Syntax

```
sword OCIDescriptorFree ( dvoid      *descp,  
                           ub4        type );
```

Parameters

descp (IN)

An allocated descriptor.

type (IN)

Specifies the type of storage to be freed. The specific types are:

- OCL_DTYPE_SNAP - snapshot descriptor
- OCL_DTYPE_LOB - a LOB value type descriptor
- OCL_DTYPE_FILE - a FILE value type descriptor
- OCL_DTYPE_ROWID - a ROWID descriptor
- OCL_DTYPE_COMPLEXOBJECTCOMP - a complex object retrieval descriptor
- OCL_DTYPE_AQENQ_OPTIONS - an AQ enqueue options descriptor
- OCL_DTYPE_AQDEQ_OPTIONS - an AQ dequeue options descriptor
- OCL_DTYPE_AQMSG_PROPERTIES - an AQ message properties descriptor
- OCL_DTYPE_AQAGENT - an AQ agent descriptor

Comments

This call frees storage associated with a descriptor. Returns OCI_SUCCESS or OCI_INVALID_HANDLE. All descriptors may be explicitly deallocated, however the OCI will deallocate a descriptor if the environment handle is deallocated.

Related Functions

OCIDescriptorAlloc()

OCIEnvInit()

Purpose

This call allocates and initializes an OCI environment handle.

Syntax

```
sword OCIEnvInit ( OCIEnv      **envhpp,  
                  ub4         mode,  
                  size_t      xtramsz,  
                  dvoid       **usrmempp );
```

Parameters

envhpp (OUT)

A pointer to a handle to the environment.

mode (IN)

Specifies initialization of an environment mode. Valid modes are:

- OCI_DEFAULT
- OCI_NO_MUTEX.

In OCI_DEFAULT mode, the OCI library always mutexes handles. In OCI_NO_MUTEX modes, there is no mutexing in this environment.

xtramsz (IN)

Specifies the amount of user memory to be allocated for the duration of the environment.

usrmempp (OUT)

Returns a pointer to the user memory of size *xtramsz* allocated by the call for the user for the duration of the environment.

Comments

This call allocates and initializes an OCI environment handle. No changes are done to an already initialized handle. If OCI_ERROR or OCI_SUCCESS_WITH_INFO is returned, the environment handle can be used to obtain ORACLE specific errors and diagnostics.

This call is processed locally, without a server round-trip.

The environment handle can be freed using *OCIHandleFree()*.

For more information about the *xtramemsz* parameter and user memory allocation, refer to “User Memory Allocation” on page 2-12.

Related Functions

OCIHandleAlloc(), *OCIHandleFree()*

OCIErrorGet()

Purpose

Returns an error message in the buffer provided and an ORACLE error.

Syntax

```
sword OCIErrorGet ( dvoid      *hndlp,  
                   ub4        recordno,  
                   text       *sqlstate,  
                   sb4        *errcodep,  
                   text       *bufp,  
                   ub4        bufsiz,  
                   ub4        type );
```

Parameters

hndlp (IN)

The error handle, in most cases, or the environment handle (for errors on *OCIEnvInit()*, *OCIHandleAlloc()*).

recordno (IN)

Indicates the status record from which the application seeks info. Starts from 1.

sqlstate (OUT)

Not supported in Version 8.0.

errcodep (OUT)

An ORACLE Error is returned.

bufp (OUT)

The error message text is returned.

bufsiz (IN)

The size of the buffer provide to get the error message.

type (IN)

The type of the handle (OCI_HTYPE_ERR or OCI_HTYPE_ENV).

Comments

Returns an error message in the buffer provided and an ORACLE error code. This function does not support SQL state. This function can be called multiple times if there are more than one diagnostic record for an error.

The error handle is originally allocated with a call to *OCIHandleAlloc()*.

Example

The following sample code demonstrates how you can use *OCIErrorGet()* in an error-handling routine. This routine prints out the type of status code returned by an OCI function, and if an error occurred, *OCIErrorGet()* retrieves the text of the message, which is printed.

```
static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    ub4 buflen;
    ub4 errcode;

    switch (status)
    {
        case OCI_SUCCESS:
            break;
        case OCI_SUCCESS_WITH_INFO:
            printf("ErrorOCI_SUCCESS_WITH_INFO\n");
            break;
        case OCI_NEED_DATA:
            printf("ErrorOCI_NEED_DATA\n");
            break;
        case OCI_NO_DATA:
            printf("ErrorOCI_NO_DATA\n");
            break;
        case OCI_ERROR:
            OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                        errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
            printf("Error%s\n", errbuf);
            break;
        case OCI_INVALID_HANDLE:
            printf("ErrorOCI_INVALID_HANDLE\n");
            break;
        case OCI_STILL_EXECUTING:
            printf("ErrorOCI_STILL_EXECUTE\n");
```

```
        break;
    case OCI_CONTINUE:
        printf("ErrorOCI_CONTINUE\n");
        break;
    default:
        break;
    }
}
```

Related Functions

OCIHandleAlloc()

OCIHandleAlloc()

Purpose

This call returns a pointer to an allocated and initialized handle.

Syntax

```
sword OCIHandleAlloc ( CONST dvoid    *parenth,
                      dvoid          **hndlpp,
                      ub4            type,
                      size_t         xtramen_sz,
                      dvoid          **usrmempp );
```

Parameters

parenth (IN)

An environment handle.

hndlpp (OUT)

Returns a handle.

type (IN)

Specifies the type of handle to be allocated. The allowed types are:

- OCI_HTYPE_ERROR - specifies generation of an error report handle of C type **OCIError**
- OCI_HTYPE_SVCCTX - specifies generation of a service context handle of C type **OCISvcCtx**
- OCI_HTYPE_STMT - specifies generation of a statement (application request) handle of C type **OCIStmt**
- OCI_HTYPE_DESCRIBE - specifies generation of a select list description handle of C type **OCIDescribe**
- OCI_HTYPE_SERVER - specifies generation of a server context handle of C type **OCIServer**
- OCI_HTYPE_SESSION - specifies generation of a user session handle of C type **OCISession**
- OCI_HTYPE_TRANS - specifies generation of a transaction context handle of C type **OCITrans**

- `OCI_HTYPE_COMPLEXOBJECT` - specifies generation of a complex object retrieval handle of C type **OCIComplexObject**
- `OCI_HTYPE_SECURITY` - specifies generation of a security handle of C type **OCISecurity**

xtrmem_sz (IN)

Specifies an amount of user memory to be allocated.

usrmemp (OUT)

Returns a pointer to the user memory of size *xtrmem_sz* allocated by the call for the user.

Comments

Returns a pointer to an allocated and initialized handle, corresponding to the type specified in *type*. A non-NULL handle is returned on success. All handles are allocated with respect to an environment handle which is passed in as a parent handle.

No diagnostics are available on error. This call returns `OCI_SUCCESS` if successful, or `OCI_INVALID_HANDLE` if an error occurs.

Handles must be allocated using *OCIHandleAlloc()* before they can be passed into an OCI call.

To allocate and initialize an environment handle, call *OCIEnvInit()*.

See Also: For more information about using the *xtrmem_sz* parameter for user memory allocation, refer to “User Memory Allocation” on page 2-12.

Example

The following sample code shows *OCIHandleAlloc()* being used to allocate a variety of handles at the beginning of an application:

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4)
    OCI_HTYPE_ERROR, 0, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4)
    OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4)
    OCI_HTYPE_SVCCTX, 0, (dvoid **) &tmp);
```

Related Functions

OCIHandleFree(), *OCIEnvInit()*

OCIHandleFree()

Purpose

This call explicitly deallocates a handle.

Syntax

```
sword OCIHandleFree ( dvoid      *hndlp,  
                      ub4        type );
```

Parameters

hndlp (IN)

A handle allocated by *OCIHandleAlloc()*.

type (IN)

Specifies the type of storage to be freed. The specific types are:

- OCL_HTYPE_ENV - an environment handle
- OCL_HTYPE_ERROR - an error report handle
- OCL_HTYPE_SVCCTX - a service context handle
- OCL_HTYPE_STMT - a statement (application request) handle
- OCL_HTYPE_DESCRIBE - a select list description handle
- OCL_HTYPE_SERVER - a server handle
- OCL_HTYPE_SESSION - a user session handle
- OCL_HTYPE_TRANS - a transaction handle
- OCL_HTYPE_COMPLEXOBJECT - a complex object retrieval handle
- OCL_HTYPE_SECURITY - a security handle

Comments

This call frees up storage associated with a handle, corresponding to the type specified in the *type* parameter.

This call returns either OCI_SUCCESS or OCI_INVALID_HANDLE.

All handles may be explicitly deallocated. The OCI will deallocate a child handle if the parent is deallocated.

Related Functions

OCIHandleAlloc(), OCIEnvInit()

OCIInitialize()

Purpose

Initializes the OCI process environment.

Syntax

```
sword OCIInitialize ( ub4          mode,
                     CONST dvoid  *ctxp,
                     CONST dvoid  *(*malocfp)
                     /* dvoid *ctxp,
                      size_t size _*/),
                     CONST dvoid  *(*ralocfp)
                     /* _ dvoid *ctxp,
                      dvoid *memptr,
                      size_t newsize _*/),
                     CONST void   (*mfreefp)
                     /* _ dvoid *ctxp,
                      dvoid *memptr _/));
```

Parameters

mode (IN)

Specifies initialization of the mode. The valid modes are:

- OCI_DEFAULT - default mode.
- OCI_THREADED - threaded environment. In this mode, internal data structures not exposed to the user are protected from concurrent accesses by multiple threads.
- OCI_OBJECT - will use object features.

ctxp (IN)

User defined context for the memory call back routines.

malocfp (IN)

User-defined memory allocation function. If *mode* is OCI_THREADED, this memory allocation routine must be thread safe.

ctxp (IN/OUT)

Context pointer for the user-defined memory allocation function.

size (IN)

Size of memory to be allocated by the user-defined memory allocation function

ralocfp (IN)

User-defined memory re-allocation function. If *mode* is OCI_THREADED, this memory allocation routine must be thread safe.

ctxp (IN/OUT)

Context pointer for the user-defined memory reallocation function.

memptr (IN/OUT)

Pointer to memory block

newsize (IN)

New size of memory to be allocated

mfreefp (IN)

User-defined memory free function. If *mode* is OCI_THREADED, this memory free routine must be thread safe.

ctxp (IN/OUT)

Context pointer for the user-defined memory free function.

memptr (IN/OUT)

Pointer to memory to be freed

Comments

This call initializes the OCI process environment.

OCIInitialize() must be invoked before any other OCI call.

This function provides the ability for the application to define its own memory management functions through callbacks. If the application has defined such functions (i.e., memory allocation, memory re-allocation, memory free), they should be registered using the callback parameters in this function.

These memory callbacks are optional. If the application passes NULL values for the memory callbacks in this function, the default process memory allocation mechanism is used.

Example

The following statement shows an example of how to call *OCIInitialize()* in both threaded and object mode, with no user-defined memory functions:

```
OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT, (dvoid *)0,  
             (dvoid * (*)(())) 0, (dvoid * (*)(())) 0, (void (*)(())) 0 );
```

Related Functions

OCIEnvInit()

OCILdaToSvcCtx()

Purpose

Converts a V7 Lda_Def to a V8 service context handle.

Syntax

```
sword OCILdaToSvcCtx ( OCISvcCtx  **svchpp,  
                      OCIError   *errhp,  
                      Lda_Def    *ldap );
```

Parameters

svchpp (IN/OUT)

The service context handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

ldap (IN/OUT)

The Oracle7 logon data area returned by *OCISvcCtxToLda()* from this service context.

Comments

Converts an Oracle7 **Lda_Def** to an Oracle8 service context handle. The action of this call can be reversed by passing the resulting service context handle to the *OCISvcCtxToLda()* function.

If the Service context has been converted to an **Lda_Def**, only Oracle7 calls may be used. It is illegal to make Oracle8 OCI calls without first resetting the **Lda_Def** to a service context.

The OCI_ATTR_IN_V8_MODE attribute of the server handle or service context handle enables an application to determine whether the application is currently in Oracle7 mode or Oracle8 mode. See Appendix B, “Handle and Descriptor Attributes”, for more information.

Related Functions

OCISvcCtxToLda()

OCILobAppend()

Purpose

Appends a LOB value at the end of another LOB as specified.

Syntax

```
sword OCILobAppend ( OCISvcCtx      *svchp,
                    OCIError        *errhp,
                    OCILobLocator    *dst_locp,
                    OCILobLocator    *src_locp );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

dst_locp (IN/OUT)

An internal LOB locator uniquely referencing the destination LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

src_locp (IN)

An internal LOB locator uniquely referencing the source LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

Comments

Appends a LOB value at the end of another LOB as specified. The data is copied from the source to the destination at the end of the destination. The source and destination LOBs must already exist. The destination LOB is extended to accommodate the newly written data.

It is an error to extend the destination LOB beyond the maximum length allowed (i.e., 4 gigabytes) or to try to copy from a NULL LOB.

Both the source and the destination LOB locators must be of the same type (i.e., they must both be BLOBs or both be CLOBs). LOB buffering must not be enabled for either type of locator.

This function does not accept a FILE locator as the source or the destination.

Related Functions

OCILobTrim(), OCILobWrite(), OCILobCopy(), OCIErrorGet()

OCILobAssign()

Purpose

Assigns one LOB/FILE locator to another.

Syntax

```
sword OCILobAssign ( OCIEnv          *envhp,  
                    OCIError        *errhp,  
                    CONST OCILobLocator *src_locp,  
                    OCILobLocator    **dst_locpp );
```

Parameters

envhp (IN/OUT)

OCI environment handle.

errhp (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCLErrorGet()*.

src_locp (IN)

LOB/FILE locator to copy from.

dst_locpp (IN/OUT)

LOB/FILE locator to copy to. The caller must have allocated space for the destination locator by calling *OCIDescriptorAlloc()*.

Comments

Assign *source* locator to *destination* locator. After the assignment, both locators refer to the same LOB value. For internal LOBs, the source locator's LOB value gets copied to the *destination* locator's LOB value only when the *destination* locator gets stored in the table. Therefore, issuing a flush of the object containing the *destination* locator will copy the LOB value.

For FILEs, only the locator that refers to the file is copied to the table. The OS file itself is not copied.

It is an error to assign a FILE locator to an internal LOB locator, and vice versa.

If the source locator is for an internal LOB that was enabled for buffering, and the source locator has been used to modify the LOB data through the LOB buffering subsystem, and the buffers have not been flushed since the write, then the source locator may not be assigned to the destination locator. This is because only one locator per LOB may modify the LOB data through the LOB buffering subsystem.

The value of the input destination locator must either be NULL, or it must have already been allocated with a call to *OCIDescriptorAlloc()*. For example, assume the following declarations:

```
OCILobLocator      *source_loc = (OCILobLocator *) 0;
OCILobLocator      *dest_loc  = (OCILobLocator *) 0;
```

An application could allocate the *source_loc* locator as follows:

```
if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &source_loc,
    (ub4) OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    handle_error;
```

Assume that it then selects a LOB from a table into the *source_loc* in order to initialize it. The application could then do one of the following to assign the value of *source_loc* to *dest_loc*:

1. Pass in NULL for the value of the destination locator and let *OCILobAssign()* allocate space for *dest_loc* and copy the source into it:

```
if (OCILobAssign(envhp, errhp, source_loc, &dest_loc))
    handle_error;
```

2. Allocate *dest_loc*, and pass the preallocated destination locator to *OCILobAssign()*:

```
if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &dest_loc,
    (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    handle_error;
if (OCILobAssign(envhp, errhp, source_loc, &dest_loc))
    handle_error;
```

Related Functions

OCIErrorGet(), *OCILobIsEqual()*, *OCILobLocatorIsInit()*, *OCILobEnableBuffering()*

OCILobCharSetForm()

Purpose

Gets the LOB locator's character set form, if any.

Syntax

```
sword OCILobCharSetForm ( OCIEnv          *envhp,  
                          OCIError        *errhp,  
                          CONST OCILobLocator *locp,  
                          ub1              *csfrm );
```

Parameters

envhp (IN/OUT)

OCI environment handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns `OCI_ERROR`. Diagnostic information can be obtained by calling *OCIErrorGet()*.

locp (IN)

LOB locator for which to get the character set form.

csfrm (OUT)

Character set form of the input LOB locator. If the input *locator* is for a BLOB or a BFILE, *csfrm* is set to 0 since there is no concept of a character set for binary LOBs/FILES. The caller must allocate space for the *csfrm* **ub1**.

Comments

Returns the character set form of the input LOB locator in the *csfrm* output parameter. This function makes sense only for character LOBs (i.e., CLOBs and NCLOBs).

Related Functions

OCIErrorGet(), *OCILobCharSetId()*, *OCILobLocatorIsInit()*

OCILobCharSetId()

Purpose

Gets the LOB locator's character set ID, if any.

Syntax

```

sword OCILobCharSetId ( OCIEEnv           *envhp,
                        OCIError          *errhp,
                        CONST OCILobLocator *locp,
                        ub2                *csid );

```

Parameters

envhp (IN/OUT)

OCI environment handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

locp (IN)

LOB locator for which to get the character set ID.

csid (OUT)

Character set ID of the input LOB locator. If the input *locator* is for a BLOB or a BFILE, *csid* is set to 0 since there is no concept of a character set for binary LOBs/FILES. The caller must allocate space for the *csid* **ub2**.

Comments

Returns the character set ID of the input LOB locator in the *csid* output parameter.

This function makes sense only for character LOBs (i.e., CLOBs, NCLOBs).

Related Functions

OCIErrorGet(), *OCILobCharSetForm()*, *OCILobLocatorIsInit()*

OCILobCopy()

Purpose

Copies all or a portion of a LOB value into another LOB value

Syntax

```
sword OCILobCopy ( OCISvcCtx      *svchp,  
                   OCIError       *errhp,  
                   OCILobLocator  *dst_locp,  
                   OCILobLocator  *src_locp,  
                   ub4            amount,  
                   ub4            dst_offset,  
                   ub4            src_offset );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

dst_locp (IN/OUT)

An internal LOB locator uniquely referencing the destination LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

src_locp (IN)

An internal LOB locator uniquely referencing the source LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

amount (IN)

The maximum number of characters or bytes, as appropriate, to be copied from the source LOB to the destination LOB.

dst_offset (IN)

This is the absolute offset for the destination LOB. For character LOBs it is the number of characters from the beginning of the LOB at which to begin writing. For

binary LOBs it is the number of bytes from the beginning of the LOB from which to begin writing. The offset starts at 1.

src_offset (IN)

This is the absolute offset for the source LOB. For character LOBs it is the number of characters from the beginning of the LOB, for binary LOBs it is the number of bytes. Starts at 1.

Comments

Copies all or a portion of an internal LOB value into another internal LOB as specified. The data is copied from the source to the destination. The source (*src_locp*) and the destination (*dst_locp*) LOBs must already exist.

If the data already exists at the destination's start position, it is overwritten with the source data. If the destination's start position is beyond the end of the current data, zero-byte fillers (for BLOBs) or spaces (for CLOBs) are written into the destination LOB from the end of the current data to the beginning of the newly written data from the source. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB. It is an error to extend the destination LOB beyond the maximum length allowed (i.e., 4 gigabytes) or to try to copy from a NULL LOB.

Both the source and the destination LOB locators must be of the same type (i.e., they must both be BLOBs or both be CLOBs). LOB buffering must not be enabled for either locator.

This function does not accept a FILE locator as the source or the destination.

The *amount* parameter indicates the maximum amount to copy. If the end of the source LOB is reached before the specified amount is copied, the operation terminates without error. This makes it possible to copy from a starting offset to the end of the LOB without first needing to determine the length of the LOB.

Note: You can call *OCILobGetLength()* to determine the length of the source LOB.

Related Functions

OCIErrorGet(), *OCILobAppend()*, *OCILobTrim()*, *OCILobWrite()*

OCILobDisableBuffering()

Purpose

Disable LOB buffering for the input locator.

Syntax

```
sword OCILobDisableBuffering ( OCISvcCtx      *svchp,  
                               OCIError       *errhp,  
                               OCILobLocator  *locp );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

locp (IN/OUT)

An internal LOB locator uniquely referencing the LOB.

Comments

Disables LOB buffering for the input internal LOB locator. The next time data is read from or written to the LOB through the input locator, the LOB buffering subsystem is *not* used. Note that this call does *not* implicitly flush the changes made in the buffering subsystem. The user must explicitly call *OCILobFlushBuffer()* to do this.

This function does not accept a FILE locator.

Related Functions

OCILobEnableBuffering(), *OCIErrorGet()*, *OCILobFlushBuffer()*

OCILOBEnableBuffering()

Purpose

Enable LOB buffering for the input locator.

Syntax

```

sword OCILOBEnableBuffering ( OCISvcCtx      *svchp,
                              OCIError       *errhp,
                              OCILOBLocator  *locp );

```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

locp (IN/OUT)

An internal LOB locator uniquely referencing the LOB.

Comments

Enables LOB buffering for the input internal LOB locator. The next time data is read from or written to the LOB through the input locator, the LOB buffering subsystem is used.

Once LOB buffering is enabled for a locator, if that locator is passed to one of the following routines, an error is returned: *OCILOBCopy()*, *OCILOBAppend()*, *OCILOBErase()*, *OCILOBGetLength()*, *OCILOBTrim()*, or *OCILOBLoadFromFile()*.

This function does not accept a FILE locator.

Related Functions

OCILOBDisableBuffering(), *OCIErrorGet()*, *OCILOBWrite()*, *OCILOBRead()*, *OCILOBFlushBuffer()*

OCILobErase()

Purpose

Erases a specified portion of the LOB data starting at a specified offset.

Syntax

```
sword OCILobErase ( OCISvcCtx      *svchp,  
                    OCIError       *errhp,  
                    OCILobLocator  *locp,  
                    ub4            *amount,  
                    ub4            offset );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

locp (IN/OUT)

An internal LOB locator that uniquely references the LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

amount (IN/OUT)

On IN, the number of characters/bytes to erase. On OUT, the actual number of characters/bytes erased.

offset (IN)

Absolute offset from the beginning of the LOB value from which to start erasing data. Starts at 1.

Comments

Erases a specified portion of the internal LOB data starting at a specified offset. The actual number of characters/bytes erased is returned. The actual number of characters/bytes and the requested number of characters/bytes will differ if the

end of the LOB value is reached before erasing the requested number of characters/bytes.

Note: For BLOBs, erasing means that zero-byte fillers overwrite the existing LOB value. For CLOBs, erasing means that spaces overwrite the existing LOB value.

If the LOB is NULL, this routine will indicate that 0 characters/bytes were erased.

This function is valid only for internal LOBs; FILEs are not allowed.

Related Functions

OCIErrorGet(), OCILobRead(), OCILobWrite()

OCILobFileClose()

Purpose

Closes a previously opened FILE.

Syntax

```
sword OCILobFileClose ( OCISvcCtx          *svchp,  
                        OCIError          *errhp,  
                        OCILobLocator      *filep );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

filep (IN/OUT)

A pointer to a FILE locator that refers to the FILE to be closed.

Comments

Closes a previously opened FILE. It is an error if this function is called for an internal LOB. No error is returned if the FILE exists but is not opened.

This function is only meaningful the first time it is called for a particular FILE locator. Subsequent calls to this function using the same FILE locator have no effect.

See Also: For more information about FILES, refer to the description of BFILES in the *Oracle8 Application Developer's Guide*.

Related Functions

OCIErrorGet(), *OCILobFileCloseAll()*, *OCILobFileExists()*

OCILobFileCloseAll()

Purpose

Closes all open FILES on a given service context.

Syntax

```
sword OCILobFileCloseAll ( OCISvcCtx   *svchp,  
                           OCIError    *errhp );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

Comments

Closes all open FILES on a given service context.

It is an error to call this function for an internal LOB.

See Also: For more information about FILES, refer to the description of BFILES in the *Oracle8 Application Developer's Guide*.

Related Functions

OCILobFileClose(), *OCIErrorGet()*, *OCILobFileExists()*, *OCILobFileIsOpen()*

OCILobFileExists()

Purpose

Tests to see if the FILE exists on the server's OS

Syntax

```
sword OCILobFileExists ( OCISvcCtx      *svchp,
                        OCIError        *errhp,
                        OCILobLocator   *filep,
                        boolean          *flag );
```

Parameters

svchp (IN)

The OCI service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

filep (IN)

Pointer to the FILE locator that refers to the file.

flag (OUT)

Returns TRUE if the FILE exists on the server; FALSE if it does not.

Comments

Checks to see if the FILE exists on the server's file system.

It is an error to call this function for an internal LOB.

See Also: For more information about FILES, refer to the description of BFILES in the *Oracle8 Application Developer's Guide*.

Related Functions

OCIErrorGet(), *OCILobFileClose()*, *OCILobFileCloseAll()*, *OCILobFileIsOpen()*

OCILobFileGetName()

Purpose

Gets the FILE locator's directory alias and file name.

Syntax

```

sword OCILobFileGetName ( OCIEnv                *envhp,
                          OCIError              *errhp,
                          CONST OCILobLocator   *filep,
                          text                  *dir_alias,
                          ub2                   *d_length,
                          text                  *filename,
                          ub2                   *f_length );

```

Parameters

envhp (IN/OUT)

OCI environment handle.

errhp (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

filep (IN)

FILE locator for which to get the directory alias and file name.

dir_alias (OUT)

Buffer into which the directory alias name is placed. The caller must allocate enough space for the directory alias name. The maximum length for the directory alias is 30 bytes.

d_length (IN/OUT)

Serves the following purposes

- IN: length of the input *dir_alias* string
- OUT: length of the returned *dir_alias* string

filename (OUT)

Buffer into which the file name is placed. The caller must allocate enough space for the file name. The maximum length for the file name is 255 bytes.

f_length (IN/OUT)

Serves the following purposes

- IN: length of the input *filename* buffer
- OUT: length of the returned *filename* string

Comments

Returns the directory alias and file name associated with this FILE locator.

It is an error to call this function for an internal LOB.

See Also: For more information about FILES, refer to the description of BFILES in the *Oracle8 Application Developer's Guide*.

Related Functions

OCILobFileSetName(), *OCIErrorGet()*

OCILobFileIsOpen()

Purpose

Tests to see if the FILE is open

Syntax

```
sword OCILobFileIsOpen ( OCISvcCtx      *svchp,  
                          OCIError      *errhp,  
                          OCILobLocator *filep,  
                          boolean       *flag );
```

Parameters

svchp (IN)

The OCI service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

filep (IN)

Pointer to the FILE locator being examined.

flag (OUT)

Returns TRUE if the FILE was opened using this particular locator; FALSE if it was not.

Comments

Checks to see if a file on the server was opened with the *filep* FILE locator.

It is an error to call this function for an internal LOB.

If the input FILE locator was never passed to the *OCILobFileOpen()* command, the file is considered not to be opened by this locator. However, a different locator may have the file open. Openness is associated with a particular locator.

See Also: For more information about FILES, refer to the description of BFILES in the *Oracle8 Application Developer's Guide*.

Related Functions

OCIErrorGet(), OCILobFileClose(), OCILobFileCloseAll()

OCILobFileOpen()

Purpose

Opens a FILE for read-only access.

Syntax

```

sword OCILobFileOpen ( OCISvcCtx          *svchp,
                       OCIError          *errhp,
                       OCILobLocator     *filep,
                       ubl                mode );

```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

filep (IN/OUT)

The FILE to open. It is an error if the locator does not refer to a FILE.

mode (IN)

Mode in which to open the file. The only valid mode is OCI_FILE_READONLY.

Comments

Opens a FILE. The FILE can be opened for read-only access. FILES may not be written through Oracle. It is an error to call this function for an internal LOB.

This function is only meaningful the first time it is called for a particular FILE locator. Subsequent calls to this function using the same FILE locator have no effect.

See Also: For more information about FILES, refer to the description of BFILES in the *Oracle8 Application Developer's Guide*.

Related Functions

OCILobFileClose(), *OCIErrorGet()*, *OCILobFileIsOpen()*, *OCILobFileNameSet()*

OCILobFileSetName()

Purpose

Sets the directory alias and file name in the FILE locator.

Syntax

```
sword OCILobFileSetName ( OCIEnv          *envhp,  
                           OCIError        *errhp,  
                           OCILobLocator   **filepp,  
                           CONST text      *dir_alias,  
                           ub2             d_length,  
                           CONST text      *filename,  
                           ub2             f_length );
```

Parameters

envhp (IN/OUT)

OCI environment handle.

errhp (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

filepp (IN/OUT)

FILE locator for which to set the directory alias and file name.

dir_alias (IN)

Buffer that contains the directory alias name to set in the FILE locator.

d_length (IN)

Length of the input *dir_alias* parameter.

filename (IN)

Buffer that contains the file name to set in the FILE locator.

f_length (IN)

Length of the input *filename* parameter.

Comments

Sets the directory alias and file name in the FILE locator.

It is an error to call this function for an internal LOB.

See Also: For more information about FILEs, refer to the description of BFILEs in the *Oracle8 Application Developer's Guide*.

Related Functions

OCILobFileGetName(), OCIErrorGet()

OCILobFlushBuffer()

Purpose

Flush/write all buffers for this lob to the server.

Syntax

```
sword OCILobFlushBuffer ( OCISvcCtx      *svchp,  
                          OCIError      *errhp,  
                          OCILobLocator *locp  
                          ub4           flag );
```

Parameters

svchp (IN/OUT)

The service context handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

locp (IN/OUT)

An internal locator uniquely referencing the LOB.

flag (IN)

When set to `OCI_LOB_BUFFER_FREE`, the buffer resources for the LOB are freed after the flush. See comments below.

Comments

Flushes to the server, changes made to the buffering subsystem that are associated with the LOB referenced by the input locator. This routine will actually write the data in the buffer to the LOB in the database. LOB buffering must have already been enabled for the input LOB locator.

The flush operation, by default, does not free the buffer resources for reallocation to another buffered LOB operation. However, if you want to free the buffer explicitly, you can set the flag parameter to `OCI_LOB_BUFFER_FREE`.

The effects of freeing the buffer are mostly transparent to the user, except that the next access to the same range in the LOB involves a round-trip to the server, and

also the cost of acquiring buffer resources and initializing it with the data read from the LOB. This option is intended for the following situations:

- If the client environment has low on-board memory.
- If the client application intends to read the buffer value after the flush and knows in advance that the current value in the buffer is the desired value. In this case there is no need to reread the data from the server.

Related Functions

OCILobEnableBuffering(), *OCIErrorGet()*, *OCILobWrite()*, *OCILobRead()*,
OCILobDisableBuffering()

OCILobGetLength()

Purpose

Gets the length of a LOB/FILE.

Syntax

```
sword OCILobGetLength ( OCISvcCtx      *svchp,
                        OCIError       *errhp,
                        OCILobLocator  *locp,
                        ub4             *lenp );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

locp (IN)

A LOB/FILE locator that uniquely references the LOB/FILE. For internal LOBs, this locator must be a locator that was obtained from the server specified by *svchp*. For FILES, the locator can be set via *OCILobFileSetName()*, via a SELECT statement, or via *OCIObjectPin*.

lenp (OUT)

On output, it is the length of the LOB/FILE if the LOB/FILE is not NULL for character LOBs it is the number of characters, for binary LOBs and BFILES it is the number of bytes in the LOB/FILE.

Comments

Gets the length of a LOB/FILE. If the LOB/FILE is NULL, the length is undefined.

The length of a FILE includes the EOF, if it exists.

The length is expressed in terms of bytes for BLOBs and BFILES, and in terms of characters for CLOBs. The length of an empty internal LOB is zero.

Note: Any zero-byte or space fillers in the LOB written by previous calls to *OCILobErase()* or *OCILobWrite()* are also included in the length count.

Related Functions

OCIErrorGet(), *OCILobFileSetName()*, *OCILobRead()*, *OCILobWrite()*, *OCILobCopy()*,
OCILobAppend(), *OCILobLoadFromFile()*

OCILobIsEqual()

Purpose

Compares two LOB/FILE locators for equality.

Syntax

```
sword OCILobIsEqual ( OCIEnv          *envhp,  
                      CONST OCILobLocator *x,  
                      CONST OCILobLocator *y,  
                      boolean           *is_equal );
```

Parameters

envhp (IN)

The OCI environment handle.

x (IN)

LOB locator to compare.

y (IN)

LOB locator to compare.

is_equal (OUT)

TRUE, if the LOB locators are equal; FALSE if they are not.

Comments

Compares the given LOB/FILE locators for equality. Two LOB/FILE locators are equal if and only if they both refer to the same LOB/FILE value.

Two NULL locators are considered *not* equal by this function.

Related Functions

OCILobAssign(), *OCILobLocatorIsInit()*

OCILobLoadFromFile()

Purpose

Load/copy all or a portion of the file into an internal LOB.

Syntax

```

sword OCILobLoadFromFile ( OCISvcCtx      *svchp,
                           OCIError       *errhp,
                           OCILobLocator  *dst_locp,
                           OCILobLocator  *src_locp,
                           ub4            amount,
                           ub4            dst_offset,
                           ub4            src_offset );

```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

dst_locp (IN/OUT)

A locator uniquely referencing the destination internal LOB which may be of type BLOB, CLOB, or NCLOB.

src_locp (IN/OUT)

A locator uniquely referencing the source FILE.

amount (IN)

The maximum number of bytes to be loaded.

dst_offset (IN)

This is the absolute offset for the destination LOB. For character LOBs it is the number of characters from the beginning of the LOB at which to begin writing. For binary LOBs it is the number of bytes from the beginning of the LOB from which to begin reading. The offset starts at 1.

src_offset (IN)

This is the absolute offset for the source FILE. It is the number of bytes from the beginning of the FILE. The offset starts at 1.

Comments

Loads/copies a portion or all of a FILE value into an internal LOB as specified. The data is copied from the source FILE to the destination internal LOB (BLOB/CLOB). No character set conversions are performed when copying the FILE data to a CLOB/NCLOB. Therefore, the FILE data must already be in the same character set as the CLOB/NCLOB in the database. No error checking is performed to verify this.

The source (*src_locp*) and the destination (*dst_locp*) LOBs must already exist. If the data already exists at the destination's start position, it is overwritten with the source data. If the destination's start position is beyond the end of the current data, zero-byte fillers (for BLOBs) or spaces (for CLOBs) are written into the destination LOB from the end of the data to the beginning of the newly written data from the source. The destination LOB is extended to accommodate the newly written data if it extends beyond the current length of the destination LOB.

It is an error to extend the destination LOB beyond the maximum length allowed (4 gigabytes) or to try to copy from a NULL FILE.

The *amount* parameter indicates the maximum amount to load. If the end of the source FILE is reached before the specified amount is loaded, the operation terminates without error. This makes it possible to load from a starting offset to the end of the FILE without first needing to determine the length of the file.

Related Functions

OCIErrorGet(), *OCILobAppend()*, *OCILobWrite()*, *OCILobTrim()*, *OCILobCopy()*,
OCILobGetLength()

OCILobLocatorIsInit()

Purpose

Tests to see if a given LOB/FILE locator is initialized.

Syntax

```

sword OCILobLocatorIsInit ( OCIEnv           *envhp,
                           OCIError        *errhp,
                           CONST OCILobLocator *locp,
                           boolean         *is_initialized );

```

Parameters

envhp (IN/OUT)

OCI environment handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

locp (IN)

The LOB/FILE locator being tested

is_initialized (OUT)

Returns TRUE if the given LOB/FILE locator is initialized; FALSE if it is not.

Comments

Tests to see if a given LOB/FILE locator is initialized.

Internal LOB locators can be initialized by one of the following methods:

- SELECTing a non-NULL LOB into the locator,
- pinning an object that contains a non-NULL LOB attribute via *OCIObjectPin()*
- setting the locator to empty via *OCIAttrSet()* (see “LOB Locator Attributes” on page B-25 for more information.)

FILE locators can be initialized by one of the following methods:

- SELECTing a non-NULL FILE into the locator

- pinning an object that contains a non-NULL FILE attribute via *OCIObjectPin()*
- calling *OCILobFileSetName()*

Related Functions

OCIErrorGet(), *OCILobIsEqual()*

OCILobRead()

Purpose

Reads a portion of a LOB/FILE, as specified by the call, into a buffer.

Syntax

```

sword OCILobRead ( OCISvcCtx          *svchp,
                  OCIError            *errhp,
                  OCILobLocator       *locp,
                  ub4                  *amtp,
                  ub4                  offset,
                  dvoid               *bufp,
                  ub4                  bufl,
                  dvoid               *ctxp,
                  OCICallbackLobRead (cbfp)
                  ( dvoid               *ctxp,
                    CONST dvoid         *bufp,
                    ub4                  len,
                    ub1                  piece )
                  ub2                  csid,
                  ub1                  csfrm );

```

Parameters

svchp (IN/OUT)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

locp (IN)

A LOB/FILE locator that uniquely references the LOB/FILE. This locator must be a locator that was obtained from the server specified by *svchp*.

amtp (IN/OUT)

On input, the number of characters (for CLOBs or NCLOBs) or bytes (for BLOBs and BFILES) to be read. On output, the actual number of bytes or characters read.

If the amount of bytes to be read is larger than the buffer length it is assumed that the LOB is being read in a streamed mode from the input offset until the end of the LOB, or until the specified number of bytes have been read, *whichever comes first*. On input if this value is 0, then the data shall be read in streamed mode from the input offset until the end of the LOB.

The streamed mode (implemented with either polling or callbacks) reads the LOB value sequentially from the input offset.

If the data is read in pieces, **amtp* always contains the length of the piece just read.

If a callback function is defined, then this callback function will be invoked each time *bufl* bytes are read off the pipe. Each piece will be written into *bufp*.

If the callback function is not defined, then the OCI_NEED_DATA error code will be returned. The application must call *OCILobRead()* over and over again to read more pieces of the LOB until the OCI_NEED_DATA error code is not returned. The buffer pointer and the length can be different in each call if the pieces are being read into different sizes and locations.

offset (IN)

On input, this is the absolute offset from the beginning of the LOB value. For character LOBs (CLOBs, NCLOBs) it is the number of characters from the beginning of the LOB, for binary LOBs/FILES it is the number of bytes. The first position is 1.

bufp (IN/OUT)

The pointer to a buffer into which the piece will be read. The length of the allocated memory is assumed to be *bufl*.

bufl (IN)

The length of the buffer in octets. This value will differ from the *amtp* value in the following cases:

- For CLOBs and for NCLOBs (*csfrm*=SQLCS_NCHAR), the *amtp* parameter is specified in terms of characters, while the *bufl* parameter is specified in terms of bytes.
- The user may allocate a single big buffer to be reused by several LOB read/write calls. In this case *bufl* may be larger than the *amtp* requested.

ctxp (IN)

The context for the callback function. Can be NULL.

cbfp (IN)

A callback that may be registered to be called for each piece. If this is NULL, then OCI_NEED_DATA will be returned for each piece.

The callback function must return OCI_CONTINUE for the read to continue. If any other error code is returned, the LOB read is aborted.

ctxp (IN)

The context for the callback function. Can be NULL.

bufp (IN/OUT)

A buffer pointer for the piece.

len (IN)

The length of the current piece in *bufp*.

piece (IN)

Which piece: OCI_FIRST_PIECE, OCI_NEXT_PIECE or OCI_LAST_PIECE.

csid (IN)

The character set ID of the buffer data.

csfrm (IN)

The character set form of the buffer data.

Comments

Reads a portion of a LOB/FILE as specified by the call into a buffer. It is an error to try to read from a NULL LOB/FILE.

Note: When reading or writing LOBs, the character set form (*csfrm*) specified should match the form of the locator itself.

For FILES, the OS file must already exist on the server, and it must have been opened via *OCILobFileOpen()* using the input locator. Oracle must have permission to read the OS file, and the user must have read permission on the directory object.

When using the polling mode for *OCILobRead()*, the first call needs to specify values for *offset* and *amtp*, but on subsequent polling calls to *OCILobRead()*, the user need not specify these values.

See Also: For more information about FILES, refer to the description of BFILES in the *Oracle8 Application Developer's Guide*.

For a code sample showing the use of LOB reads and writes, refer to "Example 5, CLOB/BLOB Operations" on page D-76.

For general information about piecewise OCI operations, refer to “Run Time Data Allocation and Piecewise Operations” on page 7-16.

Related Functions

OCIErrorGet(), OCILobWrite(), OCILobFileSetName()

OCILobTrim()

Purpose

Trims/truncates the LOB value to a shorter length.

Syntax

```
sword OCILobTrim ( OCISvcCtx      *svchp,  
                   OCIError       *errhp,  
                   OCILobLocator  *locp,  
                   ub4            newlen );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Diagnostic information can be obtained by calling *OCIErrorGet()*.

locp (IN/OUT)

An internal LOB locator that uniquely references the LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

newlen (IN)

The new length of the LOB value, which must be less than or equal to the current length.

Comments

Trims the LOB data to a specified shorter length.

The function returns an error if *newlen* is greater than the current LOB length.

This function is valid only for internal LOBs. FILES are not allowed.

Related Functions

OCIErrorGet(), *OCILobErase()*, *OCILobAppend()*, *OCILobCopy()*, *OCILobWrite()*

OCILobWrite()

Purpose

Writes a buffer into a LOB

Syntax

```

sword OCILobWrite ( OCISvcCtx      *svchp,
                    OCIError       *errhp,
                    OCILobLocator  *locp,
                    ub4            *amtp,
                    ub4            offset,
                    dvoid          *bufp,
                    ub4            buflen,
                    ub1            piece,
                    dvoid          *ctxp,
                    OCICallbackLobWrite (cbfp)
                    (/*_
                     dvoid      *ctxp,
                     dvoid      *bufp,
                     ub4        *lenp,
                     ub1        *piecep */)
                    ub2          csid,
                    ub1          csfrm );

```

Parameters

svchp (IN/OUT)

The service context handle.

errhp (IN/OUT)

Error handle. The OCI error handle. If there is an error, it is recorded in *err* and this function returns `OCI_ERROR`. Diagnostic information can be obtained by calling *OCIErrorGet()*.

locp (IN/OUT)

An internal LOB locator that uniquely references the LOB. This locator must be a locator that was obtained from the server specified by *svchp*.

amtp (IN/OUT)

On input, takes the number of characters or bytes to be written. On output, returns the actual number of bytes or characters written.

If the amount is specified on input, and the data is written in pieces, **amtp* will contain the total length of the pieces written at the end of the call (last piece written) and is undefined in between. (Note it is different from the piecewise read case). An error is returned if that amount is not sent to the server.

If *amtp* is zero, then streaming mode is assumed, and data is written until the user specifies OCI_LAST_PIECE.

offset (IN)

On input, it is the absolute offset from the beginning of the LOB value. For character LOBs it is the number of characters from the beginning of the LOB, for binary LOBs it is the number of bytes. The first position is 1.

bufp (IN)

The pointer to a buffer from which the piece will be written. The length of the allocated memory is assumed to be the value passed in *buflen*. Even if the data is being written in pieces using the polling method, *bufp* must contain the first piece of the LOB when this call is invoked. If a callback is provided, *bufp* must not be used to provide data or an error will result.

buflen (IN)

the length of the buffer in bytes. This value will differ from the *amtp* value in the following cases:

- For CLOBs and NCLOBs, the *amtp* parameter is specified in terms of characters, which the *bufl* parameter is specified in terms of bytes.
- The user may allocate a single big buffer to be reused by several LOB read/write calls. In this case *bufl* may be larger than the *amtp* requested.

Note: This parameter assumes an 8-bit byte. If your platform uses a longer byte, the value of *buflen* must be adjusted accordingly.

piece (IN)

Which piece of the buffer is being written. The default value for this parameter is OCI_ONE_PIECE, indicating the buffer will be written in a single piece.

The following other values are also possible for piecewise or callback mode: OCI_FIRST_PIECE, OCI_NEXT_PIECE and OCI_LAST_PIECE.

ctxp (IN)

The context for the callback function. Can be NULL.

cbfp (IN)

A callback that may be registered to be called for each piece in a piecewise write. If this is NULL, the standard polling method will be used.

The callback function must return OCI_CONTINUE for the write to continue. If any other error code is returned, the LOB write is aborted. The callback takes the following parameters:

ctxp (IN)

The context for the callback function. Can be NULL.

bufp (IN/OUT)

A buffer pointer for the piece.

lenp (IN/OUT)

The length of the buffer in octets (IN),
and the length of current piece in *bufp* in octets (OUT).

piecep (OUT)

Which piece: OCI_NEXT_PIECE or OCI_LAST_PIECE.

csid (IN)

The LOB character set ID of the buffer data.

csfrm (IN)

The LOB character set form of the buffer data.

Comments

Writes a buffer into a LOB as specified. If LOB data already exists it is overwritten with the data stored in the buffer.

The buffer can be written to the LOB in a single piece with this call, or it can be provided piecewise using callbacks or a standard polling method.

Note: When reading or writing LOBs, the character set form (*csfrm*) specified should match the form of the locator itself.

When using the polling mode for *OCILobWrite()*, the first call needs to specify values for *offset* and *amtp*, but on subsequent polling calls to *OCILobWrite()*, the user need not specify these values.

If the value of the *piece* parameter is OCI_FIRST_PIECE, data may need to be provided through callbacks or polling.

If a callback function is defined in the *cbfp* parameter, then this callback function will be invoked to get the next piece after a piece is written to the pipe. Each piece will be written from *bufp*.

If no callback function is defined, then *OCILobWrite()* returns the OCI_NEED_DATA error code. The application must call *OCILobWrite()* again to write more pieces of the LOB. In this mode, the buffer pointer and the length can be different in each call if the pieces are of different sizes and from different locations.

A *piece* value of OCI_LAST_PIECE terminates the piecewise write, regardless of whether the polling or callback method is used.

If the amount of data passed to Oracle (through either input mechanism) is less than the amount specified by the *amtp* parameter, an ORA-22993 error is returned.

This function is valid for internal LOBs only. FILES are not allowed, since they are read-only.

See Also: For a code sample showing the use of LOB reads and writes, refer to “Example 5, CLOB/BLOB Operations” on page D-76.

For general information about piecewise OCI operations, refer to “Run Time Data Allocation and Piecewise Operations” on page 7-16.

Related Functions

OCIErrorGet(), *OCILobRead()*, *OCILobAppend()*, *OCILobCopy()*

OCILogoff()

Purpose

This function is used to terminate a connection and session created with *OCILogon()*.

Syntax

```
sword OCILogoff ( OCISvcCtx      *svchp  
                  OCIError      *errhp );
```

Parameters

svchp (IN)

The service context handle which was used in the call to *OCILogon()*.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

Comments

This call is used to terminate a session and connection which were created with *OCILogon()*. This call implicitly deallocates the server, user session, and service context handles.

Note: For more information on logging on and off in an application, refer to the section “Application Initialization, Connection, and Session Creation” on page 2-18.

Related Functions

OCILogon()

OCILogon()

Purpose

This function is used to create a simple logon session.

Syntax

```

sword OCILogon ( OCIEnv          *envhp,
                  OCIError        *errhp,
                  OCISvcCtx       **svchp,
                  CONST text      *username,
                  ub4             uname_len,
                  CONST text      *password,
                  ub4             passwd_len,
                  CONST text      *dbname,
                  ub4             dbname_len );

```

Parameters

envhp (IN)

The OCI environment handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

svchp (IN/OUT)

The service context pointer.

username (IN)

The username.

uname_len (IN)

The length of *username*.

password (IN)

The user's password.

passwd_len (IN)

The length of *password*.

dbname (IN)

The name of the database to connect to.

dbname_len (IN)

The length of *dbname*.

Comments

This function is used to create a simple logon session for an application.

Note: Users requiring more complex sessions (e.g., TP monitor applications) should refer to the section “Application Initialization, Connection, and Session Creation” on page 2-18.

This call allocates the error and service context handles which are passed to it.

This call also implicitly allocates server and user session handles associated with the session. These handles can be retrieved by calling *OCIAQEnq()* on the service context handle.

Related Functions

OCILogoff()

OCIParamGet()

Purpose

Returns a descriptor of a parameter specified by position in the describe handle or statement handle.

Syntax

```
sword OCIParamGet ( CONST dvoid      *hndlp,  
                   ub4              htype,  
                   OCIError        *errhp,  
                   dvoid            **parmdpp,  
                   ub4              pos );
```

Parameters

hndlp (IN)

A statement handle or describe handle. The *OCIParamGet()* function will return a parameter descriptor for this handle.

htype (IN)

the type of the handle passed in the *handle* parameter. Valid types are

- OCI_DTYPE_PARM, for a parameter descriptor
- OCI_HTYPE_COR, for a complex object retrieval handle
- OCI_HTYPE_STMT, for a statement handle

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

parmdpp (OUT)

A descriptor of the parameter at the position given in the *pos* parameter.

pos (IN)

Position number in the statement handle or describe handle. A parameter descriptor will be returned for this position.

Note: OCI_NO_DATA may be returned if there are no parameter descriptors for this position.

Comments

This call returns a descriptor of a parameter specified by position in the describe handle or statement handle. Parameter descriptors are always allocated internally by the OCI library. They are read-only.

OCI_NO_DATA may be returned if there are no parameter descriptors for this position.

See Appendix B, “Handle and Descriptor Attributes”, for more detailed information about parameter descriptor attributes.

Related Functions

OCIAQEnq(), OCIAttrSet(), OCIParamSet()

OCIParamSet()

Purpose

Used to set a complex object retrieval descriptor into a complex object retrieval handle.

Syntax

```
sword OCIParamSet ( dvoid          *hndlp,  
                    ub4            htype,  
                    OCIError      *errhp,  
                    CONST dvoid    *dscp,  
                    ub4            dtyp,  
                    ub4            pos );
```

Parameters

hndlp (IN/OUT)

Handle pointer.

htype (IN)

Handle type.

errhp (IN/OUT)

Error handle.

dscp (IN)

Complex object retrieval descriptor pointer.

dtyp (IN)

Descriptor type. The descriptor type for a COR descriptor is OCI_DTYPE_COMPLEXOBJECTCOMP.

pos (IN)

Position number.

Comments

This call sets a given complex object retrieval descriptor into a complex object retrieval handle.

The handle must have been previously allocated using *OCIHandleAlloc()*, and the descriptor must have been previously allocated using *OCIDescriptorAlloc()*. Attributes of the descriptor are set using *OCIAttrSet()*.

For more information about complex object retrieval, see “Complex Object Retrieval” on page 8-21.

Related Functions

OCIParamGet()

OCIPasswordChange()

Purpose

This call allows the password of an account to be changed.

Syntax

```
sword OCIPasswordChange ( OCISvcCtx      *svchp,  
                           OCIError      *errhp,  
                           CONST text    *user_name,  
                           ub4           usernm_len,  
                           CONST text    *opasswd,  
                           ub4           opasswd_len,  
                           CONST text    *npasswd,  
                           sb4           npasswd_len,  
                           ub4           mode );
```

Parameters

svchp (IN/OUT)

A handle to a service context. The service context handle must be initialized and have a server context handle associated with it.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

user_name (IN)

Specifies the user name. It points to a character string, whose length is specified in *usernm_len*. This parameter must be NULL if the service context has been initialized with an user session handle.

usernm_len (IN)

The length of the user name string specified in *user_name*. For a valid user name string, *usernm_len* must be non-zero.

opasswd (IN)

Specifies the user's old password. It points to a character string, whose length is specified in *opasswd_len*.

opasswd_len (IN)

The length of the old password string specified in *opasswd*. For a valid password string, *opasswd_len* must be non-zero.

npasswd (IN)

Specifies the user's new password. It points to a character string, whose length is specified in *npasswd_len* which must be non-zero for a valid password string. If the password complexity verification routine is specified in the user's profile to verify the new password's complexity, the new password must meet the complexity requirements of the verification function.

npasswd_len (IN)

Then length of the new password string specified in *npasswd*. *For a valid password string, npasswd_len must be non-zero.*

mode (IN)

Can be OCI_DEFAULT and/or OCI_AUTH. If set to OCI_AUTH, the following happens:

- If a user session context is not created, this call creates the user session context and changes the password. At the end of the call, the user session context is not cleared. Hence the user remains logged in.
- If the user session context is already created, this call just changes the password and the flag has no effect on the session. Hence the user still remains logged in.

Comments

This call allows the password of an account to be changed. This call is similar to *OCISessionBegin()* with the following differences:

- If the user session is already established, it authenticates the account using the old password and then changes the password to the new password
- If the user session is not established, it establishes a user session and authenticates the account using the old password, then changes the password to the new password.

This call is useful when the password of an account is expired and *OCISessionBegin()* returns an error or warning which indicates that the password has expired.

Related Functions

OCISessionBegin()

OCIServerAttach()

Purpose

Creates an access path to a data source for OCI operations.

Syntax

```
sword OCIServerAttach ( OCIServer      *srvhp,  
                        OCIError       *errhp,  
                        CONST text      *dblink,  
                        sb4             dblink_len,  
                        ub4             mode );
```

Parameters

srvhp (IN/OUT)

An uninitialized server handle, which gets initialized by this call. Passing in an initialized server handle causes an error.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

dblink (IN)

Specifies the database (server) to use. This parameter points to a character string which specifies a connect string or a service point. If the connect string is NULL, then this call attaches to the default host. The length of *dblink* is specified in *dblink_len*. The *dblink* pointer may be freed by the caller on return.

dblink_len (IN)

The length of the string pointed to by *dblink*. For a valid connect string name or alias, *dblink_len* must be non-zero.

mode (IN)

Specifies the various modes of operation. For release 8.0, pass as OCI_DEFAULT. In this mode, calls made to the server on this server context are made in blocking mode.

Comments

This call is used to create an association between an OCI application and a particular server.

This call initializes a server context handle, which must have been previously allocated with a call to *OCIHandleAlloc()*.

The server context handle initialized by this call can be associated with a service context through a call to *OCIAttrSet()*. Once that association has been made, OCI operations can be performed against the server.

If an application is operating against multiple servers, multiple server context handles can be maintained. OCI operations are performed against whichever server context is currently associated with the service context.

Example

The following example demonstrates the use of *OCI`ServerAttach()`*. This code segment allocates the server handle, makes the attach call, allocates the service context handle, and then sets the server context into it.

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4)
    OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4)
    OCI_HTYPE_SVCCTX, 0, (dvoid **) &tmp);
/* set attribute server context in the service context */
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) srvhp,
    (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
```

Related Functions

OCI`ServerDetach()`

OCIServerDetach()

Purpose

Deletes an access to a data source for OCI operations.

Syntax

```
sword OCIServerDetach ( OCIServer   *srvhp,  
                        OCIError    *errhp,  
                        ub4          mode );
```

Parameters

srvhp (IN)

A handle to an initialized server context, which gets reset to uninitialized state. The handle is not de-allocated.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

mode (IN)

Specifies the various modes of operation. The only valid mode is OCI_DEFAULT for the default mode.

Comments

This call deletes an access to data source for OCI operations, which was established by a call to *OCIServerAttach()*.

Related Functions

OCIServerAttach()

OCIServerVersion()

Purpose

Returns the version string of the Oracle server.

Syntax

```
sword OCIServerVersion ( dvoid          *hndlp,
                        OCIError       *errhp,
                        text           *bufp,
                        ub4            bufksz,
                        ub1            hndltype );
```

Parameters

hndlp (IN)

The service context handle or the server context handle.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

bufp (IN)

The buffer in which the version information is returned.

bufksz (IN)

The length of the buffer.

hndltype (IN)

The type of handle passed to the function.

Comments

This call returns the version string of the Oracle server. For example, the following might be returned as the version string if your application is running against a 7.3.2 server:

```
Oracle7 Server Release 7.3.2.0.0 Production Release
PL/SQL Release 2.3.2.0.0 Production
CORE Version 3.5.2.0.0 Production
TNS for SEQUENT DYNIX/ptx: Version 2.3.2.0.0 Production
NLSRTL Version 3.2.2.0.0 Production
```

OCISessionBegin()

Purpose

Creates a user session and begins a user session for a given server.

Syntax

```
sword OCISessionBegin ( OCISvcCtx      *svchp,  
                        OCIError       *errhp,  
                        OCISession     *usrhp,  
                        ub4             credt,  
                        ub4             mode );
```

Parameters

svchp (IN)

A handle to a service context. There must be a valid server handle set in *svchp*.

errhp (IN)

An error handle to the retrieve diagnostic information.

usrhp (IN/OUT)

A handle to an user session context, which is initialized by this call.

credt (IN)

Specifies the type of credentials to use for establishing the user session. Valid values for *credt* are:

- **OCI_CRED_RDBMS** - authenticate using a database username and password pair as credentials. The attributes **OCI_ATTR_USERNAME** and **OCI_ATTR_PASSWORD** should be set on the user session context before this call.
- **OCI_CRED_EXT** - authenticate using external credentials. No username or password is provided.

mode (IN)

Specifies the various modes of operation. Valid modes are:

- **OCI_DEFAULT** - in this mode, the user session context returned may only ever be set with the same server context specified in *svchp*.

- `OCI_MIGRATE` - in this mode, the new user session context may be set in a service handle with a different server handle. This mode establishes the user session context. To create a migratable session, the service handle must already be set with a non-migratable user session. A migratable session must have a non-migratable parent session.
- `OCI_SYSDBA` - in this mode, the user is authenticated for SYSDBA access.
- `OCI_SYSOPER` - in this mode, the user is authenticated for SYSOPER access.
- `OCI_PRELIM_AUTH` - this mode may only be used with `OCI_SYSDBA` or `OCI_SYSOPER` to authenticate for certain administration tasks.

Comments

The *OCISessionBegin()* call is used to authenticate a user against the server set in the service context handle.

For Oracle8, *OCISessionBegin()* must be called for any given server handle before requests can be made against it. Also, *OCISessionBegin()* only supports authenticating the user for access to the Oracle server specified by the server handle in the service context. In other words, after *OCIServerAttach()* is called to initialize a server handle, *OCISessionBegin()* must be called to authenticate the user for that given server.

When *OCISessionBegin()* is called for the first time for a given server handle, the user session may not be created in migratable (`OCI_MIGRATE`) mode.

After *OCISessionBegin()* has been called for a server handle, the application may call *OCISessionBegin()* again to initialize another user session handle with different (or the same) credentials and different (or the same) operation modes. If an application wants to authenticate a user in `OCI_MIGRATE` mode, the service handle must already be associated with a non-migratable user handle. The user ID of that user handle becomes the ownership ID of the migratable user session. Every migratable session must have a non-migratable parent session.

If the `OCI_MIGRATE` mode is not specified, then the user session context can only ever be used with the same server handle set in *svchp*. If `OCI_MIGRATE` mode is specified, then the user authentication may be set with different server handles. However, the user session context may only be used with server handles which resolve to the same database instance. Security checking is done during session switching. A process or circuit is allowed to switch to a migratable session only if the ownership ID of the session matches the user ID of a non-migratable session currently connected to that same process or circuit, unless it is the creator of the session.

OCI_SYSDBA, OCI_SYSOPER, and OCI_PRELIM_AUTH may only be used with a primary user session context.

To provide credentials for a call to *OCISessionBegin()*, one of two methods are supported. The first is to provide a valid username and password pair for database authentication in the user session handle passed to *OCISessionBegin()*. This involves using *OCIAttrSet()* to set the OCI_ATTR_USERNAME and OCI_ATTR_PASSWORD attributes on the user session handle. Then *OCISessionBegin()* is called with OCI_CRED_RDBMS.

Note: When the user session handle is terminated using *OCISessionEnd()*, the username and password attributes remain unchanged and thus can be re-used in a future call to *OCISessionBegin()*. Otherwise, they must be reset to new values before the next *OCISessionBegin()* call.

The second type of credentials supported are external credentials. No attributes need to be set on the user session handle before calling *OCISessionBegin()*. The credential type is OCI_CRED_EXT. This is equivalent to the Oracle7 'connect /' syntax. If values have been set for OCI_ATTR_USERNAME and OCI_ATTR_PASSWORD, then these are ignored if OCI_CRED_EXT is used.

Example

The following example demonstrates the use of *OCISessionBegin()*. This code segment allocates the user session handle, sets the username and password attributes, calls *OCISessionBegin()*, and then sets the user session into the service context.

```
/* allocate a user session handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4)
    OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"jessica",
    (ub4)strlen("jessica"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"doogie",
    (ub4)strlen("doogie"), OCI_ATTR_PASSWORD, errhp);
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
    OCI_DEFAULT));
OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)usrhp,
    (ub4)0, OCI_ATTR_SESSION, errhp);
```

Related Functions

OCISessionEnd()

OCISessionEnd()

Purpose

Terminates a user session context created by *OCISessionBegin()*

Syntax

```
sword OCISessionEnd ( OCISvcCtx      *svchp,  
                      OCIError      *errhp,  
                      OCISession    *usrhp,  
                      ub4            mode );
```

Parameters

svchp (IN/OUT)

The service context handle. There must be a valid server handle and user session handle associated with *svchp*.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

usrhp (IN)

De-authenticate this user. If this parameter is passed as NULL, the user in the service context handle is de-authenticated.

mode (IN)

The only valid mode is OCI_DEFAULT.

Comments

The user security context associated with the service context is invalidated by this call. Storage for the user session context is not freed. The transaction specified by the service context is implicitly committed. The transaction handle, if explicitly allocated, may be freed if not being used.

Resources allocated on the server for this user are freed.

The user session handle may be reused in a new call to *OCISessionBegin()*.

Related Functions*OCISessionBegin()*

OCIStmtExecute()

Purpose

This call associates an application request with a server.

Syntax

```
sword OCIStmtExecute ( OCISvcCtx          *svchp,
                       OCIStmt           *stmtp,
                       OCIError          *errhp,
                       ub4                iters,
                       ub4                rowoff,
                       CONST OCISnapshot *snap_in,
                       OCISnapshot       *snap_out,
                       ub4                mode );
```

Parameters

svchp (IN/OUT)

Service context handle.

stmtp (IN/OUT)

An statement handle. It defines the statement and the associated data to be executed at the server. It is invalid to pass in a statement handle that has bind of data types only supported in release 8.0 when *svchp* points to an Oracle7 server.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

iters (IN)

The number of times this statement is executed for non-SELECT statements. For Select statements, if *iters* is non-zero, then defines must have been done for the statement handle. The execution fetches *iters* rows into these predefined buffers and prefetches more rows depending upon the prefetch row count. This function returns an error if *iters*=0 for non-SELECT statements.

rowoff (IN)

The starting index from which the data in an array bind is relevant for this multiple row execution.

snap_in (IN)

This parameter is optional. if supplied, must point to a snapshot descriptor of type OCI_DTYPE_SNAP. The contents of this descriptor must be obtained from the *snap_out* parameter of a previous call. The descriptor is ignored if the SQL is not a SELECT. This facility allows multiple service contexts to ORACLE to see the same consistent snapshot of the database's *committed* data. However, uncommitted data in one context is *not* visible to another context even using the same snapshot.

snap_out (OUT)

This parameter optional. if supplied, must point to a descriptor of type OCI_DTYPE_SNAP. This descriptor is filled in with an opaque representation which is the current ORACLE "system change number" suitable as a *snap_in* input to a subsequent call to *OCIStmtExecute()*. This descriptor should not be used longer than necessary in order to avoid "snapshot too old" errors.

mode (IN)

The modes are:

- OCI_DEFAULT - Calling *OCIStmtExecute()* in this mode executes the statement. It also implicitly returns describe information about the select-list.
- OCI_DESCRIBE_ONLY - This mode is for users who wish to describe a query prior to execution. Calling *OCIStmtExecute()* in this mode does not execute the statement, but it does return the select-list description. To maximize performance, it is recommended that applications execute the statement in default mode and use the implicit describe which accompanies the execution.
- OCI_COMMIT_ON_SUCCESS - When a statement is executed in this mode, the current transaction is committed after execution, provided that execution completes successfully.
- OCI_EXACT_FETCH - Used when the application knows in advance exactly how many rows it will be fetching. This mode turns prefetching off for Oracle8 mode, and requires that defines be done before the execute call. Using this mode cancels the cursor after the desired rows are fetched and may result in reduced server-side resource usage.

Comments

This function is used to execute a prepared SQL statement. Using an execute call, the application associates a request with a server.

If a SELECT statement is executed, the description of the select-list is available implicitly as a response. This description is buffered on the client side for describes,

fetches and define type conversions. Hence it is optimal to describe a select list only after an execute. See “Describing Select-List Items” on page 4-8 for more information.

Also for SELECT statements, some results are available implicitly. Rows will be received and buffered at the end of the execute. For queries with small row count, a prefetch causes memory to be released in the server if the end of fetch is reached, an optimization that may result in memory usage reduction. Set attribute call has been defined to set the number of rows to be prefetched per result set.

For SELECT statements, at the end of the execute, the statement handle implicitly maintains a reference to the service context on which it is executed. It is the user’s responsibility to maintain the integrity of the service context. The implicit reference is maintained until the statement handle is freed or the fetch is cancelled or an end of fetch condition is reached.

Note: If output variables are defined for a SELECT statement before a call to *OCIStmtExecute()*, the number of rows specified by *iters* will be fetched directly into the defined output buffers and additional rows equivalent to the prefetch count will be prefetched. If there are no additional rows, then the fetch is complete without calling *OCIStmtFetch()*.

Related Functions

OCIStmtPrepare()

OCIStmtFetch()

Purpose

Fetches rows from a query.

Syntax

```
sword OCIStmtFetch ( OCIStmt      *stmtp,  
                     OCIError    *errhp,  
                     ub4         nrows,  
                     ub2         orientation,  
                     ub4         mode );
```

Parameters

stmtp (IN)

A statement (application request) handle.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

nrows (IN)

Number of rows to be fetched from the current position.

orientation (IN)

For release 8.0, the only acceptable value is *OCI_FETCH_NEXT*, which is also the default value.

mode (IN)

For release 8.0, pass as *OCI_DEFAULT*.

Comments

The fetch call is a local call, if prefetched rows suffice. However, this is transparent to the application. If LOB columns are being read, LOB locators are fetched for subsequent LOB operations to be performed on these locators. Prefetching is turned off if LONG columns are involved.

This function can return *OCI_SUCCESS_WITH_INFO* if the data is truncated or EOF is reached.

Related Functions

OCIStmtExecute()

OCIStmtGetBindInfo()

Purpose

Gets the bind and indicator variable names.

Syntax

```
sword OCIStmtGetBindInfo ( OCIStmt      *stmtp,  
                           OCIError     *errhp,  
                           ub4          size,  
                           ub4          startloc,  
                           sb4          *found,  
                           text         *bvnp[],  
                           ub1          bvnl[],  
                           text         *invp[],  
                           ub1          inpl[],  
                           ub1          dupl[],  
                           OCIBind      *hndl[] );
```

Parameters

stmtp (IN)

The statement handle.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

size (IN)

The number of elements in each array.

startloc (IN)

Position of the bind variable at which to start getting bind information.

found (IN)

Abs(found) gives the total number of bind variables in the statement irrespective of the start position. Positive value if the number of bind variables returned is less than the size provided, otherwise negative.

bvnp (OUT)

Array of pointers to hold bind variable names.

bvnl (OUT)

Array to hold the length of the each *bvnp* element.

invp (OUT)

Array of pointers to hold indicator variable names.

inpl (OUT)

Array of pointers to hold the length of the each *invp* element.

dupl (OUT)

An array whose element value is 0 or 1 depending on whether the bind position is duplicate of another.

hdl (OUT)

An array which returns the bind handle if binds have been done for the bind position. No handle is returned for duplicates.

