# Pro\*C/C++ Developer's Guide





Pro\*C/C++ Developer's Guide, 23ai

F47578-06

Copyright © 1999, 2025, Oracle and/or its affiliates.

Primary Author: Jiji Thomas

Contributing Authors: Jack Melnick, Neelam Singh, Tim Smith, Paul Lane

Contributors: Bill Bailey, Subhranshu Banerjee, Julie Basu, Beethoven Chang, Michael Chiocca, Nancy Ikeda, Alex Keh, Thomas Kurian, Shiao-Yen Lin, Valarie Moore, Vidya Nagaraj, Ajay Popat, Ekkehard Rohwedder, Pamela Rothman, Gael Stevens

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

# Contents

P	r	ρí	fa	$\mathbf{c}$	Δ
	ı٠	ᄗ	ια	u	C

Intended Audience

	Documentation Accessibility	xxxiv		
	Related Documents Conventions  Changes in Pro*C/C++  New Features in Pro*C/C++ Release 23ai New Features in Pro*C/C++ Release 21c		xxxiv	
			XXXV	
			xxxvi	
			xxxvi	
	Nev	v Featı	ures in Pro*C/C++ Release 19c	xxxvi
Part		Intro	oduction and Concepts	
1	Int	rodu	ction	
	1.1	Wha	at is an Oracle Precompiler?	1-1
	1.2	Why	Use the Oracle Pro*C/C++ Precompiler	1-2
	1.3	Why	Use SQL	1-3
	1.4	Why	Use PL/SQL	1-3
	1.5	Pro*	C/C++ Precompiler Benefits	1-3
	1.6	Dire	ctory Structure	1-5
		1.6.1	Known Problems, Restrictions, and Workarounds	1-6
	1.7	Libra	ary Files	1-6
	1.8	Fred	quently Asked Questions	1-6
		1.8.1	What is a VARCHAR?	1-6
		1.8.2	Does Pro*C/C++ Generate Calls to the Oracle Call Interface?	1-7
		1.8.3	Why Not Code Using SQLLIB Calls and Not Use Pro*C/C++?	1-7
		1.8.4	Can I Call A PL/SQL Stored Procedure From a Pro*C/C++ Program?	1-7
		1.8.5	Can I Write C++ Code, and Precompile It Using Pro*C/C++?	1-7
		1.8.6	Can I Use Bind Variables Anywhere in a SQL Statement?	1-7
		1.8.7	I Am Confused By Character Handling in Pro*C/C++.	1-8
		1.8.8	Is There Anything Special About Character Pointers?	1-8



xxxiv

	1.8.9	Why Does SPOOL Not Work in Pro*C/C++?	1-8
	1.8.10	Where Can I Find The On-line Versions of the Example Programs?	1-9
	1.8.11	How Can I Compile and Link My Application?	1-9
	1.8.12	Does Pro*C/C++ Now Support Using Structures As Host Variables?	1-9
	1.8.13	Is It Possible to Have Recursive Functions In Pro*C/C++ If I Use Embedded SQL In the Function?	1-9
	1.8.14	Can I Use Any Release of Pro*C/C++ with Any Version of the Oracle Server?	1-9
	1.8.15	When My Application Runs, I Keep Getting an Ora-1405 Error (Fetched Column Value Is NULL).	1-10
	1.8.16	Are All SQLLIB Functions Private?	1-10
	1.8.17	How Does Oracle Support The New Object Types?	1-11
	1.8.18	Compatibility, Upgrading, and Migration	1-11
2	Precom	piler Concepts	
	2.1 Key	Concepts of Embedded SQL Programming	2-1
	2.1.1	Embedded SQL Statements	2-1
	2.1	1.1 Executable Statements and Directives	2-2
	2.1.2	Embedded SQL Syntax	2-3
	2.1.3	Static Versus Dynamic SQL Statements	2-3
	2.1.4	Embedded PL/SQL Blocks	2-3
	2.1.5	Host and Indicator Variables	2-4
	2.1.6	Oracle Datatypes	2-4
	2.1.7	Arrays	2-5
	2.1.8	Datatype Equivalencing	2-5
	2.1.9	Private SQL Areas, Cursors, and Active Sets	2-5
	2.1.10	Transactions	2-5
	2.1.11	Errors and Warnings	2-6
	2.1.12	SQL99 Syntax Support	2-6
	2.2 Step	s in Developing an Embedded SQL Application	2-6
	2.3 Guid	elines for Programming	2-7
	2.3.1	Comments	2-7
	2.3.2	Constants	2-8
	2.3.3	C99 Support	2-8
	2.3.4	Declare Section	2-8
	2.3.5	Delimiters	2-9
	2.3.6	File Length	2-9
	2.3.7	Function Prototyping	2-10
	2.3	3.7.1 ANSI C	2-10
		3.7.2 KR_C	2-10
		3.7.3 CPP	2-10
	2.3.8	Hint Length	2-10
	2.3.9	Host Variable Names	2-10



	2.3.10 Line Continuation	2-11
	2.3.11 Line Length	2-11
	2.3.12 MAXLITERAL Default Value	2-11
	2.3.13 Operators	2-11
	2.3.14 Statement Terminator	2-12
	2.4 Conditional Precompilation	2-12
	2.4.1 Symbol Definition	2-12
	2.4.2 Example SELECT Statement	2-12
	2.5 Precompile Separately	2-13
	2.5.1 Guidelines	2-13
	2.5.1.1 Referencing Cursors	2-13
	2.5.1.2 Specifying MAXOPENCURSORS	2-13
	2.5.1.3 Use a Single SQLCA	2-13
	2.6 Compile and Link	2-14
	2.7 Example Tables	2-14
	2.7.1 Example Data	2-14
	2.8 Example Program: A Simple Query	2-15
	2.9 Example Program: A Simple Query using SQL99 Syntax	2-17
3	Database Concepts  3.1 Connect to the Database	3-1
		3-1 3-2
	3.1.1 Using the ALTER AUTHORIZATION Clause to Change Passwords 3.1.1.1 Standard CONNECT	3-2
	3.1.1.2 Change Password on CONNECT	3-2
	3.1.2 Connecting Using Oracle Net Services	3-3
	3.1.3 Automatic Connects	3-3
	3.1.3.1 The AUTO_CONNECT Precompiler Option	3-3
	3.1.3.2 SYSDBA, SYSOPER, or SYSBACKUP System Privileges	3-4
	3.2 Advanced Connection Options	3-4
	3.2.1 Some Preliminaries	3-4
	3.2.2 Concurrent Logons	3-4
	3.2.3 Default Databases and Connections	3-5
	3.2.4 Explicit Connections	3-5
	3.2.4.1 Single Explicit Connection	3-6
	3.2.4.2 Multiple Explicit Connections	3-9
	3.2.4.3 Ensuring Data Integrity	3-10
	3.2.5 Implicit Connections	3-10
	3.2.5.1 Single Implicit Connections	3-10
	3.2.5.2 Multiple Implicit Connections	3-11
	3.3 Definitions of Transactions Terms	3-11
	3.4 How Data Integrity Is Ensured	3-12



3.5 How to Begin and End Transactions	3-12
3.6 Using the COMMIT Statement	3-13
3.6.1 WITH HOLD Clause in DECLARE CURSOR Statements	3-13
3.6.2 CLOSE_ON_COMMIT Precompiler Option	3-14
3.7 Using the SAVEPOINT Statement	3-14
3.8 The ROLLBACK Statement	3-15
3.8.1 Statement-Level Rollbacks	3-17
3.9 The RELEASE Option	3-17
3.10 The SET TRANSACTION Statement	3-18
3.11 Override Default Locking	3-18
3.11.1 Using FOR UPDATE OF	3-19
3.11.1.1 Restrictions	3-19
3.11.2 Using LOCK TABLE	3-19
3.12 Fetch Across COMMITs	3-20
3.13 Distributed Transactions Handling	3-20
3.14 Guidelines	3-21
3.14.1 Designing Applications	3-21
3.14.2 Obtaining Locks	3-21
3.14.3 Using PL/SQL	3-21
Datatypes and Host Variables	
4.1 Oracle Datatypes	4-1
4.1.1 Internal Data types	4-1
4.1.2 External Datatypes	4-3
4.1.2.1 VARCHAR2	4-4
4.1.2.2 NUMBER	4-5
4.1.2.3 INTEGER	4-5
4.1.2.4 FLOAT	4-5
4.1.2.5 STRING	4-5
4.1.2.6 VARNUM	4-6
4.1.2.7 LONG	4-6
4.1.2.8 VARCHAR	4-6
4.1.2.9 ROWID	4-6
4.1.2.10 DATE	4-7
4.1.2.11 RAW	4-7
4.1.2.11 RAW 4.1.2.12 VARRAW	4-7 4-8
4.1.2.11 RAW 4.1.2.12 VARRAW 4.1.2.13 LONG RAW	4-7 4-8 4-8
4.1.2.11 RAW 4.1.2.12 VARRAW 4.1.2.13 LONG RAW 4.1.2.14 UNSIGNED	4-7 4-8 4-8 4-8
4.1.2.11 RAW 4.1.2.12 VARRAW 4.1.2.13 LONG RAW 4.1.2.14 UNSIGNED 4.1.2.15 LONG VARCHAR	4-7 4-8 4-8 4-8
4.1.2.11 RAW 4.1.2.12 VARRAW 4.1.2.13 LONG RAW 4.1.2.14 UNSIGNED	4-7 4-8 4-8 4-8



	4.1	.2.18	CHARZ	4-9
	4.1	.2.19	CHARF	4-9
	4.1	.2.20	BOOLEAN	4-9
	4.1.3	Addit	tional External Datatypes	4-10
	4.1	.3.1	Datetime and Interval Datatypes	4-10
	4.1	.3.2	ANSI DATE	4-10
	4.1	.3.3	TIMESTAMP	4-10
	4.1	.3.4	TIMESTAMP WITH TIME ZONE	4-10
	4.1	.3.5	TIMESTAMP WITH LOCAL TIME ZONE	4-11
	4.1	.3.6	INTERVAL YEAR TO MONTH	4-11
	4.1	.3.7	INTERVAL DAY TO SECOND	4-11
	4.1	.3.8	Avoiding Unexpected Results Using Datetime	4-11
	4.1.4	Supp	ort for Vector Data Type	4-12
	4.1.5	Supp	oort for JSON Data Type	4-13
4.2	Host	Variat	ples	4-14
	4.2.1	Host	Variable Declaration	4-14
	4.2	.1.1	Storage-Class Specifiers	4-16
	4.2	.1.2	Type Qualifiers	4-17
	4.2.2	Host	Variable Referencing	4-17
	4.2	.2.1	Restrictions	4-17
4.3	Indica	ator Va	ariables	4-18
	4.3.1	The I	NDICATOR Keyword	4-18
	4.3.2	Exan	nple of INDICATOR Variable Usage	4-18
	4.3.3	INDI	CATOR Variable Guidelines	4-19
	4.3.4	Orac	le Restrictions	4-19
4.4	VAR	CHAR	Variables	4-19
	4.4.1	VAR	CHAR Variable Declaration	4-20
	4.4.2	VAR	CHAR Variable Referencing	4-21
	4.4.3	Retu	rn NULLs to a VARCHAR Variable	4-21
	4.4.4	Inser	t NULLs Using VARCHAR Variables	4-21
	4.4.5	Pass	VARCHAR Variables to a Function	4-22
	4.4.6	Find	the Length of the VARCHAR Array Component	4-22
	4.4.7		nple Program: Using sqlvcp()	4-23
4.5	Curso		iables	4-25
	4.5.1		are a Cursor Variable	4-26
	4.5.2		ate a Cursor Variable	4-26
	4.5.3	Oper	a Cursor Variable	4-27
		.3.1	Opening in a Standalone Stored Procedure	4-28
		.3.2	Return Types	4-28
	4.5.4		ng and Freeing a Cursor Variable	4-29
	4.5.5		or Variables with the OCI (Release 7 Only)	4-29
	4.5.6	Restr	rictions (Cursor Variables)	4-30



4.5.7 Example: cv_demo.sql and sample11.pc	4-31
4.5.7.1 cv_demo.sql	4-31
4.5.7.2 sample11.pc	4-31
4.6 CONTEXT Variables	4-33
4.7 Universal ROWIDs	4-34
4.7.1 SQLRowidGet()	4-35
4.8 Host Structures	4-36
4.8.1 Host Structures and Arrays	4-37
4.8.2 PL/SQL Records	4-37
4.8.3 Nested Structures and Unions	4-37
4.8.4 Host Indicator Structures	4-38
4.8.5 Example Program: Cursor and a Host Structure	4-39
4.9 Pointer Variables	4-41
4.9.1 Pointer Variable Declaration	4-41
4.9.2 Pointer Variable Referencing	4-41
4.9.3 Structure Pointers	4-42
4.10 Globalization Support	4-42
4.11 NCHAR Variables	4-44
4.11.1 CHARACTER SET [IS] NCHAR_CS	4-44
4.11.2 Environment Variable NLS_NCHAR	4-44
4.11.3 CONVBUFSZ Clause in VAR	4-45
4.11.4 Character Strings in Embedded SQL	4-45
4.11.5 Strings Restrictions	4-45
4.44 C. Jadiantan Variables	4-45
4.11.6 Indicator Variables	
4.11.6 Indicator Variables 4.12 Handling Boolean Data	4-46
4.12 Handling Boolean Data	
4.12 Handling Boolean Data  Advanced Topics	4-46 5-1 5-1
Advanced Topics  5.1 Character Data  5.1.1 Precompiler Option CHAR_MAP  5.1.2 Inline Usage of the CHAR_MAP Option	4-46 5-1
4.12 Handling Boolean Data  Advanced Topics  5.1 Character Data 5.1.1 Precompiler Option CHAR_MAP	4-46 5-1 5-1
Advanced Topics  5.1 Character Data  5.1.1 Precompiler Option CHAR_MAP  5.1.2 Inline Usage of the CHAR_MAP Option	4-46 5-1 5-2
Advanced Topics  5.1 Character Data  5.1.1 Precompiler Option CHAR_MAP  5.1.2 Inline Usage of the CHAR_MAP Option  5.1.3 Effect of the DBMS and CHAR_MAP Options	5-1 5-2 5-2
Advanced Topics  5.1 Character Data  5.1.1 Precompiler Option CHAR_MAP  5.1.2 Inline Usage of the CHAR_MAP Option  5.1.3 Effect of the DBMS and CHAR_MAP Options  5.1.3.1 On Input	4-46 5-1 5-2 5-2 5-3
Advanced Topics  5.1 Character Data  5.1.1 Precompiler Option CHAR_MAP  5.1.2 Inline Usage of the CHAR_MAP Option  5.1.3 Effect of the DBMS and CHAR_MAP Options  5.1.3.1 On Input  5.1.3.2 On Input	5-1 5-1 5-2 5-3 5-4
Advanced Topics  5.1 Character Data  5.1.1 Precompiler Option CHAR_MAP  5.1.2 Inline Usage of the CHAR_MAP Option  5.1.3 Effect of the DBMS and CHAR_MAP Options  5.1.3.1 On Input  5.1.3.2 On Input  5.1.3.3 On Output	5-1 5-2 5-3 5-4 5-4
Advanced Topics  5.1 Character Data  5.1.1 Precompiler Option CHAR_MAP  5.1.2 Inline Usage of the CHAR_MAP Option  5.1.3 Effect of the DBMS and CHAR_MAP Options  5.1.3.1 On Input  5.1.3.2 On Input  5.1.3.3 On Output  5.1.4 VARCHAR Variables and Pointers	5-1 5-1 5-2 5-3 5-4 5-4
Advanced Topics  5.1 Character Data  5.1.1 Precompiler Option CHAR_MAP  5.1.2 Inline Usage of the CHAR_MAP Option  5.1.3 Effect of the DBMS and CHAR_MAP Options  5.1.3.1 On Input  5.1.3.2 On Input  5.1.3.3 On Output  5.1.4 VARCHAR Variables and Pointers  5.1.4.1 On Input	5-1 5-2 5-2 5-4 5-4 5-6
Advanced Topics  5.1 Character Data 5.1.1 Precompiler Option CHAR_MAP 5.1.2 Inline Usage of the CHAR_MAP Option 5.1.3 Effect of the DBMS and CHAR_MAP Options 5.1.3.1 On Input 5.1.3.2 On Input 5.1.3.3 On Output  5.1.4 VARCHAR Variables and Pointers 5.1.4.1 On Input 5.1.4.2 On Output	5-1 5-2 5-2 5-3 5-4 5-6 5-6
Advanced Topics  5.1 Character Data 5.1.1 Precompiler Option CHAR_MAP 5.1.2 Inline Usage of the CHAR_MAP Option 5.1.3 Effect of the DBMS and CHAR_MAP Options 5.1.3.1 On Input 5.1.3.2 On Input 5.1.3.3 On Output  5.1.4 VARCHAR Variables and Pointers 5.1.4.1 On Input 5.1.4.2 On Output  5.1.5 Unicode Variables	5-1 5-2 5-2 5-4 5-6 5-6 5-6



4-31

	5.3.1	Host Variable Equivalencing	5-9
	5.3.2	User-Defined Type Equivalencing	5-10
	5.3	3.2.1 REFERENCE Clause	5-11
	5.3.3	CHARF External Datatype	5-11
	5.3.4	The EXEC SQL VAR and TYPE Directives	5-11
	5.3.5	Example: Datatype Equivalencing (sample4.pc):	5-12
5.4	4 The	C Preprocessor	5-21
	5.4.1	How the Pro*C/C++ Preprocessor Works	5-21
	5.4.2	Preprocessor Directives	5-22
	5.4	4.2.1 Directives Ignored	5-22
	5.4.3	ORA_PROC Macro	5-23
	5.4.4	Location of Header File Specification	5-23
	5.4.5	Some Preprocessor Examples	5-23
	5.4	4.5.1 About Using #define	5-24
	5.4	4.5.2 Other Preprocessor Restrictions	5-25
	5.4.6	SQL Statements Not Allowed in #include	5-25
	5.4.7	Include the SQLCA, ORACA, and SQLDA	5-25
	5.4.8	EXEC SQL INCLUDE and #include Summary	5-26
	5.4.9	Defined Macros	5-27
	5.4.10	Include Files	5-27
5.5	5 Pred	compiled Header Files	5-27
	5.5.1	Precompiled Header File Creation	5-27
	5.5.2	Use of the Precompiled Header Files	5-28
	5.5.3	Examples	5-28
	5.5	5.3.1 Redundant File Inclusion	5-28
	5.5	5.3.2 Multiple Precompiled Header Files	5-29
	5.5.4	List of Header Files	5-30
	5.5.5	Effects of Options	5-30
	5.5	5.5.1 DEFINE and INCLUDE Options	5-30
	5.5	5.5.2 CODE and PARSE Options	5-31
	5.5.6	Usage Notes	5-32
5.6	5 The	Oracle Preprocessor	5-33
	5.6.1	Symbol Definition	5-33
	5.6.2	An Oracle Preprocessor Example	5-33
5.	7 Eval	uation of Numeric Constants	5-34
	5.7.1	Numeric Constants in Pro*C/C++	5-34
	5.7.2	Numeric Constant Rules and Examples	5-35
5.8	3 SQL	LIB Extensions for OCI Release 8 Interoperability	5-35
	5.8.1	Runtime Context in the OCI Release 8 Environment	5-35
	5.8.2	Parameters in the OCI Release 8 Environment Handle	5-36
5.9	9 Inter	face to OCI Release 8	5-36
	5.9.1	SQLEnvGet()	5-37



5.9.2 SQLSvcCtxGet()	5-37
5.9.3 Embedded OCI Release 8 Calls	5-38
5.10 Embedded OCI Release 7 Calls	5-39
5.10.1 Set Up the LDA	5-39
5.10.2 Remote and Multiple Connections	5-39
5.11 New Names for SQLLIB Public Functions	5-40
5.12 X/Open Application Development	5-42
5.12.1 Oracle-Specific Issues	5-43
5.12.1.1 Connecting to Oracle	5-43
5.12.1.2 Transaction Control	5-44
5.12.1.3 OCI Calls (Release 7 Only)	5-44
5.12.1.4 Linking	5-44
Embedded SQL	
6.1 Host Variables	6-1
6.1.1 Output versus Input Host Variables	6-1
6.2 Indicator Variables	6-2
6.2.1 Insert NULLs	6-3
6.2.2 Returned NULLs	6-4
6.2.3 Fetch NULLs	6-4
6.2.4 Test for NULLs	6-4
6.2.5 Truncated Values	6-5
6.3 The Basic SQL Statements	6-5
6.3.1 The SELECT Statement	6-6
6.3.1.1 Available Clauses	6-6
6.3.2 The INSERT Statement	6-7
6.3.2.1 About Using Subqueries	6-7
6.3.3 The UPDATE Statement	6-8
6.3.4 The DELETE Statement	6-8
6.3.5 The WHERE Clause	6-8
6.4 Using Boolean Data Type	6-9
6.5 The DML Returning Clause	6-9
6.6 Cursors	6-10
6.6.1 The DECLARE CURSOR Statement	6-11
6.6.2 The OPEN Statement	6-12
6.6.3 The FETCH Statement	6-12
6.6.4 The CLOSE Statement	6-13
6.7 Scrollable Cursors	6-14
6.7.1 About Using Scrollable Cursors	6-14
6.7.1.1 DECLARE SCROLL CURSOR	6-14
6.7.1.2 OPEN for Scrollable Cursors	6-14



	6-14
6.7.1.4 CLOSE for Scrollable Cursors	6-15
6.7.2 The CLOSE_ON_COMMIT Precompiler Option	6-15
6.7.3 The PREFETCH Precompiler Option	6-15
6.8 Optimizer Hints	6-16
6.8.1 Issuing Hints	6-17
6.9 Fix Execution Plan	6-17
6.9.1 SQL File	6-19
6.9.1.1 Examples	6-19
6.9.2 LOG File	6-21
6.10 The CURRENT OF Clause	6-21
6.10.1 Restrictions (FOR UPDATE OF)	6-22
6.11 The Cursor Statements	6-22
6.12 A Complete Example Using Non-Scrollable Cursor	6-23
6.13 A Complete Example Using Scrollable Cursor	6-24
6.14 Positioned Update	6-25
6.15 Explicit Request Boundaries for Handling Planned Maintena	nce 6-26
6.15.1 Explicit Request Boundaries Limitations	6-27
6.15.2 Explicit Request Boundaries Example	6-28
Embedded PL/SQL	7.1
7.1 Advantages of PL/SQL 7.1.1 Better Performance	7-1 7-1
7.1.2 Integration with Oracle	
	7_2
9	7-2 7-2
7.1.3 Cursor FOR Loops	7-2
<ul><li>7.1.3 Cursor FOR Loops</li><li>7.1.4 Procedures and Functions</li></ul>	7-2 7-2
<ul><li>7.1.3 Cursor FOR Loops</li><li>7.1.4 Procedures and Functions</li><li>7.1.5 Packages</li></ul>	7-2 7-2 7-3
<ul><li>7.1.3 Cursor FOR Loops</li><li>7.1.4 Procedures and Functions</li><li>7.1.5 Packages</li><li>7.1.6 PL/SQL Tables</li></ul>	7-2 7-2 7-3 7-3
<ul> <li>7.1.3 Cursor FOR Loops</li> <li>7.1.4 Procedures and Functions</li> <li>7.1.5 Packages</li> <li>7.1.6 PL/SQL Tables</li> <li>7.1.7 User-Defined Records</li> </ul>	7-2 7-2 7-3 7-3 7-4
<ul> <li>7.1.3 Cursor FOR Loops</li> <li>7.1.4 Procedures and Functions</li> <li>7.1.5 Packages</li> <li>7.1.6 PL/SQL Tables</li> <li>7.1.7 User-Defined Records</li> </ul>	7-2 7-2 7-3 7-3 7-4 7-4
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables	7-2 7-2 7-3 7-3 7-4
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL	7-2 7-2 7-3 7-3 7-4 7-4 7-5
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL	7-2 7-2 7-3 7-3 7-4 7-4 7-5 7-5
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL 7.3.2 Complex Example	7-2 7-2 7-3 7-3 7-4 7-4 7-5 7-5
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL 7.3.2 Complex Example 7.3.3 VARCHAR Pseudotype	7-2 7-2 7-3 7-3 7-4 7-4 7-5 7-5 7-6 7-8
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL 7.3.2 Complex Example 7.3.3 VARCHAR Pseudotype 7.3.4 Restriction	7-2 7-2 7-3 7-3 7-4 7-4 7-5 7-5 7-6 7-8 7-9
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL 7.3.2 Complex Example 7.3.3 VARCHAR Pseudotype 7.3.4 Restriction 7.4 Indicator Variables	7-2 7-2 7-3 7-3 7-4 7-4 7-5 7-5 7-6 7-8 7-9
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL 7.3.2 Complex Example 7.3.3 VARCHAR Pseudotype 7.3.4 Restriction 7.4 Indicator Variables 7.4.1 NULLs Handling	7-2 7-2 7-3 7-3 7-4 7-4 7-5 7-5 7-6 7-8 7-9 7-10
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL 7.3.2 Complex Example 7.3.3 VARCHAR Pseudotype 7.3.4 Restriction 7.4 Indicator Variables 7.4.1 NULLs Handling 7.4.2 Truncated Values	7-2 7-2 7-3 7-3 7-3 7-4 7-4 7-5 7-5 7-6 7-8 7-9 7-10 7-10
7.1.3 Cursor FOR Loops 7.1.4 Procedures and Functions 7.1.5 Packages 7.1.6 PL/SQL Tables 7.1.7 User-Defined Records 7.2 Embedded PL/SQL Blocks 7.3 Host Variables 7.3.1 Example: Using Host Variables with PL/SQL 7.3.2 Complex Example 7.3.3 VARCHAR Pseudotype 7.3.4 Restriction 7.4 Indicator Variables 7.4.1 NULLs Handling 7.4.2 Truncated Values 7.5 Host Arrays	7-2 7-2 7-3 7-3 7-3 7-4 7-4 7-5 7-5 7-6 7-8 7-9 7-10 7-10



	7.6 Curs	sor Usage in Embedded PL/SQL	7-14
	7.7 Stor	red PL/SQL and Java Subprograms	7-15
	7.7.1	About Creating Stored Subprograms	7-15
	7.7.2	About Calling a Stored PL/SQL or Java Subprogram	7-17
	7.	7.2.1 Anonymous PL/SQL Block	7-17
	7.	7.2.2 Remote Access	7-20
	7.	7.2.3 The CALL Statement	7-20
	7.	7.2.4 CALL Example	7-21
	7.7.3	About Getting Information about Stored Subprograms	7-21
	7.8 Exte	ernal Procedures	7-22
	7.8.1	Restrictions on External Procedures	7-22
	7.8.2	About Creating the External Procedure	7-23
	7.8.3	SQLExtProcError()	7-24
	7.9 Abo	ut Using Dynamic SQL	7-24
8	Host Ar	rrays	
	8.1 Why	/ Use Arrays?	8-1
	8.2 Abo	ut Declaring Host Arrays	8-1
	8.2.1	Restrictions (Declaring Host Arrays)	8-2
	8.2.2	Maximum Size of Arrays	8-2
	8.3 Abo	ut Using Arrays in SQL Statements	8-2
	8.3.1	About Referencing Host Arrays	8-2
	8.3.2	About Using Indicator Arrays	8-3
	8.3.3	Oracle Restrictions (for Host Arrays)	8-3
	8.3.4	ANSI Restriction and Requirements	8-3
	8.4 Abo	ut Selecting into Arrays	8-4
	8.4.1	Cursor Fetches	8-4
	8.4.2	About Using sqlca.sqlerrd[2]	8-5
	8.4.3	Number of Rows Fetched	8-6
	8.4.4	Scrollable Cursor Fetches	8-6
	8.4.5	Sample Program 3: Host Arrays	8-6
	8.4.6	Sample Program: Host Arrays Using Scrollable Cursor	8-8
	8.	4.6.1 scdemo2.pc	8-9
	8.4.7	Host Array Restrictions	8-11
	8.4.8	About Fetching NULLs	8-11
	8.4.9	About Fetching Truncated Values	8-11
	8.5 Abo	ut Inserting with Arrays	8-11
	8.5.1	About Inserting with Arrays Restrictions	8-12
	8.6 Abo	ut Updating with Arrays	8-12
	8.6.1	About Updating with Arrays Restrictions	8-13
	8.7 Abo	ut Deleting with Arrays	8-13



	8.7.1 About Deleting with Arrays Restrictions	8-14
	8.8 About Using the FOR Clause	8-14
	8.8.1 FOR Clause Restrictions	8-15
	8.8.1.1 In a SELECT Statement	8-15
	8.8.1.2 With the CURRENT OF Clause	8-15
	8.9 About Using the WHERE Clause	8-15
	8.10 Arrays of Structs	8-16
	8.10.1 Arrays of Structs Usage	8-17
	8.10.2 Restrictions on Arrays of Structs	8-17
	8.10.3 About Declaring an Array of Structs	8-17
	8.10.4 Variables Guidelines	8-18
	8.10.5 About Declaring a Pointer to an Array of Structs	8-19
	8.10.6 Examples	8-19
	8.10.6.1 Example 1: A Simple Array of Structs of Scalars	8-20
	8.10.6.2 Example 2: Using Mixed Scalar Arrays with An Array of Structs	8-20
	8.10.6.3 Example 3: Using Multiple Arrays of Structs with a Cursor	8-21
	8.10.6.4 Example 4: Individual Array and Struct Member Referencing	8-22
	8.10.6.5 Example 5: Using Indicator Variables, a Special Case	8-22
	8.10.6.6 Example 6: Using a Pointer to an Array of Structs	8-23
	8.11 About Mimicking CURRENT OF	8-24
	8.12 About Using Additional Array Insert/Select Syntax	8-25
	8.13 About Using Implicit Buffered Insert	8-29
	8.14 Scrollable Cursors	8-33
9	Handling Runtime Errors	
	9.1 The Need for Error Handling	9-1
	9.2 Error Handling Alternatives	9-1
	9.2.1 Status Variables	9-2
	9.2.2 The SQL Communications Area	9-2
	9.3 The SQLSTATE Status Variable	9-3
	9.3.1 About Declaring SQLSTATE	9-3
	9.3.2 SQLSTATE Values	9-3
	9.3.3 About Using SQLSTATE	9-10
	9.3.3.1 If You Declare SQLSTATE	9-10
	9.3.3.2 If You Do not Declare SQLSTATE	9-10
	9.4 About Declaring SQLCODE	9-10
	9.5 Key Components of Error Reporting Using the SQLCA	9-11
	9.5.1 Status Codes	9-11
	9.5.2 Warning Flags	9-11
	9.5.3 Rows-Processed Count	9-11
	9.5.4 Parse Error Offsets	9-12