Comments

This call returns information about bind variables after a statement has been prepared. This includes bind names, indicator names, and whether or not binds are duplicate binds. This call also returns an associated bind handle if there is one. The call sets the *found* parameter to the total number of bind variables and not just the number of distinct bind variables.

This function does not include SELECT INTO list variables, because they are not considered to be binds.

The statement must have been prepared with a call to *OCIStmtPrepare()* prior to this call.

This call is processed locally.

Related Functions

OCIStmtPrepare()

OCIStmtGetPieceInfo()

Purpose

Returns piece information for a piecewise operation.

Syntax

```
sword OCIStmtGetPieceInfo( CONST OCIStmt  *stmtp,
                           OCIError      *errhp,
                           dvoid          **hndlpp,
                           ub4            *typep,
                           ub1            *in_outp,
                           ub4            *iterp,
                           ub4            *idxp,
                           ub1            *piecep );
```

Parameters

stmtp (IN)

The statement when executed returned OCI_NEED_DATA.

errhp (OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

hndlpp (OUT)

Returns a pointer to the bind or define handle of the bind or define whose runtime data is required or is being provided.

typep (OUT)

The type of the handle pointed to by hndlpp: OCI_HTYPE_BIND (for a bind handle) or OCI_HTYPE_DEFINE (for a define handle).

in_outp (OUT)

Returns OCI_PARAM_IN if the data is required for an IN bind value. Returns OCI_PARAM_OUT if the data is available as an OUT bind variable or a define position value.

iterp (OUT)

Returns the row number of a multiple row operation.

idxp (OUT)

The index of an array element of a PL/SQL array bind operation.

piecep (OUT)

Returns one of the following defined values OCI_ONE_PIECE, OCI_FIRST_PIECE, OCI_NEXT_PIECE and OCI_LAST_PIECE.

Comments

When an execute/fetch call returns OCI_NEED_DATA to get/return a dynamic bind/define value or piece, *OCIStmtGetPieceInfo()* returns the relevant information: bind/define handle, iteration, index number and which piece.

See the section “Run Time Data Allocation and Piecewise Operations” on page 7-16 for more information about using *OCIStmtGetPieceInfo()*.

Related Functions

OCIAQEnq(), *OCIAQEnq()*, *OCIStmtExecute()*, *OCIStmtFetch()*, *OCIStmtSetPieceInfo()*

OCIStmtPrepare()

Purpose

This call prepares a SQL or PL/SQL statement for execution.

Syntax

```
sword OCIStmtPrepare ( OCIStmt      *stmtp,  
                      OCIError     *errhp,  
                      CONST text    *stmt,  
                      ub4           stmt_len,  
                      ub4           language,  
                      ub4           mode );
```

Parameters

stmtp (IN)

A statement handle.

errhp (IN)

An error handle to retrieve diagnostic information.

stmt (IN)

SQL or PL/SQL statement to be executed. Must be a null-terminated string. The pointer to the text of the statement must be available as long as the statement is executed, or data is fetched from it.

stmt_len (IN)

Length of the statement. Must not be zero.

language (IN)

Specifies V7, V8, or native syntax. Possible values are:

- OCI_V7_SYNTAX - V7 ORACLE parsing syntax
- OCI_V8_SYNTAX - V8 ORACLE parsing syntax
- OCI_NTV_SYNTAX - syntax depends upon the version of the server.

mode (IN)

The only defined mode is OCI_DEFAULT for default mode.

Comments

An OCI application uses this call to prepare a SQL or PL/SQL statement for execution. The *OCIStmtPrepare()* call defines an application request.

This is a purely local call. Data values for this statement initialized in subsequent bind calls will be stored in a bind handle which will hang off this statement handle.

This call does not create an association between this statement handle and any particular server.

See the section “Preparing Statements” on page 4-4 for more information about using this call.

Related Functions

OCIAQEnq(), *OCIStmtExecute()*

OCIStmtSetPieceInfo()

Purpose

Sets piece information for a piecewise operation.

Syntax

```
sword OCIStmtSetPieceInfo ( dvoid          *hndlp,  
                           ub4            type,  
                           OCIError      *errhp,  
                           CONST dvoid    *bufp,  
                           ub4            *alenp,  
                           ub1            piece,  
                           CONST dvoid    *indp,  
                           ub2            *rcodep );
```

Parameters

hndlp (IN/OUT)

The bind/define handle.

type (IN)

Type of the handle.

errhp (OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

bufp (IN/OUT)

A pointer to a storage containing the data value or the piece when it is an IN bind variable, otherwise *bufp* is a pointer to storage for getting a piece or a value for OUT binds and define variables. For named data types or REFS, a pointer to the object or REF is returned.

alenp (IN/OUT)

The length of the piece or the value.

piece (IN)

The piece parameter. Valid values:

- OCI_ONE_PIECE
- OCI_FIRST_PIECE
- OCI_NEXT_PIECE
- OCI_LAST_PIECE

This parameter is used for IN bind variables only.

indp (IN/OUT)

Indicator. A pointer to a **sb2** value or pointer to an indicator structure for named data types (SQLT_NTY) and REFs (SQLT_REF), i.e., **indp* is either an **sb2** or a **dvoid *** depending upon the data type.

rcodep (IN/OUT)

Return code.

Comments

When an execute call returns OCI_NEED_DATA to get a dynamic IN/OUT bind value or piece, *OCIStmtSetPieceInfo()* sets the piece information: the buffer, the length, which piece is currently being processed, the indicator, and the return code for this column.

For more information about using *OCIStmtSetPieceInfo()* see the section “Run Time Data Allocation and Piecewise Operations” on page 7-16.

Related Functions

OCIAQEnq(), *OCIAQEnq()*, *OCIStmtExecute()*, *OCIStmtFetch()*, *OCIStmtGetPieceInfo()*

OCISvcCtxToLda()

Purpose

Toggles between a V8 service context handle and a V7 Lda_Def.

Syntax

```
sword OCISvcCtxToLda ( OCISvcCtx      *srvhp,  
                      OCIError      *errhp,  
                      Lda_Def       *ldap );
```

Parameters

svchp (IN/OUT)

The service context handle.

errhp (IN/OUT)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

ldap (IN/OUT)

A Logon Data Area for Oracle7-style OCI calls which is initialized by this call.

Comments

Toggles between an Oracle8 service context handle and an Oracle7 **Lda_Def**.

This function can only be called after a service context has been properly initialized.

Once the service context has been translated to an **Lda_Def**, it can be used in release 7.x OCI calls (e.g., *obindps()*, *ofen()*).

Note: If there are multiple service contexts which share the same server handle, only one can be in Oracle7 mode at any time.

The action of this call can be reversed by passing the resulting **Lda_Def** to the *OCILdaToSvcCtx()* function.

The OCI_ATTR_IN_V8_MODE attribute of the server handle or service context handle enables an application to determine whether the application is currently in Oracle7 mode or Oracle8 mode. See Appendix B, “Handle and Descriptor Attributes”, for more information.

OCISvcCtxToLda()

Related Functions

OCILdaToSvcCtx()

OCITransCommit()

Purpose

Commits the transaction associated with a specified service context.

Syntax

```
sword OCITransCommit ( OCISvcCtx      *svchp,  
                        OCIError       *errhp,  
                        ub4             flags );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

flags (IN)

See the “Comments” section below.

Comments

The transaction currently associated with the service context is committed. If it is a global transaction that the server cannot commit, this call additionally retrieves the state of the transaction from the database to be returned to the user in the error handle.

If the application has defined multiple transactions, this function operates on the transaction currently associated with the service context. If the application is working with only the implicit local transaction created when database changes are made, that implicit transaction is committed.

If the application is running in the object mode, then the modified or updated objects in the object cache for this transaction are also flushed and committed.

The flags parameter is used for one-phase commit optimization in global transactions. If the transaction is non-distributed, the flags parameter is ignored, and OCI_DEFAULT can be passed as its value. OCI applications managing global

transactions should pass a value of `OCI_TRANS_TWOPHASE` to the flags parameter for a two-phase commit. The default is one-phase commit.

Under normal circumstances, *OCITransCommit()* returns with a status indicating that the transaction has either been committed or rolled back. With global transactions, it is possible that the transaction is now in-doubt (i.e., neither committed nor aborted). In this case, *OCITransCommit()* attempts to retrieve the status of the transaction from the server. The status is returned.

Example

The following example demonstrates the use of a simple local transaction, as described in the section “Simple Local Transactions” on page 7-4.

```
int main()
{
    OCIEnv *envhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCISmt *stmthp;
    dvoid *tmp;
    text sqlstmt[128];

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    0, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                    52, (dvoid **) &tmp);

    OCIErrorAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                    52, (dvoid **) &tmp);

    OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, 0, 0);

    OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp, 0,
```

```
OCI_ATTR_SERVER, errhp);

OCILogon(envhp, errhp, &svchp, "SCOTT", strlen("SCOTT"),
        "TIGER", strlen("TIGER"), 0, 0);

/* update scott.emp empno=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* update scott.emp empno=7902, increment salary again, but rollback */
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
OCITransRollback(svchp, errhp, (ub4) 0);
}
```

Related Functions

OCITransRollback()

OCITransDetach()

Purpose

Detaches a transaction.

Syntax

```
sword OCITransDetach ( OCISvcCtx      *svchp,  
                       OCIError      *errhp,  
                       ub4            flags );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

flags (IN)

You must pass a value of OCI_DEFAULT for this parameter.

Comments

Detaches a global transaction from the service context handle. The transaction currently attached to the service context handle becomes inactive at the end of this call. The transaction may be resumed later by calling *OCITransStart()*, specifying a flags value of OCI_TRANS_RESUME.

When a transaction is detached, the value which was specified in the timeout parameter of *OCITransStart()* when the transaction was started is used to determine the amount of time the branch can remain inactive before being deleted by the server's PMON process.

Note: The transaction can be resumed by a different process than the one that detached it, provided that the transaction has the same authorization.

If this function is called before a transaction is actually started, this function is a no-op.

Related Functions

OCITransStart()

OCITransForget()

Purpose

Causes the server to forget a heuristically completed global transaction.

Syntax

```
sword OCITransForget ( OCISvcCtx      *svchp,  
                       OCIError       *errhp,  
                       ub4             flags );
```

Parameters

svchp (IN)

The service context handle in which the transaction resides.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

flags (IN)

You must pass OCI_DEFAULT for this parameter.

Comments

Forgets a heuristically completed global transaction. The server deletes the status of the transaction from the system's pending transaction table.

The XID of the transaction to be forgotten is set as an attribute of the transaction handle (OCI_ATTR_XID).

Related Functions

OCITransCommit(), *OCITransRollback()*

OCITransPrepare()

Purpose

Prepares a transaction for commit.

Syntax

```
sword OCITransPrepare ( OCISvcCtx      *svchp,  
                        OCIError      *errhp,  
                        ub4            flags );
```

Parameters

svchp (IN)

The service context handle.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

flags (IN)

You must pass OCI_DEFAULT for this parameter.

Comments

Prepares the specified global transaction for commit.

This call is valid only for global transactions.

The call returns OCI_SUCCESS_WITH_INFO if the transaction has not made any changes. The error handle will indicate that the transaction is read-only. The flag parameter is not currently used.

Related Functions

OCITransCommit(), OCITransForget()

OCITransRollback()

Purpose

Rolls back the current transaction.

Syntax

```
sword OCITransRollback ( dvoid          *svchp,  
                        OCIError       *errhp,  
                        ub4            flags );
```

Parameters

svchp (IN)

A service context handle. The transaction currently set in the service context handle is rolled back.

errhp (IN)

An error handle which can be passed to *OCIErrorGet()* for diagnostic information in the event of an error.

flags (IN)

You must pass a value of OCI_DEFAULT for this parameter.

Comments

The current transaction— defined as the set of statements executed since the last *OCITransCommit()* or since *OCISessionBegin()*—is rolled back.

If the application is running under object mode then the modified or updated objects in the object cache for this transaction are also rolled back.

An error is returned if an attempt is made to roll back a global transaction that is not currently active.

Related Functions

OCITransCommit()

OCITransStart()

Purpose

Sets the beginning of a transaction.

Syntax

```
sword OCITransStart ( OCISvcCtx      *svchp,  
                      OCIError      *errhp,  
                      uword         timeout,  
                      ub4           flags );
```

Parameters

svchp (IN/OUT)

The service context handle. The transaction context in the service context handle is initialized at the end of the call if the flag specified a new transaction to be started.

errhp (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns `OCI_ERROR`. Diagnostic information can be obtained by calling *OCIErrorGet()*.

timeout (IN)

The time, in seconds, to wait for a transaction to become available for resumption when `OCI_TRANS_RESUME` is specified. When `OCI_TRANS_NEW` is specified, the timeout parameter indicates the number of seconds the transaction can be inactive before it is automatically aborted by the system. A transaction is inactive between the time it is detached (with *OCITransDetach()*) and the time it is resumed with *OCITransStart()*.

flags (IN)

Specifies whether a new transaction is being started or an existing transaction is being resumed. Also specifies serializability or read-only status. More than a single value can be specified. By default, a read/write transaction is started. The flag values are:

- `OCI_TRANS_NEW` - starts a new transaction branch. By default starts a tightly coupled and migratable branch.
- `OCI_TRANS_TIGHT` - explicitly specifies a tightly coupled branch

- OCI_TRANS_LOOSE - specifies a loosely coupled branch
- OCI_TRANS_RESUME - resumes an existing transaction branch.
- OCI_TRANS_READONLY - start a read-only transaction
- OCI_TRANS_SERIALIZABLE - start a serializable transaction

Comments

This function sets the beginning of a global or serializable transaction. The transaction context currently associated with the service context handle is initialized at the end of the call if the flags parameter specifies that a new transaction should be started.

The XID of the transaction is set as an attribute of the transaction handle (OCI_ATTR_XID)

Examples

The following examples demonstrate the use of OCI transactional calls for manipulating global transactions.

Example 1

This example shows a single session operating on different branches. This concept is illustrated by Figure 7-2, “Session Operating on Multiple Branches” on page 7-6.

```
int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCISession *usrhp;
    OCISmt *stmthp1, *stmthp2;
    OCITrans *txnhp1, *txnhp2;
    dvoid *tmp;
    XID gxid;
    text sqlstmt[128];

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    0, (dvoid **) &tmp);
```

```

OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                52, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                52, (dvoid **) &tmp);

OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp1, OCI_HTYPE_STMT, 0, 0);
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp2, OCI_HTYPE_STMT, 0, 0);

OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp, 0,
            OCI_ATTR_SERVER, errhp);

/* set the external name and internal name in server handle */
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "demo", 0,
            OCI_ATTR_EXTERNAL_NAME, errhp);
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "txn demo", 0,
            OCI_ATTR_INTERNAL_NAME, errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"scott",
            (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"tiger",
            (ub4)strlen("tiger"),OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, 0);

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
            (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate transaction handle 1 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp1, OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
            OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 123, 1] */
gxid.formatID = 1000; /* format id = 1000 */

```

```
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;

OCIAttrSet((dvoid *)txnhp1, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
           OCI_ATTR_XID, errhp);

/* start global transaction 1 with 60 second time to live when detached */
OCITransStart(svchp, errhp, 60, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
OCIStmtPrepare(stmthp1, errhp, sqlstmt, strlen(sqlstmt), OCI_NIV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp1, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);

/* allocate transaction handle 2 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp2, OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCTX, (dvoid *)txnhp2, 0,
           OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 124, 1] */
gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 124 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 4;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;

OCIAttrSet((dvoid *)txnhp2, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
           OCI_ATTR_XID, errhp);

/* start global transaction 2 with 90 second time to live when detached */
OCITransStart(svchp, errhp, 90, OCI_TRANS_NEW);

/* update scott.emp empno=7934, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7934");
OCIStmtPrepare(stmthp2, errhp, sqlstmt, strlen(sqlstmt), OCI_NIV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp2, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);
```

```

/* Resume transaction 1, increment salary and commit it */
/* Set transaction handle 1 into the service handle */
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
           OCI_ATTR_TRANS, errhp);

/* attach to transaction 1, wait for 10 seconds if the transaction is busy */
/* The wait is clearly not required in this example because no other */
/* process/thread is using the transaction. It is only for illustration */
OCITransStart(svchp, errhp, 10, OCI_TRANS_RESUME);
OCIStmtExecute(svchp, stmthp1, errhp, 1, 0, 0, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* attach to transaction 2 and commit it */
/* set transaction handle2 into the service handle */
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
           OCI_ATTR_TRANS, errhp);
OCITransCommit(svchp, errhp, (ub4) 0);
}

```

Example 2

This example demonstrates a single session operating on multiple branches that share the same transaction.

```

int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCISession *usrhp;
    OCIStmt *stmthp;
    OCITrans *txnhp1, *txnhp2;
    dvoid *tmp;
    XID gxid;
    text sqlstmt[128];

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                 (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                   0, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,

```

```
        52, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
        52, (dvoid **) &tmp);

OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
        52, (dvoid **) &tmp);

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, 0, 0);

OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp, 0,
        OCI_ATTR_SERVER, errhp);

/* set the external name and internal name in server handle */
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "demo", 0,
        OCI_ATTR_EXTERNAL_NAME, errhp);
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "txn demo2", 0,
        OCI_ATTR_INTERNAL_NAME, errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
        (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"scott",
        (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"tiger",
        (ub4)strlen("tiger"),OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, 0);

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
        (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate transaction handle 1 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp1, OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
        OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 123, 1] */
gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;
```

```

OCIAttrSet((dvoid *)txnhp1, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
           OCI_ATTR_XID, errhp);

/* start global transaction 1 with 60 second time to live when detached */
OCITransStart(svchp, errhp, 60, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen(sqlstmt), OCI_NIV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);

/* allocate transaction handle 2 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp2, OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
           OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 123, 2] */
/* The global transaction will be tightly coupled with earlier transaction */
/* There is not much practical value in doing this but the example */
/* illustrates the use of tightly-coupled transaction branches */
/* In a practical case the second transaction that tightly couples with */
/* the first can be executed from a different process/thread */

gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 2 */
gxid.data[3] = 2;

OCIAttrSet((dvoid *)txnhp2, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
           OCI_ATTR_XID, errhp);

/* start global transaction 2 with 90 second time to live when detached */
OCITransStart(svchp, errhp, 90, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
/* This is possible even if the earlier transaction has locked this row */
/* because the two global transactions are tightly coupled */
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */

```

```
OCITransDetach(svchp, errhp, 0);

/* Resume transaction 1 and prepare it. This will return */
/* OCI_SUCCESS_WITH_INFO because all branches except the last branch */
/* are treated as read-only transactions for tightly-coupled transactions */

OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
           OCI_ATTR_TRANS, errhp);
if (OCITransPrepare(svchp, errhp, (ub4) 0) == OCI_SUCCESS_WITH_INFO)
{
    text errbuf[512];
    ub4 buflen;
    sb4 errcode;

    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("OCITransPrepare - %s\n", errbuf);
}

/* attach to transaction 2 and commit it */
/* set transaction handle2 into the service handle */
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
           OCI_ATTR_TRANS, errhp);
OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Related Functions

OCITransDetach()

OCI Navigation and Type Functions

This chapter describes the OCI navigational functions which are used to navigate through objects retrieved from an Oracle8 server. It also contains the descriptions of the functions which are used to obtain type descriptor objects (TDOs). The chapter contains the following sections:

- Introduction
- OCI Navigational Functions Quick Reference
- The OCI Navigational Functions

Note: The functions described in this chapter are only available if you have purchased the Oracle8 Enterprise Edition with the Objects Option.

Introduction

In an object navigational paradigm, data is represented as a graph of objects connected by references. Objects in the graph are reached by following the references. The OCI provides a navigational interface to objects in the Oracle8 server. Those calls are described in this chapter.

The OCI object environment is initialized when the application calls *OCIInitialize()* in OCI_OBJECT mode.

See Also: For more information about using the calls in this chapter, refer to Chapter 8, “OCI Object-Relational Programming”, and Chapter 11, “Object Cache and Object Navigation”.

Object Types and Lifetimes

An object instance is an occurrence of a type defined in an Oracle database. This section describes how an object instance can be represented in OCI. In OCI, an object instance can be classified based on the type, the lifetime and referenceability (see Figure 14–1 below):

1. A persistent object is an instance of an object type. A persistent object resides in a row of a table in the server and can exist longer than the duration of a session (connection). Persistent objects can be identified by object references which contain the object identifiers. A persistent object is obtained by pinning its object reference.
2. A transient object is an instance of an object type. A transient object cannot exist longer than the duration of a session, and it is used to contain temporary computing results. Transient objects can also be identified by references which contain transient object identifiers.
3. A value is an instance of an user-defined type (object type or collection type) or any built-in Oracle type. Unlike objects, values of object types are identified by memory pointers, rather than by references.

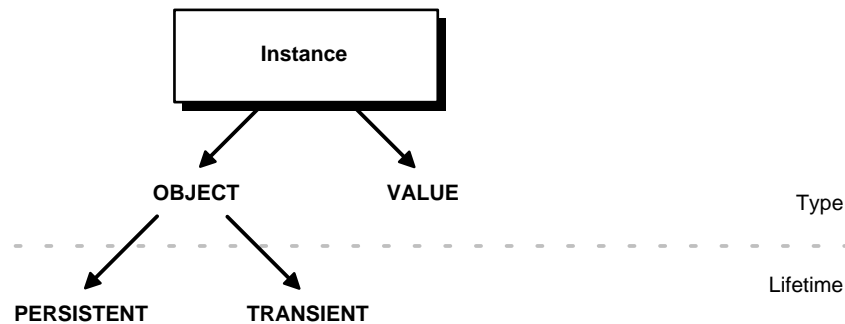
A value can be standalone or embedded. A standalone value is usually obtained by issuing a select statement. OCI also allows the client program to select a row of object table into a value by issuing a SQL statement. Thus, a referenceable object (in the database) can be represented as a value (which cannot be identified by a reference). A standalone value can also be an out-of-line attribute in an object (e.g., **VARCHAR, raw**) or an out-of-line element in a collection (e.g., **VARCHAR, raw, object**).

An embedded value is physically included in a containing instance. An embedded value can be an in-line attribute in an object (e.g. number, nested object) or an in-line element in a collection.

All values are considered to be transient by OCI, e.g. OCI does not support automatic flushing a value to the database, and the client has to explicitly execute a SQL statement to store a value into the database. For embedded values, they are flushed when their containing instance are flushed.

The following figure shows how instances can be classified according to their type and lifetime:

Figure 14–1 Classification of Instances by Type and Lifetime



The distinction between various instances is further illustrated by the following table:

	Persistent Object	Transient Object	Value
Type	object type	object type	object type, built-in, collection
Maximum Lifetime	until object is deleted	session	session
Referenceable	yes	yes	no
Embeddable	no	no	yes

Terminology

In the remainder of this chapter, the following terms will be used:

- 1) The term *object* can be generally used to refer to a persistent object, a transient object, a standalone value of object type, or an embedded value of object type.
- 2) The term *referenceable object* refers to a persistent object or a transient object.
- 3) The term *standalone object* refers to a persistent object, a transient object or a standalone value of object type.
- 4) The term *embedded object* refers to an embedded value of object type.

For a further discussion of the terms used to refer to different types of objects, please see “Persistent Objects, Transient Objects, and Values” on page 8-5.

An object is *dirty* if it has been created (newed), or marked updated or deleted.

Navigational Function Return Values

The OCI navigational functions typically return one of the following values:

Return Value	Meaning
OCI_SUCCESS	The operation succeeded
OCI_ERROR	The operation failed. The specific error can be retrieved by calling <i>OCIErrorGet()</i> on the error handle passed to the function.
OCI_INVALID_HANDLE	The environment or error handle passed to the function is NULL.

Function-specific return information follows the description of each function in this chapter. Information about specific error codes returned by each function is presented in the following section.

See Also: For more information about return codes and error handling, see the section “Error Handling” on page 2-25.

Navigational Function Error Codes

Table 14–1 lists the external Oracle error codes which can be returned by each of the OCI navigational functions. The list following the table identifies what each error represents.

Table 14–1 OCI Navigational Functions Error Codes

Function	Possible ORA Errors
OCIObjectNew()	24350, 21560, 21705, 21710
OCIObjectPin()	24350, 21560, 21700, 21702
OCIObjectUnpin()	24350, 21560, 21710
OCIObjectPinCountReset()	24350, 21560, 21710
OCIObjectLock()	24350, 21560, 21701, 21708, 21710
OCIObjectMarkUpdate()	24350, 21560, 21700, 21701, 21710
OCIObjectUnmark()	24350, 21560, 21710
OCIObjectUnmarkByRef()	24350, 21560
OCIObjectFree()	24350, 21560, 21603, 21710
OCIObjectMarkDelete()	24350, 21560, 21700, 21701, 21702, 21710
OCIObjectMarkDeleteByRef()	24350, 21560
OCIObjectFlush()	24350, 21560, 21701, 21703, 21708, 21710
OCIObjectRefresh()	24350, 21560, 21709, 21710
OCIObjectCopy()	24350, 21560, 21705, 21710
OCIObjectGetTypeRef()	24350, 21560, 21710
OCIObjectGetObjectRef()	24350, 21560, 21710
OCIObjectGetInd()	24350, 21560, 21710
OCIObjectExists()	24350, 21560, 21710
OCIObjectIsLocked()	24350, 21560, 21710
OCIObjectIsDirty()	24350, 21560, 21710
OCIObjectPinTable()	24350, 21560, 21705
OCIObjectArrayPin()	24350, 21560
OCICacheFlush()	24350, 21560, 21705

Table 14–1 OCI Navigational Functions Error Codes (Cont.)

Function	Possible ORA Errors
OCICacheRefresh()	24350, 21560, 21705
OCICacheUnpin()	24350, 21560, 21705
OCICacheFree()	24350, 21560, 21705
OCICacheUnmark()	24350, 21560, 21705
OCIObjectSetAttr()	21560, 21600, 22305, 22279, 21601
OCIObjectGetAttr()	21560, 21600, 22305

The ORA errors in Table 14–1 have the following meanings.

- ORA-21560 - name argument should not be NULL

- ORA-21600 - path expression too long
- ORA-21601 - attribute is not an instance of user-defined type
- ORA-21603 - cannot free a dirtied persistent object

- ORA-21700 - object does not exist or has been deleted
- ORA-21701 - invalid object
- ORA-21702 - object is not instantiated in the cache
- ORA-21703 - cannot flush an object that is not modified
- ORA-21704 - terminate cache or connection without flushing
- ORA-21705 - service context is invalid
- ORA-21708 - operations cannot be performed on a transient object
- ORA-21709 - operations can only be performed on a current object
- ORA-21710 - invalid pointer or value passed to the function

- ORA-22279 - cannot perform operation with LOB buffering enabled
- ORA-22305 - name argument is invalid
- ORA-24350 - this OCI call is not allowed from external subroutines

Server Roundtrips for Cache and Object Functions

For a table showing the number of server roundtrips required for individual OCI cache and object functions, refer to Appendix E, “OCI Function Server Roundtrips”.

OCI Navigational Functions Quick Reference

This section is intended to help you figure out which OCI navigational call you need to use in a given situation. Table 14–2 includes all of the navigational functions, grouped by categories of functionality. The list includes the name of each call, a brief description of its purpose, and the page number on which the full description can be found.

Table 14–2 OCI Navigational Functions Quick Reference

Function	Purpose	Page
FLUSH OR REFRESH OBJECT/CACHE		
OCICacheFlush()	Flush modified persistent objects in cache to server	14 - 11
OCIObjectFlush()	Flush a modified persistent object to the server	14 - 23
OCICacheRefresh()	Refresh pinned persistent objects	14 - 14
OCIObjectRefresh()	Refresh a persistent object	14 - 53
MARK OR UNMARK OBJECT/CACHE		
OCIObjectMarkDeleteByRef()	Mark an object deleted given a ref	14 - 40
OCIObjectMarkUpdate()	Mark an object as updated/dirty	14 - 41
OCIObjectMarkDelete()	Mark an object deleted / delete a value instance	14 - 38
OCICacheUnmark()	Unmarks objects in the cache	14 - 16
OCIObjectUnmark()	Unmarks an object	14 - 57
OCIObjectUnmarkByRef()	Unmarks an object, given a ref to it	14 - 58
GET OBJECT STATUS		
OCIObjectExists()	Get the existent status of an instance	14 - 22
OCIObjectIsDirty()	Get the dirtied status of an instance	14 - 35
OCIObjectIsLocked()	Get the locked status of an instance	14 - 36
OCIObjectGetProperty()	Get the status of a particular object property	14 - 30

Table 14–2 OCI Navigational Functions Quick Reference

Function	Purpose	Page
PIN/UNPIN/FREE		
OCIObjectPin()	Pin an object	14 - 46
OCIObjectUnpin()	Unpin an object	14 - 59
OCIObjectPinCountReset()	Unpin an object to zero pin count	14 - 49
OCICacheUnpin()	Unpin persistent objects in cache or connection	14 - 17
OCIObjectArrayPin()	Pin an array of references	14 - 18
OCIObjectPinTable()	Pin a table object with a given duration	14 - 51
OCICacheFree()	Free objects in the cache	14 - 13
OCIObjectFree()	Free a previously allocated object	14 - 24
OTHER FUNCTIONS		
OCIObjectCopy()	Copy one instance to another	14 - 20
OCIObjectGetInd()	Get null structure of an instance	14 - 28
OCIObjectGetObjectRef()	Return reference to a given object	14 - 29
OCIObjectGetTypeRef()	Get a reference to a TDO of an instance	14 - 30
OCIObjectLock()	Lock a persistent object	14 - 37
OCIObjectNew()	Create a new instance	14 - 43
TYPE INFORMATION ACCESSOR FUNCTIONS		
OCITypeArrayByName()	Get an array of TDOs given an array of object names	14 - 61
OCITypeArrayByRef()	Get an array of TDOs given an array of object references	14 - 64
OCITypeByName()	Get a TDO given an object name	14 - 66
OCITypeByRef()	Get a TDO given an object reference	14 - 69

The OCI Navigational Functions

This chapter describes the functions which belong to the object navigational component of the OCI. The entries for each function contain the following information:

Purpose

A brief description of what the function does.

Syntax

A code snippet showing the syntax for calling the function, including the ordering and types of the parameters.

Comments

Detailed information about the function (if available). This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below:

Mode	Description
IN	A parameter that passes data to Oracle
OUT	A parameter that receives data from Oracle on this or a subsequent call
IN/OUT	A parameter that passes data on the call and receives data on the return from this or a subsequent call.

Returns

A description of what value is returned by the function if the function returns something other than the standard return codes listed above.

Related Functions

A list of related calls which may provide additional useful information.

OCICacheFlush()

Purpose

Flushes modified persistent objects to the server

Syntax

```

sword OCICacheFlush ( OCIEnv          *env,
                     OCIError        *err,
                     CONST OCISvcCtx *svc,
                     dvoid           *context,
                     OCIRef          *( *get )
                     ( dvoid         *context,
                       ubl           *last ),
                     OCIRef          **ref );

```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

OCI service context.

context (IN) [optional]

Specifies an user context that is an argument to the client callback function *get*. This parameter is set to NULL if there is no user context.

get (IN) [optional]

A client-defined function which acts an iterator to retrieve a batch of dirty objects that need to be flushed. If the function is not NULL, this function will be called to get a reference of a dirty object. This is repeated until a null reference is returned by the client function or the parameter *last* is set to TRUE. The parameter *context* is passed to *get()* for each invocation of the client function. This parameter should be NULL if user callback is not given. If the object that is returned by the client function is not a dirtied persistent object, the object is ignored.

All the objects that are returned from the client function must be newed or pinned using the same service context, otherwise an error is signalled. Note that the cache flushes the returned objects in the order in which they were marked dirty.

If this parameter is passed as NULL (e.g., no client-defined function is provided), then all dirty persistent objects for the given service context are flushed in the order in which they were dirtied.

ref (OUT) [optional]

If there is an error in flushing the objects (**ref*) will point to the object that is causing the error. If *ref* is NULL, then the object will not be returned. If **ref* is NULL, then a reference will be allocated and set to point to the object. If **ref* is not NULL, then the reference of the object is copied into the given space. If the error is not caused by any of the dirtied object, the given REF is initialized to be a NULL reference (*OCIRefIsNull(*ref)* is TRUE).

The REF is allocated for session duration (OCI_DURATION_SESSION). The application can free the allocated REF using the *OCIObjectFree()* function.

Comments

This function flushes the modified persistent objects from the object cache to the server. The objects are flushed in the order that they are newed or marked updated or deleted.

This function incurs at most one network round-trip.

See *OCIObjectFlush()* for more information about flushing.

Related Functions

OCIObjectFlush()

OCICacheFree()

Purpose

Frees all objects and values in the cache for the specified connection

Syntax

```
sword OCICacheFree ( OCIEnv          *env,  
                     OCIError        *err,  
                     CONST OCISvcCtx *svc );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

An OCI service context.

Comments

If a connection is specified, this function frees the persistent objects, transient objects and values allocated for that connection. Otherwise, all persistent objects, transient objects and values in the object cache are freed. Objects are freed regardless of their pin count.

See *OCIObjectFree()* for more information about freeing an instance.

Related Functions

OCIObjectFree()

OCICacheRefresh()

Purpose

Refreshes all pinned persistent objects in the cache.

Syntax

```
sword OCICacheRefresh ( OCIEnv          *env,
                        OCIError        *err,
                        CONST OCISvcCtx *svc,
                        OCIRefreshOpt   option,
                        dvoid           *context,
                        OCIRef          *( *get)(dvoid *context),
                        OCIRef          **ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

OCI service context.

option (IN) [optional]

If OCI_REFRESH_LOADED is specified, all objects that are loaded within the transaction are refreshed. If the option is OCI_REFRESH_LOADED and the parameter *get* is not NULL, this function will ignore the parameter.

context (IN) [optional]

Specifies an user context that is an argument to the client callback function *get*. This parameter is set to NULL if there is no user context.

get (IN) [optional]

A client-defined function which acts an iterator to retrieve a batch of objects that need to be refreshed. If the function is not NULL, this function will be called to get a reference of an object. If the reference is not NULL, then the object will be

refreshed. These steps are repeated until a null reference is returned by this function. The parameter *context* is passed to *get()* for each invocation of the client function. This parameter should be NULL if user callback is not given.

ref (OUT) [optional]

If there is an error in refreshing the objects, (**ref*) will point to the object that is causing the error. If *ref* is NULL, then the object will not be returned. If **ref* is NULL, then a reference will be allocated and set to point to the object. If **ref* is not NULL, then the reference of the object is copied into the given space. If the error is not caused by any of the object, the given *ref* is initialized to be a NULL reference (*OCISRefIsNull(*ref)* is TRUE).

Comments

This function refreshes all pinned persistent objects.

All unpinned persistent objects are freed from the object cache.

For more information about refreshing, see the description of *OCIObjectRefresh()* on page 14-53, and the section “Refreshing an Object Copy” on page 11-10.

Warning: When objects are refreshed, the secondary-level memory of those objects could potentially move to a different place in memory. As a result, any pointers to attributes which were saved prior to this call may be invalidated. Examples of attributes using secondary-level memory include **OCIStrng ***, **OCIColl ***, and **OCIRaw ***.

Related Functions

OCIObjectRefresh()

OCI`CacheUnmark()`

Purpose

Unmarks all dirty objects in the object cache.

Syntax

```
sword OCICacheUnmark ( OCIEnv          *env,  
                        OCIError        *err,  
                        CONST OCISvcCtx  *svc );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns `OCI_ERROR`. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

OCI service context.

Comments

If a connection is specified, this function unmarks all dirty objects in that connection. Otherwise, all dirty objects in the cache are unmarked. See *OCIObjectUnmark()* for more information about unmarking an object.

Related Functions

OCIObjectUnmark()

OCICacheUnpin()

Purpose

Unpins persistent objects

Syntax

```
sword OCICacheUnpin ( OCIEnv          *env,  
                      OCIError        *err,  
                      CONST OCISvcCtx  *svc );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

An OCI service context handle. The objects on the specified connection are unpinned.

Comments

This function completely unpins all of the persistent objects for the given connection.

The pin count for the objects is reset to zero.

For more information about pinning and unpinning, see “Pinning an Object” on page 8-12, and “Pin Count and Unpinning” on page 8-28.

Related Functions

OCIObjectUnpin()

OCIObjectArrayPin()

Purpose

Pins an array of references

Syntax

```
sword OCIObjectArrayPin ( OCIEnv          *env,
                          OCIError        *err,
                          OCIRef          **ref_array,
                          ub4             array_size,
                          OCIComplexObject **cor_array,
                          ub4             cor_array_size,
                          OCIPinOpt       pin_option,
                          OCIDuration     pin_duration,
                          OCILockOpt      lock,
                          dvoid           **obj_array,
                          ub4             *pos );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

ref_array (IN)

Array of references to be pinned

array_size (IN)

Number of elements in the array of references

cor_array

An array of COR handles corresponding to the objects being pinned.

cor_array_size

The number of elements in *cor_array*.

pin_option (IN)

Pin option. See *OCIObjectPin()*.

pin_duration (IN)

Pin duration. See *OCIObjectPin()*.

lock (IN)

Lock option. See *OCIObjectPin()*.

obj_array (OUT)

If this argument is not NULL, the pinned objects will be returned in the array. The user must allocate this array with element type being **dvoid ***. The size of this array is identical to *array_size*.

pos (OUT)

If there is an error, this argument indicates the element that is causing the error. Note that this argument is set to 1 for the first element in the *ref_array*.

Comments

This function pins an array of references. All the pinned objects are retrieved from the database in one network roundtrip. If the user specifies an output array (*obj_array*), then the address of the pinned objects will be assigned to the elements in the array.

Related Functions

OCIObjectPin()

OCIObjectCopy()

Purpose

Copies a source instance to a destination

Syntax

```
sword OCIObjectCopy ( OCIEnv          *env,  
                      OCIError        *err,  
                      CONST OCISvcCtx *svc,  
                      dvoid            *source,  
                      dvoid            *null_source,  
                      dvoid            *target,  
                      dvoid            *null_target,  
                      OCIType          *tdo,  
                      OCIDuration      duration,  
                      ub1               option );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

An OCI service context handle, specifying the service context on which the copy operation is taking place

source (IN)

A pointer to the source instance; if it is an object, it must be pinned. See *OCIObjectPin()*.

null_source (IN)

Pointer to the NULL structure of the source object.

target (IN)

A pointer to the target instance; if it is an object is must be pinned.

null_target (IN)

A pointer to the NULL structure of the target object.

tdo (IN)

The TDO for both the source and the target. Can be retrieved with *OCIDescribeAny()*.

duration (IN)

Allocation duration of the target memory.

option (IN)

This parameter is currently unused. Pass as zero or OCI_DEFAULT.

Comments

This function copies the contents of the *source* instance to the *target* instance. This function performs a deep-copy such that all of the following is copied:

- all the top level attributes (see the exceptions below)
- all secondary memory (of the source) reachable from the top level attributes
- the NULL structure of the instance

Memory is allocated with the duration specified in the *duration* parameter.

Certain data items are not copied:

- If the option OCI_OBJECTCOPY_NOREF is specified in the *option* parameter, then all references in the source are not copied. Instead, the references in the target are set to NULL.
- If the attribute is an internal LOB, then only the LOB locator from the source object is copied. A copy of the LOB data is not made until *OCIObjectFlush()* is called. Before the target object is flushed, both the source and the target locators refer to the same LOB value.

The target or the containing instance of the target must be already have been created. This may be done with *OCIObjectNew()*.

The *source* and *target* instances must be of the same type. If the source and target are located in a different databases, then the same type must exist in both databases.

Related Functions

OCIObjectPin()

OCIObjectExists()

Purpose

Returns the existence meta-attribute of a standalone instance

Syntax

```
sword OCIObjectExists ( OCIEnv      *env,  
                        OCIError    *err,  
                        dvoid       *ins,  
                        boolean     *exist );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

ins (IN)

Pointer to an instance. If it is an object, it must be pinned.

exist (OUT)

Return value for the existence status.

Comments

This function returns the existence of an instance. If the instance is a value, this function always returns TRUE.

The instance must be a standalone persistent or transient object.

For more information about object meta-attributes, see “Object Meta-Attributes” on page 8-17.

Related Functions

OCIObjectPin()

OCIObjectFlush()

Purpose

Flushes a modified persistent object to the server

Syntax

```
sword OCIObjectFlush ( OCIEnv      *env,  
                      OCIError    *err,  
                      dvoid       *object );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object (IN)

A pointer to the persistent object. The object must be pinned before this call.

Comments

This function flushes a modified persistent object to the server. An exclusive lock is obtained implicitly for the object when it is flushed. When the object is written to the server, triggers may be fired. This function returns an error for transient objects and values, and for unmodified persistent objects.

Objects can be modified by triggers at the server. To keep objects in the cache consistent with the database, an application can free or refresh objects in the cache.

If the object to flush contains an internal LOB attribute, and the LOB attribute was modified due to an *OCIObjectCopy()* or *OCILobAssign()*, or by assigning another LOB locator to it, the flush makes a copy of the LOB value that existed in the source LOB at the time of the assignment or copy of the internal LOB locator or object.

Related Functions

OCIObjectPin(), *OCICacheFlush()*

OCIObjectFree()

Purpose

Frees and unpins an object instance

Syntax

```
sword OCIObjectFree ( OCIEnv          *env,  
                      OCIError        *err,  
                      dvoid           *instance,  
                      ub2             flags );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

instance (IN)

Pointer to a standalone instance. If it is an object, it must be pinned.

flags (IN)

If OCI_OBJECTFREE_FORCE is passed, free the object even if it is pinned or dirty. If OCI_OBJECTFREE_NONULL is passed, the null structure is not freed.

Comments

This function deallocates all the memory allocated for an object instance, including the null structure. The following rules apply for different instance types:

For Persistent Objects

This function returns an error if the client is attempting to free a dirty persistent object that has not been flushed. The client should either flush the persistent object, unmark it, or set the parameter *flags* to OCI_OBJECTFREE_FORCE.

This function calls *OCIObjectUnpin()* once to check if the object can be completely unpin. If it succeeds, the rest of the function proceeds to free the object. If it fails,

then an error is returned unless the parameter *flag* is set to `OCI_OBJECTFREE_FORCE`.

Freeing a persistent object in memory does not change the persistent state of that object at the server. For example, the object remains locked after the object is freed.

For Transient Objects

This function will call *OCIObjectUnpin()* once to check if the object can be completely unpin. If it succeeds, the rest of the function will proceed to free the object. If it fails, then an error is returned unless the parameter *flag* is set to `OCI_OBJECTFREE_FORCE`.

For Values

The memory of the object is freed immediately.

Related Functions

OCICacheFree()

OCIObjectGetAttr()

Purpose

Retrieves an object attribute

Syntax

```
sword OCIObjectGetAttr ( OCIEnv          *env,
                        OCIError        *err,
                        dvoid           *instance,
                        dvoid           *null_struct,
                        struct OCIType  *tdo,
                        CONST text      **names,
                        CONST ub4       *lengths,
                        CONST ub4       name_count,
                        CONST ub4       indexes,
                        CONST ub4       index_count,
                        OCIInd          *attr_null_status,
                        dvoid           **attr_null_struct,
                        dvoid           **attr_value,
                        struct OCIType  **attr_tdo );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

instance (IN)

Pointer to an object.

null_struct (IN)

The null structure of the object or array.

tdo (IN)

Pointer to the TDO.

names (IN)

Array of attribute names. This is used to specify the names of the attributes in the path expression.

lengths (IN)

Array of lengths of attribute names.

name_count (IN)

Number of element in the array *names*.

indexes (IN) [optional]

Not currently supported. Pass as (ub4 *)0.

index_count (IN) [optional]

Not currently supported. Pass as (ub4)0.

attr_null_status (OUT)

The null status of the attribute if the type of attribute is primitive.

attr_null_struct (OUT)

The null structure of an object or collection attribute.

attr_value (OUT)

Pointer to the attribute value.

attr_tdo (OUT)

Pointer to the TDO of the attribute.

Comments

This function gets a value from an object or from an array. If the parameter *instance* points to an object, then the path expression specifies the location of the attribute in the object. It is assumed that the object is pinned and that the value returned is valid until the object is unpinned.

Related Functions

OCIObjectSetAttr()

OCIObjectGetInd()

Purpose

Gets the NULL structure of a standalone instance

Syntax

```
sword OCIObjectGetInd ( OCIEnv      *env,  
                        OCIError    *err,  
                        dvoid        *instance,  
                        dvoid        **null_struct );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

instance (IN)

A pointer to the instance whose NULL structure is being retrieved. The instance must be standalone. If *instance* is an object, it must already be pinned.

null_struct (OUT)

The NULL structure for the instance.

Comments

This function returns the NULL structure of an instance.

Related Functions

OCIObjectPin()

OCIObjectGetObjectRef()

Purpose

Returns a reference to a given persistent object

Syntax

```
sword OCIObjectGetObjectRef ( OCIEnv      *env,
                             OCIError    *err,
                             dvoid       *object,
                             OCIRef      *object_ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object (IN)

Pointer to a persistent object. It must already be pinned.

object_ref (OUT)

A reference to the object specified in *object*. The reference must already be allocated. This can be accomplished with *OCIObjectNew()*.

Comments

This function returns a reference to the given persistent object, given a pointer to the object.

Passing a value (rather than an object) to this function causes an error.

See Also: For more information about object meta-attributes, see “Object Meta-Attributes” on page 8-17.

Related Functions

OCIObjectNew(), *OCIObjectPin()*

OCIObjectGetProperty()

Purpose

Retrieve a given property of an object.

Syntax

```
sword OCIObjectGetProperty ( OCIEnv          *envh,  
                             OCIError       *errh,  
                             CONST dvoid    *obj,  
                             OCIObjectPropId propertyId,  
                             dvoid         *property,  
                             ub4           *size );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

obj (IN)

The object whose property is returned.

propertyId (IN)

The identifier which identifies the desired property (see “Comments” below).

property (OUT)

The buffer into which the desired property is copied.

size (IN/OUT)

On input, this parameter specifies the size of the property buffer passed by caller.

On output it contains the size in bytes of the property returned. This parameter is required for string-type properties only (e.g OCI_OBJECTPROP_SCHEMA, OCI_OBJECTPROP_TABLE). For non-string properties this parameter is ignored since the size is fixed.

Comments

This function returns the specified property of the object. The desired property is identified by *propertyId*. The property value is copied into *property* and for string typed properties the string size is returned via *size*.

Objects are classified as persistent, transient and value depending upon the lifetime and referenceability of the object. Some of the properties are applicable only to persistent objects and some others only apply to persistent and transient objects. An error is returned if the user tries to get a property which is not applicable to the given object. To avoid such an error, the user should first check whether the object is persistent or transient or value (OCI_OBJECTPROP_LIFETIME property) and then appropriately query for other properties.

The different property ids and the corresponding type of *property* argument are given below.

OCI_OBJECTPROP_LIFETIME

This identifies whether the given object is a persistent object or a transient object or a value instance. The *property* argument must be a pointer to a variable of type **OCIObjectLifetime**. Possible values include:

- OCI_OBJECT_PERSISTENT
- OCI_OBJECT_TRANSIENT
- OCI_OBJECT_VALUE

OCI_OBJECTPROP_SCHEMA

This returns the schema name of the table in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the schema name an error is returned, the error message will communicate the required size. Upon success, the size of the returned schema name in bytes is returned via *size*. The *property* argument must be an array of type **text** and *size* should be set to size of array in bytes by the caller.

OCI_OBJECTPROP_TABLE

This returns the table name in which the object exists. An error is returned if the given object points to a transient instance or a value. If the input buffer is not big enough to hold the table name an error is returned, the error message will communicate the required size. Upon success, the size of the returned table name in bytes is returned via *size*. The *property* argument must be an array of type **text** and *size* should be set to size of array in bytes by the caller.

OCI_OBJECTPROP_PIN_DURATION

This returns the pin duration of the object. An error is returned if the given object points to a value instance. The *property* argument must be a pointer to a variable of type **OCIDuration**. Valid values include

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

For more information about durations, see “Object Duration” on page 11-13.

OCI_OBJECTPROP_ALLOC_DURATION

This returns the allocation duration of the object. The *property* argument must be a pointer to a variable of type **OCIDuration**. Valid values include:

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

For more information about durations, see “Object Duration” on page 11-13.

OCI_OBJECTPROP_LOCK

This returns the lock status of the object. The possible lock statuses are enumerated by **OCILockOpt**. An error is returned if the given object points to a transient or value instance. The *property* argument must be a pointer to a variable of type **OCILockOpt**. Note, the lock status of an object can also be retrieved by calling *OCIObjectIsLocked()*. Valid values include:

- OCI_LOCK_NONE - for no lock
- OCI_LOCK_X - for an exclusive lock

OCI_OBJECTPROP_MARKSTATUS

This returns the dirty status and indicates whether the object is a new object, updated object or deleted object. An error is returned if the given object points to a transient or value instance. The *property* argument must be of type **OCIObjectMarkStatus**. Valid values include:

- OCI_OBJECT_NEW
- OCI_OBJECT_DELETED
- OCI_OBJECT_UPDATED

The following macros are available to test the object mark status:

- OCI_OBJECT_IS_UPDATED(flag)
- OCI_OBJECT_IS_DELETED(flag)
- OCI_OBJECT_IS_NEW(flag)
- OCI_OBJECT_IS_DIRTY(flag)

OCI_OBJECTPROP_VIEW

This identifies whether the specified object is a view object or not. If the property value returned is TRUE, it indicates the object is a view otherwise it is not. An error is returned if the given object points to a transient or value instance. The *property* argument must be of type boolean.

Related Functions

OCIObjectLock(), OCIObjectMarkDelete(), OCIObjectMarkUpdate(), OCIObjectNew(), OCIObjectPin()

OCIObjectGetTypeRef()

Purpose

Returns a reference to the TDO of a standalone instance

Syntax

```
sword OCIObjectGetTypeRef ( OCIEnv      *env,  
                           OCIError    *err,  
                           dvoid        *instance,  
                           OCIRef       *type_ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

instance (IN)

A pointer to the standalone instance. It must be standalone, and if it is an object, it must already be pinned.

type_ref (OUT)

A reference to the type of the object. The reference must already be allocate. This can be accomplished with *OCIObjectNew()*.

Comments

This function returns a reference to the type descriptor object (TDO) of a standalone instance.

Related Functions

OCIObjectNew(), *OCIObjectPin()*

OCIObjectIsDirty()

Purpose

Check to see if an object is marked as dirty

Syntax

```
sword OCIObjectIsDirty ( OCIEnv      *env,  
                        OCIError    *err,  
                        dvoid       *ins,  
                        boolean     *dirty );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

ins (IN)

Pointer to an instance.

dirty (OUT)

Return value for the dirty status.

Comments

The instance passed to this function must be standalone. If the instance is an object, the instance must be pinned.

This function returns the dirty status of an instance. If the instance is a value, this function always returns FALSE for the dirty status.

Related Functions

OCIObjectMarkUpdate(), *OCIObjectGetProperty()*

OCIObjectIsLocked()

Purpose

Get lock status of an object.

Syntax

```
sword OCIObjectIsLocked ( OCIEnv      *env,
                          OCIError    *err,
                          dvoid       *ins,
                          boolean     *lock );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

ins (IN)

Pointer to an instance. The instance must be standalone, and if it is an object it must be pinned.

lock (OUT)

Return value for the lock status.

Comments

This function returns the lock status of an instance. If the instance is a value, this function always returns FALSE.

Related Functions

OCIObjectLock(), *OCIObjectGetProperty()*

OCIObjectLock()

Purpose

Locks a persistent object at the server

Syntax

```
sword OCIObjectLock ( OCIEnv      *env,
                      OCIError    *err,
                      dvoid        *object );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object (IN)

A pointer to the persistent object being locked. It must already be pinned.

Comments

This function locks a persistent object at the server. This function will return an error for transient objects and values.

For more information about object locking, see “Locking Objects For Update” on page 11-12.

This function returns an error if the object does not exist.

Related Functions

OCIObjectPin(), *OCIObjectIsLocked()*, *OCIObjectGetProperty()*

OCIObjectMarkDelete()

Purpose

Marks a standalone instance as deleted, given a pointer to the instance

Syntax

```
sword OCIObjectMarkDelete ( OCIEnv      *env,  
                             OCIError    *err,  
                             dvoid       *instance );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

instance (IN)

Pointer to the instance. It must be standalone, and if it is an object it must be pinned.

Comments

This function accepts a pointer to a standalone instance and marks the object as deleted. The object is freed according to the following rules:

For Persistent Objects

The object is marked deleted. The memory of the object is not freed. The object is deleted in the server when the object is flushed.

For Transient Objects

The object is marked deleted. The memory of the object is not freed.

For Values

This function frees a value immediately.

Related Functions

OCIObjectMarkDeleteByRef(), OCIObjectGetProperty()

OCIObjectMarkDeleteByRef()

Purpose

Marks an object as deleted, given a reference to the object

Syntax

```
sword OCIObjectMarkDeleteByRef ( OCIEnv      *env,  
                                OCIError    *err,  
                                OCIRef      *object_ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object_ref (IN)

Reference to the object to be deleted.

Comments

This function accepts a reference to an object, and marks the object designated by *object_ref* as deleted. The object is marked and freed as follows:

For Persistent Objects

If the object is not loaded, then a temporary object is created and is marked deleted. Otherwise, the object is marked deleted.

The object is deleted in the server when the object is flushed.

For Transient Objects

The object is marked deleted. The object is not freed until it is unpinned.

Related Functions

OCIObjectMarkDelete(), *OCIObjectGetProperty()*

OCIObjectMarkUpdate()

Purpose

Marks a persistent object as updated, or 'dirty'

Syntax

```
sword OCIObjectMarkUpdate ( OCIEnv      *env,  
                             OCIError    *err,  
                             dvoid        *object );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object (IN)

A pointer to the persistent object, which must already be pinned.

Comments

This function marks a persistent object as updated, or 'dirty.' The following special rules apply to different types of objects. The dirty status of an object may be checked by calling *OCIObjectIsDirty()*.

For Persistent Objects

This function marks the specified persistent object as updated. The persistent objects will be written to the server when the object cache is flushed. The object is not locked or flushed by this function. It is an error to update a deleted object.

After an object is marked updated and flushed, this function must be called again to mark the object as updated if it has been dirtied after it is being flushed.

For Transient Objects

This function marks the specified transient object as updated. The transient objects will NOT be written to the server. It is an error to update a deleted object.

For Values

This function is an no-op for values.

For more information about the use of this function, see “Marking Objects and Flushing Changes” on page 8-14.

Related Functions

OCIObjectPin(), OCIObjectGetProperty()

OCIObjectNew()

Purpose

Creates a standalone instance

Syntax

```

sword OCIObjectNew ( OCIEnv          *env,
                    OCIError        *err,
                    CONST OCISvcCtx *svc,
                    OCITypeCode     typecode,
                    OCIType         *tdo,
                    dvoid            *table,
                    OCIDuration     duration,
                    boolean          value,
                    dvoid            **instance );

```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN) [optional]

OCI service handle. It must be given if the program wants to associate the duration of an instance with an OCI service (e.g. free a string when the transaction is committed). This parameter is ignored if the TDO is given.

typecode (IN)

The typecode of the type of the instance. See “Typecodes” on page 3-24 for more information.

tdo (IN) [optional]

Pointer to the type descriptor object. The TDO describes the type of the instance that is to be created. Refer to *OCITypeByName()* for obtaining a TDO. The TDO is required for creating a named type (e.g. an object or a collection).

table (IN) [optional]

Pointer to a table object which specifies a table in the server. This parameter can be set to NULL if no table is given. See the description below to find out how the table object and the TDO are used together to determine the kind of instances (persistent, transient, value) to be created. Also see *OCIObjectPinTable()* for retrieving a table object.

duration (IN)

This is an overloaded parameter. The use of this parameter is based on the kind of the instance that is to be created.

- Persistent object. This parameter specifies the pin duration.
- Transient object. This parameter specifies the allocation duration and pin duration.
- Value. This parameter specifies the allocation duration.

value (IN)

Specifies whether the created object is a value. If TRUE, then a value is created. Otherwise, a referenceable object is created. If the instance is not an object, then this parameter is ignored.

instance (OUT)

Address of the newly created instance

Comments

This function creates a new instance of the type specified by the typecode or the TDO. Based on the parameters *typecode* (or *tdo*), *value* and *table*, different kinds of instances can be created:

TYPE	Value of <i>table</i> Parameter	
	Not NULL	NULL
object type (<i>value</i> =TRUE)	value	value
object type (<i>value</i> =FALSE)	persistent object	transient object
built-in type	value	value
collection type	value	value

For more information about typecodes, see “Typecodes” on page 3-24.

This function allocates the top-level memory chunk of an instance. The attributes in the top-level memory are initialized (e.g. an attribute of **`varchar2`** is initialized to a **`OCISString`** of 0 length).

If the instance is an object, the object is marked existed but is atomically null.

For Persistent Objects

The object is marked dirty and existed. The allocation duration for the object is session. The object is pinned and the pin duration is specified by the given parameter *duration*. Creating a persistent object does not cause any entries to be made into a database table until the object is flushed to the server.

For Transient Objects

The object is pinned. The allocation duration and the pin duration are specified by the given parameter *duration*.

For Values

The allocation duration is specified by the given parameter *duration*.

Objects with LOB Attributes

If the object contains an internal LOB attribute, the LOB is set to empty. The object must be marked as dirty and flushed (in order to insert the object into the table) and repinned before the user can start writing data into the LOB. When pinning the object after creating it, you must use the `OCI_PIN_LATEST` pin option in order to retrieve the newly updated LOB locator from the server.

If the object contains an external LOB attribute (`FILE`), the `FILE` locator is allocated but not initialized. The user must call `OCILobFileSetName()` to initialize the `FILE` attribute. Once the filename is set, the user can start reading from the `FILE`.

Note: Oracle8 supports only binary `FILEs` (`BFILEs`).

Related Functions

`OCIObjectPinTable()`, `OCIObjectFree()`

OCIObjectPin()

Purpose

Pin a referenceable object

Syntax

```
sword OCIObjectPin ( OCIEnv          *env,
                    OCIError        *err,
                    OCIRef          *object_ref,
                    OCIComplexObject *corhdl,
                    OCIPinOpt       pin_option,
                    OCIDuration     pin_duration,
                    OCILockOpt      lock_option,
                    dvoid            **object );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object_ref (IN)

The reference to the object.

corhdl (IN)

Handle for complex object retrieval.

pin_option (IN)

See description under “Comments” below.

pin_duration (IN)

The duration of which the object is being accessed by a client. The object is implicitly unpinned at the end of the pin duration. If OCI_DURATION_NULL is passed, there is no pin promotion if the object is already loaded into the cache. If the object is not yet loaded, then the pin duration is set to OCI_DURATION_DEFAULT in the case of OCI_DURATION_NULL.

lock_option (IN)

Lock option (e.g., exclusive). If a lock option is specified, the object is locked in the server. See *oro.h* for description about lock option.

object (OUT)

The pointer to the pinned object.

Comments

This function pins a referenceable object instance given the object reference. The process of pinning serves two purposes:

1. locate an object given its reference. This is done by the object cache which keeps track of the objects in the object cache.
2. notify the object cache that a persistent object is being in use such that the persistent object cannot be aged out. Since a persistent object can be loaded from the server whenever is needed, the memory utilization can be increased if a completely unpinned persistent object can be freed (aged out), even before the allocation duration is expired. An object can be pinned many times. A pinned object will remain in memory until it is completely unpinned (see *OCIObjectUnpin()*).

Also see *OCIObjectUnpin()* for more information about unpinning.

For Persistent Objects

When pinning a persistent object, if it is not in the cache, the object will be fetched from the persistent store. The allocation duration of the object is session. If the object is already in the cache, it is returned to the client. The object will be locked in the server if a lock option is specified.

This function will return an error for a non-existent object.

A pin option is used to specify the copy of the object that is to be retrieved:

- If *pin_option* is OCI_PIN_ANY (pin any), if the object is already in the object cache, return this object. Otherwise, the object is retrieved from the database. This option is useful when the client knows that he has the exclusive access to the data in a session.
- If *pin_option* is OCI_PIN_LATEST (pin latest), if the object is not locked, it is retrieved from the database. If the object is cached, it is refreshed with the latest version. See *OCIObjectRefresh()* for more information about refreshing.

- If *pin_option* is OCI_PIN_RECENT (pin recent), if the object is loaded into the cache in the current transaction, the object is returned. If the object is not loaded in the current transaction, the object is refreshed from the server.

For Transient Objects

This function will return an error if the transient object has already been freed. This function does not return an error if an exclusive lock is specified in the lock option.

Related Functions

OCIObjectUnpin(), *OCIObjectPinCountReset()*

OCIObjectPinCountReset()

Purpose

Completely unpins an object, setting its pin count to zero

Syntax

```
sword OCIObjectPinCountReset ( OCIEnv      *env,  
                               OCIError    *err,  
                               dvoid        *object );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

object (IN)

A pointer to an object, which must already be pinned.

Comments

This function completely unpins an object, setting its pin count to zero. When an object is completely unpinned, it can be freed implicitly by the OCI at any time without error.

The following rules apply for specific object types:

For Persistent Objects

When a persistent object is completely unpinned, it becomes a candidate for aging. The memory of an object is freed when it is aged out. Aging is used to maximize the utilization of memory. An dirty object cannot be aged out unless it is flushed.

For Transient Objects

The pin count of the object is decremented. A transient can be freed only at the end of its allocation duration or when it is explicitly freed by calling *OCIObjectFree()*.

For Values

This function will return an error for value.

For more information about the use of this function, see “Pin Count and Unpinning” on page 8-28.

Related Functions

OCIObjectPin(), *OCIObjectUnpin()*

OCIObjectPinTable()

Purpose

Pins a table object for a specified duration

Syntax

```

sword OCIObjectPinTable ( OCIEnv          *env,
                          OCIError        *err,
                          CONST OCISvcCtx *svc,
                          CONST text      *schema_name,
                          ub4             s_n_length,
                          CONST text      *object_name,
                          ub4             o_n_length,
                          dvoid           *not_used,
                          OCIDuration    pin_duration,
                          dvoid          **object );

```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

The OCI service context handle.

schema_name (IN) [optional]

The schema name of the table.

s_n_length (IN) [optional]

The length of the schema name indicated in *schema_name*.

object_name (IN)

The name of the table.

o_n_length (IN)

The length of the table name specified in *object_name*.

not_used (IN/OUT)

This parameter is not currently used. Pass as NULL.

pin_duration (IN)

The pin duration. See description in *OCIObjectPin()*.

object (OUT)

The pinned table object.

Comments

This function pins a table object with the specified pin duration.

The client can unpin the object by calling *OCIObjectUnpin()*.

The table object pinned by this call can be passed as a parameter to *OCIObjectNew()* to create a standalone persistent object.

Related Functions

OCIObjectPin(), *OCIObjectUnpin()*

OCIObjectRefresh()

Purpose

Refreshes a persistent object from the most current database snapshot

Syntax

```
sword OCIObjectRefresh ( OCIEnv      *env,  
                        OCIError    *err,  
                        dvoid       *object );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object (IN)

A pointer to the persistent object, which must already be pinned.

Comments

This function refreshes an object with data retrieved from the latest snapshot in the server. An object should be refreshed when the objects in the object cache are inconsistent with the objects at the server.

Note: When an object is flushed to the server, triggers can be fired to modify more objects in the server. The same objects (modified by the triggers) in the object cache become out-of-date, and must be refreshed before they can be locked or flushed.

This occurs when the user issues a SQL statement or PL/SQL procedure to modify any object in the server.

Warning: Modifications made to objects (dirty objects) since the last flush are lost if object are refreshed by this function.

The various meta-attribute flags and durations of an object are modified as followed after being refreshed:

Object Attribute	Status After Refresh
existent	set to appropriate value
pinned	unchanged
allocation duration	unchanged
pin duration	unchanged

The object that is refreshed will be “replaced-in-place”. When an object is replaced-in-place, the top-level memory of the object will be reused so that new data can be loaded into the same memory address. The top level memory of the null structure is also reused. Unlike the top-level memory chunk, the secondary memory chunks will be freed and reallocated. The client should be careful when holding on to a pointer to the secondary memory chunk (e.g. assigning the address of a secondary memory to a local variable), since this pointer can become invalid after the object is refreshed.

This function does nothing for transient objects or values.

Related Functions

OCICacheRefresh()

OCIObjectSetAttr()

Purpose

Set an object attribute.

Syntax

```

sword OCIObjectSetAttr ( OCIEnv          *env,
                        OCIError        *err,
                        dvoid           *instance,
                        dvoid           *null_struct,
                        struct OCIType   *tdo,
                        CONST text      **names,
                        CONST ub4       *lengths,
                        CONST ub4       name_count,
                        CONST ub4       *indexes,
                        CONST ub4       index_count,
                        CONST OCIInd     null_status,
                        CONST dvoid      *attr_null_struct,
                        CONST dvoid      *attr_value );

```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

instance (IN)

Pointer to an object instance.

null_struct (IN)

The null structure of the object instance or array.

tdo (IN)

Pointer to the TDO.

names (IN)

Array of attribute names. This is used to specify the names of the attributes in the path expression.

lengths (IN)

Array of lengths of attribute names.

name_count (IN)

Number of element in the array *names*.

indexes (IN) [optional]

Not currently supported. Pass as (ub4 *)0.

index_count (IN) [optional]

Not currently supported. Pass as (ub4)0.

attr_null_status (IN)

The null status of the attribute if the type of attribute is primitive.

attr_null_struct (IN)

The null structure of an object or collection attribute.

attr_value (IN)

Pointer to the attribute value.

Comments

This function sets the attribute of the given object with the given value. The position of the attribute is specified as a path expression which is an array of names and an array of indexes.

Example

For the path expression `stanford.cs.stu[5].addr`, the arrays will look like:

```
names = {"stanford", "cs", "stu", "addr"}
```

```
lengths = {8, 2, 3, 4}
```

```
indexes = {5}
```

Related Functions

OCIObjectGetAttr()

OCIObjectUnmark()

Purpose

Unmarks an object as dirty.

Syntax

```
sword OCIObjectUnmark ( OCIEnv      *env,  
                        OCIError    *err,  
                        dvoid       *object );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object (IN)

Pointer to the persistent object. It must be pinned.

Comments

For Persistent Objects and Transient Objects

This function unmarks the specified persistent object as dirty. Changes that are made to the object will not be written to the server. If the object is marked locked, it remains marked locked. The changes that have already made to the object will not be undone implicitly.

For Values

This function is an no-op for values. This means that the function will have no effect if called on a value.

Related Functions

OCIObjectUnmarkByRef()

OCIObjectUnmarkByRef()

Purpose

Unmarks an object as dirty, given a REF to the object.

Syntax

```
sword OCIObjectUnmarkByRef ( OCIEnv      *env,  
                             OCIError    *err,  
                             OCIRef      *ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

ref (IN)

Reference of the object. It must be pinned.