	9.5.5	Error	Message lext	9-12
9.6	Usin	g the S	SQL Communications Area (SQLCA)	9-12
	9.6.1	Abou	it Declaring the SQLCA	9-13
	9.6.2	SQL	CA Contents	9-13
	9.6.3	SQL	CA Structure	9-15
	9.6	5.3.1	sqlcaid	9-15
	9.6	3.3.2	sqlcabc	9-15
	9.6	5.3.3	sqlcode	9-15
	9.6	5.3.4	sqlerrm	9-16
	9.6	3.3.5	sqlerrp	9-16
	9.6	3.6	sqlerrd	9-16
	9.6	3.3.7	sqlwarn	9-17
	9.6	3.8	sqlext	9-17
	9.6.4	PL/S	QL Considerations	9-17
9.7	Abou	ıt Gett	ing the Full Text of Error Messages	9-17
9.8	Abou	ıt Usin	g the WHENEVER Directive	9-19
	9.8.1	WHE	NEVER Conditions	9-19
	9.8	3.1.1	SQLWARNING	9-19
	9.8	3.1.2	SQLERROR	9-19
	9.8	3.1.3	NOT FOUND	9-19
	9.8.2	WHE	NEVER Actions	9-19
	9.8	3.2.1	CONTINUE	9-20
	9.8	3.2.2	DO	9-20
	9.8	3.2.3	DO BREAK	9-20
	9.8	3.2.4	DO CONTINUE	9-20
	9.8	3.2.5	GOTO label_name	9-20
	9.8	3.2.6	STOP	9-20
	9.8.3	WHE	NEVER Examples	9-20
	9.8.4		of DO BREAK and DO CONTINUE	9-21
	9.8.5	Scop	e of WHENEVER	9-22
	9.8.6	Guid	elines for WHENEVER	9-22
	9.8	3.6.1	Placing the Statements	9-23
	9.8	3.6.2	Handling End-of-Data Conditions	9-23
	9.8	3.6.3	About Avoiding Infinite Loops	9-23
	9.8	3.6.4	About Maintaining Addressability	9-24
	9.8	3.6.5	About Returning After an Error	9-24
9.9	Abou	ıt Obta	aining the Text of SQL Statements	9-25
	9.9.1	Rest	rictions (using SQLStmtGetText())	9-26
	9.9.2	Exan	nple Program	9-27
9.1			ing the Oracle Communications Area (ORACA)	9-27
	9.10.1		out Declaring the ORACA	9-27
	9.10.2	Abo	out Enabling the ORACA	9-27



	9.10.3	ORACA Contents	9-28
	9.10.4	About Choosing Runtime Options	9-29
	9.10.5	Structure of the ORACA	9-29
	9.1	10.5.1 oracaid	9-29
	9.1	10.5.2 oracabc	9-30
	9.1	10.5.3 oracchf	9-30
	9.1	10.5.4 oradbgf	9-30
	9.1	10.5.5 orahchf	9-30
	9.1	10.5.6 orastxtf	9-30
	9.1	10.5.7 Diagnostics	9-30
	9.1	10.5.8 orastxt	9-31
	9.1	10.5.9 orasfnm	9-31
	9.1	10.5.10 oraslnr	9-31
	9.1	10.5.11 Cursor Cache Statistics	9-31
	9.1	10.5.12 orahoc	9-31
	9.1	10.5.13 oramoc	9-31
	9.1	10.5.14 oracoc	9-32
	9.1	10.5.15 oranor	9-32
	9.1	10.5.16 oranpr	9-32
	9.1	10.5.17 oranex	9-32
	9.10.6	ORACA Example	9-32
10	Precom	piler Options	
	10.1 The	e Precompiler Command	10-1
	10.1.1	Case Sensitivity	10-2
	10.2 Pre	ecompiler Options	10-2
	10.2.1	Environment Variables	10-2
	10.2.2	Configuration Files	10-3
	10.2.3	Precedence of Option Values	10-4
	10.2.4	Macro and Micro Options	10-6
	10.2.5	What Occurs During Precompilation?	10-6
	10.2.6	Scope of Options	10-7
	10.2.7	Pro*C/C++ Precompiler Issues for Windows Platforms	10-7
	10	0.2.7.1 Configuration File	10-7
	10	0.2.7.2 CODE	10-7
	10	0.2.7.3 DBMS	10-7
	10	0.2.7.4 INCLUDE	10-7
	10	0.2.7.5 PARSE	10-7
	10.3 Qui	ick Reference	10-7
	10.4 Ent	tering Options	10-10
	10.4.1	On the Command Line	10-10



	10.4.2	Inline	10-10
	10.4	.2.1 Uses for EXEC ORACLE	10-10
	10.4	.2.2 Scope of EXEC ORACLE	10-11
	10.4.3	Column Properties Support	10-11
10.5	Abou	ıt Data Use Case Domain Metadata Support	10-12
10.6	Abou	t Using the Precompiler Options	10-13
	10.6.1	AUTO_CONNECT	10-14
	10.6.2	CHAR_MAP	10-14
	10.6.3	CINCR	10-15
	10.6.4	CLOSE_ON_COMMIT	10-15
	10.6.5	CMAX	10-16
	10.6.6	CMIN	10-16
	10.6.7	CNOWAIT	10-17
	10.6.8	CODE	10-17
	10.6.9	COMMON_PARSER	10-18
	10.6.10	COMP_CHARSET	10-18
	10.6.11	CONFIG	10-19
	10.6.12	CPOOL	10-19
	10.6.13	CPP_SUFFIX	10-19
	10.6.14	CTIMEOUT	10-20
	10.6.15	DB2_ARRAY	10-20
	10.6.16	DBMS	10-21
	10.6.17	DEF_SQLCODE	10-22
	10.6.18	DEFINE	10-22
	10.6.19	DURATION	10-23
	10.6.20	DYNAMIC	10-24
	10.6.21	ERRORS	10-24
	10.6.22	ERRTYPE	10-24
	10.6.23	EVENTS	10-25
	10.6.24	FIPS	10-25
	10.6.25	HEADER	10-26
	10.6.26	HOLD_CURSOR	10-27
	10.6.27	IMPLICIT_SVPT	10-27
	10.6.28	INAME	10-28
	10.6.29	INCLUDE	10-29
	10.6.30	INTYPE	10-30
	10.6.31	LINES	10-30
	10.6.32	LNAME	10-31
	10.6.33	LTYPE	10-31
	10.6.34	MAX_ROW_INSERT	10-32
	10.6.35	MAXLITERAL	10-32
	10.6.36	MAXOPENCURSORS	10-33



10.6.37	MODE	10-34
10.6.38	NATIVE_TYPES	10-34
10.6.39	NLS_CHAR	10-35
10.6.40	NLS_LOCAL	10-35
10.6.41	OBJECTS	10-36
10.6.42	ONAME	10-36
10.6.43	ORACA	10-37
10.6.44	OUTLINE	10-37
10.6.45	OUTLNPREFIX	10-37
10.6.46	PAGELEN	10-38
10.6.47	PARSE	10-38
10.6.48	PLAN_BASELINE	10-39
10.6.49	PLAN_PREFIX	10-40
10.6.50	PLAN_RUN	10-40
10.6.51	PLAN_FIXED	10-40
10.6.52	PLAN_ENABLED	10-41
10.6.53	MEMFORPREFETCH	10-41
10.6.54	PREFETCH	10-42
10.6.55	RELEASE_CURSOR	10-42
10.6.56	RUNOUTLINE	10-43
10.6.57	SELECT_ERROR	10-43
10.6.58	STMT_CACHE	10-44
10.6.59	SYS_INCLUDE	10-44
10.6.60	THREADS	10-46
10.6.61	TYPE_CODE	10-46
10.6.62	UNSAFE_NULL	10-46
10.6.63	USERID	10-47
10.6.64	UTF16_CHARSET	10-47
10.6.65	VARCHAR	10-48
10.6.66	VERSION	10-48
Multithrea	aded Applications	
11.1 What	are Threads?	11-1
11.2 Runtir	me Contexts in Pro*C/C++	11-2
11.3 Runtir	me Context Usage Models	11-3
11.3.1	Multiple Threads Sharing a Single Runtime Context	11-3
11.3.2	Multiple Threads Sharing Multiple Runtime Contexts	11-4
11.4 User I	Interface Features for Multithreaded Applications	11-5
11.4.1	THREADS Option	11-5
11.4.2	Embedded SQL Statements and Directives	11-6
11.4.	2.1 EXEC SQL ENABLE THREADS	11-6



	11.4.2.2	EXEC SQL CONTEXT ALLOCATE	11-6
	11.4.2.3	EXEC SQL CONTEXT USE	11-6
	11.4.2.4	EXEC SQL CONTEXT FREE	11-7
	11.4.3 CO	NTEXT USE Examples	11-8
	11.4.4 Prog	gramming Considerations	11-9
	11.5 Multithrea	ded Example	11-9
	11.6 Connection	on Pooling	11-14
	11.6.1 Abo	out Using the Connection Pooling Feature	11-16
	11.6.1.1	How to Enable Connection Pooling	11-16
	11.6.1.2	Command Line Options for Connection Pooling	11-16
	11.6.1.3	Example	11-18
	11.6.1.4	Performance Tuning	11-18
	11.6.2 Den	no Program:1	11-18
	11.6.2.1	Example	11-19
	11.6.3 Den	no Program:2	11-22
	11.6.3.1	Case 1: By varying CMIN	11-23
	11.6.3.2	Case 2: By varying CMAX	11-24
	11.6.3.3	Example	11-24
12	C++ Applica	tions	
	12.1 Understar	nding C++ Support	12-1
	12.1.1 No	Special Macro Processing	12-1
	12.2 Precompi	ling for C++	12-2
	12.2.1 Cod	de Generation	12-2
	12.2.2 Abo	out Parsing Code	12-3
	12.2.3 Out	put Filename Extension	12-3
	12.2.4 Sys	tem Header Files	12-4
	12.3 Example I	Programs	12-4
	12.3.1 cpp	demo1.pc	12-4
	12.3.2 cpp	demo2.pc	12-7
	12.3.3 cpp	demo3.pc	12-10
13	Oracle Dyna	amic SQL	
	13.1 What is D	ynamic SQL?	13-1
	13.2 Advantage	es and Disadvantages of Dynamic SQL	13-2
	13.3 When to U	Jse Dynamic SQL	13-2
	13.4 Requirem	ents for Dynamic SQL Statements	13-2
	13.5 How Dyna	amic SQL Statements are Processed	13-3



13.6 Meth	lods for Using Dynamic SQL	13-3
13.6.1	Method 1	13-4
13.6.2	Method 2	13-4
13.6.3	Method 3	13-4
13.6.4	Method 4	13-4
13.6.5	Guidelines	13-5
13.6	6.5.1 About Avoiding Common Errors	13-5
13.7 Usin	g Method 1	13-6
13.7.1	Example Program: Dynamic SQL Method 1	13-7
13.8 Using	g Method 2	13-9
13.8.1	The USING Clause	13-11
13.8.2	Example Program: Dynamic SQL Method 2	13-11
13.9 Usin	g Method 3	13-14
13.9.1	PREPARE (Dynamic SQL)	13-14
13.9.2	DECLARE (Dynamic SQL)	13-15
13.9.3	OPEN (Dynamic SQL)	13-15
13.9.4	FETCH (Dynamic SQL)	13-15
13.9.5	CLOSE (Dynamic SQL)	13-16
13.9.6	Example Program: Dynamic SQL Method 3	13-16
13.10 Usi	ng Method 4	13-19
13.10.1	Need for the SQLDA	13-19
13.10.2	The DESCRIBE Statement	13-20
13.10.3	What is a SQLDA?	13-20
13.10.4	About Implementing Oracle Method 4	13-21
13.10.5	Restriction	13-21
13.11 Abo	out Using the DECLARE STATEMENT Statement	13-21
13.11.1	About Using Host Arrays	13-22
13.12 Abo	out Using PL/SQL	13-22
13.12.1	With Method 1	13-23
13.12.2	With Method 2	13-23
13.12.3	With Method 3	13-23
13.12.4	With Oracle Method 4	13-23
13.13 Dyr	namic SQL Statement Caching	13-24
13.14 Abo	out Boolean Data Type Support	13-26
ANSI Dy	namic SQL	
14.1 Basic	cs of ANSI Dynamic SQL	14-1
14.1.1	Precompiler Options	14-2
14.2 Over	view of ANSI SQL Statements	14-2
14.2.1	Example Code	14-4
14.3 Orac	ele Extensions	14-5



	14.3.1	Reference Semantics	14-6
	14.3.2	About Using Arrays for Bulk Operations	14-6
	14.3.3	Support for Arrays of Structs	14-8
	14.3.4	Support for Object Types	14-8
	14.4 ANS	SI Dynamic SQL Precompiler Options	14-8
	14.5 Full	Syntax of the Dynamic SQL Statements	14-9
	14.5.1	ALLOCATE DESCRIPTOR	14-9
	14.5.2	DEALLOCATE DESCRIPTOR	14-10
	14.5.3	GET DESCRIPTOR	14-11
	14.5.4	SET DESCRIPTOR	14-14
	14.5.5	Use of PREPARE	14-16
	14.5.6	DESCRIBE INPUT	14-17
	14.5.7	DESCRIBE OUTPUT	14-17
	14.5.8	EXECUTE	14-18
	14.5.9	Use of EXECUTE IMMEDIATE	14-19
	14.5.10	Use of DYNAMIC DECLARE CURSOR	14-19
	14.5.11	OPEN Cursor	14-20
	14.5.12	FETCH	14-20
	14.5.13	CLOSE a Dynamic Cursor	14-21
	14.5.14	Differences From Oracle Dynamic Method 4	14-21
	14.5.15	Restrictions (ANSI Dynamic SQL)	14-22
	14.6 Exa	mple Programs	14-22
	14.6.1	ansidyn1.pc	14-22
	14.6.2	ansidyn2.pc	14-28
15	Oracle D	Dynamic SQL: Method 4	
	15.1 Mee	eting the Special Requirements of Method 4	15-1
	15.1.1	What Makes Method 4 Special?	15-2
	15.1.2	What Information Does Oracle Need?	15-2
	15.1.3	Where Is the Information Stored?	15-2
	15.1.4	How is the SQLDA Referenced?	15-3
	15.1.5	How is the Information Obtained?	15-3
	15.2 Und	erstanding the SQLDA	15-3
	15.2.1	Purpose of the SQLDA	15-3
	15.2.2	Multiple SQLDAs	15-4
	15.2.3	Declaring a SQLDA	15-4
	15.2.4	Allocating a SQLDA	15-4
		ut Using the SQLDA Variables	15-5
	15.3.1	The N Variable	15-6
	15.3.2	The V Variable	15-6
	15.3.3	The L Variable	15-6



1	5.3.4	The T Variable	15-7
1	5.3.5	The I Variable	15-7
1	5.3.6	The F Variable	15-8
1	5.3.7	The S Variable	15-8
1	5.3.8	The M Variable	15-8
1	5.3.9	The C Variable	15-8
1	5.3.10	The X Variable	15-9
1	5.3.11	The Y Variable	15-9
1	5.3.12	The Z Variable	15-9
15.4	Some	e Preliminaries	15-9
1	5.4.1	Converting Data	15-9
	15.4	1.1 Internal Datatypes	15-10
	15.4	.1.2 External Datatypes	15-10
1	5.4.2	Coercing Datatypes	15-11
	15.4	.2.1 Extracting Precision and Scale	15-12
1	5.4.3	Handling NULL/Not NULL Datatypes	15-13
15.5	The E	Basic Steps	15-14
15.6	A Clo	ser Look at Each Step	15-15
1	5.6.1	Declare a Host String	15-16
1	5.6.2	Declare the SQLDAs	15-16
1	5.6.3	Allocate Storage Space for the Descriptors	15-17
1	5.6.4	Set the Maximum Number to DESCRIBE	15-17
1	5.6.5	Put the Query Text in the Host String	15-19
1	5.6.6	PREPARE the Query from the Host String	15-20
1	5.6.7	DECLARE a Cursor	15-20
1	5.6.8	DESCRIBE the Bind Variables	15-20
1	5.6.9	Reset Number of Placeholders	15-21
1	5.6.10	Get Values and Allocate Storage for Bind Variables	15-21
1	5.6.11	OPEN the Cursor	15-23
1	5.6.12	DESCRIBE the Select List	15-24
1	5.6.13	Reset Number of Select-List Items	15-25
1	5.6.14	Reset Length/Datatype of Each Select-list Item	15-26
1	5.6.15	FETCH Rows from the Active Set	15-27
1	5.6.16	Get and Process Select-List Values	15-28
1	5.6.17	Deallocate Storage	15-29
1	5.6.18	CLOSE the Cursor	15-30
1	5.6.19	About Using Host Arrays	15-30
1	5.6.20	sample12.pc	15-32
15.7	Abou	t Using Domain Metadata	15-32
15.8		pple Program: Dynamic SQL Method 4	15-33
15.9		ole Program : Dynamic SQL Method 4 using Scrollable Cursors	15-43



# 16 LOBs

16.	1 Wha	t are LOBs?	16-1
	16.1.1	Internal LOBs	16-1
	16.1.2	External LOBs	16-1
	16.1.3	Security for BFILEs	16-2
	16.1.4	LOBs versus LONG and LONG RAW	16-2
	16.1.5	LOB Locators	16-2
	16.1.6	Temporary LOBs	16-3
	16.1.7	LOB Buffering Subsystem	16-3
16.	2 How	to Use LOBs in Your Program	16-3
	16.2.1	Three Ways to Access LOBs	16-3
	16.2.2	LOB Locators in Your Application	16-5
	16.2.3	Initializing a LOB	16-6
	16.2	2.3.1 Internal LOBs	16-6
	16.2	2.3.2 External LOBs	16-6
	16.2	2.3.3 Temporary LOBs	16-7
	16.2	2.3.4 Freeing LOBs	16-7
16.	3 Rule	s for LOB Statements	16-7
	16.3.1	For All LOB Statements	16-7
	16.3.2	For the LOB Buffering Subsystem	16-7
	16.3.3	For Host Variables	16-8
16.	4 LOB	Statements	16-9
	16.4.1	APPEND	16-9
	16.4.2	ASSIGN	16-9
	16.4.3	CLOSE (for LOBs)	16-10
	16.4.4	COPY	16-10
	16.4.5	CREATE TEMPORARY	16-11
	16.4.6	DISABLE BUFFERING	16-12
	16.4.7	ENABLE BUFFERING	16-12
	16.4.8	ERASE	16-13
	16.4.9	FILE CLOSE ALL	16-13
	16.4.10	FILE SET	16-14
	16.4.11	FLUSH BUFFER	16-14
	16.4.12	FREE TEMPORARY	16-15
	16.4.13	LOAD FROM FILE	16-15
	16.4.14	OPEN (for LOBs)	16-16
	16.4.15	READ	16-17
	16.4.16	TRIM	16-18
	16.4.17	WRITE	16-19
	16.4.18	DESCRIBE	16-20
16.	5 LOB	s and the Navigational Interface	16-22



	16.5.1	Transient Objects	16-22
	16.5.2	Persistent Objects	16-22
	16.5.3	Navigational Interface Example	16-23
	16.6 LOB	B Program Examples	16-24
	16.6.1	READ a BLOB, Write a File Example	16-24
	16.6.2	Read a File, WRITE a BLOB Example	16-25
	16.6.3	lobdemo1.pc	16-27
17	Objects		
	17.1 Intro	oduction to Objects	17-1
	17.1.1	Object Types	17-1
	17.1.2	REFs to Object Types	17-2
	17.1.3	Type Inheritance	17-2
	17.2 Abo	ut Using Object Types in Pro*C/C++	17-3
	17.2.1	NULL Indicators	17-3
	17.3 The	Object Cache	17-3
	17.3.1	Persistent Versus Transient Copies of Objects	17-4
	17.4 Asso	ociative Interface	17-4
	17.4.1	When to Use the Associative Interface	17-4
	17.4.2	ALLOCATE	17-4
	17.4.3	FREE	17-5
	17.4.4	CACHE FREE ALL	17-5
	17.4.5	Accessing Objects Using the Associative Interface	17-5
	17.5 Nav	igational Interface	17-6
	17.5.1	When to Use the Navigational Interface	17-7
	17.5.2	Rules Used in the Navigational Statements	17-7
	17.5.3	OBJECT CREATE	17-8
	17.5.4	OBJECT DEREF	17-9
	17.5.5	OBJECT RELEASE	17-9
	17.5.6	OBJECT DELETE	17-10
	17.5.7	OBJECT UPDATE	17-10
	17.5.8	OBJECT FLUSH	17-10
	17.5.9	Navigational Access to Objects	17-10
		verting Object Attributes and C Types	17-12
	17.6.1	OBJECT SET	17-12
	17.6.2	OBJECT GET	17-13
	-	ect Options Set/Get	17-15
	17.7.1	CONTEXT OBJECT OPTION SET	17-15
	17.7.2	CONTEXT OBJECT OPTION GET	17-16
		v Precompiler Options for Objects	17-16
	17.8.1	VERSION	17-16



	17.8.2	DURATION	17-16
	17.8.3	OBJECTS	17-17
	17.8.4	INTYPE	17-17
	17.8.5	ERRTYPE	17-17
	17.8.6	SQLCHECK Support for Objects	17-18
	17.8.7	Type Checking at Runtime	17-18
	17.9 An (	Object Example in Pro*C/C++	17-18
	17.9.1	Associative Access	17-18
	17.9.2	Navigational Access	17-19
	17.10 Ex	ample Code for Type Inheritance	17-20
	17.11 Ex	ample Code for Navigational Access	17-27
	17.12 Ab	out Using C Structures	17-33
	17.13 Ab	out Using REFs	17-34
	17.13.1	Generating a C Structure for a REF	17-34
	17.13.2	2 Declaring REFs	17-34
	17.13.3	B Using REFs in Embedded SQL	17-34
	17.14 Ab	out Using OCIDate, OCIString, OCINumber, and OCIRaw	17-35
	17.14.1	Declaring OCIDate, OCIString, OCINumber, OCIRaw	17-35
	17.14.2	2 Use of the OCI Types in Embedded SQL	17-35
	17.14.3	Manipulating the OCI Types	17-35
	17.15 Su	mmarizing the New Database Types in Pro*C/C++	17-35
	17.16 Re	estrictions on Using Oracle Datatypes in Dynamic SQL	17-38
L8	Collection	ons	
	18.1 Coll	ections	18-1
	18.1.1	Nested Tables	18-1
	18.1.2	Varrays	18-2
	18.1.3	C and Collections	18-2
	18.2 Des	criptors for Collections	18-2
	18.2.1	Declarations for Host and Indicator Variables	18-3
	18.2.2	About Manipulating Collections	18-3
	18.	2.2.1 Autonomous Collection Access	18-3
	18.	2.2.2 Collection Element Access	18-3
	18.2.3	Rules for Access	18-3
	18.	2.3.1 Autonomous Access	18-4
	18.	2.3.2 Element Access	18-4
	18.2.4	Indicator Variables	18-4
	18.	2.4.1 Autonomous Bindings	18-4
	18.	2.4.2 Element Bindings	18-4
	18.3 OB	JECT GET and SET	18-4
	18.4 Coll	ection Statements	18-6



16.4.1 COLLECTION GET	TO-6
18.4.2 COLLECTION SET	18-7
18.4.3 COLLECTION RESET	18-8
18.4.4 COLLECTION APPEND	18-9
18.4.5 COLLECTION TRIM	18-10
18.4.6 COLLECTION DESCRIBE	18-10
18.4.6.1 Notes on the Table	18-12
18.4.7 Rules for the Use of Collections	18-12
18.5 Collection Example Code	18-12
18.5.1 Type and Table Creation	18-12
18.5.2 GET and SET Example	18-14
18.5.3 DESCRIBE Example	18-15
18.5.4 RESET Example	18-16
18.5.5 Example Program:coldemo1.pc	18-18
The Object Type Translator	
19.1 OTT Overview	19-1
19.2 What is the Object Type Translator	19-1
19.2.1 About Creating Types in the Database	19-3
19.2.2 About Invoking OTT	19-3
19.2.2.1 Command Line	19-4
19.2.2.2 Configuration File	19-4
19.2.2.3 INTYPE File	19-4
19.2.3 The OTT Command Line	19-4
19.2.3.1 OTT	19-5
19.2.3.2 Userid	19-5
19.2.3.3 INTYPE	19-5
19.2.3.4 OUTTYPE	19-5
19.2.3.5 CODE	19-5
19.2.3.6 HFILE	19-5
19.2.3.7 INITFILE	19-6
19.2.3.8 INITFUNC	19-6
19.2.4 The INTYPE File	19-6
19.2.5 OTT Datatype Mappings	19-8
19.2.5.1 Mapping Object Datatypes to C	19-9
19.2.5.2 OTT Type Mapping Example	19-10
19.2.6 NULL Indicator Structs	19-12
19.2.7 OTT Support for Type Inheritance	19-13
19.2.7.1 Substitutable Object Attributes	19-14
19.2.8 The OUTTYPE File	19-15
19.3 Using OTT with OCI Applications	19-16



	19.3.1	About Accessing and Manipulating Objects with OCI	19-17
	19.3.2	About Calling the Initialization Function	19-18
	19.3.3	Tasks of the Initialization Function	19-19
	19.4 Abou	ut Using OTT with Pro*C/C++ Applications	19-19
	19.5 OTT	Reference	19-21
	19.5.1	OTT Command Line Syntax	19-22
	19.5.2	OTT Parameters	19-22
	19.5	5.2.1 USERID	19-23
	19.5	5.2.2 INTYPE	19-23
	19.5	5.2.3 OUTTYPE	19-23
	19.5	5.2.4 CODE	19-24
	19.5	5.2.5 INITFILE	19-24
	19.5	5.2.6 INITFUNC	19-24
	19.5	5.2.7 HFILE	19-24
	19.5	5.2.8 CONFIG	19-25
	19.5	5.2.9 ERRTYPE	19-25
	19.5	5.2.10 CASE	19-25
	19.5	5.2.11 SCHEMA_NAMES	19-26
	19.5	5.2.12 TRANSITIVE	19-26
	19.5.3	Where OTT Parameters Can Appear	19-26
	19.5.4	Structure of the INTYPE File	19-27
	19.5	5.4.1 INTYPE File Type Specifications	19-27
	19.5.5	Nested #include File Generation	19-28
	19.5.6	SCHEMA_NAMES Usage	19-30
	19.5.7	Default Name Mapping	19-31
	19.5.8	Restriction	19-32
	19.5	5.8.1 File Name Comparison	19-32
20	User Exit	ts	
	20.1 What	t Is a User Exit?	20-1
	20.2 Why	Write a User Exit?	20-1
		ut Developing a User Exit	20-2
	20.4 Abou	ut Writing a User Exit	20-2
	20.4.1	Requirements for Variables	20-2
	20.5 EXE	C TOOLS Statements	20-3
	20.5.1	About Writing a Toolset User Exit	20-3
	20.5.2	EXEC TOOLS SET	20-3
	20.5.3	EXEC TOOLS GET	20-3
	20.5.4	EXEC TOOLS SET CONTEXT	20-4
	20.5.5	EXEC TOOLS GET CONTEXT	20-4
	20.5.6	EXEC TOOLS MESSAGE	20-4



	20.6 About Calling a User Exit	20-5
	20.7 About Passing Parameters to a User Exit	20-5
	20.8 About Returning Values to a Form	20-6
	20.8.1 The IAP Constants	20-6
	20.8.2 About Using WHENEVER	20-6
	20.9 An Example of Using User Exits	20-6
	20.10 About Precompiling and Compiling a User Exit	20-7
	20.11 Example Program: A User Exit	20-7
	20.12 About Using the GENXTB Utility	20-9
	20.13 About Linking a User Exit into SQL*Forms	20-9
	20.14 Guidelines	20-9
	20.14.1 About Naming the Exit	20-9
	20.14.2 About Connecting to Oracle	20-10
	20.14.3 About Issuing I/O Calls	20-10
	20.14.4 About Using Host Variables	20-10
	20.14.5 About Updating Tables	20-10
	20.14.6 About Issuing Commands	20-10
Par	rt III Appendixes	
Α	Reserved Words, Keywords, and Namespaces	
	A.1 Reserved Words and Keywords	A-1
	A.2 Oracle Reserved Namespaces	A-3
В	Performance Tuning	
	B.1 What Causes Poor Performance?	B-1
	B.2 How Can Performance Be Improved?	B-2
	B.3 About Using Host Arrays	B-2
	B.4 About Using Embedded PL/SQL	B-2
	B.5 About Optimizing SQL Statements	B-3
	B.5.1 Optimizer Hints	
	Diei Optimizer rimte	B-3
	B.5.2 Trace Facility	B-3 B-4
	·	
	B.5.2 Trace Facility	B-4
	B.5.2 Trace Facility B.6 About Statement Caching	B-4 B-4
	B.5.2 Trace Facility B.6 About Statement Caching B.7 About Using Indexes	B-4 B-4 B-5
	B.5.2 Trace Facility  B.6 About Statement Caching  B.7 About Using Indexes  B.8 About Taking Advantage of Row-Level Locking  B.9 About Eliminating Unnecessary Parsing  B.9.1 About Handling Explicit Cursors	B-4 B-4 B-5 B-5
	B.5.2 Trace Facility  B.6 About Statement Caching  B.7 About Using Indexes  B.8 About Taking Advantage of Row-Level Locking  B.9 About Eliminating Unnecessary Parsing	B-4 B-4 B-5 B-5
	B.5.2 Trace Facility  B.6 About Statement Caching  B.7 About Using Indexes  B.8 About Taking Advantage of Row-Level Locking  B.9 About Eliminating Unnecessary Parsing  B.9.1 About Handling Explicit Cursors	B-4 B-4 B-5 B-5 B-5



B.9.2	.2 Resource Use	B-8
B.9.2	.3 Infrequent Execution	B-8
B.9.2	.4 Frequent Execution	B-9
B.9.2	.5 Embedded PL/SQL Considerations	B-9
B.9.2	.6 Parameter Interactions	B-9
B.10 About	Avoiding Unnecessary Reparsing	B-10
B.11 About	Using Connection Pooling	B-10
B.12 About	Using Oracle Connection Manager in Traffic Director Mode	B-10
Syntactic	and Semantic Checking	
C.1 What I	s Syntactic and Semantic Checking?	C-1
C.2 About	Controlling the Type and Extent of Checking	C-1
C.3 About	Specifying SQLCHECK=SEMANTICS	C-2
C.3.1 A	About Enabling a Semantic Check	C-2
C.3.1	1 About Connecting to the Oracle server	C-3
C.3.1	2 About Using DECLARE TABLE	C-3
C.3.1	3 About Using DECLARE TYPE	C-4
C.4 About	Specifying SQLCHECK=SYNTAX	C-4
C.5 About	Entering the SQLCHECK Option	C-4
System-S	pecific References	
D.1 Systen	n-Specific Information	D-1
D.1.1 L	ocation of Standard Header Files	D-1
D.1.2 A	About Specifying Location of Included Files for the C Compiler	D-1
D.1.3 A	ANSI C Support	D-1
D.1.4 S	Struct Component Alignment	D-1
D.1.5	Size of an Integer and ROWID	D-1
D.1.6 E	Byte Ordering	D-1
D.1.7 A	About Connecting to the Oracle Server	D-2
D.1.8 A	About Linking in an XA Library	D-2
D.1.9 L	ocation of the Pro*C/C++ Executable	D-2
D.1.10	System Configuration File	D-2
	INCLUDE Option Syntax	D-2
	About Compiling and Linking	D-2
	User Exits	D-2
Embedde	d SQL Statements and Directives	
	ary of Precompiler Directives and Embedded SQL Statements	E-3
	The Statement Descriptions	E-6
L.Z ADUUL		



E.3	How	to Read Syntax Diagrams	E-6
E	Ξ.3.1	Required Keywords and Parameters	E-7
E	Ξ.3.2	Optional Keywords and Parameters	E-7
E	Ξ.3.3	Syntax Loops	E-8
E	Ξ.3.4	Multipart Diagrams	E-8
E	E.3.5	Oracle Names	E-8
E	Ξ.3.6	Statement Terminator	E-8
E.4	ALL	DCATE (Executable Embedded SQL Extension)	E-9
E.5	ALL	DCATE DESCRIPTOR (Executable Embedded SQL)	E-10
E.6	CAC	HE FREE ALL (Executable Embedded SQL Extension)	E-11
E.7	CAL	L (Executable Embedded SQL)	E-12
E.8	CLO	SE (Executable Embedded SQL)	E-13
E.9	COL	LECTION APPEND (Executable Embedded SQL Extension)	E-14
E.10	CO	LLECTION DESCRIBE (Executable Embedded SQL Extension)	E-14
E.11	CO	LLECTION GET (Executable Embedded SQL Extension)	E-16
E.12	CO	LLECTION RESET (Executable Embedded SQL Extension)	E-16
E.13	CO	LLECTION SET (Executable Embedded SQL Extension)	E-17
E.14	CO	LLECTION TRIM (Executable Embedded SQL Extension)	E-17
E.15	CO	MMIT (Executable Embedded SQL)	E-18
E.16	CO	NNECT (Executable Embedded SQL Extension)	E-19
E.17	CO	NTEXT ALLOCATE (Executable Embedded SQL Extension)	E-20
E.18	CO	NTEXT FREE (Executable Embedded SQL Extension)	E-21
E.19	CO	NTEXT OBJECT OPTION GET (Executable Embedded SQL Extension)	E-22
E.20	CO	NTEXT OBJECT OPTION SET (Executable Embedded SQL Ext)	E-23
E.21	CO	NTEXT USE (Oracle Embedded SQL Directive)	E-24
E.22	DE	ALLOCATE DESCRIPTOR (Embedded SQL Statement)	E-25
E.23	DE	CLARE CURSOR (Embedded SQL Directive)	E-26
E.24	DE	CLARE DATABASE (Oracle Embedded SQL Directive)	E-27
E.25	DE	CLARE STATEMENT (Embedded SQL Directive)	E-28
E.26	DE	CLARE TABLE (Oracle Embedded SQL Directive)	E-30
E.27	DE	CLARE TYPE (Oracle Embedded SQL Directive)	E-31
E.28	DE	LETE (Executable Embedded SQL)	E-32
E.29	DE	SCRIBE (Executable Embedded SQL Extension)	E-34
E.30	DE	SCRIBE DESCRIPTOR (Executable Embedded SQL)	E-35
E.31	EN.	ABLE THREADS (Executable Embedded SQL Extension)	E-37
E.32	EX	ECUTE END-EXEC (Executable Embedded SQL Extension)	E-38
E.33	EX	ECUTE (Executable Embedded SQL)	E-39
E.34	EX	ECUTE DESCRIPTOR (Executable Embedded SQL)	E-40
E.35	EX	ECUTE IMMEDIATE (Executable Embedded SQL)	E-42
E.36	FE	ГСН (Executable Embedded SQL)	E-43
E.37	FE	TCH DESCRIPTOR (Executable Embedded SQL)	E-45
E.38	FR	EE (Executable Embedded SQL Extension)	E-47



E.39	GET DESCRIPTOR (Executable Embedded SQL)	E-48
E.40	INSERT (Executable Embedded SQL)	E-50
E.41	LOB APPEND (Executable Embedded SQL Extension)	E-53
E.42	LOB ASSIGN (Executable Embedded SQL Extension)	E-53
E.43	LOB CLOSE (Executable Embedded SQL Extension)	E-54
E.44	LOB COPY (Executable Embedded SQL Extension)	E-54
E.45	LOB CREATE TEMPORARY (Executable Embedded SQL Extension)	E-55
E.46	LOB DESCRIBE (Executable Embedded SQL Extension)	E-55
E.47	LOB DISABLE BUFFERING (Executable Embedded SQL Extension)	E-56
E.48	LOB ENABLE BUFFERING (Executable Embedded SQL Extension)	E-56
E.49	LOB ERASE (Executable Embedded SQL Extension)	E-57
E.50	LOB FILE CLOSE ALL (Executable Embedded SQL Extension)	E-57
E.51	LOB FILE SET (Executable Embedded SQL Extension)	E-58
E.52	LOB FLUSH BUFFER (Executable Embedded SQL Extension)	E-58
E.53	LOB FREE TEMPORARY (Executable Embedded SQL Extension)	E-59
E.54	LOB LOAD (Executable Embedded SQL Extension)	E-59
E.55	LOB OPEN (Executable Embedded SQL Extension)	E-60
E.56	LOB READ (Executable Embedded SQL Extension)	E-60
E.57	LOB TRIM (Executable Embedded SQL Extension)	E-61
E.58	LOB WRITE (Executable Embedded SQL Extension)	E-61
E.59	OBJECT CREATE (Executable Embedded SQL Extension)	E-62
E.60	OBJECT DELETE (Executable Embedded SQL Extension)	E-63
E.61	OBJECT DEREF (Executable Embedded SQL Extension)	E-64
E.62	OBJECT FLUSH (Executable Embedded SQL Extension)	E-65
E.63	OBJECT GET (Executable Embedded SQL Extension)	E-65
E.64	OBJECT RELEASE (Executable Embedded SQL Extension)	E-66
E.65	OBJECT SET (Executable Embedded SQL Extension)	E-67
E.66	OBJECT UPDATE (Executable Embedded SQL Extension)	E-68
E.67	OPEN (Executable Embedded SQL)	E-69
E.68	OPEN DESCRIPTOR (Executable Embedded SQL)	E-71
E.69	PREPARE (Executable Embedded SQL)	E-72
E.70	REGISTER CONNECT (Executable Embedded SQL Extension)	E-74
E.71	ROLLBACK (Executable Embedded SQL)	E-75
E.72	SAVEPOINT (Executable Embedded SQL)	E-77
E.73	SELECT (Executable Embedded SQL)	E-78
E.74	SET DESCRIPTOR (Executable Embedded SQL)	E-81
E.75	TYPE (Oracle Embedded SQL Directive)	E-83
E.76	UPDATE (Executable Embedded SQL)	E-84
E.77	VAR (Oracle Embedded SQL Directive)	E-87
E.78	WHENEVER (Embedded SQL Directive)	E-89



#### F Sample Programs F-1 F.1 Sample Program Descriptions F.2 **Building the Demonstration Tables** F-6 F.3 About Building the Sample Programs F-6 F.3.1 Using pcmake.bat F-6 F.4 Using Microsoft Visual Studio F-7 F.5 Setting the Path for the Sample .pre Files F-7 Integrating Pro\*C/C++ into Microsoft Visual Studio .NET G G.1 Integrating Pro\*C/C++ within Microsoft Visual Studio .NET Projects G-1 G-1 G.1.1 Specifying the Location of the Pro\*C/C++ Executable G.1.2 Specifying the Location of the Pro\*C/C++ Header Files G-2 G.2 Adding .pc Files to a Project G-2 G-3 Adding References to .c Files to a Project G.2.2 Adding the Pro\*C/C++ Library to a Project G-3 G.2.3 **Specifying Custom Build Options** G-4

#### Index

G.3 Adding Pro\*C/C++ to the Tools Menu



G-5

#### List of Tables

1-1	precomp Directory Structure	1-5
2-1	Embedded SQL Statements	2-2
2-2	Embedded SQL Statements	2-2
4-1	Oracle Internal Data types	4-1
4-2	Oracle External Datatypes	4-3
4-3	DATE Format	4-7
4-4	C Datatypes for Host Variables	4-14
4-5	C to Oracle Datatype Compatibility	4-15
4-6	Globalization Support Parameters	4-43
5-1	CHAR_MAP Settings	5-2
5-2	Default Type Assignments	5-9
5-3	Header Files	5-30
5-4	SQLLIB Public Functions New Names	5-41
7-1	Legal Datatype Conversions	7-11
8-1	Valid Host Arrays for SELECT INTO	8-11
8-2	Host Arrays Valid in an UPDATE	8-13
8-3	DB2 Array Syntax vs. Oracle Precompiler Syntax	8-26
9-1	Predefined Class Codes	9-4
9-2	SQLSTATE Status Codes	9-6
9-3	SQL Function Codes	9-25
10-1	How Macro Option Values Set Micro Option Values	10-6
10-2	Precompiler Options	10-7
10-3	DBMS and MODE Interaction	10-21
11-1	Command Line Options for Connection Pooling	11-17
12-1	Values and Effects of the PARSE Option	12-3
13-1	Methods for Using Dynamic SQL	13-3
14-1	ANSI SQL Datatypes	14-3
14-2	DYNAMIC Option Settings	14-9
14-3	TYPE_CODE Option Settings	14-9
14-4	Definitions of Descriptor Item Names for GET DESCRIPTOR	14-12
14-5	Oracle Extensions to Definitions of Descriptor Item Names for GET DESCRIPTOR	14-12
14-6	Descriptor Item Names for SET DESCRIPTOR	14-15
14-7	Oracle Extensions to Descriptor Item Names for SET DESCRIPTOR	14-15
15-1	Oracle Internal Datatypes	15-10
15-2	Oracle External Datatypes and Datatype Codes	15-10
15-3	Precision and Scale Values for SQL Datatypes	15-13



16-2	L LOB Access Methods	16-4
16-2	2 Source LOB and Precompiler Datatypes	16-18
16-3	B LOB Attributes	16-21
17-2	Valid Choices for CONTEXT OBJECT OPTION Values	17-15
17-2	Using New Database Types in Pro*C/C++	17-36
17-3	3 Using New C Datatypes in Pro*C/C++	17-37
18-2	Dobject and Collection Attributes	18-5
18-2	Collection and Host Array Allowable Type Conversions	18-5
18-3	Attributes of a COLLECTION DESCRIBE	18-11
19-2	Dobject Datatype Mappings for Object Type Attributes	19-9
19-2	Object Datatype Mappings for Collection Types	19-9
A-1	Oracle Reserved Namespaces	A-4
B-1	HOLD_CURSOR and RELEASE _CURSOR Interactions	B-9
E-1	Functional Summary of the Embedded SQL Statements and Directives	E-3
E-2	Precompiler Directives and Embedded SQL Statements and Clauses	E-3
F-1	Sample Programs	F-1



## **Preface**

This document is a comprehensive user's guide and reference to the Pro\*C/C++. It shows you how to use the database language SQL and Oracle's procedural extension, PL/SQL, in conjunction with Pro\*C/C++ to manipulate data in an Oracle database. It explores a full range of topics, from underlying concepts to advanced programming techniques, and provides code examples.

This Preface contains these topics:

- Intended Audience
- Documentation Accessibility
- Related Documents
- Conventions

#### **Intended Audience**

The *Pro\*C/C++ Developer's Guide* is intended for developers, systems analysts, project managers, and other Oracle users who perform, or are interested in learning about, the following tasks:

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

To use this document, you need a working knowledge of applications programming in C and C++, and familiarity with the use of the Structured Query Language (SQL).

# **Documentation Accessibility**

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

#### **Access to Oracle Support**

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <a href="http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info">http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info</a> or visit <a href="http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs">http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs</a> if you are hearing impaired.

## **Related Documents**

For more information, see these Oracle resources:

- Oracle Database SQL Language Reference
- Oracle C++ Call Interface Developer's Guide



Oracle Call Interface Developer's Guide

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

# Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.



# Changes in Pro\*C/C++

This section lists out the changes in Oracle Pro\*C/C++ starting from Release 19c.

- New Features in Pro\*C/C++ Release 23ai
- New Features in Pro\*C/C++ Release 21c
- New Features in Pro\*C/C++ Release 19c

### New Features in Pro\*C/C++ Release 23ai

The following feature is new in this release:

Support for boolean data type

The Pro\*C Precompiler now supports the querying and binding of the SQL BOOLEAN data type. Using the boolean data type makes it easier to represent a boolean state in applications with a true or false value instead of using a character column (with Y or N).

#### New Features in Pro\*C/C++ Release 21c

The following feature is new in this release:

Support for C99

The Pro\*C/C++ Precompiler supports the C99 standard, the ISO/IEC 9899:1999 standards specification for C programming. C99 syntax and semantics enable application developers to use richer functionality.

### New Features in Pro\*C/C++ Release 19c

The following feature is new in this release:

Support for native compilation with Visual Studio 2017 (VS2017)

Releases starting from version 19.19 and onward now support native compilation of Pro\*C with VS2017 or above. This support removes the earlier restriction in Pro\*C Precompiler of having to use VS2013 for Pro\*C to C conversion, and subsequently VS2017 or later to build the converted C code.



# Part I

# Introduction and Concepts

#### Part I contains the following chapters:

- Introduction
- Precompiler Concepts
- Database Concepts
- Datatypes and Host Variables
- Advanced Topics
- Embedded SQL
- Embedded PL/SQL
- Host Arrays
- Handling Runtime Errors
- Precompiler Options
- Multithreaded Applications



1

# Introduction

This chapter introduces you to the Oracle Pro\*C/C++ Precompiler. You look at its role in developing application programs that manipulate Oracle data and find out what it enables your applications to do. This chapter contains the following topics:

- What is an Oracle Precompiler?
- Why Use the Oracle Pro\*C/C++ Precompiler
- Why Use SQL
- Why Use PL/SQL
- Pro\*C/C++ Precompiler Benefits
- Frequently Asked Questions

# 1.1 What is an Oracle Precompiler?

An Oracle Precompiler is a programming tool that enables the user to embed SQL statements in a high-level source program. As Figure 1-1 shows, the precompiler accepts the source program as input, translates the embedded SQL statements into standard Oracle runtime library calls, and generates a modified source program that you can compile, link, and execute in the usual way.



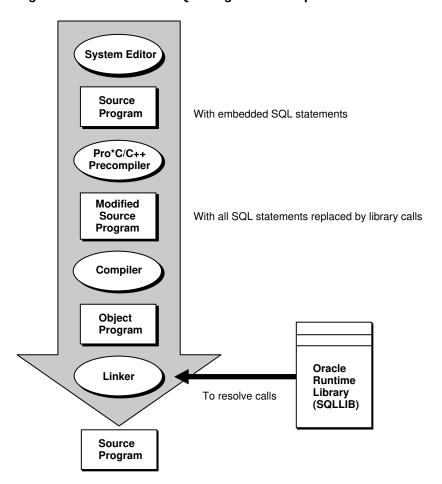


Figure 1-1 Embedded SQL Program Development

# 1.2 Why Use the Oracle Pro\*C/C++ Precompiler

The Oracle Pro\*C/C++ Precompiler lets you use the power and flexibility of SQL in your application programs. A convenient, easy to use interface lets your application access Oracle directly.

Unlike many application development tools, Pro\*C/C++ lets you create highly customized applications. For example, you can create user interfaces that incorporate the latest windowing and mouse technology. You can also create applications that run in the background without the need for user interaction.

Furthermore, Pro\*C/C++ helps you fine-tune your applications. It allows close monitoring of resource use, SQL statement execution, and various runtime indicators. With this information, you can change program parameters for maximum performance.

Although precompiling adds a step to the application development process, it saves time. The precompiler, not you, translates each embedded SQL statement into calls to the Oracle runtime library (SQLLIB). The Pro\*C/C++ precompiler also analyzes host variables, defines mappings of structures into columns, and, with SQLCHECK=FULL, performs semantic analysis of the embedded SQL statements.

# 1.3 Why Use SQL

If you want to access and manipulate Oracle data, you need SQL. Whether you use SQL interactively through SQL\*Plus or embedded in an application program depends on the job at hand. If the job requires the procedural processing power of C or C++, or must be done on a regular basis, use embedded SQL.

SQL has become the database language of choice because it is flexible, powerful, and easy to learn. Being non-procedural, it lets you specify what you want done without specifying how to do it. A few English-like statements make it easy to manipulate Oracle data one row or many rows at a time.

You can execute any SQL (not SQL\*Plus) statement from an application program. For example, you can

- CREATE, ALTER, and DROP database tables dynamically
- SELECT, INSERT, UPDATE, and DELETE rows of data
- COMMIT or ROLLBACK transactions

Before embedding SQL statements in an application program, you can test them interactively using SQL\*Plus. Usually, only minor changes are required to switch from interactive to embedded SQL.

# 1.4 Why Use PL/SQL

An extension to SQL, PL/SQL is a transaction processing language that supports procedural constructs, variable declarations, and robust error handling. Within the same PL/SQL block, you can use SQL and all the PL/SQL extensions.

The main advantage of embedded PL/SQL is better performance. Unlike SQL, PL/SQL provides the ability to group SQL statements logically and send them to Oracle in a block rather than one by one. This reduces network traffic and processing overhead.

#### **Related Topics**

Embedded PL/SQL

# 1.5 Pro\*C/C++ Precompiler Benefits

As Figure 1-2 shows, Pro\*C/C++ offers many features and benefits, which help you to develop effective, reliable applications.



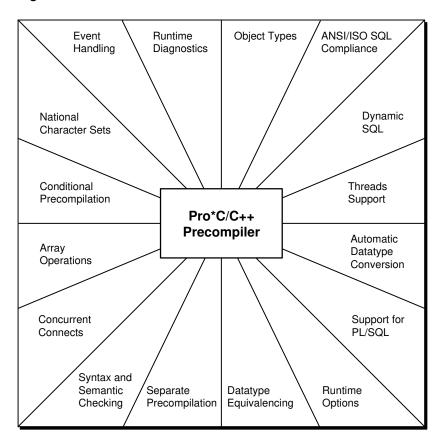


Figure 1-2 Features and Benefits

#### Pro\*C/C++ enables:

- Writing applications in C or C++.
- Following the ANSI/ISO standards for embedding SQL statements in a high-level language.
- Taking advantage of dynamic SQL, an advanced programming technique that lets your Program accept or build any valid SQL statement at runtime.
- Designing and developing highly customized applications.
- Writing shared server process applications.
- Automatically converting between Oracle internal datatypes and high-level language datatypes.
- Improved performance by embedding PL/SQL transaction processing blocks in your application program.
- Specifying useful precompiler options inline and on the command line and change their values during precompilation.
- The use of datatype equivalencing to control the way Oracle interprets input data and formats output data.
- Separately precompiling several program modules, then link them into one executable Program.
- Complete checking of the syntax and semantics of embedded SQL data manipulation statements and PL/SQL blocks.

- Concurrent access to Oracle databases on multiple nodes using Oracle Net.
- The use of arrays as input and output program variables.
- Conditionally precompiling sections of code in your host program so that it can run in different environments.
- Direct interface with SQL\*Forms through the use of user exits written in a high-level language.
- Handling errors and warnings with the SQL Communications Area (SQLCA) and the WHENEVER or DO statement.
- The use of an enhanced set of diagnostics provided by the Oracle Communications Area (ORACA).
- Working with user-defined object types in the database.
- The use of collections (varrays and nested tables) in the database.
- The use of LOBs (Large Objects) in the database.
- The use of National Character Set data stored in the database.
- The use of OCI (Oracle Call Interface) functions in your program.
- The use of multi-threaded applications.
- Microsoft Visual Studio .NET 2002/2003 support.

Pro\*C/C++ is a full-featured tool that supports a professional approach to embedded SQL programming.



Pro\*C/C++ does not support 16-bit code generation.

# 1.6 Directory Structure

When you install Oracle software, a directory structure is created on your hard drive for the Oracle products. A main Oracle directory contains the Oracle subdirectories and files that are necessary to run Pro\*C/C++.

When you install Pro\*C/C++, Oracle Universal Installer creates a directory called \precomp in the ORACLE\_BASE\ORACLE\_HOME directory. This subdirectory contains the Pro\*C/C++ executable files, library files, and sample programs listed in Table 1-1.

Table 1-1 precomp Directory Structure

Directory Name	Contents
\admin	Configuration files
\demo\proc	Sample programs for Pro*C/C++
\demo\sql	SQL scripts for sample programs
\doc\proc	Readme files for Pro*C/C++
\lib\msvc	Library files for Pro*C/C++
\mesg	Message files



Table 1-1 (Cont.) precomp Directory Structure

Directory Name	Contents
\public	Header files



The \precomp directory can contain files for other products, such as Pro\*COBOL.

### 1.6.1 Known Problems, Restrictions, and Workarounds

Although all Windows operating systems allow spaces in file names and directory names, the Oracle Pro\*C/C++ and Oracle Pro\*COBOL precompilers will not precompile files that include spaces in the filename or directory name. For example, do not use the following formats:

- proc iname=test one.pc
- proc iname=d:\dir1\second dir\sample1.pc

# 1.7 Library Files

When linking Pro\*C/C++ applications, you use library files. The Pro\*C/C++ library files are installed as follows:

```
ORACLE_HOME\precomp\LIB\orasq112.1ib

ORACLE_HOME\precomp\LIB\ottclasses.zip

ORACLE_HOME\precomp\LIB\msvc\orasqx12.1ib
```

Pro\*C/C++ application program interface (API) calls are implemented in DLL files provided with your Pro\*C/C++ software. To use the DLLs, you must link your application with the import libraries (.lib files) that correspond to the Pro\*C/C++ DLLs. Also, you must ensure that the DLL files are installed on the computer that is running your Pro\*C/C++ application.

Microsoft provides you with three libraries: libc.lib, libcmt.lib, and msvcrt.lib. The Oracle DLLs use the msvcrt.lib runtime library. You must link the applications with msvcrt.lib instead of the other two Microsoft libraries.

# 1.8 Frequently Asked Questions

This section presents some questions that are frequently asked about Pro\*C/C++, and about Oracle in relation to Pro\*C/C++. The answers are more informal than the documentation in the rest of this Guide, but do provide references to places where you can find the reference material.

### 1.8.1 What is a VARCHAR?

Here is a short description of VARCHARs:



VARCHAR	Description
VARCHAR2	A kind of column in the database that contains variable-length character data. This is what Oracle calls an "internal datatype", because it is a possible column type.
VARCHAR	An Oracle "external datatype" (datatype code 9). You use this only if you are doing dynamic SQL Method 4, or datatype equivalencing.
VARCHAR[n] varchar[n]	This is a Pro*C/C++ "pseudotype" that you can declare as a host variable in your Pro*C/C++ program. It is actually generated by Pro*C/C++ as a struct, with a 2-byte length element, and a [n]-byte character array.

#### **Related Topics**

- Datatypes and Host Variables
- ANSI Dynamic SQL
- Oracle Dynamic SQL: Method 4

#### 1.8.2 Does Pro\*C/C++ Generate Calls to the Oracle Call Interface?

No. Pro\*C/C++ generates data structures and calls to its runtime library: SQLLIB.

## 1.8.3 Why Not Code Using SQLLIB Calls and Not Use Pro\*C/C++?

SQLLIB is not externally documented, is unsupported, and might change from release to release. Also, Pro\*C/C++ is an ANSI/ISO compliant product, that follows the standard requirements for embedded SQL.

SQLLIB is not an API. While it has user-callable functions, it is primarily a runtime library for the precompiler suite of languages.

If you need to do API coding for the database, either use the Oracle Call Interface, the client side API for the Oracle RDBMS, or mix OCI and Pro\*C/C++.

#### **Related Topics**

SQLLIB Extensions for OCI Release 8 Interoperability

#### 1.8.4 Can I Call A PL/SQL Stored Procedure From a Pro\*C/C++ Program?

Certainly. See Embedded PL/SQL. There is a demo program, "About Calling a Stored PL/SQL or Java Subprogram".

#### 1.8.5 Can I Write C++ Code, and Precompile It Using Pro\*C/C++?

Yes. See C++ Applications.

## 1.8.6 Can I Use Bind Variables Anywhere in a SQL Statement?

For example, I would d like to be able to input the name of a table in my SQL statements at runtime. But when I use host variables, I get precompiler errors.

In general, you can use host variables at anywhere in a SQL or PL/SQL, statement where expressions are allowed.

However, the following SQL statement, where table\_name is a host variable, is illegal:

```
EXEC SQL SELECT ename, sal INTO :name, :salary FROM :table name;
```

To solve your problem, you need to use dynamic SQL. There is a demo program that you can adapt to do this, "Example Program: Dynamic SQL Method 1".

#### **Related Topics**

- Host Variable Referencing
- Oracle Dynamic SQL

## 1.8.7 I Am Confused By Character Handling in Pro\*C/C++.

There are many options, but we can simplify. First of all, if you need compatibility with previous precompiler releases, and Oracle7, the safest thing to do is use VARCHAR[n] host variables.

The default datatype for all other character variables in Pro\*C/C++ is CHARZ. Briefly, this means that you must null-terminate the string on input, and it is both blank-padded and null-terminated on output.

In release 8.0, the CHAR\_MAP precompiler option was introduced to specify the default mapping of char variables.

If neither VARCHAR nor CHARZ works for your application, and you need total C-like behavior (null termination, absolutely no blank-padding), use the TYPE command and the C typedef statement, and use datatype equivalencing to convert your character host variables to STRING. There is an example program that shows how to use the TYPE command starting on "Example Program: Using sqlvcp()".

#### **Related Topics**

- VARCHAR Variable Declaration
- CHARZ
- Precompiler Option CHAR MAP
- User-Defined Type Equivalencing

## 1.8.8 Is There Anything Special About Character Pointers?

Yes. When Pro\*C/C++ binds an input or output host variable, it must know the length. When you use VARCHAR[n], or declare a host variable of type char[n], Pro\*C/C++ knows the length from your declaration. But when you use a character pointer as a host variable, and use malloc() to define the buffer in your program, Pro\*C/C++ has no way of knowing the length.

On output you must not only allocate the buffer, but pad it out with some non-null characters, then null-terminate it. On input or output, Pro\*C/C++ calls strlen() for the buffer to get the length.

#### **Related Topics**

Pointer Variables

#### 1.8.9 Why Does SPOOL Not Work in Pro\*C/C++?

SPOOL is a special command used in SQL\*Plus. It is not an embedded SQL command.

#### **Related Topics**

Key Concepts of Embedded SQL Programming

## 1.8.10 Where Can I Find The On-line Versions of the Example Programs?

Each Oracle installation should have a demo directory. If the directory is not there, or it does not contain the example programs, see your system or database administrator.

## 1.8.11 How Can I Compile and Link My Application?

Compiling and linking are very platform specific. Your system-specific Oracle documentation has instructions on how to link a Pro\*C/C++ application. On UNIX systems, there is a makefile called demo\_proc.mk in the demo directory. To link, say, the demo program sample1.pc, you would enter the command line

```
make -f demo proc.mk sample1
```

If you need to use special precompiler options, you can run Pro\*C/C++ separately, then do the make. Or, you can create your own custom makefile. For example, if your program contains embedded PL/SQL code, you can enter

```
proc cv_demo userid=username/password sqlcheck=semantics
make -f demo proc.mk build OBJS=sample1.o EXE=sample1
```

On VMS systems, there is a script called LNPROC that you use to link your Pro\*C/C++ applications.

## 1.8.12 Does Pro\*C/C++ Now Support Using Structures As Host Variables?

How does this work with the array interface?

You can use arrays inside a single structure, or an array of structures with the array interface.

#### **Related Topics**

- Host Structures
- Pointer Variables

# 1.8.13 Is It Possible to Have Recursive Functions In Pro\*C/C++ If I Use Embedded SQL In the Function?

Yes. However, for embedded SQL, you must use cursor variables.

# 1.8.14 Can I Use Any Release of Pro\*C/C++ with Any Version of the Oracle Server?

When you run a precompiler or OCI application against a database server, Oracle recommends that the release of the database server software be equal to or higher than the client software release, but this configuration is not strictly required. For example, if your Oracle Database client software is release 8.1.7, then it is recommended that your Oracle Database server software be release 8.1.7 or higher to run a precompiler application on the client against the server.



More information about upgrading your applications can be found in the *Oracle Database Upgrade Guide*.

# 1.8.15 When My Application Runs, I Keep Getting an Ora-1405 Error (Fetched Column Value Is NULL).

You are selecting a NULL into a host variable that does not have an associated indicator variable. This is not in compliance with the ANSI/ISO standards, and was changed beginning with Oracle7.

If possible, rewrite your program using indicator variables, and use indicators in future development.

Alternatively, if precompiling with MODE=ORACLE and DBMS=V7 or V8, specify UNSAFE NULL=YES on the command line to disable the ORA-01405 message.

#### **Related Topics**

- Indicator Variables
- UNSAFE NULL

## 1.8.16 Are All SQLLIB Functions Private?

No. There are some SQLLIB functions that you can call to get information about your program, or its data. The SQLLIB public functions are shown here:

<b>SQLLIB Public Functions</b>	Description
SQLSQLDAAlloc()	Used to allocate a SQL descriptor array (SQLDA) for dynamic SQL Method 4. See "How is the SQLDA Referenced? ".
SQLCDAFromResultSetCurs or()	Used to convert a Pro*C/C++ cursor variable to an OCI cursor data area. See "New Names for SQLLIB Public Functions".
SQLSQLDAFree()	Used to free a SQLDA allocated using SQLSQLDAAlloc(). See "New Names for SQLLIB Public Functions".
SQLCDAToResultSetCursor()	Used to convert an OCI cursor data area to a Pro*C/C++ cursor variable. See "New Names for SQLLIB Public Functions".
SQLErrorGetText()	Returns a long error message. See "sqlerrm".
SQLStmtGetText()	Used to return the text of the most recently executed SQL statement. See "About Obtaining the Text of SQL Statements".
SQLLDAGetNamed()	Used to obtain a valid Logon Data Area for a named connection, when OCI calls are used in a Pro*C/C++ program. See "New Names for SQLLIB Public Functions".
SQLLDAGetCurrent()	Used to obtain a valid Logon Data Area for the most recent connection, when OCI calls are used in a Pro*C/C++ program. See "New Names for SQLLIB Public Functions".
SQLColumnNullCheck()	Returns an indication of NULL status for dynamic SQL Method 4. See "Handling NULL/Not NULL Datatypes ".
SQLNumberPrecV6()	Returns precision and scale of numbers. See "Extracting Precision and Scale ".
SQLNumberPrecV7()	A variant of SQLNumberPrecV6(). See "Extracting Precision and Scale ".
SQLVarcharGetLength()	Used for obtaining the padded size of a VARCHAR[n]. See "Find the Length of the VARCHAR Array Component".



SQLLIB Public Functions	Description
SQLEnvGet()	Returns the OCI environment handle for a given SQLLIB runtime context. See "SQLEnvGet()".
SQLSvcCtxGet()	Returns the OCI service context for the database connection. See SQLSvcCtxGet().
SQLRowidGet()	Returns the universal ROWID of the last row inserted. See "SQLRowidGet()".
SQLExtProcError()	Returns control to PL/SQL when an error occurs in an external C procedure. See "SQLExtProcError()".

In the preceding list, the functions are thread-safe SQLLIB public functions. Use these functions in all new applications. For more information about these thread-safe public functions (including their old names), see the table "New Names for SQLLIB Public Functions".

## 1.8.17 How Does Oracle Support The New Object Types?

See the chapters Objects and The Object Type Translator for how to use Object types in Pro\*C/C++ applications.

# 1.8.18 Compatibility, Upgrading, and Migration

Pro\*C/C++ adopts a similar compatibility rule to OCI-based applications. This compatibility is subject to the same limitations that OCI imposes on backward compatibility.

The additional "array insert" and "array select" syntax will help migrating DB2 precompiler applications to the Pro\*C/C++ application. This is because you will not need to change DB2 array INSERT and SELECT syntax to that of Oracle Pro\*C/C++.

The "Implicit Buffered Insert" feature supported by Pro\*C/C++ helps you to migrate DB2 precompiler applications to Pro\*C/C++ applications without using the array syntax of Pro\*C/C++ for better performance.



# **Precompiler Concepts**

This chapter explains how embedded SQL programs do their work. You examine the special environment in which they operate and the impact of this environment on the design of your applications. After covering the key concepts of embedded SQL programming and the steps you take in developing an application, this chapter uses a simple program to illustrate the main points.

This chapter contains the following topics:

- Key Concepts of Embedded SQL Programming
- · Steps in Developing an Embedded SQL Application
- Guidelines for Programming
- Example Tables
- Example Program: A Simple Query
- Example Program: A Simple Query using SQL99 Syntax

# 2.1 Key Concepts of Embedded SQL Programming

This section lays the conceptual foundation on which later chapters build. This section contains these topics:

- Embedded SQL Statements
- Embedded SQL Syntax
- Static Versus Dynamic SQL Statements
- Embedded PL/SQL Blocks
- Host and Indicator Variables
- Oracle Datatypes
- Arrays
- Datatype Equivalencing
- Private SQL Areas Cursors and Active Sets
- Transactions
- Errors and Warnings
- SQL99 Syntax Support

#### 2.1.1 Embedded SQL Statements

The term *embedded SQL* refers to SQL statements placed within an application program. Because it houses the SQL statements, the application program is called a *host program*, and the language in which it is written is called the *host language*. For example, Pro\*C/C++ provides the ability to embed certain SQL statements in a C or C++ host program.