Comments

This function unmarks an object as dirty. This function is identical to *OCIObjectUnmark()*, except that it takes a REF to the object as an argument.

For Persistent Objects and Transient Objects

This function unmarks the specified persistent object as dirty. Changes that are made to the object will not be written to the server. If the object is marked locked, it remains marked locked. The changes that have already made to the object will not be undone implicitly.

For Values

This function is a no-op for values.

Related Functions

OCIObjectUnmark()

OCIObjectUnpin()

Purpose

Unpins an object

Syntax

```
sword OCIObjectUnpin ( OCIEnv      *env,  
                      OCIError    *err,  
                      dvoid        *object );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

object (IN)

A pointer to an object, which must already be pinned.

Comments

There is a pin count associated with each object which is incremented whenever an object is pinned. When the pin count of the object is zero, the object is said to be completely unpinned. An unpinned object can be freed implicitly by the OCI at any time without error.

This function unpins an object. An object is completely unpinned when any of the following is true:

1. The object's pin count reaches zero (i.e., it is unpinned a total of N times after being pinned a total of N times).
2. It is the end of the object's pin duration.
3. The function *OCIObjectPinCountReset()* is called on the object.

When an object is completely unpinned, it can be freed implicitly by the OCI at any time without error.

The following rules apply for unpinning different types of objects:

For Persistent Objects

When a persistent object is completely unpinned, it becomes a candidate for aging. The memory of an object is freed when it is aged out. Aging is used to maximize the utilization of memory. An dirty object cannot be aged out unless it is flushed.

For Transient Objects

The pin count of the object is decremented. A transient can be freed only at the end of its allocation duration or when it is explicitly deleted by calling *OCIObjectFree()*.

For Values

This function returns an error for values.

Related Functions

OCIObjectPin(), *OCIObjectPinCountReset()*

OCITypeArrayByName()

Purpose

Get an array of types given an array of names.

Syntax

```
sword OCITypeArrayByName ( OCIEnv          *envhp,
                           OCIError        *errhp,
                           CONST OCISvcCtx *svc,
                           ub4             array_len,
                           CONST text      *schema_name[],
                           ub4             s_length[],
                           CONST text      *type_name[],
                           ub4             t_length[],
                           CONST text      *version_name[],
                           ub4             v_length[],
                           OCIDuration     pin_duration,
                           OCITypeGetOpt   get_option,
                           OCIType        *tdo[] );
```

Parameters

envhp (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

errhp (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

OCI service handle.

array_len (IN)

Number of *schema_name/type_name/version_name* entries to be retrieved.

schema_name (IN, optional)

Array of schema names associated with the types to be retrieved. The array must have *array_len* elements if specified. If 0 is supplied, the default schema is assumed,

otherwise it MUST have *array_len* number of elements. 0 can be supplied for one or more of the entries to indicate that the default schema is desired for those entries.

s_length (IN)

Array of *schema_name* lengths with each entry corresponding to the length of the corresponding *schema_name* entry in the *schema_name* array in bytes. The array must either have *array_len* number of elements or it MUST be 0 if *schema_name* is not specified.

type_name (IN)

Array of the names of the types to retrieve. This MUST have *array_len* number of elements.

t_length (IN)

Array of the lengths of type names in the *type_name* array in bytes.

version_name (IN)

Array of the version names of the types to retrieve corresponding. This can be 0 to indicate retrieval of the most current versions, or it MUST have *array_len* number of elements.

If 0 is supplied, the most current version is assumed, otherwise it MUST have *array_len* number of elements. 0 can be supplied for one or more of the entries to indicate that the current version is desired for those entries.

Note: In release 8.0 the version parameters are ignored.

v_length (IN)

Array of the lengths of version names in the *version_name* array in bytes.

Note: In release 8.0 the version parameters are ignored.

pin_duration (IN)

Pin duration (e.g. until the end of current transaction) for the types retrieved. See *oro.h* for a description of each option.

get_option (IN)

Options for loading the types. It can be one of two values:

- OCI_TYPEGET_HEADER - for only the header to be loaded, or
- OCI_TYPEGET_ALL - for the TDO and all ADO and MDOs to be loaded.

tdo (OUT)

Output array for the pointers to each pinned type in the object cache. It must have space for *array_len* pointers. Use *OCIObjectGetObjectRef()* to obtain the CREF to each pinned type descriptor.

Comments

Gets pointers to the existing types associated with the schema/type name array.

The *get_option* parameter can be used to control the portion of the TDO that gets loaded per roundtrip.

This function returns an error if

- any of the required parameters is null.
- one or more object types associated with a schema/type name entry do not exist.

To retrieve a single type, rather than an array, use *OCITypeByName()*.

Related Functions

OCITypeArrayByRef(), *OCITypeByName()*, *OCITypeByRef()*

OCITypeArrayByRef()

Purpose

Get an array of types given an array of references.

Syntax

```
sword OCITypeArrayByRef ( OCIEnv          *envhp,  
                          OCIError        *errhp,  
                          ub4             array_len,  
                          CONST OCIRef    *type_ref[],  
                          OCIDuration     pin_duration,  
                          OCITypeGetOpt   get_option,  
                          OCIType        *tdo[] );
```

Parameters

envhp (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

errhp (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

array_len (IN)

Number of schema_name/type_name/version_name entries to be retrieved.

type_ref (IN)

Array of **OCIRef** * pointing to the particular version of the type descriptor object to obtain. The array must have *array_len* elements if specified.

pin_duration (IN)

Pin duration (e.g. until the end of current transaction) for the types retrieved. See *oro.h* for a description of each option.

get_option (IN)

Options for loading the types. It can be one of two values:

- OCI_TYPEGET_HEADER - for only the header to be loaded
- OCI_TYPEGET_ALL - for the TDO and all ADO and MDOs to be loaded.

tdo (OUT)

Output array for the pointers to each pinned type in the object cache. It must have space for *array_len* pointers. Use *OCIObjectGetObjectRef()* to obtain the CREF to each pinned type descriptor.

Comments

Gets pointers to the with the schema/type name array.

This function returns an error if:

- any of the required parameters is null.
- one or more object types associated with a schema/type name entry does not exist.

To retrieve a single type, rather than an array of types, use *OCITypeByName()*.

Related Functions

OCITypeArrayByName(), *OCITypeByRef()*, *OCITypeByName()*

OCITypeByName()

Name

OCI Get Existing Type By Name

Purpose

Get the most current version of an existing type by name.

Syntax

```
sword OCITypeByName ( OCIEnv          *env,  
                      OCIError       *err,  
                      CONST OCISvcCtx *svc,  
                      CONST text      *schema_name,  
                      ub4             s_length,  
                      CONST text      *type_name,  
                      ub4             t_length,  
                      CONST text      *version_name,  
                      ub4             v_length,  
                      OCIDuration     pin_duration,  
                      OCITypeGetOpt   get_option  
                      OCIType        **tdo );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

OCI service handle.

schema_name (IN, optional)

Name of schema associated with the type. By default, the user's schema name is used.

s_length (IN)

Length of the *schema_name* parameter.

type_name (IN)

Name of the type to get.

t_length (IN)

Length of the *type_name* parameter.

version_name (IN, optional)

User-readable version of the type. Pass as (text *) 0 to retrieve the most current version. For release 8.0 only a single version is supported.

v_length (IN)

Length of *version_name* in bytes. Pass as 0 if the most current version is to be retrieved.

pin_duration (IN)

Pin duration. Refer to the section “Object Duration” on page 11-13 for more information.

get_option ((IN)

Options for loading the types. It can be one of two values:

- OCI_TYPEGET_HEADER for only the header to be loaded, or
- OCI_TYPEGET_ALL for the TDO and all ADO and MDOs to be loaded.

tdo (OUT)

Pointer to the pinned type in the object cache.

Comments

Gets a pointer to the existing type associated with schema/type name.

This function returns an error if any of the required parameters is NULL, or if the object type associated with schema/type name does not exist.

Note: Schema and type names are CASE-SENSITIVE. If they have been created via SQL, you need to use uppercase names.

An application can retrieve an array of TDOs by calling *OCITypeArrayByName()*, or *OCITypeArrayByRef()*.

See Also

OCITypeByRef(), OCITypeArrayByName(), OCITypeArrayByRef()

OCITypeByRef()

Name

OCI Type By Reference

Purpose

Get a type given a reference.

Syntax

```
sword OCITypeByRef ( OCIEnv          *env,
                    OCIError        *err,
                    CONST OCISRef    *type_ref,
                    OCIDuration     pin_duration,
                    OCITypeGetOpt    get_option,
                    OCIType         *tdo );
```

Comments

Gets a pointer to a type given a REF. This is similar to *OCITypeByName()*.

This function returns an error if

- any of the required parameters is null.
- one or more object types associated with a schema/type name entry does not exist.

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

type_ref (IN)

An **OCISRef *** pointing to the particular version of the type descriptor object to obtain.

pin_duration (IN)

Pin duration (e.g. until the end of current transaction) for the type to retrieve. See `oro.h` for a description of each option.

get_option (IN)

Options for loading the type. It can be one of two values:

- `OCI_TYPEGET_HEADER` - for only the header to be loaded, or
- `OCI_TYPEGET_ALL` - for the TDO and all ADO and MDOs to be loaded.

tdo (OUT)

Pointer to the pinned type in the object cache.

See Also

OCITypeByName(), OCITypeArrayByName(), OCITypeArrayByRef()

OCI Datatype Mapping and Manipulation Functions

This chapter describes the OCI datatype mapping and manipulation functions, which is Oracle's external C Language interface to Oracle8 predefined types.

The following sections are included in this chapter:

- Introduction
- OCI Datatype Mapping Functions Quick Reference
- The OCI Datatype Mapping and Manipulation Functions

Note: The functions described in this chapter are only available if you have purchased the Oracle8 Enterprise Edition with the Objects Option.

Introduction

This chapter describes the OCI datatype mapping and manipulation functions in detail.

See Also: For more information about the functions listed in this chapter, refer to Chapter 9, “Object-Relational Datatypes”.

Datatype Mapping and Manipulation Function Return Values

The OCI datatype mapping and manipulation functions typically return one of the following values:

Table 15–1 Function Return Values

Return Value	Meaning
OCI_SUCCESS	The operation succeeded
OCI_ERROR	The operation failed. The specific error can be retrieved by calling <i>OCIErrorGet()</i> on the error handle passed to the function.
OCI_INVALID_HANDLE	The environment or error handle passed to the function is NULL.

Function-specific return information follows the description of each function in this chapter. For more information about return codes and error handling, see the section “Error Handling” on page 2-25.

Functions Returning Other Values

Some functions return values other than those listed in Table 15–1. When using these functions be sure to take into account that they return a value directly from the function call, rather than through an OUT parameter.

- *OCICollMax()*
- *OCIRawPtr()*
- *OCIRawSize()*
- *OCISqlHexSize()*
- *OCISqlsEqual()*
- *OCISqlsNull()*
- *OCISqlStringPtr()*
- *OCISqlStringSize()*

Server Roundtrips for Datatype Mapping and Manipulation Functions

For a table showing the number of server roundtrips required for individual OCI datatype mapping and manipulation functions, refer to Appendix E, “OCI Function Server Roundtrips”.

Examples

For more information about these functions, including some code examples, refer to Chapter 9, “Object-Relational Datatypes”.

OCI Datatype Mapping Functions Quick Reference

This section is intended to help you figure out which function you need to use in a given situation.

Table 15–2 OCI Datatype Mapping and Manipulation Functions Quick Reference

Function	Purpose	Page
COLLECTION ITERATOR FUNCTIONS		
OCICollAppend()	Collection append element	15 - 9
OCICollAssignElem()	Collection assign element	15 - 13
OCICollAssign()	Assign collection	15 - 11
OCICollSize()	Get current size of collection (in number of elements)	15 - 19
OCICollTrim()	Trim elements from the collection	15 - 21
OCICollGetElem()	Get pointer to an element	15 - 15
OCICollMax()	Return maximum number of elements in collection	15 - 18
OCIIterCreate()	Create iterator to scan the varray elements	15 - 43
OCIIterGetCurrent()	Get current collection element	15 - 46
OCIIterDelete()	Delete iterator	15 - 45
OCIIterInit()	Initialize iterator to scan the given collection	15 - 47
OCIIterNext()	Get next collection element	15 - 48
OCIIterPrev()	Get previous collection element,	15 - 50
DATE FUNCTIONS		
OCIDateToText()	Convert date to String	15 - 39
OCIDateAddDays()	Add or subtract days	15 - 22
OCIDateAddMonths()	Add or subtract months	15 - 23
OCIDateDaysBetween()	Get number of days between two dates	15 - 28
OCIDateCheck()	Check if the given date is valid	15 - 25
OCIDateCompare()	Compare dates	15 - 27
OCIDateLastDay()	Get date of last day of month	15 - 33
OCIDateNextDay()	get date of next day	15 - 34

Table 15–2 OCI Datatype Mapping and Manipulation Functions Quick Reference

Function	Purpose	Page
OCIDateFromText ()	Convert string to date	15 - 29
OCIDateSysDate()	Get current system date and time	15 - 38
OCIDateZoneToZone()	Convert date from one time zone to another zone	15 - 41
OCIDateAssign()	Assign date	15 - 24
OCIDateGetDate()	Get the date portion of a date	15 - 31
OCIDateGetTime()	Get the time portion of a date	15 - 32
OCIDateSetDate()	Set the date portion of a date	15 - 36
OCIDateSetTime()	Set the time portion of a date	15 - 37
NUMBER FUNCTIONS		
OCINumberToInt()	Convert number to integer	15 - 88
OCINumberToReal()	Convert number to real	15 - 90
OCINumberToText()	Convert number to string	15 - 91
OCINumberAbs()	Absolute value	15 - 52
OCINumberArcCos()	Arc cosine	15 - 54
OCINumberAdd()	Add numbers	15 - 53
OCINumberAssign()	Assign number	15 - 58
OCINumberArcSin ()	Arc sine	15 - 55
OCINumberArcTan()	Arc tangent	15 - 56
OCINumberArcTan2()	Arc tangent 2	15 - 57
OCINumberExp()	Arbitrary base exponentiation	15 - 80
OCINumberCeil()	Ceiling of number	15 - 70
OCINumberCmp()	Compare numbers	15 - 60
OCINumberCos()	Cosine	15 - 61
OCINumberHypCos()	Cosine hyperbolic	15 - 70
OCINumberDiv()	Divide numbers	15 - 62
OCINumberPower()	Exponentiation to base e	15 - 80
OCINumberFloor()	Floor of number	15 - 64
OCINumberFromInt()	Convert integer to number	15 - 65

Table 15–2 OCI Datatype Mapping and Manipulation Functions Quick Reference

Function	Purpose	Page
OCINumberIsZero()	Comparison with zero	15 - 74
OCINumberLn()	Logarithm natural	15 - 75
OCINumberLog ()	Logarithm to arbitrary base	15 - 76
OCINumberMod()	Modulo division	15 - 77
OCINumberMul()	Multiply numbers	15 - 78
OCINumberNeg()	Negate number	15 - 79
OCINumberIntPower()	Take an arbitrary base to an arbitrary integer power	15 - 63
OCINumberFromReal()	Convert real to number	15 - 67
OCINumberRound()	Round Oracle number to a specified decimal place	15 - 81
OCINumberSetZero()	Initialize number to zero	15 - 82
OCINumberFromText()	Convert string to number	15 - 68
OCINumberSign()	Obtain sign of an Oracle number	15 - 83
OCINumberSin()	Sine	15 - 84
OCINumberHypSin()	Sine Hyperbolic	15 - 71
OCINumberSqrt()	Square root of number	15 - 85
OCINumberSub()	Subtract numbers	15 - 86
OCINumberTan()	Tangent	15 - 87
OCINumberHypTan()	Tangent hyperbolic	15 - 72
OCINumberTrunc()	Truncate an Oracle number at a specified decimal place	15 - 93
REF FUNCTIONS		
OCIRefToHex()	Convert REF to hexadecimal string	15 - 107
OCIRefAssign()	Assign one REF to another	15 - 100
OCIRefClear()	Clear or nullify a REF	15 - 101
OCIRefsEqual()	Compare two REFs for equality	15 - 105
OCIRefFromHex()	Convert hexadecimal string to REF	15 - 102
OCIRefHexSize()	Return size of hexadecimal representation of REF	15 - 104
OCIRefsIsNull()	Test if a REF is NULL	15 - 106

Table 15–2 OCI Datatype Mapping and Manipulation Functions Quick Reference

Function	Purpose	Page
TABLE FUNCTIONS		
OCITableDelete()	Delete element	15 - 115
OCITableExists()	Test whether element exists	15 - 116
OCITableFirst()	Return first index of table	15 - 117
OCITableLast()	Return last index of table	15 - 118
OCITableNext()	Return next available index of table	15 - 119
OCITablePrev()	Return previous available index of table	15 - 121
OCITableSize()	Return current size of table	15 - 123
STRING FUNCTIONS		
OCIStrAssign()	Assign string to string	15 - 110
OCIStrAllocSize()	Get allocated size of string memory in bytes	15 - 109
OCIStrAssignText()	Assign text string to string	15 - 111
OCIStrPtr()	Get string pointer	15 - 112
OCIStrSize()	Get string size	15 - 114
OCIStrResize()	Resize string memory	15 - 113
RAW FUNCTIONS		
OCIRawAssignBytes()	Assign raw bytes to raw	15 - 95
OCIRawAssignRaw()	Assign raw to raw	15 - 96
OCIRawAllocSize()	Get allocated size of raw memory in bytes	15 - 94
OCIRawPtr()	Get raw data Pointer	15 - 97
OCIRawSize()	Get raw size	15 - 99
OCIRawResize()	Resize memory of variable-length raw	15 - 98

The OCI Datatype Mapping and Manipulation Functions

This chapter describes the OCI datatype mapping and manipulation functions. The entries for each function contain the following information:

Purpose

A brief statement of the purpose of the function.

Syntax

A code snippet showing the syntax for calling the function, including the ordering and types of the parameters.

Comments

Detailed information about the function (if available). This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below:

Mode	Description
IN	A parameter that passes data to Oracle
OUT	A parameter that receives data from Oracle on this or a subsequent call
IN/OUT	A parameter that passes data on the call and receives data on the return from this or a subsequent call.

Returns

A description of what value is returned by the function if the function returns something other than the standard return codes listed in the table above.

Related Functions

A list of related functions.

OCICollAppend()

Purpose

Appends an element to a collection

Syntax

```
sword OCICollAppend ( OCIEnv          *env,
                      OCIError        *err,
                      CONST dvoid      *elem,
                      CONST dvoid      *elemind,
                      OCIColl          *coll );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

elem (IN)

Pointer to the element which is appended to the end of the given collection.

elemind (IN) [optional]

Pointer to the element's null indicator information; if (*elemind* == NULL) then the null indicator information of the appended element will be set to non-null.

coll (IN/OUT)

Updated collection.

Comments

Appends the given element to the end of the given collection.

Appending an element is equivalent to:

- increasing the size of the collection by 1 element
- updating (deep-copying) the last element's data with the given element's data

Note that the pointer to the given element *elem* will not be saved by this function. So *elem* is strictly an input parameter. This function returns an error if the current size of the collection is equal to the max size (upper-bound) of the collection prior to appending the element.

This function returns an error if any of the input parameters is NULL.

Related Functions

OCIErrorGet()

OCICollAssign()

Purpose

Assigns (deep-copies) one collection to another

Syntax

```
sword OCICollAssign ( OCIEnv          *env,  
                     OCIError        *err,  
                     CONST OCIColl    *rhs,  
                     OCIColl          *lhs );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

rhs (IN)

Right-hand side (source) collection to be assigned from.

lhs (OUT)

Left-hand side (target) collection to be assigned to.

Comments

Assigns *rhs* (source) to *lhs* (target). The *lhs* collection may be decreased or increased depending upon the size of *rhs*. If the *lhs* contains any elements then the elements will be deleted prior to the assignment. This function performs a deep copy. The memory for the elements comes from the object cache.

An error is returned if the element types of the *lhs* and *rhs* collections do not match. Also, an error is returned if the upper-bound of the *lhs* collection is less than the current number of elements in the *rhs* collection.

This function returns an error if:

- any of the input parameters is NULL
- there is a type mismatch between the *lhs* and *rhs* collections
- the upper bound of *lhs* collection is less than the current number of elements in the *rhs* collection

Related Functions

OCIErrorGet()

OCICollAssignElem()

Purpose

Assign an element to a collection

Syntax

```
sword OCICollAssignElem ( OCIEnv          *env,  
                          OCIError        *err,  
                          sb4             index,  
                          CONST dvoid     *elem,  
                          CONST dvoid     *elemind,  
                          OCIColl         *coll );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

index (IN)

Index of the element whose is assigned to.

elem (IN)

Element which is assigned from (source element).

elemind (IN) [optional]

Pointer to the element's null indicator information; if (*elemind* == NULL) then the null indicator information of the assigned element will be set to non-null.

coll (IN/OUT)

Collection to be updated.

Comments

Assigns the given element value *elem* to the element at *coll*[*index*].

If the collection is of type nested table, the element at the given index may not exist (i.e. may have been deleted). In this case, the given element is inserted at index *index*. Otherwise, the element at index *index* is updated with the value of *elem*.

Note that the given element is deep-copied and *elem* is strictly an input parameter.

This function returns an error if any input parameter is NULL or if the given index is beyond the bounds of the given collection.

Related Functions

OCIErrorGet()

OCICollGetElem()

Purpose

Gets a pointer to the element at the given index

Syntax

```
sword OCICollGetElem ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCIColl    *coll,
                      sb4              index,
                      boolean          *exists,
                      dvoid            **elem,
                      dvoid            **elemind );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

coll (IN)

Pointer to the element in this collection is returned.

index (IN)

Index of the element whose pointer is returned.

exists (OUT)

Set to *FALSE* if the element at the specified index does not exist; otherwise, set to *TRUE*.

elem (OUT)

Address of the desired element is returned.

elemind (OUT) [optional]

Address of the null indicator information is returned; if (*elemind* == *NULL*) then the null indicator information will NOT be returned.

Comments

Gets the address of the element at the given position. Optionally this function also returns the address of the element's null indicator information.

The following table describes for each collection element type what the corresponding element pointer type is. The element pointer is returned via the *elem* parameter of *OCICollGetElem()*.

Element Type	*elem is set to
Oracle Number (OCINumber)	OCINumber*
Date (OCIDate)	OCIDate*
Variable-length string (OCIStrng*)	OCIStrng**
Variable-length raw (OCIRaw*)	OCIRaw**
object reference (OCIStrng*)	OCIStrng**
lob locator (OCILobLocator*)	OCILobLocator**
object type (e.g. person)	person*

The element pointer returned by *OCICollGetElem()* is in a form such that it can not only be used to access the element data but also is in a form that can be used as the target (i.e., left-hand-side) of an assignment statement.

For example, assume the user is iterating over the elements of a collection whose element type is object reference (**OCIStrng***). A call to *OCICollGetElem()* returns pointer to a reference handle (i.e. **OCIStrng****). After getting, the pointer to the collection element, the user may wish to modify it by assigning a new reference.

This can be accomplished via the ref assignment function shown below:

```
sword OCIStrngAssign( OCIStrng *env, OCIStrng *err, CONST OCIStrng *source,
    OCIStrng **target );
```

Note that the *target* parameter of *OCIStrngAssign()* is of type **OCIStrng****. Hence *OCICollGetElem()* returns **OCIStrng****. If **target* equals NULL, a new REF will be allocated by *OCIStrngAssign()* and returned via the *target* parameter.

Similarly, if the collection element was of type string (**OCIStrng***), *OCICollGetElem()* returns pointer to string handle (i.e. **OCIStrng****). If a new string is assigned, via *OCIStrngAssign()* or *OCIStrngAssignText()* the type of the target must be **OCIStrng****.

If the collection element is of type Oracle number, *OCICollGetElem()* returns **OCINumber***. The prototype of *OCINumberAssign()* is shown below:

```
sword OCINumberAssign(OCIError *err, CONST OCINumber *from,  
                     OCINumber *to);
```

This function returns an error if any of the input parameters is NULL.

Related Functions

OCIErrorGet()

OCICollMax()

Purpose

Gets the maximum size (in number of elements) of the given collection

Syntax

```
sb4 OCICollMax ( OCIEnv          *env,  
                  CONST OCIColl    *coll );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

coll (IN)

Collection whose number of elements is returned. *coll* must point to a valid collection descriptor.

Comments

Returns the maximum number of elements that the given collection can hold. A value of zero indicates that the collection has no upper bound.

Returns

the upper bound of the given collection

Related Functions

OCIErrorGet()

OCICollSize()

Purpose

Gets the current size (in number of elements) of the given collection

Syntax

```
sword OCICollSize ( OCIEnv          *env,
                    OCLError        *err,
                    CONST OCIColl    *coll
                    sb4              *size );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCLErrorGet()*.

coll (IN)

Collection whose number of elements is returned. Must point to a valid collection descriptor.

size (OUT)

Current number of elements in the collection.

Comments

Returns the current number of elements in the given collection.

For the case of nested table, this count will NOT be decremented upon deleting elements. So, this count includes any “holes” created by deleting elements. A trim operation (*OCICollTrim()*) will decrement the count by the number of trimmed elements. To get the count minus the deleted elements use *OCITableSize()*.

The following pseudocode shows some examples:

```
OCICollSize(...);  
// assume 'size' returned is equal to 5  
OCITableDelete(...); // delete one element  
OCICollSize(...);  
// 'size' returned is still 5
```

To get the count minus the deleted elements use *OCITableSize()*. Continuing the above example:

```
OCITableSize(...)  
// 'size' returned is equal to 4
```

A trim operation (*OCICollTrim()*) decrements the count by the number of trimmed elements. Continuing the above example:

```
OCICollTrim(...,1..); // trim one element  
OCICollSize(...);  
// 'size' returned is equal to 4
```

This function returns an error if an error occurs during the loading of the collection into object cache or if any of the input parameters is null.

Related Functions

OCIErrorGet()

OCI CollTrim()

Purpose

Trims the given number of elements from the end of the collection

Syntax

```

sword OCICollTrim ( OCIEnv      *env,
                   OCIError    *err,
                   sb4          trim_num,
                   OCIColl     *coll );

```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

trim_num (IN)

Number of elements to trim.

coll (IN/OUT)

This function removes (frees) *trim_num* elements from the end of *coll*.

Comments

Trim the collection by the given number of elements. The elements are removed from the end of the collection. An error is returned if *trim_num* is greater than the current size of the collection.

This function returns an error if *trim_num* is greater than the current size of the collection.

Related Functions

OCIErrorGet()

OCIDateAddDays()

Purpose

Adds or subtracts days from a given date.

Syntax

```
sword OCIDateAddDays ( OCIError          *err,  
                       CONST OCIDate      *date,  
                       sb4                num_days,  
                       OCIDate            *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

date (IN)

This function adds or subtracts *num_days* from *date*.

num_days (IN)

Number of days to be added or subtracted (a negative value will be subtracted).

result (IN/OUT)

Result of adding days to, or subtracting days from, *date*.

Comments

Adds or subtracts *num_days* from the date *date*.

This function returns and error if an invalid date is passed to it.

Related Functions

OCIErrorGet()

OCIDateAddMonths()

Purpose

Adds or subtracts months from a given date.

Syntax

```
sword OCIDateAddMonths ( OCIError          *err,  
                        CONST OCIDate      *date,  
                        sb4                 num_months,  
                        OCIDate            *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

date (IN)

This function adds or subtracts *num_months* from *date*.

num_months (IN)

Number of months to be added or subtracted (a negative value is subtracted).

result (IN/OUT)

Result of adding days to, or subtracting days from, *date*.

Comments

Adds or subtracts *num_months* from the date *date*.

If the input *date* is the last day of a month, then the appropriate adjustments are made to ensure that the output date is also the last day of the month. For example, Feb. 28 + 1 month = March 31, and November 303 months = August 31. Otherwise the *result* date has the same day component as *date*.

This function returns an error if invalid date is passed to it.

Related Functions

OCIErrorGet()

OCIDateAssign()

Purpose

Performs date assignment

Syntax

```
sword OCIDateAssign ( OCLError      *err,
                     CONST OCIDate  *from,
                     OCIDate        *to );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

from (IN)

Date to be assigned.

to (OUT)

Target of assignment.

Comments

This function assigns a value from one **OCIDate** variable to another.

Related Functions

OCLErrorGet()

OCIDateCheck()

Purpose

Checks if the given date is valid.

Syntax

```
sword OCIDateCheck ( OCLError      *err,
                    CONST OCIDate  *date,
                    uword          *valid );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

date (IN)

Date to be checked

valid (OUT)

Returns zero for a valid date, otherwise the ORed combination of all error bits specified below:

Macro name	Bit number	Error
-----	-----	-----
OCI_DATE_INVALID_DAY	0x1	Bad day
OCI_DATE_DAY_BELOW_VALID	0x2	Bad day low/high bit (1=low)
OCI_DATE_INVALID_MONTH	0x4	Bad month
OCI_DATE_MONTH_BELOW_VALID	0x8	Bad month low/high bit (1=low)
OCI_DATE_INVALID_YEAR	0x10	Bad year
OCI_DATE_YEAR_BELOW_VALID	0x20	Bad year low/high bit (1=low)
OCI_DATE_INVALID_HOUR	0x40	Bad hour
OCI_DATE_HOUR_BELOW_VALID	0x80	Bad hour low/high bit (1=low)
OCI_DATE_INVALID_MINUTE	0x100	Bad minute
OCI_DATE_MINUTE_BELOW_VALID	0x200	Bad minute Low/high bit (1=low)
OCI_DATE_INVALID_SECOND	0x400	Bad second
OCI_DATE_SECOND_BELOW_VALID	0x800	Bad second Low/high bit (1=low)
OCI_DATE_DAY_MISSING_FROM_1582	0x1000	Day is one of those "missing" from 1582
OCI_DATE_YEAR_ZERO	0x2000	Year may not equal zero
OCI_DATE_INVALID_FORMAT	0x8000	Bad date format input

So, for example, if the date passed in was 2/0/1990 25:61:10 in (month/day/year hours:minutes:seconds format), the error returned would be
OCI_DATE_INVALID_DAY | OCI_DATE_DAY_BELOW_VALID |
OCI_DATE_INVALID_HOUR | OCI_DATE_INVALID_MINUTE

Comments

Checks if the given date is valid.

This function returns an error if *date* or *valid* pointer is NULL.

Related Functions

OCIErrorGet()

OCIDateCompare()

Purpose

Compares two dates.

Syntax

```
sword OCIDateCompare ( OCIError          *err,  
                        CONST OCIDate     *date1,  
                        CONST OCIDate     *date2,  
                        sword              *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

date1, date2 (IN)

Dates to be compared.

result (OUT)

Comparison result:

Comparison result	Output in <i>result</i> parameter
date1 < date2	-1
date1 = date2	0
date1 > date2	1

Comments

Compares two dates.

This function returns an error if an invalid date is passed to it.

Related Functions

OCIErrorGet()

OCIDateDaysBetween()

Purpose

Gets the number of days between two dates

Syntax

```
sword OCIDateDaysBetween ( OCIError          *err,  
                           CONST OCIDate      *date1,  
                           CONST OCIDate      *date2,  
                           sb4                *num_days );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

date1 (IN)

Input date.

date2 (IN)

Input date.

num_days (OUT)

Number of days between *date1* and *date2*.

Comments

Returns the number of days between *date1* and *date2*. The time is ignored in this computation.

This function returns an error if invalid date is passed to it.

Related Functions

OCIErrorGet()

OCIDateFromText()

Purpose

Converts a character string to a date type.

Syntax

```
sword OCIDateFromText ( OCIError          *err,
                        CONST text        *date_str,
                        ub4               d_str_length,
                        CONST text        *fmt,
                        ub1               fmt_length,
                        CONST text        *lang_name,
                        ub4               lang_length,
                        OCIDate           *date );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

date_str (IN)

Input string to be converted to Oracle date.

d_str_length (IN)

Size of the input string, if the length is -1 then *date_str* is treated as a NULL terminated string.

fmt (IN)

Conversion format. If *fmt* is a null pointer, then the string is expected to be in 'DD-MON-YY' format.

fmt_length (IN)

Length of the *fmt* parameter.

lang_name (IN)

Language in which the names and abbreviations of days and months are specified. If *lang_name* is a NULL string, (text *) 0, then the default language of the session is used.

lang_length (IN)

Length of the *lang_name* parameter.

date (OUT)

Given string converted to date.

Comments

Converts the given string to Oracle date according to the specified format.

Refer to the TO_DATE conversion function described in Chapter 3 of the *Oracle8 SQL Reference* for a description of format and NLS arguments.

This function returns an error if it receives an invalid format, language, or input string.

Related Functions

OCIErrorGet()

OCIDateGetDate()

Purpose

Get the year, month, and day stored in an Oracle date.

Syntax

```
void OCIDateGetDate ( CONST OCIDate      *date,  
                      sb2                *year,  
                      ub1                *month,  
                      ub1                *day );
```

Parameters

date (IN)

Oracle date whose year, month, day data is retrieved.

year (OUT)

Year value returned.

month (OUT)

Month value returned.

day (OUT)

Day value returned.

Comments

Returns year, month, day information stored in the given date.

Related Functions

OCIDateSetDate()

OCIDateGetTime()

Purpose

Get the time stored in an Oracle date.

Syntax

```
void OCIDateGetTime ( CONST OCIDate      *date,
                      ub1                 *hour,
                      ub1                 *min,
                      ub1                 *sec );
```

Parameters

date (IN)

Oracle date whose time data is retrieved.

hour (OUT)

Hour value returned.

min (OUT)

Minute value returned.

sec (OUT)

Second value returned.

Comments

Returns time information stored in the given date. The time information returned is: hour, minute and seconds.

Related Functions

OCIDateSetTime()

OCIDateLastDay()

Purpose

Gets the date of the last day of the month.

Syntax

```
sword OCIDateLastDay ( OCIError          *err,  
                       CONST OCIDate      *date,  
                       OCIDate            *last_day );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

date (IN)

Input date.

last_day (OUT)

Last day of the month in *date*.

Comments

Returns the date of the last day of the month specified in *date*.

This function returns an error if invalid date is passed to it.

Related Functions

OCIErrorGet()

OCIDateNextDay()

Purpose

Gets the date of next day of the week, after a given date.

Syntax

```
sword OCIDateNextDay ( OCIError          *err,
                       CONST OCIDate      *date,
                       CONST text         *day,
                       ub4                 day_length,
                       OCIDate            *next_day );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

date (IN)

Returned date should be later than this date.

day (IN)

First day of week named by this is returned.

day_length (IN)

Length in bytes of string *day*.

next_day (OUT)

First day of the week named by *day* later than *date*.

Comments

Returns the date of the first day of the week named by *day* that is later than date *date*.

Example

Get the date of the next Monday after April 18, 1996 (a Thursday).

```
OCIDateNextDay(&err, '18-APR-96', 'MONDAY', strlen('MONDAY'), &next_day)
```

OCIDateNextDay() returns '22-APR-96'.

This function returns an error if an invalid date or day is passed to it.

Related Functions

OCIErrorGet()

OCIDateSetDate()

Purpose

Set the values in an Oracle date.

Syntax

```
void OCIDateSetDate ( OCIDate      *date,
                      sb2          year,
                      ub1          month,
                      ub1          day );
```

Parameters

date (OUT)

Oracle date whose time data is set.

year (IN)

Year value to be set.

month (IN)

Month value to be set.

day (IN)

Day value to be set.

Comments

Sets the date with the given information.

Related Functions

OCIDateGetDate()

OCIDateSetTime()

Purpose

Set the time information in an Oracle date.

Syntax

```
void OCIDateSetTime ( OCIDate      *date,  
                      ubl          hour,  
                      ubl          min,  
                      ubl          sec );
```

Parameters

date (OUT)

Oracle date whose time data is set.

hour (IN)

Hour value to be set.

min (IN)

Minute value to be set.

sec (IN)

Second value to be set.

Comments

Sets the date with the given time information.

Related Functions

OCIDateGetTime()

OCIDateSysDate()

Purpose

Gets current system date and time.

Syntax

```
sword OCIDateSysDate ( OCIError      *err,
                      OCIDate      *sys_date );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

sys_date (OUT)

Current system date and time.

Comments

Returns the current system date and time.

Related Functions

OCLErrorGet()

OCIDateToText()

Purpose

Converts a date type to a character string.

Syntax

```
sword OCIDateToText ( OCLError          *err,
                      CONST OCIDate      *date,
                      CONST text          *fmt,
                      ub1                 fmt_length,
                      CONST text          *lang_name,
                      ub4                 lang_length,
                      ub4                 *buf_size,
                      text                 *buf );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

date (IN)

Oracle date to be converted.

fmt (IN)

Conversion format, if NULL string pointer, (*text* *) 0, then the date is converted to a character string in the default date format, "DD-MON-YY".

fmt_length (IN)

Length of the *fmt* parameter.

lang_name (IN)

Specifies the language in which names and abbreviations of months and days are returned; default language of session is used if *lang_name* is NULL ((*text* *) 0).

lang_length (IN)

Length of the *lang_name* parameter.

buf_size (IN/OUT)

- Size of the buffer (IN);
- Size of the resulting string is returned via this parameter(OUT).

buf (OUT)

Buffer into which the converted string is placed.

Comments

Converts the given date to a string according to the specified format. The converted NULL-terminated date string is stored in *buf*.

Refer to the TO_DATE conversion function described in Chapter 3 of the *Oracle8 SQL Reference* for a description of format and NLS arguments.

This function returns an error if the buffer is too small, or if the function is passed an invalid format or unknown language. Overflow also causes an error. For example, converting a value of 10 into format '9' causes an error.

Related Functions

OCIErrorGet()

OCIDateZoneToZone()

Purpose

Converts a date from one time zone to another.

Syntax

```
sword OCIDateZoneToZone ( OCLError          *err,  
                          CONST OCIDate      *date1,  
                          CONST text         *zon1,  
                          ub4                zon1_length,  
                          CONST text         *zon2,  
                          ub4                zon2_length,  
                          OCIDate            *date2 );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

date1 (IN)

Date to convert.

zon1 (IN)

Zone of input date.

zon1_length (IN)

Length in bytes of *zon1*.

zon2 (IN)

Zone to be converted to.

zon2_length (IN)

Length in bytes of *zon2*.

date2 (OUT)

Converted date (in *zon2*).

Comments

Converts date from one time zone to another. Given date *date1* in time zone *zon1*, returns date *date2* in time zone *zon2*.

For a list of valid zone strings, refer to the description of the “NEW_TIME” function in Chapter 3 of the *Oracle8 SQL Reference*. Examples of valid zone strings include:

- “AST”, Atlantic Standard Time
- “ADT”, Atlantic Daylight Time
- “BST”, Bering Standard Time
- “BDT”, Bering Daylight Time

This function returns an error if an invalid date or time zone is passed to it.

Related Functions

OCIErrorGet()

OCIIterCreate()

Purpose

Creates an iterator to scan collection elements.

Syntax

```
sword OCIIterCreate ( OCIEnv          *env,  
                     OCIError        *err,  
                     CONST OCIColl   *coll,  
                     OCIIter         **itr );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

coll (IN)

Collection which will be scanned. For release 8.0, valid collection types include varrays and nested tables.

itr (OUT)

Address to the allocated collection iterator is returned by this function.

Comments

Creates an iterator to scan the elements of the collection. The iterator is created in the object cache. The iterator is initialized to point to the beginning of the collection.

If *OCIIterNext()* is called immediately after creating the iterator then the first element of the collection is returned. If *OCIIterPrev()* is called immediately after creating the iterator then “at beginning of collection” error is returned.

This function returns an error if any of the input parameters is NULL.

Related Functions

OCIErrorGet(), OCIIterDelete()

OCIIterDelete()

Purpose

Deletes a collection iterator.

Syntax

```
sword OCIIterDelete ( OCIEnv          *env,  
                     OCIError       *err,  
                     OCIIter        **itr );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

itr (IN/OUT)

The allocated collection iterator which is destroyed and set to NULL prior to returning.

Comments

Deletes an iterator which was previously created by a call to *OCIIterCreate()*.

This function returns an error if any of the input parameters is null.

Related Functions

OCIErrorGet(), *OCIIterCreate()*

OCIIterGetCurrent()

Purpose

Gets a pointer to the current iterator collection element.

Syntax

```
sword OCIIterGetCurrent ( OCIEnv          *env,
                          OCIError        *err,
                          CONST OCIIter   *itr,
                          dvoid           **elem,
                          dvoid           **elemind );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

itr (IN)

Iterator which points to the current element.

elem (OUT)

Address of the element pointed by the iterator is returned.

elemind (OUT) [optional]

Address of the element's NULL indicator information is returned; if (*elem_ind* == NULL) then the NULL indicator information will *not* be returned.

Comments

Returns pointer to the current iterator collection element and its corresponding NULL information. This function returns an error if any input parameter is NULL.

Related Functions

OCIErrorGet()

OCIIterInit()

Purpose

Initializes an iterator to scan a collection.

Syntax

```
sword OCIIterInit ( OCIEnv           *env,  
                   OCIError        *err,  
                   CONST OCIColl    *coll,  
                   OCIIter          *itr );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

coll (IN)

Collection which will be scanned. For release 8.0, valid collection types include varrays and nested tables.

itr (IN/OUT)

Pointer to an allocated collection iterator.

Comments

Initializes given iterator to point to the beginning of given collection. Returns an error if any input parameter is NULL. This function can be used to:

- reset an iterator to point back to the beginning of the collection, or
- reuse an allocated iterator to scan a different collection.

Related Functions

OCIErrorGet()

OCIIterNext()

Purpose

Gets a pointer to the next iterator collection element.

Syntax

```
sword OCIIterNext ( OCIEnv          *env,  
                   OCIError        *err,  
                   OCIIter         *itr,  
                   dvoid           **elem,  
                   dvoid           **elemind,  
                   boolean          *eoc);
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

itr (IN/OUT)

Iterator is updated to point to the next element.

elem (OUT)

After updating the iterator to point to the next element, address of the element is returned.

elemind (OUT) [optional]

Address of the element's NULL indicator information is returned; if (*elem_ind* == NULL) then the NULL indicator information will *not* be returned.

eoc (OUT)

TRUE if iterator is at End of Collection (i.e. next element does not exist); otherwise, FALSE.

Comments

Returns a pointer to the next iterator collection element and its corresponding NULL information. Updates the iterator to point to the next element.

If the iterator is pointing to the last element of the collection prior to executing this function, then calling this function will set the *eoc* flag to TRUE. The iterator will be left unchanged in that case.

This function returns an error if any input parameter is NULL.

Related Functions

OCIErrorGet(), *OCIIterPrev()*

OCIIterPrev()

Purpose

Gets a pointer to the previous iterator collection element

Syntax

```
sword OCIIterPrev ( OCIEnv          *env,  
                   OCIError        *err,  
                   OCIIter         *itr,  
                   dvoid           **elem,  
                   dvoid           **elemind,  
                   boolean          *boc );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

itr (IN/OUT)

Iterator which is updated to point to the previous element.

elem (OUT)

Address of the previous element; returned after the iterator is updated to point to it.

elemind (OUT) [optional]

Address of the element's NULL indicator; if (*elem_ind* == NULL) then the NULL indicator will *not* be returned.

boc (OUT)

TRUE if iterator is at beginning of collection (i.e. previous element does not exist); otherwise, FALSE.

Comments

Returns pointer to the previous iterator collection element and its corresponding NULL information. The iterator is updated to point to the previous element.

If the iterator is pointing to the first element of the collection prior to executing this function, then calling this function will set *boc* to TRUE. The iterator is left unchanged in that case.

This function returns an error if any input parameter is NULL.

Related Functions

OCIErrorGet(), *OCIIterNext()*

OCINumberAbs()

Purpose

Computes the absolute value of an Oracle number.

Syntax

```
sword OCINumberAbs ( OCIError          *err,
                    CONST OCINumber    *number,
                    OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Input number.

result (OUT)

The absolute value of the input number.

Comments

Computes the absolute value of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberAdd()

Purpose

Adds two Oracle numbers together.

Syntax

```
sword OCINumberAdd ( OCLError          *err,  
                     CONST OCINumber    *number1,  
                     CONST OCINumber    *number2,  
                     OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number1, number2 (IN)

Numbers to be added.

result (OUT)

Result of adding *number1* to *number2*.

Comments

Adds *number1* to *number2* and returns result in *result*.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCLErrorGet()

OCINumberArcCos()

Purpose

Takes the arc cosine of an Oracle number.

Syntax

```
sword OCINumberArcCos ( OCIError          *err,
                        CONST OCINumber    *number,
                        OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Argument of the arc cosine.

result (OUT)

Result of the arc cosine in radians.

Comments

Takes the arc cosine in radians of an Oracle number.

This function returns an error if any of the number arguments is NULL, or if *number* < -1 or if *number* > 1.

Related Functions

OCIErrorGet()

OCINumberArcSin()

Purpose

Takes the arc sine of an Oracle number.

Syntax

```

sword OCINumberArcSin ( OCIError          *err,
                        CONST OCINumber    *number,
                        OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Argument of the arc sine.

result (OUT)

Result of the arc sine in radians.

Comments

Takes the arc sine in radians of an Oracle number.

This function returns an error if any of the number arguments is NULL, or if *number* < -1 or if *number* > 1.

Related Functions

OCIErrorGet()

OCINumberArcTan()

Purpose

Takes the arc tangent of an Oracle number.

Syntax

```

sword OCINumberArcTan ( OCLError          *err,
                        CONST OCINumber    *number,
                        OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number (IN)

Argument of the arc tangent.

result (OUT)

Result of the arc tangent in radians.

Comments

Takes the arc tangent in radians of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCLErrorGet()

OCINumberArcTan2()

Purpose

Takes the arc tangent of two Oracle numbers.

Syntax

```

sword OCINumberArcTan2 ( OCIError          *err,
                        CONST OCINumber     *number1,
                        CONST OCINumber     *number2,
                        OCINumber           *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number1 (IN)

Argument 1 of the arc tangent.

number2 (IN)

Argument 2 of the arc tangent.

result (OUT)

Result of the arc tangent in radians.

Comments

Takes the atan2(*number1*, *number2*).

This function returns an error if any of the number arguments is NULL, or if *number2* = 0.

Related Functions

OCIErrorGet()

OCINumberAssign()

Purpose

Assigns one Oracle number to another Oracle number.

Syntax

```
sword OCINumberAssign ( OCIError          *err,
                        CONST OCINumber    *from,
                        OCINumber          *to );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

from (IN)

Number to be assigned.

to (OUT)

Number copied into.

Comments

Assigns number *from* to *to*.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCLErrorGet()

OCINumberCeil()

Purpose

Computes the ceiling value of an Oracle number.

Syntax

```
sword OCINumberCeil ( OCIError          *err,  
                     CONST OCINumber    *number,  
                     OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Input number.

result (OUT)

Output which will contain the ceiling value of the input number.

Comments

Computes the ceiling value of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberCmp()

Purpose

Compares two Oracle numbers.

Syntax

```
sword OCINumberCmp ( OCIErrror          *err,
                     CONST OCINumber     *number1,
                     CONST OCINumber     *number2,
                     sword                *result );
```

Parameters

- err (IN/OUT)**
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrrorGet()*.
- number1, number2 (IN)**
Numbers to compare.
- result (OUT)**
Comparison result:

Comparison result	Output in <i>result</i> parameter
number1 < number2	negative
number1 = number2	0
number1 > number2	positive

Comments

Compares two Oracle numbers.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrrorGet()

OCINumberCos()

Purpose

Takes the cosine of an Oracle number.

Syntax

```

sword OCINumberCos ( OCIError          *err,
                    CONST OCINumber    *number,
                    OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Argument of the cosine in radians.

result (OUT)

Result of the cosine.

Comments

Takes the cosine in radians of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberDiv()

Purpose

Divides two Oracle numbers.

Syntax

```
sword OCINumberDiv ( OCIError          *err,
                    CONST OCINumber    *number1,
                    CONST OCINumber    *number2,
                    OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number1 (IN)

Pointer to the numerator.

number2 (IN)

Pointer to the denominator.

result (OUT)

Division result.

Comments

Divides *number1* by *number2* and returns result in *result*.

This function returns an error if:

- any of the number arguments is NULL
- there is an underflow error
- there is a divide-by-zero error

Related Functions

OCLErrorGet()

OCINumberExp()

Purpose

Raises *e* to the specified Oracle number power.

Syntax

```
sword OCINumberExp ( OCIError          *err,
                    CONST OCINumber    *number,
                    OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

This function raises *e* to this Oracle number power.

result (OUT)

Output of exponentiation.

Comments

Raises *e* to a given power, specified by an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberFloor()

Purpose

Computes the floor value of an Oracle number.

Syntax

```
sword OCINumberFloor ( OCIError          *err,
                      CONST OCINumber    *number,
                      OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Input number.

result (OUT)

The floor value of the input number.

Comments

Computes the floor value of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberFromInt()

Purpose

Converts integer to Oracle number.

Syntax

```

sword OCINumberFromInt ( OCIError          *err,
                        CONST dvoid        *inum,
                        uword              inum_length,
                        uword              inum_s_flag,
                        OCINumber          *number );
  
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

inum (IN)

Pointer to the integer to convert.

inum_length (IN)

Size of the integer.

inum_s_flag (IN)

Flag that designates the sign of the integer, as follows:

Predefined Constant	Use
OCI_NUMBER_UNSIGNED	Unsigned values
OCI_NUMBER_SIGNED	Signed values

number (OUT)

Given integer converted to Oracle number.

Comments

This is a native type conversion function. It converts any Oracle standard machine-native integer type (e.g. **ub4**, **sb2**) to an Oracle number.

This function returns an error if the number is too big to fit into an Oracle number, if *number* or *inum* is NULL, or if an invalid sign flag value is passed in *inum_s_flag*.

Related Functions

OCIErrorGet()

OCINumberFromReal()

Purpose

Converts a real (floating-point) type to an Oracle number.

Syntax

```

sword OCINumberFromReal ( OCIError          *err,
                          CONST dvoid       *rnum,
                          uword             rnum_length,
                          OCINumber        *number );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

rnum (IN)

Pointer to the floating point number to convert.

rnum_length (IN)

The size of the desired result. Will be equal to *sizeof*{float | double | long double}.

number (OUT)

Given float converted to Oracle number.

Comments

This is a native type conversion function. It converts a machine-native floating point type to an Oracle number.

This function returns an error if *number* or *rnum* is NULL, or if *rnum_length* equals zero.

Related Functions

OCIErrorGet()

OCINumberFromText()

Purpose

Converts character string to Oracle number.

Syntax

```

sword OCINumberFromText ( OCLError          *err,
                          CONST text         *str,
                          ub4                str_length,
                          CONST text         *fmt,
                          ub4                fmt_length,
                          CONST text         *nls_params,
                          ub4                nls_p_length,
                          OCINumber         *number );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

str (IN)

Input string to convert to Oracle number.

str_length (IN)

Size of the input string.

fmt (IN)

Conversion format.

fmt_length (IN)

Length of the *fmt* parameter.

nls_params (IN)

NLS format specification, if NULL string ("") then the default parameters for the session is used.

nls_p_length (IN)

Length of the *nls_params* parameter.

number (OUT)

Given string converted to number.

Comments

Converts the given string to a number according to the specified format. Refer to the TO_NUMBER conversion function described in the *Oracle8 SQL Reference* for a description of format and NLS parameters.

This function returns an error if there is an invalid format, an invalid NLS format, or an invalid input string, if *number* or *str* is NULL, or if *str_length* is zero.

Related Functions

OCIErrorGet()

OCINumberHypCos()

Purpose

Takes the hyperbolic cosine of an Oracle number.

Syntax

```

sword OCINumberHypCos ( OCIError          *err,
                        CONST OCINumber    *number,
                        OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Argument of the cosine hyperbolic.

result (OUT)

Result of the cosine hyperbolic.

Comments

Takes the hyperbolic cosine of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Warning: An Oracle number overflow causes an unpredictable result value.

Related Functions

OCIErrorGet()

OCINumberHypSin()

Purpose

Takes the hyperbolic sine of an Oracle number.

Syntax

```

sword OCINumberHypSin ( OCIError          *err,
                        CONST OCINumber    *number,
                        OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Argument of the sine hyperbolic.

result (OUT)

Result of the sine hyperbolic.

Comments

Takes the hyperbolic sine of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Warning: An Oracle number overflow causes an unpredictable result value.

Related Functions

OCIErrorGet(), *OCINumberHypCos()*, *OCINumberHypTan()*

OCINumberHypTan()

Purpose

Takes the hyperbolic tangent of an Oracle number.

Syntax

```

sword OCINumberHypTan ( OCIError          *err,
                        CONST OCINumber    *number,
                        OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Argument of the tangent hyperbolic.

result (OUT)

Result of the tangent hyperbolic.

Comments

Takes the hyperbolic tangent of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Warning: An Oracle number overflow causes an unpredictable result value.

Related Functions

OCIErrorGet(), *OCINumberHypCos()*, *OCINumberHypSin()*

OCINumberIntPower()

Purpose

Raises a given base to a given integer power.

Syntax

```
sword OCINumberIntPower ( OCLError          *err,  
                          CONST OCNumber     *base,  
                          CONST sword        exp,  
                          OCNumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

base (IN)

Base of the exponentiation.

exp (IN)

Exponent to which the base is raised.

result (OUT)

Output of exponentiation.

Comments

Raises an arbitrary base to an arbitrary integer power.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCLErrorGet()

OCINumberIsZero()

Purpose

Tests if the given number is equal to zero.

Syntax

```
sword OCINumberIsZero ( OCIError          *err,
                        CONST OCINumber    *number,
                        boolean             *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Number to compare.

result (OUT)

Set to TRUE if equal to zero; otherwise, set to FALSE.

Comments

Tests if the given number is equal to zero.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberLn()

Purpose

Takes the natural logarithm (base *e*) of an Oracle number.

Syntax

```

sword OCINumberLn ( OCIError          *err,
                   CONST OCINumber    *number,
                   OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Logarithm of this number is computed.

result (OUT)

Logarithm result.

Comments

Takes the logarithm (base *e*) of the given Oracle number.

This function returns an error if any of the number arguments is NULL, or if *number1* is less than or equal to zero.

Related Functions

OCIErrorGet()

OCINumberLog()

Purpose

Takes the logarithm, to any base, of an Oracle number.

Syntax

```

sword OCINumberLog ( OCIError          *err,
                    CONST OCINumber    *base,
                    CONST OCINumber    *number,
                    OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

base (IN)

Base of the logarithm.

number (IN)

Operand.

result (OUT)

Logarithm result.

Comments

Takes the logarithm with the specified base of an Oracle number.

This function returns an error if:

- any of the number arguments is NULL.
- *number* <= 0
- *base* <= 0

Related Functions

OCLErrorGet()

OCINumberMod()

Purpose

Gets the modulus (remainder) of the division of two Oracle numbers.

Syntax

```

sword OCINumberMod ( OCIError          *err,
                     CONST OCINumber    *number1,
                     CONST OCINumber    *number2,
                     OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number1 (IN)

Pointer to the numerator.

number2 (IN)

Pointer to the denominator.

result (OUT)

Remainder of the result.

Comments

Finds the remainder of the division of two Oracle numbers.

This function returns an error if *number1* or *number2* is NULL, or if there is a divide-by-zero error.

Related Functions

OCLErrorGet()

OCINumberMul()

Purpose

Multiplies two Oracle numbers

Syntax

```
sword OCINumberMul ( OCIError          *err,
                    CONST OCINumber    *number1,
                    CONST OCINumber    *number2,
                    OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number1 (IN)

Number to multiply.

number2 (IN)

Number to multiply.

result (OUT)

Multiplication result.

Comments

Multiplies *number1* with *number2* and returns result in *result*.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCLErrorGet()

OCINumberNeg()

Purpose

Negates an Oracle number.

Syntax

```
sword OCINumberNeg ( OCIError          *err,  
                     CONST OCINumber    *number,  
                     OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number (IN)

Number to negate.

result (OUT)

Contains negated value of *number*.

Comments

Negates an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCLErrorGet()

OCINumberPower()

Purpose

Raises a given base to a given exponent.

Syntax

```
sword OCINumberPower ( OCIError          *err,
                      CONST OCINumber    *base,
                      CONST OCINumber    *number,
                      OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

base (IN)

Base of the exponentiation.

number (IN)

Exponent to which the base is to be raised.

result (OUT)

Output of exponentiation.

Comments

Raises an arbitrary base to an arbitrary power.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberRound()

Purpose

Rounds an Oracle number to a specified decimal place.

Syntax

```

sword OCINumberRound ( OCIError          *err,
                        CONST OCINumber    *number,
                        sword               decplace,
                        OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Number to round.

decplace (IN)

Number of decimal digits to the right of the decimal point to round to. Negative values are allowed.

result (OUT)

Output of rounding.

Comments

Rounds an Oracle number to a specified decimal place.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberSetZero()

Purpose

Initializes an Oracle number to zero

Syntax

```
void OCINumberSetZero ( OCLError      *err
                        OCINumber      *num );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

num (IN/OUT)

Number to initialize to zero value.

Comments

Initializes the given number to value 0.

Related Functions

OCLErrorGet()

OCINumberSign()

Purpose

Gets sign of an Oracle number.

Syntax

```

sword OCINumberSign ( OCLError          *err,
                      CONST OCINumber    *number,
                      sword               *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number (IN)

Number whose sign is returned.

result (OUT)

Possible values:

Value of <i>number</i>	Output in <i>result</i> parameter
<i>number</i> < 0	-1
<i>number</i> == 0	0
<i>number</i> > 0	1

Comments

Obtains the sign of an Oracle number.

This function returns an error if *number* or *result* is NULL.

Related Functions

OCLErrorGet()

OCINumberSin()

Purpose

Takes the sine of an Oracle number.

Syntax

```

sword OCINumberSin ( OCIError          *err,
                    CONST OCINumber    *number,
                    OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Argument of the sine in radians.

result (OUT)

Result of the sine.

Comments

Takes the sine in radians of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCINumberSqrt()

Purpose

Computes the square root of an Oracle number.

Syntax

```
sword OCINumberSqrt ( OCIError          *err,
                     CONST OCINumber    *number,
                     OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Input number.

result (OUT)

Output which will contain the square root of the input number.

Comments

Computes the square root of an Oracle number.

This function returns an error if *number* is NULL or *number* is negative.

Related Functions

OCIErrorGet()

OCINumberSub()

Purpose

Subtract two Oracle numbers.

Syntax

```
sword OCINumberSub ( OCIError          *err,
                    CONST OCINumber    *number1,
                    CONST OCINumber    *number2,
                    OCINumber          *result );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number1, number2 (IN)

This function subtracts *number2* from *number1*.

result (OUT)

Subtraction result.

Comments

Subtracts *number2* from *number1* and returns result in *result*.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCLErrorGet()

OCINumberTan()

Purpose

Takes the tangent of an Oracle number.

Syntax

```

sword OCINumberTan ( OCIError          *err,
                    CONST OCINumber    *number,
                    OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number (IN)

Argument of the tangent in radians.

result (OUT)

Result of the tangent.

Comments

Takes the tangent in radians of an Oracle number.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCLErrorGet()

OCINumberToInt()

Purpose

Converts an Oracle number type to integer.

Syntax

```
sword OCINumberToInt ( OCLError          *err,
                      CONST OCINumber    *number,
                      uword               rsl_length,
                      uword               rsl_flag,
                      dvoid               *rsl );
```

Parameters

err (IN/OUT)
The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number (IN)
Number to convert.

rsl_length (IN)
Size of the desired result.

rsl_flag (IN)
Flag denoting the desired sign of the output; set as follows:

Predefined Constant	Use
OCI_NUMBER_UNSIGNED	Unsigned values
OCI_NUMBER_SIGNED	Signed values

rsl (OUT)
Pointer to space for the result.

Comments

This is a native type conversion function. It converts the given Oracle number into an integer of the form **xbn** (e.g. **ub2**, **ub4**, **sb2**, etc.)

This function returns an error if *number* or *rsl* is NULL, if *number* is too big (overflow) or too small (underflow), or if an invalid sign flag value is passed in *rsl_flag*.

Related Functions

OCIErrorGet()

OCINumberToReal()

Purpose

Converts an Oracle number type to Real.

Syntax

```
sword OCINumberToReal ( OCLError          *err,
                        CONST OCINumber    *number,
                        uword               rsl_length,
                        dvoid               *rsl );
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number (IN)

Number to convert.

rsl_length (IN)

The size of the desired result. This will be equal to *sizeof*{ float | double | long double }.

rsl (OUT)

Pointer to space for storing the result.

Comments

This is a native type conversion function. It converts an Oracle number into a machine-native real type. This function only converts numbers up to LDBL_DIG, DBL_DIG, or FLT_DIG digits of precision and removes trailing zeroes. The above constants are defined in *float.h*.

This function returns an error if *number* or *rsl* is NULL, or if *rsl_length* = 0.

Related Functions

OCLErrorGet(), *OCINumberFromReal()*

OCINumberToText()

Purpose

Converts an Oracle number to a character string.

Syntax

```

sword OCINumberToText ( OCLError          *err,
                        CONST OCINumber    *number,
                        CONST text         *fmt,
                        ub4                 fmt_length,
                        CONST text         *nls_params,
                        ub4                 nls_p_length,
                        ub4                 *buf_size,
                        text                *buf );
  
```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCLErrorGet()*.

number (IN)

Oracle number to convert.

fmt (IN)

Conversion format.

fmt_length (IN)

Length of the *fmt* parameter.

nls_params (IN)

NLS format specification. If NULL string (i.e., (text *)0), then the default parameters for the session is used.

nls_p_length (IN)

Length of the *nls_params* parameter.

buf_size (IN)

Size of the buffer.

buf (OUT)

Buffer into which the converted string is placed.

Comments

Converts a given number to a character string according to a specified format. Refer to the TO_NUMBER conversion function described in the *Oracle8 SQL Reference* for a description of format and NLS parameters.

The converted number string is stored in *buf*, up to a maximum of *buf_size* bytes. This function returns an error if:

- *number* or *buf* is NULL
- buffer is too small
- invalid format or invalid NLS format is passed
- number to text translation for given format causes an overflow

Related Functions

OCIErrorGet()

OCINumberTrunc()

Purpose

Truncates an Oracle number at a specified decimal place.

Syntax

```

sword OCINumberTrunc ( OCIError          *err,
                      CONST OCINumber    *number,
                      sword               decplace,
                      OCINumber          *result );

```

Parameters

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

number (IN)

Input number.

decplace (IN)

Number of decimal digits to the right of the decimal point at which to truncate. Negative values are allowed.

result (OUT)

Output of truncation.

Comments

Truncates an Oracle number at a specified decimal place.

This function returns an error if any of the number arguments is NULL.

Related Functions

OCIErrorGet()

OCIRawAllocSize()

Purpose

Gets allocated size of raw memory in bytes.

Syntax

```
sword OCIRawAllocSize ( OCIEnv          *env,  
                        OCIError        *err,  
                        CONST OCIRaw    *raw,  
                        ub4              *allocsize );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

raw (IN)

Raw data whose allocated size in bytes is returned. This must be a non-NULL pointer.

allocsize (OUT)

The allocated size of raw memory in bytes is returned.

Comments

Retrieves the allocated size of the raw memory in bytes. The allocated size is greater than or equal to the actual raw size.

Related Functions

OCIErrorGet()

OCIRawAssignBytes()

Purpose

Assigns raw bytes of type **ub1*** to Oracle **OCIRaw*** datatype.

Syntax

```
sword OCIRawAssignBytes ( OCIEnv          *env,
                          OCIError        *err,
                          CONST ub1       *rhs,
                          ub4             rhs_len,
                          OCIRaw          **lhs );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

rhs (IN)

Right-hand side (source) of the assignment, of datatype **ub1**.

rhs_len (IN)

Length of the *rhs* raw bytes.

lhs (IN/OUT)

Left-hand side (target) of the assignment **OCIRaw** data.

Comments

Assigns *rhs* raw bytes to *lhs* raw datatype. The *lhs* raw may be resized depending upon the size of the *rhs*. The raw bytes assigned are of type **ub1**.

Related Functions

OCIErrorGet()

OCIRawAssignRaw()

Purpose

Assign one Oracle raw datatype to another Oracle raw datatype.

Syntax

```
sword OCIRawAssignRaw ( OCIEEnv          *env,
                        OCIError          *err,
                        CONST OCIRaw      *rhs,
                        OCIRaw           **lhs );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

rhs (IN)

Right-hand side (source) of the assignment; **OCIRaw** data.

lhs (IN/OUT)

Left-hand side (target) of the assignment; **OCIRaw** data.

Comments

Assigns *rhs* raw to *lhs* raw. The *lhs* raw may be resized depending upon the size of the *rhs*.

Related Functions

OCIErrorGet()

OCIRawPtr()

Purpose

Gets pointer to raw data.

Syntax

```
ub1 *OCIRawPtr ( OCIEnv          *env,  
                  CONST OCIRaw    *raw );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

raw (IN)

Pointer to the data of a given row is returned.

Returns

pointer to the data of a given row.

Comments

Returns a pointer to the data of a given row.

Related Functions

OCIErrorGet()

OCIRawResize()

Purpose

Resizes the memory of a given variable-length raw.

Syntax

```
sword OCIRawResize ( OCIEnv      *env,  
                     OCIError    *err,  
                     ub2         new_size,  
                     OCIRaw      **raw );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

new_size (IN)

New size of the raw data in bytes.

raw (IN)

Variable-length raw pointer; the raw is resized to *new_size*.

Comments

This function resizes the memory of the given variable-length raw in the object cache. The previous contents of the raw are *not* preserved. This function may allocate the raw in a new memory region in which case the original memory occupied by the given raw will be freed. If the input raw is NULL (*raw* == NULL), then this function will allocate memory for the raw data.

If the *new_size* is 0, then this function frees the memory occupied by *raw* and a NULL pointer value is returned.

Related Functions

OCIErrorGet()

OCIRawSize()

Purpose

Gets the size of a given raw.

Syntax

```
ub4 OCIRawSize ( OCIEnv          *env,  
                  CONST OCIRaw    *raw );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

raw (IN/OUT)

Raw whose size is returned.

Returns

size of the raw in bytes.

Comments

Returns the size of the given raw in bytes.

Related Functions

OCIErrorGet()

OCIRefAssign()

Purpose

Assigns one REF to another, such that both reference the same object.

Syntax

```
sword OCIRefAssign ( OCIEnv          *env,
                    OCIError        *err,
                    CONST OCIRef    *source,
                    OCIRef          **target );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

source (IN)

REF to copy from.

target (IN/OUT)

REF to copy to.

Comments

Copies *source* REF to *target* REF; both then reference the same object. If the *target* REF pointer is NULL (i.e. **target* == NULL), then *OCIRefAssign()* will allocate memory for the *target* REF in the OCI object cache prior to the copy.

Related Functions

OCIErrorGet()

OCIRefClear()

Purpose

Clears or nullifies a REF

Syntax

```
void OCIRefClear ( OCIEnv      *env,  
                  OCIRef      *ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

ref (IN/OUT)

REF to clear.