To manipulate and query Oracle data, you use the INSERT, UPDATE, DELETE, and SELECT statements. INSERT adds rows of data to database tables, UPDATE modifies rows, DELETE removes unwanted rows, and SELECT retrieves rows that meet your search condition.

The powerful SET ROLE statement lets you dynamically manage database privileges. A *role* is a named group of related system and object privileges, or a named group of related system or object privileges granted to users or other roles. Role definitions are stored in the Oracle data dictionary. Your applications can use the SET ROLE statement to enable and disable roles as needed.

Only SQL statements—not SQL\*Plus statements—are valid in an application program. (SQL\*Plus has additional statements for setting environment parameters, editing, and report formatting.)

#### 2.1.1.1 Executable Statements and Directives

Embedded SQL includes all the interactive SQL statements plus others that allow you to transfer data between Oracle and a host program. There are two types of embedded SQL statements: *executable statements* and *directives*. Executable statements result in calls to the runtime library SQLLIB. You use them to connect to Oracle, to define, query, and manipulate Oracle data, to control access to Oracle data, and to process transactions. They can be placed wherever C or C++ language executable statements can be placed.

Directives, on the other hand, do not result in calls to SQLLIB and do not operate on Oracle data. You use them to declare Oracle objects, communications areas, and SQL variables. They can be placed wherever C or C++ variable declarations can be placed.

Table 2-1 groups the various embedded SQL statements (not a complete list):

Table 2-1 Embedded SQL Statements

DIRECTIVE	PURPOSE
ARRAYLEN*	To use host arrays with PL/SQL
BEGIN DECLARE SECTION*	To declare host variables (optional)
END DECLARE SECTION*	
DECLARE*	To name Oracle schema objects
INCLUDE*	To copy in files
TYPE*	To equivalence datatypes
VAR*	To equivalence variables
WHENEVER*	To handle runtime errors

Table 2-2 Embedded SQL Statements

EXECUTABLE STATEMENT	PURPOSE
ALLOCATE*	To define and control Oracle data
ALTER	-
ANALYZE	-
DELETE	DML
INSERT	-
SELECT	-
UPDATE	-



Table 2-2 (Cont.) Embedded SQL Statements

EXECUTABLE STATEMENT	PURPOSE			
COMMIT	To process transactions			
ROLLBACK	-			
SAVEPOINT	-			
SET TRANSACTION	-			
DESCRIBE*	To use dynamic SQL			
EXECUTE*	-			
PREPARE*	-			
ALTER SESSION	To control sessions			
SET ROLE	-			
*Has no interactive counterpart				

## 2.1.2 Embedded SQL Syntax

In your application program, you can freely mix complete SQL statements with complete C statements and use C variables or structures in SQL statements. The only special requirement for building SQL statements into your host program is that you begin them with the keywords EXEC SQL and end them with a semicolon. Pro\*C/C++ translates all EXEC SQL statements into calls to the runtime library SQLLIB.

Many embedded SQL statements differ from their interactive counterparts only through the addition of a new clause or the use of program variables. The following example compares interactive and embedded ROLLBACK statements:

```
ROLLBACK WORK: -- interactive EXEC SQL ROLLBACK WORK; -- embedded
```

These statements have the same effect, but you would use the first in an interactive SQL environment (such as when running SQL\*Plus), and the second in a Pro\*C/C++ program.

## 2.1.3 Static Versus Dynamic SQL Statements

Most application programs are designed to process static SQL statements and fixed transactions. In this case, you know the makeup of each SQL statement and transaction before runtime; that is, you know which SQL commands will be issued, which database tables might be changed, which columns will be updated, and so on.

However, some applications might be required to accept and process any valid SQL statement at runtime. So, you might not know until runtime all the SQL commands, database tables, and columns involved.

Dynamic SQL is an advanced programming technique that lets your program accept or build SQL statements at run time and take explicit control over datatype conversion.

#### 2.1.4 Embedded PL/SQL Blocks

Pro\*C/C++ treats a PL/SQL block like a single embedded SQL statement. You can place a PL/SQL block anywhere in an application program that you can place a SQL statement. To embed PL/SQL in your host program, you simply declare the variables to be shared with

PL/SQL and bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END-EXEC.

From embedded PL/SQL blocks, you can manipulate Oracle data flexibly and safely because PL/SQL supports all SQL data manipulation and transaction processing commands.

#### 2.1.5 Host and Indicator Variables

Host variables are the key to communication between Oracle and your program. A *host variable* is a scalar or aggregate variable declared in C and shared with Oracle, meaning that both your program and Oracle can reference its value.

Your program uses *input* host variables to pass data to Oracle. Oracle uses *output* host variables to pass data and status information to your program. The program assigns values to input host variables; Oracle assigns values to output host variables.

Host variables can be used anywhere a SQL expression can be used. In SQL statements, host variables must be prefixed with a colon (:) to set them apart from the SQL keywords.

You can also use a C *struct* to contain a number of host variables. When you name the structure in an embedded SQL statement, prefixed with a colon, Oracle uses each of the components of the struct as a host variable.

You can associate any host variable with an optional indicator variable. An *indicator variable* is a short integer variable that "indicates" the value or condition of its host variable. You use indicator variables to assign NULLs to input host variables and to detect NULLs or truncated values in output host variables. A *NULL* is a missing, unknown, or inapplicable value.

In SQL statements, an indicator variable must be prefixed with a colon and immediately follow its associated host variable. The keyword INDICATOR can be placed between the host variable and its indicator for additional clarity.

If the host variables are packaged in a struct, and you want to use indicator variables, you simply create a struct that has an indicator variable for each host variable in the host structure, and name the indicator struct in the SQL statement, immediately following the host variable struct, and prefixed with a colon. You can also use the INDICATOR keyword to separate a host structure and its associated indicator structure.

## 2.1.6 Oracle Datatypes

Typically, a host program inputs data to Oracle, and Oracle outputs data to the program. Oracle stores input data in database tables and stores output data in program host variables. To store a data item, Oracle must know its *datatype*, which specifies a storage format and valid range of values.

Oracle recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle stores data in database columns. Oracle also uses internal datatypes to represent database pseudocolumns, which return specific data items but are not actual columns in a table.

External datatypes specify how data is stored in host variables. When your host program inputs data to Oracle, if necessary, Oracle converts between the external datatype of the input host variable and the internal datatype of the target database column. When Oracle outputs data to your host program, if necessary, Oracle converts between the internal datatype of the source database column and the external datatype of the output host variable.



#### 2.1.7 Arrays

Pro\*C/C++ lets you define array host variables (called *host arrays*) and arrays of structures and operate on them with a single SQL statement. Using the array SELECT, FETCH, DELETE, INSERT, and UPDATE statements, you can query and manipulate large volumes of data with ease. You can also use host arrays inside a host variable **struct**.

## 2.1.8 Datatype Equivalencing

Pro\*C/C++ adds flexibility to your applications by letting you *equivalence* datatypes. That means you can customize the way Oracle interprets input data and formats output data.

On a variable-by-variable basis, you can equivalence supported C datatypes to the Oracle external datatypes. You can also equivalence user-defined datatypes to Oracle external datatypes.

#### 2.1.9 Private SQL Areas, Cursors, and Active Sets

To process a SQL statement, Oracle opens a work area called a *private SQL area*. The private SQL area stores information needed to execute the SQL statement. An identifier called a *cursor* lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.

For static SQL statements, there are two types of cursors: *implicit* and *explicit*. Oracle implicitly declares a cursor for all data definition and data manipulation statements, including SELECT statements (queries) that return only one row. However, for queries that return more than one row, to process beyond the first row, you must explicitly declare a cursor (or use host arrays).

The set of rows returned is called the *active set*; its size depends on how many rows meet the query search condition. You use an explicit cursor to identify the row currently being processed, called the *current row*.

Imagine the set of rows being returned to a terminal screen. A screen cursor can point to the first row to be processed, then the next row, and so on. In the same way, an explicit cursor "points" to the current row in the active set. This allows your program to process the rows one at a time.

### 2.1.10 Transactions

A *transaction* is a series of logically related SQL statements (two UPDATEs that credit one bank account and debit another, for example) that Oracle treats as a unit, so that all changes brought about by the statements are made permanent or undone at the same time.

All the data manipulation statements executed since the last data definition, COMMIT, or ROLLBACK statement was executed make up the current transaction.

To help ensure the consistency of your database, Pro\*C/C++ lets you define transactions using the COMMIT, ROLLBACK, and SAVEPOINT statements.

COMMIT makes permanent any changes made during the current transaction. ROLLBACK ends the current transaction and undoes any changes made since the transaction began. SAVEPOINT marks the current point in the processing of a transaction; used with ROLLBACK, it undoes part of a transaction.



## 2.1.11 Errors and Warnings

When you execute an embedded SQL statement, it either succeeds or fails, and might result in an error or warning. You need a way to handle these results. Pro\*C/C++ provides two error handling mechanisms: the SQL Communications Area (SQLCA) and the WHENEVER statement.

The SQLCA is a data structure that you include (or hard-code) in your host program. It defines program variables used by Oracle to pass runtime status information to the program. With the SQLCA, you can take different actions based on feedback from Oracle about work just attempted. For example, you can check to see if a DELETE statement succeeded and, if so, how many rows were deleted.

With the WHENEVER statement, you can specify actions to be taken automatically when Oracle detects an error or warning condition. These actions are: continuing with the next statement, calling a function, branching to a labeled statement, or stopping.

## 2.1.12 SQL99 Syntax Support

The SQL standard enables the portability of SQL applications across all conforming software products. Oracle features are compliant with the ANSI/ISO SQL99 standard, including ANSI compliant joins. Pro\*C/C++ supports all SQL99 features that are supported by Oracle database, which means that the SQL99 syntax for the SELECT, INSERT, DELETE, and UPDATE statements and the body of the cursor in a DECLARE CURSOR statement are supported.

# 2.2 Steps in Developing an Embedded SQL Application

Figure 2-1 shows the embedded SQL application development process.



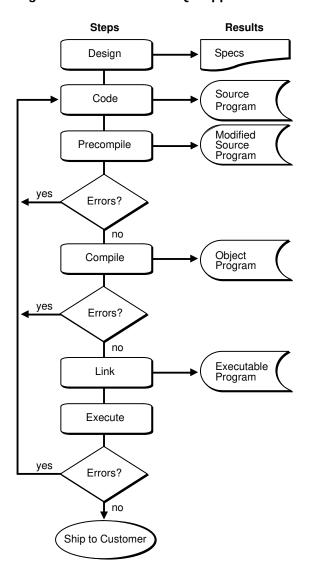


Figure 2-1 Embedded SQL Application Development Process

As you can see, precompiling results in a modified source file that can be compiled normally. Though precompiling adds a step to the traditional development process, that step lets you write very flexible applications.

# 2.3 Guidelines for Programming

This section deals with embedded SQL syntax, coding conventions, and C-specific features and restrictions. Topics are arranged alphabetically for quick reference.

#### 2.3.1 Comments

You can place C-style Comments (/\* ... \*/) in a SQL statement wherever blanks can be placed (except between the keywords EXEC SQL). Also, you can place ANSI-style Comments (-- ...) within SQL statements at the end of a line, as the following example shows:

```
EXEC SQL SELECT ENAME, SAL
    INTO :emp_name, :salary -- output host variables
```

```
FROM EMP
WHERE DEPTNO = :dept_number;
```

You can use C++ style Comments (//) in your Pro\*C/C++ source if you precompile using the CODE=CPP precompiler option.

## 2.3.2 Constants

An *L* or *l* suffix specifies a **long** integer constant, a *U* or *u* suffix specifies an **unsigned** integer constant, a *OX* or *Ox* prefix specifies a hexadecimal integer constant, and an *F* or *f* suffix specifies a **float** floating-point constant. These forms are *not* allowed in SQL statements.

#### 2.3.3 C99 Support

Starting with release 21c, the Pro\*C/C++ Precompiler supports the C99 standard, the ISO/IEC 9899:1999 standards specification for C programming.

C99 introduced the following new features:

- Inline functions
- Intermingled declarations and code
- Several new data types including long long int, optional extended integer types, an explicit boolean data type, and a complex type to represent complex numbers
- Variable length arrays
- Flexible array members
- Support for one-line comments beginning with //
- Designated initializers (for example, initializing a structure by field names: struct point  $p = \{ x = 1, y = 2 \};$ )
- Compound literals (for instance, it is possible to construct structures in function calls: function((struct x) {1, 2}))
- Support for variadic (macros with a variable number of arguments)
- Restrict qualification
- Keyword static in array indices in parameter declarations

#### 2.3.4 Declare Section

A *Declare Section* contains the host variable declarations and is of the form:

```
EXEC SQL BEGIN DECLARE SECTION;
/* Declare all host variables inside this section: */
    char *uid = "username/password";
    ...
EXEC SQL END DECLARE SECTION;
```

#### A Declare Section begins with the statement:

```
EXEC SQL BEGIN DECLARE SECTION;
```

#### and ends with the statement:

```
EXEC SQL END DECLARE SECTION;
```



Between these two statements only the following are allowed:

- Host-variable and indicator-variable declarations
- Non-host C/C++ variables
- EXEC SQL DECLARE statements
- EXEC SQL INCLUDE statements
- EXEC SQL VAR statements
- EXEC SQL TYPE statements
- EXEC ORACLE statements
- C/C++ comments

A Declare Section is required when MODE=ANSI or CODE=CPP (in a C++ application) or PARSE=NONE or PARTIAL.

More than one Declare Section is allowed. They can be in different code modules.

#### **Related Topics**

About Parsing Code

#### 2.3.5 Delimiters

While C uses single quotes to delimit single characters, as in

```
ch = getchar();
switch (ch)
{
case 'U': update(); break;
case 'I': insert(); break;
```

SQL uses single quotes to delimit character strings, as in

```
EXEC SQL SELECT ENAME, SAL FROM EMP WHERE JOB = 'MANAGER';
```

While C uses double quotes to delimit character strings, as in

```
printf("\nG'Day, mate!");
```

SQL uses double quotes to delimit identifiers containing special or lowercase characters, as in

```
EXEC SQL CREATE TABLE "Emp2" (empno number(4), ...);
```

## 2.3.6 File Length

Pro\*C/C++ cannot process arbitrarily long source files. There is a limit to the number of lines allowed. The following aspects of the source file are contributing factors to the file-size constraint:

- Complexity of the embedded SQL statements (for example, the number of bind and define variables).
- Whether a database name is used (for example, connecting to a database with an AT clause).
- Number of embedded SQL statements.

To prevent problems related to this limitation, use multiple program units to sufficiently reduce the size of the source files.

## 2.3.7 Function Prototyping

The ANSI C standard (X3.159-1989) provides for function prototyping. A *function prototype* declares a function and the data types of its arguments, so that the C compiler can detect missing or mismatched arguments.

The CODE option, which you can enter on the command line or in a configuration file, determines the way that the precompiler generates C or C++ code.

#### 2.3.7.1 ANSI C

When you precompile your program with CODE=ANSI\_C, the precompiler generates fully prototyped function declarations. For example:

```
extern void sqlora(long *, void *);
```

#### 2.3.7.2 KR C

When you precompile with the option CODE=KR\_C (KR for "Kernighan and Ritchie"), the precompiler generates function prototypes in the same way that it does for ANSI\_C, except that function parameter lists are commented out. For example:

```
extern void sqlora(/*_ long *, void * _*/);
```

So, make sure to set the precompiler option CODE to KR\_C if you use a C compiler that does not support ANSI C. When the CODE option is set to ANSI\_C, the precompiler can also generate other ANSI-specific constructs; for example, the **const** type qualifier.

#### 2.3.7.3 CPP

When you compile with CODE=CPP you will generate C++ compatible function prototypes. Use this option setting with C++ compilers.

#### **Related Topics**

C++ Applications

#### 2.3.8 Hint Length

Maximum length of a sql hint in an embedded sql statement is limited to 256 characters. Any hint exceeding this limit will be truncated.

#### 2.3.9 Host Variable Names

Host variable names can consist of upper or lowercase letters, digits, and underscores, but must begin with a letter. They can be any length, but only the first 31 characters are significant to Pro\*C/C++. Your C compiler or linker might require a shorter maximum length, so check your C compiler user's guide.

For portability, you may wish to restrict the length of host variable names to 18 or fewer characters (the length mandated by the SQL standard).

#### **Related Topics**

Reserved Words, Keywords, and Namespaces

#### 2.3.10 Line Continuation

You can continue SQL statements from one line to the next. You must use a backslash (\) to continue a string literal from one line to the next, as the following example shows:

```
EXEC SQL INSERT INTO dept (deptno, dname) VALUES (50, 'PURCHAS\ ING');
```

In this context, the precompiler treats the backslash as a continuation character.

## 2.3.11 Line Length

The maximum line length is 4096 for lines consisting of only ASCII characters, or 1024 for multibyte characters.

#### 2.3.12 MAXLITERAL Default Value

The precompiler option MAXLITERAL lets you specify the maximum length of string literals generated by the precompiler. The MAXLITERAL default value is 1024. Specify a smaller value if required. For example, if your C compiler cannot handle string literals longer than 512 characters, you then specify MAXLITERAL=512. Check your C compiler user's guide.

## 2.3.13 Operators

The logical operators and the "equal to" relational operator are different in C and SQL, as the following list shows. These C operators are *not* allowed in SQL statements:

SQL Operator	C Operator
NOT	!
AND	&&
OR	II
=	==

The following C operators also *not* allowed in SQL statements:

Туре	C Operator
address	&
bitwise	&,  , ^, ~
compound assignment	+=, -=, *=, and so on.
conditional	?:
decrement	
increment	++
indirection	*
modulus	%
shift	>>,<<



#### 2.3.14 Statement Terminator

Embedded SQL statements are always terminated by a semicolon, as the following example shows:

```
EXEC SQL DELETE FROM emp WHERE deptno = :dept number;
```

# 2.4 Conditional Precompilation

Conditional precompilation includes (or excludes) sections of code in your host program based on certain conditions. For example, you might want to include one section of code when precompiling under UNIX and another section when precompiling under VMS. Conditional precompilation lets you write programs that can run in different environments.

Conditional sections of code are marked by statements that define the environment and actions to take. You can code C or C++ statements as well as EXEC SQL statements in these sections. The following statements let you exercise conditional control over precompilation:

```
EXEC ORACLE DEFINE symbol; -- define a symbol
EXEC ORACLE IFDEF symbol; -- if symbol is defined
EXEC ORACLE IFNDEF symbol; -- if symbol is not defined
EXEC ORACLE ELSE; -- otherwise
EXEC ORACLE ENDIF; -- end this control block
```

All EXEC ORACLE statements must be terminated with a semi-colon.

#### 2.4.1 Symbol Definition

You can define a symbol in two ways. Either include the statement:

```
EXEC ORACLE DEFINE symbol;
```

in your host program or define the symbol on the command line using the syntax:

```
... DEFINE=symbol ...
```

where symbol is not case-sensitive.



The #define preprocessor directive is not the same as the EXEC ORACLE DEFINE statement.

Some port-specific symbols are predefined for you when Pro\*C/C++ is installed on your system. For example, predefined operating symbols include CMS, MVS, MS-DOS, UNIX, and VMS.

#### 2.4.2 Example SELECT Statement

In the following example, the SELECT statement is precompiled only when the symbol site2 is defined:

```
EXEC ORACLE IFDEF site2;
EXEC SQL SELECT DNAME
```



```
INTO :dept_name
    FROM DEPT
    WHERE DEPTNO= :dept_number;
EXEC ORACLE ENDIF;
```

You can "comment out" C, C++, or embedded SQL code by placing it between IFDEF and ENDIF and **not** defining the symbol.

# 2.5 Precompile Separately

You can precompile several C or C++ program modules separately, then link them into one executable program. This supports modular programming, which is required when the functional components of a program are written and debugged by different developers. The individual program modules need not be written in the same language.

#### 2.5.1 Guidelines

The following guidelines will help you avoid some common problems.

#### 2.5.1.1 Referencing Cursors

Cursor names are SQL identifiers, whose scope is the precompilation unit. Hence, cursor operations cannot span precompilation units (files). That is, you cannot DECLARE a cursor in one file, and OPEN or FETCH from it in another file. So, when doing a separate precompilation, make sure all definitions and references to a given cursor are in one file.

#### 2.5.1.2 Specifying MAXOPENCURSORS

When you precompile the program module that CONNECTs to Oracle, specify a value for MAXOPENCURSORS that is high enough for any of the program modules. If you use MAXOPENCURSORS for another program module, one that does not do a CONNECT, then that value for MAXOPENCURSORS is ignored. Only the value in effect for the CONNECT is used at runtime.

#### 2.5.1.3 Use a Single SQLCA

If you want to use just one SQLCA, you must declare it as global in one of the program modules and as external in the other modules. Use the *extern* storage class, and the following define in your code:

```
#define SQLCA STORAGE CLASS extern
```

which tells the precompiler to look for the SQLCA in another program module. Unless you declare the SQLCA as external, each program module uses its own local SQLCA.



All source files in an application must be uniquely named, or else an error will be generated.

# 2.6 Compile and Link

To get an executable program, you must compile the output .c source files produced by the precompiler, then link the resulting object modules with modules needed from SQLLIB and system-specific Oracle libraries. If you are mixing precompiler code and OCI calls, be sure to also link in the OCI runtime library (liboci.a on UNIX systems).

The linker resolves symbolic references in the object modules. If these references conflict, the link fails. This can happen when you try to link third-party software into a precompiled program. Not all third-party software is compatible with Oracle. So, linking your program *shared* might cause an obscure problem. In some cases, linking *standalone* or *two-task* might solve the problem.

Compiling and linking are system dependent. On most platforms, example *makefiles* or batch files are supplied that you can use to precompile, compile, and link a Pro\*C/C++ application. See your system-specific documentation.

# 2.7 Example Tables

Most programming examples in this guide use two example database tables: DEPT and EMP. Their definitions follow:

```
CREATE TABLE DEPT
   (DEPTNO NUMBER (2) NOT NULL,
    DNAME VARCHAR2 (14),
    LOC
           VARCHAR2(13))
CREATE TABLE EMP
   (EMPNO NUMBER (4) NOT NULL,
    ENAME VARCHAR2(10),
    JOB
            VARCHAR2(9),
    MGR
            NUMBER(4),
    HIREDATE DATE,
    SAL
             NUMBER (7,2),
    COMM
             NUMBER (7,2),
    DEPTNO
             NUMBER(2))
```

LOC

#### 2.7.1 Example Data

Respectively, the DEPT and EMP tables contain the following rows

#### of data:

DEPTNO DNAME

10 20 30 40	ACCOU RESEA SALES OPERA		AS AGO				
EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	30
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		30



7782	CLARK	MANAGER	7839	09-JUN-81	2450	10
7788	SCOTT	ANALYST	7566	19-APR-87	3000	20
7839	KING	PRESIDENT		17-NOV-81	5000	10
7844	TURNER	SALESMAN	7698	08-SEP-81	1500	30
7876	ADAMS	CLERK	7788	23-MAY-87	1100	20
7900	JAMES	CLERK	7698	03-DEC-81	950	30
7902	FORD	ANALYST	7566	03-DEC-81	3000	20
7934	MILLER	CLERK	7782	23-JAN-82	1300	10

# 2.8 Example Program: A Simple Query

One way to get acquainted with Pro\*C/C++ and embedded SQL is to study a program example. The following program is also available on-line in the file sample1.pc in your Pro\*C/C++ demo directory.

The program connects to Oracle, then loops, prompting the user for an employee number. It queries the database for the employee's name, salary, and commission, displays the information, and then continues the loop. The information is returned to a host structure. There is also a parallel indicator structure to signal whether any of the output values SELECTed might be NULL.

Precompile example programs using the precompiler option MODE=ORACLE.

#### Note:

For simplicity in demonstrating this feature, this example does not perform the password management techniques that a deployed system normally uses. In a production environment, follow the Oracle Database password management guidelines, and disable any sample accounts. See *Oracle Database Security Guide* for password management guidelines and other security recommendations.

```
sample1.pc
 * Prompts the user for an employee number,
 * then queries the emp table for the employee's
  name, salary and commission. Uses indicator
   variables (in an indicator struct) to determine
   if the commission is NULL.
#include <stdio.h>
#include <string.h>
/* Define constants for VARCHAR lengths. */
#define UNAME LEN
#define PWD LEN
                          40
/* Declare variables. No declare section is needed if MODE=ORACLE.*/
VARCHAR username[UNAME LEN];
/* VARCHAR is an Oracle-supplied struct */
varchar password[PWD LEN];
/* varchar can be in lower case also. */
Define a host structure for the output values of a SELECT statement.
```

```
struct {
   VARCHAR emp_name[UNAME_LEN];
    float
            salary;
             commission;
   float
} emprec;
Define an indicator struct to correspond to the host output struct. */
struct
   short
            emp_name_ind;
   short
           sal_ind;
   short
            comm ind;
} emprec ind;
/* Input host variable. */
int
         emp number;
          total queried;
/* Include the SQL Communications Area.
  You can use #include or EXEC SQL INCLUDE. */
#include <sqlca.h>
/* Declare error handling function. */
void sql error();
main()
   char temp char[32];
/* Connect to ORACLE--
 * Copy the username into the VARCHAR.
    strncpy((char *) username.arr, "SCOTT", UNAME LEN);
/* Set the length component of the VARCHAR. */
   username.len = strlen((char *) username.arr);
/* Copy the password. */
    strncpy((char *) password.arr, "TIGER", PWD LEN);
    password.len = strlen((char *) password.arr);
/* Register sql_error() as the error handler. */
    EXEC SQL WHENEVER SQLERROR DO sql error("ORACLE error--\n");
/* Connect to ORACLE. Program will call sql error()
 * if an error occurs when connecting to the default database.
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user: %s\n", username.arr);
/* Loop, selecting individual employee's results */
    total queried = 0;
    for (;;)
/* Break out of the inner loop when a
 * 1403 ("No data found") condition occurs.
        EXEC SQL WHENEVER NOT FOUND DO break;
        for (;;)
        {
            emp number = 0;
            printf("\nEnter employee number (0 to quit): ");
            gets(temp char);
            emp number = atoi(temp_char);
            if (emp number == 0)
               break;
```

```
EXEC SQL SELECT ename, sal, NVL(comm, 0)
                INTO :emprec INDICATOR :emprec ind
                FROM EMP
                WHERE EMPNO = :emp_number;
/* Print data. */
           printf("\n\nEmployee\tSalary\t\tCommission\n");
           printf("----\t\t----\n");
/* Null-terminate the output string data. */
           emprec.emp name.arr[emprec.emp name.len] = '\0';
           printf("%-8s\t%6.2f\t\t",
                emprec.emp_name.arr, emprec.salary);
            if (emprec_ind.comm_ind == -1)
               printf("NULL\n");
           else
               printf("%6.2f\n", emprec.commission);
           total queried++;
        } /* end inner for (;;) */
       if (emp number == 0) break;
       printf("\nNot a valid employee number - try again.\n");
    } /* end outer for(;;) */
    printf("\n\nTotal rows returned was %d.\n", total queried);
   printf("\nG'day.\n\n\n");
/* Disconnect from ORACLE. */
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
void sql error(msg)
char *msg;
   char err msg[128];
   int buf_len, msg_len;
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   printf("\n%s\n", msg);
   buf len = sizeof (err msg);
    sqlglm(err msg, &buf len, &msg len);
   if (msg len > buf len)
   msg len = buf len;
   printf("%.*s\n", msg len, err msg);
   EXEC SQL ROLLBACK RELEASE;
    exit(1);
```

# 2.9 Example Program: A Simple Query using SQL99 Syntax

This program is similar to the previous example, but uses SQL99 syntax for SELECT, INSERT, DELETE and UPDATE statements and the body of the cursor in a DECLARE CURSOR statement is supported.

Precompile example programs using the precompiler option MODE=ORACLE.

```
/*
    * sq199.pc
    *
    Prompts the user for an employee number,
    then queries the emp table for the employee's
    name, salary and department.
    *
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
/* Define constants for VARCHAR lengths. */
#define UNAME_LEN 30
#define
           PWD LEN
/* Declare variables. No declare section is needed if MODE=ORACLE. */
VARCHAR
           username[UNAME LEN];
/* VARCHAR is an Oracle-supplied struct */
varchar password[PWD LEN];
 /* varchar can be in lower case also. */
^{\prime\prime} Define a host structure for the output values of a SELECT statement. ^{*\prime}
struct{
  VARCHAR emp name[UNAME_LEN];
   float
            salarv;
  VARCHAR dept name [UNAME LEN] ;
  } emprec;
/* Define an indicator struct to correspond to the host output struct. */
struct{
             emp_name ind;
   short
            sal ind;
  short
             dept name;
  short
  } emprec ind;
/* Input host variable. */
        emp_number;
int
int
            total_queried;
/\star Include the SQL Communications Area. You can use #include or EXEC SQL
INCLUDE. */
#include <sqlca.h>
/* Declare error handling function. */
void sql error(msg)
   char *msg;
    char err msg[128];
    size t buf len, msg len;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\n%s\n", msg);
    buf len = sizeof (err msg);
    sqlglm(err msg, &buf len, &msg len);
    printf("%.*s\n", msg len, err msg);
    EXEC SQL ROLLBACK RELEASE;
  exit(EXIT FAILURE);
void main(){
  char temp char[32];
  ^{\prime \star} Connect to ORACLE-- ^{\star} Copy the username into the VARCHAR. ^{\star}/
   strncpy((char *) username.arr, "scott", UNAME LEN);
  /* Set the length component of the VARCHAR. */
   username.len = (unsigned short) strlen((char *) username.arr);
   /* Copy the password. */
     strncpy((char *) password.arr, "tiger", PWD_LEN);
     password.len = (unsigned short) strlen((char *) password.arr);
  /* Register sql error() as the error handler. */
```

```
EXEC SQL WHENEVER SQLERROR DO sql error("ORACLE error--\n");
  /* Connect to ORACLE. Program will call sql error() * if an error occurs
when connecting to the default database. */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
   printf("\nConnected to ORACLE as user: %s\n", username.arr);
  /* Loop, selecting individual employee's results */
    total queried = 0;
    for (;;)
                 emp number = 0;
                 printf("\nEnter employee number (0 to quit): ");
                 gets(temp char);
                 emp_number = atoi(temp_char);
                 if (emp number == 0)
                       break;
  ^{\prime\star} Branch to the notfound label when the ^{\star} 1403 ("No data found") condition
occurs. */
  EXEC SQL WHENEVER NOT FOUND GOTO notfound;
  /* The following query uses SQL99 syntax - RIGHT OUTER JOIN */
   EXEC SQL SELECT e.ename, e.sal, d.dname
   INTO :emprec INDICATOR :emprec ind
   FROM EMP e RIGHT OUTER JOIN dept d
   ON e.deptno = d.deptno
   WHERE e.EMPNO = :emp number;
  /* Print data. */
 printf("\n\nEmployee Salary
                                Department Name\n");
 printf("-----\n");
  /* Null-terminate the output string data. */
   emprec.emp name.arr[emprec.emp name.len] = '\0';
   emprec.dept name.arr[emprec.dept name.len]='\0';
   printf("%s %7.2f %s", emprec.emp name.arr,
  emprec.salary, emprec.dept name.arr);
   total queried++;
    continue;
 not.found:
    printf("\nNot a valid employee number - try again.\n");
 printf("\n\nTotal rows returned was %d.\n", total queried);
printf("\nG'day.\n\n\n");
/* Disconnect from ORACLE. */
  EXEC SQL ROLLBACK WORK RELEASE;
exit(EXIT SUCCESS);
```

# **Database Concepts**

This chapter explains some basic database concepts and how to perform transaction processing. You learn the basic techniques that safeguard the consistency of your database, including how to control if changes to Oracle data are made permanent or undone.