Comments

Clears or nullifies the given REF. A REF is considered to be a NULL REF if it no longer points to an object. Logically, a NULL REF is a dangling REF.

Note that a null ref is still a valid SQL value and is not SQL-ly null. It can be used as a valid non-null constant ref value for NOT NULL column or attribute of a row in a table.

If a NULL pointer value is passed as a REF, then this function is a no-op.

Related Functions

OCIErrorGet()

OCIRefFromHex()

Purpose

Converts the given hexadecimal string into a REF.

Syntax

```
sword OCIRefFromHex ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCISvcCtx *svc,
                      CONST text      *hex,
                      ub4              length,
                      OCIRef          **ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

svc (IN)

OCI service context handle; if the resulting ref is initialized with this service context.

hex (IN)

Hexadecimal text string (previously output by *OCIRefToHex()*) to convert into a REF.

length (IN)

Length of the hexadecimal text string.

ref (IN/OUT)

The REF into which the hexadecimal string is converted. If **ref* is NULL on input, then space for the REF is allocated in the object cache, otherwise the memory occupied by the given REF is re-used.

Comments

Converts the given hexadecimal text string into a REF. This function ensures that the resulting REF is well formed. It does *not* ensure that the object pointed to by the resulting REF exists or not.

Related Functions

OCLErrorGet()

OCIRefHexSize()

Purpose

Returns the size of the hex representation of a REF.

Syntax

```
ub4 OCIRefHexSize ( OCIEnv          *env,  
                    CONST OCIRef     *ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

ref (IN)

REF whose size in hexadecimal representation in bytes is returned.

Returns

The size of the hexadecimal representation of the REF.

Comments

Returns the size of the buffer in bytes required for the hexadecimal representation of the ref. A buffer of at least this size must be passed to the ref-to-hex (*OCIRefToHex()*) conversion function.

Related Functions

OCIRefAssign()

OCIRefIsEqual()

Purpose

Compares two REFs to determine if they are equal.

Syntax

```
boolean OCIRefIsEqual ( OCIEnv          *env,
                        CONST OCIRef     *x,
                        CONST OCIRef     *y );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

x (IN)

REF to compare.

y (IN)

REF to compare.

Returns

TRUE if the two REFs are equal

FALSE if the two REFs are not equal, or *x* is NULL, or *y* is NULL

Comments

Compares the given REFs for equality.

Two REFs are equal if and only if they are both referencing the same object, whether persistent or transient.

Note: Two NULL REFs are considered *not* equal by this function.

Related Functions

OCIErrorGet()

OCIRefIsNull()

Purpose

Tests if a REF is NULL

Syntax

```
boolean OCIRefIsNull ( OCIEnv          *env,  
                      CONST OCIRef     *ref );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

ref (IN)

REF to test for NULL.

Returns

TRUE if the given REF is NULL.

FALSE if the given REF is not NULL.

Comments

Returns TRUE if the given REF is NULL; otherwise, returns FALSE.

A REF is NULL if and only if:

- it is supposed to be referencing a persistent object, but the object's identifier is NULL, or
- it is supposed to be referencing a transient object, but it is currently not pointing to an object.

Note: A REF is a *dangling REF* if the object that it points to does not exist.

Related Functions

OCIErrorGet()

OCIRefToHex()

Purpose

Converts a REF to a hexadecimal string

Syntax

```
sword OCIRefToHex ( OCIEnv          *env,  
                   OCIError        *err,  
                   CONST OCIRef     *ref,  
                   text             *hex,  
                   ub4              *hex_length );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

ref (IN)

REF to be converted into a hexadecimal string; if *ref* is a NULL REF (i.e. *OCIRefsNull(ref) == TRUE*) then zero *hex_length* value is returned.

hex (OUT)

Buffer that is large enough to contain the resulting hexadecimal string; the contents of the string is opaque to the caller.

hex_length (IN/OUT)

On input specifies the size of the *hex* buffer on output specifies the actual size of the hexadecimal string being returned in *hex*.

Comments

Converts the given REF into a hexadecimal string, and returns the length of the string. The resulting string is opaque to the caller.

This function returns an error if the given buffer is not big enough to hold the resulting string.

Related Functions

OCIErrorGet(), OCIRefHexSize(), OCIRefIsNull()

OCIStringAllocSize()

Purpose

Gets allocated size of string memory in bytes.

Syntax

```
sword OCIStringAllocSize ( OCIEnv          *env,  
                           CONST OCIString *vs,  
                           ub4             *allocsize );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

vs (IN)

String whose allocated size in bytes is returned. *vs* must be a non-NULL pointer.

allocsize (OUT)

The allocated size of string memory in bytes is returned.

Comments

Returns the allocated size of the string memory in bytes. The allocated size is greater than or equal to the actual string size.

Related Functions

OCIErrorGet()

OCIStringAssign()

Purpose

Assigns one string to another string.

Syntax

```
sword OCIStringAssign ( OCIEnv          *env,  
                        OCIError        *err,  
                        CONST OCIString *rhs,  
                        OCIString       **lhs );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

rhs (IN)

Right-hand side (source) of the assignment.

lhs (IN/OUT)

Left-hand side (target) of the assignment.

Comments

Assigns *rhs* string to *lhs* string. The *lhs* string may be resized depending upon the size of the *rhs*. The assigned string is NULL-terminated.

The length field will not include the extra byte needed for null termination.

This function returns an error if the assignment operation runs out of space.

Related Functions

OCIErrorGet()

OCIStringAssignText()

Purpose

Assigns the source text string to the target string.

Syntax

```
sword OCIStringAssignText ( OCIEnv          *env,  
                           OCIError        *err,  
                           CONST text      *rhs,  
                           ub2             rhs_len,  
                           OCIString       **lhs );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

rhs (IN)

Right-hand side (source) of the assignment, a text string.

rhs_len (IN)

Length of the *rhs* string.

lhs (IN/OUT)

Left-hand side (target) of the assignment.

Comments

Assigns *rhs* string to *lhs* string. The *lhs* string may be resized depending upon the size of the *rhs*. The assigned string is NULL-terminated.

The length field will not include the extra byte needed for null termination.

Related Functions

OCIErrorGet()

OCIStringPtr()

Purpose

Gets a pointer to a given string.

Syntax

```
text *OCIStringPtr ( OCIEnv          *env,  
                    CONST OCIString  *vs );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

vs (IN)

Pointer to the text of this string is returned.

Returns

pointer to the text of the string.

Comments

Returns the pointer to the text of the given string.

Related Functions

OCIErrorGet()

OCIStringResize()

Purpose

Resizes the memory of a given string.

Syntax

```
sword OCIStringResize ( OCIEnv      *env,  
                        OCIError    *err,  
                        ub4          new_size,  
                        OCIString    **str );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

new_size (IN)

New memory size of the string in bytes. *new_size* must include space for the NULL character ('\0') as string terminator.

str (IN/OUT)

Allocated memory for the string which is freed from the OCI object cache.

Comments

This function resizes the memory of the given variable-length string in the object cache. Contents of the string are *not* preserved. This function may allocate the string in a new memory region, in which case the original memory occupied by the given string is freed. If *str* is NULL, this function allocates memory for the string. If *new_size* is 0, this function frees the memory occupied by *str* and a NULL pointer value is returned.

Related Functions

OCIErrorGet()

OCIStringSize()

Purpose

Gets string size.

Syntax

```
ub4 OCIStringSize ( OCIEnv          *env,  
                   CONST OCIString  *vs );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

vs (IN)

String whose size is returned.

Returns

size of the string in bytes.

Comments

Returns the size of the given string in bytes. The returned size does not include an extra byte for NULL termination.

Related Functions

OCIErrorGet()

OCITableDelete()

Purpose

Deletes the element at the specified index.

Syntax

```
sword OCITableDelete ( OCIEnv          *env,  
                      OCIError        *err,  
                      sb4             index,  
                      OCITable        *tbl );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

index (IN)

Index of the element which must be deleted.

tbl (IN)

Table whose element is deleted.

Comments

Deletes the element at the given *index*.

Note: The position ordinals of the remaining elements of the table are not changed by *OCITableDelete()*. The delete operation creates “holes” in the table.

This function returns an error if the element at the given index has already been deleted or if the given index is not valid for the given table. It is also an error if any input parameter is NULL.

Related Functions

OCIErrorGet()

OCITableExists()

Purpose

Tests whether an element exists at the given index.

Syntax

```
sword OCITableExists ( OCIEnv          *env,  
                       OCIError        *err,  
                       CONST OCITable   *tbl,  
                       sb4              index,  
                       boolean          *exists );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

tbl (IN)

Table in which the given index is checked.

index (IN)

Index of the element which is checked for existence.

exists (OUT)

Set to TRUE if element at given *index* exists; otherwise, it is set to FALSE.

Comments

Tests whether an element exists at the given index, *index*.

This function returns an error if any input parameter is NULL.

Related Functions

OCIErrorGet()

OCITableFirst()

Purpose

Returns the first index of an existing element in a given table.

Syntax

```
sword OCITableFirst ( OCIEnv          *env,
                     OCIError        *err,
                     CONST OCITable  *tbl,
                     sb4              *index );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

tbl (IN)

Table to scan.

index (OUT)

First index of the element which exists in the given table is returned.

Comments

Returns the index of the first element which exists in a given table.

For example, if *OCITableDelete()* deleted the first 5 elements of a table, *OCITableFirst()* returns 6.

See *OCITableDelete()* for information regarding non-data “holes” in tables.

This function returns an error if the table is empty.

Related Functions

OCIErrorGet(), *OCITableDelete()*

OCI TableLast()

Purpose

Returns the index of the last existing element of a table.

Syntax

```
sword OCI TableLast ( OCIEnv          *env,
                     OCIError        *err,
                     CONST OCITable  *tbl,
                     sb4             *index );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

tbl (IN)

Table to scan.

index (OUT)

Index of the last existing element in the table.

Comments

Returns the index of the last existing element in the given table.

This function returns an error if the table is empty.

Related Functions

OCIErrorGet()

OCITableNext()

Purpose

Returns the index of the next existing element of a table.

Syntax

```
sword OCITableNext ( OCIEnv           *env,  
                    OCIError        *err,  
                    sb4             index,  
                    CONST OCITable  *tbl,  
                    sb4             *next_index  
                    boolean          *exists );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

index (IN)

Index for starting point of scan.

tbl (IN)

Table to scan.

next_index (OUT)

Index of the next existing element after *tbl(index)*.

exists (OUT)

FALSE if no next index is available, else TRUE.

Comments

Returns the smallest position *j*, greater than *index*, such that exists(*j*) is TRUE

See Also: Refer to the description of *OCIStringAllocSize()* on page 15-109, regarding the existence of non-data “holes” in tables.

Related Functions

OCITablePrev()

OCITablePrev()

Purpose

Returns the index of the previous existing element of a table.

Syntax

```
sword OCITablePrev ( OCIEnv           *env,  
                    OCIError        *err,  
                    sb4             index,  
                    CONST OCITable  *tbl,  
                    sb4             *prev_index  
                    boolean         *exists );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns *OCI_ERROR*. Obtain diagnostic information by calling *OCIErrorGet()*.

index (IN)

Index for starting point of scan.

tbl (IN)

Table to scan.

prev_index (OUT)

Index of the previous existing element before *tbl(index)*.

exists (OUT)

FALSE if no previous index is available, else TRUE.

Comments

Return the largest position *j*, less than *index*, such that exists(*j*) is TRUE

See Also: Refer to the description of *OCIStringAllocSize()* on page 15-109, regarding the existence of non-data “holes” in tables.

Related Functions

OCITableNext()

OCITableSize()

Purpose

Return size of the given table (not including deleted elements).

Syntax

```
sword OCITableSize ( OCIEnv           *env,
                    OCIError        *err,
                    CONST OCITable  *tbl
                    sb4             *size );
```

Parameters

env (IN/OUT)

The OCI environment handle initialized in object mode. See the description of *OCIInitialize()* in Chapter 13 for more information.

err (IN/OUT)

The OCI error handle. If there is an error, it is recorded in *err* and this function returns OCI_ERROR. Obtain diagnostic information by calling *OCIErrorGet()*.

tbl (IN)

Nested table whose number of elements is returned.

size (OUT)

Current number of elements in the nested table. The count does not include deleted elements.

Comments

Returns the count of elements in the given table.

This count will be decremented upon deleting elements from the nested table. So this count does not include any “holes” created by deleting elements. To get the count not including the deleted elements, use *OCICollSize()*.

For example:

```
OCITableSize(...);
// assume 'size' returned is equal to 5
OCITableDelete(...); // delete one element
```

```
OCITableSize(...);  
// 'size' returned is equal to 4
```

To get the count plus the count of deleted elements use *OCICollSize()*. Continuing the above example:

```
OCICollSize(...)  
// 'size' returned is still equal to 5
```

This function returns an error if an error occurs during the loading of the nested table into the object cache, or if any of the input parameters is NULL.

Related Functions

OCICollSize()

OCI External Procedure Functions

The chapter contains the following sections:

- Introduction
- The OCI External Procedure Functions

Introduction

This chapter describes the OCI External Procedure Functions. These functions enable users of external procedures to raise errors, allocate some memory, and get OCI context information. For more information about using these functions, refer to the *PL/SQL User's Guide and Reference*.

Return Codes

Success and error return codes are defined for certain external procedure interface functions. If a particular interface function returns `OCIEXTPROC_SUCCESS` or `OCIEXTPROC_ERROR`, then applications must use these macros to check for return values.

`OCIEXTPROC_SUCCESS` - External Procedure Success Return Code

`OCIEXTPROC_ERROR` - External Procedure Failure Return Code

With_Context Type

The C callable interface to PL/SQL external procedures requires the *with_context* parameter to be passed. The type of this structure is **OCIExtProcContext**, which is opaque to the user.

The user can declare the *with_context* parameter in the application as

```
OCIExtProcContext *with_context;
```

The OCI External Procedure Functions

The remainder of this chapter specifies the release 8.0 OCI external procedure functions for C. For each function, the following information is listed:

Purpose

A brief description of the action performed by the function.

Syntax

A code snippet showing the syntax for calling the function, including the ordering and types of the parameters.

Parameters

A description of each of the function's parameters. This includes the parameter's mode. The mode of a parameter has three possible values, as described below:

Mode	Description
IN	A parameter that passes data to Oracle
OUT	A parameter that receives data from Oracle on this or a subsequent call
IN/OUT	A parameter that passes data on the call and receives data on the return from this or a subsequent call.

Comments

More detailed information about the function (if available). This may include restrictions on the use of the function, or other information that might be useful when using the function in an application.

Returns

A list of possible return values for the function.

Example

A complete or partial code example demonstrating the use of the function call being described. Not all function descriptions include an example.

Related Functions

A list of related function calls.

OCIExtProcAllocCallMemory()

Purpose

Allocate N bytes of memory for the duration of the External Procedure.

Syntax

```
dvoid * OCIExtProcAllocCallMemory ( OCIExtProcContext    *with_context,
                                     size_t                amount )
```

Parameters

with_context (IN)

The **with_context** pointer that is passed to the C External Procedure. See “With_Context Type” on page 16-2.

amount (IN)

The number of bytes to allocate.

Comments

This call allocates *amount* bytes of memory for the duration of the call of the external procedure.

Any memory allocated by this call is freed by PL/SQL upon return from the external procedure. The application must not use any kind of 'free' function on memory allocated by *OCIExtProcAllocCallMemory()*. Use this function to allocate memory for function returns.

A zero return value should be treated as an error

Returns

An untyped (opaque) Pointer to the allocated memory.

Example

```
text *ptr = (text *)OCIExtProcAllocCallMemory(wctx, 1024)
```

Related Functions

OCIExtProcRaiseExcp()

Purpose

Raise an Exception to PL/SQL.

Syntax

```
size_t OCIExtProcRaiseExcp ( OCIExtProcContext    *with_context,  
                             int                  errnum )
```

Parameters

with_context (IN)

The **with_context** pointer that is passed to the C External Procedure. See “With_Context Type” on page 16-2.

errnum (IN)

Oracle Error number to signal to PL/SQL. *errnum* must be a positive number and in the range 1 to 32767.

Comments

Calling this function signals an exception back to PL/SQL. After a successful return from this function, the external procedure must start its exit handling and return back to PL/SQL. Once an exception is signalled to PL/SQL, IN/OUT and OUT arguments, if any, are not processed at all.

Returns

OCIEXTPROC_SUCCESS - If the call was successful.

OCIEXTPROC_ERROR - If the call failed.

Related Functions

OCIExtProcRaiseExcpWithMsg()

OCIExtProcRaiseExcpWithMsg()

Purpose

Raise an exception with a message.

Syntax

```
size_t OCIExtProcRaiseExcpWithMsg ( OCIExtProcContext  *with_context,
                                     int                errnum,
                                     char                *errmsg,
                                     size_t              msglen )
```

Parameters

with_context (IN)

The **with_context** pointer that is passed to the C External Procedure. See “With_Context Type” on page 16-2.

errnum (IN)

Oracle Error number to signal to PL/SQL. The value of *errnum* must be a positive number and in the range 1 to 32767

errmsg (IN)

The error message associated with the *errnum*.

len (IN)

The length of the error message. Pass zero if *errmsg* is a null terminated string.

Comments

Raise an exception to PL/SQL. In addition, substitute the following error message string within the standard Oracle error message string. See the description of *OCIExtProcRaiseExcp()* for more information.

Returns

OCIEXTPROC_SUCCESS - If the call was successful.

OCIEXTPROC_ERROR - If the call failed.

Related Functions

OCIExtProcRaiseExcp()

OCIExtProcGetEnv()

Purpose

Get OCI Environment

Syntax

```
sword OCIExtProcGetEnv ( OCIExtProcContext    *with_context,
                        OCIEnv                  envh,
                        OCISvcCtx              svch,
                        OCIError               errh )
```

Parameters

with_context (IN)

The **with_context** pointer that is passed to the C External Procedure. See “With_Context Type” on page 16-2.

envh (OUT)

The OCI Environment handle.

svch (OUT)

The OCI Service handle.

errh (OUT)

The OCI Error handle.

Comments

Gets the OCI environment, service context, and error handles.

The primary purpose of this function is to allow OCI callbacks to use the database in the same transaction. The OCI handles obtained by this function should be used in OCI callbacks to the database. If these handles are obtained through standard OCI calls, then these handles use a new connection to the database and cannot be used for callbacks in the same transaction. In one external procedure you can use either callbacks or a new connection, but not both.

Returns

OCI_SUCCESS, on successful completion of the function.

OCI_ERROR, on error.

Related Functions

Part IV

Appendices

This part of the book contains the appendices:

- Appendix A, “Upgrading Release 7.x OCI Applications to Release 8.0”, discusses issues involved in upgrading Release 7.x OCI applications to Release 8.0. This includes lists of obsolete and obsolescent OCI calls.
- Appendix B, “Handle and Descriptor Attributes”, lists the attributes of the various OCI handles.
- Appendix C, “Oracle Reserved Words, Keywords and Namespaces”, provides information about reserved words, keywords and reserved namespaces.
- Appendix D, “Code Examples”, includes code examples.
- Appendix E, “OCI Function Server Roundtrips”, provides information about the server roundtrips required by most OCI functions.
- Appendix F, “Oracle8 OCI New Features”, provides detailed information about features and enhancements available in the Oracle8 OCI.

Upgrading Release 7.x OCI Applications to Release 8.0

This appendix covers issues of compatibility between Oracle7 and Oracle8 OCI applications and servers. It also discusses issues involved in upgrading applications from the 7.x OCI to the 8.0 OCI.

The appendix contains the following sections:

- Compatibility and Upgrade Overview
- Obsolescent OCI Routines
- Obsolete OCI Routines
- Compatibility
- Upgrading

Compatibility and Upgrade Overview

Release 8.0 of the Oracle server provides support for applications written with either the 7.x OCI and the 8.0 OCI.

The remaining sections of this chapter discuss changes in the OCI library routines, issues concerning compatibility between different versions of the OCI and server, as well as issues involved in migrating an application from the release 7.x OCI to the release 8.0 OCI.

Obsolescent OCI Routines

Release 8.0 of the Oracle Call Interface contains an entirely new set of functions which were not available in release 7.3. The earlier calls are still available, but Oracle recommends that existing applications start using the new calls to improve performance and provide increased functionality.

Table A-1 lists the 7.x OCI calls with their release 8.0 equivalents. For more information about the 8.0 OCI calls, see the earlier chapters of this volume. For more information about the 7.x calls, see the *Programmer's Guide to the Oracle Call Interface, Release 7.3*. These calls are obsolescent, meaning that Oracle may not support these calls in future versions of the OCI.

Note: In many cases the new OCI routines do not map directly onto the 7.x routines, so it may not be possible to simply replace one function call and parameter list with another. Additional program logic may be required before or after the new call is made. See the remaining chapters of this guide for more information.

Table A-1 Obsolescent OCI Routines

7.x OCI Routine	Equivalent or Similar 8.0 OCI Routine
obindps(), obndra(), obndrn(), obndrv()	OCIBindByName(), OCIBindByPos() (Note: additional bind calls may be necessary for some data types)
obreak()	OCIBreak()
ocan()	none
oclose()	Note: cursors are not used in Release 8.0
ocof(), ocon()	OCIStmtExecute() in OCI_COMMIT_ON_SUCCESS mode
ocom()	OCITransCommit()
odefin(), odefinps()	OCIDefineByPos() (Note: additional define calls may be necessary for some data types)

Table A–1 Obsolescent OCI Routines (Cont.)

7.x OCI Routine	Equivalent or Similar 8.0 OCI Routine
odescr()	Note: schema objects are described with OCIDescribeAny(). A describe, as used in release 7.x , will most often be done by calling OCIAttrGet() on the statement handle after SQL statement execution.
odessp()	OCIDescribeAny()
oerhms()	OCIErrorGet()
oexec(), oexn()	OCISStmtExecute() (or ociflsh())
oexfet()	OCISStmtExecute(), OCISStmtFetch() (Note: result set rows can be implicitly prefetched)
ofen(), ofetch()	OCISStmtFetch()
oflng()	none
ogetpi()	OCISStmtGetPieceInfo()
olog()	OCISvcCtxLogon()
ologof()	OCISvcCtxLogoff()
onbclr(), onbset(), onbtst()	Note: non-blocking mode can be set or checked by calling OCIAttrSet() or OCIAttrGet() on the server context handle or service context handle
oopen()	Note: cursors are not used in Release 8.0
oopt()	none
oparse()	Note: there is no explicit parse step in the 8.0 OCI.
opinit()	OCIInitialize()
orol()	OCITransRollback()
osetpi()	OCISStmtSetPieceInfo()
sqlld2()	none
sqllda()	none
odsc()	Note: see odescr() above
oerrmsg()	OCIErrorGet()
olon()	OCISvcCtxLogon()
orlon()	OCISvcCtxLogon()

Table A-1 Obsolescent OCI Routines (Cont.)

7.x OCI Routine	Equivalent or Similar 8.0 OCI Routine
oname()	Note: see odescr() above
osql3()	Note: see oparse() above

See Also: For information about the additional functionality provided by new release 8.0 functions not listed here, see the remaining chapters of this guide.

Obsolete OCI Routines

Some OCI routines that were available in previous versions of the OCI are now obsolete, meaning that they are not supported for Release 8.0. They are listed in Table A-2:

Table A-2 Obsolete OCI Routines

Obsolete OCI Routine	Equivalent or Similar 8.0 OCI Routine
obind()	OCIBindByName(), OCIBindByPos() (Note: additional bind calls may be necessary for some data types)
obindn()	OCIBindByName(), OCIBindByPos() (Note: additional bind calls may be necessary for some data types)
odfinn()	OCIDefineByPos() (Note: additional define calls may be necessary for some data types)
odsrbn()	Note: see odescr() in Table A-1
ologon()	OCISvcCtxLogon()
osql()	Note: see oparse() Table A-1

Compatibility

This section addresses compatibility between different versions of the OCI and Oracle server.

7.x Applications

Existing 7.x applications with no new release 8.0 OCI calls have two choices:

- do not relink the application
- relink with the new 8.0 OCI library

In either case, the application will work against both Oracle7 and Oracle8 servers. The application will not be able to use Oracle8's object features, and will not get any of the performance or scalability benefits provided by the Oracle8 OCI.

8.0 Applications

New applications written completely in the release 8.0 OCI will work seamlessly against both Oracle7 and Oracle8 servers with the following exceptions:

- Against Oracle7 servers, none of Oracle8's object features are supported, and the following datatypes are not supported:
 - SQLT_NTY - named data type
 - SQLT_REF - reference to named data type in host language representation.
 - SQLT_CLOB - a character LOB data type.
 - SQLT_BLOB - a binary LOB data type.
 - SQLT_NCLOB - a national character set LOB data type.
 - SQLT_NCHAR - fixed or varying national character set datatype.
 - SQLT_BFILE - a binary FILE LOB data type.
 - SQLT_RSET - result set data type.
- Against Oracle7 Servers, the following calls or features are not supported, or are supported with restrictions:

Table A–3 8.0 OCI Restrictions When Running Against Oracle7 Servers

Function	Restrictions
OCIBindObject()	not supported
OCIPasswordChange()	not supported
OCIDefineObject()	not supported
OCIDescribeAny()	only supports description of select lists or stored procedures
OCIErrorGet()	only a subset of Oracle error codes can be returned
OCISmtFetch()	prefetching options not supported
OCILob*() calls	LOB/FILE calls are not supported
OCIAttrSet()	setting NCHAR attributes not supported
OCIAttrGet()	getting NCHAR attributes not supported

Upgrading

Programmers who wish to incorporate new release 8.0 functionality into existing OCI applications have two options:

- Completely rewrite the application to use only new OCI calls
- Incorporate new release 8.0 OCI calls into the application, while still using 7.x calls for some operations.

This manual, along with *Volume I*, should provide the information necessary to rewrite an existing application to use only new OCI calls.

Adding 8.0 Calls to 7.x Applications

The following guidelines apply to programmers who want to incorporate new Oracle8 datatypes and features by using new OCI calls, while keeping 7.x calls for some operations:

- Change the existing logon to use *OCILogon* instead of *olog()* (or other logon call). The service context handle can be used with new OCI calls or can be converted into a **Lda_Def** to be used with 7.x OCI calls.

Note: See the description of *OCIServerAttach()* on page 13-125 and the description of *OCISessionBegin()* on page 13-129 for information about the logon calls necessary for applications which are maintaining multiple sessions.

- Once the server context handle has been initialized, it can be used with Oracle8 OCI calls.
- To use Oracle7 OCI calls, convert the server context handle to an **Lda_Def** using *OCISvcCtxToLda()*, and pass the resulting **Lda_Def** to the 7.x calls.

Note: If there are multiple service contexts which share the same server handle, only one can be in Oracle7 mode at any time.

- To begin using 8.0 calls again, the application must convert the **Lda_Def** back to a server context handle using *OCILdaToSvcCtx()*.
- The application may toggle between the **Lda_Def** and server context as often as necessary in the application.

This approach allows an application to use a single connection, but two different APIs, to accomplish different tasks.

You can mix and match OCI 7.x and OCI 8.0 calls within a transaction, but *not* within a statement. So you can execute one SQL or PL/SQL statement with OCI 7.x

calls and the next SQL or PL/SQL statement within that transaction with OCI 8.0 calls.

Warning: You can *not* open a cursor, and parse with OCI 7.x calls and then execute the statement with OCI 8.0 calls.

Application Linking Issues

This section discusses issues related to application linking, including the use of non-deferred linking and single-task linking with various OCI versions.

Non-deferred linking

Application developers are cautioned that Oracle plans to desupport non-deferred mode linking beginning with the release of Oracle9 (it will continue to be supported with all the releases of Oracle8). Recognizing these plans, application developers should no longer use non-deferred mode linking in developing new applications. Version 7.3 of the OCI supports two linking modes:

1. **Non-deferred linking:** The Oracle6 OCI (client) only supported non-deferred linking which meant that for each SQL statement, a parse, a bind and a define call were each executed separately with individual round trips between the client and the server. This significantly increased network traffic between the client and the server and reduced both the performance and scalability of OCI applications.
2. **Deferred linking:** Unlike the Oracle6 OCI, the Oracle7 OCI supports both non-deferred linking and deferred linking. Deferred mode linking essentially defers the bind and define steps until the statement executes - that is it automatically bundles and defers the bind and define calls to execution time. Further, when the application is linked with deferred mode and a special parsing call is used (the OPARSE call with the DEFFLG set to a non-zero value), even the parse call can be deferred to execution time. Note that deferred mode linking does not depend on the specific OCI calls that the application uses, only on the link option that is selected.

Deferred mode linking therefore significantly reduces the number of round trips between the client and the server and as a result improves the performance and scalability of OCI applications. The default behavior of Oracle7 OCI connected to the Oracle7 server is deferred mode linking. However, Oracle7 OCI also supports non-deferred linking by setting specific link time options.

Further, Oracle8 OCI has two types of calls: first, all the Oracle7 OCI calls are supported with Oracle8 OCI i.e. they will work with a Oracle8 OCI client by

relinking the version 8 OCI libraries. Second, there are additional Oracle8-specific OCI calls. The default mode with the first type of calls continues to be deferred mode linking; however, non-deferred mode linking is supported for these calls through all releases of Oracle8 by setting link time options. However, Oracle8-specific calls use a different paradigm and as a result non-deferred mode linking is not necessary.

The various combinations of client-side libraries and server with which non-deferred linking is currently supported are summarized in the following table:

Table A–4 Supported Linking Modes for Various Client and Server Versions

Client Server	Oracle6 OCI	Oracle7 OCI	Oracle8 OCI (7.x calls)	Oracle8 OCI (8.0 calls)	Oracle9 OCI
Oracle9	Not supported	Default: deferred Non-deferred supported	Default: deferred Non-deferred supported	Not supported	Not supported
Oracle8	Not supported	Default: deferred Non-deferred supported	Default: deferred Non-deferred supported	Not supported	Not supported
Oracle7	Non-deferred mode only	Default: deferred Non-deferred supported	Default: deferred Non-deferred supported	Not supported	Not supported
Oracle6	Non-deferred mode only	Default: deferred Non-deferred supported	Not supported	Not supported	Not supported

Oracle will continue to support deferred-mode linking with all the releases of Oracle8 (all 8.* releases). This has the following implications:

Applications using Oracle6 OCI libraries

Since the Oracle6 OCI library is not supported against the Oracle8 database, applications using the Oracle6 OCI library cannot be run against an Oracle8 database.

Applications using Oracle7 OCI libraries

Applications using Oracle7 OCI libraries can run in two configurations against an Oracle8 database:

1. They can be run with Oracle7 OCI libraries against an Oracle8 database in non-deferred mode provided link time options are set appropriately.
2. They can also be relinked with the Oracle8 OCI libraries and run in non-deferred mode provided link time options are set appropriately. Oracle will support the first configuration through all the releases of Oracle8. However, the second configuration will not be supported in Oracle9. Therefore, applications that require non-deferred linking will not be able to upgrade to Oracle9 client-side libraries.

Applications using Oracle8 OCI libraries

Applications using Oracle8- specific OCI calls, such as those used to access Oracle8's object types, do not need to use non-deferred mode linking since the Oracle8 OCI uses a different paradigm. Applications using only Oracle7 OCI calls will be able to use non-deferred mode linking but only through release 8.1

Single-task linking

Single-task linking is a feature used by a limited number of Oracle's customers, primarily on the OpenVMS platform. Some Oracle platforms support single-task linking, others no longer support it. Application developers are cautioned that Oracle will desupport single task on ALL platforms beginning with the first server release after Oracle8. Oracle will continue to support single-task linking for all Oracle8.x releases on those platforms that do support it today. Application developers are referred to the product-line specific documentation to determine whether or not their platform supports single-task linking today.

With single-task linking, Oracle supports two configurations to link Oracle products and user-written applications against the Oracle database:

1. **Single-task linking:** In this case, applications are directly linked against the Oracle shareable image making single-task connection to Oracle
2. **Two-task linking:** In this case, applications linked in a standalone configuration can only connect to Oracle using Net8's two task drivers such as Net8 DECnet or Net8 VMS Mailbox on the OpenVMS platform. This is the typical configuration used in the large majority of client-server applications. With two task linking applications and tools connect with the Oracle7 database through a programmatic interface that creates a shadow process for each user process. This shadow process runs a copy of the Oracle shareable image on behalf of the user process using Net8 protocols to communicate between the user and shadow processes. Therefore, with this interface, user routines that invoke the

Oracle7 Server functions run as one process or task, and the Oracle7 routines that execute these functions run as the second task.

Oracle will continue to support single-task linking with all the releases of Oracle8 (all 8.* releases) but will desupport it beginning with the first release after Oracle8. Application developers who would like to use single-task linking to run their applications will not be able to do so against the first server release after Oracle8.

Handle and Descriptor Attributes

This Appendix describes attributes for OCI handles and descriptors, which can be read with *OCIAttrGet()*, and can be modified with *OCIAttrSet()*.

The following handle types are included:

- Environment Handle Attributes
- Service Context Handle Attributes
- Server Handle Attributes
- User Session Handle Attributes
- Transaction Handle Attributes
- Statement Handle Attributes
- Bind Handle Attributes
- Define Handle Attributes
- Describe Handle Attributes
- Parameter Descriptor Attributes
- LOB Locator Attributes
- Complex Object Attributes
- Advanced Queueing Descriptor Attributes

Conventions

For each handle type, the attributes which can be read or changed are listed.

Each attribute listing includes the following information:

Mode

The following modes are possible:

READ - the attribute can be read using *OCIAttrGet()*

WRITE - the attribute can be modified using *OCIAttrSet()*

READ/WRITE - the attribute can be read using *OCIAttrGet()*, and it can be modified using *OCIAttrSet()*.

Description

This is a description of the purpose of the attribute.

Attribute Datatype

This is the datatype of the attribute. If necessary, a distinction is made between the datatype for READ and WRITE modes.

Possible Values

In some cases, only certain values are allowed, and they are listed here.

Example

In some cases an example is included.

Environment Handle Attributes

OCI_ATTR_CACHE_MAX_SIZE

Mode

READ/WRITE

Description

Sets the maximum size (high watermark) for the client-side object cache as a percentage of the optimal size. The default value is 10%. See the section “Object Cache Parameters” on page 11-5 for more information.

Attribute Datatype

ub4 *

OCI_ATTR_CACHE_OPT_SIZE

Mode

READ/WRITE

Description

Sets the optimal size for the client-side object cache in bytes. The default value is 200k bytes. See the section “Object Cache Parameters” on page 11-5 for more information.

Attribute Datatype

ub4 *

OCI_ATTR_OBJECT

Mode

READ

Description

Returns TRUE if the environment was initialized in object mode.

Attribute Datatype

boolean *

OCI_ATTR_FNCODE

Mode
READ

Description

Returns the function code of the last OCI operation on a handle. Each OCI function has a **ub4** value.

The OCI function codes are listed in Table B-1 on page B - 5.

Attribute Datatype
ub4 *

OCI_ATTR_PINOPTION

Mode
READ/WRITE

Description

This attribute sets the value of OCI_PIN_DEFAULT for the application associated with the environment handle.

For example, if OCI_ATTR_PINOPTION is set to OCI_PIN_RECENT, then if *OCIObjectPin()* is called with the *pin_option* parameter set to OCI_PIN_DEFAULT, then the object is pinned in OCI_PIN_RECENT mode.

Attribute Datatype
OCIPinOpt *

OCI_ATTR_ALLOC_DURATION

Mode
READ/WRITE

Description

This attribute sets the value of OCI_DURATION_DEFAULT for allocation durations for the application associated with the environment handle.

Attribute Datatype
OCIDuration *

Table B–1 OCI Function Codes

#	OCI Routine	#	OCI Routine	#	OCI Routine
1	OCIInitialize	27	OCIDefineArrayOfStruct	53	(NOT USED)
2	OCIHandleAlloc	28	OCISmtFetch	54	OCIAttrGet
3	OCIHandleFree	29	OCISmtGetBindInfo	55	OCIAttrSet
4	OCIDescriptorAlloc	30	(NOT USED)	56	OCIParmSet
5	OCIDescriptorFree	31	(NOT USED)	57	OCIParmGet
6	OCIEnvInit	32	OCIDescribeAny	58	OCISmtGetPieceInfo
7	OCIServerAttach	33	OCITransStart	59	OCILdaToSvcCtx
8	OCIServerDetach	34	OCITransDetach	60	(NOT USED)
9	(NOT USED)	35	OCITransCommit	61	OCISmtSetPieceInfo
10	OCISessionBegin	36	(NOT USED)	62	OCITransForget
11	OCISessionEnd	37	OCLErrorGet	63	OCITransPrepare
12	OCIPasswordChange	38	OCILobFileOpen	64	OCITransRollback
13	OCISmtPrepare	39	OCILobFileClose	65	OCIDefineByPos
14	(NOT USED)	40	(NOT USED)	66	OCIBindByPos
15	(NOT USED)	41	(NOT USED)	67	OCIBindByName
16	(NOT USED)	42	OCILobCopy	68	OCILobAssign
17	OCIBindDynamic	43	OCILobAppend	69	OCILobIsEqual
18	OCIBindObject	44	OCILobErase	70	OCILobLocatorIsInit
19	(NOT USED)	45	OCILobGetLength	71	OCILobEnableBuffering
20	OCIBindArrayOfStruct	46	OCILobTrim	72	OCILobCharSetID
21	OCISmtExecute	47	OCILobRead	73	OCILobCharSetForm
22	(NOT USED)	48	OCILobWrite	74	OCILobFileSetName
23	(NOT USED)	49	(NOT USED)	75	OCILobFileGetName
24	(NOT USED)	50	OCIBreak	76	OCILogon
25	OCIDefineObject	51	OCIServerVersion	77	OCILogoff
26	OCIDefineDynamic	52	(NOT USED)	78	OCILobDisableBuffering
				79	OCILobFlushBuffer
				80	OCILobLoadFromFile

OCI_ATTR_PIN_DURATION

Mode

READ/WRITE

Description

This attribute sets the value of OCI_DURATION_DEFAULT for pin durations for the application associated with the environment handle.

Attribute Datatype

OCIDuration *

Service Context Handle Attributes

OCI_ATTR_SQLCODE

Mode

READ

Description

Returns the code of the last SQL command processed on the service context handle. Each SQL command has a **ub4** value.

The SQL command codes are listed in Table B-2 on page B - 8.

Attribute Datatype**ub2 ***

OCI_ATTR_ENV

Mode

READ

Description

returns the environment context associated with the service context.

Attribute Datatype**OCIEnv ****

OCI_ATTR_SERVER

Mode

READ/WRITE

Description

When read, returns the pointer to the server context attribute of the service context.

When changed, sets the server context attribute of the service context.

Attribute Datatype**OCIServer ** (READ) / OCIServer * (WRITE)**

Table B–2 SQL Command Codes

Code	SQL Function	Code	SQL Function	Code	SQL Function
01	CREATE TABLE	35	LOCK	69	(NOT USED)
02	SET ROLE	36	NOOP	70	ALTER RESOURCE COST
03	INSERT	37	RENAME	71	CREATE SNAPSHOT LOG
04	SELECT	38	COMMENT	72	ALTER SNAPSHOT LOG
05	UPDATE	39	AUDIT	73	DROP SNAPSHOT LOG
06	DROP ROLE	40	NO AUDIT	74	CREATE SNAPSHOT
07	DROP VIEW	41	ALTER INDEX	75	ALTER SNAPSHOT
08	DROP TABLE	42	CREATE EXTERNAL DATABASE	76	DROP SNAPSHOT
09	DELETE	43	DROP EXTERNAL DATABASE	77	CREATE TYPE
10	CREATE VIEW	44	CREATE DATABASE	78	DROP TYPE
11	DROP USER	45	ALTER DATABASE	79	ALTER ROLE
12	CREATE ROLE	46	CREATE ROLLBACK SEGMENT	80	ALTER TYPE
13	CREATE SEQUENCE	47	ALTER ROLLBACK SEGMENT	81	CREATE TYPE BODY
14	ALTER SEQUENCE	48	DROP ROLLBACK SEGMENT	82	ALTER TYPE BODY
15	(NOT USED)	49	CREATE TABLESPACE	83	DROP TYPE BODY
16	DROP SEQUENCE	50	ALTER TABLESPACE	84	DROP LIBRARY
17	CREATE SCHEMA	51	DROP TABLESPACE	85	TRUNCATE TABLE
18	CREATE CLUSTER	52	ALTER SESSION	86	TRUNCATE CLUSTER
19	CREATE USER	53	ALTER USER	87	CREATE BITMAPFILE
20	CREATE INDEX	54	COMMIT (WORK)	88	ALTER VIEW
21	DROP INDEX	55	ROLLBACK	89	DROP BITMAPFILE
22	DROP CLUSTER	56	SAVEPOINT	90	SET CONSTRAINTS
23	VALIDATE INDEX	57	CREATE CONTROL FILE	91	CREATE FUNCTION
24	CREATE PROCEDURE	58	ALTER TRACING	92	ALTER FUNCTION
25	ALTER PROCEDURE	59	CREATE TRIGGER	93	DROP FUNCTION
26	ALTER TABLE	60	ALTER TRIGGER	94	CREATE PACKAGE
27	EXPLAIN	61	DROP TRIGGER	95	ALTER PACKAGE
28	GRANT	62	ANALYZE TABLE	96	DROP PACKAGE
29	REVOKE	63	ANALYZE INDEX	97	CREATE PACKAGE BODY
30	CREATE SYNONYM	64	ANALYZE CLUSTER	98	ALTER PACKAGE BODY
31	DROP SYNONYM	65	CREATE PROFILE	99	DROP PACKAGE BODY
32	ALTER SYSTEM SWITCH LOG	66	DROP PROFILE	157	CREATE DIRECTORY
33	SET TRANSACTION	67	ALTER PROFILE	158	DROP DIRECTORY
34	PL/SQL EXECUTE	68	DROP PROCEDURE	159	CREATE LIBRARY

OCI_ATTR_SESSION

Mode

READ/WRITE

Description

When read, returns the pointer to the authentication context attribute of the service context.

When changed, sets the authentication context attribute of the service context.

Attribute Datatype

OCISession ** (READ) / **OCISession** * (WRITE)

OCI_ATTR_TRANS

Mode

READ/WRITE

Description

When read, returns the pointer to the transaction context attribute of the service context.

When changed, sets the transaction context attribute of the service context.

Attribute Datatype

OCITrans ** (READ) / **OCITrans** * (WRITE)

OCI_ATTR_IN_V8_MODE

Mode

READ

Description

Allows you to determine whether an application has switched to Oracle7 mode (e.g., through an *OCISvcCtxToLda()* call). A non-zero (true) return value indicates that the application is currently running in Oracle8 mode, a zero (false) return value indicates that the application is currently running in Oracle7 mode.

Attribute Datatype

ub1 *

Example

The following code sample shows how this parameter might be used:

```
in_v8_mode = 0;
OCIAttrGet ((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (ub1 *)&in_v8_mode,
            (ub4) 0, OCI_ATTR_IN_V8_MODE, errhp);
if (in_v8_mode)
    fprintf (stdout, "In V8 mode\n");
else
    fprintf (stdout, "In V7 mode\n");
```

Server Handle Attributes

OCI_ATTR_ENV

Mode

READ

Description

Returns the environment context associated with the server context.

Attribute Datatype

OCIEnv **

OCI_ATTR_FNCODE

Mode

READ

Description

Returns the function code of the last OCI operation on a handle. Each OCI function has a **ub4** value.

The OCI function codes are listed in Table B-1 on page B-5.

Attribute Datatype**ub4** *

OCI_ATTR_EXTERNAL_NAME

Mode

READ/WRITE

Description

The external name is the user-friendly global name stored in `sys.props$.value$` where `name = 'GLOBAL_DB_NAME'`. It is not guaranteed to be unique unless all databases register their names with a network directory service.

Database names can be exchanged with the server in case of distributed transaction coordination. Server database names can only be accessed if the database is open at the time the *OCISessionBegin()* call is issued.

Attribute Datatype**text** ** (READ) / **text** * (WRITE)**OCI_ATTR_INTERNAL_NAME****Mode**

READ/WRITE

Description

Sets the client database name that will be recorded when performing global transactions. The name can be used by the DBA to track transactions that may be pending in a prepared state due to failures.

Attribute Datatype**text** ** (READ) / **text** * (WRITE)**OCI_ATTR_IN_V8_MODE****Mode**

READ

Description

Allows you to determine whether an application has switched to Oracle7 mode (e.g., through an *OCISvcCtxToLda()* call). A non-zero (true) return value indicates that the application is currently running in Oracle8 mode, a zero (false) return value indicates that the application is currently running in Oracle7 mode.

Attribute Datatype**ub1** ***OCI_ATTR_FOCBK****Mode**

READ/WRITE

Description

See “Application Failover Callbacks” on page 7-36 for more information.

Attribute Datatype**OCIFocbkStruct** *

User Session Handle Attributes

OCI_ATTR_USERNAME

Mode

WRITE

Description

Specifies a username to use for authentication.

Attribute Datatype

text *

OCI_ATTR_PASSWORD

Mode

WRITE

Description

Specifies a password to use for authentication.

Attribute Datatype

text *

Transaction Handle Attributes

OCI_ATTR_TRANS_NAME

Mode

READ/WRITE

Description

Can be used to establish or read a text string which identifies a transaction. This is an alternative to using the XID to identify the transaction. The text string can be up to 64 bytes long.

Attribute Datatype

text ** (READ) / **text** * (WRITE)

OCI_ATTR_XID

Mode

READ/WRITE

Description

Can set or read an XID which identifies a transaction.

Attribute Datatype

XID ** (READ) / **XID** * (WRITE)

Statement Handle Attributes

OCI_ATTR_FNCODE

Mode

READ

Description

Returns the function code of the last OCI operation on a handle. Each OCI function has a **ub4** value. The OCI function codes are listed in Table B-1 on page B-5.

Attribute Datatype

ub4 *

OCI_ATTR_ROW_COUNT

Mode

READ

Description

Returns the number of rows processed so far. The default value is 1.

Attribute Datatype

ub4 *

OCI_ATTR_SQLFNCODE

Mode

READ

Description

Returns the function code of the SQL command associated with the statement.

Attribute Datatype

ub2 *

Notes

The SQL command codes are listed in Table B-2 on page B - 8.

OCI_ATTR_ENV

Mode

READ

Description

Returns the environment context associated with the statement.

Attribute Datatype

OCIEnv **

OCI_ATTR_STMT_TYPE

Mode

READ

Description

The type of statement associated with the handle. Possible values are:

- OCI_STMT_SELECT
- OCI_STMT_UPDATE
- OCI_STMT_DELETE
- OCI_STMT_INSERT
- OCI_STMT_CREATE
- OCI_STMT_DROP
- OCI_STMT_ALTER
- OCI_STMT_BEGIN (PL/SQL statement)
- OCI_STMT_DECLARE (PL/SQL statement)

Attribute Datatype

ub2 *

OCI_ATTR_ROWID

Mode
READ

Description

Returns the rowid of the current row inserted, updated or fetched in a character string format. If execute had been a multiple row operation then, **len** should contain the iteration number of the row the application is interested in. When connected to an Oracle7 Server only the rowid of the last row inserted, updated, or fetched can be obtained.

Attribute Datatype
OCIRowid **

OCI_ATTR_PARAM_COUNT

Mode
READ

Description

This attribute can be used to get the number of columns in the select-list for the statement associated with the statement handle.

Attribute Datatype
ub4 *

Example

The following code sample shows how this attribute might be used:

```
/* Describe of a select-list */
text *selstmt = "SELECT * FROM EMP";
ub4 paramcnt;
OCIParam *parmdp;

err = OCIStmtPrepare (stmhp, errhp, selstmt,
                     (ub4)strlen((char *)selstmt),
                     (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);

err = OCIStmtExecute (svchp, stmhp, errhp, (ub4)1, (ub4)0,
                     (const OCISnapshot*) 0, (OCISnapshot*)0, OCI_DESCRIBE_ONLY);
```

```
/* get the number of columns in the select list */
err = OCIAttrGet ((dvoid *)stmhp, (ub4)OCI_HTYPE_STMT, (dvoid *)
                 &parmcnt, (ub4 *) 0, (ub4)OCI_ATTR_PARAM_COUNT, errhp);

/* get describe information for each column */
for (i = 0; i < parmcnt; i++) {
    OCIParamGet (dvoid *)stmhp, OCI_HTYPE_STMT, errhp, &parmdp, i);
/* get the attributes for each column */
}
```

OCI_ATTR_PREFETCH_ROWS

Mode

WRITE

Description

Sets the number of top level rows to be prefetched. The default value is 1 row.

Attribute Datatype

ub4 *

OCI_ATTR_PREFETCH_MEMORY

Mode

WRITE

Description

Sets the memory level for top level rows to be prefetched. Rows up to the specified top level row count are fetched if it occupies no more than the specified memory usage limit. The default value is 0, which means that memory size is not included in computing the number of rows to prefetch.

Attribute Datatype

ub4 *

Bind Handle Attributes

OCI_ATTR_FNCODE

Mode

READ

Description

Returns the function code of the last OCI operation on a handle. Each OCI function has a **ub4** value. The OCI function codes are listed in Table B-1 on page B-5.

Attribute Datatype**ub4 ***

OCI_ATTR_CHAR_COUNT

Mode

WRITE

Description

See “Character Count Attribute” on page 5-26.

Attribute Datatype**ub4 ***

OCI_ATTR_CHARSET_ID

Mode

READ/WRITE

Description

Character set ID of the bind handle.

Attribute Datatype**ub2 ***

OCI_ATTR_CHARSET_FORM

Mode

READ/WRITE

Description

Character set form of the bind handle.

Attribute Datatype

ub1 *

OCI_ATTR_MAXDATA_SIZE

Mode

READ/WRITE

Description

See “OCI_ATTR_MAXDATA_SIZE Attribute” on page 5-26.

Attribute Datatype

sb4 *

OCI_ATTR_PDSCL

Mode

WRITE

Description

Sets the number of digits to the right of the decimal point for fields where the data type is SQLT_PDN.

Attribute Datatype

ub2 *

OCI_ATTR_PDFMT

Mode

WRITE

Description

Specifies a format string.

Attribute Datatype

text **

OCI_ATTR_ROWS_RETURNED

Mode

READ

Description

This attribute returns the number of rows that are going to be returned in the current iteration when we are in the OUT callback function for binding a DML statement with RETURNING clause.

Attribute Datatype

ub4 *

Define Handle Attributes

OCI_ATTR_FNCODE

Mode

READ

Description

Returns the function code of the last OCI operation on a handle. Each OCI function has a **ub4** value.

The OCI function codes are listed in Table B-1 on page B-5.

Attribute Datatype

ub4 *

OCI_ATTR_CHAR_COUNT

Mode

WRITE

Description

Sets the number of characters in a character type data. This specifies the number of characters desired in the define buffer. The define buffer length as specified in the define call must be greater than number of characters.

Attribute Datatype

ub4 *

OCI_ATTR_CHARSET_ID

Mode

READ/WRITE

Description

The character set ID of the define handle.

Attribute Datatype

ub2 *

OCI_ATTR_CHARSET_FORM

Mode
READ/WRITE

Description
The character set form of the define handle.

Attribute Datatype
ub1 *

OCI_ATTR_PDSCL

Mode
WRITE

Description
Sets the number of digits to the right of the decimal point for fields where the data type is SQLT_PDN.

Attribute Datatype
ub2 *

OCI_ATTR_PDFMT

Mode
WRITE

Description
Specifies a format string.

Attribute Datatype
text **

Describe Handle Attributes

OCI_ATTR_PARAM_COUNT

Mode

READ

Description

Returns the number of parameters in the describe handle. When the describe handle is a description of the select list, this refers to the number of columns in the select list.

Attribute Datatype

ub4 *

Parameter Descriptor Attributes

For a detailed list of parameter descriptor attributes, refer to Chapter 6, “Describing Schema Metadata”.

LOB Locator Attributes

OCI_ATTR_LOBEMPTY

Mode

WRITE

Description

Sets the internal LOB locator to empty. The locator can then be used as a bind variable for an INSERT or UPDATE statement to initialize the LOB to empty. Once the LOB is empty, *OCIlobWrite()* can be called to populate the LOB with data. This attribute is only valid for internal LOBs (i.e., BLOB, CLOB, NCLOB).

Applications should pass address of a **ub4** which has a value of 0; e.g., declare

```
ub4 lobEmpty = 0
```

then pass address &lobEmpty.

Attribute Datatype**ub4 ***

Complex Object Attributes

For information about complex object retrieval, see “Complex Object Retrieval” on page 8-21.

Complex Object Retrieval Handle Attributes

OCI_ATTR_COMPLEXOBJECT_LEVEL

Mode

WRITE

Description

The depth level for complex object retrieval.

Attribute Datatype

ub4 *

OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE

Mode

WRITE

Description

Whether to fetch collection attributes in an object type out-of-line.

Attribute Datatype

ub1 *

Complex Object Retrieval Descriptor Attributes

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE

Mode

WRITE

Description

A type of REF to follow for complex object retrieval.

Attribute Datatype

dvoid *

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL

Mode

WRITE

Description

Depth level for following REFs of type OCI_ATTR_COMPLEXOBJECT_COMP_TYPE.

Attribute Datatype

ub4 *

Advanced Queueing Descriptor Attributes

For more information about Advanced Queueing, properties, and options, refer to the Advanced Queueing chapter of the *Oracle8 Application Developer's Guide*.

OCIAQEnqOptions Descriptor Attributes

The following attributes are properties of the **OCIAQEnqOptions** descriptor:

OCI_ATTR_RELATIVE_MSGID

Mode

READ/WRITE

Description

Specifies the message identifier of the message which is referenced in the sequence deviation operation. This value is valid if and only if OCI_ENQ_BEFORE is specified in OCI_ATTR_SEQUENCE_DIVISION. This value is ignored if the sequence deviation is not specified.

Attribute Datatype

OCIRaw *

OCI_ATTR_SEQUENCE_DEVIATION

Mode

READ/WRITE

Description

Specifies whether the message being enqueued should be dequeued before other message(s) already in the queue.

Attribute Datatype

ub4

Possible Values

The only valid values are:

- OCI_ENQ_BEFORE - the message is enqueued ahead of the message specified by OCI_ATTR_RELATIVE_MSGID.

- **OCI_ENQ_TOP** - the message is enqueued ahead of any other messages.

OCI_ATTR_VISIBILITY

Mode

READ/WRITE

Description

Specifies the transactional behavior of the enqueue request.

Attribute Datatype

ub4

Possible Values

The only valid values are:

- **OCI_ENQ_ON_COMMIT** - the enqueue is part of the current transaction. The operation is complete when the transaction commits. This is the default case.
- **OCI_ENQ_IMMEDIATE** - the enqueue is not part of the current transaction. The operation constitutes a transaction of its own.

OCIAQDeqOptions Descriptor Attributes

The following attributes are properties of the **OCIAQDeqOptions** descriptor:

OCI_ATTR_CONSUMER_NAME

Mode

READ/WRITE

Description

Name of the consumer. Only those messages matching the consumer name are accessed. If a queue is not set up for multiple consumers, this field should be set to **NULL**.

Attribute Datatype

text *

OCI_ATTR_CORRELATION

Mode

READ/WRITE

Description

Specifies the correlation identifier of the message to be dequeued. Special pattern matching characters, such as the percent sign (%) and the underscore (_) can be used. If more than one message satisfies the pattern, the order of dequeuing is undetermined.

Attribute Datatype

text *

OCI_ATTR_DEQ_MODE

Mode

READ/WRITE

Description

Specifies the locking behavior associated with the dequeue.

Attribute Datatype

ub4

Possible Values

The only valid values are:

- OCI_DEQ_BROWSE - read the message without acquiring any lock on the message. This is equivalent to a SELECT statement.
- OCI_DEQ_LOCKED - read and obtain a write lock on the message. The lock lasts for the duration of the transaction. This is equivalent to a SELECT FOR UPDATE statement.
- OCI_DEQ_REMOVE - read the message and update or delete it. This is the default. The message can be retained in the queue table based on the retention properties.

OCI_ATTR_DEQ_MSGID

Mode

READ/WRITE

Description

Specifies the message identifier of the message to be dequeued.

Attribute Datatype

OCIRaw *

OCI_ATTR_NAVIGATION

Mode

READ/WRITE

Description

Specifies the position of the message that will be retrieved. First, the position is determined. Second, the search criterion is applied. Finally, the message is retrieved.

Attribute Datatype

ub4

Possible Values

The only valid values are:

- OCI_DEQ_FIRST_MSG - retrieves the first message which is available and matches the search criteria. This will reset the position to the beginning of the queue.
- OCI_DEQ_NEXT_MSG - retrieves the next message which is available and matches the search criteria. If the previous message belongs to a message group, AQ will retrieve the next available message which matches the search criteria and belongs to the message group. This is the default.
- OCI_DEQ_NEXT_TRANSACTION - skips the remainder of the current transaction group (if any) and retrieves the first message of the next transaction group. This option can only be used if message grouping is enabled for the current queue.

OCI_ATTR_VISIBILITY

Mode

READ/WRITE

Description

Specifies whether the new message is dequeued as part of the current transaction. The visibility parameter is ignored when using the `BROWSE` mode.

Attribute Datatype

ub4

Possible Values

The only valid values are:

- `OCI_DEQ_ON_COMMIT` - the dequeue will be part of the current transaction. This is the default case.
- `OCI_DEQ_IMMEDIATE` - the dequeued message is not part of the current transaction. It constitutes a transaction on its own.

OCI_ATTR_WAIT

Mode

READ/WRITE

Description

Specifies the wait time if there is currently no message available which matches the search criteria. This parameter is ignored if messages in the same group are being dequeued.

Attribute Datatype

ub4

Possible Values

Any ub4 value is valid, but the following predefined constants are provided:

- `OCI_DEQ_WAIT_FOREVER` - wait forever. This is the default.
- `OCI_DEQ_NO_WAIT` - do not wait.

OCIAQMsgProperties Descriptor Attributes

The following attributes are properties of the **OCIAQMsgProperties** descriptor:

OCI_ATTR_ATTEMPTS

Mode
READ

Description
Specifies the number of attempts that have been made to dequeue the message.
This parameter cannot be set at enqueue time.

Attribute Datatype
sb4

Possible Values
Any sb4 value is valid.

OCI_ATTR_CORRELATION

Mode
READ/WRITE

Description
Specifies the identification supplied by the producer for a message at enqueueing.

Attribute Datatype
text *

Possible Values
Any string up to 128 bytes is valid.

OCI_ATTR_DELAY

Mode
READ/WRITE

Description
Specifies the number of seconds to delay the enqueued message. The delay represents the number of seconds after which a message is available for dequeuing. Dequeuing by msgid overrides the delay specification. A message enqueued with

delay set will be in the `WAITING` state, when the delay expires the messages goes to the `READY` state. `DELAY` processing requires the queue monitor to be started. Note that delay is set by the producer who enqueues the message.

Attribute Datatype**sb4****Possible Values**

Any sb4 value is valid, but the following predefined constant is available:

- `OCI_MSG_NO_DELAY` - indicates the message is available for immediate dequeuing.

OCI_ATTR_ENQ_TIME**Mode****READ****Description**

Specifies the time the message was enqueued. This value is determined by the system and cannot be set by the user.

Attribute Datatype**OCIDate****OCI_ATTR_EXCEPTION_QUEUE****Mode****READ/WRITE****Description**

Specifies the name of the queue to which the message is moved to if it cannot be processed successfully. Messages are moved in two cases: If the number of unsuccessful dequeue attempts has exceeded *max_retries*; or if the message has expired. All messages in the exception queue are in the `EXPIRED` state.

The default is the exception queue associated with the queue table. If the exception queue specified does not exist at the time of the move the message will be moved to the default exception queue associated with the queue table and a warning will be logged in the alert file. If the default exception queue is used, the parameter will return a `NULL` value at dequeue time.

This attribute must refer to a valid queue name.

Attribute Datatype
text *

OCI_ATTR_EXPIRATION

Mode
READ/WRITE

Description

Specifies the expiration of the message. It determines, in seconds, the duration the message is available for dequeuing. This parameter is an offset from the delay. Expiration processing requires the queue monitor to be running.

While waiting for expiration, the message remains in the **READY** state. If the message is not dequeued before it expires, it will be moved to the exception queue in the **EXPIRED** state.

Attribute Datatype
sb4

Possible Values

Any sb4 value is valid, but the following predefined constant is available:

- **OCI_MSG_NO_EXPIRATION** - the message will not expire.

OCI_ATTR_MSG_STATE

Mode
READ

Description

Specifies the state of the message at the time of the dequeue. This parameter cannot be set at enqueue time.

Attribute Datatype
ub4

Possible Values

These are the only values which are returned:

- **OCI_MSG_WAITING** - the message delay has not yet been reached.
- **OCI_MSG_READY** - the message is ready to be processed.

- `OCI_MSG_PROCESSED` - the message has been processed and is retained.
- `OCI_MSG_EXPIRED` - the message has been moved to the exception queue.

OCI_ATTR_PRIORITY

Mode

READ/WRITE

Description

Specifies the priority of the message. A smaller number indicates higher priority. The priority can be any number, including negative numbers.

The default value is zero.

Attribute Datatype

sb4

OCI_ATTR_RECIPIENT_LIST

Mode

WRITE

Description

This parameter is only valid for queues which allow multiple consumers. The default recipients are the queue subscribers. This parameter is not returned to a consumer at dequeue time.

Attribute Datatype

OCIAQAgent **

OCIAQAgent Descriptor Attributes

The following attributes are properties of the **OCIAQAgent** descriptor:

OCI_ATTR_AGENT_ADDRESS

Mode

READ/WRITE

Description

Protocol-specific address of the recipient. If the protocol is 0 (default), the address is of the form [schema.]queue[@dblink].

Attribute Datatype

text *

Possible Values

Can be any string up to 128 bytes.

OCI_ATTR_AGENT_NAME

Mode

READ/WRITE

Description

Name of a producer or consumer of a message.

Attribute Datatype

text *

Possible Values

Can be any Oracle identifier, up to 30 bytes.

OCI_ATTR_AGENT_PROTOCOL

Mode

READ/WRITE

Description

Protocol to interpret the address and propagate the message. The default (and currently the only supported) value is 0.

Attribute Datatype

ub1

Possible Values

The only valid value is zero, which is also the default.

Oracle Reserved Words, Keywords and Namespaces

This appendix lists words that have a special meaning to Oracle. Each word plays a specific role in the context in which it appears. For example, in an INSERT statement, the reserved word INTO introduces the tables to which rows will be added. But, in a FETCH or SELECT statement, the reserved word INTO introduces the output host variables to which column values will be assigned.

The following sections are included:

- Oracle Reserved Words and Keywords
- PL/SQL Reserved Words
- Oracle Reserved Namespaces

Oracle Reserved Words and Keywords

Oracle reserved words have a special meaning to Oracle and so cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes.

Keywords also have a special meaning to Oracle but are not reserved words and so can be redefined. However, some might eventually become reserved words, so care should be taken when using them as variable or function names in an application.

The following table lists the Oracle reserved words and keywords:

Table C–1 Keywords and Reserved Words

Word	Type	Word	Type
	Reserved word	AS	Reserved Word
&	Reserved word	ASC	Reserved Word
:	Reserved word	AT	Key Word
,	Reserved word	AUDIT	Reserved Word
-	Reserved word	AUTHENTICATED	Key Word
=	Reserved word	AUTHORIZATION	Key Word
>	Reserved word	AUTOEXTEND	Key Word
[Reserved word	AUTOMATIC	Key Word
<	Reserved word	AVG	Key Word
(Reserved word		
.	Reserved word	BACKUP	Key Word
+	Reserved word	BECOME	Key Word
]	Reserved word	BEFORE	Key Word
)	Reserved word	BEGIN	Key Word
!	Reserved word	BETWEEN	Reserved Word
/	Reserved word	BFILE	Key Word
*	Reserved word	BITMAP	Key Word
^	Reserved word	BLOB	Key Word
@	Reserved word	BLOCK	Key Word
		BODY	Key Word
ACCESS	Reserved Word	BY	Reserved Word
ACCOUNT	Key Word		
ACTIVATE	Key Word	CACHE	Key Word
ADD	Reserved Word	CACHE_INSTANCES	Key Word
ADMIN	Key Word	CANCEL	Key Word
ADVISE	Key Word	CASCADE	Key Word
AFTER	Key Word	CAST	Key Word
ALL	Reserved Word	CFILE	Key Word
ALL_ROWS	Key Word	CHAINED	Key Word
ALLOCATE	Key Word	CHANGE	Key Word
ALTER	Reserved Word	CHAR	Reserved Word
ANALYZE	Key Word	CHAR_CS	Key Word
AND	Reserved Word	CHARACTER	Key Word
ANY	Reserved Word	CHECK	Reserved Word
ARCHIVE	Key Word	CHECKPOINT	Key Word
ARCHIVELOG	Key Word	CHOOSE	Key Word
ARRAY	Key Word	CHUNK	Key Word
ARRAYLEN	Key Word	CLEAR	Key Word

Table C–1 Keywords and Reserved Words (Cont.)

Word	Type	Word	Type
CLOB	Key Word	DATAFILE	Key Word
CLONE	Key Word	DATAFILES	Key Word
CLOSE	Key Word	DATAOBJNO	Key Word
CLOSE_CACHED_OPEN_CURSORS	Key Word	DATE	Reserved Word
CLUSTER	Reserved Word	DBA	Key Word
COALESCE	Key Word		
COBOL	Key Word		
COLUMN	Reserved Word		
COLUMNS	Key Word	DEALLOCATE	Key Word
COMMENT	Reserved Word	DEBUG	Key Word
COMMIT	Key Word	DEC	Key Word
COMMITTED	Key Word	DECIMAL	Reserved Word
COMPATIBILITY	Key Word	DECLARE	Key Word
COMPILE	Key Word	DEFAULT	Reserved Word
COMPLETE	Key Word	DEFERRABLE	Key Word
COMPOSITE_LIMIT	Key Word	DEFERRED	Key Word
COMPRESS	Reserved Word	DEGREE	Key Word
COMPUTE	Key Word	DELETE	Reserved Word
CONNECT	Reserved Word	DEREF	Key Word
CONNECT_TIME	Key Word	DESC	Reserved Word
CONSTRAINT	Key Word	DIRECTORY	Key Word
CONSTRAINTS	Key Word	DISABLE	Key Word
CONTENTS	Key Word	DISCONNECT	Key Word
CONTINUE	Key Word	DISMOUNT	Key Word
CONTROLFILE	Key Word	DISTINCT	Reserved Word
CONVERT	Key Word	DISTRIBUTED	Key Word
COST	Key Word	DML	Key Word
COUNT	Key Word	DOUBLE	Key Word
CPU_PER_CALL	Key Word	DROP	Reserved Word
CPU_PER_SESSION	Key Word	DUMP	Key Word
CREATE	Reserved Word		
CURRENT	Reserved Word	EACH	Key Word
CURRENT_SCHEMA	Key Word	ELSE	Reserved Word
CURRENT_USER	Key Word	ENABLE	Key Word
CURSOR	Reserved Word	END	Key Word
CYCLE	Key Word	ENFORCE	Key Word
		ENTRY	Key Word
DANGLING	Key Word	ESCAPE	Key Word
DATABASE	Key Word	ESTIMATE	Key Word

Table C–1 Keywords and Reserved Words (Cont.)

Word	Type	Word	Type
EVENTS	Key Word	GO	Key Word
EXCEPT	Key Word	GOTO	Key Word
EXCEPTIONS	Key Word	GRANT	Reserved Word
EXCHANGE	Key Word	GROUP	Reserved Word
EXCLUDING	Key Word	GROUPS	Key Word
EXCLUSIVE	Reserved Word		
EXEC	Key Word	HASH	Key Word
EXECUTE	Key Word	HASHKEYS	Key Word
EXISTS	Reserved Word	HAVING	Reserved Word
EXPIRE	Key Word	HEADER	Key Word
EXPLAIN	Key Word	HEAP	Key Word
EXTENT	Key Word		
EXTENTS	Key Word	IDENTIFIED	Reserved Word
EXTERNALLY	Key Word	IDGENERATORS	Key Word
		IDLE_TIME	Key Word
FAILED_LOGIN_ATTEMPTS	Key Word	IF	Key Word
FALSE	Key Word	IMMEDIATE	Reserved Word
FAST	Key Word	IN	Reserved Word
FETCH	Key Word	INCLUDING	Key Word
FILE	Reserved Word	INCREMENT	Reserved Word
FIRST_ROWS	Key Word	INDEX	Reserved Word
FLAGGER	Key Word	INDEXED	Key Word
FLOAT	Reserved Word	INDEXES	Key Word
FLOB	Key Word	INDICATOR	Key Word
FLUSH	Key Word	IND_PARTITION	Key Word
FOR	Reserved Word	INITIAL	Reserved Word
FORCE	Key Word	INITIALLY	Key Word
FOREIGN	Key Word	INITRANS	Key Word
FORTRAN	Key Word	INSERT	Reserved Word
FOUND	Key Word	INSTANCE	Key Word
FREELIST	Key Word	INSTANCES	Key Word
FREELISTS	Key Word	INSTEAD	Key Word
FROM	Reserved Word	INT	Key Word
FULL	Key Word	INTEGER	Reserved Word
FUNCTION	Key Word	INTERMEDIATE	Key Word
		INTERSECT	Reserved Word
GLOBAL	Key Word	INTO	Reserved Word
GLOBALLY	Key Word	IS	Reserved Word
GLOBAL_NAME	Key Word	ISOLATION	Key Word

Table C–1 Keywords and Reserved Words (Cont.)

Word	Type	Word	Type
ISOLATION_LEVEL	Key Word	MAXSIZE	Key Word
KEEP	Key Word	MAXTRANS	Key Word
KEY	Key Word	MAXVALUE	Key Word
KILL	Key Word	MIN	Key Word
		MEMBER	Key Word
		MINIMUM	Key Word
LABEL	Key Word	MINEXTENTS	Key Word
LANGUAGE	Key Word	MINUS	Reserved Word
LAYER	Key Word	MINVALUE	Key Word
LESS	Key Word	MLSLABEL	Reserved Word
LEVEL	Reserved Word		
LIBRARY	Key Word		
LIKE	Reserved Word	MODE	Reserved Word
LIMIT	Key Word	MODIFY	Reserved Word
LINK	Key Word	MODULE	Key Word
LIST	Key Word	MOUNT	Key Word
LISTS	Key Word	MOVE	Key Word
LOB	Key Word	MTS_DISPATCHERS	Key Word
LOCAL	Key Word	MULTISET	Key Word
LOCK	Reserved Word		
LOCKED	Key Word	NATIONAL	Key Word
LOG	Key Word	NCHAR	Key Word
LOGFILE	Key Word	NCHAR_CS	Key Word
LOGGING	Key Word	NCLOB	Key Word
LOGICAL_READS_PER_CALL	Key Word	NEEDED	Key Word
LOGICAL_READS_PER_SESSION	Key Word	NESTED	Key Word
LONG	Reserved Word	NETWORK	Key Word
		NEW	Key Word
MANAGE	Key Word	NEXT	Key Word
MANUAL	Key Word	NOARCHIVELOG	Key Word
MASTER	Key Word	NOAUDIT	Reserved Word
MAX	Key Word	NOCACHE	Key Word
MAXARCHLOGS	Key Word	NOCOMPRESS	Reserved Word
MAXDATAFILES	Key Word	NOCYCLE	Key Word
MAXEXTENTS	Reserved Word	NOFORCE	Key Word
MAXINSTANCES	Key Word	NOLOGGING	Key Word
MAXLOGFILES	Key Word	NOMAXVALUE	Key Word
MAXLOGHISTORY	Key Word	NOMINVALUE	Key Word
MAXLOGMEMBERS	Key Word	NONE	Key Word

Table C–1 Keywords and Reserved Words (Cont.)

Word	Type	Word	Type
NOORDER	Key Word	PACKAGE	Key Word
NOOVERRIDE	Key Word	PACKED	Key Word
NOPARALLEL	Key Word	PARALLEL	Key Word
NORESETLOGS	Key Word	PARTITION	Key Word
NOREVERSE	Key Word	PASSWORD	Key Word
NORMAL	Key Word	PASSWORD_GRACE_TIME	Key Word
NOSORT	Key Word	PASSWORD_LIFE_TIME	Key Word
NOT	Reserved Word	PASSWORD_LOCK_TIME	Key Word
NOTFOUND	Reserved Word	PASSWORD_REUSE_MAX	Key Word
NOTHING	Key Word	PASSWORD_REUSE_TIME	Key Word
NOWAIT	Reserved Word	PASSWORD_VERIFY_FUNCTION	Key Word
NULL	Reserved Word	PCTFREE	Reserved Word
NUMBER	Reserved Word	PCTINCREASE	Key Word
NUMERIC	Key Word	PCTTHRESHOLD	Key Word
NVARCHAR2	Key Word	PCTUSED	Key Word
OBJECT	Key Word	PCTVERSION	Key Word
OBJNO	Key Word	PERCENT	Key Word
OBJNO_REUSE	Key Word	PERMANENT	Key Word
OF	Reserved Word	PLAN	Key Word
OFF	Key Word	PLI	Key Word
OFFLINE	Reserved Word	PLSQL_DEBUG	Key Word
OID	Key Word	POST_TRANSACTION	Key Word
OIDINDEX	Key Word	PRECISION	Key Word
OLD	Key Word	PRESERVE	Key Word
ON	Reserved Word	PRIMARY	Key Word
ONLINE	Reserved Word	PRIOR	Reserved Word
ONLY	Key Word	PRIVATE	Key Word
OPCODE	Key Word	PRIVATE_SGA	Key Word
OPEN	Key Word	PRIVILEGE	Key Word
OPTIMAL	Key Word	PRIVILEGES	Reserved Word
OPTIMIZER_GOAL	Key Word	PROCEDURE	Key Word
OPTION	Reserved Word	PROFILE	Key Word
OR	Reserved Word	PUBLIC	Reserved Word
ORDER	Reserved Word	PURGE	Key Word
ORGANIZATION	Key Word	QUEUE	Key Word
OVERFLOW	Key Word	QUOTA	Key Word
OWN	Key Word		

Table C–1 Keywords and Reserved Words (Cont.)

Word	Type	Word	Type
RANGE	Key Word	SCAN_INSTANCES	Key Word
RAW	Reserved Word	SCHEMA	Key Word
RBA	Key Word	SCN	Key Word
READ	Key Word	SCOPE	Key Word
		SD_ALL	Key Word
REAL	Key Word	SD_INHIBIT	Key Word
REBUILD	Key Word	SD_SHOW	Key Word
RECOVER	Key Word	SECTION	Key Word
RECOVERABLE	Key Word	SEGMENT	Key Word
RECOVERY	Key Word	SEG_BLOCK	Key Word
REF	Key Word	SEG_FILE	Key Word
REFERENCES	Key Word	SELECT	Reserved Word
REFERENCING	Key Word	SEQUENCE	Key Word
REFRESH	Key Word	SERIALIZABLE	Key Word
RENAME	Reserved Word	SESSION	Reserved Word
REPLACE	Key Word	SESSION_CACHED_CURSORS	Key Word
RESET	Key Word	SESSIONS_PER_USER	Key Word
RESETLOGS	Key Word	SET	Reserved Word
RESIZE	Key Word	SHARE	Reserved Word
RESOURCE	Reserved Word	SHARED	Key Word
RESTRICTED	Key Word	SHARED_POOL	Key Word
RETURN	Key Word	SHRINK	Key Word
RETURNING	Key Word	SIZE	Reserved Word
REUSE	Key Word	SKIP	Key Word
REVERSE	Key Word	SKIP_UNUSABLE_INDEXES	Key Word
REVOKE	Reserved Word	SMALLINT	Reserved Word
ROLE	Key Word	SNAPSHOT	Key Word
ROLES	Key Word	SOME	Key Word
ROLLBACK	Key Word	SORT	Key Word
ROW	Reserved Word	SPECIFICATION	Key Word
ROWID	Reserved Word	SPLIT	Key Word
ROWLABEL	Reserved Word	SQL	Key Word
ROWNUM	Reserved Word	SQLBUF	Reserved Word
ROWS	Reserved Word	SQLCODE	Key Word
RULE	Key Word	SQLERROR	Key Word
		SQLSTATE	Key Word
SAMPLE	Key Word	SQL_TRACE	Key Word
SAVEPOINT	Key Word	STANDBY	Key Word
SB4	Key Word	START	Reserved Word

Table C–1 Keywords and Reserved Words (Cont.)

Word	Type	Word	Type
STATEMENT_ID	Key Word	TX	Key Word
STATISTICS	Key Word	TYPE	Key Word
STOP	Key Word		
STORAGE	Key Word	UB2	Key Word
STORE	Key Word	UBA	Key Word
STRUCTURE	Key Word	UID	Reserved Word
SUCCESSFUL	Reserved Word	UNARCHIVED	Key Word
SUM	Key Word	UNDER	Key Word
SWITCH	Key Word	UNDO	Key Word
SYS_OP_ENFORCE_NOT_NULLS	Key Word	UNION	Reserved Word
SYS_OP_NTCIMGS	Key Word	UNIQUE	Reserved Word
SYNONYM	Reserved Word	UNLIMITED	Key Word
SYSDATE	Reserved Word	UNLOCK	Key Word
SYSDBA	Key Word	UNPACKED	Key Word
SYSOPER	Key Word	UNRECOVERABLE	Key Word
SYSTEM	Key Word	UNTIL	Key Word
		UNUSABLE	Key Word
TABLE	Reserved Word	UNUSED	Key Word
TABLES	Key Word	UPDATABLE	Key Word
TABLESPACE	Key Word	UPDATE	Reserved Word
TABLESPACE_NO	Key Word	USAGE	Key Word
TABNO	Key Word	USE	Key Word
TEMPORARY	Key Word	USER	Reserved Word
THAN	Key Word	USING	Key Word
THE	Key Word		
THEN	Reserved Word	VALIDATE	Reserved Word
THREAD	Key Word	VALIDATION	Reserved Word
TIMESTAMP	Key Word	VALUE	Reserved Word
TIME	Key Word	VALUES	Reserved Word
TO	Reserved Word	VARCHAR	Reserved Word
TOplevel	Key Word	VARCHAR2	Reserved Word
TRACE	Key Word	VARYING	Key Word
TRACING	Key Word	VIEW	Reserved Word
TRANSACTION	Key Word		
TRANSITIONAL	Key Word	WHEN	Key Word
TRIGGER	Reserved Word	WHENEVER	Reserved Word
TRIGGERS	Key Word	WHERE	Reserved Word
TRUE	Key Word	WITH	Reserved Word
TRUNCATE	Key Word	WITHOUT	Key Word

Table C–1 Keywords and Reserved Words (Cont.)

Word	Type	Word	Type
WORK	Key Word		
WRITE	Key Word		
XID	Key Word		

PL/SQL Reserved Words

For information about PL/SQL reserved words and keywords, refer to the *PL/SQL User’s Guide and Reference*.

Oracle Reserved Namespaces

Table C-2 contains a list of namespaces that are reserved by Oracle. The initial characters of function names in Oracle libraries are restricted to the character strings in this list. Because of potential name conflicts, use function names that do not begin with these characters.

For example, the SQL*Net Transparent Network Service functions all begin with the characters NS,” so you need to avoid naming functions that begin with “NS.”

Table C-2 Oracle Reserved Namespaces

Namespace	Library
XA	external functions for XA applications only
SQ	external SQLLIB functions used by Oracle Precompiler and SQL*Module applications
O, OCI	external OCI functions internal OCI functions
UPI, KP	function names from the Oracle UPI layer
NA	SQL*Net Native services product
NC	SQL*Net RPC project
ND	SQL*Net Directory
NL	SQL*Net Network Library layer
NM	SQL*Net Net Management Project
NR	SQL*Net Interchange
NS	SQL*Net Transparent Network Service
NT	SQL*Net Drivers
NZ	SQL*Net Security Service
OSN	SQL*Net V1
TTC	SQL*Net Two task
GEN, L, ORA	Core library functions
LI, LM, LX	function names from the Oracle NLS layer
S	function names from system-dependent libraries

The list in Table C-2 is not a comprehensive list of all functions within the Oracle Reserved Namespaces. For a complete list of functions within a particular namespace, refer to the document that corresponds to the appropriate Oracle library.

Code Examples

This Appendix contains code examples illustrating the use of OCI calls. These programs are provided for demonstration purposes, and are not guaranteed to run on all platforms. When a specific header or SQL file is required by the application, its listing is included after the application code.

These and other demonstration programs may be available in the demo directory of your Oracle installation.

- Example 1, SQL Processing
- Example 2, Object Retrieval
- Example 3, DML with RETURNING Clause
- Example 4, Describing an Object
- Example 5, CLOB/BLOB Operations
- Example 6, LOB Buffering
- Example 7, REF Pinning and Navigation

Example 1, SQL Processing

```

/*
 *      -- cdemo81.c --
 *  An example program which adds new employee
 *  records to the personnel data base. Checking
 *  is done to insure the integrity of the data base.
 *  The employee numbers are automatically selected using
 *  the current maximum employee number as the start.
 *
 *  The program queries the user for data as follows:
 *
 *  Enter employee name:
 *  Enter employee job:
 *  Enter employee salary:
 *  Enter employee dept:
 *
 *  The program terminates if return key (CR) is entered
 *  when the employee name is requested.
 *
 *  If the record is successfully inserted, the following is printed:
 *
 *  "ename" added to department "dname" as employee # "empno"
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>

static text *username = (text *) "SCOTT";
static text *password = (text *) "TIGER";

/* Define SQL statements to be used in program. */
static text *insert = (text *) "INSERT INTO emp(empno, ename, job, sal, \
    deptno)VALUES (:empno, :ename, :job, :sal, :deptno)";
static text *seldept = (text *) "SELECT dname FROM dept WHERE deptno = :1";
static text *maxemp = (text *) "SELECT NVL(MAX(empno), 0) FROM emp";
static text *selemp = (text *) "SELECT ename, job FROM emp";

static OCIEnv *envhp;
static OCIServer *srvhp;
static OCIError *errhp;
static OCISvcCtx *svchp;
static OCISmt *stmthp, *stmthp1;

```

```

static OCIDefine *defnp = (OCIDefine *) 0;

static OCIBind *bnd1p = (OCIBind *) 0;          /* the first bind handle */
static OCIBind *bnd2p = (OCIBind *) 0;          /* the second bind handle */
static OCIBind *bnd3p = (OCIBind *) 0;          /* the third bind handle */
static OCIBind *bnd4p = (OCIBind *) 0;          /* the fourth bind handle */
static OCIBind *bnd5p = (OCIBind *) 0;          /* the fifth bind handle */
static OCIBind *bnd6p = (OCIBind *) 0;          /* the sixth bind handle */

static void checkerr(/*_ OCIErrror *errhp, sword status _*/);
static void cleanup(/*_ void _*/);
static void myfflush(/*_ void _*/);
int main(/*_ int argc, char *argv[] _*/);

static sword status;

int main(argc, argv)
int argc;
char *argv[];
{

    sword      empno, sal, deptno;
    sword      len, len2, rv, dsize, dsize2;
    sb4        enamelen = 10;
    sb4        joblen = 9;
    sb4        deptlen = 14;
    sb2        sal_ind, job_ind;
    sb2        db_type, db2_type;
    sb1        name_buf[20], name2_buf[20];
    text       *cp, *ename, *job, *dept;

    sb2        ind[2];
    ub2        alen[2];
    ub2        rlen[2];
    OCIDescribe *dschndl1 = (OCIDescribe *) 0,
               *dschndl2 = (OCIDescribe *) 0,
               *dschndl3 = (OCIDescribe *) 0;
    OCISession *authp = (OCISession *) 0;

    (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                        (dvoid * (*)(dvoid *, size_t)) 0,
                        (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                        (void (*)(dvoid *, dvoid *)) 0 );

    (void) OCIEnvInit( (OCIEnv **) &envhp, OCI_DEFAULT, (size_t) 0,

```

```

        (dvoid **) 0 );

(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
        (size_t) 0, (dvoid **) 0);

/* server contexts */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
        (size_t) 0, (dvoid **) 0);

(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
        (size_t) 0, (dvoid **) 0);

(void) OCIServerAttach( srvhp, errhp, (text *)"inst1_alias",
        strlen("inst1_alias"), 0);

/* set attribute server context in the service context */
(void) OCIAttrSet( (dvoid *) svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp,
        (ub4) 0, OCI_ATTR_SERVER, (OCIError *) errhp);

(void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&authp,
        (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);

(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
        (dvoid *) username, (ub4) strlen((char *)username),
        (ub4) OCI_ATTR_USERNAME, errhp);

(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
        (dvoid *) password, (ub4) strlen((char *)password),
        (ub4) OCI_ATTR_PASSWORD, errhp);

checkerr(errhp, OCISessionBegin ( svchp, errhp, authp, OCI_CRED_RDBMS,
        (ub4) OCI_DEFAULT));

(void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
        (dvoid *) authp, (ub4) 0,
        (ub4) OCI_ATTR_SESSION, errhp);

checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
        OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp1,
        OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));

/* Retrieve the current maximum employee number. */
checkerr(errhp, OCISTmtPrepare(stmthp, errhp, maxemp,

```

```

(ub4) strlen((char *) maxemp),
(ub4) OCI_NIV_SYNTAX, (ub4) OCI_DEFAULT));

/* bind the input variable */
checkerr(errhp, OCIDefineByPos(stmthp, &defnp, errhp, 1, (dvoid *) &empno,
    (sword) sizeof(sword), SQLT_INT, (dvoid *) 0, (ub2 *) 0,
    (ub2 *) 0, OCI_DEFAULT));

/* execute and fetch */
if (status = OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
    (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT))
{
    if (status == OCI_NO_DATA)
        empno = 10;
    else
    {
        checkerr(errhp, status);
        cleanup();
        return OCI_ERROR;
    }
}

checkerr(errhp, OCISmtPrepare(stmthp, errhp, insert,
    (ub4) strlen((char *) insert),
    (ub4) OCI_NIV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtPrepare(stmthp1, errhp, seldept,
    (ub4) strlen((char *) seldept),
    (ub4) OCI_NIV_SYNTAX, (ub4) OCI_DEFAULT));

/* Allocate output buffers. Allow for \n and '\0'. */
ename = (text *) malloc((size_t) enamelen + 2);
job = (text *) malloc((size_t) joblen + 2);

/* Bind the placeholders in the INSERT statement. */
if ((status = OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":ENAME",
    -1, (dvoid *) ename,
    enamelen+1, SQLT_STR, (dvoid *) 0,
    (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT)) ||
    (status = OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":JOB",
    -1, (dvoid *) job,
    joblen+1, SQLT_STR, (dvoid *) &job_ind,
    (ub2 *) 0, (ub2 *) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT)) ||

```

```

        (status = OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":SAL",
            -1, (dvoid *) &sal,
            (sword) sizeof(sal), SQLT_INT, (dvoid *) &sal_ind,
            (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT)) ||
        (status = OCIBindByName(stmthp, &bnd4p, errhp, (text *) ":DEPTNO",
            -1, (dvoid *) &deptno,
            (sword) sizeof(deptno), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT)) ||
        (status = OCIBindByName(stmthp, &bnd5p, errhp, (text *) ":EMPNO",
            -1, (dvoid *) &empno,
            (sword) sizeof(empno), SQLT_INT, (dvoid *) 0,
            (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT)))
    {
        checkerr(errhp, status);
        cleanup();
        return OCI_ERROR;
    }

    /* Bind the placeholder in the "seldept" statement. */
    if (status = OCIBindByPos(stmthp1, &bnd6p, errhp, 1,
        (dvoid *) &deptno, (sword) sizeof(deptno), SQLT_INT,
        (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT))
    {
        checkerr(errhp, status);
        cleanup();
        return OCI_ERROR;
    }

    /* Allocate the dept buffer now that you have length. */
    /* the deptlen should eventually get from dschndl3. */
    deptlen = 14;
    dept = (text *) malloc((size_t) deptlen + 1);

    /* Define the output variable for the select-list. */
    if (status = OCIDefineByPos(stmthp1, &defnp, errhp, 1,
        (dvoid *) dept, deptlen+1, SQLT_STR,
        (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, OCI_DEFAULT))
    {
        checkerr(errhp, status);
        cleanup();
        return OCI_ERROR;
    }

    for (;;)
    {

```



```

/* Prompt for employee name. Break on no name. */
printf("\nEnter employee name (or CR to EXIT): ");
fgets((char *) ename, (int) enamelen+1, stdin);
cp = (text *) strchr((char *) ename, '\n');
if (cp == ename)
{
    printf("Exiting... ");
    cleanup();
    return OCI_SUCCESS;
}
if (cp)
    *cp = '\0';
else
{
    printf("Employee name may be truncated.\n");
    myfflush();
}
/* Prompt for the employee's job and salary. */
printf("Enter employee job: ");
job_ind = 0;
fgets((char *) job, (int) joblen + 1, stdin);
cp = (text *) strchr((char *) job, '\n');
if (cp == job)
{
    job_ind = -1;          /* make it NULL in table */
    printf("Job is NULL.\n"); /* using indicator variable */
}
else if (cp == 0)
{
    printf("Job description may be truncated.\n");
    myfflush();
}
else
    *cp = '\0';

printf("Enter employee salary: ");
scanf("%d", &sal);
myfflush();
sal_ind = (sal <= 0) ? -2 : 0; /* set indicator variable */

/*
 * Prompt for the employee's department number, and verify
 * that the entered department number is valid
 * by executing and fetching.
 */

```

```

do
{
    printf("Enter employee dept: ");
    scanf("%d", &deptno);
    myfflush();
    if ((status = OCISmtExecute(svchp, stmthp1, errhp, (ub4) 1, (ub4) 0,
        (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT))
        && (status != OCI_NO_DATA))
    {
        checkerr(errhp, status);
        cleanup();
        return OCI_ERROR;
    }
    if (status == OCI_NO_DATA)
        printf("The dept you entered doesn't exist.\n");
} while (status == OCI_NO_DATA);

/*
 * Increment empno by 10, and execute the INSERT
 * statement. If the return code is 1 (duplicate
 * value in index), then generate the next
 * employee number.
 */
empno += 10;
if ((status = OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
    (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT))
    && status != 1)
{
    checkerr(errhp, status);
    cleanup();
    return OCI_ERROR;
}
while (status == 1)
{
    empno += 10;
    if ((status = OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (CONST OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT))
        && status != 1)
    {
        checkerr(errhp, status);
        cleanup();
        return OCI_ERROR;
    }
} /* end for (;;) */

```

```

        /* Commit the change. */
        if (status = OCITransCommit(svchp, errhp, 0))
        {
            checkerr(errhp, status);
            cleanup();
            return OCI_ERROR;
        }
        printf("\n\n%s added to the %s department as employee number %d\n",
            ename, dept, empno);
    }
}

```

```

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    sb4 errcode = 0;

    switch (status)
    {
        case OCI_SUCCESS:
            break;
        case OCI_SUCCESS_WITH_INFO:
            (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
            break;
        case OCI_NEED_DATA:
            (void) printf("Error - OCI_NEED_DATA\n");
            break;
        case OCI_NO_DATA:
            (void) printf("Error - OCI_NODATA\n");
            break;
        case OCI_ERROR:
            (void) OCIErrGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
            (void) printf("Error - %.*s\n", 512, errbuf);
            break;
        case OCI_INVALID_HANDLE:
            (void) printf("Error - OCI_INVALID_HANDLE\n");
            break;
        case OCI_STILL_EXECUTING:
            (void) printf("Error - OCI_STILL_EXECUTE\n");
            break;
        case OCI_CONTINUE:

```

```

        (void) printf("Error - OCI_CONTINUE\n");
        break;
    default:
        break;
    }
}

/*
 * Exit program with an exit code.
 */
void cleanup()
{
    if (stmthp)
        checkerr(errhp, OCIHandleFree((dvoid *) stmthp, OCI_HTYPE_STMT));
    if (stmthp1)
        checkerr(errhp, OCIHandleFree((dvoid *) stmthp1, OCI_HTYPE_STMT));

    if (errhp)
        (void) OCIServerDetach( srvhp, errhp, OCI_DEFAULT );
    if (srvhp)
        checkerr(errhp, OCIHandleFree((dvoid *) srvhp, OCI_HTYPE_SERVER));
    if (svchp)
        (void) OCIHandleFree((dvoid *) svchp, OCI_HTYPE_SVCCTX);
    if (errhp)
        (void) OCIHandleFree((dvoid *) errhp, OCI_HTYPE_ERROR);
    return;
}

void myfflush()
{
    ebl buf[50];

    fgets((char *) buf, 50, stdin);
}

```

Example 2, Object Retrieval

```

/* NAME
   cdemo82.c - oci object sample program ; run cdemo82.sql */

#ifdef CDEMO82_ORACLE
#include <cdemo82.h>
#endif

#define SCHEMA "CDEMO82"

/*****
static void pin_display_addr(envhp, errhp, addrref)
OCIEnv *envhp;
OCIError *errhp;
OCIRef *addrref;
{
    sword status;
    address *addr = (address *)0;

    checkerr(errhp, OCIObjectPin(envhp, errhp, addrref, (OCIComplexObject *)0,
                                OCI_PIN_ANY, OCI_DURATION_SESSION, OCI_LOCK_NONE,
                                (dvoid **)&addr));

    if (addr)
    {
        printf("address.state = %.2s address.zip = %.10s\n",
               OCIStrPtr(envhp, addr->state), OCIStrPtr(envhp, addr->zip));
    }
    else
    {
        printf("Pinned address pointer is null\n");
    }

    checkerr(errhp, OCIObjectUnpin(envhp, errhp, (dvoid *) addr));
}

*****/
static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    ub4 buflen;

```

```

ub4 errcode;

switch (status)
{
case OCI_SUCCESS:
    break;
case OCI_SUCCESS_WITH_INFO:
    printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
case OCI_NEED_DATA:
    printf("Error - OCI_NEED_DATA\n");
    break;
case OCI_NO_DATA:
    printf("Error - OCI_NO_DATA\n");
    break;
case OCI_ERROR:
    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                 errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error - %s\n", errbuf);
    break;
case OCI_INVALID_HANDLE:
    printf("Error - OCI_INVALID_HANDLE\n");
    break;
case OCI_STILL_EXECUTING:
    printf("Error - OCI_STILL_EXECUTE\n");
    break;
case OCI_CONTINUE:
    printf("Error - OCI_CONTINUE\n");
    break;
default:
    break;
}
}

/*****
/*
** execute "selvalstmt" statement -- selects from a table with an object.
**
**
*/
static void selectval(envhp, svchp, stmthp, errhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIError *errhp;
{

```

```

OCIType *addr_tdo = (OCIType *) 0;
OCIDefine *defn1p = (OCIDefine *) 0, *defn2p = (OCIDefine *) 0;
address *addr = (address *)NULL;
sword custno =0;
int i = 0;
OCIRef *addrref = (OCIRef *) 0;
OCIRef *type_ref = (OCIRef *) 0;
sb4 status;
OCIDescribe *dschp = (OCIDescribe *) 0;
OCIParam *pamp;

/* allocate describe handle for OCIDescribeAny */
checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **) &dschp,
                               (ub4) OCI_HTYPE_DESCRIBE,
                               (size_t) 0, (dvoid **) 0));

/* define the application request */
checkerr(errhp, OCISstmtPrepare(stmthp, errhp, (text *) selvalstmt,
                               (ub4) strlen(selvalstmt),
                               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

/* bind the input variable */
checkerr(errhp, OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *)
                               &custno,
                               (sb4) sizeof(sword), SQLT_INT, (dvoid *) 0, (ub2 *)0,
                               (ub2 *)0, (ub4) OCI_DEFAULT));

checkerr(errhp, OCIDefineByPos(stmthp, &defn2p, errhp, (ub4) 2, (dvoid *) 0,
                               (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                               (ub2 *)0, (ub4) OCI_DEFAULT));

/* checkerr(errhp, OCITypeByName(envhp, errhp, svchp, (const text *) 0,
                               (ub4) 0, (const text *) "ADDRESS_VALUE",
                               (ub4) strlen((const char *) "ADDRESS_VALUE"),
                               (CONST text *) 0, (ub4) 0,
                               OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                               &addr_tdo)); */

checkerr(errhp, OCIDescribeAny(svchp, errhp, (text *) "ADDRESS_VALUE",
                               (ub4) strlen((char *) "ADDRESS_VALUE"), OCI_OTYPE_NAME,
                               (ub1)1,
                               (ub1) OCI_PTYPE_TYPE, dschp));

checkerr(errhp, OCIAttrGet((dvoid *) dschp, (ub4) OCI_HTYPE_DESCRIBE,

```

```

        (dvoid *)&pamp, (ub4 *)0, (ub4)OCI_ATTR_PARAM, errhp));

checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &type_ref, (ub4 *) 0,
        (ub4) OCI_ATTR_REF_TDO, (OCIError *) errhp));

checkerr(errhp, OCIObjectPin(envhp, errhp, type_ref, (OCIComplexObject *) 0,
        OCI_PIN_ANY, OCI_DURATION_SESSION, OCI_LOCK_NONE,
        (dvoid **)&addr_tdo));

if(!addr_tdo)
{
    printf("NULL tdo returned\n");
    goto done_selectval;
}

checkerr(errhp, OCIDefineObject(defn2p, errhp, addr_tdo, (dvoid **) &addr,
        (ub4 *) 0, (dvoid **) 0, (ub4 *) 0));

checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (OCISnapshot *) NULL, (OCISnapshot *) NULL, (ub4)
        OCI_DEFAULT));

/* execute and fetch */
do
{
    if (addr)
        printf("custno = %d address.state = %.2s address.zip = %.10s\n", custno,
            OCIStrPtr(envhp, addr->state), OCIStrPtr(envhp, addr->zip));
    else
        printf("custno = %d fetched address is NULL\n", custno);

    addr = (address *)NULL;
}
while ((status = OCISmtFetch(stmthp, errhp, (ub4) 1, (ub4) OCI_FETCH_NEXT,
        (ub4) OCI_DEFAULT)) == OCI_SUCCESS ||
        status == OCI_SUCCESS_WITH_INFO);

if ( status!= OCI_NO_DATA )
    checkerr(errhp, status);

printf("\n\n");

```



```

done_selectval:

    checkerr(errhp, OCIHandleFree((dvoid *) defn1p, (ub4) OCI_HTYPE_DEFINE));
    checkerr(errhp, OCIHandleFree((dvoid *) defn2p, (ub4) OCI_HTYPE_DEFINE));

}

/*****
** execute "selobjstmt" -- selects records from a table with a REF.
**/
static void selectobj(envhp, svchp, stmthp, errhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIError *errhp;
{
    OCIType *addr_tdo = (OCIType *) 0;
    OCIDefine *defn1p = (OCIDefine *) 0, *defn2p = (OCIDefine *) 0;
    sword status;
    OCIRef *addrref = (OCIRef *) 0, *addrref1 = (OCIRef *) 0;
    sword custno = 0;
    int i = 0;
    address *addr;
    ub4 ref_len;

    /* define the application request */
    checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) selobjstmt,
                                   (ub4) strlen(selobjstmt),
                                   (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *)
                                   &custno, (sb4) sizeof(sword), SQLT_INT, (dvoid *) 0, (ub2 *) 0,
                                   (ub2 *) 0, (ub4) OCI_DEFAULT));

    addrref = (OCIRef *) NULL;

    checkerr(errhp, OCIDefineByPos(stmthp, &defn2p, errhp, (ub4) 2, (dvoid *)
                                   NULL, (sb4) 0, SQLT_REF, (dvoid *) 0, (ub2 *) 0,
                                   (ub2 *) 0, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCIDefineObject(defn2p, errhp, (OCIType *) NULL,
                                   (dvoid **)&addrref, &ref_len, (dvoid **) 0, (ub4 *) 0));

    checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,

```

```

        (OCISnapshot *) NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

do
{
    printf("custno = %d fetched address\n", custno);

    if ( addrref )
    {
        pin_display_addr(envhp, errhp, addrref);
    }
    else
        printf("Address ref is NULL\n");

}
while ((status = OCISmtFetch(stmthp, errhp, (ub4) 1, (ub4) OCI_FETCH_NEXT,
                             (ub4) OCI_DEFAULT)) == OCI_SUCCESS ||
        status == OCI_SUCCESS_WITH_INFO);

if ( status != OCI_NO_DATA )
    checkerr(errhp, status);

printf("\n\n");
checkerr(errhp, OCIHandleFree((dvoid *) defn1p, (ub4) OCI_HTYPE_DEFINE));
checkerr(errhp, OCIHandleFree((dvoid *) defn2p, (ub4) OCI_HTYPE_DEFINE));

}

/*****
/*****
/*
** execute "insstmt"
**
*/
static void insert(envhp, svchp, stmthp, errhp, insstmt, nrows)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCISmt *stmthp;
OCIError *errhp;
text *insstmt;
ub2 nrows;
{
    OCIType *addr_tdo = (OCIType *) 0;
    address addr;

```

```

null_address naddrs;
address *addr = &naddrs;
null_address *naddr = &naddrs;
sword custno =300;
OCIBind *bnd1p = (OCIBind *) 0, *bnd2p = (OCIBind *) 0;
char buf[20];
ub2 i;
OCISRef *type_ref = (OCISRef *) 0;
OCIDescribe *dschp = (OCIDescribe *) 0;
OCIParam *pamp;

/* allocate describe handle for OCIDescribeAny */
checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **) &dschp,
                               (ub4) OCI_HTYPE_DESCRIBE,
                               (size_t) 0, (dvoid **) 0));

/* define the application request */
checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) insstmt,
                               (ub4) strlen(insstmt),
                               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

/* bind the input variable */
checkerr(errhp, OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":custno",
                              (sb4) -1, (dvoid *) &custno,
                              (sb4) sizeof(sword), SQLT_INT,
                              (dvoid *) 0, (ub2 *)0, (ub2 *)0, (ub4) 0, (ub4 *) 0,
                              (ub4) OCI_DEFAULT));

checkerr(errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":addr",
                              (sb4) -1, (dvoid *) 0,
                              (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                              (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT));

/* checkerr(errhp, OCISRefByName(envhp, errhp, svchp, (const text *) 0,
                               (ub4) 0, (const text *) "ADDRESS_VALUE",
                               (ub4) strlen((const char *) "ADDRESS_VALUE"),
                               (CONST text *) 0, (ub4) 0,
                               OCI_DURATION_SESSION, OCI_TYPEGET_HEADER,
                               &addr_tdo)); */

checkerr(errhp, OCIDescribeAny(svchp, errhp, (text *) "ADDRESS_VALUE",
                               (ub4) strlen((char *) "ADDRESS_VALUE"), OCI_OTYPE_NAME,
                               (ub1)1, (ub1) OCI_PTYPE_TYPE, dschp));

checkerr(errhp, OCIAttrGet((dvoid *) dschp, (ub4) OCI_HTYPE_DESCRIBE,

```

```

        (dvoid *)&pamp, (ub4 *)0, (ub4)OCI_ATTR_PARAM, errhp));

checkerr(errhp, OCIAAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &type_ref, (ub4 *) 0,
        (ub4) OCI_ATTR_REF_TDO, (OCIError *) errhp));

checkerr(errhp, OCIObjectPin(envhp, errhp, type_ref, (OCIComplexObject *) 0,
        OCI_PIN_ANY, OCI_DURATION_SESSION, OCI_LOCK_NONE,
        (dvoid **)&addr_tdo));

if(!addr_tdo)
{
    printf("Null tdo returned\n");
    goto done_insert;
}

checkerr(errhp, OCIBindObject(bnd2p, errhp, addr_tdo, (dvoid **) &addr,
        (ub4 *) 0, (dvoid **) &naddr, (ub4 *) 0));

for(i = 0; i <= nrows; i++)
{
    addr->state = (OCIStrng *) 0;
    sprintf(buf, "%cA", 65+i%27);
    checkerr(errhp, OCIStrngAssignText(envhp, errhp, (CONST text*) buf,
        2, &addr->state));
    addr->zip = (OCIStrng *) 0;
    sprintf(buf, "94%d ", i+455);
    checkerr(errhp, OCIStrngAssignText(envhp, errhp, (CONST text*) buf, 10,
        &addr->zip));

    naddr->null_object = 0;
    naddr->null_state = 0;
    naddr->null_zip = 0;

    checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (OCISnapshot *) NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));
}
checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) 0));

done_insert:

checkerr(errhp, OCIHandleFree((dvoid *) bnd1p, (ub4) OCI_HTYPE_BIND));
checkerr(errhp, OCIHandleFree((dvoid *) bnd2p, (ub4) OCI_HTYPE_BIND));

}

```

```

/*****/
int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCISstmt *stmthp;
    OCISession *usrhp;

    OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT, (dvoid *)0, (dvoid * (*)())
                  0, (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    52, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                    52, (dvoid **) &tmp);

    OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                    52, (dvoid **) &tmp);

    /* set attribute server context in the service context */
    OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                (dvoid *) srvhp, (ub4) 0,
                (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

    /* allocate a user context handle */
    OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                   (size_t) 0, (dvoid **) 0);

    OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
                (dvoid *)"cdemo82", (ub4)strlen("cdemo82"),
                OCI_ATTR_USERNAME, errhp);

    OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
                (dvoid *)"cdemo82", (ub4)strlen("cdemo82"),
                OCI_ATTR_PASSWORD, errhp);
}

```

```
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
                                OCI_DEFAULT));

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
           (dvoid *)usrhp, (ub4)0,
           OCI_ATTR_SESSION, errhp);

checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                                (ub4) OCI_HTYPE_STMT, 50, (dvoid **) &tmp));

/* execute "insstmt" */
printf("--- Test insertion into extent table.\n");
insert(envhp, svchp, stmthp, errhp, insstmt, 26);

/* execute "selstmt" */
printf("--- Test selection of a table with one object column.\n");
selectval(envhp, svchp, stmthp, errhp);

/* execute "selobjstmt" */
printf("--- Test selection of a table with one object REF.\n");
selectobj(envhp, svchp, stmthp, errhp);

checkerr(errhp, OCIHandleFree((dvoid *) stmthp, (ub4) OCI_HTYPE_STMT));

OCISessionEnd(svchp, errhp, usrhp, (ub4)OCI_DEFAULT);
OCIServerDetach( srvhp, errhp, (ub4) OCI_DEFAULT );
checkerr(errhp, OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER));
checkerr(errhp, OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX));
checkerr(errhp, OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR));

}
```

cdemo82.h

```
/*
    NAME
        cdemo82.h - header file for oci object sample program
*/

#ifndef CDEMO82_ORACLE
#define CDEMO82_ORACLE
```

```

#ifndef OCI_ORACLE
#include <oci.h>
#endif

/*-----
PRIVATE TYPES AND CONSTANTS
-----*/
#define SERVER "ORACLE"
#define ADDRESS_TYPE_NAME "ADDRESS_OBJECT"
#define EMB_ADDRESS_TYPE_NAME "EMBEDDED_ADDRESS"
#define ADDREXT "ADDREXT"
#define EMBADDREXT "EMBADDREXT"
#define RETURN_ON_ERROR(error) if (error) return (error)
#define BIG_RECORD_SIZE 1000

struct address
{
    OCIStr *state;
    OCIStr *zip;
};
typedef struct address address;

struct null_address
{
    sb4 null_object;
    sb4 null_state;
    sb4 null_zip;
};
typedef struct null_address null_address;

struct embaddress
{
    OCIStr *state;
    OCIStr *zip;
    OCIStr *preaddrref;
};
typedef struct embaddress embaddress;

struct null_embaddress
{
    sb4 null_state;
    sb4 null_zip;
    sb4 null_preaddrref;
};

```

```

typedef struct null_embaddress null_embaddress;

struct person
{
    OCISString      *name;
    OCINumber       age;
    address         addr;
};
typedef struct person person;

struct null_person
{
    sb4             null_name;
    sb4             null_age;
    null_address    null_addr;
};

typedef struct null_person null_person;

static const text *const names[] =
{ (text *) "CUSTOMERVAL", (text *) "ADDRESS", (text *) "STATE" };

static const text *const selvalstmt = (text *)
    "SELECT custno, addr FROM customerval";

static const text *const selobjstmt = (text *)
    "SELECT custno, addr FROM customerobj";

static const text *const selref = (text *)
    "SELECT REF(extaddr) from extaddr";

static const text *const deleteref = (text *)
    "DELETE extaddr";

static const text *const insertref = (text *)
    "insert into extaddr values(address_object('CA', '98765'))";

static const text *const modifyref = (text *)
    "update extaddr set object_column = address_object('TX', '61111')";

static const text *const selembref = (text *)
    "SELECT REF(exbextaddr) from embextaddr";

static const text *const bndref = (text *)
    "update extaddr set object_column.state = 'GA' where object_column = :addrref";

```



```

static const text *const insstmt =
(text *)"INSERT INTO customerval (custno, addr) values (:custno, :addr)";

dvoid *tmp;

/*-----
                                PUBLIC FUNCTIONS
-----*/
OCIRef *cbfunc(/*_ dvoid *context _*/);

/*-----
                                PRIVATE FUNCTIONS
-----*/
static void checkerr(/*_ OCIError *errhp, sword status _*/);
static void selectval(/*_ OCIEnv *envhp, OCISvcCtx *svchp,
                      OCISmt *stmthp, OCIError *errhp _*/);
static void selectobj(/*_ OCIEnv *envhp, OCISvcCtx *svchp,
                     OCISmt *stmthp, OCIError *errhp _*/);
static void insert(/*_ OCIEnv *envhp, OCISvcCtx *svchp,
                  OCISmt *stmthp, OCIError *errhp,
                  text *insstmt, ub2 nrows _*/);

static void pin_display_addr(/*_ OCIEnv *envhp, OCIError *errhp,
                             OCIRef *addrref _*/);

int main(/*_ void _*/);

```

cdemo82.sql

```

Rem cdemo82.sql
Rem
Rem      NAME
Rem      cdemo82.sql - sql to be executed before cdemo82
Rem

set echo on;
connect internal;
drop user cdemo82 cascade;
create user cdemo82 identified by cdemo82;
grant connect, resource to cdemo82;
connect cdemo82/cdemo82;
drop table customerval;
drop table customerobj;
drop table extaddr;

```

```
drop table embextaddr;
drop type embedded_address;
drop type address_object;
drop type person;
drop table emp;
create type address_object as object (state char(2), zip char(10));
create type embedded_address as object (state char(2), zip char(10),
    preaddr REF address_object);
drop type address_value;
create type address_value as object (state char(2), zip char(10));
create table customerval (custno number, addr address_value);
insert into customerval values(100, address_value('CA', '94065'));
create table extaddr of address_object;
create table customerobj (custno number, addr REF address_object);
insert into extaddr values (address_object('CA', '94065'));
insert into customerobj values(1000, null);
update customerobj set addr = (select ref(extaddr) from extaddr where
    zip='94065');
insert into extaddr values (address_object('CA', '98765'));
insert into extaddr values (address_object('CA', '95117'));
select REFTOHEX(ref(extaddr)) from extaddr;
create table embextaddr of embedded_address;
insert into embextaddr values (embedded_address('CA', '95117', NULL));
select objectTOHEX(p) from embextaddr p;
drop table extper;
drop table empref;
drop table emp;
drop type person;
create type person as object ( name char(20), age number, address
    address_object );
create table emp (emp_id number, emp_info person);
create table empref (emp_id number, emp_info REF person);
create table extper of person;
create or replace procedure upd_addr(addr IN OUT address_object) is
begin
    addr.state := 'CA';
    addr.zip := '95117';
end;
/
commit;
set echo off;
```

Example 3, DML with RETURNING Clause

```

/*  NAME
    cdemord1.c - C DEMO program for DML with RETURNING clause - #1.

    DESCRIPTION
    This Demo program demonstrates the use of  INSERT/UPDATE/DELETE
    statements with a RETURNING clause in it.
*/

#include <cdemodr1.h>

/*----- Global Variables -----*/

static boolean logged_on = FALSE;

/* TAB1 columns */
static int   in1[MAXITER];          /* for INTEGER      */
static text  in2[MAXITER][40];      /* for CHAR(40)     */
static text  in3[MAXITER][40];      /* for VARCHAR2(40) */
static float in4[MAXITER];          /* for FLOAT        */
static int   in5[MAXITER];          /* for DECIMAL      */
static float in6[MAXITER];          /* for DECIMAL(8,3) */
static int   in7[MAXITER];          /* for NUMERIC      */
static float in8[MAXITER];          /* for NUMERIC(7,2) */
static ub1   in9[MAXITER][7];       /* for DATE         */
static ub1   in10[MAXITER][40];     /* for RAW(40)      */

/* output buffers */
static int   *p1[MAXITER];          /* for INTEGER      */
static text  *p2[MAXITER];          /* for CHAR(40)     */
static text  *p3[MAXITER];          /* for VARCHAR2(40) */
static float *p4[MAXITER];          /* for FLOAT        */
static int   *p5[MAXITER];          /* for DECIMAL      */
static float *p6[MAXITER];          /* for DECIMAL(8,3) */
static int   *p7[MAXITER];          /* for NUMERIC      */
static float *p8[MAXITER];          /* for NUMERIC(7,2) */
static ub1   *p9[MAXITER];          /* for DATE         */
static ub1   *p10[MAXITER];         /* for RAW(40)      */

static short *ind[MAXCOLS][MAXITER]; /* indicators */
static ub2   *rc[MAXCOLS][MAXITER];  /* return codes */
static ub4   *rl[MAXCOLS][MAXITER];  /* return lengths */

```

```

/* skip values for binding TAB1 */
static ub4   s1 = (ub4) sizeof(in1[0]);
static ub4   s2 = (ub4) sizeof(in2[0]);
static ub4   s3 = (ub4) sizeof(in3[0]);
static ub4   s4 = (ub4) sizeof(in4[0]);
static ub4   s5 = (ub4) sizeof(in5[0]);
static ub4   s6 = (ub4) sizeof(in6[0]);
static ub4   s7 = (ub4) sizeof(in7[0]);
static ub4   s8 = (ub4) sizeof(in8[0]);
static ub4   s9 = (ub4) sizeof(in9[0]);
static ub4   s10= (ub4) sizeof(in10[0]);

/* Rows returned in each iteration */
static ub2 rowsret[MAXITER];

/* indicator skips */
static ub4   indsk[MAXCOLS] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
/* return length skips */
static ub4   rlsk[MAXCOLS] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
/* return code skips */
static ub4   rcsk[MAXCOLS] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

static int   lowc1[MAXITER], highc1[MAXITER];

static ub4   pos[MAXCOLS];

static OCLError *errhp;

/*-----end of Global variables-----*/

/*===== UTILITY FUNCTIONS =====*/
/*
 * These functions are generic functions that can be used in any
 * OCI program.
 */

/* ----- */
/* Initialize environment, allocate handles */
/* ----- */
sword init_handles(envhp, svchp, errhp, srvhp, authp, init_mode)
OCIEnv **envhp;
OCISvcCtx **svchp;
OCLError **errhp;
OCIError **srvhp;
OCIError **authp;

```

```

ub4 init_mode;
{
    (void) printf("Environment setup ....\n");

    /* Initialize the OCI Process */
    if (OCIInitialize(init_mode, (dvoid *)0,
                     (dvoid * (*)(dvoid *, size_t)) 0,
                     (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                     (void (*)(dvoid *, dvoid *)) 0 ))
    {
        (void) printf("FAILED: OCIInitialize()\n");
        return OCI_ERROR;
    }

    /* Inititalize the OCI Environment */
    if (OCIEnvInit((OCIEnv **) envhp, (ub4) OCI_DEFAULT,
                  (size_t) 0, (dvoid **) 0 ))
    {
        (void) printf("FAILED: OCIEnvInit()\n");
        return OCI_ERROR;
    }

    /* Allocate a service handle */
    if (OCIHandleAlloc((dvoid *) *envhp, (dvoid **) svchp,
                      (ub4) OCI_HTYPE_SVCCTX, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc() on svchp\n");
        return OCI_ERROR;
    }

    /* Allocate an error handle */
    if (OCIHandleAlloc((dvoid *) *envhp, (dvoid **) errhp,
                      (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc() on errhp\n");
        return OCI_ERROR;
    }

    /* Allocate a server handle */
    if (OCIHandleAlloc((dvoid *) *envhp, (dvoid **) srvhp,
                      (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc() on srvhp\n");
        return OCI_ERROR;
    }
}

```

```

/* Allocate a authentication handle */
if (OCIHandleAlloc((dvoid *) *envhp, (dvoid **) authp,
                  (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
{
    (void) printf("FAILED: OCIHandleAlloc() on authp\n");
    return OCI_ERROR;
}

return OCI_SUCCESS;
}

/* ----- */
/* Attach to server with a given mode. */
/* ----- */
sword attach_server(mode, srvhp, errhp, svchp)
ub4 mode;
OCIServer *srvhp;
OCIError *errhp;
OCISvcCtx *svchp;
{
    text *cstring = (text *)"";

    if (OCIServerAttach(srvhp, errhp, (text *) cstring,
                      (sb4) strlen((char *)cstring), (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIServerAttach()\n");
        return OCI_ERROR;
    }

    /* Set the server handle in the service handle */
    if (OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                  (dvoid *) srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER, errhp))
    {
        (void) printf("FAILED: OCIAttrSet() server attribute\n");
        return OCI_ERROR;
    }

    return OCI_SUCCESS;
}

/* ----- */
/* Logon to the database using given username, password & credentials*/
/* ----- */
sword log_on(authp, errhp, svchp, uid, pwd, credt, mode)
OCISession *authp;

```

```

OCIError *errhp;
OCISvcCtx *svchp;
text *uid;
text *pwd;
ub4 credt;
ub4 mode;
{
    /* Set attributes in the authentication handle */
    if (OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                  (dvoid *) uid, (ub4) strlen((char *) uid),
                  (ub4) OCI_ATTR_USERNAME, errhp))
    {
        (void) printf("FAILED: OCIAttrSet() userid\n");
        return OCI_ERROR;
    }
    if (OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                  (dvoid *) pwd, (ub4) strlen((char *) pwd),
                  (ub4) OCI_ATTR_PASSWORD, errhp))
    {
        (void) printf("FAILED: OCIAttrSet() passwd\n");
        return OCI_ERROR;
    }

    (void) printf("Logging on as %s ....\n", uid);

    if (OCISessionBegin(svchp, errhp, authp, credt, mode))
    {
        (void) printf("FAILED: OCIAttrSet() passwd\n");
        return OCI_ERROR;
    }

    (void) printf("%s logged on.\n", uid);

    /* Set the authentication handle in the Service handle */
    if (OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                  (dvoid *) authp, (ub4) 0, (ub4) OCI_ATTR_SESSION, errhp))
    {
        (void) printf("FAILED: OCIAttrSet() session\n");
        return OCI_ERROR;
    }

    return OCI_SUCCESS;
}

/*-----*/

```

```

/* Allocate all required bind handles                                     */
/*-----*/

sword init_bind_handle(stmthp, bndhp, nbinds)
OCIStmt *stmthp;
OCIBind *bndhp[];
int nbinds;
{
    int i;
    /*
     * This function init the specified number of bind handles
     * from the given statement handle.
     */
    for (i = 0; i < nbinds; i++)
        bndhp[i] = (OCIBind *) 0;

    return OCI_SUCCESS;
}

/* ----- */
/* Print the returned raw data.                                         */
/* ----- */
void print_raw(raw, rawlen)
ub1 *raw;
ub4 rawlen;
{
    ub4 i;
    ub4 lim;
    ub4 clen = 0;

    if (rawlen > 120)
    {
        ub4 llen = rawlen;

        while (llen > 120)
        {
            lim = clen + 120;
            for(i = clen; i < lim; ++i)
                (void) printf("%02.2x", (ub4) raw[i] & 0xFF);

            (void) printf("\n");
            llen -= 120;
            clen += 120;
        }
        lim = clen + llen;
    }
}

```



```

    }
    else
        lim = rawlen;

    for(i = clen; i < lim; ++i)
        (void) printf("%02.2x", (ub4) raw[i] & 0xFF);

    (void) printf("\n");

    return;
}

/* ----- */
/* Free the specified handles */
/* ----- */
void free_handles(envhp, svchp, srvhp, errhp, authp, stmthp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIError *errhp;
OCIError *errhp;
OCISession *authp;
OCIStmt *stmthp;
{
    (void) printf("Freeing handles ...\n");

    if (srvhp)
        (void) OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER);
    if (svchp)
        (void) OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX);
    if (errhp)
        (void) OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR);
    if (authp)
        (void) OCIHandleFree((dvoid *) authp, (ub4) OCI_HTYPE_SESSION);
    if (stmthp)
        (void) OCIHandleFree((dvoid *) stmthp, (ub4) OCI_HTYPE_STMT);
    if (envhp)
        (void) OCIHandleFree((dvoid *) envhp, (ub4) OCI_HTYPE_ENV);

    return;
}

/* ----- */
/* Print the error message */
/* ----- */
void report_error(errhp)

```

```

OCIError *errhp;
{
    text msgbuf[512];
    sb4 errcode = 0;

    (void) OCIErrorGet((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                      msgbuf, (ub4) sizeof(msgbuf), (ub4) OCI_HTYPE_ERROR);
    (void) printf("ERROR CODE = %d\n", errcode);
    (void) printf("%.5s\n", 512, msgbuf);
    return;
}

/*-----*/
/* Logout and detach from the server */
/*-----*/
void logout_detach_server(svchp, srvhp, errhp, authp, userid)
OCISvcCtx *svchp;
OCIServer *srvhp;
OCIError *errhp;
OCISession *authp;
text *userid;
{
    if (OCISessionEnd(svchp, errhp, authp, (ub4) 0))
    {
        (void) printf("FAILED: OCISessionEnd()\n");
        report_error(errhp);
    }

    (void) printf("%s Logged off.\n", userid);

    if (OCIServerDetach(srvhp, errhp, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISessionEnd()\n");
        report_error(errhp);
    }

    (void) printf("Detached from server.\n");

    return;
}

/*-----*/
/* Finish demo and clean up */
/*-----*/
sword finish_demo(loggedon, envhp, svchp, srvhp, errhp, authp, stmthp, userid)

```

```

boolean loggedon;
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIServer *srvhp;
OCIError *errhp;
OCISession *authp;
OCIStmt *stmthp;
text *userid;
{

    if (loggedon)
        logout_detach_server(svchp, srvhp, errhp, authp, userid);

    free_handles(envhp, svchp, srvhp, errhp, authp, stmthp);

    return OCI_SUCCESS;
}

/*===== END OF UTILITY FUNCTIONS =====*/

/*===== MAIN =====*/
int main(argc, argv)
int argc;
char *argv[];
{
    text *username = (text *)"scott";
    text *password = (text *)"tiger";

    OCIEnv *envhp;
    OCIServer *srvhp;
    OCISvcCtx *svchp;
    OCISession *authp;
    OCIStmt *stmthp;
    OCIBind *bndhp[MAXBINDS];
    int i;

    /* Initialize the Environment and allocate handles */
    if (init_handles(&envhp, &svchp, &errhp, &srvhp, &authp, (ub4)OCI_DEFAULT))
    {
        (void) printf("FAILED: init_handles()\n");
        return finish_demo(logged_on, envhp, svchp, srvhp, errhp, authp,
                           stmthp, username);
    }
}

```

```

/* Attach to the database server */
if (attach_server((ub4) OCI_DEFAULT, srvhp, errhp, svchp))
{
    (void) printf("FAILED: attach_server()\n");
    return finish_demo(logged_on, envhp, svchp, srvhp, errhp, authp,
                      stmthp, username);
}

/* Logon to the server and begin a session */
if (log_on(authp, errhp, svchp, username, password,
          (ub4) OCI_CRED_RDBMS, (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: log_on()\n");
    return finish_demo(logged_on, envhp, svchp, srvhp, errhp, authp,
                      stmthp, username);
}
logged_on = TRUE;

/* Allocate a statement handle */
if (OCIHandleAlloc((dvoid *)envhp, (dvoid **) &stmthp,
                  (ub4)OCI_HTYPE_STMT, (CONST size_t) 0, (dvoid **) 0))
{
    (void) printf("FAILED: alloc statement handle\n");
    return finish_demo(logged_on, envhp, svchp, srvhp, errhp, authp,
                      stmthp, username);
}

/* bind handles will be implicitly allocated in the bind calls */
/* need to initialize them to null prior to first usage in bind calls */

for (i = 0; i < MAXBINDS; i++)
    bndhp[i] = (OCIBind *) 0;

/* Demonstrate INSERT with RETURNING clause */
if (demo_insert(svchp, stmthp, bndhp, errhp))
    (void) printf("FAILED: demo_insert()\n");
else
    (void) printf("SUCCESS: demo_insert()\n");

/* Demonstrate UPDATE with RETURNING clause */
if (demo_update(svchp, stmthp, bndhp, errhp))
    (void) printf("FAILED: demo_update()\n");
else
    (void) printf("SUCCESS: demo_update()\n");

```

```

/* Demonstrate DELETE with RETURNING clause */
if (demo_delete(svchp, stmthp, bndhp, errhp))
    (void) printf("FAILED: demo_delete()\n");
else
    (void) printf("SUCCESS: demo_delete()\n");

/* clean up */
return finish_demo(logged_on, envhp, svchp, srvhp, errhp, authp,
                  stmthp, username);
}

/* ===== End Main ===== */

/* ===== Local Functions ===== */
/* ----- */
/* bind all the columns of TAB1 by positions. */
/* ----- */
static sword bind_pos(OCIStmt *stmthp, OCIBind *bndhp[], OCIError *errhp)
{
    if (OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
                    (dvoid *) &in1[0], (sb4) sizeof(in1[0]), SQLT_INT,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[1], errhp, (ub4) 2,
                    (dvoid *) in2[0], (sb4) sizeof(in2[0]), SQLT_AFC,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[2], errhp, (ub4) 3,
                    (dvoid *) in3[0], (sb4) sizeof(in3[0]), SQLT_CHR,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[3], errhp, (ub4) 4,
                    (dvoid *) &in4[0], (sb4) sizeof(in4[0]), SQLT_FLT,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[4], errhp, (ub4) 5,
                    (dvoid *) &in5[0], (sb4) sizeof(in5[0]), SQLT_INT,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[5], errhp, (ub4) 6,
                    (dvoid *) &in6[0], (sb4) sizeof(in6[0]), SQLT_FLT,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[6], errhp, (ub4) 7,

```

```

        (dvoid *) &in7[0], (sb4) sizeof(in7[0]), SQLT_INT,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[7], errhp, (ub4) 8,
        (dvoid *) &in8[0], (sb4) sizeof(in8[0]), SQLT_FLT,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[8], errhp, (ub4) 9,
        (dvoid *) in9[0], (sb4) sizeof(in9[0]), SQLT_DAT,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByPos(stmthp, &bndhp[9], errhp, (ub4) 10,
        (dvoid *) in10[0], (sb4) sizeof(in10[0]), SQLT_BIN,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: OCIBindByPos()\n");
    report_error(errhp);
    return OCI_ERROR;
}

return OCI_SUCCESS;
}

/* ----- */
/* bind all the columns of TAB1 by name. */
/* ----- */
static sword bind_name(OCIStmt *stmthp, OCIBind *bndhp[], OCIError *errhp)
{
    if (OCIBindByName(stmthp, &bndhp[10], errhp,
        (text *) ":out1", (sb4) strlen((char *) ":out1"),
        (dvoid *) 0, (sb4) sizeof(int), SQLT_INT,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
    || OCIBindByName(stmthp, &bndhp[11], errhp,
        (text *) ":out2", (sb4) strlen((char *) ":out2"),
        (dvoid *) 0, (sb4) MAXCOLLEN, SQLT_AFC,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
    || OCIBindByName(stmthp, &bndhp[12], errhp,
        (text *) ":out3", (sb4) strlen((char *) ":out3"),
        (dvoid *) 0, (sb4) MAXCOLLEN, SQLT_CHR,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)

```

```

|| OCIBindByName(stmthp, &bndhp[13], errhp,
                (text *) ":out4", (sb4) strlen((char *) ":out4"),
                (dvoid *) 0, (sb4) sizeof(float), SQLT_FLT,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
|| OCIBindByName(stmthp, &bndhp[14], errhp,
                (text *) ":out5", (sb4) strlen((char *) ":out5"),
                (dvoid *) 0, (sb4) sizeof(int), SQLT_INT,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
|| OCIBindByName(stmthp, &bndhp[15], errhp,
                (text *) ":out6", (sb4) strlen((char *) ":out6"),
                (dvoid *) 0, (sb4) sizeof(float), SQLT_FLT,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
|| OCIBindByName(stmthp, &bndhp[16], errhp,
                (text *) ":out7", (sb4) strlen((char *) ":out7"),
                (dvoid *) 0, (sb4) sizeof(int), SQLT_INT,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
|| OCIBindByName(stmthp, &bndhp[17], errhp,
                (text *) ":out8", (sb4) strlen((char *) ":out8"),
                (dvoid *) 0, (sb4) sizeof(float), SQLT_FLT,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
|| OCIBindByName(stmthp, &bndhp[18], errhp,
                (text *) ":out9", (sb4) strlen((char *) ":out9"),
                (dvoid *) 0, (sb4) DATBUFLLEN, SQLT_DAT,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
|| OCIBindByName(stmthp, &bndhp[19], errhp,
                (text *) ":out10", (sb4) strlen((char *) ":out10"),
                (dvoid *) 0, (sb4) MAXCOLLEN, SQLT_BIN,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC))
{
    (void) printf("FAILED: OCIBindByName()\n");
    report_error(errhp);
    return OCI_ERROR;
}

return OCI_SUCCESS;
}

```

```

/* ----- */
/* bind array structs for TAB1 columns.                */
/* ----- */
static sword bind_array(OCIBind *bndhp[], OCIError *errhp)
{
    if (OCIBindArrayOfStruct(bndhp[0], errhp, s1, indsk[0], rlsk[0], rcsk[0])
        || OCIBindArrayOfStruct(bndhp[1], errhp, s2, indsk[1], rlsk[1], rcsk[1])
        || OCIBindArrayOfStruct(bndhp[2], errhp, s3, indsk[2], rlsk[2], rcsk[2])
        || OCIBindArrayOfStruct(bndhp[3], errhp, s4, indsk[3], rlsk[3], rcsk[3])
        || OCIBindArrayOfStruct(bndhp[4], errhp, s5, indsk[4], rlsk[4], rcsk[4])
        || OCIBindArrayOfStruct(bndhp[5], errhp, s6, indsk[5], rlsk[5], rcsk[5])
        || OCIBindArrayOfStruct(bndhp[6], errhp, s7, indsk[6], rlsk[6], rcsk[6])
        || OCIBindArrayOfStruct(bndhp[7], errhp, s8, indsk[7], rlsk[7], rcsk[7])
        || OCIBindArrayOfStruct(bndhp[8], errhp, s9, indsk[8], rlsk[8], rcsk[8])
        || OCIBindArrayOfStruct(bndhp[9], errhp, s10, indsk[9], rlsk[9], rcsk[9]))
    {
        (void) printf("FAILED: OCIBindArrayOfStruct()\n");
        report_error(errhp);
        return OCI_ERROR;
    }

    return OCI_SUCCESS;
}

/* ----- */
/* bind dynamic for returning TAB1 columns.                */
/* ----- */
static sword bind_dynamic(OCIBind *bndhp[], OCIError *errhp)
{
    /*
     * Note here that both IN & OUT BIND callback functions have to be
     * provided. However, since the bind variables in the RETURNING
     * clause are pure OUT Binds the IN callback fuctions (cbf_no_data)
     * is essentially a "do-nothing" function.
     *
     * Also note here that although in this demonstration the IN and OUT
     * callback functions are same, in practice you can have a different
     * callback function for each bind handle.
     */

    ub4 i;

    for (i = 0; i < MAXCOLS; i++)
        pos[i] = i;
}

```



```

if (OCIBindDynamic(bndhp[10], errhp, (dvoid *) &pos[0], cbf_no_data,
                  (dvoid *) &pos[0], cbf_get_data)
|| OCIBindDynamic(bndhp[11], errhp, (dvoid *) &pos[1], cbf_no_data,
                  (dvoid *) &pos[1], cbf_get_data)
|| OCIBindDynamic(bndhp[12], errhp, (dvoid *) &pos[2], cbf_no_data,
                  (dvoid *) &pos[2], cbf_get_data)
|| OCIBindDynamic(bndhp[13], errhp, (dvoid *) &pos[3], cbf_no_data,
                  (dvoid *) &pos[3], cbf_get_data)
|| OCIBindDynamic(bndhp[14], errhp, (dvoid *) &pos[4], cbf_no_data,
                  (dvoid *) &pos[4], cbf_get_data)
|| OCIBindDynamic(bndhp[15], errhp, (dvoid *) &pos[5], cbf_no_data,
                  (dvoid *) &pos[5], cbf_get_data)
|| OCIBindDynamic(bndhp[16], errhp, (dvoid *) &pos[6], cbf_no_data,
                  (dvoid *) &pos[6], cbf_get_data)
|| OCIBindDynamic(bndhp[17], errhp, (dvoid *) &pos[7], cbf_no_data,
                  (dvoid *) &pos[7], cbf_get_data)
|| OCIBindDynamic(bndhp[18], errhp, (dvoid *) &pos[8], cbf_no_data,
                  (dvoid *) &pos[8], cbf_get_data)
|| OCIBindDynamic(bndhp[19], errhp, (dvoid *) &pos[9], cbf_no_data,
                  (dvoid *) &pos[9], cbf_get_data))
{
    (void) printf("FAILED: OCIBindDynamic()\n");
    report_error(errhp);
    return OCI_ERROR;
}

return OCI_SUCCESS;
}

/* ----- */
/* bind input variables.                               */
/* ----- */
static sword bind_input(OCIStmt *stmthp, OCIBind *bndhp[], OCIError *errhp)
{
    /* bind the input data by positions */
    if (bind_pos(stmthp, bndhp, errhp))
        return OCI_ERROR;