This chapter contains the following topics:

- Connect to the Database
- Advanced Connection Options
- Definitions of Transactions Terms
- How Transactions Guard Your Database
- How to Begin and End Transactions
- Using the COMMIT Statement
- Using the SAVEPOINT Statement
- The ROLLBACK Statement
- The RELEASE Option
- The SET TRANSACTION Statement
- Override Default Locking
- Fetch Across COMMITs
- Distributed Transactions Handling
- Guidelines

#### 3.1 Connect to the Database

The complete syntax of the CONNECT statement will be discussed in the next few sections. Here it is:

```
EXEC SQL CONNECT { :user IDENTIFIED BY :oldpswd | :usr_psw }
  [[ AT { dbname | :host_variable }] USING :connect_string ]
   [ {ALTER AUTHORIZATION :newpswd | IN { SYSDBA | SYSOPER | SYSBACKUP } MODE} ] ;
```

Your Pro\*C/C++ program must connect to the database before querying or manipulating data. To log on, simply use the CONNECT statement

```
EXEC SQL CONNECT :username IDENTIFIED BY :password ;
```

where username and password are char or VARCHAR host variables.

Or, you can use the statement

```
EXEC SQL CONNECT :usr pwd;
```

where the host variable *usr\_pwd* contains your username and password separated by a slash character (/).

These are simplified subsets of the CONNECT statement.

The CONNECT statement must be the first SQL statement executed by the program. That is, other SQL statements can physically but not logically precede the CONNECT statement in the precompilation unit.

To supply the Oracle username and password separately, you define two host variables as character strings or VARCHARs. (If you supply a username containing both username and password, only one host variable is needed.)

Make sure to set the username and password variables before the CONNECT is executed, or it will fail. Your program can prompt for the values, or you can hard-code them as follows:

```
char *username = "SCOTT";
char *password = "TIGER";
...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL CONNECT :username IDENTIFIED BY :password;
```

However, you cannot hard-code a username and password into the CONNECT statement. You also cannot use quoted literals. For example, both of the following statements are *invalid*:

```
EXEC SQL CONNECT SCOTT IDENTIFIED BY TIGER;
EXEC SQL CONNECT 'SCOTT' IDENTIFIED BY 'TIGER';
```

Hard coding usernames and passwords is not recommended practise.

#### **Related Topics**

CONNECT (Executable Embedded SQL Extension)

## 3.1.1 Using the ALTER AUTHORIZATION Clause to Change Passwords

Pro\*C/C++ provides client applications with a convenient way to change a user password at runtime through a simple extension to the EXEC SQL CONNECT statement.

This section describes the possible outcomes of different variations of the ALTER AUTHORIZATION clause.

#### 3.1.1.1 Standard CONNECT

If an application issues the following statement

```
EXEC SQL CONNECT ..; /* No ALTER AUTHORIZATION clause */
```

it performs a normal connection attempt. The possible results include the following:

- The application will connect without issue.
- The application will connect, but will receive a password warning. The warning indicates
  that the password has expired but is in a grace period which will allow Logons. At this
  point, the user is encouraged to change the password before the account becomes locked.
- The application will fail to connect. Possible causes include the following:
  - The password is incorrect.
  - The account has expired, and is possibly in a locked state.

#### 3.1.1.2 Change Password on CONNECT

The following CONNECT statement



```
EXEC SQL CONNECT .. ALTER AUTHORIZATION :newpswd;
```

indicates that the application wants to change the account password to the value indicated by newpswd. After the change is made, an attempt is made to connect as user/newpswd. This can have the following results:

- The application will connect without issue
- The application will fail to connect. This could be due to either of the following:
  - Password verification failed for some reason. In this case the password remains unchanged.
  - The account is locked. Changes to the password are not permitted.

## 3.1.2 Connecting Using Oracle Net Services

To connect using an Oracle Net Services driver, substitute a service name, as defined in your tnsnames.ora configuration file or in Oracle Names.

If you are using Oracle Names, the name server obtains the service name from the network definition database.

See *Oracle Net Services Administrator's Guide* for more information about Oracle Net Services.

#### 3.1.3 Automatic Connects

You can automatically connect to Oracle with the username

```
CLUSTER$username
```

where username is the current operating system username, and CLUSTER\$username is a valid Oracle database username. (The actual value for CLUSTER\$ is defined in the INIT.ORA parameter file.) You simply pass to the Pro\*C/C++ Precompiler a slash character, as follows:

```
char *oracleid = "/";
...
EXEC SQL CONNECT :oracleid;
```

This automatically connects you as user CLUSTER\$username. For example, if your operating system username is RHILL, and CLUSTER\$RHILL is a valid Oracle username, connecting with '/' automatically logs you on to Oracle as user CLUSTER\$RHILL.

You can also pass a '/' in a string to the precompiler. However, the string cannot contain trailing blanks. For example, the following CONNECT statement will fail:

```
char oracleid[10] = "/ ",
...
EXEC SQL CONNECT :oracleid;
```

#### 3.1.3.1 The AUTO\_CONNECT Precompiler Option

If AUTO\_CONNECT=YES, and the application is not already connected to a database when it processes the first executable SQL statement, it attempts to connect using the userid

```
CLUSTER$<username>
```



where *username* is your current operating system user or task name and CLUSTER\$username is a valid Oracle userid. The default value of AUTO\_CONNECT is NO.

When AUTO\_CONNECT=NO, you must use the CONNECT statement in your program to connect to Oracle.

#### 3.1.3.2 SYSDBA, SYSOPER, or SYSBACKUP System Privileges

Append the following optional string after all other clauses to log on with either SYSDBA, SYSOPER, or SYSBACKUP system privileges:

```
[IN { SYSDBA | SYSOPER | SYSBACKUP } MODE]
```

#### For example:

```
EXEC SQL CONNECT ... IN SYSDBA MODE ;
```

Here are the restrictions that apply to this option:

- This option is not permitted when using the AUTO\_CONNECT=YES precompiler option setting.
- This option is not permitted when using the ALTER AUTHORIZATION keywords in the CONNECT statement.

#### **Related Topics**

Using the ALTER AUTHORIZATION Clause to Change Passwords

# 3.2 Advanced Connection Options

This section describes the available options for advanced connections.

#### 3.2.1 Some Preliminaries

The communicating points in a network are called *nodes*. Oracle Net lets you transmit information (SQL statements, data, and status codes) over the network from one node to another.

A *protocol* is a set of rules for accessing a network. The rules establish such things as procedures for recovering after a failure and formats for transmitting data and checking errors.

The Oracle Net syntax for connecting to the default database in the local domain is simply to use the service name for the database.

If the service name is not in the default (local) domain, you must use a global specification (all domains specified). For example:

HR.XX.ORACLE.COM

## 3.2.2 Concurrent Logons

Pro\*C/C++ supports distributed processing through Oracle Net. Your application can concurrently access any combination of local and remote databases or make multiple connections to the same database. In Figure 3-1, an application program communicates with one local and three remote Oracle databases. ORA2, ORA3, and ORA4 are simply logical names used in CONNECT statements.



Application Local Program Oracle Database **Oracle Net** ORA2 ORA4 ORA3 Remote Remote Oracle Oracle Database Remote Database Oracle Database

Figure 3-1 Connecting through Oracle Net

By eliminating the boundaries in a network between different machines and operating systems, Oracle Net provides a distributed processing environment for Oracle tools. This section shows you how Pro\*C/C++ supports distributed processing through Oracle Net. You learn how your application can

- Directly or indirectly access other databases
- Concurrently access any combination of local and remote databases
- Make multiple connections to the same database

For details on installing Oracle Net and identifying available databases, see Identifying and Accessing the Database and your system-specific Oracle documentation.

#### 3.2.3 Default Databases and Connections

Each node has a *default* database. If you specify a database name, but no domain in your CONNECT statement, you connect to the default database on the named local or remote node.

A default connection is made by a CONNECT statement that has no AT clause. The connection can be to any default or nondefault database at any local or remote node. SQL statements without an AT clause are executed against the default connection. Conversely, a nondefault connection is made by a CONNECT statement that has an AT clause. SQL statements with an AT clause are executed against the nondefault connection.

All database names must be unique, but two or more database names can specify the same connection. That is, you can have multiple connections to any database on any node.

## 3.2.4 Explicit Connections

Usually, you establish a connection to Oracle as follows:

EXEC SQL CONNECT :username IDENTIFIED BY :password;



#### You can also use

```
EXEC SQL CONNECT :usr pwd;
```

where usr\_pwd contains username/password.

You can automatically connect to Oracle with the userid

```
CLUSTER$username
```

where *username* is your current operating system user or task name and CLUSTER\$*username* is a valid Oracle userid. You simply pass to the precompiler a slash (/) character, as follows:

```
char oracleid = '/';
...
EXEC SQL CONNECT :oracleid;
```

This automatically connects you as user CLUSTER\$username.

If you do not specify a database and node, you are connected to the default database at the current node. If you want to connect to a different database, you must explicitly identify that database.

With *explicit connections*, you connect to another database directly, giving the connection a name that will be referenced in SQL statements. You can connect to several databases at the same time and to the same database multiple times.

# 3.2.4.1 Single Explicit Connection

In the following example, you connect to a single nondefault database at a remote node:

```
/* declare needed host variables */
char username[10] = "scott";
char password[10] = "tiger";
char db_string[20] = "NYNON";

/* give the database connection a unique name */
EXEC SQL DECLARE DB_NAME DATABASE;

/* connect to the nondefault database */
EXEC SQL CONNECT :username IDENTIFIED BY :password
    AT DB NAME USING :db string;
```

The identifiers in this example serve the following purposes:

- The host variables username and password identify a valid user.
- The host variable db\_string contains the Oracle Net syntax for connecting to a nondefault database at a remote node.
- The undeclared identifier DB\_NAME names a nondefault connection; it is an identifier used by Oracle, not a host or program variable.

The USING clause specifies the network, machine, and database associated with DB\_NAME. Later, SQL statements using the AT clause (with DB\_NAME) are executed at the database specified by *db\_string*.

Alternatively, you can use a character host variable in the AT clause, as the following example shows:

```
/* declare needed host variables */
char username[10] = "scott";
```



```
char password[10] = "tiger";
char db_name[10] = "oracle1";
char db_string[20] = "NYNON";

/* connect to the nondefault database using db_name */
EXEC SQL CONNECT :username IDENTIFIED BY :password
   AT :db_name USING :db_string;
```

If db\_name is a host variable, the DECLARE DATABASE statement is not needed. Only if DB\_NAME is an undeclared identifier must you execute a DECLARE DB\_NAME DATABASE statement before executing a CONNECT ... AT DB\_NAME statement.

## 3.2.4.1.1 SQL Operations

If granted the privilege, you can execute any SQL data manipulation statement at the nondefault connection. For example, you might execute the following sequence of statements:

```
EXEC SQL AT DB_NAME SELECT ...
EXEC SQL AT DB_NAME INSERT ...
EXEC SQL AT DB NAME UPDATE ...
```

In the next example, *db\_name* is a host variable:

```
EXEC SQL AT :db name DELETE ...
```

If *db\_name* is a host variable, all database tables referenced by the SQL statement must be defined in DECLARE TABLE statements. Otherwise, the precompiler issues a warning.

#### **Related Topics**

- About Using DECLARE TABLE
- DECLARE TABLE (Oracle Embedded SQL Directive)

### 3.2.4.1.2 PL/SQL Blocks

You can execute a PL/SQL block using the AT clause. The following example shows the syntax:

#### 3.2.4.1.3 Cursor Control

Cursor control statements such as OPEN, FETCH, and CLOSE are exceptions—they never use an AT clause. If you want to associate a cursor with an explicitly identified database, use the AT clause in the DECLARE CURSOR statement, as follows:

```
EXEC SQL AT :db_name DECLARE emp_cursor CURSOR FOR ...

EXEC SQL OPEN emp_cursor ...

EXEC SQL FETCH emp_cursor ...

EXEC SQL CLOSE emp_cursor;
```

If db\_name is a host variable, its declaration must be within the scope of all SQL statements that refer to the DECLAREd cursor. For example, if you OPEN the cursor in one subprogram, then FETCH from it in another subprogram, you must declare db\_name globally.

When OPENing, CLOSing, or FETCHing from the cursor, you do not use the AT clause. The SQL statements are executed at the database named in the AT clause of the DECLARE CURSOR statement or at the default database if no AT clause is used in the cursor declaration.

The AT :host\_variable clause provides the ability to change the connection associated with a cursor. However, you cannot change the association while the cursor is open. Consider the following example:

```
EXEC SQL AT :db_name DECLARE emp_cursor CURSOR FOR ... strcpy(db_name, "oracle1");

EXEC SQL OPEN emp_cursor;

EXEC SQL FETCH emp_cursor INTO ... strcpy(db_name, "oracle2");

EXEC SQL OPEN emp_cursor; /* illegal, cursor still open */

EXEC SQL FETCH emp_cursor INTO ...
```

This is illegal because *emp\_cursor* is still open when you try to execute the second OPEN statement. Separate cursors are not maintained for different connections; there is only one *emp\_cursor*, which must be closed before it can be reopened for another connection. To debug the last example, simply close the cursor before reopening it, as follows:

```
EXEC SQL CLOSE emp_cursor; -- close cursor first
strcpy(db_name, "oracle2");
EXEC SQL OPEN emp_cursor;
EXEC SQL FETCH emp cursor INTO ...
```

## 3.2.4.1.4 Dynamic SQL

Dynamic SQL statements are similar to cursor control statements in that some never use the AT clause.

For dynamic SQL Method 1, you must use the AT clause if you want to execute the statement at a nondefault connection. An example follows:

```
EXEC SQL AT :db_name EXECUTE IMMEDIATE :sql_stmt;
```

For Methods 2, 3, and 4, you use the AT clause only in the DECLARE STATEMENT statement if you want to execute the statement at a nondefault connection. All other dynamic SQL statements such as PREPARE, DESCRIBE, OPEN, FETCH, and CLOSE never use the AT clause. The next example shows Method 2:

```
EXEC SQL AT :db_name DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM :sql_string;
EXEC SQL EXECUTE sql stmt;
```

#### The following example shows Method 3:

```
EXEC SQL AT :db_name DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM :sql_string;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL OPEN emp_cursor ...
EXEC SQL FETCH emp_cursor INTO ...
EXEC SQL CLOSE emp cursor;
```



## 3.2.4.2 Multiple Explicit Connections

You can use the AT *db\_name* clause for multiple explicit connections, just as you can for a single explicit connection. In the following example, you connect to two nondefault databases concurrently:

```
/* declare needed host variables */
char username[10] = "scott";
char password[10] = "tiger";
char db_string1[20] = "NYNON1";
char db_string2[20] = "CHINON";
...
/* give each database connection a unique name */
EXEC SQL DECLARE DB_NAME1 DATABASE;
EXEC SQL DECLARE DB_NAME2 DATABASE;
/* connect to the two nondefault databases */
EXEC SQL CONNECT :username IDENTIFIED BY :password
    AT DB_NAME1 USING :db_string1;
EXEC SQL CONNECT :username IDENTIFIED BY :password
AT DB_NAME2 USING :db_string2;
```

The identifiers DB\_NAME1 and DB\_NAME2 are declared and then used to name the default databases at the two nondefault nodes so that later SQL statements can refer to the databases by name.

Alternatively, you can use a host variable in the AT clause, as the following example shows:

You can also use this method to make multiple connections to the same database, as the following example shows:

```
strcpy(db_string, "NYNON");
for (i = 0; i < ndefs; i++)
{
    /* connect to the nondefault database */
    printf("Database name: ");
    gets(db_name);
    EXEC SQL CONNECT :username IDENTIFIED BY :password
        AT :db_name USING :db_string;
}
...</pre>
```



You must use different database names for the connections, even though they use the same OracleNet string. However, you can connect twice to the same database using just one database name because that name identifies both the default and nondefault databases.

# 3.2.4.3 Ensuring Data Integrity

Your application program must ensure the integrity of transactions that manipulate data at two or more remote databases. That is, the program must commit or roll back *all* SQL statements in the transactions. This might be impossible if the network fails or one of the systems crashes.

For example, suppose you are working with two accounting databases. You debit an account on one database and credit an account on the other database, then issue a COMMIT at each database. It is up to your program to ensure that both transactions are committed or rolled back.

# 3.2.5 Implicit Connections

Implicit connections are supported through the Oracle distributed query facility, which does not require explicit connections, but only supports the SELECT statement. A distributed query allows a single SELECT statement to access data on one or more nondefault databases.

The distributed query facility depends on database links, which assign a name to a CONNECT statement rather than to the connection itself. At run time, the embedded SELECT statement is executed by the specified Oracle Server, which *implicitly* connects to the nondefault database(s) to get the required data.

## 3.2.5.1 Single Implicit Connections

In the next example, you connect to a single nondefault database. First, your program executes the following statement to define a database link (database links are usually established interactively by the DBA or user):

```
EXEC SQL CREATE DATABASE LINK db_link
   CONNECT TO username IDENTIFIED BY password
   USING 'NYNON';
```

Then, the program can query the nondefault EMP table using the database link, as follows:

```
EXEC SQL SELECT ENAME, JOB INTO :emp_name, :job_title
   FROM emp@db_link
   WHERE DEPTNO = :dept_number;
```

The database link is not related to the database name used in the AT clause of an embedded SQL statement. It simply tells Oracle where the nondefault database is located, the path to it, and what Oracle username and password to use. The database link is stored in the data dictionary until it is explicitly dropped.

In our example, the default Oracle Server logs on to the nondefault database through Oracle Net using the database link *db\_link*. The query is submitted to the default Server, but is "forwarded" to the nondefault database for execution.

To make referencing the database link easier, you can interactively create a synonym as follows:

```
EXEC SQL CREATE SYNONYM emp FOR emp@db_link;
```

Then, your program can guery the nondefault EMP table, as follows:

```
EXEC SQL SELECT ENAME, JOB INTO :emp_name, :job_title
  FROM emp
  WHERE DEPTNO = :dept_number;
```

This provides location transparency for *emp*.

## 3.2.5.2 Multiple Implicit Connections

In the following example, you connect to two nondefault databases concurrently. First, you execute the following sequence of statements to define two database links and create two synonyms:

Then, your program can query the nondefault EMP and DEPT tables, as follows:

```
EXEC SQL SELECT ENAME, JOB, SAL, LOC
FROM emp, dept
WHERE emp.DEPTNO = dept.DEPTNO AND DEPTNO = :dept number;
```

Oracle executes the query by performing a join between the nondefault EMP table at *db\_link1* and the nondefault DEPT table at *db\_link2*.

# 3.3 Definitions of Transactions Terms

Before delving into the subject of transactions, you should know the terms defined in this section.

The jobs or tasks that Oracle manages are called *sessions*. A *user session* is invoked when you run an application program or a tool such as SQL\*Forms, and connect to the database.

Oracle allows user sessions to work simultaneously and share computer resources. To do this, Oracle must control *concurrency*, the accessing of the same data by many users. Without adequate concurrency controls, there might be a loss of *data integrity*. That is, changes to data or structures might be made in the wrong order.

Oracle uses *locks* (sometimes called *enqueues*) to control concurrent access to data. A lock gives you temporary ownership of a database resource such as a table or row of data. Thus, data cannot be changed by other users until you finish with it.

You need never explicitly lock a resource, because default locking mechanisms protect Oracle data and structures. However, you can request *data locks* on tables or rows when it is to your advantage to override default locking. You can choose from several *modes* of locking such as *row share* and *exclusive*.

A *deadlock* can occur when two or more users try to access the same database object. For example, two users updating the same table might wait if each tries to update a row currently locked by the other. Because each user is waiting for resources held by another user, neither can continue until Oracle breaks the deadlock. Oracle signals an error to the participating transaction that had completed the least amount of work, and the "deadlock detected while waiting for resource" Oracle error code is returned to *sqlcode* in the SQLCA.

When a table is being queried by one user and updated by another at the same time, Oracle generates a *read-consistent* view of the table's data for the query. That is, once a query begins and as it proceeds, the data read by the query does not change. As update activity continues, Oracle takes *snapshots* of the table's data and records changes in a *rollback segment*. Oracle uses information in the rollback segment to build read-consistent query results and to undo changes if necessary.

# 3.4 How Data Integrity Is Ensured

Oracle is transaction oriented. That is, Oracle uses transactions to ensure data integrity. A transaction is a series of one or more logically related SQL statements you define to accomplish some task. Oracle treats the series of SQL statements as a unit so that all the changes brought about by the statements are either *committed* (made permanent) or *rolled back* (undone) at the same time. If your application program fails in the middle of a transaction, the database is automatically restored to its former (pre-transaction) state.

The coming sections show you how to define and control transactions. Specifically, you learn how to:

- Connect to the database.
- Make concurrent connections.
- Begin and end transactions.
- Use the COMMIT statement to make transactions permanent.
- Use the SAVEPOINT statement with the ROLLBACK TO statement to undo parts of transactions.
- Use the ROLLBACK statement to undo whole transactions.
- Specify the RELEASE option to free resources and log off the database.
- Use the SET TRANSACTION statement to set read-only transactions.
- Use the FOR UPDATE clause or LOCK TABLE statement to override default locking.

For details about the SQL statements discussed in this chapter, see *Oracle Database SQL Language Reference*.

# 3.5 How to Begin and End Transactions

You begin a transaction with the first executable SQL statement (other than CONNECT) in your program. When one transaction ends, the next executable SQL statement automatically begins another transaction. Thus, every executable statement is part of a transaction. Because they cannot be rolled back and need not be committed, declarative SQL statements are not considered part of a transaction.

You end a transaction in one of the following ways:

- Code a COMMIT or ROLLBACK statement, with or without the RELEASE option. This
   explicitly makes permanent or undoes changes to the database.
- Code a data definition statement (ALTER, CREATE, or GRANT, for example), which issues
  an automatic COMMIT before and after executing. This implicitly makes permanent
  changes to the database.

A transaction also ends when there is a system failure or your user session stops unexpectedly because of software problems, hardware problems, or a forced interrupt. Oracle rolls back the transaction.

If your program fails in the middle of a transaction, Oracle detects the error and rolls back the transaction. If your operating system fails, Oracle restores the database to its former (pretransaction) state.

# 3.6 Using the COMMIT Statement

If you do not subdivide your program with the COMMIT or ROLLBACK statement, Oracle treats the whole program as a single transaction (unless the program contains data definition statements, which issue automatic COMMITS).

You use the COMMIT statement to make changes to the database permanent. Until changes are COMMITted, other users cannot access the changed data; they see it as it was before your transaction began. Specifically, the COMMIT statement

- Makes permanent all changes made to the database during the current transaction
- Makes these changes visible to other users
- Erases all savepoints (see the next section)
- Releases all row and table locks, but not parse locks
- Closes cursors referenced in a CURRENT OF clause or, when MODE=ANSI, closes all
  explicit cursors for the connection specified in the COMMIT statement
- Ends the transaction

The COMMIT statement has no effect on the values of host variables or on the flow of control in your program.

When MODE=ORACLE, explicit cursors that are not referenced in a CURRENT OF clause remain open across COMMITs. This can boost performance.

Because they are part of normal processing, COMMIT statements should be placed inline, on the main path through your program. Before your program terminates, it must explicitly COMMIT pending changes. Otherwise, Oracle rolls them back. In the following example, you commit your transaction and disconnect from Oracle:

```
EXEC SQL COMMIT WORK RELEASE;
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all Oracle resources (locks and cursors) held by your program and logs off the database.

You need not follow a data definition statement with a COMMIT statement because data definition statements issue an automatic COMMIT before *and* after executing. So, whether they succeed or fail, the prior transaction is committed.

#### **Related Topics**

Fetch Across COMMITs

# 3.6.1 WITH HOLD Clause in DECLARE CURSOR Statements

Any cursor that has been declared with the clause WITH HOLD after the word CURSOR remains open after a COMMIT. The following example shows how to use this clause:

```
EXEC SQL

DECLARE C1 CURSOR WITH HOLD

FOR SELECT ENAME FROM EMP

WHERE EMPNO BETWEEN 7600 AND 7700

END-EXEC.
```



The cursor must not be declared for UPDATE. The WITH HOLD clause is used in DB2 to override the default, which is to close all cursors on commit. Pro\*COBOL provides this clause in order to ease migrations of applications from DB2 to Oracle. When MODE=ANSI, Oracle uses the DB2 default, but all host variables must be declared in a Declare Section. To avoid having a Declare Section, use the precompiler option CLOSE\_ON\_COMMIT described next.

#### **Related Topics**

DECLARE CURSOR (Embedded SQL Directive)

# 3.6.2 CLOSE\_ON\_COMMIT Precompiler Option

The precompiler option CLOSE\_ON\_COMMIT is available to override the default behavior of MODE=ANSI (if you specify MODE=ANSI on the command line, any cursors not declared with the WITH HOLD clause are closed on commit):

```
CLOSE ON COMMIT = {YES | NO}
```

The default is NO. This option must be entered only on the command line or in a configuration file.



Use this option carefully; applications may be slowed if cursors are opened and closed many times because of the need to re-parse for each OPEN statement.

#### **Related Topics**

CLOSE\_ON\_COMMIT

#### **Related Topics**

- Macro and Micro Options
- CLOSE\_ON\_COMMIT

# 3.7 Using the SAVEPOINT Statement

You use the SAVEPOINT statement to mark and name the current point in the processing of a transaction. Each marked point is called a *savepoint*. For example, the following statement marks a savepoint named *start\_delete*:

```
EXEC SQL SAVEPOINT start delete;
```

Savepoints let you divide long transactions, giving you more control over complex procedures. For example, if a transaction performs several functions, you can mark a savepoint before each function. Then, if a function fails, you can easily restore the Oracle data to its former state, recover, then reexecute the function.

To undo part of a transaction, you use savepoints with the ROLLBACK statement and its TO SAVEPOINT clause. In the following example, you access the table MAIL\_LIST to insert new listings, update old listings, and delete (a few) inactive listings. After the delete, you check the third element of *sqlerrd* in the SQLCA for the number of rows deleted. If the number is unexpectedly large, you roll back to the savepoint *start\_delete*, undoing just the delete.

```
for (;;)
```



```
printf("Customer number? ");
   gets(temp);
   cust_number = atoi(temp);
   printf("Customer name? ");
   gets(cust name);
  EXEC SQL INSERT INTO mail list (custno, cname, stat)
     VALUES (:cust number, :cust name, 'ACTIVE');
for (;;)
   printf("Customer number? ");
   gets(temp);
   cust number = atoi(temp);
   printf("New status? ");
   gets (new status);
   EXEC SQL UPDATE mail list
     SET stat = :new status
     WHERE custno = :cust_number;
/* mark savepoint */
EXEC SQL SAVEPOINT start_delete;
EXEC SQL DELETE FROM mail list
    WHERE stat = 'INACTIVE';
if (sqlca.sqlerrd[2] < 25) /* check number of rows deleted */
    printf("Number of rows deleted is %d\n", sqlca.sqlerrd[2]);
else
    printf("Undoing deletion of %d rows\n", sqlca.sqlerrd[2]);
    EXEC SQL WHENEVER SQLERROR GOTO sql error;
    EXEC SQL ROLLBACK TO SAVEPOINT start delete;
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL COMMIT WORK RELEASE;
exit(0);
sql error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf("Processing error\n");
exit(1);
```

Rolling back to a savepoint erases any savepoints marked after that savepoint. The savepoint to which you roll back, however, is not erased. For example, if you mark five savepoints, then roll back to the third, only the fourth and fifth are erased.

If you give two savepoints the same name, the earlier savepoint is erased. A COMMIT or ROLLBACK statement erases all savepoints.

#### **Related Topics**

· About Using the WHENEVER Directive

# 3.8 The ROLLBACK Statement

You use the ROLLBACK statement to undo pending changes made to the database. For example, if you make a mistake, such as deleting the wrong row from a table, you can use

ROLLBACK to restore the original data. The TO SAVEPOINT clause lets you roll back to an intermediate statement in the current transaction, so you do not have to undo all your changes.

If you start a transaction that does not complete (a SQL statement might not execute successfully, for example), ROLLBACK lets you return to the starting point, so that the database is not left in an inconsistent state. Specifically, the ROLLBACK statement

- Undoes all changes made to the database during the current transaction
- Erases all savepoints
- Ends the transaction
- Releases all row and table locks, but not parse locks
- Closes cursors referenced in a CURRENT OF clause or, when MODE=ANSI, closes all
  explicit cursors

The ROLLBACK statement has no effect on the values of host variables or on the flow of control in your program.

When MODE=ORACLE, explicit cursors not referenced in a CURRENT OF clause remain open across ROLLBACKs.

Specifically, the ROLLBACK TO SAVEPOINT statement

- Undoes changes made to the database since the specified savepoint was marked
- Erases all savepoints marked after the specified savepoint
- Releases all row and table locks acquired since the specified savepoint was marked



You cannot specify the RELEASE option in a ROLLBACK TO SAVEPOINT statement.

Because they are part of exception processing, ROLLBACK statements should be placed in error handling routines, off the main path through your program. In the following example, you roll back your transaction and disconnect from Oracle:

```
EXEC SQL ROLLBACK WORK RELEASE;
```

The optional keyword WORK provides ANSI compatibility. The RELEASE option frees all resources held by your program and disconnects from the database.

If a WHENEVER SQLERROR GOTO statement branches to an error handling routine that includes a ROLLBACK statement, your program might enter an infinite loop if the ROLLBACK fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the ROLLBACK statement, as shown in the following example:



```
...
sql_error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf("Processing error\n");
exit(1);
```

Oracle automatically rolls back transactions if your program terminates abnormally.

#### **Related Topics**

The RELEASE Option

## 3.8.1 Statement-Level Rollbacks

Before executing any SQL statement, Oracle marks an implicit savepoint (not available to you). Then, if the statement fails, Oracle automatically rolls it back and returns the applicable error code to *sqlcode* in the SQLCA. For example, if an INSERT statement causes an error by trying to insert a duplicate value in a unique index, the statement is rolled back.

Oracle can also roll back single SQL statements to break deadlocks. Oracle signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Only work started by the failed SQL statement is lost; work done before that statement in the current transaction is saved. Thus, if a data definition statement fails, the automatic commit that precedes it is not undone.

Before executing a SQL statement, Oracle must parse it, that is, examine it to make sure it follows syntax rules and refers to valid database objects. Errors detected while executing a SQL statement cause a rollback, but errors detected while parsing the statement do not.

# 3.9 The RELEASE Option

Oracle automatically rolls back changes if your program terminates abnormally. Abnormal termination occurs when your program does not explicitly commit or roll back work and disconnect from Oracle using the RELEASE option. Normal termination occurs when your program runs its course, closes open cursors, explicitly commits or rolls back work, disconnects from Oracle, and returns control to the user.

Your program will exit gracefully if the last SQL statement it executes is either

```
EXEC SQL COMMIT WORK RELEASE;

OF

EXEC SQL ROLLBACK WORK RELEASE;
```

where the token WORK is optional. Otherwise, locks and cursors acquired by your user session are held after program termination until Oracle recognizes that the user session is no longer active. This might cause other users in a multiuser environment to wait longer than necessary for the locked resources.



# 3.10 The SET TRANSACTION Statement

You use the SET TRANSACTION statement to begin a read-only transaction. Because they allow "repeatable reads," read-only transactions are useful for running multiple queries against one or more tables while other users update the same tables. An example of the SET TRANSACTION statement follows:

```
EXEC SQL SET TRANSACTION READ ONLY;
```

The SET TRANSACTION statement must be the first SQL statement in a read-only transaction and can appear only once in a transaction. The READ ONLY parameter is required. Its use does not affect other transactions.

Only the SELECT, COMMIT, and ROLLBACK statements are allowed in a read-only transaction. For example, including an INSERT, DELETE, or SELECT FOR UPDATE OF statement causes an error.

During a read-only transaction, all queries refer to the same snapshot of the database, providing a multitable, multiquery, read-consistent view. Other users can continue to query or update data as usual.

A COMMIT, ROLLBACK, or data definition statement ends a read-only transaction. (Recall that data definition statements issue an implicit COMMIT.)

In the following example, as a store manager, you check sales activity for the day, the past week, and the past month by using a read-only transaction to generate a summary report. The report is unaffected by other users updating the database during the transaction.