    /* bind input array attributes*/
    return (bind_array(bndhp, errhp));
}

```

```

/* ----- */
/* bind output variables.                                */
/* ----- */
static sword bind_output(OCIStmt *stmthp, OCIBind *bndhp[], OCIError *errhp)
{

    /* bind the returning bind buffers by names */
    if (bind_name(stmthp, bndhp, errhp))
        return OCI_ERROR;

    /* bind the returning bind buffers dynamically */
    return (bind_dynamic(bndhp, errhp));
}

/* ----- */
/* bind row indicator variables.                        */
/* ----- */
static sword bind_low_high(OCIStmt *stmthp, OCIBind *bndhp[], OCIError *errhp)
{
    if (OCIBindByName(stmthp, &bndhp[23], errhp,
        (text *) ":low", (sb4) strlen((char *) ":low"),
        (dvoid *) &lowcl[0], (sb4) sizeof(lowcl[0]), SOLT_INT,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT)
    || OCIBindByName(stmthp, &bndhp[24], errhp,
        (text *) ":high", (sb4) strlen((char *) ":high"),
        (dvoid *) &highcl[0], (sb4) sizeof(highcl[0]), SOLT_INT,
        (dvoid *) 0, (ub2 *)0, (ub2 *)0,
        (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIBindByName()\n");
        report_error(errhp);
        return OCI_ERROR;
    }

    if (OCIBindArrayOfStruct(bndhp[23], errhp, sl, indsk[0], rlsk[0], rcsk[0])
    || OCIBindArrayOfStruct(bndhp[24], errhp, sl, indsk[0], rlsk[0], rcsk[0]))
    {
        (void) printf("FAILED: OCIBindArrayOfStruct()\n");
        report_error(errhp);
        return OCI_ERROR;
    }

    return OCI_SUCCESS;
}

```

```

/* ----- */
/* Demonstrate INSERT with RETURNING clause. */
/* ----- */
static sword demo_insert(OCISvcCtx *svchp, OCISmt *stmthp,
                        OCIBind *bndhp[], OCIError *errhp)
{
    int i, j;

    /*
     * This function inserts values for 10 columns in table TAB1 and
     * uses the RETURNING clause to get back the inserted column values.
     * It inserts MAXITER (10) such rows. Thus it expects MAXITER values
     * for each column to be returned.
     */
    /* The Insert Statement with RETURNING clause */
    text *sqlstmt = (text *)
        "INSERT INTO TAB1 VALUES (:1, :2, :3, :4, :5, :6, :7, :8, :9, :10) \
        RETURNING C1, C2, C3, C4, C5, C6, C7, C8, C9, C10 \
        INTO :out1, :out2, :out3, :out4, :out5, :out6, \
        :out7, :out8, :out9, :out10";

    /* Prepare the statement */
    if (OCISmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISmtPrepare() insert\n");
        report_error(errhp);
        return OCI_ERROR;
    }

    /* Initialise the buffers for update */
    for (i = 0; i < MAXITER; i++)
    {
        in1[i] = i + 1;
        memset((void *)in2[i], (int) 'A'+i%26, (size_t) 40);
        memset((void *)in3[i], (int) 'a'+i%26, (size_t) 40);
        in4[i] = 400.555 + (float) i;
        in5[i] = 500 + i;
        in6[i] = 600.250 + (float) i;
        in7[i] = 700 + i;
        in8[i] = 800.350 + (float) i;
        in9[i][0] = 119;
        in9[i][1] = 185 + (ub1)i%10;
    }
}

```

```

        in9[i][2] = (ub1)i%12 + 1;
        in9[i][3] = (ub1)i%25 + 1;
        in9[i][4] = 0;
        in9[i][5] = 0;
        in9[i][6] = 0;
        for (j = 0; j < 40; j++)
            in10[i][j] = (ub1) (i%0x10);

        rowsret[i] = 0;
    }

    /* Bind all the input buffers to place holders (:1, :2, :3, etc ) */
    if (bind_input(stmthp, bndhp, errhp))
        return OCI_ERROR;

    /* Bind all the output buffers to place holders (:out1, :out2 etc */
    if (bind_output(stmthp, bndhp, errhp))
        return OCI_ERROR;

    /* Execute the Insert statement */
    if (OCISstmtExecute(svchp, stmthp, errhp, (ub4) MAXITER, (ub4) 0,
                        (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                        (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtExecute() insert\n");
        report_error(errhp);
        return OCI_ERROR;
    }

    /* Commit the changes */
    (void) OCITransCommit(svchp, errhp, (ub4) 0);

    /* Print out the values in the return rows */
    (void) printf("\n\n DEMONSTRATING INSERT...RETURNING \n");
    (void) print_return_data((int)MAXITER);

    return OCI_SUCCESS;
}

/* ----- */
/* Demonstrate UPDATE with RETURNING clause. */
/* ----- */
static sword demo_update(OCISvcCtx *svchp, OCISstmt *stmthp,
                        OCIBind *bndhp[], OCIError *errhp)
{

```

```

int    i, j;
int    range_size = 3;                                /* iterations */

/*
 * This function updates columns in table TAB1, for certain rows
 * depending on the values of the :low and :high values in
 * in the WHERE clause. It executes this statement 3 times, (3 iterations)
 * each time with a different set of values for :low and :high
 * Thus for each iteration, multiple rows are returned depending
 * on the number of rows that matched the WHERE clause.
 *
 * The rows it updates here are the rows that were inserted by the
 * cdemodr1.sql script.
 */

/* The Update Statement with RETURNING clause */
text *sqlstmt = (text *)
    "UPDATE TAB1 SET C1 = C1 + :1, C2 = :2, C3 = :3, \
      C4 = C4 + :4, C5 = C5 + :5, C6 = C6 + :6, \
      C7 = C7 + :7, C8 = C8 + :8, C9 = :9, C10 = :10 \
      WHERE C1 >= :low AND C1 <= :high \
      RETURNING C1, C2, C3, C4, C5, C6, C7, C8, C9, C10 \
      INTO :out1, :out2, :out3, :out4, :out5, :out6, \
      :out7, :out8, :out9, :out10";

/* Prepare the statement */
if (OCISmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: OCISmtPrepare() update\n");
    report_error(errhp);
    return OCI_ERROR;
}

/* Initialise the buffers for insertion */
for (i = 0; i < MAXITER; i++)
{
    in1[i] = 300 + i;
    memset((void *)in2[i], (int) 'a'+i%26, (size_t) 40);
    memset((void *)in3[i], (int) 'A'+i%26, (size_t) 40);
    in4[i] = 400.555 + (float)i;
    in5[i] = 500 + i;
    in6[i] = 600.280 + (float)i;
    in7[i] = 700 + i;

```

```

        in8[i] = 800.620 + (float)i;
        in9[i][0] = 119;
        in9[i][1] = 185 - (ub1)i%10;
        in9[i][2] = (ub1)i%12 + 1;
        in9[i][3] = (ub1)i%25 + 1;
        in9[i][4] = 0;
        in9[i][5] = 0;
        in9[i][6] = 0;
        for (j = 0; j < 40; j++)
            in10[i][j] = (ub1) (i%0x08);

        rowsret[i] = 0;
    }

    /* Bind all the input buffers to place holders (:1, :2, :3, etc ) */
    if (bind_input(stmthp, bndhp, errhp))
        return OCI_ERROR;

    /* Bind all the output buffers to place holders (:out1, :out2 etc */
    if (bind_output(stmthp, bndhp, errhp))
        return OCI_ERROR;

    /* bind row indicator low, high */
    if (bind_low_high(stmthp, bndhp, errhp))
        return OCI_ERROR;

    /* update rows
        between 101 and 103; -- expecting 3 rows returned (update 3 rows)
        between 105 and 106; -- expecting 2 rows returned (update 2 rows)
        between 109 and 113; -- expecting 5 rows returned (update 5 rows)
    */
    lowcl[0] = 101;
    highcl[0] = 103;

    lowcl[1] = 105;
    highcl[1] = 106;

    lowcl[2] = 109;
    highcl[2] = 113;

    (void) printf("\n\n DEMONSTRATING UPDATE...RETURNING \n");
    if (OCISmtExecute(svchp, stmthp, errhp, (ub4) range_size, (ub4) 0,
        (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
        (ub4) OCI_DEFAULT))
    {

```

```

        (void) printf("FAILED: OCISstmtExecute() update\n");
        report_error(errhp);
        return OCI_ERROR;
    }

    /* Commit the changes */
    (void) OCITransCommit(svcHP, errhp, (ub4) 0);

    /* Print out the values in the return rows */
    (void) print_return_data(range_size);

    return OCI_SUCCESS;
}

/* ----- */
/* Demonstrate DELETE with RETURNING clause. */
/* ----- */
static sword demo_delete(OCISvcCtx *svchp, OCISstmt *stmthp,
                        OCIBind *bndhp[], OCIError *errhp)
{
    int i, range_size = 3; /* iterations */
    sword retval;

    /*
     * This function deletes certain rows from table TAB1
     * depending on the values of the :low and :high values in
     * the WHERE clause. It executes this statement 3 times, (3 iterations)
     * each time with a different set of values for :low and :high
     * Thus for each iteration, multiples rows are returned depending
     * on the number of rows that matched the WHERE clause.
     *
     * The rows it deletes here are the rows that were inserted by the
     * cdemodr1.sql script.
     */

    /* The Delete Statement with RETURNING clause */
    text *sqlstmt = (text *)
        "DELETE FROM TAB1 WHERE C1 >= :low AND C1 <= :high \
         RETURNING C1, C2, C3, C4, C5, C6, C7, C8, C9, C10 \
         INTO :out1, :out2, :out3, :out4, :out5, :out6, \
         :out7, :out8, :out9, :out10";

    /* Prepare the statement */
    if (OCISstmtPrepare(stmthp, errhp, sqlstmt, (ub4)strlen((char *)sqlstmt),
                        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))

```

```

{
    (void) printf("FAILED: OCISstmtPrepare() delete\n");
    report_error(errhp);
    return OCI_ERROR;
}

/* Bind all the output buffers to place holders (:out1, :out2 etc */
if (bind_output(stmthp, bndhp, errhp))
    return OCI_ERROR;

/* bind row indicator low, high */
if (bind_low_high(stmthp, bndhp, errhp))
    return OCI_ERROR;

/* delete rows
    between 201 and 203; -- expecting 3 rows returned (3 rows deleted)
    between 205 and 209; -- expecting 5 rows returned (2 rows deleted)
    between 211 and 213; -- expecting 3 rows returned (5 rows deleted)
*/
lowcl[0] = 201;
highcl[0] = 203;

lowcl[1] = 205;
highcl[1] = 209;

lowcl[2] = 211;
highcl[2] = 213;

for (i=0; i<MAXITER; i++)
    rowsret[i] = 0;

(void) printf("\n\n Demonstrating DELETE....RETURNING \n");
if ((retval = OCISstmtExecute(svchp, stmthp, errhp, (ub4) range_size, (ub4) 0,
    (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
    (ub4) OCI_DEFAULT)) != OCI_SUCCESS &&
    retval != OCI_SUCCESS_WITH_INFO)
{
    (void) printf("FAILED: OCISstmtExecute() delete, retval = %d\n", retval);
    report_error(errhp);
}

/* Commit the changes */
(void) OCITransCommit(svchp, errhp, (ub4) 0);

```



```

/* Print out the values in the return rows */
(void) print_return_data(range_size);

return OCI_SUCCESS;
}

/* ----- */
/* IN bind callback that does not do any data input. */
/* ----- */
static sb4 cbf_no_data(dvoid *ctxp, OCIBind *bindp, ub4 iter, ub4 index,
                      dvoid **bufpp, ub4 *alenpp, ub1 *piecep, dvoid **indpp)
{
    /*
     * This is a dummy input callback function that provides input data
     * for the bind variables in the RETURNING clause.
     */
    *bufpp = (dvoid *) 0;
    *alenpp = 0;
    *indpp = (dvoid *) 0;
    *piecep = OCI_ONE_PIECE;

    return OCI_CONTINUE;
}

/* ----- */
/* Outbind callback for returning data. */
/* ----- */
static sb4 cbf_get_data(dvoid *ctxp, OCIBind *bindp, ub4 iter, ub4 index,
                      dvoid **bufpp, ub4 **alenp, ub1 *piecep,
                      dvoid **indpp, ub2 **rcodepp)
{
    /*
     * This is the callback function that is called to receive the OUT
     * bind values for the bind variables in the RETURNING clause
     */

    static ub4 rows = 0;
    ub4 pos = *((ub4 *)ctxp);

    /* For each iteration the OCI_ATTR_ROWS_RETURNED tells us the number
     * of rows returned in that iteration. So we can use this information
     * to dynamically allocate storage for all the returned rows for that
     * bind.
     */
    if (index == 0)

```

```

{
    (void) OCIAttrGet((CONST dvoid *)bindp, OCI_HTYPE_BIND, (dvoid *)&rows,
                     (ub4 *) sizeof(ub4), OCI_ATTR_ROWS_RETURNED, errhp);
    rowsret[iter] = (ub2)rows;

    /* Dynamically allocate storage */
    if (alloc_buffer(pos, iter, rows))
        return OCI_ERROR;
}

/* Provide the address of the storage where the data is to be returned */
switch(pos)
{
case 0:
    rl[pos][iter][index] = sizeof(int);
    *bufpp = (dvoid *) (p1[iter]+ index);
    break;
case 1:
    rl[pos][iter][index] = (ub4) MAXCOLLEN;
    *bufpp = (dvoid *) (p2[iter]+(index * MAXCOLLEN));
    break;
case 2:
    rl[pos][iter][index] = (ub4) MAXCOLLEN;
    *bufpp = (dvoid *) (p3[iter]+(index * MAXCOLLEN));
    break;
case 3:
    rl[pos][iter][index] = sizeof(float);
    *bufpp = (dvoid *) (p4[iter]+ index);
    break;
case 4:
    rl[pos][iter][index] = sizeof(int);
    *bufpp = (dvoid *) (p5[iter]+index);
    break;
case 5:
    rl[pos][iter][index] = sizeof(float);
    *bufpp = (dvoid *) (p6[iter]+index );
    break;
case 6:
    rl[pos][iter][index] = sizeof(int);
    *bufpp = (dvoid *) (p7[iter]+ index);
    break;
case 7:
    rl[pos][iter][index] = sizeof(float);
    *bufpp = (dvoid *) (p8[iter]+index);
    break;

```

```

case 8:
    rl[pos][iter][index] = DATBUFLLEN;
    *bufpp = (dvoid *) (p9[iter]+(index * DATBUFLLEN));
    break;
case 9:
    rl[pos][iter][index] = (ub4) MAXCOLLEN;
    *bufpp = (dvoid *) (p10[iter]+(index * MAXCOLLEN));
    break;
default:
    *bufpp = (dvoid *) 0;
    *alenp = (ub4 *) 0;
    (void) printf("ERROR: invalid position number: %d\n", *((ub2 *)ctxp));
}

*piecep = OCI_ONE_PIECE;

/* provide address of the storage where the indicator will be returned */
ind[pos][iter][index] = 0;
*indpp = (dvoid *) &ind[pos][iter][index];

/* provide address of the storage where the return code will be returned */
rc[pos][iter][index] = 0;
*rcepp = &rc[pos][iter][index];

/*
 * provide address of the storage where the actual length will be
 * returned
 */
*alenp = &rl[pos][iter][index];

return OCI_CONTINUE;
}

/* ----- */
/* allocate buffers for callback. */
/* ----- */
static sword alloc_buffer(ub4 pos, ub4 iter, ub4 rows)
{
    switch(pos)
    {
    case 0:
        p1[iter] = (int *) malloc(sizeof(int) * rows);
        break;
    case 1:

```

```

        p2[iter] = (text *) malloc(rows * MAXCOLLEN);
        break;
    case 2:
        p3[iter] = (text *) malloc(rows * MAXCOLLEN);
        break;
    case 3:
        p4[iter] = (float *) malloc(sizeof(float) * rows);
        break;
    case 4:
        p5[iter] = (int *) malloc(sizeof(int) * rows);
        break;
    case 5:
        p6[iter] = (float *) malloc(sizeof(float) * rows);
        break;
    case 6:
        p7[iter] = (int *) malloc(sizeof(int) * rows);
        break;
    case 7:
        p8[iter] = (float *) malloc(sizeof(float) * rows);
        break;
    case 8:
        p9[iter] = (ub1 *) malloc(rows * DATBUFLLEN);
        break;
    case 9:
        p10[iter] = (ub1 *) malloc(rows * MAXCOLLEN);
        break;
    default:
        (void) printf("ERROR: invalid position number: %d\n", pos);
        return OCI_ERROR;
}

ind[pos][iter] = (short *) malloc(rows * sizeof(short));
rc[pos][iter] = (ub2 *) malloc(rows * sizeof(ub2));
rl[pos][iter] = (ub4 *) malloc(rows * sizeof(ub4));

return OCI_SUCCESS;
}

/* ----- */
/* print the returned data. */
/* ----- */
static sword print_return_data(iters)
int iters;
{
    int i, j;

```

```

for (i = 0; i < iters; i++)
{
    (void) printf("\n*** ITERATION *** : %d\n", i);
    (void) printf("(...returning %d rows)\n", rowsret[i]);

    for (j = 0; j < rowsret[i] ; j++)
    {
        /* Column 1 */
        (void) printf("COL1 [%d]: ind = %d, rc = %d, retl = %d\n",
                      j, ind[0][i][j], rc[0][i][j], rl[0][i][j]);
        if (ind[0][i][j] == -1)
            (void) printf("COL1 [%d]: NULL\n", j);
        else
            (void) printf("COL1 [%d]: %d\n", j, *(p1[i]+j) );

        /* Column 2 */
        (void) printf("COL2 [%d]: ind = %d, rc = %d, retl = %d\n",
                      j, ind[1][i][j], rc[1][i][j], rl[1][i][j]);
        if (ind[1][i][j] == -1)
            (void) printf("COL2 [%d]: NULL\n", j);
        else
            (void) printf("COL2 [%d]: %.*s\n", j, rl[1][i][j], p2[i]+(j*MAXCOLLEN) );

        /* Column 3 */
        (void) printf("COL3 [%d]: ind = %d, rc = %d, retl = %d\n",
                      j, ind[2][i][j], rc[2][i][j], rl[2][i][j]);
        if (ind[2][i][j] == -1)
            (void) printf("COL3 [%d]: NULL\n", j);
        else
            (void) printf("COL3 [%d]: %.*s\n", j, rl[2][i][j], p3[i]+(j*MAXCOLLEN) );
        /* Column 4 */
        (void) printf("COL4 [%d]: ind = %d, rc = %d, retl = %d\n",
                      j, ind[3][i][j], rc[3][i][j], rl[3][i][j]);
        if (ind[3][i][j] == -1)
            (void) printf("COL4 [%d]: NULL\n", j);
        else
            (void) printf("COL4 [%d]: %8.3f\n", j, *(p4[i]+j) );

        /* Column 5 */
        (void) printf("COL5 [%d]: ind = %d, rc = %d, retl = %d\n",
                      j, ind[4][i][j], rc[4][i][j], rl[4][i][j]);
        if (ind[4][i][j] == -1)
            (void) printf("COL5 [%d]: NULL\n", j);
        else

```

```

        (void) printf("COL5 [%d]: %d\n", j, *(p5[i]+j) );

/* Column 6 */
(void) printf("COL6 [%d]: ind = %d, rc = %d, retl = %d\n",
              j, ind[5][i][j], rc[5][i][j], rl[5][i][j]);
if (ind[5][i][j] == -1)
    (void) printf("COL6 [%d]: NULL\n", j);
else
    (void) printf("COL6 [%d]: %8.3f\n", j, *(p6[i]+j) );

/* Column 7 */
(void) printf("COL7 [%d]: ind = %d, rc = %d, retl = %d\n",
              j, ind[6][i][j], rc[6][i][j], rl[6][i][j]);
if (ind[6][i][j] == -1)
    (void) printf("COL7 [%d]: NULL\n", j);
else
    (void) printf("COL7 [%d]: %d\n", j, *(p7[i]+j) );

/* Column 8 */
(void) printf("COL8 [%d]: ind = %d, rc = %d, retl = %d\n",
              j, ind[7][i][j], rc[7][i][j], rl[7][i][j]);
if (ind[7][i][j] == -1)
    (void) printf("COL8 [%d]: NULL\n", j);
else
    (void) printf("COL8 [%d]: %8.3f\n", j, *(p8[i]+j) );

/* Column 9 */
(void) printf("COL9 [%d]: ind = %d, rc = %d, retl = %d\n",
              j, ind[8][i][j], rc[8][i][j], rl[8][i][j]);
if (ind[8][i][j] == -1)
    (void) printf("COL9 [%d]: NULL\n", j);
else
    (void) printf("COL9 [%d]: %u-%u-%u\n", j,
                  *(p9[i]+(j*DATABUFLEN+3)),
                  *(p9[i]+(j*DATABUFLEN+2)),
                  *(p9[i]+(j*DATABUFLEN+0)) - 100,
                  *(p9[i]+(j*DATABUFLEN+1)) - 100 );

/* Column 10 */
(void) printf("COL10 [%d]: ind = %d, rc = %d, retl = %d\n",
              j, ind[9][i][j], rc[9][i][j], rl[9][i][j]);
if (ind[9][i][j] == -1)
    (void) printf("COL10 [%d]: NULL\n", j);
else
{

```

```

        (void) printf("COL10 [%d]: ", j);
        print_raw(p10[i]+(j*MAXCOLLEN), rl[9][i][j]);
    }
    (void) printf("\n");
}

return OCI_SUCCESS;
}

```

cdemodr1.h

```

/*-----
 * Include Files
 */
#include <stdio.h>
#include <string.h>
#include <oci.h>

/*-----
 * Define Constants
 */

#define MAXBINDS      25
#define MAXROWS       5          /* max no of rows returned per iter */
#define MAXCOLS       10
#define MAXITER       10        /* max no of iters in execute */
#define MAXCOLLEN     40        /* if changed, update cdemodr1.sql */
#define DATEBUFLLEN   7

int main(/*_ int argc, char *argv[] _*/);
static sword init_handles(/*_ OCIEnv **envhp, OCISvcCtx **svchp,
                        OCIError **errhp, OCIServer **svrhp,
                        OCISession **authhp, ub4 mode _*/);

static sword attach_server(/*_ ub4 mode, OCIServer *svrhp,
                        OCIError *errhp, OCISvcCtx *svchp _*/);
static sword log_on(/*_ OCISession *authhp, OCIError *errhp, OCISvcCtx *svchp,
                    text *uid, text *pwd, ub4 credt, ub4 mode _*/);
static sword alloc_bind_handle(/*_ OCISmt *stnhp, OCIBind *bndhp[],
                               int nbinds _*/);
static void print_raw(/*_ ub1 *raw, ub4 rawlen _*/);

```

```

static void free_handles(/*_ OCIEEnv *envhp, OCISvcCtx *svchp, OCISever
    *srvhp, OCIError *errhp, OCISession *authp, OCISmt *stmthp _*/);
void report_error(/*_ OCIError *errhp _*/);
void logout_detach_server(/*_ OCISvcCtx *svchp, OCISever *srvhp,
    OCIError *errhp, OCISession *authp,
    text *userid _*/);
sword finish_demo(/*_ boolean loggedon, OCIEEnv *envhp, OCISvcCtx *svchp,
    OCISever *srvhp, OCIError *errhp, OCISession *authp,
    OCISmt *stmthp, text *userid _*/);
static sword demo_insert(/*_ OCISvcCtx *svchp, OCISmt *stmthp,
    OCIBind *bndhp[], OCIError *errhp _*/);
static sword demo_update(/*_ OCISvcCtx *svchp, OCISmt *stmthp,
    OCIBind *bndhp[], OCIError *errhp _*/);
static sword demo_delete(/*_ OCISvcCtx *svchp, OCISmt *stmthp,
    OCIBind *bndhp[], OCIError *errhp _*/);
static sword bind_name(/*_ OCISmt *stmthp, OCIBind *bndhp[],
    OCIError *errhp _*/);
static sword bind_pos(/*_ OCISmt *stmthp, OCIBind *bndhp[],
    OCIError *errhp _*/);
static sword bind_input(/*_ OCISmt *stmthp, OCIBind *bndhp[],
    OCIError *errhp _*/);
static sword bind_output(/*_ OCISmt *stmthp, OCIBind *bndhp[],
    OCIError *errhp _*/);
static sword bind_array(/*_ OCIBind *bndhp[], OCIError *errhp _*/);
static sword bind_dynamic(/*_ OCIBind *bndhp[], OCIError *errhp _*/);
static sb4 cbf_no_data(/*_ dvoid *ctxp, OCIBind *bindp, ub4 iter, ub4 index,
    dvoid **bufpp, ub4 *alenpp, ub1 *piecep, dvoid **indpp _*/);
static sb4 cbf_get_data(/*_ dvoid *ctxp, OCIBind *bindp, ub4 iter, ub4 index,
    dvoid **bufpp, ub4 **alenpp, ub1 *piecep,
    dvoid **indpp, ub2 **rcodepp _*/);
static sword alloc_buffer(/*_ ub4 pos, ub4 iter, ub4 rows _*/);
static sword print_return_data(/*_ int iter _*/);

```


Example 4, Describing an Object

```

/*
    NAME
        cdemodsc.c

    DESCRIPTION
        Tests OCIDescribeAny() on an object.

        cdemodsc takes the user name and password and a type name
        (created in the database) as command line arguments and
        dumps all the information about the type --
        its attribute types, methods,
        method parameters, etc.

*/

#ifdef CDEMODSC_ORACLE
#include "cdemodsc.h"
#endif

/*****
static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    sb4 errcode;

    switch (status)
    {
    case OCI_SUCCESS:
        break;
    case OCI_SUCCESS_WITH_INFO:
        break;
    case OCI_NEED_DATA:
        break;
    case OCI_NO_DATA:
        break;
    case OCI_ERROR:
        DISCARD OCIErrGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                           errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
        DISCARD printf("Error - %s\n", errbuf);
        exit(1);
        break;
*****/

```

```

        case OCI_INVALID_HANDLE:
            break;
        case OCI_STILL_EXECUTING:
            break;
        case OCI_CONTINUE:
            break;
        default:
            break;
    }
}

/*-----*/

static void chk_methodlst(envhp, errhp, svchp, pamp, count, comment)
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
dvoid *pamp;
ub4 count;
const text *comment;
{
    sword retval;
    ub4 pos;
    dvoid *parmdp;

    for (pos = 1; pos <= count; pos++)
    {
        checkerr(errhp, OCIParmGet((dvoid *)pamp, (ub4) OCI_DTYPE_PARAM, errhp,
                                   (dvoid *)&parmdp, (ub4) pos));
        chk_method(envhp, errhp, svchp, parmdp, comment);
    }
}

/*-----*/

static void chk_method(envhp, errhp, svchp, pamp, comment)
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
dvoid *pamp;
const text *comment;
{
    sword retval;
    text method[MAXNAME],
        *namep;

```

```

ub4    size;
ub4    num_arg;
ub1    has_result,
        is_selfish,
        is_virtual,
        is_inline,
        is_constructor,
        is_destructor,
        is_constant,
        is_operator,
        is_map,
        is_order,
        is_rnds,
        is_rnps,
        is_wnds,
        is_wnps;
OCITypeEncap encap;
dvoid *list_arg;

/* get name of the method */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &namep, (ub4 *) &size,
                          (ub4) OCI_ATTR_NAME, (OCIErr *) errhp));

(void) strncpy((char *)method, (char *)namep, (size_t) size);
method[size] = '\0';

/* get the number of arguments */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &num_arg, (ub4 *) 0,
                          (ub4) OCI_ATTR_NUM_ARGS, (OCIErr *) errhp));

/* encapsulation (public?) */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &encap, (ub4 *) 0,
                          (ub4) OCI_ATTR_ENCAPSULATION, (OCIErr *) errhp));

/* has result */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *) &has_result, (ub4 *) 0,
                          (ub4) OCI_ATTR_HAS_RESULT, (OCIErr *) errhp));

/* map method */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *) &is_map, (ub4 *) 0,

```

```

        (ub4)OCI_ATTR_IS_MAP, (OCIError *) errhp));

/* order method */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_order, (ub4 *)0,
        (ub4)OCI_ATTR_IS_ORDER, (OCIError *) errhp));

/* selfish method */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_selfish, (ub4 *)0,
        (ub4)OCI_ATTR_IS_SELFISH, (OCIError *) errhp));

/* virtual method */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_virtual, (ub4 *)0,
        (ub4)OCI_ATTR_IS_VIRTUAL, (OCIError *) errhp));

/* inline method */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_inline, (ub4 *)0,
        (ub4)OCI_ATTR_IS_INLINE, (OCIError *) errhp));

/* constant method */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_constant, (ub4 *)0,
        (ub4)OCI_ATTR_IS_CONSTANT, (OCIError *) errhp));

/* operator */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_operator, (ub4 *)0,
        (ub4)OCI_ATTR_IS_OPERATOR, (OCIError *) errhp));

/* constructor method */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_constructor, (ub4 *)0,
        (ub4)OCI_ATTR_IS_CONSTRUCTOR, (OCIError *) errhp));

/* destructor method */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_destructor, (ub4 *)0,
        (ub4)OCI_ATTR_IS_DESTRUCTOR, (OCIError *) errhp));

checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&is_rnds, (ub4 *)0,
        (ub4)OCI_ATTR_IS_RNDS, (OCIError *) errhp));

```

```

checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *)&is_rnps, (ub4 *)0,
                          (ub4)OCI_ATTR_IS_RNPS, (OCIError *) errhp));
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *)&is_wnds, (ub4 *)0,
                          (ub4)OCI_ATTR_IS_WNDS, (OCIError *) errhp));
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *)&is_wnps, (ub4 *)0,
                          (ub4)OCI_ATTR_IS_WNPS, (OCIError *) errhp));

/* get list of arguments */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *)&list_arg, (ub4 *)0,
                          (ub4)OCI_ATTR_LIST_ARGUMENTS, (OCIError *) errhp));

SPACING;
printf ( "\n%s\n", comment);
SPACING;
printf ( "Name:                %s\n", method);
SPACING;
printf ( "Number of args:       %d\n", num_arg);
SPACING;
printf ( "Encapsulation:         %s\n",
        (encap==OCI_TYPEENCAP_PUBLIC) ? "public" : "private");
SPACING;
printf ( "Has result:             %d\n", has_result);
SPACING;
printf ( "Is selfish:             %d\n", is_selfish);
SPACING;
printf ( "Is virtual:             %d\n", is_virtual);
SPACING;
printf ( "Is inline:              %d\n", is_inline);
SPACING;
printf ( "Is constructor:        %d\n", is_constructor);
SPACING;
printf ( "Is destructor:         %d\n", is_destructor);
SPACING;
printf ( "Is constant:           %d\n", is_constant);
SPACING;
printf ( "Is operator:           %d\n", is_operator);
SPACING;
printf ( "Is map:                %d\n", is_map);
SPACING;
printf ( "Is order:              %d\n", is_order);
SPACING;

```

```

printf ( "Is RNDs:           %d\n", is_rnds);
SPACING;
printf ( "Is RNPS:           %d\n", is_rnps);
SPACING;
printf ( "Is WNPS:           %d\n", is_wnps);
printf("\n");

if (has_result)
    chk_arg(envhp, errhp, svchp, list_arg, OCI_PTYPE_TYPE_RESULT, 0, 1);
if (num_arg > 0)
    chk_arg(envhp, errhp, svchp, list_arg, OCI_PTYPE_TYPE_ARG, 1, num_arg + 1);
}

/*-----*/

static void chk_arglst(envhp, errhp, svchp, pamp)
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
dvoid *pamp;
{
    dvoid *arglst;
    ub4 numargs;
    ub1 ptype;
    sword retval;

    /* get list of arguments */
    checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &arglst, (ub4 *) 0,
        (ub4) OCI_ATTR_LIST_ARGUMENTS, (OCIError *) errhp));

    /* get number of parameters */
    checkerr(errhp, OCIAttrGet((dvoid*) arglst, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &numargs, (ub4 *) 0,
        (ub4) OCI_ATTR_NUM_PARAMS, (OCIError *) errhp));

    checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &ptype, (ub4 *) 0,
        (ub4) OCI_ATTR_PTYPE, (OCIError *) errhp));

    switch (ptype)
    {
    case OCI_PTYPE_FUNC:
        chk_arg (envhp, errhp, svchp, arglst, OCI_PTYPE_ARG, 0, numargs);
        break;
    }
}

```

```

    case OCI_PTYPE_PROC:
        chk_arg (envhp, errhp, svchp, arglst, OCI_PTYPE_ARG, 1, numargs);
    }
}

/*-----*/

static void chk_arg (envhp, errhp, svchp, pamp, type, start, end)
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
dvoid *pamp;
ub1 type;
ub4 start;
ub4 end;
{
    text argname[NPOS][30];
    text *namep;
    ub4 sizep;
    ub2 collen[NPOS];
    ub2 coldesr[NPOS];
    dvoid *pamdp;
    ub4 i, pos;
    sword retval;
    ub2 level[NPOS];
    ub1 radix[NPOS], def[NPOS];
    ub4 iomode[NPOS];
    ub1 precision[NPOS], scale[NPOS], isnull[NPOS];

    for (pos = start; pos < end; pos++)
    {

        checkerr(errhp, OCIParmGet((dvoid *)pamp, (ub4) OCI_DTYPE_PARAM, errhp,
                                   (dvoid *)&pamdp, (ub4) pos));

        /* get data type */
        checkerr(errhp, OCIAttrGet((dvoid*) pamdp, (ub4) OCI_DTYPE_PARAM,
                                   (dvoid*) &coldesr[pos], (ub4 *) 0,
                                   (ub4) OCI_ATTR_DATA_TYPE,
                                   (OCIError *) errhp));

        /* method's result has no name */
        iomode[pos] = 0;
        def[pos] = 0;
    }
}

```

```

sizep = 0;
if (type != OCI_PTYPE_TYPE_RESULT)
{
    /* has default */
    checkerr(errhp, OCIAttrGet((dvoid *)parmdp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid *)&def[pos], (ub4 *)0,
                              (ub4)OCI_ATTR_HAS_DEFAULT, (OCIError *) errhp));

    /* get iomode */
    checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &iomode[pos], (ub4 *) 0,
                              (ub4) OCI_ATTR_IOMODE, (OCIError *) errhp));

    /* get argument name */
    checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &namep, (ub4 *) &sizep,
                              (ub4) OCI_ATTR_NAME, (OCIError *) errhp));

    (void) strncpy((char *)argname[pos], (char *)namep,
                  (size_t) sizep);
}
argname[pos][sizep] = '\0';

/* the following are not for type arguments and results */
precision[pos] = 0;
scale[pos] = 0;
collen[pos] = 0;
level[pos] = 0;
radix[pos] = 0;
isnull[pos] = FALSE;
if (type != OCI_PTYPE_TYPE_ARG && type != OCI_PTYPE_TYPE_RESULT)
{
    /* get the data size */
    checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &collen[pos], (ub4 *) 0,
                              (ub4) OCI_ATTR_DATA_SIZE, (OCIError *) errhp));

    /* get the precision of the attribute */
    checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &precision, (ub4 *) 0,
                              (ub4) OCI_ATTR_PRECISION, (OCIError *) errhp));

    /* get the scale of the attribute */
    checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &scale, (ub4 *) 0,

```



```

        (ub4) OCI_ATTR_SCALE, (OCIError *) errhp));

/* get the level of the attribute */
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &level[pos], (ub4 *) 0,
        (ub4) OCI_ATTR_LEVEL, (OCIError *) errhp));

/* get the radix of the attribute */
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &radix[pos], (ub4 *) 0,
        (ub4) OCI_ATTR_RADIX, (OCIError *) errhp));

/* is null */
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &isnull, (ub4 *) 0,
        (ub4) OCI_ATTR_IS_NULL, (OCIError *) errhp));

/* should get error 24328 */
if (OCIAttrGet((dvoid*) parm, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &isnull, (ub4 *) 0,
        (ub4) OCI_ATTR_INDEX_ONLY, (OCIError *) errhp)
    != OCI_ERROR)
    printf("ERROR: should get error here\n");
    }
}

SPACING;
(void)
printf("Argument Name  Length  Datatype  Level Radix Default Iomode Prec Scl
Null\n");
SPACING;
(void)
printf
("_____ \n");
for (i = start; i < end; i++)
{
    SPACING;
    (void) printf( "%15s%6d%8d%6d%6d      %c%6d%9d%4d%4d\n", argname[i],
        collen[i], coldesr[i], level[i], radix[i],
        (def[i])?'y':'n', iomode[i], precision[i], scale[i],
        isnull[i]);
}
printf("\n");
}

```

```

static void chk_collection (envhp, errhp, svchp, pamp, is_array)
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
dvoid *pamp;
sword is_array;
{
    text          schema[MAXNAME],
                  type[MAXNAME],
                  *namep;
    ub4           size;
    ub2           len;
    ub4           num_elems;
    OCITypeCode   typecode;
    sword         retval;

    /* get the data size */
    checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &len, (ub4 *) 0,
                              (ub4) OCI_ATTR_DATA_SIZE, (OCIError *) errhp));

    /* get the name of the collection */
    checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &namep, (ub4 *) &size,
                              (ub4) OCI_ATTR_TYPE_NAME, (OCIError *) errhp));

    (void) strncpy((char *)type, (char *)namep, (size_t) size);
    type[size] = '\0';

    /* get the name of the schema */
    checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &namep, (ub4 *) &size,
                              (ub4) OCI_ATTR_SCHEMA_NAME, (OCIError *) errhp));

    (void) strncpy((char *)schema, (char *)namep, (size_t) size);
    schema[size] = '\0';

    /* get the data type */
    checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &typecode, (ub4 *) 0, (ub4) OCI_ATTR_DATA_TYPE,
                              (OCIError *) errhp));

    num_elems = 0;
    if (is_array)

```

```

/* get the number of elements */
checkerr(errhp, OCIAttrGet((dvoid*) parm, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &num_elems, (ub4 *) 0,
                          (ub4) OCI_ATTR_NUM_ELEMS, (OCIError *) errhp));

SPACING;
(void)
printf ( "Schema      Type              Length  Datatype Elements\n");
SPACING;
(void)
printf ( "_____ \n");
SPACING;
(void) printf( "%10s%16s%6d%11d%9d\n", schema, type, len, typecode,
              num_elems);
printf("\n");
}

/*-----*/

static void chk_column(envhp, errhp, svchp, parm, parmcnt)
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
dvoid *parm;
ub4 parmcnt;
{
    text colname1[NPOS][30], colname2[NPOS][30], colname3[NPOS][30];
    text *namep;
    ub4 sizep;
    ub2 collen[NPOS];
    ub2 coldesr[NPOS];
    dvoid *parmdp;
    ub4 i, pos;
    sword retval;

    /* loop through all the attributes in the type and get all information */
    for (pos = 1; pos <= parmcnt; pos++)
    {
        /* get the parameter list for each attribute */
        checkerr(errhp, OCIParmGet((dvoid *) parm, (ub4) OCI_DTYPE_PARAM, errhp,
                                   (dvoid *)&parmdp, (ub4) pos));

        /* size of the attribute (non object or REF) objects */
        checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                                   (dvoid*) &collen[pos-1], (ub4 *) 0,

```

```

        (ub4) OCI_ATTR_DATA_SIZE, (OCIError *) errhp));

/* name of the attribute */
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &namep, (ub4 *) &sizep,
        (ub4) OCI_ATTR_NAME, (OCIError *) errhp));

(void) strncpy((char *)colname1[pos-1], (char *)namep, (size_t) sizep);
colname1[pos-1][sizep] = '\0';

/* get the schema name */
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &namep, (ub4 *) &sizep,
        (ub4) OCI_ATTR_SCHEMA_NAME, (OCIError *) errhp));

(void) strncpy((char *)colname2[pos-1], (char *)namep, (size_t) sizep);
colname2[pos-1][sizep] = '\0';

/* name of the attribute */
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &namep, (ub4 *) &sizep,
        (ub4) OCI_ATTR_TYPE_NAME, (OCIError *) errhp));

(void) strncpy((char *)colname3[pos-1], (char *)namep, (size_t) sizep);
colname3[pos-1][sizep] = '\0';

/* get data type */
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &coldesr[pos-1], (ub4 *) 0,
        (ub4) OCI_ATTR_DATA_TYPE,
        (OCIError *) errhp));

if (coldesr[pos-1] == SQLT_NTY || coldesr[pos-1] == SQLT_REF)
{
    /* call tst_desc_type here if the type is object or REF */
    tab += 5;
    SPACING;
    printf("!!!!ATTRIBUTE IS A TYPE OR REF!!!!\n");
    SPACING;
    printf("ATTRIBUTE NAME IS %s\n", colname3[pos-1]);
    SPACING;
    printf("ATTRIBUTE TYPE IS %d\n", coldesr[pos-1]);
    tst_desc_type(envhp, errhp, svchp, colname3[pos-1]);
    tab -= 5;
    printf("\n");
}

```

```

    }

}

SPACING;
(void)
    printf ( "Column Name      Schema      Type              Length  Datatype\n");
SPACING;
(void)
    printf ( "_____ \n");
for (i = 1; i <= parmcnt; i++)
{
    SPACING;
    (void) printf( "%15s%10s%16s%6d%8d\n",  colname1[i-1], colname2[i-1],
                colname3[i-1], collen[i-1], coldesr[i-1] );
}
printf("\n");
}

/*-----*/

static void tst_desc_type(envhp, errhp, svchp, objname)
OCIEnv *envhp;
OCIError *errhp;
OCISvcCtx *svchp;
text *objname;
{
    OCIDescribe *dschp = (OCIDescribe *) 0;
    sword retval;
    OCITypeCode typecode,
        collection_typecode;
    text  schema[MAXNAME],
        version[MAXNAME],
        *namep,
        *type_name;
    ub4  size,
        text_len;
    OCIRef *type_ref;
    ub2  num_attr,
        num_method;
    ub1  is_incomplete,
        is_system,
        is_predefined,
        is_transient,
        is_sysgen,

```

```

        has_table,
        has_lob,
        has_file;
dvoid *list_attr,
    *list_method,
    *map_method,
    *order_method,
    *collection_dschnp,
    *some_object;
OCIParam *pamp;
ub1 objtype;

/* must allocate describe handle first for OCIDescribeAny */
checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **) &dschnp,
    (ub4) OCI_HTYPE_DESCRIBE,
    (size_t) 0, (dvoid **) 0));

/* call OCIDescribeAny and passing in the type name */
checkerr(errhp, OCIDescribeAny(svchnp, errhp, (text *)objname,
    (ub4) strlen((char *)objname), OCI_OTYPE_NAME, (ub1)1,
    (ub1) OCI_PTYPE_TYPE, dschnp));

/* get the parameter list for the requested type */
checkerr(errhp, OCIAttrGet((dvoid *) dschnp, (ub4) OCI_HTYPE_DESCRIBE,
    (dvoid *)&pamp, (ub4 *)0, (ub4)OCI_ATTR_PARAM, errhp));

/* get the schema name for the requested type */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
    (dvoid*) &namep, (ub4 *) &size,
    (ub4) OCI_ATTR_SCHEMA_NAME, (OCIError *) errhp));

(void) strncpy((char *)schema, (char *)namep, (size_t) size);
schema[size] = '\0';

/* get the type code for the requested type */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
    (dvoid*) &typecode, (ub4 *) 0,
    (ub4) OCI_ATTR_TYPECODE, (OCIError *) errhp));

/* get other information for collection type */
if (typecode == OCI_TYPECODE_NAMEDCOLLECTION)
{
    checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
        (dvoid *)&collection_typecode, (ub4 *)0,
        (ub4)OCI_ATTR_COLLECTION_TYPECODE, (OCIError *)errhp));
}

```

```

    checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                               (dvoid *)&collection_dschnp, (ub4 *)0,
                               (ub4)OCI_ATTR_COLLECTION_ELEMENT, (OCIError *)errhp));
    checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                               (dvoid *)&collection_dschnp, (ub4 *)0,
                               (ub4)OCI_ATTR_COLLECTION_ELEMENT, (OCIError *)errhp));
}

/* get the ref to the type descriptor */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                           (dvoid*) &type_ref, (ub4 *) 0,
                           (ub4) OCI_ATTR_REF_TDO, (OCIError *) errhp));

/* get the type version */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                           (dvoid*) &namep, (ub4 *) &size,
                           (ub4) OCI_ATTR_VERSION, (OCIError *) errhp));

(void) strncpy((char *)version, (char *)namep, (size_t) size);
version[size] = '\0';

/* incomplete type */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                           (dvoid*) &is_incomplete, (ub4 *) 0,
                           (ub4) OCI_ATTR_IS_INCOMPLETE_TYPE, (OCIError *) errhp));

/* system type */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                           (dvoid*) &is_system, (ub4 *) 0,
                           (ub4) OCI_ATTR_IS_SYSTEM_TYPE, (OCIError *) errhp));

/* predefined type */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                           (dvoid*) &is_predefined, (ub4 *) 0,
                           (ub4) OCI_ATTR_IS_PREDEFINED_TYPE, (OCIError *) errhp));

/* transient type */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                           (dvoid*) &is_transient, (ub4 *) 0,
                           (ub4) OCI_ATTR_IS_TRANSIENT_TYPE, (OCIError *) errhp));

/* system generated type */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                           (dvoid*) &is_sysgen, (ub4 *) 0,
                           (ub4) OCI_ATTR_IS_SYSTEM_GENERATED_TYPE, (OCIError*) errhp));

```

```

/* has nested table */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &has_table, (ub4 *) 0,
                          (ub4) OCI_ATTR_HAS_NESTED_TABLE, (OCIError *) errhp));

/* has lob */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &has_lob, (ub4 *) 0,
                          (ub4) OCI_ATTR_HAS_LOB, (OCIError *) errhp));

/* has file */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &has_file, (ub4 *) 0,
                          (ub4) OCI_ATTR_HAS_FILE, (OCIError *) errhp));

/* get the list of attributes */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *) &list_attr, (ub4 *) 0,
                          (ub4) OCI_ATTR_LIST_TYPE_ATTRS, (OCIError *) errhp));

/* number of attributes */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &num_attr, (ub4 *) 0,
                          (ub4) OCI_ATTR_NUM_TYPE_ATTRS, (OCIError *) errhp));

/* get method list */
checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *) &list_method, (ub4 *) 0,
                          (ub4) OCI_ATTR_LIST_TYPE_METHODS, (OCIError *) errhp));

/* get number of methods */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &num_method, (ub4 *) 0,
                          (ub4) OCI_ATTR_NUM_TYPE_METHODS, (OCIError *) errhp));

/* get map method list */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &map_method, (ub4 *) 0,
                          (ub4) OCI_ATTR_MAP_METHOD, (OCIError *) errhp));

/* get order method list*/
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &order_method, (ub4 *) 0,
                          (ub4) OCI_ATTR_ORDER_METHOD, (OCIError *) errhp));

```



```

SPACING;
printf ( "TYPE      : Attributes : \n");
SPACING;
printf ( "Schema:           %s\n", schema);
SPACING;
printf ( "Typecode:         %d\n", typecode);
if (typecode == OCI_TYPECODE_NAMEDCOLLECTION)
{
    SPACING;
    printf ( "Collection typecode: %d\n", collection_typecode);
}
SPACING;
printf ( "Version:           %s\n", version);
SPACING;
printf ( "Number of attrs:    %d\n", num_attr);
SPACING;
printf ( "Number of methods: %d\n", num_method);
SPACING;
printf ( "Is incomplete:      %d\n", is_incomplete);
SPACING;
printf ( "Is system:         %d\n", is_system);
SPACING;
printf ( "Is predefined:      %d\n", is_predefined);
SPACING;
printf ( "Is sys-gen:         %d\n", is_sysgen);
SPACING;
printf ( "Is transient:       %d\n", is_transient);
SPACING;
printf ( "Has nested table:    %d\n", has_table);
SPACING;
printf ( "Has LOB:            %d\n", has_lob);
SPACING;
printf ( "Has file:           %d\n", has_file);
printf("\n");

if (num_attr > 0)
    chk_column(envhp, errhp, svchp, list_attr, num_attr);
else if (typecode == OCI_TYPECODE_NAMEDCOLLECTION)
    chk_collection(envhp, errhp, svchp, collection_dschnp,
                  collection_typecode == OCI_TYPECODE_VARRAY);
if (map_method != (dvoid *)0)
    chk_method(envhp, errhp, svchp, map_method, "TYPE MAP
METHOD\n-----");
if (order_method != (dvoid *)0)

```

```

        chk_method(envhp, errhp, svchp, order_method, "TYPE ORDER
METHOD\n-----");
        if (num_method > 0)
            chk_methodlst(envhp, errhp, svchp, list_method, num_method, "TYPE
METHOD\n-----");
    }

/*****/
int main(int argc, char *argv[])
{
    OCIEnv *envhp = (OCIEnv *) 0;
    OCIServer *srvhp = (OCIServer *) 0;
    OCIErr *errhp = (OCIErr *) 0;
    OCISvcCtx *svchp = (OCISvcCtx *) 0;
    OCISession *usrhp = (OCISession *) 0;
    dvoid *tmp;
    int i;

    tab = 0;

    if (argc < 4)
    {
        (void) printf("Usage -- cdemort <username> <password> <upper case
typename>\n");
        return (0);
    }

    (void) OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT,
                        (dvoid *)0, (dvoid * (*)()) 0,
                        (dvoid * (*)()) 0, (void (*)()) 0 );

    (void) OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp,
                          (ub4) OCI_HTYPE_ENV,
                          52, (dvoid **) &tmp);

    (void) OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp,
                          (ub4) OCI_HTYPE_ERROR,
                          52, (dvoid **) &tmp);

    (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp,
                          (ub4) OCI_HTYPE_SERVER,
                          52, (dvoid **) &tmp);

```

```

checkerr(errhp, OCIServerAttach( srvhp, errhp, (text *) "",
                                (sb4) strlen(""), (ub4) OCI_DEFAULT));

checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp,
                                (ub4) OCI_HTYPE_SVCCTX,
                                52, (dvoid **) &tmp));

checkerr(errhp, OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                            (dvoid *) srvhp, (ub4) 0,
                            (ub4) OCI_ATTR_SERVER, (OCIError *) errhp));

checkerr(errhp, OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp,
                                (ub4)OCI_HTYPE_SESSION, 0, (dvoid **)0));

checkerr(errhp, OCIAttrSet((dvoid *) usrhp, (ub4)OCI_HTYPE_SESSION,
                            (dvoid *)argv[1], (ub4)strlen(argv[1]),
                            (ub4)OCI_ATTR_USERNAME, errhp));

checkerr(errhp, OCIAttrSet((dvoid *) usrhp, (ub4)OCI_HTYPE_SESSION,
                            (dvoid *)argv[2], (ub4)strlen(argv[2]),
                            (ub4)OCI_ATTR_PASSWORD, errhp));

checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp,
                                OCI_CRED_RDBMS, OCI_DEFAULT));

checkerr(errhp, OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
                            (dvoid *)usrhp, (ub4)0,
                            (ub4)OCI_ATTR_SESSION, errhp));

/* dump an object with all the types */
SPACING;
(void) printf("%s\n", argv[3]);
tst_desc_type(envhp, errhp, svchp, argv[3]);
printf("\n");

checkerr(errhp, OCISessionEnd (svchp, errhp, usrhp, OCI_DEFAULT));

(void) OCIServerDetach( srvhp, errhp, (ub4) OCI_DEFAULT );

checkerr(errhp, OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX));
checkerr(errhp, OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR));
checkerr(errhp, OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER));

return (0);

```

```

}

/* end of file cdemodsc.c */

```

cdemodsc.h

```

/*  NAME
    cdemodsc.h - header file for cdemodsc.c
*/

/*-----*/
#ifndef CDEMODSC
#define CDEMODSC

#ifndef S
#include <s.h>
#endif

#ifndef HSTDEF
#include <hstdef.h>
#endif

#ifndef SQLDEF
#include <sqldef.h>
#endif

#ifndef OCIDEF
#include <ocidef.h>
#endif
#ifndef OCI_ORACLE
#include <oci.h>
#endif

#include <lnx.h>

/*-----*/
/*
** #define
*/

#define MAXNAME      30
#define MAXOBJLEN    60
#define MAXOBS      7
#define NPOS         40

```

```

#define SPACING      for (glindex = 0; glindex < tab; glindex++)\
                    printf(" ")

/*-----*/
/*
** Prototypes for functions in cdemodsc.c
*/
static void chk_column(/*_ OCIEEnv *envhp, OCIEError *errhp, OCISvcCtx *svchp,
dvoid *dschp, ub4 parmcnt _*/);
static void chk_method(/*_ OCIEEnv *envhp, OCIEError *errhp, OCISvcCtx *svchp,
dvoid *dschp, const text *comment _*/);
static void chk_methodlst(/*_ OCIEEnv *envhp, OCIEError *errhp, OCISvcCtx *svchp,
dvoid *dschp, ub4 count, const text *comment _*/);
static void chk_arglst(/*_ OCIEEnv *envhp, OCIEError *errhp, OCISvcCtx *svchp,
dvoid *dschp _*/);
static void chk_arg(/*_ OCIEEnv *envhp, OCIEError *errhp, OCISvcCtx *svchp, dvoid
*dschp, ub1 type, ub4 start, ub4 end _*/);
static void chk_collection (/*_ OCIEEnv *envhp, OCIEError *errhp, OCISvcCtx
*svchp, dvoid *dschp, sword is_array _*/);
static void tst_desc_type(/*_ OCIEEnv *envhp, OCIEError *errhp, OCISvcCtx *svchp,
text *objname _*/);
static void checkerr(/*_ OCIEError *errhp, sword status _*/);

/* Prototype for main function */
int main(/*_ int argc, char *argv[] _*/);
int tab;
int glindex;

#endif /* CDEMOSC */

```

Example 5, CLOB/BLOB Operations

```

/*  NAME
    cdemolb2.c - Demonstrates writing and reading of CLOB/BLOB columns
                  with stream mode and with callback functions.

DESCRIPTION
    This program takes 2 input files (the first a text file and the
    second a binary file) and stores the files into CLOB, BLOB columns.

    On output, the program reads the newly populated CLOB/BLOB columns
    and writes them to the output files (txtfile1.log, binfile1.log,
    txtfile2.log, binfile2.log), where

    txtfile1.log -- created for stream reading CLOB contents to it
    binfile1.log -- created for stream reading BLOB contents to it

    txtfile2.log -- created for callback reading CLOB contents to it
    binfile2.log -- created for callback reading BLOB contents to it

    Sample usage: cdemolb2 cdemolb.dat giffile.dat

    cdemolb.dat  -- a text file in the demo directory
    giffile.dat  -- a gif file in the demo directory

    After successful execution of the program, the files, cdemolb.dat,
    txtfile1.log, and txtfile2.log should be identical.  giffile.dat,
    binfile1.log, and binfile2.log should be identical.

*/

#include <stdio.h>
#include <oci.h>

static sb4 init_handles(/*_ void _*/);
static sb4 log_on(/*_ void _*/);
static sb4 setup_table(/*_ void _*/);
static sb4 select_locator(/*_ int rowind _*/);
static ub4 file_length(/*_ FILE *fp _*/);
static sb4 test_file_to_lob(/*_ int rowind, char *tfname, char *bfname _*/);
static void test_lob_to_file(/*_ int rowind _*/);
static void stream_write_lob(/*_ int rowind, OCILobLocator *lobl,
                             FILE *fp, ub4 filelen _*/);

```

```

static void callback_write_lob(/*_ int rowind, OCILobLocator *lobl,
                               FILE *fp, ub4 filelen _*/);
static void stream_read_lob(/*_ int rowind, OCILobLocator *lobl, FILE *fp _*/);
static void callback_read_lob(/*_ int rowind, OCILobLocator *lobl, FILE *fp _*/);
static sb4 cbk_fill_buffer(/*_ dvoid *ctxp, dvoid *bufxp, ub4 *lenp,
                           ub1 *piece _*/);
static sb4 cbk_write_buffer(/*_ dvoid *ctxp, CONST dvoid *bufxp, ub4 lenp,
                           ub1 piece _*/);

static void logout(/*_ void _*/);
static void drop_table(/*_ void _*/);
static void report_error(/*_ void _*/);

int main(/*_ int argc, char *argv[] _*/);

#define TRUE      1
#define FALSE     0

#define MAXBUFLen 5000

static OCIEnv      *envhp;
static OCIServer   *srvhp;
static OCISvcCtx   *svchp;
static OCIError    *errhp;
static OCISession  *authp;
static OCISmt      *stmthp;
static OCILobLocator *clob, *blob;
static OCIDefine    *defnp1 = (OCIDefine *) 0, *defnp2 = (OCIDefine *) 0;
static OCIBind      *bndhp = (OCIBind *) 0;

static FILE *fp1, *fp2;

static ub4  txtfilelen = 0;
static ub4  binfilelen = 0;

static boolean istxtfile;
static boolean tab_exists = FALSE;

/*-----end of Inclusions-----*/

int main(argc, argv)
int argc;
char *argv[];
{
    int rowind;

```

```

if (argc != 3)
{
    (void) printf("Usage: %s txtfilename binfilename\n", argv[0]);
    return 0;
}

if (init_handles())
{
    (void) printf("FAILED: init_handles()\n");
    return OCI_ERROR;
}

if (log_on())
{
    (void) printf("FAILED: log_on()\n");
    return OCI_ERROR;
}

if (setup_table())
{
    (void) printf("FAILED: setup_table()\n");
    logout();
    return OCI_ERROR;
}

tab_exists = TRUE;

for (rowind = 1; rowind <= 2; rowind++)
{
    if (select_locator(rowind))
    {
        (void) printf("FAILED: select_locator()\n");
        logout();
        return OCI_ERROR;
    }

    if (test_file_to_lob(rowind, argv[1], argv[2]))
    {
        (void) printf("FAILED: load files to lobs\n");
        logout();
        return OCI_ERROR;
    }

    test_lob_to_file(rowind);
}

```



```

    }

    logout();

    return OCI_SUCCESS;
}

/* ----- */
/* initialize environment, allocate handles, etc. */
/* ----- */

sb4 init_handles()
{
    if (OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                     (dvoid * (*)(dvoid *, size_t)) 0,
                     (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                     (void (*)(dvoid *, dvoid *)) 0 ))
    {
        (void) printf("FAILED: OCIInitialize()\n");
        return OCI_ERROR;
    }

    /* initialize environment handle */
    if (OCIEnvInit((OCIEnv **) &envhp, (ub4) OCI_DEFAULT,
                  (size_t) 0, (dvoid **) 0 ))
    {
        (void) printf("FAILED: OCIEnvInit()\n");
        return OCI_ERROR;
    }

    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp,
                      (ub4) OCI_HTYPE_SVCCTX, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }

    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp,
                      (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }

    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &stmthp,

```

```

        (ub4) OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }

    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp,
        (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }

    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &authp,
        (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }

    if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &clob,
        (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIDescriptorAlloc()\n");
        return OCI_ERROR;
    }

    /* allocate the lob locator variables */
    if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &blob,
        (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIDescriptorAlloc()\n");
        return OCI_ERROR;
    }

    return OCI_SUCCESS;
}

/* ----- */
/* attach to the server and log on as SCOTT/TIGER */
/* ----- */

sb4 log_on()
{
    text *uid = (text *)"SCOTT";

```

```

text *pwd = (text *)"TIGER";
text *cstring = (text *) "";

/* attach to the server */
if (OCIServerAttach(srvhp, errhp, (text *) cstring,
                    (sb4) strlen((char *)cstring), (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: OCIServerAttach()\n");
    return OCI_ERROR;
}

if (OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
               (dvoid *) uid, (ub4) strlen((char *)uid),
               (ub4) OCI_ATTR_USERNAME, errhp))
{
    (void) printf("FAILED: OCIAttrSet()\n");
    return OCI_ERROR;
}

if (OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
               (dvoid *) pwd, (ub4) strlen((char *)pwd),
               (ub4) OCI_ATTR_PASSWORD, errhp))
{
    (void) printf("FAILED: OCIAttrSet()\n");
    return OCI_ERROR;
}

/* set the server attribute in the service context */
if (OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
               (dvoid *) srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER, errhp))
{
    (void) printf("FAILED: OCIAttrSet()\n");
    return OCI_ERROR;
}

/* log on */
if (OCISessionBegin(svchp, errhp, authp, (ub4) OCI_CRED_RDBMS,
                    (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: OCISessionBegin()\n");
    return OCI_ERROR;
}

/* set the session attribute in the service context */
if (OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) authp,

```

```

        (ub4) 0, (ub4) OCI_ATTR_SESSION, errhp))
    {
        (void) printf("FAILED: OCIAttrSet()\n");
        return OCI_ERROR;
    }

    return OCI_SUCCESS;
}

/* ----- */
/* Create table FOO with CLOB, BLOB columns and insert one row.      */
/* Both columns are empty lobes, not null lobes.                    */
/* ----- */

sb4 setup_table()
{
    int colc;

    text *crtstmt = (text *) "CREATE TABLE FOO (A CLOB, B BLOB, C INTEGER)";
    text *insstmt =
        (text *) "INSERT INTO FOO VALUES (EMPTY_CLOB(), EMPTY_BLOB(), :1)";

    if (OCISstmtPrepare(stmthp, errhp, crtstmt, (ub4) strlen((char *) crtstmt),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtPrepare() crtstmt\n");
        return OCI_ERROR;
    }

    if (OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (CONST OCISnapshot *) 0, (OCISnapshot *) 0,
        (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtExecute() crtstmt\n");
        return OCI_ERROR;
    }

    if (OCISstmtPrepare(stmthp, errhp, insstmt, (ub4) strlen((char *) insstmt),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtPrepare() insstmt\n");
        return OCI_ERROR;
    }
}

```

```

if (OCIBindByPos(stmthp, &bndhp, errhp, (ub4) 1,
                (dvoid *) &colc, (sb4) sizeof(colc), SQLT_INT,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: OCIBindByPos()\n");
    return OCI_ERROR;
}

for (colc = 1; colc <= 2; colc++)
{
    if (OCISTmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                      (CONST OCISnapshot *) 0, (OCISnapshot *) 0,
                      (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISTmtExecute() insstmt\n");
        return OCI_ERROR;
    }
}

(void) OCITransCommit(svchp, errhp, (ub4)0);

return OCI_SUCCESS;
}

/*-----*/
/* Select lob locators from the CLOB, BLOB columns. */
/* We need the 'FOR UPDATE' clause since we need to write to the lob. */
/*-----*/

sb4 select_locator(int rowind)
{
    int colc = rowind;
    text *sqlstmt = (text *)"SELECT A, B FROM FOO WHERE C = :1 FOR UPDATE";

    if (OCISTmtPrepare(stmthp, errhp, sqlstmt, (ub4) strlen((char *)sqlstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISTmtPrepare() sqlstmt\n");
        return OCI_ERROR;
    }

    if (OCIBindByPos(stmthp, &bndhp, errhp, (ub4) 1,
                    (dvoid *) &colc, (sb4) sizeof(colc), SQLT_INT,
                    (dvoid *) 0, (ub2 *)0, (ub2 *)0,

```

```

        (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIBindByPos()\n");
        return OCI_ERROR;
    }

    if (OCIDefineByPos(stmthp, &defnp1, errhp, (ub4) 1,
        (dvoid *) &clob, (sb4) -1, (ub2) SQLT_CLOB,
        (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT)
    || OCIDefineByPos(stmthp, &defnp2, errhp, (ub4) 2,
        (dvoid *) &blob, (sb4) -1, (ub2) SQLT_BLOB,
        (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIDefineByPos()\n");
        return OCI_ERROR;
    }

    /* execute the select and fetch one row */
    if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
        (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtExecute() sqlstmt\n");
        return OCI_ERROR;
    }

    return OCI_SUCCESS;
}

/* ----- */
/* Read operating system files into local buffers and then write the */
/* buffers to lob. */
/* ----- */

sb4 test_file_to_lob(int rowind, char *txtfile, char *binfile)
{
    (void) printf("\n===> Testing loading files into lob. ....\n\n");

    fp1 = fopen((const char *)txtfile, (const char *) "r");
    fp2 = fopen((const char *)binfile, (const char *) "rb");

    if ( !(fp1 && fp2))
    {
        (void) printf("ERROR: Failed to open file(s).\n");
        return -1;
    }
}

```

```

    }

    txtfilelen = file_length(fp1);
    binfilelen = file_length(fp2);

    switch (rowind)
    {
        case 1:
            stream_write_lob(rowind, clob, fp1, txtfilelen);
            stream_write_lob(rowind, blob, fp2, binfilelen);
            break;
        case 2:
            istxtfile = TRUE;
            callback_write_lob(rowind, clob, fp1, txtfilelen);
            istxtfile = FALSE;
            callback_write_lob(rowind, blob, fp2, binfilelen);
            break;
        default:
            (void) printf("ERROR: Invalid row indicator.\n");
            break;
    }

    (void) fclose(fp1);
    (void) fclose(fp2);

    return 0;
}

/* ----- */
/* get the length of the input file.                */
/* ----- */

ub4 file_length(FILE *fp)
{
    fseek(fp, 0, SEEK_END);
    return (ub4) (ftell(fp));
}

/* ----- */
/* Read operating system files into local buffers and then write the */
/* buffers to lobs using stream mode.                                */
/* ----- */

void stream_write_lob(int rowind, OCILobLocator *lobl, FILE *fp, ub4 filelen)
{

```

```

ub4   offset = 1;
ub4   loblen = 0;
ub1   bufp[MAXBUFLen];
ub4   amtp = filelen;
ub1   piece;
sword retval;
int   readval;
ub4   len = 0;
ub4   nbytes;
ub4   remainder = filelen;

(void) printf("--> To do streamed write lob, amount = %d\n", filelen);

(void) OCILobGetLength(svchp, errhp, lobl, &loblen);
(void) printf("Before stream write, LOB length = %d\n\n", loblen);

(void) fseek(fp, 0, 0);

if (filelen > MAXBUFLen)
    nbytes = MAXBUFLen;
else
    nbytes = filelen;

if (fread((void *)bufp, (size_t)nbytes, 1, fp) != 1)
{
    (void) printf("ERROR: read file.\n");
    return;
}

remainder -= nbytes;

if (remainder == 0)          /* exactly one piece in the file */
{
    (void) printf("Only one piece, no need for stream write.\n");
    if (retval = OCILobWrite(svchp, errhp, lobl, &amtp, offset, (dvoid *) bufp,
                            (ub4) nbytes, OCI_ONE_PIECE, (dvoid *) 0,
                            (sb4 *) (dvoid *, dvoid *, ub4 *, ub1 *)) 0,
        (ub2) 0, (ub1) SQLCS_IMPLICIT) != OCI_SUCCESS)
    {
        (void) printf("ERROR: OCILobWrite(), retval = %d\n", retval);
        return;
    }
}
else                          /* more than one piece */
{

```



```

if (OCILobWrite(svchp, errhp, lobl, &amp;tp, offset, (dvoid *) bufp,
               (ub4) MAXBUFLen, OCI_FIRST_PIECE, (dvoid *)0,
               (sb4 *) (dvoid *, dvoid *, ub4 *, ub1 *)) 0,
    (ub2) 0, (ub1) SQLCS_IMPLICIT) != OCI_NEED_DATA)
{
    (void) printf("ERROR: OCILobWrite().\n");
    return;
}

piece = OCI_NEXT_PIECE;

do
{
    if (remainder > MAXBUFLen)
        nbytes = MAXBUFLen;
    else
    {
        nbytes = remainder;
        piece = OCI_LAST_PIECE;
    }

    if (fread((void *)bufp, (size_t)nbytes, 1, fp) != 1)
    {
        (void) printf("ERROR: read file.\n");
        piece = OCI_LAST_PIECE;
    }

    retval = OCILobWrite(svchp, errhp, lobl, &amp;tp, offset, (dvoid *) bufp,
                        (ub4) nbytes, piece, (dvoid *)0,
                        (sb4 *) (dvoid *, dvoid *, ub4 *, ub1 *)) 0,
                        (ub2) 0, (ub1) SQLCS_IMPLICIT);

    remainder -= nbytes;

} while (retval == OCI_NEED_DATA && !feof(fp));
}

if (retval != OCI_SUCCESS)
{
    (void) printf("Error: stream writing LOB.\n");
    return;
}

(void) OCILobGetLength(svchp, errhp, lobl, &loblen);
(void) printf("After stream write, LOB length = %d\n\n", loblen);

```

```

    return;
}

/* ----- */
/* Read operating system files into local buffers and then write the */
/* buffers to lobs using callback function.                          */
/* ----- */

void callback_write_lob(int rowind, OCILobLocator *lobl, FILE *fp, ub4 filelen)
{
    ub4    offset = 1;
    ub4    loblen = 0;
    ub1    bufp[MAXBUFLLEN];
    ub4    amtp = filelen;
    ub4    nbytes;
    sword  retval;

    (void) printf("--> To do callback write lob, amount = %d\n", filelen);

    (void) OCILobGetLength(svchp, errhp, lobl, &loblen);
    (void) printf("Before callback write, LOB length = %d\n\n", loblen);

    (void) fseek(fp, 0, 0);

    if (filelen > MAXBUFLLEN)
        nbytes = MAXBUFLLEN;
    else
        nbytes = filelen;

    if (fread((void *)bufp, (size_t)nbytes, 1, fp) != 1)
    {
        (void) printf("ERROR: read file.\n");
        return;
    }

    if (filelen < MAXBUFLLEN)        /* exactly one piece in the file */
    {
        (void) printf("Only one piece, no need for callback write.\n");
        if (retval = OCILobWrite(svchp, errhp, lobl, &amtp, offset, (dvoid *) bufp,
                                (ub4) nbytes, OCI_ONE_PIECE, (dvoid *) 0,
                                (sb4 *) (dvoid *, dvoid *, ub4 *, ub1 *)) 0,
            (ub2) 0, (ub1) SQLCS_IMPLICIT) != OCI_SUCCESS)
        {
            (void) printf("ERROR: OCILobWrite().\n");
        }
    }
}

```

```

        return;
    }
}
else /* more than one piece */
{
    if (retval = OCILobWrite(svchp, errhp, lobl, &amtp, offset, (dvoid *)bufp,
                            (ub4)nbytes, OCI_FIRST_PIECE, (dvoid *)0,
                            cbk_fill_buffer, (ub2) 0, (ub1) SQLCS_IMPLICIT))
    {
        (void) printf("ERROR: OCILobWrite().\n");
        report_error();
        return;
    }
}

(void) OCILobGetLength(svchp, errhp, lobl, &loblen);
(void) printf("After callback write, LOB length = %d\n\n", loblen);

return;
}

/* ----- */
/* callback function to read the file into buffer. */
/* ----- */

sb4 cbk_fill_buffer(ctxp, bufxp, lenp, piece)
dvoid *ctxp;
dvoid *bufxp;
ub4 *lenp;
ub1 *piece;
{
    FILE *fp = (istxtfile ? fp1 : fp2);
    ub4 filelen = (istxtfile ? txtfilelen : binfilelen);
    ub4 nbytes;
    static ub4 len = MAXBUFLen; /* because 1st piece has been written */

    if ((filelen - len) > MAXBUFLen)
        nbytes = MAXBUFLen;
    else
        nbytes = filelen - len;

    *lenp = nbytes;

    if (fread((void *)bufxp, (size_t)nbytes, 1, fp) != 1)

```

```

{
    (void) printf("ERROR: read file. Abort callback fill buffer\n");
    *piece = OCI_LAST_PIECE;
    len = MAXBUFLLEN;          /* reset it for the next callback_write_lob() */
    return OCI_CONTINUE;
}

len += nbytes;

if (len == filelen)           /* entire file has been read */
{
    *piece = OCI_LAST_PIECE;
    len = MAXBUFLLEN;          /* reset it for the next callback_write_lob() */
}
else
    *piece = OCI_NEXT_PIECE;

return OCI_CONTINUE;
}

/* ----- */
/* Read lob into local buffers and then write them to operating */
/* system files. */
/* ----- */

void test_lob_to_file(int rowind)
{
    ub4    offset = 1;
    ub4    loblen = 0;
    ub1    bufp[MAXBUFLLEN];
    ub4    amtp = MAXBUFLLEN;
    text    txtfilename[20], binfilename[20];

    (void) sprintf((char *) txtfilename, (char *)"txtfile%d.log", rowind);
    (void) sprintf((char *) binfilename, (char *)"binfile%d.log", rowind);

    (void) printf("\n===> Testing writing lob to files ..... \n\n");

    fp1 = fopen((char *)txtfilename, (const char *) "w");
    fp2 = fopen((char *)binfilename, (const char *) "wb");

    if ( !(fp1 && fp2))
    {
        (void) printf("ERROR: Failed to open file(s).\n");
        return;
    }
}

```

```

    }

    switch (rowind)
    {
        case 1:
            stream_read_lob(rowind, clob, fp1);
            stream_read_lob(rowind, blob, fp2);
            break;
        case 2:
            istxtfile = TRUE;
            callback_read_lob(rowind, clob, fp1);

            istxtfile = FALSE;
            callback_read_lob(rowind, blob, fp2);
            break;
        default:
            (void) printf("ERROR: Invalid row indicator.\n");
            break;
    }

    (void) fclose(fp1);
    (void) fclose(fp2);

    return;
}

/* ----- */
/* Read lobes using stream mode into local buffers and then write */
/* them to operating system files. */
/* ----- */

void stream_read_lob(int rowind, OCILobLocator *lobl, FILE *fp)
{
    ub4    offset = 1;
    ub4    loblen = 0;
    ub1    bufp[MAXBUFLen];
    ub4    amtp = 4096000000;
    sword  retval;
    ub4    piece = 0;
    ub4    remainder;          /* the number of bytes for the last piece */

    (void) OCILobGetLength(svchp, errhp, lobl, &loblen);
    /*amtp = loblen;*/

    (void) printf("--> To stream read LOB, loblen = %d.\n", loblen);

```

```

memset(bufp, '\0', MAXBUFLen);

retval = OCILobRead(svchp, errhp, lobl, &amtp, offset, (dvoid *) bufp,
                  (loblen < MAXBUFLen ? loblen : MAXBUFLen), (dvoid *)0,
                  (sb4 (*)(dvoid *, const dvoid *, ub4, ub1)) 0,
                  (ub2) 0, (ub1) SQLCS_IMPLICIT);

(void) printf(" amtp passed in was 4gb, amtp returned is %u\n", amtp);

switch (retval)
{
    case OCI_SUCCESS:                /* only one piece */
        (void) printf("stream read %d th piece\n", ++piece);
        (void) fwrite((void *)bufp, (size_t)loblen, 1, fp);
        break;
    case OCI_ERROR:
        report_error();
        break;
    case OCI_NEED_DATA:              /* there are 2 or more pieces */

        remainder = loblen;

        (void) fwrite((void *)bufp, MAXBUFLen, 1, fp); /* full buffer to write */

        do
        {
            memset(bufp, '\0', MAXBUFLen);
            /*amtp = 0;*/

            remainder -= MAXBUFLen;

            retval = OCILobRead(svchp, errhp, lobl, &amtp, offset, (dvoid *) bufp,
                              (ub4) MAXBUFLen, (dvoid *)0,
                              (sb4 (*)(dvoid *, const dvoid *, ub4, ub1)) 0,
                              (ub2) 0, (ub1) SQLCS_IMPLICIT);

            /* the amount read returned is undefined for FIRST, NEXT pieces */
            (void) printf("stream read %d th piece, amtp = %u\n", ++piece, amtp);

            if (remainder < MAXBUFLen)      /* last piece not a full buffer piece */
                (void) fwrite((void *)bufp, (size_t)remainder, 1, fp);
            else
                (void) fwrite((void *)bufp, MAXBUFLen, 1, fp);
        } while (retval == OCI_NEED_DATA);
}

```

```

        } while (retval == OCI_NEED_DATA);
        break;
    default:
        (void) printf("Unexpected ERROR: OCILobRead() LOB.\n");
        break;
    }
    return;
}

/* ----- */
/* Read lob using callback function into local buffers and */
/* then write them to operating system files. */
/* ----- */

void callback_read_lob(int rowind, OCILobLocator *lobl, FILE *fp)
{
    ub4    offset = 1;
    ub4    loblen = 0;
    ub1    bufp[MAXBUFLen];
    ub4    amtp = 4096000000;
    sword retval;

    (void) OCILobGetLength(svchp, errhp, lobl, &loblen);

    (void) printf("--> To callback read LOB, loblen = %d.\n", loblen);

    if (retval = OCILobRead(svchp, errhp, lobl, &amtp, offset, (dvoid *) bufp,
                           (ub4) MAXBUFLen, (dvoid *) bufp, cbk_write_buffer,
                           (ub2) 0, (ub1) SQLCS_IMPLICIT))
    {
        (void) printf("ERROR: OCILobRead() LOB.\n");
        report_error();
    }
    return;
}

/* ----- */
/* callback function to write buffer to the file. */
/* ----- */

sb4 cbk_write_buffer(ctxp, bufxp, lenp, piece)
    dvoid *ctxp;
    CONST dvoid *bufxp;
    ub4 lenp;
    ub1 piece;

```

```

{
    static ub4 piece_count = 0;
    FILE *fp = (istxtfile ? fp1 : fp2);

    piece_count++;

    switch (piece)
    {
        case OCI_LAST_PIECE:
            (void) fwrite((void *)bufxp, (size_t)lenp, 1, fp);
            (void) printf("callback read the %d th piece\n\n", piece_count);
            piece_count = 0;
            return OCI_CONTINUE;

        case OCI_FIRST_PIECE:
        case OCI_NEXT_PIECE:
            (void) fwrite((void *)bufxp, (size_t)lenp, 1, fp);
            break;
        default:
            (void) printf("callback read error: unkown piece = %d.\n", piece);
            return OCI_ERROR;
    }

    (void) printf("callback read the %d th piece\n", piece_count);

    return OCI_CONTINUE;
}

/*-----*/
/* Drop table FOO before logging off from the server.      */
/*-----*/

void drop_table()
{
    text *sqlstmt = (text *) "DROP TABLE FOO";

    if (OCISmtPrepare(stmthp, errhp, sqlstmt, (ub4) strlen((char *) sqlstmt),
                     (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISmtPrepare() sqlstmt\n");
        return;
    }

    if (OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                     (CONST OCISnapshot *) 0, (OCISnapshot *) 0,

```



```

        (ub4) OCI_DEFAULT))
    (void) printf("FAILED: OCISstmtExecute() sqlstmt\n");
    return;
}

/*-----*/
/* Logoff and disconnect from the server. Free handles.          */
/*-----*/

void logout()
{
    if (tab_exists)
        drop_table();

    (void) OCISessionEnd(svchp, errhp, authp, (ub4) 0);
    (void) OCIServerDetach(srvhp, errhp, (ub4) OCI_DEFAULT);

    (void) printf("Logged off and detached from server.\n");

    (void) OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER);
    (void) OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX);
    (void) OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR);
    (void) OCIHandleFree((dvoid *) authp, (ub4) OCI_HTYPE_SESSION);
    (void) OCIDescriptorFree((dvoid *) clob, (ub4) OCI_DTYPE_LOB);
    (void) OCIDescriptorFree((dvoid *) blob, (ub4) OCI_DTYPE_LOB);
    (void) OCIHandleFree((dvoid *) stmthp, (ub4) OCI_HTYPE_STMT);

    (void) printf("All handles freed\n");
    return;
}

/* ----- */
/* retrieve error message and print it out.                */
/* ----- */
void report_error()
{
    text msgbuf[512];
    sb4 errcode = 0;

    (void) OCIErrorGet((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                      msgbuf, (ub4) sizeof(msgbuf), (ub4) OCI_HTYPE_ERROR);
    (void) printf("ERROR CODE = %d\n", errcode);
    (void) printf("%.5s\n", 512, msgbuf);
    return;
}

```

Example 6, LOB Buffering

```
/*  NAME
    cdemolbs.c - Demonstrates reading and writing to LOBs through
                  the LOB Buffering Subsystem.

    DESCRIPTION
    This program reads from an input binary/text file, writing into an
    initialized B/CLOB column in buffered mode. It then reads in buffered
    mode from the B/CLOB column and populates an output file. After building
    the executable (assume it is called cdemolbs), run program as follows:
        cdemolbs src.txt src.bin dst.txt dst.bin
    where src.txt and src.bin are text and binary files of size <= 512Kbytes.
    IMPORTANT: . This program works only for single-byte CLOBs.
               . Before running this program, ensure that the database is
                 started up and a table FOO does not exist in the SCOTT/
                 TIGER sample account.
*/

#include <stdio.h>
#include <string.h>
#include <oci.h>

/----- Public Constants and Variables -----*/

/* Constants */
#define TRUE      1
#define FALSE     0
#define MAXBUFLN 32768
#define MAXLBSLEN 524288

/* OCI Handles */
static OCIEnv      *envhp;
static OCIServer   *srvhp;
static OCISvcCtx    *svchp;
static OCIErr      *errhp;
static OCISession   *authp;
static OCISmt       *stmthp;
static OCILobLocator *clob, *blob;
static OCIDefine    *defnp1 = (OCIDefine *) 0, *defnp2 = (OCIDefine *) 0;
static OCIBind      *bndhp = (OCIBind *) 0;

/* Miscellaneous */
static FILE          *fp1, *fp2;
```

```

static ub4      txtfilelen = 0;
static ub4      binfilelen = 0;
static boolean  istxtfile;
static boolean  tab_exists = FALSE;

/*----- Public functions - Specification -----*/

int main        (/*_ int argc, char *argv[] _*/);

static sb4 init_handles  (/*_ void _*/);
static sb4 init_table    (/*_ void _*/);
static sb4 log_on        (/*_ void _*/);
static void log_off      (/*_ void _*/);
static sb4 write_lobs    (/*_ int rowind, char *txtfile, char *binfile _*/);
static sb4 read_lobs     (/*_ int rowind, char *txtfile, char *binfile _*/);

/*----- Private functions - Specification -----*/

static sb4 select_clob   (/*_ int rowind _*/);
static sb4 select_blob   (/*_ int rowind _*/);
static sb4 select_lobs   (/*_ int rowind _*/);
static sb4 buf_write_lob (/*_ int rowind, OCILobLocator *locator, FILE *fp,
                          ub4 filelen _*/);
static sb4 buf_read_lob  (/*_ int rowind, OCILobLocator *locator,
                          FILE *fp _*/);
static void drop_table   (/*_ void _*/);
static void report_error (/*_ void _*/);
static ub4 file_length   (/*_ FILE *fp _*/);

/*----- Public functions -----*/

/*----- main -----*/

/* main driver */
int main(argc, argv)
int argc;
char *argv[];
{
    int rowind;

    /* validate input arguments */
    if (argc != 5)
    {
        (void) printf("Usage: %s srctxtfile srcbinfile desttxtfile destbinfile\n",
                      argv[0]);
    }

```

```

        return 0;
    }
    /* initialize OCI handles */
    if (init_handles())
    {
        (void) printf("FAILED: init_handles()\n");
        return OCI_ERROR;
    }
    /* log on to server */
    if (log_on())
    {
        (void) printf("FAILED: log_on()\n");
        return OCI_ERROR;
    }
    /* init demo table */
    if (init_table())
    {
        (void) printf("FAILED: init_table()\n");
        log_off();
        return OCI_ERROR;
    }
    /* write to LOBs in row 1 through the buffering subsystem,
                                   reading from src files */
    rowind = 1;
    if (write_lobs(rowind, argv[1], argv[2]))
    {
        (void) printf("FAILED: write files to lob\n");
        log_off();
        return OCI_ERROR;
    }
    /* read from LOBs in row 1 through buffering subsystem,
                                   writing to dest files */
    rowind = 1;
    if (read_lobs(rowind, argv[3], argv[4]))
    {
        (void) printf("FAILED: write lob to files\n");
        log_off();
        return OCI_ERROR;
    }
    /* clean up and log off from server */
    log_off();

    return OCI_SUCCESS;
}

```

```

/*----- init_handles -----*/

/* initialize environment, and allocate all handles */
sb4 init_handles()
{
    if (OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                     (dvoid * (*)(dvoid *, size_t)) 0,
                     (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                     (void (*)(dvoid *, dvoid *)) 0 ))
    {
        (void) printf("FAILED: OCIInitialize()\n");
        return OCI_ERROR;
    }
    /* initialize environment handle */
    if (OCIEnvInit((OCIEnv **) &envhp, (ub4) OCI_DEFAULT,
                  (size_t) 0, (dvoid **) 0 ))
    {
        (void) printf("FAILED: OCIEnvInit()\n");
        return OCI_ERROR;
    }
    /* initialize service context */
    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp,
                      (ub4) OCI_HTYPE_SVCCTX, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }
    /* initialize error handle */
    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp,
                      (ub4) OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }
    /* initialize statement handle */
    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &stmthp,
                      (ub4) OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }
    /* initialize server handle */
    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp,
                      (ub4) OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0))
    {

```

```

        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }
    /* initialize session/authentication handle */
    if (OCIHandleAlloc((dvoid *) envhp, (dvoid **) &authp,
                      (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIHandleAlloc()\n");
        return OCI_ERROR;
    }

    /* allocate the lob locator variables */
    if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &clob,
                          (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIDescriptorAlloc()\n");
        return OCI_ERROR;
    }
    if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &blob,
                          (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    {
        (void) printf("FAILED: OCIDescriptorAlloc()\n");
        return OCI_ERROR;
    }

    return OCI_SUCCESS;
}

/*----- init_table -----*/

/* create table FOO with initialized CLOB, BLOB columns, and insert two rows */
sb4 init_table()
{
    int   colc;
    text *crtstmt = (text *) "CREATE TABLE FOO (C1 CLOB, C2 BLOB, C3 INTEGER)";
    text *insstmt =
        (text *) "INSERT INTO FOO VALUES (EMPTY_CLOB(), EMPTY_BLOB(), :1)";

    /* prepare create statement */
    if (OCISTmtPrepare(stmthp, errhp, crtstmt, (ub4) strlen((char *) crtstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISTmtPrepare() crtstmt\n");
        return OCI_ERROR;
    }

```

```

    }
    /* execute create statement */
    if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                      (CONST OCISnapshot *) 0, (OCISnapshot *) 0,
                      (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtExecute() crtstmt\n");
        return OCI_ERROR;
    }
    /* prepare insert statement */
    if (OCIStmtPrepare(stmthp, errhp, insstmt, (ub4) strlen((char *) insstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtPrepare() insstmt\n");
        return OCI_ERROR;
    }
    /* associate variable colc with bind placeholder #1 in the SQL statement */
    if (OCIBindByPos(stmthp, &bndhp, errhp, (ub4) 1,
                    (dvoid *) &colc, (sb4) sizeof(colc), SQLT_INT,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIBindByPos()\n");
        return OCI_ERROR;
    }
    /* insert two rows */
    for (colc = 1; colc <= 2; colc++)
    {
        if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                          (CONST OCISnapshot *) 0, (OCISnapshot *) 0,
                          (ub4) OCI_DEFAULT))
        {
            (void) printf("FAILED: OCIStmtExecute() insstmt\n");
            return OCI_ERROR;
        }
    }

    /* commit the Xn */
    (void) OCITransCommit(svchp, errhp, (ub4) 0);

    /* set flag to be used by log_off() to drop the table */
    tab_exists = TRUE;

    return OCI_SUCCESS;
}

```

```

/*----- log_on -----*/

/* attach to the server and log on as SCOTT/TIGER */
sb4 log_on()
{
    text *uid      = (text *)"SCOTT";
    text *pwd      = (text *)"TIGER";
    text *cstring  = (text *)"inst1_alias";

    /* attach to the server */
    if (OCIServerAttach(srvhp, errhp, (text *) cstring,
                        (sb4) strlen((char *)cstring), (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIServerAttach()\n");
        return OCI_ERROR;
    }

    /* set username and password attributes of the server handle */
    if (OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                  (dvoid *) uid, (ub4) strlen((char *)uid),
                  (ub4) OCI_ATTR_USERNAME, errhp))
    {
        (void) printf("FAILED: OCIAttrSet()\n");
        return OCI_ERROR;
    }
    if (OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                  (dvoid *) pwd, (ub4) strlen((char *)pwd),
                  (ub4) OCI_ATTR_PASSWORD, errhp))
    {
        (void) printf("FAILED: OCIAttrSet()\n");
        return OCI_ERROR;
    }

    /* set the server attribute in the service context */
    if (OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCIX,
                  (dvoid *) srvhp, (ub4) 0, (ub4) OCI_ATTR_SERVER, errhp))
    {
        (void) printf("FAILED: OCIAttrSet()\n");
        return OCI_ERROR;
    }

    /* log on */
    if (OCISessionBegin(svchp, errhp, authp, (ub4) OCI_CRED_RDEMS,

```



```

        (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISessionBegin()\n");
        return OCI_ERROR;
    }

    /* set the session attribute in the service context */
    if (OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) authp,
        (ub4) 0, (ub4) OCI_ATTR_SESSION, errhp))
    {
        (void) printf("FAILED: OCIAttrSet()\n");
        return OCI_ERROR;
    }
    return OCI_SUCCESS;
}

/*----- logoff -----*/

/* Logoff and disconnect from the server. Free handles */
void log_off()
{
    if (tab_exists)
        drop_table();

    (void) OCISessionEnd(svchp, errhp, authp, (ub4) 0);
    (void) OCIServerDetach(srvhp, errhp, (ub4) OCI_DEFAULT);

    (void) printf("Logged off and detached from server.\n");

    (void) OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER);
    (void) OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX);
    (void) OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR);
    (void) OCIHandleFree((dvoid *) authp, (ub4) OCI_HTYPE_SESSION);
    (void) OCIDescriptorFree((dvoid *) clob, (ub4) OCI_DTYPE_LOB);
    (void) OCIDescriptorFree((dvoid *) blob, (ub4) OCI_DTYPE_LOB);
    (void) OCIHandleFree((dvoid *) stmthp, (ub4) OCI_HTYPE_STMT);

    (void) printf("All handles freed\n");
    return;
}

/*----- write_lob -----*/

/* write from files to LOBs */
sb4 write_lobs (rowind, txtfile, binfile)

```

```
int rowind;
char *txtfile;
char *binfile;
{
    ub4 loblen = 0;
    text *svptstmt = (text *)"SAVEPOINT cdemolbs_svpt";
    text *rlbkstmt = (text *)"ROLLBACK TO SAVEPOINT cdemolbs_svpt";
    ub4 txtfilelen = 0;
    ub4 binfilelen = 0;

    /* validate row indicator */
    if (!rowind || (rowind > 2))
    {
        (void) printf("ERROR: Invalid row indicator.\n");
        return OCI_ERROR;
    }
    /* open source files */
    fp1 = fopen((CONST char *)txtfile, (CONST char *) "r");
    fp2 = fopen((CONST char *)binfile, (CONST char *) "r");
    if (!(fp1 && fp2))
    {
        (void) printf("ERROR: Failed to open file(s).\n");
        return -1;
    }
    if ((txtfilelen = file_length(fp1)) > MAXLBSLEN)
    {
        (void) printf("ERROR: %s - length > 512Kbytes", txtfile);
        return -1;
    }
    if ((binfilelen = file_length(fp2)) > MAXLBSLEN)
    {
        (void) printf("ERROR: %s - length > 512Kbytes", binfile);
        return -1;
    }

    /* reset file pointers to start of file */
    (void) fseek(fp1, 0, 0);
    (void) fseek(fp2, 0, 0);

    /* set savepoint for Xn before commencing buffered mode operations */
    if (OCISTmtPrepare(stmthp, errhp, svptstmt, (ub4) strlen((char *)svptstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISTmtPrepare() svptstmt\n");
        return OCI_ERROR;
    }
}
```

```

}
if (OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                  (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                  (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: OCISstmtExecute() svptstmt\n");
    report_error();
    return OCI_ERROR;
}

(void) printf("\n===> Writing CLOB from txtfile in buffered mode.....\n\n");

/* fetch the CLOB's locator from the table for update */
if (select_clob(rowind))
{
    (void) printf("FAILED: select_clob()\n");
    log_off();
    return OCI_ERROR;
}
/* report LOB length before buffered write begins */
(void) OCILobGetLength(svchp, errhp, clob, &loblen);
(void) printf("Before buffered write, CLOB length = %d\n", loblen);

/* enable the CLOB locator for buffering operations */
if (OCILobEnableBuffering(svchp, errhp, clob))
{
    (void) printf("FAILED: OCILobEnableBuffering() CLOB\n");
    return OCI_ERROR;
}
/* write the text file contents into CLOB through the buffering subsystem */
if (buf_write_lob(rowind, clob, fpl, txtfilelen) > 0)
{
    /* if buffered write operation failed, rollback Xn to savepoint & exit */
    if (OCISstmtPrepare(stmthp, errhp, rlbkstmt,
                      (ub4) strlen((char *)rlbkstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtPrepare() rlbkstmt\n");
        return OCI_ERROR;
    }
    if (OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                      (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                      (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtExecute() rlbkstmt\n");
    }
}

```

```

        report_error();
        return OCI_ERROR;
    }

    (void) printf("FAILED: buf_write_lob() CLOB\n");
    return OCI_ERROR;
}

/* commit the Xn if the CLOB's buffer was flushed successfully */
(void) OCITransCommit(svchp, errhp, (ub4)0);

/* disable CLOB locator from buffering */
if (OCILobDisableBuffering(svchp, errhp, clob))
{
    (void) printf("FAILED: OCILobDisableBuffering() CLOB\n");
    return OCI_ERROR;
}

(void) printf("\n==> Writing BLOB from binfile in buffered mode.....\n\n");

/* fetch the BLOB's locator from the table for update */
if (select_blob(rowind))
{
    (void) printf("FAILED: select_blob()\n");
    log_off();
    return OCI_ERROR;
}

/* report LOB length before buffered write begins */
(void) OCILobGetLength(svchp, errhp, blob, &loblen);
(void) printf("Before buffered write, BLOB length = %d\n\n", loblen);

/* enable the BLOB locator for buffering operations */
if (OCILobEnableBuffering(svchp, errhp, blob))
{
    (void) printf("FAILED: OCILobEnableBuffering() BLOB\n");
    return OCI_ERROR;
}

/* write the bin file contents into BLOB through the buffering subsystem */
if (buf_write_lob(rowind, blob, fp2, binfilelen) > 0)
{
    /* if buffered write operation failed, rollback Xn to savepoint & exit */
    if (OCIStmtPrepare(stmthp, errhp, rlbkstmt,
        (ub4) strlen((char *)rlbkstmt),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtPrepare() rlbkstmt\n");
    }
}

```

```

        return OCI_ERROR;
    }
    if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
        (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtExecute() rlbkstmt\n");
        report_error();
        return OCI_ERROR;
    }

    (void) printf("FAILED: buf_write_lob() BLOB\n");
    return OCI_ERROR;
}
/* commit the Xn if the BLOB's buffer was flushed successfully */
(void) OCITransCommit(svchp, errhp, (ub4)0);

/* disable BLOB locator from buffering */
if (OCILobDisableBuffering(svchp, errhp, blob))
{
    (void) printf("FAILED: OCILobDisableBuffering() BLOB\n");
    return OCI_ERROR;
}

/* close input files */
(void) fclose(fp1);
(void) fclose(fp2);

return OCI_SUCCESS;
}

/*----- read_lobs -----*/

/* read from LOBs into files */
sb4 read_lobs (rowind, txtfile, binfile)
int  rowind;
char *txtfile;
char *binfile;
{
    ub4 loblen = 0;
    text *svptstmt = (text *) "SAVEPOINT cdemolbs_svpt";
    text *rlbkstmt = (text *) "ROLLBACK TO SAVEPOINT cdemolbs_svpt";

    if (!rowind || (rowind > 2))
    {

```

```
(void) printf("ERROR: Invalid row indicator.\n");
return -1;
}

/* open destination files */
fp1 = fopen((CONST char *)txtfile, (CONST char *) "w");
fp2 = fopen((CONST char *)binfile, (CONST char *) "w");
if (!(fp1 && fp2))
{
    (void) printf("ERROR: Failed to open file(s).\n");
    return -1;
}

/* reset file pointers to start of file */
(void) fseek(fp1, 0, 0);
(void) fseek(fp2, 0, 0);

/* fetch the BLOB's locator from the table for reads */
if (select_lob(rowind))
{
    (void) printf("FAILED: select_lob()\n");
    log_off();
    return OCI_ERROR;
}

/* report CLOB length before buffered read begins */
(void) OCILobGetLength(svchp, errhp, clob, &loblen);
(void) printf("Before buffered read, CLOB length = %d\n\n", loblen);

/* report BLOB length before buffered read begins */
(void) OCILobGetLength(svchp, errhp, blob, &loblen);
(void) printf("Before buffered read, BLOB length = %d\n\n", loblen);

/* set savepoint for Xn before commencing buffered mode operations */
if (OCISTmtPrepare(stmthp, errhp, svptstmt, (ub4) strlen((char *)svptstmt),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: OCISTmtPrepare() svptstmt\n");
    return OCI_ERROR;
}
if (OCISTmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                  (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                  (ub4) OCI_DEFAULT))
{
    (void) printf("FAILED: OCISTmtExecute() svptstmt\n");
    report_error();
}
```

```

    return OCI_ERROR;
}

/* enable the locators for buffering operations */
if (OCILOBEnableBuffering(svchp, errhp, clob))
{
    (void) printf("FAILED: OCILOBEnableBuffering() CLOB\n");
    return OCI_ERROR;
}
if (OCILOBEnableBuffering(svchp, errhp, blob))
{
    (void) printf("FAILED: OCILOBEnableBuffering() BLOB\n");
    return OCI_ERROR;
}

(void) printf("\n===> Reading CLOB into dst.txt in buffered mode...\n\n");

/* read the CLOB into buffer and write the contents to a text file */
if (buf_read_lob(rowind, clob, fp1) > 0)
{
    /* if buffered read operation failed, rollback Xn to savepoint & exit */
    if (OCIStmtPrepare(stmthp, errhp, rlbkstmt,
                      (ub4) strlen((char *)rlbkstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtPrepare() rlbkstmt\n");
        return OCI_ERROR;
    }
    if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                      (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                      (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtExecute() rlbkstmt\n");
        report_error();
        return OCI_ERROR;
    }

    (void) printf("FAILED: buf_read_lob() CLOB\n");
    return OCI_ERROR;
}

(void) printf("\n===> Reading BLOB into dst.bin in buffered mode...\n\n");

/* read the BLOB into buffer and write the contents to a binary file */
if (buf_read_lob(rowind, blob, fp2) > 0)

```

```

{
    /* if buffered read operation failed, rollback Xn to savepoint & exit */
    if (OCIStmtPrepare(stmthp, errhp, rlbkstmt,
                      (ub4) strlen((char *)rlbkstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtPrepare() rlbkstmt\n");
        return OCI_ERROR;
    }
    if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                      (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                      (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtExecute() rlbkstmt\n");
        report_error();
        return OCI_ERROR;
    }

    (void) printf("FAILED: buf_read_clob()\n");
    return OCI_ERROR;
}

/* commit the Xn if buffered reads went off successfully */
(void) OCITransCommit(svchp, errhp, (ub4)0);

/* disable locator for buffering */
if (OCILobDisableBuffering(svchp, errhp, clob))
{
    (void) printf("FAILED: OCILobDisableBuffering() \n");
    return OCI_ERROR;
}
if (OCILobDisableBuffering(svchp, errhp, blob))
{
    (void) printf("FAILED: OCILobDisableBuffering() \n");
    return OCI_ERROR;
}

/* close output files */
(void) fclose(fp1);
(void) fclose(fp2);

return OCI_SUCCESS;
}

/*----- Public functions -----*/

```



```

/*----- select_clob -----*/

/* select locator from the CLOB column */
sb4 select_clob(rowind)
int rowind;
{
    int    colc    = rowind;
    text *sqlstmt = (text *)"SELECT C1 FROM FOO WHERE C3 = :1 FOR UPDATE";
    /* we need the 'FOR UPDATE' clause since we need to write to the lob */

    /* prepare select statement */
    if (OCIStmtPrepare(stmthp, errhp, sqlstmt, (ub4) strlen((char *)sqlstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtPrepare() sqlstmt\n");
        return OCI_ERROR;
    }
    /* associate variable colc with bind placeholder #1 in the SQL statement */
    if (OCIBindByPos(stmthp, &bndhp, errhp, (ub4) 1,
                    (dvoid *) &colc, (sb4) sizeof(colc), SQLT_INT,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIBindByPos()\n");
        return OCI_ERROR;
    }
    /* associate clob var with its define handle */
    if (OCIDefineByPos(stmthp, &defnp1, errhp, (ub4) 1,
                    (dvoid *) &clob, (sb4) -1, (ub2) SQLT_CLOB,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIDefineByPos() CLOB\n");
        return OCI_ERROR;
    }
    /* execute the select and fetch one row */
    if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                    (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                    (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIStmtExecute() sqlstmt\n");
        return OCI_ERROR;
    }
    return OCI_SUCCESS;
}

```

```

/*----- select_blob -----*/

/* select locator from the BLOB column */
sb4 select_blob(rowind)
int rowind;
{
    int colc = rowind;
    text *sqlstmt = (text *)"SELECT C2 FROM FOO WHERE C3 = :1 FOR UPDATE";
    /* we need the 'FOR UPDATE' clause since we need to write to the lob */

    /* prepare select statement */
    if (OCISstmtPrepare(stmthp, errhp, sqlstmt, (ub4) strlen((char *)sqlstmt),
                        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtPrepare() sqlstmt\n");
        return OCI_ERROR;
    }
    /* associate variable colc with bind placeholder #1 in the SQL statement */
    if (OCIBindByPos(stmthp, &bndhp, errhp, (ub4) 1,
                    (dvoid *) &colc, (sb4) sizeof(colc), SQLT_INT,
                    (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIBindByPos()\n");
        return OCI_ERROR;
    }
    /* associate blob var with its define handle */
    if (OCIDefineByPos(stmthp, &defnp2, errhp, (ub4) 1,
                      (dvoid *) &blob, (sb4) -1, (ub2) SQLT_BLOB,
                      (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIDefineByPos()\n");
        return OCI_ERROR;
    }
    /* execute the select and fetch one row */
    if (OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                       (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                       (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtExecute() sqlstmt\n");
        return OCI_ERROR;
    }
    return OCI_SUCCESS;
}

```

```

/*----- select_lobs -----*/

/* select lob locators from the CLOB, BLOB columns */
sb4 select_lobs(rowind)
int rowind;
{
    int    colc    = rowind;
    text *sqlstmt = (text *)"SELECT C1, C2 FROM FOO WHERE C3 = :1";
    /* we don't need the 'FOR UPDATE' clause since
       we are just reading the LOBs */

    /* prepare select statement */
    if (OCISstmtPrepare(stmthp, errhp, sqlstmt, (ub4) strlen((char *)sqlstmt),
                        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISstmtPrepare() sqlstmt\n");
        return OCI_ERROR;
    }
    /* associate variable colc with bind placeholder #1 in the SQL statement */
    if (OCIBindByPos(stmthp, &bndhp, errhp, (ub4) 1,
                    (dvoid *) &colc, (sb4) sizeof(colc), SQLT_INT,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIBindByPos()\n");
        return OCI_ERROR;
    }
    /* associate clob and blob vars with their define handles */
    if (OCIDefineByPos(stmthp, &defnp1, errhp, (ub4) 1,
                    (dvoid *) &clob, (sb4) -1, (ub2) SQLT_CLOB,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT)
    || OCIDefineByPos(stmthp, &defnp2, errhp, (ub4) 2,
                    (dvoid *) &blob, (sb4) -1, (ub2) SQLT_BLOB,
                    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCIDefineByPos()\n");
        return OCI_ERROR;
    }
    /* execute the select and fetch one row */
    if (OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                    (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                    (ub4) OCI_DEFAULT))
    {

```

```

        (void) printf("FAILED: OCISstmtExecute() sqlstmt\n");
        return OCI_ERROR;
    }
    return OCI_SUCCESS;
}

/*----- buf_write_lob -----*/
/*
 * Read operating system files into local buffers and then write these local
 * buffers to LOBs using buffering system.
 */
sb4 buf_write_lob(rowind, locator, fp, filelen)
int rowind;
OCILobLocator *locator;
FILE *fp;
ub4 filelen;
{
    ub4 offset = 1;
    ub1 bufp[MAXBUFLLEN];
    ub4 amtp;
    ub4 nbytes = 0;
    ub4 remainder = filelen;

    /* reset per read/write buffer size and perform the first read */
    amtp = nbytes = (filelen > MAXBUFLLEN) ? MAXBUFLLEN : filelen;

    /* write into the LOB's client-side buffer
       (upto max 16 pages of 32K each) */
    while (remainder > 0)
    {
        if (fread((void *)bufp, (size_t)nbytes, (size_t)1, fp) != 1)
        {
            (void) printf("ERROR: read file.\n");
            return(OCI_ERROR);
        }
        if (feof(fp))
        {
            (void) printf("Exit - End of file reached\n", amtp, offset);
            break;
        }
        (void) printf("Write %d bytes out of remaining %d bytes at off %d\n",
                      amtp, remainder, offset);
        if (OCILobWrite(svchp, errhp, locator, &amtp, (ub4) offset,
                       (dvoid *) bufp,
                       (ub4) nbytes, OCI_ONE_PIECE, (dvoid *) 0,

```

```

        (sb4 (*)(dvoid *, dvoid *, ub4 *, ub1 *)) 0,
        (ub2) 0, (ub1) SQLCS_IMPLICIT))
    {
        (void) printf("FAILED: OCILobWrite() \n");
        return(OCI_ERROR);
    }
    if (amtp < nbytes)
    {
        (void) printf("FAILED: Full file not written \n");
        return(OCI_ERROR);
    }
    amtp = nbytes;
    offset += nbytes;
    remainder -= nbytes;
}

/* flush the buffers back to the server */
(void) printf("Flush LOB's buffer to server\n");
if (OCILobFlushBuffer(svchp, errhp, locator, OCI_LOB_BUFFER_NOFREE))
{
    (void) printf("FAILED: OCILobFlushBuffer() \n");
    return OCI_ERROR;
}
return OCI_SUCCESS;
}

/*----- buf_read_lob -----*/

/*
 * Read LOBs using buffered mode into local buffers and writes them into
 * operating system files.
 */
sb4 buf_read_lob(rowind, locator, fp)
int rowind;
OCILobLocator *locator;
FILE *fp;
{
    ub4 offset = 1;
    ub1 bufp[MAXBUFLen];
    ub4 amtp = 0;
    ub4 nbytes = 0;

    /* set amount to be read per iteration */
    amtp = nbytes = MAXBUFLen;

```

```

/*
 * read from CLOB and write to text file (in the process, populating upto
 * 16 pages of 32K each in the buffering subsystem).
 */
while (amtp)
{
    (void) printf("Reading %d bytes from offset %d\n", amtp, offset);
    if (OCILobRead(svchp, errhp, locator, &amtp, (ub4) offset, (dvoid *) bufp,
        (ub4) nbytes, (dvoid *) 0,
        (sb4 *) (dvoid *, CONST dvoid *, ub4, ub1)) 0,
        (ub2) 0, (ub1) SQLCS_IMPLICIT))
    {
        (void) printf("FAILED: OCILobRead() \n");
        return OCI_ERROR;
    }
    (void) fwrite((void *) bufp, (size_t) amtp, (size_t) 1, fp);      /* write
buffer to file */
    offset += nbytes;
}
return OCI_SUCCESS;
}

/*----- drop_table -----*/

/* Drop table FOO before logging off from the server */
void drop_table()
{
    text *dropstmt = (text *) "DROP TABLE FOO";

    /* prepare drop statement */
    if (OCISmtPrepare(stmthp, errhp, dropstmt, (ub4) strlen((char *) dropstmt),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISmtPrepare() dropstmt\n");
        return;
    }
    /* execute drop statement */
    if (OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
        (CONST OCISnapshot *) 0, (OCISnapshot *) 0,
        (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISmtExecute() dropstmt\n");
        return;
    }
    return;
}

```

```
}

/*----- report_error -----*/

/* retrieve error message and print it out */
void report_error()
{
    text msgbuf[512];
    sb4 errcode = 0;

    (void) OCLErrorGet((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                      msgbuf, (ub4) sizeof(msgbuf), (ub4) OCI_HTYPE_ERROR);
    (void) printf("ERROR CODE = %d\n", errcode);
    (void) printf("%.512s\n", 512, msgbuf);
    return;
}

/*----- file_length -----*/

/* get the length of the input file */
ub4 file_length(fp)
FILE *fp;
{
    fseek(fp, 0, SEEK_END);
    return (ub4) (ftell(fp));
}
```

Example 7, REF Pinning and Navigation

```
/*
    NAME
        cdemobj.c

    DESCRIPTION
        Demo of selection of a REF and display the pinned object through
        navigational interface.
*/

#ifdef CDEMO_OBJ_ORACLE
#include "cdemobj.h"
#endif

/* statement to select a ref from an extent table customer_tab */
static const text *const selref = (text *)
    "SELECT REF(customer_tab) from customer_tab";

/* statement to create the type address */
static const text *const create_type_address = (text *)
"CREATE TYPE address AS OBJECT (\
    no          NUMBER,\
    street      VARCHAR(60),\
    state       CHAR(2),\
    zip         CHAR(10)\
)";

/* statement to create the typed table address_tab */
static const text *const create_type_addr_tab = (text *)
"create type addr_tab is table of address";

/* statement to create the type person */
static const text *const create_type_person = (text *)
"CREATE TYPE person AS OBJECT (\
    firstname   CHAR(20),\
    lastname    varchar(20),\
    age         int,\
    salary      float,\
    bonus       double precision,\
    retirement_fund int,\
    number_of_kids smallint,\
    years_of_school numeric(10, 2),\
    preaddr     addr_tab,\
    birthday    date,\
```



```

    number_of_pets    real,\
    comment1          raw(200),\
    comment2          clob,\
    comment3          varchar2(200),\
    addr              ADDRESS\
);

/* statement to create the type customer */
static const text *const create_type_customer = (text *)
"CREATE TYPE customer AS OBJECT (\
    account          char(20),\
    aperson          REF person\
)";

/* statement to create the typed table person */
static const text *const create_table_person = (text *)
"create table person_tab of person nested table preaddr store as
person_preaddr_table";

/* statement to create the typed table customer_tab */
static const text *const create_table_customer = (text *)
"create table customer_tab of customer";

/* statement to insert data into table customer_tab */
static const text *const insert_customer = (text *)
"insert into customer_tab values('00001', null)";

/* statement to insert data into table person_tab */
static const text *const insert_person = (text *)
"insert into person_tab values('Sandy', 'Wood', 25, 32000, 10000, 20000, 3,\
    15, addr_tab(),\
    to_date('1961 08 23', 'YYYY MM DD'), 2,\
    '1234567890', 'This is a test', 'This is a test',\
    ADDRESS(8888, 'Fenley Road', 'CA', '91406'))";

/* statement to insert data into the nested table in person_tab */
static const text *const insert_address1 = (text *)
"insert into the (select preaddr from person_tab where\
    firstname='Sandy') values\
    (715, 'South Henry', 'ca', '95117)";
static const text *const insert_address2 = (text *)
"insert into the (select preaddr from person_tab where\
    firstname='Sandy') values\
    (6830, 'Woodley Ave', 'ca', '90416)";

```

```

/* statement to update the ref in the table customer_tab */
static const text *const update_customer = (text *)
"update customer_tab set aperson = (select ref(person_tab)\
                                     from person_tab where\
                                     firstname = 'Sandy')";

/*****
 * Check the error and display the error message
 *****/
static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    sb4 errcode;

    switch (status)
    {
        case OCI_SUCCESS:
            break;
        case OCI_SUCCESS_WITH_INFO:
            break;
        case OCI_NEED_DATA:
            break;
        case OCI_NO_DATA:
            break;
        case OCI_ERROR: /* get the error back and display on the screen */
            (void) OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                               errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
            (void) printf("Error - %s\n", errbuf);
            break;
        case OCI_INVALID_HANDLE:
            break;
        case OCI_STILL_EXECUTING:
            break;
        case OCI_CONTINUE:
            break;
        default:
            break;
    }
}

/*****
 * Display attribute value of an object
 *****/
static void display_attr_val(envhp, errhp, names, typecode, attr_value)

```

```

OCIEnv *envhp;           /* environment handle */
OCIError *errhp;         /* error handle */
text *names;             /* the name of the attribute */
OCITypeCode typecode;    /* the type code */
dvoid *attr_value;       /* the value pointer */
{
    text      str_buf[200];
    double    dnum;
    ub4       text_len, str_len;
    OCIRaw     *raw = (OCIRaw *) 0;
    OCIStrng   *vs = (OCIStrng *) 0;

    /* display the data based on the type code */
    switch (typecode)
    {
        case OCI_TYPECODE_DATE :           /* fixed length string */
            str_len = 200;
            (void) OCIDateToText(errhp, (CONST OCIDate *) attr_value,
                                (CONST text*)
                                "Month dd, SYYY, HH:MI A.M.",
                                (ub1) 27, (CONST text*) "American", (ub4) 8,
                                (ub4 *)&str_len, str_buf);
            str_buf[str_len+1] = '\0';
            (void) printf("attr %s = %s\n", names, (text *) str_buf);
            break;
        case OCI_TYPECODE_RAW :             /* RAW */
            raw = *(OCIRaw **) attr_value;
            (void) printf("attr %s = %s\n", names, (text *) OCIRawPtr(envhp, raw));
            break;
        case OCI_TYPECODE_CHAR :           /* fixed length string */
        case OCI_TYPECODE_VARCHAR :       /* varchar */
        case OCI_TYPECODE_VARCHAR2 :      /* varchar2 */
            vs = *(OCIStrng **) attr_value;
            (void) printf("attr %s = %s\n", names, (text *)
                           OCIStrngPtr(envhp, vs));
            break;
        case OCI_TYPECODE_SIGNED8 :        /* BYTE - sb1 */
            (void) printf("attr %s = %d\n", names, *(sb1 *) attr_value);
            break;
        case OCI_TYPECODE_UNSIGNED8 :     /* UNSIGNED BYTE - ub1 */
            (void) printf("attr %s = %d\n", names, *(ub1 *) attr_value);
            break;
        case OCI_TYPECODE_OCTET :         /* OCT */
            (void) printf("attr %s = %d\n", names, *(ub1 *) attr_value);
            break;
    }
}

```

```

        case OCI_TYPECODE_UNSIGNED16:                /* UNSIGNED SHORT */
        case OCI_TYPECODE_UNSIGNED32:                /* UNSIGNED LONG */
        case OCI_TYPECODE_REAL:                      /* REAL */
        case OCI_TYPECODE_DOUBLE:                   /* DOUBLE */
        case OCI_TYPECODE_INTEGER:                   /* INT */
        case OCI_TYPECODE_SIGNED16:                  /* SHORT */
        case OCI_TYPECODE_SIGNED32:                  /* LONG */
        case OCI_TYPECODE_DECIMAL:                   /* DECIMAL */
        case OCI_TYPECODE_FLOAT:                     /* FLOAT */
        case OCI_TYPECODE_NUMBER:                    /* NUMBER */
        case OCI_TYPECODE_SMALLINT:                  /* SMALLINT */
            (void) OCINumberToReal(errhp, (CONST OCINumber *) attr_value,
                                   (uword)sizeof(dnum),
                                   (dvoid *) &dnum);
            (void) printf("attr %s = %f\n", names, dnum);
            break;
        default:
            (void) printf("attr %s - typecode %d\n", names, typecode);
            break;
    }
}

/*****
 * Dump the info of any object
 *****/
static void dump_object(envhp, errhp, svchp, tdo, obj, null_obj)
OCIEnv *envhp;                /* environment handle */
OCIError *errhp;              /* error handle */
OCISvcCtx *svchp;             /* service handle */
OCIType *tdo;                 /* type descriptor */
dvoid *obj;                   /* object pointer */
dvoid *null_obj;              /* parallel null struct pointer */
{
    text *names[50];
    text *lengths[50];
    text *indexes[50];
    ub2 count, pos;
    OCITypeElem *ado;
    ub4 text_len, str_len;
    ub4 i;
    OCITypeCode typecode;
    OCIIInd attr_null_status;
    dvoid *attr_null_struct;
    dvoid *attr_value;

```

```

OCIType      *attr_tdo, *element_type;
dvoid        *object;
dvoid        *null_object;
OCIType      *object_tdo;
ub1          status;
OCISRef      *type_ref;
text         str_buf[200];
double       dnum;
dvoid        *element = (dvoid *) 0, *null_element = (dvoid *) 0;
boolean      exist, eoc, boc;
sb4          index;
OCIDescribe  *dschp = (OCIDescribe *) 0, *dschp1 = (OCIDescribe *) 0;
text         *namep, *typenamep;
dvoid        *list_attr;
OCIIter      *itr = (OCIIter *) 0;
dvoid        *pamp = (dvoid *) 0, *parmdp = (dvoid *) 0, *pamp1 = (dvoid *) 0,
             *pamp2 = (dvoid *) 0;
OCISRef      *elem_ref = (OCISRef *) 0;

checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **) &dschp,
                               (ub4) OCI_HTYPE_DESCRIBE,
                               (size_t) 0, (dvoid **) 0));

checkerr(errhp, OCIDescribeAny(svchp, errhp, (dvoid *) tdo,
                               (ub4) 0, OCI_OTYPE_PTR, (ub1)1,
                               (ub1) OCI_PTYPE_TYPE, dschp));

checkerr(errhp, OCIAttrGet((dvoid *) dschp, (ub4) OCI_HTYPE_DESCRIBE,
                          (dvoid *)&pamp, (ub4 *)0, (ub4)OCI_ATTR_PARAM, errhp));

checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &typenamep, (ub4 *) &str_len,
                          (ub4) OCI_ATTR_NAME, (OCIError *) errhp));
typenamep[str_len] = '\0';

printf("starting displaying instance of type '%s'\n", typenamep);

/* loop through all attributes in the type */
checkerr(errhp, OCIAttrGet((dvoid*) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &count, (ub4 *) 0,
                          (ub4) OCI_ATTR_NUM_TYPE_ATTRS, (OCIError *) errhp));

checkerr(errhp, OCIAttrGet((dvoid *) pamp, (ub4) OCI_DTYPE_PARAM,
                          (dvoid *)&list_attr, (ub4 *)0,
                          (ub4)OCI_ATTR_LIST_TYPE_ATTRS, (OCIError *)errhp));

```

```

/* loop through all attributes in the type */
for (pos = 1; pos <= count; pos++)
{
    checkerr(errhp, OCIParamGet((dvoid *) list_attr,
                                (ub4) OCI_DTYPE_PARAM, errhp,
                                (dvoid *)&parmdp, (ub4) pos));

    checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                                (dvoid*) &namep, (ub4 *) &str_len,
                                (ub4) OCI_ATTR_NAME, (OCIError *) errhp));
    namep[str_len] = '\0';

    /* get the attribute */
    if (OCIObjectGetAttr(envhp, errhp, obj, null_obj, tdo,
                        &namep, &str_len, 1,
                        (ub4 *)0, 0, &attr_null_status, &attr_null_struct,
                        &attr_value, &attr_tdo) != OCI_SUCCESS)
        (void) printf("BUG -- OCIObjectGetAttr, expect OCI_SUCCESS.\n");

    /* get the type code of the attribute */
    checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
                                (dvoid*) &typecode, (ub4 *) 0,
                                (ub4) OCI_ATTR_TYPECODE,
                                (OCIError *) errhp));

    /* support only fixed length string, ref and embedded object */
    switch (typecode)
    {
        case OCI_TYPECODE_OBJECT :                /* embedded object */
            printf("attribute %s is an embedded object.
                    Display instance ....\n",
                    namep);
            /* recursive call to dump nested object data */
            dump_object(envhp, errhp, svchp, attr_tdo, attr_value,
                        attr_null_struct);
            break;
        case OCI_TYPECODE_REF :                    /* embedded object */
            printf("attribute %s is a ref. Pin and display instance ...\n",
                    namep);
            /* pin the object */
            if (OCIObjectPin(envhp, errhp, *(OCIRef **)attr_value,
                            (OCIComplexObject *)0, OCI_PIN_ANY,
                            OCI_DURATION_SESSION, OCI_LOCK_NONE,

```

```

        (dvoid **)&object) != OCI_SUCCESS)
    (void) printf("BUG -- OCIObjectPin, expect OCI_SUCCESS.\n");
/* allocate the ref */
if (( status = OCIObjectNew(envhp, errhp, svchp, OCI_TYPECODE_REF,
    (OCIType *)0,
    (dvoid *)0, OCI_DURATION_DEFAULT, TRUE, (dvoid **)&type_ref))
    != OCI_SUCCESS)
    (void) printf("BUG -- OCIObjectNew, expect OCI_SUCCESS.\n");
/* get the ref of the type from the object */
if (( status = OCIObjectGetTypeRef(envhp, errhp, object, type_ref))
    != OCI_SUCCESS)
    (void) printf("BUG -- ORIOGTR, expect OCI_SUCCESS.\n");
/* pin the type ref to get the type object */
if (OCIObjectPin(envhp, errhp, type_ref, (OCIComplexObject *)0,
    OCI_PIN_ANY, OCI_DURATION_SESSION, OCI_LOCK_NONE,
    (dvoid **)&object_tdo) !=
    OCI_SUCCESS)
    (void) printf("BUG -- OCIObjectPin, expect OCI_SUCCESS.\n");
/* get null struct of the object */
if (( status = OCIObjectGetInd(envhp, errhp, object,
    &null_object)) != OCI_SUCCESS)
    (void) printf("BUG -- ORIOGNS, expect OCI_SUCCESS.\n");
/* call the function recursively to dump the pinned object */
dump_object(envhp, errhp, svchp, object_tdo, object,
    null_object);
case OCI_TYPECODE_NAMEDCOLLECTION:
    checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **)&dschp1,
        (ub4) OCI_HTYPE_DESCRIBE,
        (size_t) 0, (dvoid **) 0));

    checkerr(errhp, OCIDescribeAny(svchp, errhp, (dvoid *) attr_tdo,
        (ub4) 0, OCI_OTYPE_PTR, (ub1)1,
        (ub1) OCI_PTYPE_TYPE, dschp1));

    checkerr(errhp, OCIAttrGet((dvoid *) dschp1, (ub4)
        OCI_HTYPE_DESCRIBE,
        (dvoid *)&pamp1, (ub4 *)0, (ub4)OCI_ATTR_PARAM, errhp));

/* get the collection type code of the attribute */
checkerr(errhp, OCIAttrGet((dvoid*) pamp1, (ub4) OCI_DTYPE_PARAM,
    (dvoid*) &typecode, (ub4 *) 0,
    (ub4) OCI_ATTR_COLLECTION_TYPECODE,
    (OCIError *) errhp));
switch (typecode)

```

```

{
    case OCI_TYPECODE_VARRAY:                /* variable array */
        (void) printf
        ("\n---> Dump the table from the top to the bottom.\n");
        checkerr(errhp, OCIAttrGet((dvoid*) pamp1, (ub4)
            OCI_DTYPE_PARAM,
            (dvoid*) &pamp2, (ub4 *) 0,
            (ub4) OCI_ATTR_COLLECTION_ELEMENT,
            (OCIError *) errhp));
        checkerr(errhp, OCIAttrGet((dvoid*) pamp2,
            (ub4) OCI_DTYPE_PARAM,
            (dvoid*) &elem_ref, (ub4 *) 0,
            (ub4) OCI_ATTR_REF_TDO,
            (OCIError *) errhp));
        checkerr(OCITypeByRef(envhp, errhp, elem_ref, OCI_PIN_DEFAULT,
            0, &element_type));
        /* initialize the iterator */
        checkerr(errhp, OCIIterCreate(envhp, errhp, (CONST OCIColl*)
            attr_value, &itr));
        /* loop through the iterator */
        for(eoc = FALSE; !OCIIterNext(envhp, errhp, itr, (dvoid **)
            &element,
            (dvoid **) &null_element, &eoc) && !eoc;)
        {
            /* if type is named type, call the same function recursively */
            if (typecode == OCI_TYPECODE_OBJECT)
                dump_object(envhp, errhp, svchp, element_type, element,
                    null_element);
            else /* else, display the scaler type attribute */
                display_attr_val(envhp, errhp, namep, typecode, element);
        }
        break;

    case OCI_TYPECODE_TABLE:                /* nested table */
        (void) printf
        ("\n---> Dump the table from the top to the bottom.\n");
        /* go to the first element and print out the index */
        checkerr(errhp, OCIAttrGet((dvoid*) pamp1, (ub4)
            OCI_DTYPE_PARAM,
            (dvoid*) &pamp2, (ub4 *) 0,
            (ub4) OCI_ATTR_COLLECTION_ELEMENT,
            (OCIError *) errhp));
        checkerr(errhp, OCIAttrGet((dvoid*) pamp2,
            (ub4) OCI_DTYPE_PARAM,
            (dvoid*) &elem_ref, (ub4 *) 0,

```



```

        (ub4) OCI_ATTR_REF_TDO,
        (OCIError *) errhp));
checkerr(errhp, OCITYPEByRef(envhp, errhp, elem_ref,
        OCI_DURATION_SESSION,
        OCI_TYPEGET_HEADER, &element_type));
attr_value = *(dvoid **)attr_value;
/* move to the first element in the nested table */
checkerr(errhp, OCITableFirst(envhp, errhp, (CONST OCITable*)
        attr_value, &index));
(void) printf
("    The index of the first element is : %d.\n", index);
/* print out the element */
checkerr(errhp, OCICollGetElem(envhp, errhp,
        (CONST OCIColl *) attr_value, index,
        &exist, (dvoid **) &element,
        (dvoid **) &>null_element));
/* if it is named type, recursively call the same function */
checkerr(errhp, OCIAttrGet((dvoid*) pamp2,
        (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &typecode, (ub4 *) 0,
        (ub4) OCI_ATTR_TYPECODE,
        (OCIError *) errhp));
if (typecode == OCI_TYPECODE_OBJECT)
    dump_object(envhp, errhp, svchp, element_type,
        (dvoid *)element, (dvoid *)null_element);
else
    display_attr_val(envhp, errhp, namep, typecode, element);

for(;;!OCITableNext(envhp, errhp, index, (CONST OCITable *)
    attr_value,
    &index, &exist) && exist;)
{
    checkerr(errhp, OCICollGetElem(envhp, errhp, (CONST OCIColl *)
        attr_value, index,
        &exist, (dvoid **) &element,
        (dvoid **) &>null_element));
    if (typecode == OCI_TYPECODE_OBJECT)
        dump_object(envhp, errhp, svchp, element_type,
            (dvoid *)element, (dvoid *)null_element);
    else
        display_attr_val(envhp, errhp, namep, typecode, element);
}
break;
default:
break;

```

```

    }
    checkerr(errhp, OCIHandleFree((dvoid *) dschp1, (ub4)
                                OCI_HTYPE_DESCRIBE));
    break;
default: /* scalar type, display the attribute value */
    if (attr_null_status == OCI_IND_NOTNULL)
    {
        display_attr_val(envhp, errhp, namep, typecode, attr_value);
    }
    else
        printf("attr %s is null\n", namep);
    break;
}
}

checkerr(errhp, OCIHandleFree((dvoid *) dschp, (ub4) OCI_HTYPE_DESCRIBE));
printf("finishing displaying instance of type '%s'\n", typenamep);
}

/*****
 * Setup the schema and insert the data
 *****/
void setup(envhp, svchp, stmthp, errhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCISmt *stmthp;
OCIError *errhp;
{
    /* create the schema and populate the data */
    checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) create_type_address,
                                (ub4) strlen((const char *) create_type_address),
                                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                (OCISnapshot *)
                                NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) create_type_addr_tab,
                                (ub4) strlen((const char *) create_type_addr_tab),
                                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                (OCISnapshot *)
                                NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));
}

```

```

checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) create_type_person,
                             (ub4) strlen((const char *) create_type_person),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (OCISnapshot *)
                             NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) create_type_customer,
                             (ub4) strlen((const char *) create_type_customer),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (OCISnapshot *)
                             NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) create_table_person,
                             (ub4) strlen((const char *) create_table_person),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (OCISnapshot *)
                             NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *)
                             create_table_customer,
                             (ub4) strlen((const char *) create_table_customer),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (OCISnapshot *)
                             NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) insert_customer,
                             (ub4) strlen((const char *) insert_customer),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                             (OCISnapshot *)
                             NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) insert_person,
                             (ub4) strlen((const char *) insert_person),
                             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

```

```

        checkerr(errhp, OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                      (OCISnapshot *)
                                      NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

        checkerr(errhp, OCISstmtPrepare(stmthp, errhp, (text *) insert_address1,
                                      (ub4) strlen((const char *) insert_address1),
                                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

        checkerr(errhp, OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                      (OCISnapshot *)
                                      NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

        checkerr(errhp, OCISstmtPrepare(stmthp, errhp, (text *) insert_address2,
                                      (ub4) strlen((const char *) insert_address2),
                                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

        checkerr(errhp, OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                      (OCISnapshot *)
                                      NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

        checkerr(errhp, OCISstmtPrepare(stmthp, errhp, (text *) update_customer,
                                      (ub4) strlen((const char *) update_customer),
                                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

        checkerr(errhp, OCISstmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                      (OCISnapshot *)
                                      NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

        checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) 0));
    }

/
*****/
void select_pin_display(envhp, svchp, stmthp, errhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCISstmt *stmthp;
OCIError *errhp;
{
    sword status = OCI_SUCCESS;
    OCIDefine *defnp;
    OCIRef *custref = (OCIRef *) 0, *per_type_ref = (OCIRef *) 0;
    OCIRef *cust_type_ref = (OCIRef *) 0;

```

```

ub4    custsize;
customer *cust = (customer *) 0, *custnew = (customer *) 0;
null_customer *null_cust = (null_customer *) 0,
    *null_custnew = (null_customer *) 0;
person *per = (person *) 0;
null_person *null_per = (null_person *) 0;
null_address *nt_null = (null_address *) 0;
OCIType *pertos = (OCIType *) 0, *custtdo = (OCIType *) 0;
address *addr = (address *) 0;
sb4 index;
boolean exist;
dvoid *tabobj = (dvoid *) 0;

(void) printf("\n=====\\n");

/* allocate ref */
if (( status = OCIObjectNew(envhp, errhp, svchp, OCI_TYPECODE_REF,
    (OCIType *)0,
    (dvoid *)0, OCI_DURATION_DEFAULT, TRUE,
    (dvoid **) &per_type_ref))
    != OCI_SUCCESS)
    (void) printf("BUG -- OCIObjectNew, expect OCI_SUCCESS.\\n");

/* allocate ref */
if (( status = OCIObjectNew(envhp, errhp, svchp, OCI_TYPECODE_REF,
    (OCIType *)0,
    (dvoid *)0, OCI_DURATION_DEFAULT, TRUE,
    (dvoid **) &cust_type_ref))
    != OCI_SUCCESS)
    (void) printf("BUG -- OCIObjectNew, expect OCI_SUCCESS.\\n");

/* define the application request */
checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) selref,
    (ub4) strlen((const char *) selref),
    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCIHandleAlloc( (dvoid *) stmthp, (dvoid **) &defnp,
    (ub4) OCI_HTYPE_DEFINE,
    0, (dvoid **) 0));

checkerr(errhp, OCIDefineByPos(stmthp, &defnp, errhp, (ub4) 1, (dvoid *) 0,
    (sb4) 0, SQLT_REF, (dvoid *) 0, (ub2 *)0,
    (ub2 *)0, (ub4) OCI_DEFAULT));

checkerr(errhp, OCIDefineObject(defnp, errhp, (OCIType *) 0,

```

```
        (dvoid **) &custref,
        &custsize, (dvoid **) 0, (ub4 *) 0));

checkerr(errhp, OCISstmtExecute(svchp, stmthp, errhp, (ub4) 0,
        (ub4) 0, (OCISnapshot *)
        NULL, (OCISnapshot *) NULL,
        (ub4) OCI_DEFAULT));

while ((status = OCISstmtFetch(stmthp, errhp, (ub4) 1, (ub4) OCI_FETCH_NEXT,
        (ub4) OCI_DEFAULT)) == 0)
{
    (void) printf("\n-----\n");

    /* pin the ref and get the typed table to get to person */
    checkerr(errhp, OCIObjectPin(envhp, errhp, custref,
        (OCIComplexObject *)0,
        OCI_PIN_ANY, OCI_DURATION_SESSION,
        OCI_LOCK_NONE, (dvoid **) &cust));
    (void) printf("The customer account number is %s\n",
        OCISstringPtr(envhp, cust->account));
    if (( status = OCIObjectGetInd(envhp, errhp, (dvoid *) cust,
        (dvoid **) &null_cust)) != OCI_SUCCESS)
    {
        (void) printf("BUG -- ORIOGNS, expect OCI_SUCCESS.\n");
    }
    else
    {
        (void) printf("null_cus = %d, null_account = %d, null_aperson = %d\n",
            null_cust->null_cus, null_cust->null_account,
            null_cust->null_aperson);
    }

    checkerr(errhp, OCIObjectPin(envhp, errhp, cust->aperson,
        (OCIComplexObject *)0,
        OCI_PIN_ANY, OCI_DURATION_SESSION,
        OCI_LOCK_NONE, (dvoid **) &per));

    if (( status = OCIObjectGetInd(envhp, errhp, (dvoid *) per,
        (dvoid **) &null_per)) != OCI_SUCCESS)
    {
        (void) printf("BUG -- ORIOGNS, expect OCI_SUCCESS.\n");
    }
    else
    {

```

```

        checkerr(errhp, OCIObjectGetTypeRef(envhp, errhp, (dvoid *)per,
                                           per_type_ref));
        checkerr(errhp, OCIObjectPin(envhp, errhp, per_type_ref,
                                     (OCIComplexObject *)0, OCI_PIN_ANY,
                                     OCI_DURATION_SESSION, OCI_LOCK_NONE,
                                     (dvoid **) &pertdo));
        dump_object(envhp, errhp, svchp, pertdo, (dvoid *) per,
                    (dvoid *) null_per);
    }
}

if ( status != OCI_NO_DATA )
    checkerr(errhp, status);