```
EXEC SQL SET TRANSACTION READ ONLY;
EXEC SQL SELECT sum(saleamt) INTO :daily FROM sales
   WHERE saledate = SYSDATE;

EXEC SQL SELECT sum(saleamt) INTO :weekly FROM sales
   WHERE saledate > SYSDATE - 7;

EXEC SQL SELECT sum(saleamt) INTO :monthly FROM sales
   WHERE saledate > SYSDATE - 30;

EXEC SQL COMMIT WORK;
   /* simply ends the transaction since there are no changes
   to make permanent */
/* format and print report */
```

# 3.11 Override Default Locking

By default, Oracle automatically locks many data structures for you. However, you can request specific data locks on rows or tables when it is to your advantage to override default locking. Explicit locking lets you share or deny access to a table for the duration of a transaction or ensure multitable and multiquery read consistency.

With the SELECT FOR UPDATE OF statement, you can explicitly lock specific rows of a table to make sure they do not change before an UPDATE or DELETE is executed. However, Oracle automatically obtains row-level locks at UPDATE or DELETE time. So, use the FOR UPDATE OF clause only if you want to lock the rows *before* the UPDATE or DELETE.

You can explicitly lock entire tables using the LOCK TABLE statement.

# 3.11.1 Using FOR UPDATE OF

When you DECLARE a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement, you use the FOR UPDATE OF clause to acquire exclusive row locks. SELECT FOR UPDATE OF identifies the rows that will be updated or deleted, then locks each row in the active set. This is useful, for example, when you want to base an update on the existing values in a row. You must make sure the row is not changed by another user before your update.

The FOR UPDATE OF clause is optional. For example, instead of coding

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
   SELECT ename, job, sal FROM emp WHERE deptno = 20
   FOR UPDATE OF sal;
```

you can drop the FOR UPDATE OF clause and simply code

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, job, sal FROM emp WHERE deptno = 20;
```

The CURRENT OF clause signals the precompiler to add a FOR UPDATE clause if necessary. You use the CURRENT OF clause to refer to the latest row FETCHed from a cursor.

#### **Related Topics**

The CURRENT OF Clause

### 3.11.1.1 Restrictions

If you use the FOR UPDATE OF clause, you cannot reference multiple tables.

An explicit FOR UPDATE OF or an implicit FOR UPDATE acquires exclusive row locks. All rows are locked at the OPEN, not as they are FETCHed. Row locks are released when you COMMIT or ROLLBACK (except when you ROLLBACK to a savepoint). Therefore, you cannot FETCH from a FOR UPDATE cursor after a COMMIT.

# 3.11.2 Using LOCK TABLE

You use the LOCK TABLE statement to lock one or more tables in a specified lock mode. For example, the statement in the following section, locks the EMP table in *row share* mode. Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use.

```
EXEC SQL LOCK TABLE EMP IN ROW SHARE MODE NOWAIT;
```

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an *exclusive* lock. While one user has an exclusive lock on a table, no other users can INSERT, UPDATE, or DELETE rows in that table.

The optional keyword NOWAIT tells Oracle not to wait for a table if it has been locked by another user. Control is immediately returned to your program, so it can do other work before trying again to acquire the lock. (You can check *sqlcode* in the SQLCA to see if the LOCK TABLE failed.) If you omit NOWAIT, Oracle waits until the table is available; the wait has no set limit.

A table lock never keeps other users from querying a table, and a query never acquires a table lock. So, a query never blocks another query or an update, and an update never blocks a

query. Only if two different transactions try to update the same row will one transaction wait for the other to complete.

Any LOCK TABLE statement implicitly closes all cursors.

Table locks are released when your transaction issues a COMMIT or ROLLBACK.

#### **Related Topics**

Lock Modes

# 3.12 Fetch Across COMMITs

If you want to intermix COMMITs and FETCHes, do not use the CURRENT OF clause. Instead, SELECT the ROWID of each row, then use that value to identify the current row during the update or delete. An example follows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
        SELECT ename, sal, ROWID FROM emp WHERE job = 'CLERK';
...

EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary, :row_id;
        ...
    EXEC SQL UPDATE emp SET sal = :new_salary
        WHERE ROWID = :row_id;
    EXEC SQL COMMIT;
...
}
```

Note, however, that the FETCHed rows are *not* locked. So, you might get inconsistent results if another user modifies a row after you read it but before you update or delete it.

# 3.13 Distributed Transactions Handling

A *distributed database* is a single logical database comprising multiple physical databases at different nodes. A *distributed statement* is any SQL statement that accesses a remote node using a database link. A *distributed transaction* includes at least one distributed statement that updates data at multiple nodes of a distributed database. If the update affects only one node, the transaction is non-distributed.

When you issue a COMMIT, changes to each database affected by the distributed transaction are made permanent. If instead you issue a ROLLBACK, all the changes are undone. However, if a network or machine fails during the commit or rollback, the state of the distributed transaction might be unknown or *in doubt*. In such cases, if you have FORCE TRANSACTION system privileges, you can manually commit or roll back the transaction at your local database by using the FORCE clause. The transaction must be identified by a quoted literal containing the transaction ID, which can be found in the data dictionary view DBA\_2PC\_PENDING. Some examples follow:

```
EXEC SQL COMMIT FORCE '22.31.83'; ...
EXEC SQL ROLLBACK FORCE '25.33.86';
```

FORCE commits or rolls back only the specified transaction and does not affect your current transaction. You cannot manually roll back in-doubt transactions to a savepoint.

The COMMENT clause in the COMMIT statement lets you specify a Comment to be associated with a distributed transaction. If ever the transaction is in doubt, Oracle stores the text specified by COMMENT in the data dictionary view DBA\_2PC\_PENDING along with the transaction ID. The text must be a quoted literal not exceeding 50 characters in length, which means that the maximum length for a COMMIT clause in a COMMIT statement used in embedded SQL is 50 characters. An example follows:

EXEC SQL COMMIT COMMENT 'In-doubt trans; notify Order Entry';



The COMMENT clause will be deprecated in a future release. Oracle recommends that you use transaction naming instead.

### **Related Topics**

Transactions

# 3.14 Guidelines

The following guidelines will help you avoid some common problems.

# 3.14.1 Designing Applications

When designing your application, group logically related actions together in one transaction. A well-designed transaction includes all the steps necessary to accomplish a given task—no more and no less.

Data in the tables you reference must be left in a consistent state. So, the SQL statements in a transaction should change the data in a consistent way. For example, a transfer of funds between two bank accounts should include a debit to one account and a credit to another. Both updates should either succeed or fail together. An unrelated update, such as a new deposit to one account, should not be included in the transaction.

# 3.14.2 Obtaining Locks

If your application programs include SQL locking statements, make sure the Oracle users requesting locks have the privileges needed to obtain the locks. Your DBA can lock any table. Other users can lock tables they own or tables for which they have a privilege, such as ALTER, SELECT, INSERT, UPDATE, or DELETE.

# 3.14.3 Using PL/SQL

If a PL/SQL block is part of a transaction, COMMITs and ROLLBACKs inside the block affect the whole transaction. In the following example, the ROLLBACK undoes changes made by the UPDATE *and* the INSERT:

```
EXEC SQL INSERT INTO EMP ...
EXEC SQL EXECUTE
BEGIN
UPDATE emp ...
EXCEPTION
```



WHEN DUP\_VAL\_ON\_INDEX THEN ROLLBACK;

• • •

END; END-EXEC;

...



# Datatypes and Host Variables

This chapter provides the basic information you need to write a Pro\*C/C++ program. This chapter contains the following topics:

- Oracle Datatypes
- Host Variables
- Indicator Variables
- VARCHAR Variables
- Cursor Variables
- CONTEXT Variables
- Universal ROWIDs
- Host Structures
- Pointer Variables
- Globalization Support
- NCHAR Variables
- · Handling Boolean Data

This chapter also includes several complete demonstration programs that you can study. These programs illustrate the techniques described. They are available on-line in your demo directory, so you can compile and run them, and modify them for your own uses.

# 4.1 Oracle Datatypes

Oracle recognizes two kinds of datatypes: *internal* and *external*. Internal datatypes specify how Oracle stores column values in database tables, as well as the formats used to represent pseudocolumn values such as NULL, SYSDATE, USER, and so on. External datatypes specify the formats used to store values in input and output host variables.

#### **Related Topics**

Oracle Built-in Data Types

# 4.1.1 Internal Data types

For values stored in database columns, Oracle uses the internal data types shown in Table 4-1

Table 4-1 Oracle Internal Data types

Name	Description
VARCHAR2	Variable-length character string, <= 4000 bytes.
NVARCHAR2 or NCHAR VARYING	Variable-length single-byte or National Character string,<= 4000 bytes.
NUMBER	Numeric value having precision and scale, represented in a base-100 format.

Table 4-1 (Cont.) Oracle Internal Data types

Name	Description
LONG	Variable-length character string <=2**31-1 bytes.
BINARY_FLOAT	32-bit floating point number, 4 bytes.
BINARY_DOUBLE	64-bit floating point number, 8 bytes.
TIMESTAMP	Year, month, and day values of date, as well as hour, minute, and second values of time, 7 or 11 bytes.
DATE	Fixed-length date + time value, 7 bytes.
INTERVAL YEAR	Stores a period of time in years and months, 5 bytes.
INTERVAL DAY	Stores a period of time in days, hours, minutes, and seconds, 11 bytes.
RAW	Variable-length binary data, <=2000 bytes.
LONG RAW	Variable-length binary data, <=2**31-1 bytes.
ROWID	Binary value.
UROWID	Binary value, <=4000 bytes.
CHAR	Fixed-length character string, <=2000 bytes.
NCHAR	Fixed-length single-byte or National Character string, <= 2000 bytes.
CLOB	Character data, <= 4 Gbytes.
NCLOB	National Character Set data, <= 4 Gbytes.
BLOB	Binary data, <= 4 Gbytes.
BFILE	External file binary data, <= 4 Gbytes.
BOOLEAN	The BOOLEAN data type comprises the distinct truth values: True and False.

These internal data types can be quite different from C data types. For example, C has no data type that is equivalent to the Oracle NUMBER data type. However, NUMBERs can be converted between C data types such as **float** and **double**, with some restrictions. For example, the Oracle NUMBER data type allows up to 38 decimal digits of precision, while no current C implementations can represent **double** with that degree of precision.

The Oracle NUMBER data type represents values exactly (within the precision limits), while floating-point formats cannot represent values such as 10.0 exactly.

Use the LOB data types to store unstructured data (text, graphic images, video clips, or sound waveforms). BFILE data is stored in an operating system file outside the database. LOB types store *locators* that specify the location of the data.

### Note:

All forms of LONG data types (LONG, LONG RAW, LONG VARCHAR, LONG VARRAW) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the LONG data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use CLOB and NCLOB data types for large amounts of character data.

For more information, see:

Migrating Columns from LONGs to LOBs



NCHAR and NVARCHAR2 are used to store multibyte character data.

You can use Boolean data type to store logical values: TRUE or False, in database columns.

#### **Related Topics**

• LOBs

# 4.1.2 External Datatypes

As shown in Table 4-2, the external datatypes include all the internal datatypes plus several datatypes that closely match C constructs. For example, the STRING external datatype refers to a C null-terminated string.

**Table 4-2 Oracle External Datatypes** 

Name	Description
VARCHAR2	Variable-length character string, <= 65535 bytes.
NUMBER	Decimal number, represented using a base-100 format.
INTEGER	Signed integer.
FLOAT	Real number.
STRING	Null-terminated variable length character string.
VARNUM	Decimal number, like NUMBER, but includes representation length component.
LONG	Fixed-length character string, up to 2**31-1 bytes.
VARCHAR	Variable-length character string, <= 65533 bytes.
ROWID	Binary value, external length is system dependent.
DATE	Fixed-length date/time value, 7 bytes.
VARRAW	Variable-length binary data, <= 65533 bytes.
RAW	Fixed-length binary data, <= 65535 bytes.
LONG RAW	Fixed-length binary data, <= 2**31-1 bytes.
UNSIGNED	Unsigned integer.
LONG VARCHAR	Variable-length character string, <= 2**31-5 bytes.
LONG VARRAW	Variable-length binary data, <= 2**31-5 bytes.
CHAR	Fixed-length character string, <= 65535 bytes.
CHARZ	Fixed-length, null-terminated character string, <= 65534 bytes.
CHARF	Used in TYPE or VAR statements to force CHAR to default to CHAR, instead of VARCHAR2 or CHARZ.
BOOLEAN	Boolean value (True, False, or Unknown)



### Note:

All forms of LONG data types (LONG, LONG RAW, LONG VARCHAR, LONG VARRAW) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the LONG data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use CLOB and NCLOB data types for large amounts of character data.

For more information, see:

Migrating Columns from LONGs to LOBs

Brief descriptions of the Oracle datatypes follow.

## 4.1.2.1 VARCHAR2

You use the VARCHAR2 datatype to store variable-length character strings. The maximum length of a VARCHAR2 value is 64K bytes.

You specify the maximum length of a VARCHAR2(n) value in bytes, not characters. So, if a VARCHAR2(n) variable stores multibyte characters, its maximum length can be less than n characters.

When you precompile using the option CHAR\_MAP=VARCHAR2, Oracle assigns the VARCHAR2 datatype to all host variables that you declare as **char[n]** or **char.** 

## 4.1.2.1.1 On Input

Oracle reads the number of bytes specified for the input host variable, strips any trailing blanks, then stores the input value in the target database column. Be careful. An uninitialized host variable can contain NULLs. So, always blank-pad a character input host variable to its declared length, and do not null-terminate it.

If the input value is longer than the defined width of the database column, Oracle generates an error. If the input value contains nothing but blanks, Oracle treats it like a NULL.

Oracle can convert a character value to a NUMBER column value if the character value represents a valid number. Otherwise, Oracle generates an error.

## 4.1.2.1.2 On Output

Oracle returns the number of bytes specified for the output host variable, blank-padding if necessary. It then assigns the output value to the target host variable. If a NULL is returned, Oracle fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, Oracle truncates the value before assigning it to the host variable. If there is an indicator variable associated with the host variable, Oracle sets it to the original length of the output value.

Oracle can convert NUMBER column values to character values. The length of the character host variable determines precision. If the host variable is too short for the number, scientific notation is used. For example, if you SELECT the column value 123456789 into a character host variable of length 6, Oracle returns the value '1.2E08'. If a NULL is selected explicitly, the value in the host variable is indeterminate. The value of the indicator variable needs to be checked for NULL-ness.



## 4.1.2.2 NUMBER

You use the NUMBER datatype to store fixed or floating-point numbers. You can specify precision and scale. The maximum precision of a NUMBER value is 38. The magnitude range is 1.0E-130 to 9.99...9E125 (38 nines followed by 88 zeroes). Scale can range from -84 to 127.

NUMBER values are stored in a variable-length format, starting with an exponent byte and followed by 19 mantissa bytes. The high-order bit of the exponent byte is a sign bit, which is set for positive numbers. The low-order 7 bits represent the magnitude.

The mantissa forms a 38-digit number with each byte representing 2 of the digits in a base-100 format. The sign of the mantissa is specified by the value of the first (left-most) byte. If greater than 101 then the mantissa is negative and the first digit of the mantissa is equal to the left-most byte minus 101.

On output, the host variable contains the number as represented internally by Oracle. To accommodate the largest possible number, the output host variable must be 22 bytes long. Only the bytes used to represent the number are returned. Oracle does not blank-pad or null-terminate the output value. If you need to know the length of the returned value, use the VARNUM datatype instead.

There is seldom a need to use this external datatype.

#### 4.1.2.3 INTEGER

You use the INTEGER datatype to store numbers that have no fractional part. An integer is a signed, 2-byte, 4-byte or 8-byte binary number. The order of the bytes in a word is system dependent. You must specify a length for input and output host variables. On output, if the column value is a real number, Oracle truncates any fractional part.

### 4.1.2.4 FLOAT

You use the FLOAT datatype to store numbers that have a fractional part or that exceed the capacity of the INTEGER datatype. The number is represented using the floating-point format of your computer and typically requires 4 or 8 bytes of storage. You must specify a length for input and output host variables.

Oracle can represent numbers with greater precision than most floating-point implementations because the internal format of Oracle numbers is decimal. This can cause a loss of precision when fetching into a FLOAT variable.

## 4.1.2.5 STRING

The STRING datatype is like the VARCHAR2 datatype, except that a STRING value is always null-terminated. When you precompile using the option CHAR\_MAP=STRING, Oracle assigns the STRING datatype to all host variables that you declare as char[n] or char.

### 4.1.2.5.1 On Input

Oracle uses the specified length to limit the scan for the null terminator. If a null terminator is not found, Oracle generates an error. If you do not specify a length, Oracle assumes the maximum length of 2000 bytes. The minimum length of a STRING value is 2 bytes. If the first character is a null terminator and the specified length is 2, Oracle inserts a null unless the column is defined as NOT NULL. If the column is defined as NOT NULL, an error occurs. An all-blank value is stored intact.



## 4.1.2.5.2 On Output

Oracle appends a null byte to the last character returned. If the string length exceeds the specified length, Oracle truncates the output value and appends a null byte. If a NULL is SELECTEd, Oracle returns a null byte in the first character position. If a NULL is selected explicitly, the value in the host variable is indeterminate. The value of the indicator variable needs to be checked for NULL-ness.

## 4.1.2.6 VARNUM

The VARNUM datatype is like the NUMBER datatype, except that the first byte of a VARNUM variable stores the length of the representation.

On input, you must set the first byte of the host variable to the length of the value. On output, the host variable contains the length followed by the number as represented internally by Oracle. To accommodate the largest possible number, the host variable must be 22 bytes long. After SELECTing a column value into a VARNUM host variable, you can check the first byte to get the length of the value.

Normally, there is little reason to use this datatype.

### 4.1.2.7 LONG

You use the LONG datatype to store fixed-length character strings.

The LONG datatype is like the VARCHAR2 datatype, except that the maximum length of a LONG value is 2147483647 bytes or two gigabytes.

## **4.1.2.8 VARCHAR**

You use the VARCHAR datatype to store variable-length character strings. VARCHAR variables have a 2-byte length field followed by a <=65533-byte string field. However, for VARCHAR array elements, the maximum length of the string field is 65530 bytes. When you specify the length of a VARCHAR variable, be sure to include 2 bytes for the length field. For longer strings, use the LONG VARCHAR datatype. If a NULL is selected explicitly, the value in the host variable is indeterminate. The value of the indicator variable needs to be checked for NULL-ness.

#### 4.1.2.9 ROWID

Rows in Index-Organized tables do not have permanent physical addresses. The logical ROWID is accessed using the same syntax as the physical ROWID. For this reason, the physical ROWID includes a *data object number* (schema objects in the same segment).

To support both logical and physical ROWIDs (as well as ROWIDs of non-Oracle tables) the *universal ROWID* was defined.

You can use character host variables to store rowids in a readable format. When you SELECT or FETCH a rowid into a character host variable, Oracle converts the binary value to an 18-byte character string and returns it in the format

BBBBBBBB.RRRR.FFFF

where BBBBBBB is the block in the database file, RRRR is the row in the block (the first row is 0), and FFFF is the database file. These numbers are hexadecimal. For example, the rowid



000000000.000A.0007 points to the 11th row in the 15th block in the 7th database file.

Typically, you FETCH a rowid into a character host variable, then compare the host variable to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement. That way, you can identify the latest row fetched by a cursor.



If you need full portability or your application communicates with a non-Oracle database using Oracle Open Gateway technology, specify a maximum length of 256 (not 18) bytes when declaring the host variable. Though you can assume nothing about the host variable's contents, the host variable will behave normally in SQL statements.

#### **Related Topics**

- Universal ROWIDs
- About Mimicking CURRENT OF

### 4.1.2.10 DATE

You use the DATE datatype to store dates and times in 7-byte, fixed-length fields. As Table 4-3 shows, the century, year, month, day, hour (in 24-hour format), minute, and second are stored in that order from left to right.

Table 4-3 DATE Format

Date Datatype	Century	Year	Month	Day	Hour	Minutes	Second
Byte	1	2	3	4	5	6	7
Meaning	Century	Year	Month	Day	Hour	Minute	Second
Example 17-OCT-1994 at 1:23:12 PM	119	194	10	17	14	24	13

The century and year bytes are in excess-100 notation. The hour, minute, and second are in excess-1 notation. Dates before the Common Era (B.C.E.) are less than 100. The epoch is January 1, 4712 B.C.E. For this date, the century byte is 53 and the year byte is 88. The hour byte ranges from 1 to 24. The minute and second bytes range from 1 to 60. The time defaults to midnight (1, 1, 1).

Normally, there is little reason to use the DATE datatype.

## 4.1.2.11 RAW

You use the RAW datatype to store binary data or byte strings. The maximum length of a RAW value is 65535 bytes.

RAW data is like CHARACTER data, except that Oracle assumes nothing about the meaning of RAW data and does no character set conversions when you transmit RAW data from one system to another.



## 4.1.2.12 VARRAW

You use the VARRAW datatype to store variable-length binary data or byte strings. The VARRAW datatype is like the RAW datatype, except that VARRAW variables have a 2-byte length field followed by a data field <= 65533 bytes in length. For longer strings, use the LONG VARRAW datatype.

When you specify the length of a VARRAW variable, be sure to include 2 bytes for the length field. The first two bytes of the variable must be interpretable as an integer.

To get the length of a VARRAW variable, simply refer to its length field.

#### 4.1.2.13 LONG RAW

You use the LONG RAW datatype to store binary data or byte strings. The maximum length of a LONG RAW value is 2147483647 bytes or two gigabytes.

LONG RAW data is like LONG data, except that Oracle assumes nothing about the meaning of LONG RAW data and does no character set conversions when you transmit LONG RAW data from one system to another.

### 4.1.2.14 UNSIGNED

You use the UNSIGNED datatype to store unsigned integers. An unsigned integer is a binary number of 2 or 4 bytes. The order of the bytes in a word is system dependent. You must specify a length for input and output host variables. On output, if the column value is a floating-point number, Oracle truncates the fractional part.

## 4.1.2.15 LONG VARCHAR

You use the LONG VARCHAR datatype to store variable-length character strings. LONG VARCHAR variables have a 4-byte length field followed by a string field. The maximum length of the string field is 2147483643 (2\*\*31 - 5) bytes. When you specify the length of a LONG VARCHAR for use in a VAR or TYPE statement, do not include the 4 length bytes.

#### **4.1.2.16 LONG VARRAW**

You use the LONG VARRAW datatype to store variable-length binary data or byte strings. LONG VARRAW variables have a 4-byte length field followed by a data field. The maximum length of the data field is 2147483643 bytes. When you specify the length of a LONG VARRAW for use in a VAR or TYPE statement, do not include the 4 length bytes.

#### 4.1.2.17 CHAR

You use the CHAR datatype to store fixed-length character strings. The maximum length of a CHAR value is 65535 bytes.

## 4.1.2.17.1 On Input

Oracle reads the number of bytes specified for the input host variable, does *not* strip trailing blanks, then stores the input value in the target database column.

If the input value is longer than the defined width of the database column, Oracle generates an error. If the input value is all-blank, Oracle treats it like a character value.



## 4.1.2.17.2 On Output

Oracle returns the number of bytes specified for the output host variable, doing blank-padding if necessary, then assigns the output value to the target host variable. If a NULL is returned, Oracle fills the host variable with blanks.

If the output value is longer than the declared length of the host variable, Oracle truncates the value before assigning it to the host variable. If an indicator variable is available, Oracle sets it to the original length of the output value. If a NULL is selected explicitly, the value in the host variable is indeterminate. The value of the indicator variable needs to be checked for NULLness.

### 4.1.2.18 CHARZ

When DBMS=V7 or V8, Oracle, by default, assigns the CHARZ datatype to all character host variables in a Pro\*C/C++ program. The CHARZ datatype indicates fixed-length, null-terminated character strings. The maximum length of a CHARZ value is 65534 bytes.

## 4.1.2.18.1 On Input

The CHARZ and STRING datatypes work the same way. You must null-terminate the input value. The null terminator serves only to delimit the string; it does not become part of the stored data.

## 4.1.2.18.2 On Output

CHARZ host variables are blank-padded if necessary, then null-terminated. The output value is always null-terminated, even if data must be truncated. If a NULL is selected explicitly, the value in the host variable is indeterminate. The value of the indicator variable needs to be checked for NULL-ness.

### 4.1.2.19 CHARF

The CHARF datatype is used in EXEC SQL TYPE and EXEC SQL VAR statements. When you precompile with the DBMS option set to V7 or V8, specifying the external datatype CHAR in a TYPE or VAR statement equivalences the C type or variable to the fixed-length, null-terminated datatype CHARZ.

However, you might not want either of these type equivalences, but rather an equivalence to the fixed-length external type CHAR. If you use the external type CHARF, the C type or variable is *always* equivalenced to the fixed-length ANSI datatype CHAR, regardless of the DBMS value. CHARF never allows the C type to be equivalenced to VARCHAR2 or CHARZ. Alternatively, when you set the option CHAR\_MAP=CHARF, all host variables declared as char[n] or char are equivalenced to a CHAR string. If a NULL is selected explicitly, the value in the host variable is indeterminate. The value of the indicator variable needs to be checked for NULL-ness.

## 4.1.2.20 BOOLEAN

Applications can insert TRUE or FALSE values in BOOLEAN type column while inserting data in the database. The values can be:

- "TRUE" or "true"
- "FALSE" or "false"



Any non-zero integer represents TRUE and the integer zero (0) represents FALSE.

Unless prohibited by a NOT NULL constraint, the boolean data type also supports the truth value UNKNOWN as the null value.

# 4.1.3 Additional External Datatypes

This section describes additional external datatypes.

## 4.1.3.1 Datetime and Interval Datatypes

The datetime and interval datatypes are briefly summarized here.

#### **Related Topics**

Oracle Database SQL Language Reference

## 4.1.3.2 ANSI DATE

The ANSI DATE is based on the DATE, but contains no time portion. (Therefore, it also has no time zone.) ANSI DATE follows the ANSI specification for the DATE datatype. When assigning an ANSI DATE to a DATE or a timestamp datatype, the time portion of the Oracle DATE and the timestamp are set to zero. When assigning a DATE or a timestamp to an ANSI DATE, the time portion is ignored.

You are encouraged to instead use the TIMESTAMP datatype which contains both date and time.

## 4.1.3.3 TIMESTAMP

The TIMESTAMP datatype is an extension of the DATE datatype. It stores the year, month, and day of the DATE datatype, plus the hour, minute, and second values. It has no time zone. The TIMESTAMP datatype has the form:

TIMESTAMP(fractional seconds precision)

where <code>fractional\_seconds\_precision</code> (which is optional) specifies the number of digits in the fractional part of the <code>SECOND</code> datetime field and can be a number in the range 0 to 9. The default is 6.

### 4.1.3.4 TIMESTAMP WITH TIME ZONE

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

TIMESTAMP(fractional seconds precision) WITH TIME ZONE

where <code>fractional\_seconds\_precision</code> optionally specifies the number of digits in the fractional part of the <code>SECOND</code> datetime field and can be a number in the range 0 to 9. The default is 6.

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



## 4.1.3.5 TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ) is another variant of TIMESTAMP that includes a time zone displacement in its value. Storage is in the same format as for TIMESTAMP. This type differs from TIMESTAMP WITH TIME ZONE in that data stored in the database is normalized to the database time zone, and the time zone displacement is not stored as part of the column data. When users retrieve the data, Oracle returns it in the users' local session time zone.

The time zone displacement is the difference (in hours and minutes) between local time and UTC (Coordinated Universal Time—formerly Greenwich Mean Time). The TIMESTAMP WITH LOCAL TIME ZONE datatype has the form:

TIMESTAMP(fractional seconds precision) WITH LOCAL TIME ZONE

where <code>fractional\_seconds\_precision</code> optionally specifies the number of digits in the fractional part of the <code>SECOND</code> datetime field and can be a number in the range 0 to 9. The default is 6.

### 4.1.3.6 INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH stores a period of time using the YEAR and MONTH datetime fields. The INTERVAL YEAR TO MONTH datatype has the form:

INTERVAL YEAR (year precision) TO MONTH

where the optional <code>year\_precision</code> is the number of digits in the <code>YEAR</code> datetime field. The default value of <code>year\_precision</code> is 2.

## 4.1.3.7 INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND stores a period of time in terms of days, hours, minutes, and seconds. The INTERVAL DAY TO SECOND datatype has the form:

INTERVAL DAY (day precision) TO SECOND(fractional seconds precision)

#### where:

• day\_precision is the number of digits in the DAY datetime field. It is optional. Accepted values are 0 to 9. The default is 2.

fractional\_seconds\_precision is the number of digits in the fractional part of the SECOND datetime field. It is optional. Accepted values are 0 to 9. The default is 6.

# 4.1.3.8 Avoiding Unexpected Results Using Datetime



To avoid unexpected results in your DML operations on datetime data, you can verify the database and session time zones by querying the built-in SQL functions DBTIMEZONE and SESSIONTIMEZONE. If the time zones have not been set manually, Oracle uses the operating system time zone by default. If the operating system time zone is not a valid Oracle time zone, Oracle uses UTC as the default value.



# 4.1.4 Support for Vector Data Type

As of Oracle Database 23ai, 23.5 Release, Pro\*C does not support the native vector data type directly. However, you can use the vector data type in your applications to insert data into and fetch data from a vector column in the following two ways.

- String type (SQLT CHR / SQLT STR / SQLT VCS / SQLT AVC / SQLT AFC)
- CLOB type (SQLT CLOB)

#### **Using the String Type**

The application is responsible for providing vector data in the string format.

#### Example:

```
Varchar2 c2arr[100];
strcpy (c2, "[4.4, 5.6, 7.0]");
c2.len=strlen((char *)c2.arr);
exec sql insert into table_name values (:c2); // This statement will insert
one row with 3 vector elements)
```

Data is fetched as a string, as in "[first element, Second element, Third Element, ...]."

#### Example:

```
Exec sql select coll into :c2arr from table name;
```

#### **Using The CLOB Type**

Applications can use LOB APIs to access a vector column.

The following are the steps to insert data into a vector column.

- Allocate a LOB descriptor.
- 2. Create a temporary LOB using the LOB descriptor.
- 3. Write vector data in the proper format using the temporary LOB descriptor.
- 4. Insert the temporary LOB in the vector column.
- 5. Free the temporary LOB.

#### **Fetching Vector Data**

Vector data can be fetched with a LOB descriptor. Existing LOB DESCRIBE and LOB READ calls can be used to get the vector data length and to read vector data.

#### Example:

- Allocate a LOB descriptor.
- 2. EXEC SQL SELECT column name INTO :c2lob
- 3. EXEC SQL LOB DESCRIBE :c2lob GET LENGTH INTO :length variable;



- 4. EXEC SQL LOB READ :amount FROM :c2lob INTO :string\_variable WITH LENGTH :length variable;
- 5. Free the LOB descriptor.

# 4.1.5 Support for JSON Data Type

In this release, Pro\*C does not support the native JSON data type directly. However, you can use the JSON data type in your applications to insert data into and fetch data from a JSON column in the following two ways.