(void) printf("\n\n");
}

/*****
 * Clean up the schema and the data
 *****/
void cleanup(envhp, svchp, stmthp, errhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIError *errhp;
{
    /* clean up the schema */
    checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) "drop table
customer_tab",
                                (ub4) strlen((const char *) "drop table customer_tab"
                                ),
                                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1,
                                (ub4) 0, (OCISnapshot *)
                                NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCIStmtPrepare(stmthp, errhp, (text *) "drop table
person_tab",
                                (ub4) strlen((const char *) "drop table person_tab" ),
                                (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1,
                                (ub4) 0, (OCISnapshot *)

```

```

        NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTmtPrepare(stmthp, errhp, (text *) "drop type customer",
                               (ub4) strlen((const char *) "drop table customer" ),
                               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTmtExecute(svchp, stmthp, errhp, (ub4) 1,
                               (ub4) 0, (OCISnapshot *)
                               NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTmtPrepare(stmthp, errhp, (text *) "drop type person",
                               (ub4) strlen((const char *) "drop table person" ),
                               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTmtExecute(svchp, stmthp, errhp, (ub4) 1,
                               (ub4) 0, (OCISnapshot *)
                               NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTmtPrepare(stmthp, errhp, (text *) "drop type addr_tab",
                               (ub4) strlen((const char *) "drop table addr_tab" ),
                               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTmtExecute(svchp, stmthp, errhp, (ub4) 1,
                               (ub4) 0, (OCISnapshot *)
                               NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTmtPrepare(stmthp, errhp, (text *) "drop type address",
                               (ub4) strlen((const char *) "drop type address"),
                               (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTmtExecute(svchp, stmthp, errhp, (ub4) 1,
                               (ub4) 0, (OCISnapshot *)
                               NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) 0));
}

/*****/
int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;

```



```

OCISession *usrhp;
OCIStmt *stmthp;
dvoid *tmp;

/* initialize the process */
(void) OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT,
                    (dvoid *)0, (dvoid * (*)()) 0,
                    (dvoid * (*)()) 0, (void (*)()) 0 );

/* initialize the environmental handle */
(void) OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

/* get the error handle */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp,
                     (ub4) OCI_HTYPE_ERROR,
                     52, (dvoid **) &tmp);

/* two server contexts */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp,
                     (ub4) OCI_HTYPE_SERVER,
                     52, (dvoid **) &tmp);

/* attach the server */
(void) OCIServerAttach( srvhp, errhp, (text *) "", (sb4) 0, (ub4)
                     OCI_DEFAULT);

/* get the service handle */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp,
                     (ub4) OCI_HTYPE_SVCCTX,
                     52, (dvoid **) &tmp);

/* set attribute server context in the service context */
(void) OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                  (dvoid *) srvhp, (ub4) 0,
                  (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

/* get the user handle */
(void) OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp,
                    (ub4)OCI_HTYPE_SESSION, 0, (dvoid **)0);

/* set the attribute user name */
(void) OCIAttrSet((dvoid *) usrhp, (ub4)OCI_HTYPE_SESSION,
                  (dvoid *)"scott", (ub4)strlen("scott"),
                  (ub4)OCI_ATTR_USERNAME, errhp);

/* set the attribute password */

```

```
(void) OCIAttrSet((dvoid *) usrhp, (ub4)OCI_HTYPE_SESSION,
                 (dvoid *)"tiger", (ub4)strlen("tiger"),
                 (ub4)OCI_ATTR_PASSWORD, errhp);

/* authenticate */
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
                                OCI_DEFAULT));

/* set the attribute user context of the service handle */
(void) OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
                 (dvoid *)usrhp, (ub4)0,
                 (ub4)OCI_ATTR_SESSION, errhp);

/* get the statement handle */
checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                               (ub4) OCI_HTYPE_STMT, 50, (dvoid **) &tmp));

(void) printf("\n*****\n");
(void) printf("--- Setup the schema and insert the data.\n");
setup(envhp, svchp, stmthp, errhp);

(void) printf("\n*****\n");
(void) printf("--- Select a REF, pin the REF, then display the object.\n");
select_pin_display(envhp, svchp, stmthp, errhp);

(void) printf("\n*****\n");
(void) printf("--- Clean up the schema and the data.\n");
cleanup(envhp, svchp, stmthp, errhp);

checkerr(errhp, OCISessionEnd (svchp, errhp, usrhp, OCI_DEFAULT));

/* detach */
(void) OCIServerDetach( srvhp, errhp, (ub4) OCI_DEFAULT );
checkerr(errhp, OCIHandleFree((dvoid *) stmthp, (ub4) OCI_HTYPE_STMT));
checkerr(errhp, OCIHandleFree((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX));
checkerr(errhp, OCIHandleFree((dvoid *) errhp, (ub4) OCI_HTYPE_ERROR));
checkerr(errhp, OCIHandleFree((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER));

return (0);
}
```

OCI Function Server Roundtrips

This appendix provides information about server roundtrips incurred during various OCI calls. This information can be useful to programmers when determining the most efficient way to accomplish a particular task in an application.

The appendix contains the following sections:

- Overview
- LOB Function Roundtrips
- Object and Cache Function Roundtrips
- Describe Operation Roundtrips
- Datatype Mapping and Manipulation Function Roundtrips
- Other Local Functions

Overview

This appendix provides information about server roundtrips incurred during various OCI calls. This information can be useful when determining the most efficient way to accomplish a particular task in an application.

LOB Function Roundtrips

Table E-1 lists the server roundtrips incurred by the *OCILob*()* calls. Information about the read and write calls is listed after the table.

Table E-1 Server Roundtrips for *OCILob*()* Calls

Function	# of Server Roundtrips
OCILobAppend()	1
OCILobAssign()	0
OCILobCharSetForm()	0
OCILobCharSetId()	0
OCILobCopy()	1
OCILobDisableBuffering()	0
OCILobEnableBuffering()	0
OCILobErase()	1
OCILobFileClose()	1
OCILobFileCloseAll()	1
OCILobFileExists()	1
OCILobFileGetName()	0
OCILobFileIsOpen()	1
OCILobFileOpen()	1
OCILobFileSetName()	0
OCILobFlushBuffer()	1 per modified page in the buffer for this LOB
OCILobGetLength()	1
OCILobIsEqual()	0
OCILobLoadFromFile()	1
OCILobLocatorIsInit()	0
OCILobTrim()	1

OCILobRead()

The number of roundtrips required depends on how the call is used:

- In polling mode without callbacks, 1 roundtrip required per *OCILobRead()* call.
- In polling mode with callbacks, 1 roundtrip is required, and then the callback function is called until all data is read.
- If data is read in one piece using the input buffer, 1 roundtrip is required.

OCILobWrite()

The number of roundtrips required depends on how the call is used:

- In polling mode without callbacks, 1 roundtrip required per *OCILobWrite()* call.
- In polling mode with callbacks, 1 roundtrip is required, and then the callback function is called until all data is written.
- If data is written in one piece using the input buffer, 1 roundtrip is required.

Object and Cache Function Roundtrips

Table E-2 lists the number of server round trips required for the object and cache functions. These values assume the cache is in a “warm” state, meaning that the type descriptor objects required by the application have been loaded.

Table E-2 Server Roundtrips for Object and Cache Functions

Function	# of Server Roundtrips
OCIObjectNew()	0
OCIObjectPin()	1; 0 if the desired object is already in cache
OCIObjectUnpin()	0
OCIObjectPinCountReset()	0
OCIObjectLock()	1
OCIObjectMarkUpdate()	0
OCIObjectUnmark()	0
OCIObjectUnmarkByRef()	0
OCIObjectFree()	0
OCIObjectMarkDelete()	0
OCIObjectMarkDeleteByRef()	0
OCIObjectFlush()	1
OCIObjectRefresh()	1
OCIObjectCopy()	0
OCIObjectGetTypeRef()	0
OCIObjectGetObjectRef()	0
OCIObjectGetInd()	0
OCIObjectExists()	0
OCIObjectIsLocked()	0
OCIObjectIsDirty()	0
OCIObjectPinTable()	1
OCIObjectArrayPin()	1
OCICacheFlush()	1
OCICacheRefresh()	1
OCICacheUnpin()	0
OCICacheFree()	0
OCICacheUnmark()	0

Describe Operation Roundtrips

The number of server round trips required by *OCIDescribeAny()*, *OCIAttrGet()*, and *OCIParamGet()* are listed in Table E-3:

Table E-3 Server Roundtrips for Describe Operations

Function	# of Server Roundtrips
OCIDescribeAny()	1 roundtrip to get the REF of the type descriptor object
OCIAttrGet()	<p>2 roundtrips to describe a type if the type objects are not in the object cache</p> <p>1 roundtrip for each collection element, or each type attribute, method, or method argument descriptor. 1 more roundtrip if using OCI_ATTR_TYPE_NAME, or OCI_ATTR_SCHEMA_NAME on the collection element, type attribute, or method argument.</p> <p>0 if all the type objects to be described are already in the object cache following the first OCIAttrGet() call.</p>
OCIParamGet()	0

Datatype Mapping and Manipulation Function Roundtrips

The number of round trips for the datatype mapping and manipulation functions are listed in Table E-4. The asterisks in the table indicate that all functions with a particular prefix incur the same number of server roundtrips. For example, *OCINumberAdd()*, *OCINumberPower()*, and *OCINumberFromText()* all incur zero server roundtrips.

Table E-4 Server Roundtrips for Datatype Manipulation Functions

Function	# of Server Roundtrips
OCINumber*()	0
OCIDate*()	0
OCIStrng*()	0
OCIRaw*()	0
OCISRef*()	0
OCIColl*()	0; 1 if the collection is not loaded in the cache
OCITable*()	0; 1 if the nested table is not loaded in the cache
OCIIter*()	0; 1 if the collection is not loaded in the cache

Other Local Functions

The following functions are local and do not require a server roundtrip:

Table E-5 Locally Processed Functions

Local Function Name	Notes
OCIAttrGet()	
OCIAttrSet()	
OCIBindByName()	
OCIBindByPos()	
OCIBindDynamic()	
OCIBindObject()	
OCIBindArrayOfStruct()	
OCIDefineByPos()	
OCIDefineDynamic()	
OCIDefineArrayOfStruct()	
OCIDefineObject()	
OCIDescriptorAlloc()	
OCIDescriptorFree()	
OCIEnvInit()	
OCIErrorGet()	
OCIHandleAlloc()	
OCIHandleFree()	
OCILdaToSvcCtx()	
OCISvcCtxToLda()	
OCIStmtGetBindInfo()	
OCIStmtPrepare()	
OCIStmtGetBindInfo()	
OCIStmtPrepare()	
OCIStmtFetch()	may be local if retrieving pre-fetched rows

Oracle8 OCI New Features

This chapter provides a detailed overview of the new features of the Oracle8 OCI. This information supplements that which is contained in Chapter 1, “Introduction and New Features”. This chapter includes the following sections:

- Introduction
- Oracle8 OCI Enhancements
- Benefits of the OCI's New Features

Introduction

The Oracle Call Interface (OCI) is an application programming interface (API) that allows an application developer to use a third-generation language's native procedures or function calls to access the Oracle database server and control all phases of SQL statement execution. The OCI provides a library of standard database access and retrieval functions in the form of a dynamic runtime library, OCILIB, that can be linked in by the application. This eliminates the need to embed SQL or PL/SQL within 3GL programs. The OCI supports the datatypes, calling conventions, syntax and semantics of a number of third-generation languages including C, C++, COBOL and FORTRAN. Oracle is also planning to provide support for Java.

The Oracle Call Interface offers programmers the following key benefits:

- It provides the greatest degree of control over program execution.
- It allows them to use familiar 3GL programming techniques and application development tools such as browsers and debuggers.
- It supports dynamic SQL (method 4).
- It is available on the broadest range of platforms of all the Oracle Programmatic Interfaces.

Oracle8 OCI Enhancements

The Oracle8 OCI has many new features that can be broadly categorized in two primary areas:

- Encapsulated/Opaque Interfaces
- Simplified user authentication and password management
- Extensions to improve application performance and scalability
- Consistent interface for transaction management
- OCI extensions to support client-side access to Oracle8 objects
- OCI support for Oracle Advanced Queueing

Encapsulated/Opaque Interfaces

All the data structures that are used by Oracle8 OCI are encapsulated in the form of opaque interfaces that are called handles. A handle is an opaque pointer to a storage area allocated by the OCILIB that stores context information, connection

information, error information, or bind information about a SQL or PL/SQL statement. A client allocates a certain type of handle, populates one or more of those handles through well-defined interfaces, and sends requests to the server using those handles. In turn, applications can access the specific information contained in the handle by using accessor functions. The Oracle8 OCI library manages a hierarchy of handles. Encapsulating the OCI interfaces using these handles has several benefits to the application developer including:

- Reduction in the amount of server side state information that needs to be retained thereby reducing server side memory usage.
- Improved application developer productivity by eliminating the need for global variables, making error reporting easier and providing consistency in the way OCI variables are accessed and used.
- Further, the encapsulation of OCI structures in the form of handles makes them opaque to the application developer allowing changes to be made to the underlying structure without affecting applications.

Simplified User Authentication and Password Management

The Oracle8 OCI provides application developers simplified user authentication and password management in two ways: (i) it provides the ability for a single OCI application to authenticate and maintain multiple users, and (ii) Allows the application to update a user's password which is particularly helpful if an expired password message is returned by an authentication attempt.

The Oracle8 OCI supports two types of login sessions:

- It provides a simplified login function for sessions where a single user connects to the database using a login name and password.
- It supports a setup in which a single OCI application authenticates and maintains multiple sessions by separating the login session (the session created when a user logs into an Oracle database) from the user sessions (all other sessions created by a user). This is an important difference from Oracle 7.3, in which sessions could be created implicitly by starting new transactions once the user has logged in to the database, a process called *session cloning*. These "user" sessions in Oracle 7.3 inherited the privileges and security context from the login session. Oracle8 OCI requires a client to provide all the necessary authentication information for each user session. This allows an OCI application to support multiple users.

Extensions to Improve Application Performance and Scalability

The Oracle8 OCI has several enhancements to improve application performance and scalability. Application performance has been improved by reducing the number of client to server round trips required and scalability improvements have been facilitated by reducing the amount of state information that needs to be retained on the server side. Some of these features include:

- Increased client-side processing, and reduced server-side requirements
- Implicit prefetching of SELECT statement result sets to eliminate the describe round trip
- Elimination of open and close cursor round trips
- Improved support for multi-threaded environments

Consistent Interface for Transaction Management

The Oracle8 OCI supports several improvements to provide a single unified interface for transaction management in a variety of configurations. Some of the major improvements are:

- Consistent support for a variety of configurations including standard 2-tier client-server configurations, server-to-server transaction coordination, and 3-tier TP-monitor configurations
- Consistent support for local and global transactions including support for the XA interface's TM_JOIN operation
- Improved scalability by providing the ability to concentrate connections, processes, and sessions across users on dblink connections and eliminating the need for separate sessions to be created for each branch of a global transaction
- Allowing clients to authenticate different users and allow transactions to be started on their behalf

Oracle8 OCI Object Support

The Oracle8 OCI provides the most comprehensive application programming interface for programmers seeking to use the Oracle8 server's object capabilities. These features can be divided into five major categories:

- Client-side Object Cache
- Runtime environment for objects
- Associative and navigational interfaces to access and manipulate objects

- Type management functions to access information about object types in an Oracle database
- Type mapping and manipulation functions for manipulating data attributes of Oracle8 types
- Object Type Translator utility, which maps internal Oracle8 schema information to client-side language bind variables

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 instances which have been fetched by an OCI application from the server to the client side. The object cache is created when the OCI environment is initialized. Multiple applications running against the same server will each have their own object cache. The cache tracks the objects which are currently in memory, maintains references to objects, manages automatic object swapping and tracks the meta-attributes or type information about objects. The cache provides the following OCI applications:

- Improved application performance by reducing the number of client-to-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

Associative and Navigational Interfaces

Applications using the Oracle8 OCI can access objects in the Oracle8 server through two types of interfaces - (i) Using SQL SELECT, INSERT, and UPDATE statements and (ii) Using a C-style “pointer chasing” scheme to access objects in the client-side cache by traversing the corresponding smart pointers or REFs

- The Oracle8 OCI provides a set of functions with extensions to support object manipulation using SQL SELECT, INSERT, and UPDATE statements.
- To access Oracle8 objects these SQL statements use a consistent set of steps as if they were accessing relational tables.
- The Oracle8 OCI provides the following four sets of functions required to access objects using SQL statements:
 - Binding/defining object type instances and references as input/output variables of SQL statements
 - Executing SQL statements that contain object type instances and references

- Fetching object type instances and references
- Describing a select-list item of an Oracle8 object type
- The Oracle8 OCI also provides a set of functions using a C-style “pointer chasing” scheme to access objects once they have been fetched into the client-side cache by traversing the corresponding smart pointers or REFs. This “navigational interface” provides functions for:
 - Instantiating a copy of a referenceable persistent object, that is, of a persistent object with object ID in the client-side cache by “pinning” its smart pointer or REF.
 - Traversing a sequence of objects that are “connected” to each other by traversing the REFs that point from one to the other.
 - Dynamically getting and setting values of an object’s attributes.

Runtime Environment for Objects

The Oracle8 OCI provides a runtime environment for objects that offers a set of functions for managing how Oracle8 objects are used on the client-side. These functions provide the necessary functionality for:

- Connecting to an Oracle8 server in order to access its object functionality including initializing a session, logging on to a database server, and registering a connection.
- Setting up the client-side object cache and tuning its parameters.
- Getting errors and warning messages.
- Controlling transactions that access objects in the server.
- Associatively accessing objects through SQL.
- Describing a PL/SQL procedure or function whose parameters or result are of Oracle type system types.

Type Management, Mapping and Manipulation Functions

The Oracle8 OCI provides two sets of functions to work with Oracle8 objects:

- Type Mapping functions allow applications to map attributes of an Oracle8 schema which are represented in the server as internal Oracle8 datatypes such as Oracle’s number, date and string types to their corresponding host language types such as integer, months and days.

- Type Manipulation functions allow host language applications to manipulate individual attributes of an Oracle8 schema such as setting/getting their values and flushing their values to the server.

Additionally, the *OCIDescribeAny()* function can provide information about objects stored in the database.

Object Type Translator

The Object Type Translator (OTT) utility translates schema information about Oracle8 object types into client-side language bindings. That is, the Oracle8 OTT translates type information into declarations of host language variables (structures and classes). The OTT takes an “intype” file which contains metadata information about Oracle8 schema objects (an Oracle8 data dictionary) and generates an “outtype” file and the necessary header/implementation files that must be included in a C application that runs against the object schema. Both OCI applications (and Pro*C precompiler) applications may include code generated by the OTT. The OTT has many benefits including:

Improves application developer productivity: OTT eliminates the need for application developers to write by hand the host language variables that correspond to schema objects.

Maintains SQL as the data-definition language of choice: By providing the ability to automatically map Oracle8 schema objects that are created using SQL to host language variables automatically, OTT facilitates using SQL as the data-definition language of choice. This in turn allows Oracle8 to support a consistent model of the user’s data, enterprise-wide.

Facilitates schema evolution of object types: OTT provides the ability to regenerate *#include* files when the schema is changed allowing Oracle8 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. With Oracle8, OTT can only generate C structs which can either be used with OCI programs or with the Pro*C precompiler programs.

OCI Support for Oracle Advanced Queueing

The OCI provides an interface to Oracle8’s Advanced Queueing feature. Oracle AQ (Oracle Advanced Queueing) provides message queuing as an integrated part of the Oracle server. Oracle AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database*. By providing

an integrated solution Oracle AQ frees application developers to devote their efforts to their specific business logic rather than having to construct a messaging infrastructure.

For more information about the OCI advanced queueing features, refer to “OCI and Advanced Queueing” on page 7-40.

Benefits of the OCI's New Features

The enhancements to the new OCI provide several benefits:

- Comprehensive support for Oracle8 objects
- Improved application performance
- Greater scalability; Enhanced application extensibility
- Simplified migration of existing applications.

Each of these benefits is described below.

Comprehensive Support for Oracle8 Objects

As has been described above, the OCI provides the most comprehensive support for Oracle8 objects of all the programmatic interfaces and provides the most highly tunable interface to access, modify and manipulate Oracle8 object types on the client side. Further, the many tools and features of the OCI significantly enhance developer productivity when creating applications that use Oracle8 objects.

Improved Application Performance

The Oracle8 OCI facilitates improved application performance by reducing the number of client-to-server round trips in three ways.

- Since Oracle8 OCI does not fundamentally work around the concept of cursors, no calls are required to open and close cursors.
- The describe round trip is eliminated due to Oracle8 OCI's ability to implicitly prefetch SELECT statement result sets.
- Oracle8 OCI also reduces the number of client-server round trips when working with Oracle8 objects - (i) The Oracle8 OCI's use of a client-side cache allows applications to update multiple objects from the client to/from the server in a single round trip using a flush or refresh operation; (ii) Oracle8 OCI's complex object retrieval mechanism provides a transparent but configurable approach to prefetching connected objects from the server in a single round trip.

Greater Scalability

Applications written to use Oracle8 OCI will have greater scalability due to the Oracle8 OCI's reduced use of server side memory, its ability to pool concurrent transactions, and its improved support for multi-threaded environments.

- Oracle8 OCI's use of handles enables it to carry out more client-side processing and as a result reduce significantly the amount of state information that needs to be retained on the server. As a result, server side memory usage is significantly reduced and applications, therefore, scale better.
- The Oracle8 OCI allows multiple concurrent transactions to be pooled on a single connection to the server. This substantially reduces the number of connections required between the client and the server. As a result, three tier architectures that use Oracle8 OCI scale well.

Simplified Migration of Existing Applications

The OCI has been significantly improved with many features, and applications written to work with Oracle7 OCI have a very smooth migration path to Oracle8 OCI due to the interoperability of the Oracle7 OCI (version7 client) with Oracle8 (server) and Oracle8 OCI (version 8 client) with Oracle7 (server). Specifically:

- Applications that use Oracle7 OCI work unchanged against the Oracle8 server (all release 7.3 OCI function calls work against the Oracle8 server).
- Applications that use Oracle8 OCI work against an Oracle7 server provided they do not use any of the object capabilities of the OCI or the server.
- Oracle7 OCI and Oracle8 OCI calls can be mixed in the same application and in the same transaction provided they are not mixed within the statement.

As a result, customers migrating an existing Oracle7 OCI application have the following three alternatives:

- ***Retain Oracle7 OCI client:*** Customers can retain their Oracle7 OCI applications without making any modifications - they will continue to work against an Oracle8 server.
- ***Upgrade to Oracle8 OCI client but do not modify application:*** Customers who choose to upgrade from a Oracle7 OCI client to Oracle8 OCI client need only relink the new version of OCILIB and need NOT recompile their application. Relinked Oracle7 OCI applications work unchanged against an Oracle8 server.
- ***Upgrade to Oracle8 OCI client and modify application:*** To avail themselves of the performance and scalability benefits provided by the new OCI, however,

customers will need to modify their existing applications to use the new OCI calls, relink them with the new OCILIB and run them against an Oracle8 server.

Further, if application developers need to use any of the object capabilities of the Oracle8 server, they will need to upgrade their client to use Oracle8 OCI.

Enhanced Application Extensibility

All the data structures that are used by Oracle8 OCI are encapsulated in the form of opaque interfaces that are called handles. This encapsulation of the OCI's interfaces allows changes to be made to the underlying data structures without affecting applications. For example, some of the services that are currently provided by the database and externalized through the OCI's APIs could in the future be provided by an application server. By using the OCI's opaque handles applications will not need to change significantly if accessing these services from the application server - this facilitates application extensibility.

Index

A

aborting OCI calls, 2-31
ADO. See attribute descriptor object
advanced queueing
 dequeue function, 13-8
 description, 7-40
 enqueue function, 13-11
 examples, 13-12
 OCI and, 7-40
 OCI descriptors for, 7-40
 OCI functions for, 7-40
 OCI vs. PL/SQL, 7-41
allocation duration
 example, 11-13
 of objects, 11-13
application failover
 callback example, 7-38
 callback registration, 7-37
 OCI callbacks, 7-36
applications
 linking, 2-32
AQ. See advanced queueing.
arguments
 attributes, 6-17
array binds, 10-3
array defines, 10-6
arrays
 skip parameter for, 5-19
arrays of structures, 5-17
 indicator variables, 5-20
 OCI calls used, 5-20
 skip parameters, 5-18
atomic nullness, 8-28

attribute descriptor object, 9-23
attributes
 of handles, 2-11
 of objects, 8-17
 of parameter descriptors, 6-5
 of parameters, 6-5
authentication
 of user, 7-11

B

BFILE
 datatype, 3-20
BFILE datatype, 3-20
bind handle
 attributes, B-19
 description, 2-10
bind operation, 4-5, 5-2, 10-2
 associations made, 5-3
 example, 5-6
 LOBs, 5-10
 named data types, 5-10, 10-2
 named vs. positional, 5-4
 OCI array interface, 5-4
 OCI_DATA_AT_EXEC mode, 5-11
 PL/SQL, 5-5
 positional vs. named, 5-4
 ref cursor variables, 5-12
 REFs, 5-10, 10-3
 static arrays, 5-10
 steps used, 5-6
binding
 arrays, 10-3
 OCINumber, 10-8

- PL/SQL placeholders, 2-32
- summary, 5-12
- BLOB
 - datatype, 3-21
- BLOB datatype, 3-21
- branches
 - detaching, 7-7
 - resuming, 7-7
- buffering LOB operations, 7-28

C

- C datatypes
 - manipulating with OCI, 9-5
- cache functions
 - server roundtrips, E-4
- callback registration
 - application failover, 7-37
- callbacks
 - application failover, 7-36
 - for LOB operations, 7-31
 - for reading LOBs, 7-32
 - for writing LOBs, 7-34
 - from external procedures, 7-35
 - LOB streaming interface, 7-31
 - parameter modes, 13-52
- canceling OCI calls, 2-31
- CASE OTT parameter, 12-27
- CHAR
 - external datatype, 3-16
- character set form, 5-25
- character set ID, 5-25
- CHARZ
 - external datatype, 3-17
- checkerr() function
 - code listing, 2-25
- CLOB
 - datatype, 3-21
- CLOB datatype, 3-21
- CODE OTT parameter, 12-26
- coherency
 - of object cache, 11-4
- collections
 - attributes, 6-13
 - describing, 6-2

- columns
 - attributes, 6-15
- commit, 2-23
 - in object applications, 11-13
 - one-phase for global transactions, 7-8
 - two-phase for global transactions, 7-8
- complex object retrieval, 8-21
 - implementing, 8-23
 - navigational prefetching, 8-25
- complex object retrieval (COR) descriptor, 2-15
 - attributes, B-27
- complex object retrieval (COR) handle, 2-10
 - attributes, B-26
- CONFIG OTT parameter, 12-27
- configuration files
 - and the OTT, 12-5
- consistency
 - of object cache, 11-4
- copying objects, 8-31
- COR, see complex object retrieval
- creating objects, 8-31

D

- data definition language
 - SQL statements, 1-4
- data manipulation language
 - SQL statements, 1-5
- data structures
 - new for 8.0, 2-5
- database connection
 - for object applications, 8-10
- datatype
 - conversions, 3-22
 - external, 3-4, 3-7
 - internal, 3-4, 3-5
 - Oracle, 3-2
- datatype code
 - internal, 3-5
- datatype mapping
 - Oracle methodology, 9-5
- datatype mapping and manipulation functions
 - server roundtrips, E-6
- datatype mappings, 12-9
- datatypes

- BFILE, 3-20
- BLOB, 3-21
- CLOB, 3-21
- FILE, 3-20
- for piecewise operations, 7-17
- manipulating with OCI, 9-5
- mapping from Oracle to C, 9-3
- NCLOB, 3-21
- DATE
 - external datatype, 3-14
- DDL. See data definition language
- default file name extensions, 12-35
- default name mapping, 12-35
- define
 - arrays, 10-6
- define handle
 - attributes, B-22
 - description, 2-10
- define operation, 4-11, 5-13, 10-4
 - example, 5-14
 - LOBs, 5-16
 - named data types, 5-16, 10-4
 - piecewise fetch, 5-17
 - PL/SQL output variables, 5-17
 - REFs, 5-16, 10-4
 - static arrays, 5-17
 - steps used, 5-14
- defining
 - OCINumber, 10-8
- deletes
 - positioned, 2-31
- describe
 - object describe code example, D-55
 - of collections, 6-2
 - of packages, 6-2
 - of sequences, 6-2
 - of stored functions, 6-2
 - of stored procedures, 6-2
 - of synonyms, 6-2
 - of tables, 6-2
 - of types, 6-2
 - of views, 6-2
- describe handle
 - attributes, B-24
 - description, 2-10

- describe operation, 4-8
 - implicit, 4-9
 - server roundtrips, E-5
- descriptor, 2-12
 - complex object retrieval, 2-15
 - parameter, 2-14
 - ROWID, 2-15
 - snapshot, 2-13
- descriptor objects, 9-23
- descriptors
 - allocating, 2-18
- detaching branches, 7-7
- DML. See data manipulation language
- DML with RETURNING clause
 - See RETURNING clause
- duration
 - of objects, 11-13
- durations
 - example, 11-13

E

- embedded objects
 - fetching, 8-15
- embedded SQL, 1-7
 - mixing with OCI calls, 1-7
- encapsulated interfaces, F-2
- environment handle
 - attributes, B-3
 - description, 2-8
- error codes
 - navigational functions, 14-5
- error handle
 - description, 2-8
- error handling
 - example, 2-25
- errors
 - handling, 2-25
 - handling in object applications, 8-32
- ERRTYPE OTT parameter, 12-27
- executing SQL statements, 4-6
- execution
 - against multiple servers, 4-5
 - modes, 4-7
- execution snapshots, 4-7

- extensions
 - default file name, 12-35
- external datatype, 3-4
 - CHAR, 3-16
 - CHARZ, 3-17
 - DATE, 3-14
 - FLOAT, 3-11
 - INTEGER, 3-11
 - LOBs, 3-19
 - LONG, 3-13
 - LONG RAW, 3-15
 - LONG VARCHAR, 3-16
 - LONG VARRAW, 3-16
 - MLSLABEL, 3-18
 - named data types, 3-18
 - NUMBER, 3-10
 - RAW, 3-15
 - REF, 3-19
 - ROWID, 3-14
 - SQLT_BLOB, 3-19
 - SQLT_CLOB
 - external datatype
 - SQLT_NCLOB, 3-19
 - SQLT_NTY, 3-18
 - SQLT_REF, 3-19
 - STRING, 3-12
 - UNSIGNED, 3-16
 - VARCHAR, 3-13
 - VARCHAR2, 3-9
 - VARNUM, 3-13
 - VARRAW, 3-15
- external datatypes, 3-7
 - conversions, 3-22
- external procedure functions
 - return codes, 16-2
 - with_context type, 16-2
- external procedures
 - OCI callbacks, 7-35

F

- fetch
 - piecewise, 7-16, 7-20
- fetch operation, 4-12
 - LOB data, 4-12

- setting prefetch count, 4-12
- FILE
 - associating with OS file, 7-27
 - datatype, 3-20
 - locator, 7-26
- FLOAT
 - external datatype, 3-11
- flushing, 11-10
 - object changes, 8-14
- flushing objects, 11-10
- freeing objects, 8-31, 11-8
- functions
 - attributes, 6-7

G

- global transactions, 7-4
- GTRID. See transaction identifier

H

- handle attributes, 2-11
 - reading, 2-11
 - setting, 2-11
- handles, 2-6
 - advantages of, 2-8
 - allocating, 2-7, 2-18
 - bind handle, 2-10
 - C datatypes, 2-6
 - child freed when parent freed, 2-8
 - define handle, 2-10
 - describe handle, 2-10
 - environment handle, 2-8
 - error handle, 2-8
 - freeing, 2-7
 - hierarchy of, 2-7
 - server handle, 2-9
 - service context handle, 2-8
 - statement handle, 2-10
 - transaction handle, 2-9
 - types, 2-6
 - user session handle, 2-9
- HFILE OTT parameter, 12-26

I

- indicator variable, 2-29
 - arrays of structures, 5-20
 - for named data types, 2-28
 - for REF, 2-28
 - named data type defines, 10-5
 - PL/SQL OUT binds, 10-5
 - REF defines, 10-5
 - with named data type bind, 10-3
 - with REF bind, 10-3
- indicator variables
 - for named data types, 2-30
 - for REFs, 2-30
- INITFILE OTT parameter, 12-26
- INITFUNC OTT parameter, 12-26
- initialization function
 - calling, 12-20
 - tasks of, 12-22
- insert
 - piecewise, 7-16, 7-18
- INTEGER
 - external datatype, 3-11
- internal datatype, 3-4, 3-5
 - datatype codes, 3-5
- internal datatypes
 - conversions, 3-22
- intype file, 12-29
 - providing when running OTT, 12-8
 - structure of, 12-29
- INTYPE OTT parameter, 12-25

K

- keywords, C-2

L

- linking, 2-32
 - issues, A-7
 - modes, A-7
 - support for single-task, A-9
- lists
 - attributes, 6-19
- LOB, 7-24
 - binding, 5-10

- creating, 7-26
- defining, 5-16
- external data type, 3-19
- fetching data, 4-12
- locator, 2-13
- modifying, 7-26
- OCI functions, 7-28
- OCI operations on, 7-24
- LOB attributes
 - of transient objects, 7-27
- LOB buffering, 7-28
 - code example, D-96
- LOB functions
 - server roundtrips
- LOB locator, 2-13, 7-24
 - attributes, B-25
- LOB operations
 - buffering, 7-28
 - callbacks, 7-31
 - code example, D-76
- locator, 2-12
 - for LOB datatype, 2-13, 7-24
- locking, 11-12
- locking objects, 11-12
- LONG
 - external datatype, 3-13
- LONG RAW
 - external datatype, 3-15
- LONG VARCHAR
 - external datatype, 3-16
- LONG VARRAW
 - external datatype, 3-16

M

- marking objects, 11-9
- MDO. See method descriptor object
- meta-attributes
 - of objects, 8-17
 - of persistent objects, 8-17
 - of transient objects, 8-20
- method descriptor object, 9-23
- migration
 - 7.x to 8.0, A-4
- MLSLABEL

- external datatype, 3-18
- multiple servers
 - executing statement against, 4-5
- multi-threaded development
 - basic concepts, 7-14

N

- named data type
 - binding and defining, 10-6
 - indicator variable for, 2-28
- named data types
 - binding, 5-10, 10-2
 - defining, 5-16, 10-4
 - definition, 3-18
 - external data types, 3-18
 - indicator variables, 2-30
- namespaces
 - reserved, C-11
- navigation, 11-16
- navigational functions
 - error codes, 14-5
 - return values, 14-4
 - terminology, 14-4
- NCHAR
 - issues, 5-25
- NCLOB
 - datatype, 3-21
- NCLOB datatype, 3-21
- nested table
 - element ordering, 9-21
- new features, F-2
 - benefits, F-8
 - enhancements, F-2
 - introduction, F-2
- no-op
 - definition, 14-57
- null indicator struct
 - generated by OTT, 8-9
- null undicator struct, 8-28
- nullness
 - atomic, 8-28
 - of objects, 8-28
- NULLs
 - detecting, 2-30

- inserting, 2-29
- inserting into database, 2-28
- inserting using indicator variables, 2-28

NUMBER

- external datatype, 3-10

O

- object
 - allocation duration, 11-13
 - array pin, 8-13
 - attributes
 - manipulating, 8-13
 - duration, 11-13
 - LOB attribute of, 7-27
 - memory layout of instance, 11-15
 - nullness, 8-28
 - pin count, 8-28
 - pin duration, 11-13
 - pinning, 8-12
 - secondary memory, 11-15
 - top-level memory, 11-15
 - unpinning, 8-28
- object application
 - database connection, 8-10
- object applications
 - commit, 11-13
 - rollback, 11-13
- object cache, 11-2
 - coherency, 11-4
 - consistency, 11-4
 - initializing, 8-10
 - loading objects, 11-6
 - memory parameters, 11-5
 - operations on, 11-6
 - removing objects, 11-6
 - setting the size of, 11-5
- object functions
 - See navigational functions.
 - server roundtrips, E-4
- object identifier
 - for persistent objects, 8-5
- object reference, 8-32
- object reference. See REF
- object retrieval

- code example, D-11
- object runtime environment
 - initializing, 8-10
- object type translator
 - sample output, 8-9
 - See OTT
 - use with OCI, 8-8
- objects
 - accessing with OCI, 12-19
 - attributes, 8-17
 - client-side cache, 11-2
 - copying, 8-31
 - creating, 8-31
 - flushing, 11-10
 - flushing changes, 8-14
 - freeing, 8-31, 11-8
 - lifetime, 14-2
 - LOB attributes of transient objects, 7-27
 - locking, 11-12
 - manipulating with OCI, 12-19
 - marking, 8-14, 11-9
 - memory management, 11-2
 - meta-attributes, 8-17
 - navigation, 11-16
 - simple, 11-16
 - OCI object application structure, 8-4
 - persistent, 8-5, 8-6
 - pinning, 11-6
 - refreshing, 11-10
 - representing in C applications, 8-8
 - terminology, 14-2
 - transient, 8-5, 8-7
 - types, 8-5, 14-2
 - unmarking, 11-10
 - unpinning, 11-8
 - use with OCI, 8-3
- obsolescent OCI functions OCI functions
 - obsolescent, A-2
- obsolete OCI functions OCI functions
 - obsolete, A-4
- OCI
 - object support, 1-8
 - overview, 1-2
 - parts of, 1-10
 - release 8.0 new features, 1-10
- OCI application
 - compiling, 1-11
 - general structure, 2-3
 - initialization example, 2-19
 - linking, 1-11
 - steps, 2-16
 - structure, 2-3
 - structure using objects, 8-4
 - terminating, 2-24
 - with objects
 - initializing, 8-10
- OCI applications
 - using the OTT with, 12-18
- OCI environment
 - initializing for objects, 8-10
- OCI functions
 - canceling calls, 2-31
 - return codes, 2-25, 2-27
- OCI navigational functions, 11-18
 - flush functions, 11-19
 - mark functions, 11-19
 - meta-attribute accessor functions, 11-19
 - miscellaneous functions, 11-20
 - naming scheme, 11-18
 - pin/unpin/free functions, 11-18
- OCI process
 - initializing, 2-17
 - initializing for objects, 8-10
 - modes, 2-17
- OCI program. See OCI application
- OCI relational functions
 - guide to reference entries, 13-7, 16-3
 - quick reference, 13-3
- OCI Release 8
 - accessing and manipulating objects, 12-19
- OCI_ATTR_ALLOC_DURATION
 - environment handle attribute, B-4
- OCI_ATTR_CACHE
 - attribute, 6-15
- OCI_ATTR_CACHE_MAX_SIZE
 - environment handle attribute, B-3
- OCI_ATTR_CACHE_OPT_SIZE
 - environment handle attribute, B-3
- OCI_ATTR_CHAR_COUNT
 - bind handle attribute, B-19

- define handle attribute, B-22
- use of, 5-26
- OCI_ATTR_CHARSET_FORM
 - attribute, 6-11, 6-14, 6-16
 - bind handle attribute, B-20
 - define handle attribute, B-23
- OCI_ATTR_CHARSET_ID
 - attribute, 6-11, 6-14, 6-16, 6-18
 - bind handle attribute, B-19
 - define handle attribute, B-22
- OCI_ATTR_CLUSTERED
 - attribute, 6-7
- OCI_ATTR_COLLECTION_ELEMENT
 - attribute, 6-10
- OCI_ATTR_COLLECTION_TYPECODE
 - attribute, 6-9
- OCI_ATTR_COMPLEXOBJECT_COLL_
 - OUTOFLINE
 - COR handle attribute, B-26
- OCI_ATTR_COMPLEXOBJECT_LEVEL
 - COR handle attribute, B-26
- OCI_ATTR_COMPLEXOBJECTCOMP_TYPE
 - COR descriptor attribute, B-27
- OCI_ATTR_COMPLEXOBJECTCOMP_
 - TYPE_LEVEL
 - COR descriptor attribute, B-27
- OCI_ATTR_DATA_SIZE
 - attribute, 6-10, 6-13, 6-15, 6-17
- OCI_ATTR_DATA_TYPE
 - attribute, 6-10, 6-13, 6-15, 6-17
- OCI_ATTR_DBA
 - attribute, 6-7
- OCI_ATTR_ENCAPSULATION
 - attribute, 6-12
- OCI_ATTR_ENV
 - server handle attribute, B-11
 - service context handle attribute, B-7
- OCI_ATTR_EXTERNAL_NAME
 - server handle attribute, B-11
- OCI_ATTR_FNCODE
 - bind handle attribute, B-19
 - define handle attribute, B-22
 - environment handle attribute, B-4
 - server handle attribute, B-11
 - statement handle attribute, B-15
- OCI_ATTR_FOCBK
 - server handle attribute, B-12
- OCI_ATTR_HAS_DEFAULT
 - attribute, 6-17
- OCI_ATTR_HAS_FILE
 - attribute, 6-9
- OCI_ATTR_HAS_LOB
 - attribute, 6-9
- OCI_ATTR_HAS_NESTED_TABLE
 - attribute, 6-9
- OCI_ATTR_HW_MARK
 - attribute, 6-15
- OCI_ATTR_IN_V8_MODE
 - server handle attribute, B-12
 - service context handle attribute, B-9
- OCI_ATTR_INCR
 - attribute, 6-15
- OCI_ATTR_INDEX_ONLY
 - attribute, 6-7
- OCI_ATTR_INTERNAL_NAME
 - server handle attribute, B-12
- OCI_ATTR_IOMODE
 - attribute, 6-18
- OCI_ATTR_IS_CONSTRUCTOR
 - attribute, 6-12
- OCI_ATTR_IS_DESTRUCTOR
 - attribute, 6-12
- OCI_ATTR_IS_INCOMPLETE_TYPE
 - attribute, 6-9
- OCI_ATTR_IS_MAP
 - attribute, 6-12
- OCI_ATTR_IS_NULL
 - attribute, 6-16, 6-18
- OCI_ATTR_IS_OPERATOR
 - attribute, 6-12
- OCI_ATTR_IS_ORDER
 - attribute, 6-12
- OCI_ATTR_IS_PREDEFINED_TYPE
 - attribute, 6-9
- OCI_ATTR_IS_RNDS
 - attribute, 6-12
- OCI_ATTR_IS_RNPS
 - attribute, 6-12
- OCI_ATTR_IS_SELFISH
 - attribute, 6-12

OCI_ATTR_IS_SYSTEM_GENERATED_TYPE	OCI_ATTR_NUM_HANDLES
attribute, 6-9	attribute, 6-19
OCI_ATTR_IS_SYSTEM_TYPE	OCI_ATTR_NUM_PARAMS
attribute, 6-9	attribute, 6-6
OCI_ATTR_IS_TRANSIENT_TYPE	OCI_ATTR_NUM_TYPE_ATTRS
attribute, 6-9	attribute, 6-10
OCI_ATTR_IS_WNDS	OCI_ATTR_NUM_TYPE_METHODS
attribute, 6-12	attribute, 6-10
OCI_ATTR_IS_WNPS	OCI_ATTR_OBJECT
attribute, 6-12	environment handle attribute, B-3
OCI_ATTR_LEVEL	OCI_ATTR_OBJID
attribute, 6-17	attribute, 6-7, 6-14, 6-15
OCI_ATTR_LINK	OCI_ATTR_ORDER
attribute, 6-14, 6-18	attribute, 6-15
OCI_ATTR_LIST_ARGUMENTS	OCI_ATTR_ORDER_METHOD
attribute, 6-7, 6-12	attribute, 6-10
OCI_ATTR_LIST_COLUMNS	OCI_ATTR_OVERLOAD
attribute, 6-7	attribute, 6-8
OCI_ATTR_LIST_SUBPROGRAMS	OCI_ATTR_PARAM_COUNT
attribute, 6-8	describe handle attribute, B-24
OCI_ATTR_LIST_TYPE	statement handle attribute, B-17
attribute, 6-19	OCI_ATTR_PARTITIONED
OCI_ATTR_LIST_TYPE_ATTRS	attribute, 6-7
attribute, 6-10	OCI_ATTR_PASSWORD
OCI_ATTR_LIST_TYPE_METHODS	user session handle attribute, B-13
attribute, 6-10	OCI_ATTR_PDFMT
OCI_ATTR_LOBEMPTY	bind handle attribute, B-21
LOB locator attribute, B-25	define handle attribute, B-23
OCI_ATTR_MAP_METHOD	OCI_ATTR_PDSCL
attribute, 6-10	bind handle attribute, B-20
OCI_ATTR_MAX	define handle attribute, B-23
attribute, 6-15	OCI_ATTR_PIN_DURATION
OCI_ATTR_MAXDATA_SIZE	environment handle attribute, B-6
bind handle attribute, B-20	OCI_ATTR_PINOPTION
use with binding, 5-26	environment handle attribute, B-4
OCI_ATTR_MIN	OCI_ATTR_POSITION
attribute, 6-15	attribute, 6-17
OCI_ATTR_NAME	OCI_ATTR_PRECISION
attribute, 6-8, 6-10, 6-12, 6-13, 6-14, 6-15, 6-17	attribute, 6-11, 6-13, 6-16, 6-17
OCI_ATTR_NUM_ATTRS	OCI_ATTR_PREFETCH_MEMORY
attribute, 6-6	statement handle attribute, B-18
OCI_ATTR_NUM_COLS	OCI_ATTR_PREFETCH_ROWS
attribute, 6-7	statement handle attribute, B-18
OCI_ATTR_NUM_ELEMENTS	OCI_ATTR_PTYPE
attribute, 6-13	attribute, 6-6

- OCI_ATTR_RADIX
 - attribute, 6-18
- OCI_ATTR_REF_TDO
 - attribute, 6-9, 6-11, 6-14, 6-16, 6-18
- OCI_ATTR_ROWID
 - statement handle attribute, B-17
- OCI_ATTR_ROWS_RETURNED
 - bind handle attribute, B-21
 - use with callbacks, 5-25
- OCI_ATTR_SCALE
 - attribute, 6-11, 6-13, 6-16, 6-17
- OCI_ATTR_SCHEMA
 - attribute, 6-14
- OCI_ATTR_SCHEMA_NAME
 - attribute, 6-11, 6-14, 6-16, 6-18
- OCI_ATTR_SEQ
 - attributes, 6-15
- OCI_ATTR_SERVER
 - service context handle attribute, B-7
- OCI_ATTR_SESSION
 - service context handle attribute, B-9
- OCI_ATTR_SQLCODE
 - service context handle attribute, B-7
- OCI_ATTR_STMT_TYPE
 - statement handle attribute, B-16
- OCI_ATTR_SUB_NAME
 - attribute, 6-18
- OCI_ATTR_TABLESPACE
 - attribute, 6-7
- OCI_ATTR_TIMESTAMP
 - attribute, 6-6
- OCI_ATTR_TRANS
 - service context handle attribute, B-9
- OCI_ATTR_TRANS_NAME
 - transaction handle attribute, B-14
- OCI_ATTR_TYPE_NAME
 - attribute, 6-11, 6-14, 6-16, 6-18
- OCI_ATTR_TYPECODE
 - attribute, 6-9, 6-10, 6-13, 6-17
- OCI_ATTR_USERNAME
 - user session handle attribute, B-13
- OCI_ATTR_VERSION
 - attribute, 6-9
- OCI_ATTR_XID
 - transaction handle attribute, B-14

- OCI_PTYPE_ARG
 - attributes, 6-17
- OCI_PTYPE_COL
 - attributes, 6-15
- OCI_PTYPE_COLL
 - attributes, 6-13
- OCI_PTYPE_FUNC
 - attributes, 6-7
- OCI_PTYPE_LIST
 - attributes, 6-19
- OCI_PTYPE_PKG
 - attributes, 6-8
- OCI_PTYPE_PROC
 - attributes, 6-7
- OCI_PTYPE_SYN
 - attributes, 6-14
- OCI_PTYPE_TABLE
 - attributes, 6-7
- OCI_PTYPE_TYPE
 - attributes, 6-9
- OCI_PTYPE_TYPE_ATTR
 - attributes, 6-10
- OCI_PTYPE_TYPE_FUNC
 - attributes, 6-12
- OCI_PTYPE_TYPE_PROC
 - attributes, 6-12
- OCI_PTYPE_VIEW
 - attributes, 6-7
- OCI_TYPECODE
 - values, 3-24, 3-25
- OCI_TYPECODE values, 3-24
- OCIAQDeq(), 13-8
- OCIAQEnq(), 13-11
- OCIArray, 9-17
 - binding and defining, 9-17, 10-6
- OCIArray manipulation
 - code example, 9-19
- OCIAttrGet(), 13-23
 - used for describing, 4-9
- OCIAttrSet(), 13-25
- OCIBindArrayOfStruct(), 13-28
- OCIBindByName(), 13-30
- OCIBindByPos(), 13-34
- OCIBindDynamic(), 13-38
- OCIBindObject(), 13-42

- OCICacheFlush(), 14-11
- OCICacheFree(), 14-13
- OCICacheRefresh(), 14-14
- OCICacheUnmark(), 14-16
- OCICacheUnpin(), 14-17
- OCIColl, 9-17
 - binding and defining, 9-17
- OCICollAppend(), 15-9
- OCICollAssign(), 15-11
- OCICollAssignElem(), 15-13
- OCICollGetElem(), 15-15
- OCICollMax(), 15-18
- OCICollSize(), 15-19
- OCICollTrim(), 15-21
- OCIComplexObject
 - use of, 8-23
- OCIComplexObjectComp
 - use of, 8-23
- OCIDate, 9-7
 - binding and defining, 9-7, 10-6
- OCIDate manipulation
 - code example, 9-9
- OCIDateAddDays(), 15-22
- OCIDateAddMonths(), 15-23
- OCIDateAssign(), 15-24
- OCIDateCheck(), 15-25
- OCIDateCompare(), 15-27
- OCIDateDaysBetween(), 15-28
- OCIDateFromText(), 15-29
- OCIDateGetDat(), 15-31
- OCIDateGetTime(), 15-32
- OCIDateLastDay(), 15-33
- OCIDateNextDay(), 15-34
- OCIDateSetDate(), 15-36
- OCIDateSetTime(), 15-37
- OCIDateSysDate(), 15-38
- OCIDateToText(), 15-39
- OCIDateZoneToZone(), 15-41
- OCIDefineArrayOfStruct(), 13-46
- OCIDefineByPos(), 13-48
- OCIDefineDynamic(), 13-52
- OCIDefineObject(), 13-55
- OCIDescAlloc(), 13-60
- OCIDescFree(), 13-62
- OCIDescribeAny(), 13-57
 - usage examples, 6-20
 - using, 6-2
- OCIDuration
 - use of, 11-7, 11-13
- OCIEnvInit(), 13-63
- OCIErrorGet(), 13-65
- OCIExtProcAllocCallmemory(), 16-4
- OCIExtProcGetEnv(), 16-8
- OCIExtProcRaiseExcp(), 16-5
- OCIExtProcRaiseExcpWithMsg(), 16-6
- OCIHandleAlloc(), 13-68
- OCIHandleFree(), 13-70
- OCIInd
 - use of, 8-29
- OCIInitialize(), 13-72
- OCIIter, 9-17
 - binding and defining, 9-17
 - usage example, 9-19
- OCIIterCreate(), 15-43
- OCIIterDelete(), 15-45
- OCIIterGetCurrent(), 15-46
- OCIIterInit(), 15-47
- OCIIterNext(), 15-48
- OCIIterPrev(), 15-50
- OCILdaToSvcCtx(), 13-75
- OCILobAppend(), 13-76
- OCILobAssign(), 13-78
- OCILobCharSet(), 13-80, 13-81
- OCILobCopy(), 13-82
- OCILobDisableBuffering(), 13-84
- OCILobEnableBuffering(), 13-85
- OCILobErase(), 13-86
- OCILobFileClose(), 13-88
- OCILobFileCloseAll(), 13-89
- OCILobFileExists(), 13-90
- OCILobFileIsOpen(), 13-93
- OCILobFileOpen(), 13-95
- OCILobFlushBuffer(), 13-98
- OCILobGetFile(), 13-91
- OCILobGetLength(), 13-100
- OCILobIsEqual(), 13-102
- OCILobLoadFromFile(), 13-103
- OCILobLocatorIsInit(), 13-105
- OCILobRead(), 13-107
- OCILobSetFile(), 13-96

- OCILobTrim(), 13-111
- OCILobWrite(), 13-112
- OCILockOpt
 - possible values, 14-32
- OCILogoff(), 13-116
- OCILogon(), 13-117
 - using, 2-18
- OCINumber, 9-10
 - bind example, 10-8
 - binding and defining, 9-10, 10-6
 - define example, 10-8
- OCINumber manipulation
 - code example, 8-13, 9-13
- OCINumberAbs(), 15-52
- OCINumberAdd(), 15-53
- OCINumberArcCos(), 15-54
- OCINumberArcSin(), 15-55
- OCINumberArcTan(), 15-56
- OCINumberArcTan2(), 15-57
- OCINumberAssign(), 15-58
- OCINumberCeil(), 15-59
- OCINumberCompare(), 15-60
- OCINumberCos(), 15-61
- OCINumberDiv(), 15-62
- OCINumberExp(), 15-63
- OCINumberFloor(), 15-64
- OCINumberFromInt(), 15-65
- OCINumberFromReal(), 15-67
- OCINumberFromText(), 15-68
- OCINumberHypCos(), 15-70
- OCINumberHypSin(), 15-71
- OCINumberHypTan(), 15-72
- OCINumberIntPower(), 15-73
- OCINumberIsZero(), 15-74
- OCINumberLn(), 15-75
- OCINumberLog(), 15-76
- OCINumberMod(), 15-77
- OCINumberMul(), 15-78
- OCINumberNeg(), 15-79
- OCINumberPower(), 15-80
- OCINumberRound(), 15-81
- OCINumberSetZero(), 15-82
- OCINumberSign(), 15-83
- OCINumberSin(), 15-84
- OCINumberSqrt(), 15-85
- OCINumberSub(), 15-86
- OCINumberTan(), 15-87
- OCINumberToInt(), 15-88
- OCINumberToReal(), 15-90
- OCINumberToText(), 15-91
- OCINumberTrunc(), 15-93
- OCIObjectArrayPin(), 14-18
- OCIObjectCopy(), 14-20
- OCIObjectExists(), 14-22
- OCIObjectFlush(), 14-23
- OCIObjectFree(), 14-24
- OCIObjectGetAttr(), 14-26
- OCIObjectGetInd(), 14-28
- OCIObjectGetObjectRef(), 14-29
- OCIObjectGetTypeRef(), 14-34
- OCIObjectIsDirty(), 14-35
- OCIObjectIsLocked(), 14-36
- OCIObjectLifetime
 - possible values, 14-31
- OCIObjectLock(), 14-37
- OCIObjectMarkDelete(), 14-38
- OCIObjectMarkDeleteByRef(), 14-40
- OCIObjectMarkStatus
 - possible values, 14-32
- OCIObjectMarkUpdate(), 14-41
- OCIObjectNew(), 14-43
- OCIObjectPin(), 14-46
- OCIObjectPinCountReset(), 14-49
- OCIObjectPinTable(), 14-51
- OCIObjectRefresh(), 14-53
- OCIObjectSetAttr(), 14-55
- OCIObjectUnmark(), 14-57
- OCIObjectUnmarkByRef(), 14-58
- OCIObjectUnpin(), 14-59
- OCIParmGet(), 13-119
 - used for describing, 4-9
- OCIParmSet(), 13-121
- OCIPasswordChange(), 13-123
- OCIPinOpt
 - use of, 11-7
- OCIRaw, 9-16
 - binding and defining, 9-16, 10-6
- OCIRaw manipulation
 - code example, 9-17
- OCIRawAllocSize(), 15-94

- OCIRawAssignBytes(), 15-95
- OCIRawAssignRaw(), 15-96
- OCIRawPtr(), 15-97
- OCIRawResize(), 15-98
- OCIRawSize(), 15-99
- OCISRef, 9-22
 - binding and defining, 9-22
 - usage example, 9-22
- OCISRefAssign(), 15-100
- OCISRefClear(), 15-101
- OCISRefFromHex(), 15-102
- OCISRefHexSize(), 15-104
- OCISRefIsEqual(), 15-105
- OCISRefIsNull(), 15-106
- OCISRefToHex(), 15-107
- OCISBreak()
 - use of, 2-31
- OCIServerAttach(), 13-125
- OCIServerDetach(), 13-127
- OCIServerVersion(), 13-128
- OCISessionBegin(), 13-129
- OCISessionEnd(), 13-132
- OCISstmtExecute(), 13-134
 - prefetch during, 4-6
 - use of iters parameter, 4-6
- OCISstmtFetch(), 13-137
- OCISstmtGetBind(), 13-139
- OCISstmtGetPieceInfo(), 13-141
- OCISstmtPrepare(), 13-143
 - preparing SQL statements, 4-4
- OCISstmtSetPieceInfo(), 13-145
- OCISString, 9-15
 - binding and defining, 9-15, 10-6
- OCISString manipulation
 - code example, 9-15
- OCISStringAllocSize(), 15-109
- OCISStringAssign(), 15-110
- OCISStringAssignText(), 15-111
- OCISStringPtr(), 15-112
- OCISStringResize(), 15-113
- OCISStringSize(), 15-114
- OCISvcCtxBreak(), 13-45
- OCISvcCtxToLda(), 13-147
- OCITable, 9-17
 - binding and defining, 9-17, 10-6
- OCITableDelete(), 15-115
- OCITableExists(), 15-116
- OCITableFirst(), 15-117
- OCITableLast(), 15-118
- OCITableNext(), 15-119
- OCITablePrev(), 15-121
- OCITableSize(), 15-123
- OCITransCommit(), 13-149
- OCITransDetach(), 13-152
- OCITransForget(), 13-154
- OCITransPrepare(), 13-155
- OCITransRollback(), 13-156
- OCITransStart(), 13-157
- OCIType
 - description
- OCITypeArrayByName(), 14-61
- OCITypeArrayByRef(), 14-64
- OCITypeByName(), 14-66
- OCITypeByRef(), 14-69
- OCITypeElem
 - description
- OCITypeMethod
 - description
- OID. See object identifier
- opaque interfaces, F-2
- Oracle Call Interface. See OCI
- Oracle datatypes, 3-2
 - mapping to C, 9-3
- Oracle Security Services, 7-43
- Oracle8 datatypes
 - binding and defining, 10-6
- oratypes.h
 - contents, 3-27
- ORE. See object runtime environment
- OTT
 - command line, 12-6
 - command line syntax, 12-23
 - creating types in the database, 12-4
 - datatype mappings, 12-9
 - invoking, 12-5
 - outtype file, 12-16
 - overview, 12-2
 - parameters, 12-24
 - providing an intype file, 12-8
 - reference, 12-22

- restrictions, 12-37
- using, 12-1
- OTT parameters
 - CASE, 12-27
 - CODE, 12-26
 - CONFIG, 12-27
 - ERRTYPE, 12-27
 - HFILE, 12-26
 - INITFILE, 12-26
 - INITFUNC, 12-26
 - INTYPE, 12-25
 - OUTTYPE, 12-25
 - SCHEMA_NAMES, 12-28
 - USERID, 12-24
 - where they appear, 12-28
- OTT. See object type translator
- outtype file, 12-29
 - when running OTT, 12-16
- OUTTYPE OTT parameter, 12-25

P

- packages
 - attributes, 6-8
 - describing, 6-2
- parameter descriptor, 2-14
 - attributes, B-24
- parameter descriptor object
- parameter descriptors
 - attributes, 6-5
- parameters
 - attributes, 6-5
 - modes, 13-7, 16-3
 - passing strings, 2-28
 - string length, 13-6
- password management, 7-11, 7-12
- PDO. See parameter descriptor object
- persistent objects, 8-6
 - meta-attributes, 8-17
- piecewise fetch, 7-20
- piecewise operations, 7-18
 - fetch, 7-16, 7-21
 - in PL/SQL, 7-20
 - insert, 7-16
 - update, 7-16

- valid datatypes, 7-17
- pin count, 8-28
- pin duration
 - example, 11-13
 - of objects, 11-13
- pinning, 11-6
- pinning objects, 11-6
- PL/SQL, 1-6
 - binding and defining nested tables, 5-27
 - binding and defining ref cursors, 5-27
 - binding placeholders, 2-32
 - defining output variables, 5-17
 - piecewise operations, 7-20
 - uses in OCI applications, 2-32
 - using in OCI applications, 2-32
 - using in OCI programs, 5-7
- positioned deletes, 2-31
- positioned updates, 2-31
- preface
 - Send Us Your Comments, xxvii
- prefetching
 - during OCISmtExecute(), 4-6
 - setting prefetch memory size, 4-12
 - setting row count, 4-12
- procedures
 - attributes, 6-7

Q

- query
 - explicit describe, 4-10
- query. See SQL query
- Quick reference to OCI relational functions, 13-3

R

- RAW
 - external datatype, 3-15
- REF
 - binding, 5-10, 10-3
 - defining, 5-16, 10-4
 - external data types, 3-19
 - indicator variable for, 2-28
 - retrieving from server, 8-11
- ref cursor variables

- binding, 5-12
- ref cursors
 - binding and defining, 5-27
- reference. See REF
- refreshing, 11-10
- refreshing objects, 11-10
- REFs
 - indicator variables for, 2-30
- relational functions
 - server roundtrips, E-7
- release 8.0 enhancements, F-2
- reserved namespaces, C-11
- reserved words, C-2
- resuming branches, 7-7
- return values
 - navigational functions, 14-4
- RETURNING clause
 - binding with, 5-22
 - code example, D-25
 - error handling, 5-23
 - using with OCI
 - with REFs, 5-23
- rollback, 2-23
 - in object applications, 11-13
- roundtrips
 - See server roundtrips
- ROWID
 - external data type, 3-14
 - used for positioned updates and deletes, 2-31
- ROWID descriptor, 2-15

S

- sb1
 - definition, 3-27
- sb2
 - definition, 3-27
- sb4
 - definition, 3-27
- SCHEMA_NAMES OTT parameter, 12-28
 - usage, 12-33
- secondary memory
 - of object, 11-15
- security handle, 2-10
- select-list

- describing, 4-8
- Send Us Your Comments
 - boilerplate, xxvii
- sequences
 - attributes, 6-15
 - describing, 6-2
- server handle
 - attributes, B-11
 - description, 2-9
 - setting in service context, 2-9
- server roundtrips
 - cache functions, E-4
 - datatype mapping and manipulation
 - functions, E-6
 - describe operation, E-5
 - LOB functions
 - object functions, E-4
 - relational functions, E-7
- service context handle
 - attributes, B-7
 - description, 2-8
 - elements of, 2-8
- single-task linking
 - support, A-9
- skip parameter
 - for standard arrays, 5-19
- skip parameters
 - for arrays of structures, 5-18
- snapshot descriptor, 2-13
- snapshots
 - executing against, 4-7
- SQL processing
 - code example, D-2
- SQL query
 - binding placeholders. See bind operation
 - defining output variables, 4-11, 5-13, 10-4
 - defining output variables. See define operation
 - fetching results, 4-12
 - statement type, 1-5
- SQL statements, 1-4
 - binding placeholders in, 4-5, 5-2, 10-2
 - determining type prepared, 4-4
 - executing, 4-6
 - preparing for execution, 4-4
 - processing, 4-2

- types
 - control statements, 1-5
 - data definition language, 1-4
 - data manipulation language, 1-5
 - embedded SQL, 1-7
 - PL/SQL, 1-6
 - queries, 1-5
- SQLT typecodes, 3-25
- SQLT_NTY
 - bind example, 10-13
 - define example, 10-14
 - description, 3-18
- SQLT_REF
 - definition, 3-19
 - description, 3-19
- statement handle
 - attributes, B-15
 - description, 2-10
- static arrays
 - binding, 5-10
 - defining, 5-17
- stored functions
 - describing, 6-2
- stored procedures
 - describing, 6-2
- STRING
 - external datatype, 3-12
- strings
 - passing as parameters, 2-28
- structures
 - arrays of, 5-17
- sword
 - definition, 3-27
- synonyms
 - attributes, 6-14
 - describing, 6-2

T

- tables
 - attributes, 6-7
 - describing, 6-2
- TDO
 - definition, 10-2
 - description, 9-23

- obtaining, 9-23
- type descriptor object. See TDO.
- TDO. See type descriptor object
- terminology
 - navigational functions, 14-4
 - used in this manual, 1-8
- thread safety, 7-13
 - advantages of, 7-13
 - and three-tier architectures, 7-13
 - basic concepts, 7-14
 - implementing with OCI, 7-14
 - mixing 7.x and 8.0 calls, 7-15
 - required OCI calls, 7-14
- three-tier architectures
 - and thread safety, 7-13
- top-level memory
 - of object, 11-15
- transaction handle
 - attributes, B-14
 - description, 2-9
- transaction identifier, 7-5
- transactional complexity
 - levels in OCI, 7-3
- transactions
 - committing, 2-23
 - global, 7-4
 - branch states, 7-7
 - branches, 7-5
 - one-phase commit, 7-8
 - transaction identifier, 7-5
 - two-phase commit, 7-8
 - global examples, 7-9
 - initialization parameters, 7-10
 - local, 7-4
 - OCI functions for
 - transactions, 7-3
 - read-only, 7-4
 - rolling back, 2-23
 - serializable, 7-4
- transient objects, 8-7
 - LOB attributes, 7-27
 - meta-attributes, 8-20
- type attributes
 - attributes, 6-10
- type descriptor object, 9-23

- type functions
 - attributes, 6-12
- type procedures
 - attributes, 6-12
- type reference, 8-32
- typecodes, 3-24
- types
 - attributes, 6-9
 - describing, 6-2

U

- ub1
 - definition, 3-27
- ub2
 - definition, 3-27
- ub4
 - definition, 3-27
- unmarking, 11-10
- unmarking objects, 11-10
- unpinning, 8-28, 11-8
- unpinning objects, 11-8
- UNSIGNED
 - external datatype, 3-16
- update
 - piecewise, 7-16, 7-18
- updates
 - positioned, 2-31
- upgrading
 - 7.x to 8.0, A-4
 - 7.x to 8.0 OCI, A-6
- user authentication, 7-11
- user memory
 - allocating, 2-15
- user session handle
 - attributes, B-13
 - description, 2-9
 - setting in service context, 2-9
- USERID OTT parameter, 12-24

V

- values, 8-5
 - in object applications, 8-7
- VARCHAR

- external datatype, 3-13
- VARCHAR2
 - external datatype, 3-9
- VARNUM
 - external datatype, 3-13
- VARRAW
 - external datatype, 3-15
- views
 - attributes, 6-7
 - describing, 6-2

W

- with_context
 - argument to external procedure functions, 16-2

X

- XID. See transaction identifier
- xtramem_sz parameter
 - using, 2-15