- String type (SQLT\_CHR / SQLT\_STR / SQLT\_VCS / SQLT\_AVC / SQLT\_AFC)
- CLOB type (SQLT CLOB)

#### **Using the String Type**

The application is responsible for providing valid JSON data in the string format.

#### Example:

```
char c2arr[50];
strcpy(c2arr, "{\"id\":1, \"name\":\"Adam\", \"age\":29}");
EXEC SQL INSERT INTO table name values(:c2arr);
```

Data is fetched as a string, as in JSON string format "{"key1":value1, ...]"

#### Example:

```
EXEC SQL SELECT coll INTO : c2arr FROM table name;
```

#### Using the CLOB Type

Applications can use existing LOB APIs to access JSON columns.

Use the following steps to insert data into a JSON column.

- Allocate a LOB descriptor.
- 2. Create a temporary LOB using the LOB descriptor.
- 3. Write JSON data in a proper format using the temporary LOB descriptor.
- 4. Insert the temporary LOB in the JSON column.
- Free the temporary LOB.



LOB Statements statements for examples for the preceding steps.

### **Fetching JSON Data**

JSON data can be fetched using a LOB descriptor. Existing LOB DESCRIBE and LOB READ calls can be used to get the JSON data length and to read JSON data.



Use the following steps to fetch data from a JSON column:

- Allocate a LOB descriptor.
- 2. Select JSON column into the LOB descriptor.
- 3. Get the length using LOB DESCRIBE.
- 4. Read the JSON data using LOB READ.
- Free the LOB descriptor.



LOB Statements statements for examples for the preceding steps.

# 4.2 Host Variables

Host variables are the key to communication between your host program and Oracle. Typically, a precompiler program inputs data from a host variable to Oracle, and Oracle outputs data to a host variable in the program. Oracle stores input data in database columns, and stores output data in program host variables.

A host variable can be any arbitrary C expression that resolves to a scalar type. But, a host variable must also be an *Ivalue*. Host arrays of most host variables are also supported.

## 4.2.1 Host Variable Declaration

You declare a host variable according to the rules of the C programming language, specifying a C datatype supported by the Oracle program interface. The C datatype must be compatible with that of the source or target database column.

If MODE=ORACLE, you do not have to declare host variables in a special Declare Section. However, if you do not use a Declare Section, the FIPS flagger warns you about this, as the Declare Section is part of the ANSI SQL Standard. If CODE=CPP (you are compiling C++ code) or PARSE=NONE or PARSE=PARTIAL, then you must have a Declare Section.

Table 4-4 shows the C datatypes and the pseudotypes that you can use when declaring host variables. Only these datatypes can be used for host variables.

Table 4-4 C Datatypes for Host Variables

C Datatype or Pseudotype	Description
char	single character
char[n]	n-character array (string)
int	integer
short	small integer
long	large integer
long long	very large (8-byte) integer
float	floating-point number (usually single precision)
double	floating-point number (always double precision)
VARCHAR[n]	variable-length string



Table 4-4 (Cont.) C Datatypes for Host Variables

C Datatype or Pseudotype	Description
bool	Boolean data

Table 4-5 shows the compatible Oracle internal datatypes.

Table 4-5 C to Oracle Datatype Compatibility

Internal Type	С Туре	Description
VARCHAR2(Y)	char	single character
(Note 1)		
CHAR(X)	char[n]	n-byte character array
(Note 1)	VARCHAR[n]	n-byte variable-length character array
	int	integer
	short	small integer
	long	large integer
	long long	very large (8-byte) integer
	float	floating-point number
	double	double-precision floating-point
		number
NUMBER	int	integer
NUMBER(P,S)	short	small integer
(Note 2)	int	integer
	long	large integer
	float	floating-point number
	double	double-precision floating-point
	char	number
	char[n]	single character
	VARCHAR[n]	n-byte character array
		n-byte variable-length character array
DATE	char[n]	n-byte character array
	VARCHAR[n]	n-byte variable-length character array
LONG	char[n]	n-byte character array
	VARCHAR[n]	n-byte variable-length character array
RAW(X)	unsigned char[n]	n-byte character array
(Note 1)	VARCHAR[n]	n-byte variable-length character array
LONG RAW	unsigned char[n]	n-byte character array
	VARCHAR[n]	n-byte variable-length character array
ROWID	unsigned char[n]	n-byte character array
	VARCHAR[n]	n-byte variable-length character array
		· •



Table 4-5 (Cont.) C to Oracle Datatype Compatibility

Internal Type	С Туре	Description
BOOLEAN	bool	boolean data
	short	small integer
	int	integer
	long	large integer
	float	floating-point number
	double	double-precision floating-point
	char	number
	char[n]	single character
Notes:		
1. X ranges from 1 to 2000. 1 is the default value. Y ranges from 1 to 4000.		
2. P ranges from 1 to 38. S ranges from -84 to 127.		

One-dimensional arrays of simple C types can also serve as host variables. For char[n] and VARCHAR[n], n specifies the maximum string length, not the number of strings in the array. Two-dimensional arrays are allowed only for char[m][n] and VARCHAR[m][n], where m specifies the number of strings in the array and n specifies the maximum string length.

Pointers to simple C types are supported. Pointers to char[n] and VARCHAR[n] variables should be declared as pointer to char or VARCHAR (with no length specification). Arrays of pointers, however, are not supported.

## 4.2.1.1 Storage-Class Specifiers

Pro\*C/C++ lets you use the **auto**, **extern**, and **static** storage-class specifiers when you declare host variables. However, you cannot use the **register** storage-class specifier to store host variables, since the precompiler takes the address of host variables by placing an ampersand (&) before them. Following the rules of C, you can use the **auto** storage class specifier only within a block.

To comply with the ANSI C standard, the Pro\*C/C++ Precompiler provides the ability to declare an **extern char[n]** host variable with or without a maximum length, as the following examples shows:

```
extern char protocol[15];
extern char msg[];
```

However, you should always specify the maximum length. In the last example, if *msg* is an output host variable declared in one precompilation unit but defined in another, the precompiler has no way of knowing its maximum length. If you have not allocated enough storage for *msg* in the second precompilation unit, you might corrupt memory. (Usually, "enough" is the number of bytes in the longest column value that might be SELECTed or FETCHed into the host variable, plus one byte for a possible null terminator.)

If you neglect to specify the maximum length for an **extern char[]** host variable, the precompiler issues a warning message. The precompiler also assumes that the host variable will store a CHARACTER column value, which cannot exceed 255 characters in length. So, if

you want to SELECT or FETCH a VARCHAR2 or a LONG column value of length greater than 255 characters into the host variable, you *must* specify a maximum length.

# 4.2.1.2 Type Qualifiers

You can also use the **const** and **volatile** type qualifiers when you declare host variables.

A **const** host variable must have a constant value, that is, your program cannot change its initial value. A **volatile** host variable can have its value changed in ways unknown to your program (for example, by a device attached to the system).

# 4.2.2 Host Variable Referencing

You use host variables in SQL data manipulation statements. A host variable must be prefixed with a colon (:) in SQL statements but must not be prefixed with a colon in C statements, as the following example shows:

```
char buf[15];
int emp_number;
float salary;
...
gets(buf);
emp_number = atoi(buf);

EXEC SQL SELECT sal INTO :salary FROM emp
    WHERE empno = :emp_number;
```

Though it might be confusing, you can give a host variable the same name as an Oracle table or column, as this example shows:

```
int empno;
char ename[10];
float sal;
...
EXEC SQL SELECT ename, sal INTO :ename, :sal FROM emp
    WHERE empno = :empno;
```

### 4.2.2.1 Restrictions

A host variable name is a C identifier, hence it must be declared and referenced in the same upper/lower case format. It cannot substitute for a column, table, or other Oracle object in a SQL statement, and must not be an Oracle reserved word.

A host variable must resolve to an address in the program. For this reason, function calls and numeric expressions cannot serve as host variables. The following code is *invalid*:

#### **Related Topics**

Reserved Words, Keywords, and Namespaces

# 4.3 Indicator Variables

You can associate every host variable with an optional indicator variable. An indicator variable must be defined as a 2-byte integer and, in SQL statements, must be prefixed with a colon and immediately follow its host variable (unless you use the keyword INDICATOR). If you are using Declare Sections, you must also declare indicator variables inside the Declare Sections.

This applies to relational columns, not object types.

#### **Related Topics**

Objects

# 4.3.1 The INDICATOR Keyword

To improve readability, you can precede any indicator variable with the optional keyword INDICATOR. You must still prefix the indicator variable with a colon. The correct syntax is:

```
:host_variable INDICATOR :indicator_variable
```

#### which is equivalent to

```
:host variable:indicator variable
```

You can use both forms of expression in your host program.

Possible indicator values, and their meanings, are:

Indicator Values	Meanings
0	The operation was successful
-1	A NULL was returned, inserted, or updated.
-2	Output to a character host variable from a "long" type was truncated, but the original column length cannot be determined.
>0	The result of a SELECT or FETCH into a character host variable was truncated. In this case, if the host variable is a multibyte character variable, the indicator value is the original column length in characters. If the host variable is not a multibye character variable, then the indicator length is the original column length in bytes.

# 4.3.2 Example of INDICATOR Variable Usage

Typically, you use indicator variables to assign NULLs to input host variables and detect NULLs or truncated values in output host variables. In the example later, you declare three host variables and one indicator variable, then use a SELECT statement to search the database for an employee number matching the value of host variable *emp\_number*. When a matching row is found, Oracle sets output host variables *salary* and *commission* to the values of columns SAL and COMM in that row and stores a return code in indicator variable *ind\_comm*. The next statements use *ind\_comm* to select a course of action.

```
EXEC SQL BEGIN DECLARE SECTION;
  int    emp_number;
  float salary, commission;
  short comm_ind; /* indicator variable */
EXEC SQL END DECLARE SECTION;
  char temp[16];
```



#### **Related Topics**

Indicator Variables

## 4.3.3 INDICATOR Variable Guidelines

The following guidelines apply to declaring and referencing indicator variables. An indicator variable must

- Be declared explicitly (in the Declare Section if present) as a 2-byte integer.
- Be prefixed with a colon (:) in SQL statements.
- Immediately follow its host variable in SQL statements and PL/SQL blocks (unless preceded by the keyword INDICATOR).

An indicator variable must *not*:

- Be prefixed with a colon in host language statements.
- Follow its host variable in host language statements.
- Be an Oracle reserved word.

# 4.3.4 Oracle Restrictions

When DBMS=V7 or V8, if you SELECT or FETCH a NULL into a host variable that has no indicator, Oracle issues the following error message:

```
ORA-01405: fetched column value is NULL
```

When precompiling with MODE=ORACLE and DBMS=V7 or V8 specified, you can specify UNSAFE NULL=YES to disable the ORA-01405 message.

#### **Related Topics**

UNSAFE\_NULL

# 4.4 VARCHAR Variables

You can use the VARCHAR pseudotype to declare variable-length character strings. When your program deals with strings that are output from, or input to, VARCHAR2 or LONG columns, you might find it more convenient to use VARCHAR host variables instead of standard C strings. The datatype name VARCHAR can be uppercase or lowercase, but it cannot be mixed case. In this Guide, uppercase is used to emphasize that VARCHAR is not a native C datatype.

# 4.4.1 VARCHAR Variable Declaration

Think of a VARCHAR as an extended C type or pre-declared **struct**. For example, the precompiler expands the VARCHAR declaration

```
VARCHAR username[20];
```

into the following **struct** with array and length members:

```
struct
{
    unsigned short len;
    unsigned char arr[20];
} username;
```

The advantage of using VARCHAR variables is that you can explicitly reference the length member of the VARCHAR structure after a SELECT or FETCH. Oracle puts the length of the selected character string in the length member. You can then use this member to do things such as adding the null ('\0') terminator.

```
username.arr[username.len] = '\0';
```

or using the length in a *strncpy* or *printf* statement; for example:

```
printf("Username is %.*s\n", username.len, username.arr);
```

You specify the maximum length of a VARCHAR variable in its declaration. The length must lie in the range 1.65533. For example, the following declaration is *invalid* because no length is specified:

```
VARCHAR null_string[]; /* invalid */
```

The length specification holds the current length of the value stored in the array member.

You can declare multiple VARCHARs on a single line; for example:

```
VARCHAR emp_name[ENAME_LEN], dept_loc[DEPT_NAME_LEN];
```

The length specifier for a VARCHAR can be a **#define**d macro, or any complex expression that can be resolved to an integer at precompile time.

You can also declare pointers to VARCHAR datatypes.



Do not attempt to use a typedef statement such as:

```
typedef VARCHAR buf[64];
```

This causes errors during C compilation.

### **Related Topics**

VARCHAR Variables and Pointers



# 4.4.2 VARCHAR Variable Referencing

In SQL statements, you reference VARCHAR variables using the **struct** name prefixed with a colon, as the following example shows:

After the query is executed, *part\_desc.len* holds the actual length of the character string retrieved from the database and stored in *part\_desc.arr*.

In C statements, you reference VARCHAR variables using the component names, as the next example shows:

```
printf("\n\nEnter part description: ");
gets(part_desc.arr);
/* You must set the length of the string
  before using the VARCHAR in an INSERT or UPDATE */
part desc.len = strlen(part desc.arr);
```

### 4.4.3 Return NULLs to a VARCHAR Variable

Oracle automatically sets the length component of a VARCHAR output host variable. If you SELECT or FETCH a NULL into a VARCHAR, the server does not change the length or array members.



If you select a NULL into a VARCHAR host variable, and there is no associated indicator variable, an ORA-01405 error occurs at run time. Avoid this by coding indicator variables with all host variables. (As a temporary fix, use the UNSAFE\_NULL=YES precompiler option.

#### **Related Topics**

• DBMS

# 4.4.4 Insert NULLs Using VARCHAR Variables

If you set the length of a VARCHAR variable to zero before performing an UPDATE or INSERT statement, the column value is set to NULL. If the column has a NOT NULL constraint, Oracle returns an error.

### 4.4.5 Pass VARCHAR Variables to a Function

VARCHARS are structures, and most C compilers permit passing of structures to a function by value, and returning structures by copy out from functions. However, in Pro\*C/C++ you must pass VARCHARS to functions by reference. The following example shows the correct way to pass a VARCHAR variable to a function:

```
VARCHAR emp_name[20];
...
emp_name.len = 20;
SELECT ename INTO :emp_name FROM emp
WHERE empno = 7499;
...
print_employee_name(&emp_name); /* pass by pointer */
...
print_employee_name(name)
VARCHAR *name;
{
    ...
    printf("name is %.*s\n", name->len, name->arr);
    ...
}
```

# 4.4.6 Find the Length of the VARCHAR Array Component

When the precompiler processes a VARCHAR declaration, the actual length of the array element in the generated structure can be longer than that declared. For example, on a Sun Solaris system, the Pro\*C/C++ declaration

```
VARCHAR my_varchar[12];
```

is expanded by the precompiler to

```
struct my_varchar
{
    unsigned short len;
    unsigned char arr[12];
};
```

However, the precompiler or the C compiler on this system pads the length of the array component to 14 bytes. This alignment requirement pads the total length of the structure to 16 bytes: 14 for the padded array and 2 bytes for the length.

The SQLVarcharGetLength() (replaces the non-threaded sqlvcp()) function—part of the SQLLIB runtime library—returns the actual (possibly padded) length of the array member.

You pass the <code>SQLVarcharGetLength()</code> function the length of the data for a VARCHAR host variable or a VARCHAR pointer host variable, and <code>SQLVarcharGetLength()</code> returns the total length of the array component of the VARCHAR. The total length includes any padding that might be added by your C compiler.

```
The syntax of SQLVarcharGetLength() is
```

```
SQLVarcharGetLength (dvoid *context, unsigned long *datlen, unsigned long *totlen);
```

For single-threaded applications, use sqlvcp(). Put the length of the VARCHAR in the datlen parameter before calling sqlvcp(). When the function returns, the totlen parameter contains

the total length of the array element. Both parameters are pointers to unsigned long integers, so must be passed by reference.

#### **Related Topics**

New Names for SQLLIB Public Functions

# 4.4.7 Example Program: Using sqlvcp()

The following example program shows how you can use the function in a Pro\*C/C++ application. The example also uses the sqlgls() function. The example declares a VARCHAR pointer, then uses the sqlvcp() function to determine the size required for the VARCHAR buffer. The program FETCHes employee names from the EMP table and prints them. Finally, the example uses the sqlgls() function to print out the SQL statement and its function code and length attributes. This program is available on-line as sqlvcp.pc in your demo directory.

```
^{\star} The sqlvcp.pc program demonstrates how you can use the
   sqlvcp() function to determine the actual size of a
   VARCHAR struct. The size is then used as an offset to
   increment a pointer that steps through an array of
 * VARCHARs.
 ^{\star} This program also demonstrates the use of the sqlgls()
 * function, to get the text of the last SQL statement executed.
 ^{\star} sqlgls() is described in the "Error Handling" chapter of
 ^{\star} The Developer's Guide to the Oracle Pro^{\star}C/C++ Precompiler.
#include <stdio.h>
#include <sqlca.h>
#include <sqlcpr.h>
/* Fake a VARCHAR pointer type. */
struct my vc ptr
    unsigned short len;
    unsigned char arr[32767];
/* Define a type for the VARCHAR pointer */
typedef struct my_vc_ptr my_vc_ptr;
my_vc_ptr *vc_ptr;
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR *names;
int limit; /* for use in FETCH FOR clause */
char *username = "scott/tiger";
EXEC SQL END DECLARE SECTION;
void sql error();
extern void sqlvcp(), sqlgls();
main()
    unsigned int vcplen, function code, padlen, buflen;
    int i;
    char stmt buf[120];
    EXEC SQL WHENEVER SQLERROR DO sql error();
```



```
EXEC SQL CONNECT :username;
   printf("\nConnected.\n");
/* Find number of rows in table. */
   EXEC SQL SELECT COUNT(*) INTO :limit FROM emp;
/* Declare a cursor for the FETCH statement. */
   EXEC SQL DECLARE emp name cursor CURSOR FOR
    SELECT ename FROM emp;
   EXEC SQL FOR : limit OPEN emp name cursor;
/* Set the desired DATA length for the VARCHAR. */
   vcplen = 10;
/* Use SQLVCP to help find the length to malloc. */
   sqlvcp(&vcplen, &padlen);
   printf("Actual array length of VARCHAR is %ld\n", padlen);
/* Allocate the names buffer for names.
   Set the limit variable for the FOR clause. */
   names = (VARCHAR *) malloc((sizeof (short) +
   (int) padlen) * limit);
   if (names == 0)
       printf("Memory allocation error.\n");
       exit(1);
/* Set the maximum lengths before the FETCH.
   Note the "trick" to get an effective VARCHAR *.
    for (vc_ptr = (my_vc_ptr *) names, i = 0; i < limit; i++)</pre>
       vc_ptr->len = (short) padlen;
       vc_ptr = (my_vc_ptr *)((char *) vc_ptr +
       padlen + sizeof (short));
/* Execute the FETCH. */
   EXEC SQL FOR : limit FETCH emp name cursor INTO : names;
/* Print the results. */
   printf("Employee names--\n");
    for (vc_ptr = (my_vc_ptr *) names, i = 0; i < limit; i++)</pre>
       printf
        ("%.*s\t(%d)\n", vc_ptr->len, vc_ptr->arr, vc_ptr->len);
       vc ptr = (my vc ptr *) ((char *) vc ptr +
                  padlen + sizeof (short));
/* Get statistics about the most recent
* SQL statement using SQLGLS. Note that
   the most recent statement in this example
   is not a FETCH, but rather "SELECT ENAME FROM EMP"
   (the cursor).
   buflen = (long) sizeof (stmt buf);
/* The returned value should be 1, indicating no error. */
    sqlgls(stmt buf, &buflen, &function code);
```

```
if (buflen != 0)
        /* Print out the SQL statement. */
       printf("The SQL statement was--\n%.*s\n", buflen, stmt buf);
        /* Print the returned length. */
       printf("The statement length is %ld\n", buflen);
        /* Print the attributes. */
       printf("The function code is %ld\n", function_code);
       EXEC SQL COMMIT RELEASE;
       exit(0);
    }
   else
    {
       printf("The SQLGLS function returned an error.\n");
       EXEC SQL ROLLBACK RELEASE;
       exit(1);
}
void
sql error()
   char err msg[512];
   int buf_len, msg_len;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    buf len = sizeof (err msg);
    sqlglm(err_msg, &buf_len, &msg_len);
    printf("%.*s\n", msg_len, err_msg);
   EXEC SQL ROLLBACK RELEASE;
   exit(1);
}
```

#### **Related Topics**

Handling Runtime Errors

# 4.5 Cursor Variables

You can use *cursor variables* in your Pro\*C/C++ program for queries. A cursor variable is a handle for a cursor that must be defined and opened on the Oracle server using PL/SQL. See Cursor Variables for complete information about cursor variables.

The advantages of cursor variables are:

Ease of maintenance

Queries are centralized, in the stored procedure that opens the cursor variable. If you need to change the cursor, you only need to make the change in one place: the stored procedure. There is no need to change each application.

Convenient security

The user of the application is the username used when the Pro\*C/C++ application connects to the server. The user must have *execute* permission on the stored procedure that opens the cursor but not *read* permission on the tables used in the query. This

capability can be used to limit access to the columns in the table, and access to other stored procedures.

### 4.5.1 Declare a Cursor Variable

You declare a cursor variable in your Pro\*C/C++ program using the Pro\*C/C++ pseudotype SQL\_CURSOR. For example:

You can declare a cursor variable using the type specification SQL\_CURSOR, in all upper case, or sql\_cursor, in all lower case; you cannot use mixed case.

A cursor variable is just like any other host variable in the Pro\*C/C++ program. It has scope, following the scope rules of C. You can pass it as a parameter to other functions, even functions external to the source file in which you declared it. You can also define functions that return cursor variables, or pointers to cursor variables.



A SQL\_CURSOR is implemented as a C **struct** in the code that Pro\*C/C++ generates. So you can always pass it by pointer to another function, or return a pointer to a cursor variable from a function. But you can only pass it or return it by value if your C compiler supports these operations.

### 4.5.2 Allocate a Cursor Variable

Before you can use a cursor variable, either to open it or to FETCH it, you must allocate the cursor. You do this using the new precompiler command ALLOCATE. For example, to allocate the SQL\_CURSOR *emp\_cursor* that was declared in the example earlier, you write the statement:

```
EXEC SQL ALLOCATE :emp cursor;
```

Allocating a cursor does *not* require a call to the server, either at precompile time or at runtime. If the ALLOCATE statement contains an error (for example, an undeclared host variable), Pro\*C/C++ issues a precompile-time error. Allocating a cursor variable *does* cause heap memory to be used. For this reason, you can free a cursor variable in a program loop. Memory allocated for cursor variables is *not* freed when the cursor is closed, but only when an explicit CLOSE is executed, or the connection is closed:

```
EXEC SQL CLOSE :emp_cursor;
```

#### **Related Topics**

Closing and Freeing a Cursor Variable



# 4.5.3 Open a Cursor Variable

You must open a cursor variable on the Oracle database server. You cannot use the embedded SQL OPEN command to open a cursor variable. You can open a cursor variable either by calling a PL/SQL stored procedure that opens the cursor (and defines it in the same statement). Or, you can open and define a cursor variable using an anonymous PL/SQL block in your Pro\*C/C++ program.

For example, consider the following PL/SQL package, stored in the database:

```
CREATE PACKAGE demo cur pkg AS
   TYPE EmpName IS RECORD (name VARCHAR2(10));
   TYPE cur type IS REF CURSOR RETURN EmpName;
   PROCEDURE open emp_cur (
            curs IN OUT cur type,
              dept_num IN NUMBER);
END;
CREATE PACKAGE BODY demo cur pkg AS
   CREATE PROCEDURE open emp cur (
             curs IN OUT cur type,
              dept num IN NUMBER) IS
   BEGIN
       OPEN curs FOR
           SELECT ename FROM emp
               WHERE deptno = dept num
               ORDER BY ename ASC;
    END;
END;
```

After this package has been stored, you can open the cursor *curs* by calling the *open\_emp\_cur* stored procedure from your Pro\*C/C++ program, and FETCH from the cursor in the program. For example:

```
sql_cursor emp_cursor;
char emp_name[11];
...

EXEC SQL ALLOCATE :emp_cursor; /* allocate the cursor variable */
...
/* Open the cursor on the server side. */

EXEC SQL EXECUTE
    begin
        demo_cur_pkg.open_emp_cur(:emp_cursor, :dept_num);
    end;
;
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
    EXEC SQL FETCH :emp_cursor INTO :emp_name;
    printf("%s\n", emp_name);
}
```

To open a cursor using a PL/SQL anonymous block in your Pro\*C/C++ program, you define the cursor in the anonymous block. For example:

```
sql_cursor emp_cursor;
int dept_num = 10;
...
```



```
EXEC SQL EXECUTE
    BEGIN
        OPEN :emp_cursor FOR SELECT ename FROM emp
        WHERE deptno = :dept_num;
    END;
END-EXEC;
...
```

The earlier examples show how to use PL/SQL to open a cursor variable. You can also open a cursor variable using embedded SQL with the CURSOR clause:

```
...
sql_cursor emp_cursor;
...
EXEC ORACLE OPTION(select_error=no);
EXEC SQL
    SELECT CURSOR(SELECT ename FROM emp WHERE deptno = :dept_num)
    INTO :emp_cursor FROM DUAL;
EXEC ORACLE OPTION(select_error=yes);
```

In the statement earlier, the emp\_cursor cursor variable is bound to the first column of the outermost select. The first column is itself a query, but it is represented in the form compatible with a sql\_cursor host variable since the CURSOR(...) conversion clause is used.

Before using queries which involve the CURSOR clause, you must set the SELECT\_ERROR option to NO. This will prevent the cancellation of the parent cursor and allow the program to run without errors.

### 4.5.3.1 Opening in a Standalone Stored Procedure

In the example earlier, a reference cursor was defined inside a package, and the cursor was opened in a procedure in that package. But it is not always necessary to define a reference cursor inside the package that contains the procedures that open the cursor.

If you need to open a cursor inside a standalone stored procedure, you can define the cursor in a separate package, and then reference that package in the standalone stored procedure that opens the cursor. Here is an example:

```
PACKAGE dummy IS

TYPE EmpName IS RECORD (name VARCHAR2(10));

TYPE emp_cursor_type IS REF CURSOR RETURN EmpName;

END;

-- and then define a standalone procedure:

PROCEDURE open_emp_curs (

emp_cursor IN OUT dummy.emp_cursor_type;

dept_num IN NUMBER) IS

BEGIN

OPEN emp_cursor FOR

SELECT ename FROM emp WHERE deptno = dept_num;

END;

END;
```

### 4.5.3.2 Return Types

When you define a reference cursor in a PL/SQL stored procedure, you must declare the type that the cursor returns.

#### **Related Topics**

Cursor Variable Declaration

# 4.5.4 Closing and Freeing a Cursor Variable

Use the CLOSE command to close a cursor variable. For example, to close the *emp\_cursor* cursor variable that was OPENed in the examples earlier, use the embedded SQL statement:

```
EXEC SQL CLOSE :emp_cursor;
```

The cursor variable is a host variable, and so you must precede it with a colon.

You can reuse ALLOCATEd cursor variables. You can open, FETCH, and CLOSE as many times as needed for your application. However, if you disconnect from the server, then reconnect, you must re-ALLOCATE cursor variables.

Cursors are deallocated by the FREE embedded SQL statement. For example:

```
EXEC SQL FREE :emp_cursor;
```

If the cursor is still open, it is closed and the memory allocated for it is released.

# 4.5.5 Cursor Variables with the OCI (Release 7 Only)

You can share a Pro\*C/C++ cursor variable with an OCI function. To do so, you must use the SQLLIB conversion functions, SQLCDAFromResultSetCursor() (formerly known as sqlcdat()) and SQLCDAToResultSetCursor (formerly known as sqlcurt()). These functions convert between OCI cursor data areas and Pro\*C/C++ cursor variables.

The SQLCDAFromResultSetCursor() function translates an allocated cursor variable to an OCI cursor data area. The syntax is:

```
void SQLCDAFromResultSetCursor(dvoid *context, Cda_Def *cda, void *cur,
    sword *retval);
```

#### where the parameters are:

Parameters	Description
context	A pointer to the SQLLIB runtime context.
cda	A pointer to the destination OCI cursor data area.
cur	A pointer to the source Pro*C/C++ cursor variable.
retval	0 if no error, otherwise a SQLLIB (SQL) error number.



In the case of an error, the V2 and rc return code fields in the CDA also receive the error codes. The *rows processed count* field in the CDA is not set.

For non-threaded or default context applications, pass the defined constant SQL\_SINGLE\_RCTX as the context.

The SQLCDATOResultSetCursor() function translates an OCI cursor data area to a Pro\*C/C++ cursor variable. The syntax is:

```
void SQLCDAToResultSetCursor(dvoid *context, void *cur, Cda_Def *cda,
   int *retval);
```



#### where the parameters are:

Parameters	Description	
context	A pointer to the SQLLIB runtime context.	
cur	A pointer to the destination Pro*C/C++ cursor variable.	
cda	A pointer to the source OCI cursor data area.	
retval	0 if no error, otherwise an error code.	

### Note:

The SQLCA structure is not updated by this routine. The SQLCA components are only set after a database operation is performed using the translated cursor.

For non-threaded applications, pass the defined constant SQL\_SINGLE\_RCTX as the context.

ANSI and K&R prototypes for these functions are provided in the sql2oci.h header file. Memory for both *cda* and *cur* must be allocated prior to calling these functions.

#### **Related Topics**

New Names for SOLLIB Public Functions

# 4.5.6 Restrictions (Cursor Variables)

The following restrictions apply to the use of cursor variables:

- If you use the same cursor variable in Pro\*C/C++ and OCI V7, then you must use either SQLLDAGetCurrent() or SQLLDAGetName() immediately after connecting.
- You cannot translate a cursor variable to an OCI release 8 equivalent.
- You cannot use cursor variables with dynamic SQL.
- You can only use cursor variables with the ALLOCATE, FETCH, FREE, and CLOSE commands
- The DECLARE CURSOR command does not apply to cursor variables.
- You cannot FETCH from a CLOSEd cursor variable.
- You cannot FETCH from a non-ALLOCATEd cursor variable.
- If you precompile with MODE=ANSI, it is an error to close a cursor variable that is already closed.
- You cannot use the AT clause with the ALLOCATE command, nor with the FETCH and CLOSE commands if they reference a cursor variable.
- Cursor variables cannot be stored in columns in the database.
- A cursor variable itself cannot be declared in a package specification. Only the type of the cursor variable can be declared in the package specification.
- A cursor variable cannot be a component of a PL/SQL record.



# 4.5.7 Example: cv\_demo.sql and sample11.pc

The following example programs—a PL/SQL script and a Pro\*C/C++ program—demonstrate how you can use cursor variables. These sources are available on-line in your demo directory. Also see another version of the same application, cv demo.pc, in the demo directory.

### 4.5.7.1 cv\_demo.sql

### 4.5.7.2 sample11.pc

```
Fetch from the EMP table, using a cursor variable.
  The cursor is opened in the stored PL/SQL procedure
  open_cur, in the EMP_DEMO_PKG package.
* This package is available on-line in the file
   sample11.sql, in the demo directory.
*/
#include <stdio.h>
#include <sqlca.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
/* Error handling function. */
void sql error(msg)
   char *msg;
   size t clen, fc;
   char cbuf[128];
    clen = sizeof (cbuf);
    sqlgls((char *)cbuf, (size t *)&clen, (size t *)&fc);
    printf("\n%s\n", msg);
    printf("Statement is--\n%s\n", cbuf);
    printf("Function code is %ld\n\n", fc);
    sqlglm((char *)cbuf, (size_t *) &clen, (size_t *) &clen);
```



```
printf ("\n%.*s\n", clen, cbuf);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(EXIT_FAILURE);
void main()
    char temp[32];
    EXEC SQL BEGIN DECLARE SECTION;
       char *uid = "scott/tiger";
        SQL_CURSOR emp_cursor;
        int dept_num;
        struct
            int emp num;
            char emp name[11];
           char job[10];
           int manager;
           char hire date[10];
           float salary;
           float commission;
           int dept_num;
        } emp info;
        struct
            short emp_num_ind;
            short emp name ind;
            short job_ind;
            short manager_ind;
            short hire_date_ind;
            short salary_ind;
            short commission_ind;
            short dept_num_ind;
        } emp_info_ind;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL WHENEVER SQLERROR do sql error("Oracle error");
/* Connect to Oracle. */
    EXEC SQL CONNECT : uid;
/* Allocate the cursor variable. */
    EXEC SQL ALLOCATE :emp_cursor;
/* Exit the inner for (;;) loop when NO DATA FOUND. */
    EXEC SQL WHENEVER NOT FOUND DO break;
    for (;;)
        printf("\nEnter department number (0 to exit): ");
        gets(temp);
        dept num = atoi(temp);
        if (dept num <= 0)
           break;
        EXEC SQL EXECUTE
            begin
                emp_demo_pkg.open_cur(:emp_cursor, :dept_num);
```

```
end:
       END-EXEC;
       printf("\nFor department %d--\n", dept num);
       printf("ENAME SAL COMM\n");
       printf("----
                              ---
                                      ----\n");
/* Fetch each row in the EMP table into the data struct.
  Note the use of a parallel indicator struct. */
       for (;;)
        {
            EXEC SQL FETCH :emp cursor
                INTO :emp_info INDICATOR :emp_info_ind;
            printf("%s ", emp_info.emp_name);
            printf("%8.2f ", emp_info.salary);
            if (emp info ind.commission ind != 0)
                printf("
                          NULL\n");
            else
                printf("%8.2f\n", emp info.commission);
       }
   }
/* Close the cursor. */
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL CLOSE :emp cursor;
/* Disconnect from Oracle. */
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(EXIT SUCCESS);
```

## 4.6 CONTEXT Variables

A *runtime context*, usually simply called a context, is a handle to a an area in client memory which contains zero or more connections, zero or more cursors, their inline options (such as MODE, HOLD\_CURSOR, RELEASE\_CURSOR, SELECT\_ERROR, and so on.) and other additional state information.

To define a context host variable use pseudo-type *sql\_context*. For example:

```
sql context my context;
```

Use the CONTEXT ALLOCATE precompiler directive to allocate and initialize memory for a context:

```
EXEC SQL CONTEXT ALLOCATE : context ;
```

where context is a host variable that is a handle to the context. For example:

```
EXEC SQL CONTEXT ALLOCATE :my_context;
```

Use the CONTEXT USE precompiler directive to define which context is to be used by the embedded SQL statements (such as CONNECT, INSERT, DECLARE CURSOR, and so on.) from that point on in the source file, not in the flow of program logic. That context is used until another CONTEXT USE statement is encountered. The syntax is:

```
EXEC SQL CONTEXT USE {:context | DEFAULT} ;
```

The keyword DEFAULT specifies that the default (also known as global) context is to be used in all the embedded SQL statements that are executed subsequently, until another CONTEXT USE directive is encountered. A simple example is:

```
EXEC SQL CONTEXT USE :my context ;
```

If the context variable my\_context has not been defined and allocated already, an error is returned.

The CONTEXT FREE statement frees the memory used by the context after it is no longer needed:

```
EXEC SQL CONTEXT FREE : context ;

An example is:

EXEC SQL CONTEXT FREE :my_context ;
```

The following example demonstrates the use of a default context in the same application as a user-defined context:

#### **CONTEXT USE Example**

```
#include <sqlca.h>
#include <ociextp.h>
main()
{
    sql_context ctx1;
    char *usr1 = "scott/tiger";
    char *usr2 = "system/manager";

    /* Establish connection to SCOTT in global runtime context */
    EXEC SQL CONNECT :usr1;

    /* Establish connection to SYSTEM in runtime context ctx1 */
    EXEC SQL CONTEXT ALLOCATE :ctx1;
    EXEC SQL CONTEXT USE :ctx1;
    EXEC SQL CONNECT :usr2;

    /* Insert into the emp table from schema SCOTT */
    EXEC SQL CONTEXT USE DEFAULT;
    EXEC SQL INSERT INTO emp (empno, ename) VALUES (1234, 'WALKER');
    ...
}
```

### 4.7 Universal ROWIDs

There are two kinds of table organization used in the database server: *heap tables* and *index-organized tables*.

Heap tables are the default. The physical row address (ROWID) is a permanent property that is used to identify a row in a heap table. The external character format of the physical ROWID is an 18-byte character string in base-64 encoding.

An index-organized table does not have physical row addresses as permanent identifiers. A *logical* ROWID is defined for these tables. When you use a SELECT ROWID ... statement from an index-organized table the ROWID is an opaque structure that contains the primary key of the table, control information, and an optional physical "guess". You can use this ROWID in a SQL statement containing a clause such as "WHERE ROWID = ..." to retrieve values from the table.

Universal ROWID can be used for both physical ROWID and logical ROWID. You can use universal ROWIDs to access data in heap tables, or index-organized tables, since the table organization can change with no effect on applications. The column datatype used for ROWID is UROWID(length), where length is optional.

Use the universal ROWID in all new applications.

Use a universal ROWID variable this way:

- Declare it as type pointer to OCIRowid.
- Allocate memory for the universal ROWID variable.
- Use the universal ROWID as a host bind variable.
- Free the memory when finished.

#### For example:

```
OCIRowid *my_urowid;
...

EXEC SQL ALLOCATE :my_urowid;

/* Bind my_urowid as type SQLT_RDD -- no implicit conversion */

EXEC SQL SELECT rowid INTO :my_urowid FROM my_table WHERE ...;
...

EXEC SQL UPDATE my_table SET ... WHERE rowid = :my_urowid;

EXEC SQL FREE my_urpwid;
...
```

You also have the option of using a character host variable of width between 19 (18 bytes plus the null-terminator) and 4001 as the host bind variable for universal ROWID. Character-based universal ROWIDs are supported for heap tables only for backward compatibility. Because universal ROWID can be variable length, there can be truncation.

Use the character variable this way:

```
/* n is based on table characteristics */
int n=4001;
char my_urowid_char[n];
...

EXEC SQL ALLOCATE :my_urowid_char;
/* Bind my_urowid_char as SQLT_STR */
EXEC SQL SELECT rowid INTO :my_urowid_char FROM my_table WHERE ...;
EXEC ORACLE OPTION(CHAR_MAP=STRING);
EXEC SQL UPDATE my_table SET ... WHERE rowid = :my_urowid_char;
EXEC SQL FREE :my_urowid_char;
...
```

#### **Related Topics**

- Positioned Update
- Logical Storage Structures

# 4.7.1 SQLRowidGet()

A SQLLIB function, SQLRowidGet(), provides the ability to retrieve a pointer to the universal ROWID of the last row inserted, updated, or selected. The function prototype and its arguments are:

```
void SQLRowidGet (dvoid *rctx, OCIRowid **urid) ;
rctx (IN)
```



is a pointer to a runtime context. For the default context or a non-threaded case, pass SQL SINGLE RCTX.

```
urid (OUT)
```

is a pointer to a universal ROWID pointer. When a normal execution finishes, this will point to a valid ROWID. In case of an error, NULL is returned.



The universal ROWID pointer must have been previously allocated to call SQLRowidGet(). Use FREE afterward on the universal ROWID.

# 4.8 Host Structures

You can use a C structure to contain host variables. You reference a structure containing host variables in the INTO clause of a SELECT or a FETCH statement, and in the VALUES list of an INSERT statement. Every component of the host structure must be a legal Pro\*C/C++ host variable, as defined in Table 4-4.

When a structure is used as a host variable, only the name of the structure is used in the SQL statement. However, each of the members of the structure sends data to Oracle, or receives data from Oracle on a query. The following example shows a host structure that is used to add an employee to the EMP table:

```
typedef struct
{
    char emp_name[11]; /* one greater than column length */
    int emp_number;
    int dept_number;
    int dept_number;
    float salary;
} emp_record;
...
/* define a new structure of type "emp_record" */
emp_record new_employee;

strcpy(new_employee.emp_name, "CHEN");
new_employee.emp_number = 9876;
new_employee.dept_number = 20;
new_employee.salary = 4250.00;

EXEC SQL INSERT INTO emp (ename, empno, deptno, sal)
    VALUES (:new employee);
```

The order that the members are declared in the structure must match the order that the associated columns occur in the SQL statement, or in the database table if the column list in the INSERT statement is omitted.

For example, the following use of a host structure is *invalid*, and causes a runtime error:



```
SELECT empno, ename, sal
INTO :emp_record FROM emp;
```

The example is wrong because the components of the structure are not declared in the same order as the associated columns in the select list. The correct form of the SELECT statement is:

```
SELECT empno, sal, ename  /* reverse order of sal and ename */
   INTO :emp_record FROM emp;
```

# 4.8.1 Host Structures and Arrays

An *array* is a collection of related data items, called *elements*, associated with a single variable name. When declared as a host variable, the array is called a *host array*. Likewise, an indicator variable declared as an array is called an *indicator array*. An indicator array can be associated with any host array.

Host arrays can increase performance by letting you manipulate an entire collection of data items with a single SQL statement. With few exceptions, you can use host arrays wherever scalar host variables are allowed. Also, you can associate an indicator array with any host array.

You can use host arrays as components of host structures. In the following example, a structure containing arrays is used to INSERT three new entries into the EMP table:

```
struct
{
    char emp_name[3][10];
    int emp_number[3];
    int dept_number[3];
} emp_rec;
...
strcpy(emp_rec.emp_name[0], "ANQUETIL");
strcpy(emp_rec.emp_name[1], "MERCKX");
strcpy(emp_rec.emp_name[2], "HINAULT");
emp_rec.emp_number[0] = 1964; emp_rec.dept_number[0] = 5;
emp_rec.emp_number[1] = 1974; emp_rec.dept_number[1] = 5;
emp_rec.emp_number[2] = 1985; emp_rec.dept_number[2] = 5;

EXEC SQL INSERT INTO emp (ename, empno, deptno)
    VALUES (:emp_rec);
```

#### **Related Topics**

Host Arrays

## 4.8.2 PL/SQL Records

You cannot bind a C struct to a PL/SQL record.

### 4.8.3 Nested Structures and Unions

You cannot nest host structures. The following example is invalid:

```
struct
{
    int emp_number;
    struct
```



Also, you cannot use a C **union** as a host structure, nor can you nest a **union** in a structure that is to be used as a host structure.

### 4.8.4 Host Indicator Structures

When you need to use indicator variables, but your host variables are contained in a host structure, you set up a second structure that contains an indicator variable for each host variable in the host structure.

For example, suppose you declare a host structure student\_record as follows:

```
struct
{
    char s_name[32];
    int s_id;
    char grad_date[9];
} student_record;
```

If you want to use the host structure in a guery such as

```
EXEC SQL SELECT student_name, student_idno, graduation_date
   INTO :student_record
   FROM college_enrollment
   WHERE student idno = 7200;
```

and you need to know if the graduation date can be NULL, then you must declare a separate host indicator structure. You declare this as

```
struct
{
    short s_name_ind; /* indicator variables must be shorts */
    short s_id_ind;
    short grad_date_ind;
} student_record_ind;
```

Reference the indicator structure in the SQL statement in the same way that you reference a host indicator variable:

```
EXEC SQL SELECT student_name, student_idno, graduation_date
   INTO :student_record INDICATOR :student_record_ind
   FROM college_enrollment
   WHERE student_idno = 7200;
```

When the query completes, the NULL/NOT NULL status of each selected component is available in the host indicator structure.

### Note:

This Guide conventionally names indicator variables and indicator structures by appending \_ind to the host variable or structure name. However, the names of indicator variables are completely arbitrary. You can adopt a different convention, or use no convention at all.

# 4.8.5 Example Program: Cursor and a Host Structure

The demonstration program in this section shows a query that uses an explicit cursor, selecting data into a host structure. This program is available in the file <code>sample2.pc</code> in your <code>demo</code> directory.

```
sample2.pc
 ^{\star} This program connects to ORACLE, declares and opens a cursor,
 ^{\star} \, fetches the names, salaries, and commissions of all
 * salespeople, displays the results, then closes the cursor.
#include <stdio.h>
#include <sqlca.h>
#define UNAME LEN
                     20
#define PWD LEN
                       40
* Use the precompiler typedef'ing capability to create
* null-terminated strings for the authentication host
 * variables. (This isn't really necessary--plain char *'s
 * does work as well. This is just for illustration.)
typedef char asciiz[PWD LEN];
EXEC SQL TYPE asciiz IS STRING (PWD LEN) REFERENCE;
asciiz username;
asciiz password;
struct emp info
   asciiz emp name;
   float salary;
float commission;
};
/* Declare function to handle unrecoverable errors. */
void sql_error();
main()
    struct emp_info *emp_rec_ptr;
/* Allocate memory for emp_info struct. */
    if ((emp rec ptr =
        (struct emp info *) malloc(sizeof(struct emp info))) == 0)
```



```
fprintf(stderr, "Memory allocation error.\n");
       exit(1);
/* Connect to ORACLE. */
    strcpy(username, "SCOTT");
    strcpy(password, "TIGER");
    EXEC SQL WHENEVER SQLERROR DO sql error("ORACLE error--");
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user: %s\n", username);
/* Declare the cursor. All static SQL explicit cursors
 * contain SELECT commands. 'salespeople' is a SQL identifier,
 * not a (C) host variable.
    EXEC SQL DECLARE salespeople CURSOR FOR
       SELECT ENAME, SAL, COMM
           FROM EMP
           WHERE JOB LIKE 'SALES%';
/* Open the cursor. */
    EXEC SQL OPEN salespeople;
/* Get ready to print results. */
    printf("\n);
    printf("Salesperson Salary Commission\n");
    printf("-----
                        -----
/* Loop, fetching all salesperson's statistics.
 * Cause the program to break the loop when no more
 * data can be retrieved on the cursor.
    EXEC SQL WHENEVER NOT FOUND DO break;
    for (;;)
       EXEC SQL FETCH salespeople INTO :emp rec ptr;
       printf("%-11s%9.2f%13.2f\n", emp rec ptr->emp name,
               emp rec ptr->salary, emp rec ptr->commission);
/* Close the cursor. */
    EXEC SQL CLOSE salespeople;
    printf("\nArrivederci.\n\n");
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
void
sql error(msg)
char *msg;
    char err_msg[512];
    int buf len, msg len;
```



```
EXEC SQL WHENEVER SQLERROR CONTINUE;

printf("\n%s\n", msg);

/* Call sqlglm() to get the complete text of the
 * error message.
 */

buf_len = sizeof (err_msg);
 sqlglm(err_msg, &buf_len, &msg_len);
 printf("%.*s\n", msg_len, err_msg);

EXEC SQL ROLLBACK RELEASE;
 exit(1);
}
```

## 4.9 Pointer Variables

C supports *pointers*, which "point" to other variables. A pointer holds the address (storage location) of a variable, not its value.

### 4.9.1 Pointer Variable Declaration

You define pointers as host variables following the normal C practice, as the next example shows:

```
int *int_ptr;
char *char ptr;
```

# 4.9.2 Pointer Variable Referencing

In SQL statements, prefix pointers with a colon, as shown in the following example:

```
EXEC SQL SELECT intcol INTO :int ptr FROM ...
```

Except for pointers to character strings, the size of the referenced value is given by the size of the base type specified in the declaration. For pointers to character strings, the referenced value is assumed to be a NULL-terminated string. Its size is determined at run time by calling the strlen() function.

You can use pointers to reference the members of a **struct**. First, declare a pointer host variable, then set the pointer to the address of the desired member, as shown in the example later. The datatypes of the **struct** member and the pointer variable must be the same. Most compilers will warn you of a mismatch.

```
struct
{
    int i;
    char c;
} structvar;
int *i_ptr;
char *c_ptr;
...
main()
{
    i_ptr = &structvar.i;
    c_ptr = &structvar.c;
/* Use i_ptr and c_ptr in SQL statements. */
...
```



#### **Related Topics**

Globalization Support

### 4.9.3 Structure Pointers

You can use a pointer to a structure as a host variable. The following example

- Declares a structure
- Declares a pointer to the structure
- Allocates memory for the structure
- Uses the struct pointer as a host variable in a query
- Dereferences the struct components to print the results

```
struct EMP_REC
{
    int emp_number;
    float salary;
};
char *name = "HINAULT";
...
struct EMP_REC *sal_rec;
sal_rec = (struct EMP_REC *) malloc(sizeof (struct EMP_REC));
...
EXEC SQL SELECT empno, sal INTO :sal_rec
    FROM emp
    WHERE ename = :name;

printf("Employee number and salary for %s: ", name);
printf("%d, %g\n", sal_rec->emp_number, sal_rec->salary);
```

In the SQL statement, pointers to host structures are referred to in exactly the same way as a host structure. The "address of" notation (&) is not required; in fact, it is an error to use it.

# 4.10 Globalization Support

Although the widely-used 7- or 8-bit ASCII and EBCDIC character sets are adequate to represent the Roman alphabet, some Asian languages, such as Japanese, contain thousands of characters. These languages can require at least 16 bits (two bytes) to represent each character. How does Oracle deal with such dissimilar languages?

Oracle provides Globalization Support, which lets you process single-byte and multibyte character data and convert between character sets. It also lets your applications run in different language environments. With Globalization Support, number and date formats adapt automatically to the language conventions specified for a user session. Thus, Globalization Support allows users around the world to interact with Oracle in their native languages.

You control the operation of language-dependent features by specifying various Globalization Support or NLS parameters. Default values for these parameters can be set in the Oracle initialization file. The following table shows what each Globalization Support parameter specifies.



Table 4-6 Globalization Support Parameters

Globalization Support Parameter	Specifies	
NLS_LANGUAGE	language-dependent conventions	
NLS_TERRITORY	territory-dependent conventions	
NLS_DATE_FORMAT	date format	
NLS_DATE_LANGUAGE	language for day and month names	
NLS_NUMERIC_CHARACTERS	decimal character and group separator	
NLS_CURRENCY	local currency symbol	
NLS_ISO_CURRENCY	ISO currency symbol	
NLS_SORT	sort sequence	

The main parameters are NLS\_LANGUAGE and NLS\_TERRITORY. NLS\_LANGUAGE specifies the default values for language-dependent features, which include:

- Language for server messages
- Language for day and month names
- Sort sequence

NLS\_TERRITORY specifies the default values for territory-dependent features, which include

- Date format
- Decimal character
- Group separator
- Local currency symbol
- ISO currency symbol

You can control the operation of language-dependent Globalization Support features for a user session by specifying the parameter NLS\_LANG as follows:

```
NLS_LANG = <language>_<territory>.<character set>
```

where *language* specifies the value of NLS\_LANGUAGE for the user session, *territory* specifies the value of NLS\_TERRITORY, and *character set* specifies the encoding scheme used for the terminal. An *encoding scheme* (usually called a character set or code page) is a range of numeric codes that corresponds to the set of characters a terminal can display. It also includes codes that control communication with the terminal.

You define NLS\_LANG as an environment variable (or the equivalent on your system). For example, on UNIX using the C shell, you might define NLS\_LANG as follows:

```
setenv NLS_LANG French_France.WE8ISO8859P1
```

During an Oracle database session you can change the values of Globalization Support parameters. Use the ALTER SESSION statement as follows:

```
ALTER SESSION SET <globalization support_parameter> = <value>
```

Pro\*C/C++ fully supports all the Globalization Support features that allow your applications to process foreign language data stored in an Oracle database. For example, you can declare foreign language character variables and pass them to string functions such as INSTRB, LENGTHB, and SUBSTRB. These functions have the same syntax as the INSTR, LENGTH,



and SUBSTR functions, respectively, but operate on a byte-by-byte basis rather than a character-by-character basis.

You can use the functions NLS\_INITCAP, NLS\_LOWER, and NLS\_UPPER to handle special instances of case conversion. And, you can use the function NLSSORT to specify WHERE-clause comparisons based on linguistic rather than binary ordering. You can even pass globalization support parameters to the TO\_CHAR, TO\_DATE, and TO\_NUMBER functions.

### 4.11 NCHAR Variables

Three internal database datatypes can store National Character Set data. They are *NCHAR*, *NCLOB*, and *NVARCHAR2* (also known as *NCHAR VARYING*). You use these datatypes only in relational columns.

# 4.11.1 CHARACTER SET [IS] NCHAR\_CS

To specify which host variables hold National Character Set data, insert the clause "CHARACTER SET [IS] NCHAR\_CS" in character variable declarations. Then you are able to store National Character Set data in those variables. You can omit the token IS. NCHAR\_CS is the name of the National Character Set.

#### For example:

```
char character set is nchar_cs *str = "<Japanese_string>";
```

In this example, *<Japanese\_string>* consists of Unicode characters that are in the National Character Set *AL16UTF16*, as defined by the variable NLS\_NCHAR.

You can accomplish the same thing by entering NLS\_CHAR=str on the command line, and coding in your application:

```
char *str = "<Japanese string>"
```

Pro\*C/C++ treats variables declared this way as of the character set specified by the environment variable *NLS\_NCHAR*. The variable size of an NCHAR variable is specified as a byte count, the same way that ordinary C variables are.

To select data into str, use the following simple query:

```
EXEC SQL
   SELECT ENAME INTO :str FROM EMP WHERE DEPT = n'<Japanese_string1>';
```

Or, you can use *str* in the following SELECT:

```
EXEC SQL
   SELECT DEPT INTO :dept FROM DEPT_TAB WHERE ENAME = :str;
```

# 4.11.2 Environment Variable NLS\_NCHAR

Pro\*C/C++ supports National Character Sets with database support when NLS\_LOCAL=NO. When NLS\_LOCAL=NO, and the new environmental variable NLS\_NCHAR is set to a valid National Character Set, the database server supports NCHAR.

NLS\_NCHAR specifies the character set used for National Character Set data (NCHAR, NVARCHAR2, NCLOB). If it is not specified, the character set defined or indirectly defined by NLS\_LANG will be used.

NLS\_NCHAR must have a valid National Character Set specification (not a language name, that is set by NLS\_LANG) at both precompile-time and runtime. SQLLIB performs a runtime

check when the first SQL statement is executed. If the precompile-time and runtime character sets are different, SQLLIB will return an error code.

### 4.11.3 CONVBUFSZ Clause in VAR

You can override the default assignments by equivalencing host variables to Oracle external datatypes, using the EXEC SQL VAR statement, This is called *host variable equivalencing*.

The EXEC SQL VAR statement can have an optional clause: <code>CONVBUFSZ (<size>)</code>. You specify the size, <code><size></code>, in bytes, of the buffer in the Oracle runtime library used to perform conversion of the specified host variable between character sets.

#### The new syntax is:

```
EXEC SQL VAR host_variable IS datatype [CONVBUFSZ [IS] (size)];

or

EXEC SQL VAR host_variable [CONVBUFSZ [IS] (size)];

where datatype is:

type_name [ ( { length | precision, scale } ) ]
```

#### **Related Topics**

VAR (Oracle Embedded SQL Directive)

# 4.11.4 Character Strings in Embedded SQL

A multibyte character string in an embedded SQL statement consists of a character literal that identifies the string as multibyte, immediately followed by the string. The string is enclosed in the usual single quotes.

For example, an embedded SQL statement such as

```
EXEC SQL SELECT empno INTO :emp_num FROM emp
  WHERE ename = N'<Japanese_string>';
```

contains a multibyte character string (<Japanese\_string> could actually be Kanji), since the N character literal preceding the string identifies it as a multibyte string. Since Oracle is case-insensitive, you can use "n" or "N" in the example.

# 4.11.5 Strings Restrictions

You cannot use datatype equivalencing (the TYPE or VAR commands) with multibyte character strings.

Dynamic SQL method 4 is not available for multibyte character string host variables in Pro\*C/C++.

### 4.11.6 Indicator Variables

You can use indicator variables with host character variables that are multibyte characters (as specified using the NLS CHAR option).

# 4.12 Handling Boolean Data

With the Boolean data type support, applications can:

- Bind boolean, integer, and char types of data in BOOLEAN columns.
- Fetch BOOLEAN column data in boolean, integer, and char data type variables.
- Fetch metadata about a BOOLEAN column.

Fetching data from BOOLEAN type column returns true or false if data is fetched in 'boolean' type variable. If output variable type is INTEGER then 1 or 0 integer value is returned. If the column value is NULL, then the declared variable is untouched.

### See Also:

- About Boolean Data Type Support
- Using Boolean Data Type



# **Advanced Topics**

This chapter discusses advanced techniques in Pro\*C/C++ and contains the following topics:

- Character Data
- Datatype Conversion
- Datatype Equivalencing
- The C Preprocessor
- Precompiled Header Files
- The Oracle Preprocessor
- Evaluation of Numeric Constants
- SQLLIB Extensions for OCI Release 8 Interoperability
- Interface to OCI Release 8
- Embedded OCI Release 7 Calls
- New Names for SQLLIB Public Functions
- X/Open Application Development

### 5.1 Character Data

This section explains how the Pro\*C/C++ Precompiler handles character host variables. There are four host variable character types:

- Character arrays
- · Pointers to strings
- VARCHAR variables
- Pointers to VARCHARs

Do not confuse VARCHAR (a host variable data structure supplied by the precompiler) with VARCHAR2 (an Oracle internal datatype for variable-length character strings).

## 5.1.1 Precompiler Option CHAR MAP

The CHAR\_MAP precompiler option is available to specify the default mapping of char[n] and char host variables. Oracle maps them to CHARZ. CHARZ implements the ANSI Fixed Character format. Strings are fixed-length, blank-padded and null-terminated. VARCHAR2 values (including nulls) are always fixed-length and blank-padded. Table 5-1 shows the possible settings of CHAR\_MAP:

Table 5-1 CHAR\_MAP Settings

CHAR_MAP Setting	Is Default for	Description
VARCHAR2	-	All values (including null) are fixed-length blank-padded.
CHARZ	DBMS=V7, DBMS=V8	Fixed-length blank-padded, then null-terminated. Conforms to the ANSI Fixed Character type.
STRING	New format	null-terminated. Conforms to ASCII format used in C programs.
CHARF	Previously, only through VAR or TYPE declarations.	Fixed-length blank-padded. null is left unpadded.

The default mapping is CHAR\_MAP=CHARZ, which was the case in previous versions of Pro\*C/C++.

Use CHAR\_MAP=VARCHAR2 instead of the old DBMS=V6\_CHAR, which is obsolete.

# 5.1.2 Inline Usage of the CHAR\_MAP Option

Unless you declared a char or char[n] variable otherwise, the inline CHAR\_MAP option determines its mapping. The following code fragment illustrates the results of setting this option inline in Pro\*C/C++:

```
char ch array[5];
strncpy(ch_array, "12345", 5);
^{-} char map=charz is the default in Oracle7 and Oracle8 */
EXEC ORACLE OPTION (char_map=charz);
/* Select retrieves a string "AB" from the database */
SQL SELECT ... INTO :ch array FROM ... WHERE ...;
/* ch array == { 'A', 'B', ' ', ' ', '\0' } */
strncpy (ch array, "12345", 5);
EXEC ORACLE OPTION (char map=string) ;
/* Select retrieves a string "AB" from the database */
EXEC SQL SELECT ... INTO :ch array FROM ... WHERE ...;
/* ch array == { 'A', 'B', \sqrt{0'}, '4', '5' } */
strncpy( ch_array, "12345", 5);
EXEC ORACLE OPTION (char map=charf);
/* Select retrieves a string "AB" from the database */
EXEC SQL SELECT ... INTO :ch_array FROM ... WHERE ...;
/* ch array == { 'A', 'B', '', '', ''} */
```

# 5.1.3 Effect of the DBMS and CHAR\_MAP Options

The DBMS and CHAR\_MAP options determine how Pro\*C/C++ treats data in character arrays and strings. These options allow your program to observe compatibility with ANSI fixed-length strings, or to maintain compatibility with previous releases of Oracle and Pro\*C/C++ that use variable-length strings.

The DBMS option affects character data both on input (from your host variables to the Oracle table) and on output (from an Oracle table to your host variables).

Character Array and the CHAR\_MAP Option

The mapping of character arrays can also be set by the CHAR\_MAP option independent of the DBMS option. DBMS=V7 or DBMS=V8 both use CHAR\_MAP=CHARZ, which can be overridden by specifying either CHAR\_MAP=VARCHAR2 or STRING or CHARF.

#### **Related Topics**

Precompiler Options

### 5.1.3.1 On Input

#### **Character Array**

On input, the DBMS option determines the format that a host variable character array must have in your program. When the CHAR\_MAP=VARCHAR2, host variable character arrays must be blank padded, and should not be null-terminated. When the DBMS=V7 or V8, character arrays must be null-terminated ('\0').

When the CHAR\_MAP option is set to VARCHAR2 trailing blanks are removed up to the first non-blank character before the value is sent to the database. An un-initialized character array can contain null characters. To make sure that the nulls are not inserted into the table, you must blank-pad the character array to its length. For example, if you execute the statements:

you will find that the string "MILLER" was inserted as "MILLER\0\0\0\0" (with four null bytes appended to it). This value does not meet the following search condition:

```
. . . WHERE ename = 'MILLER';
```

To INSERT the character array when CHAR\_MAP is set to VARCHAR2, you should execute the statements

When DBMS=V7 or V8, input data in a character array must be null-terminated. So, make sure that your data ends with a null.

### Note:

During the execution of insert statements, when CHAR\_MAP=CHARZ or CHARF, any bind variables with right spaces are trimmed.

#### **Character Pointer**

The pointer must address a null-terminated buffer that is large enough to hold the input data. Your program must allocate enough memory to do this.

### 5.1.3.2 On Input

The following example illustrates all possible combinations of the effects of the CHAR\_MAP option settings on the value retrieved from a database into a character array.

#### Assume a database

```
TABLE strdbase ( ..., strval VARCHAR2(6));
```

which contains the following strings in the column strval:

```
""AB" -- string of length 0
"AB" -- string of length 2
"KING" -- string of length 4
"QUEEN" -- string of length 5
"MILLER" -- string of length 6
```

In a Pro\*C/C++ program, initialize the 5-character host array *str* with 'X' characters and use for the retrieval of all the values in column strval:

```
char str[5] = {'X', 'X', 'X', 'X', 'X'};
short str_ind;
...
EXEC SQL SELECT strval INTO :str:str ind WHERE ...;
```

with the following results for the array, str, and the indicator variable, str\_ind, as CHAR\_MAP is set to VARCHAR2, CHARF, CHARZ and STRING:

```
Strval = "" "AB" "KING" "QUEEN" "MILLER"

VARCHAR2 " " -1 "AB " 0 "KING " 0 "QUEEN" 0 "MILLE" 6

CHARF "XXXXX" -1 "AB " 0 "KING " 0 "QUEEN" 0 "MILLE" 6

CHARZ " 0" -1 "AB 0" 0 "KING0" 0 "QUEE0" 5 "MILLO" 6

STRING "0XXXX" -1 "AB0XX" 0 "KING0" 0 "QUEE0" 5 "MILLO" 6
```

where 0 stands for the null character, '\0'.

## 5.1.3.3 On Output

#### **Character Array**

On output, the DBMS and CHAR\_MAP options determines the format that a host variable character array will have in your program. When CHAR\_MAP=VARCHAR2, host variable character arrays are blank padded up to the length of the array, but never null-terminated. When DBMS=V7 or V8 (or CHAR\_MAP=CHARZ), character arrays are blank padded, then null-terminated in the final position in the array.

Consider the following example of character output:

```
CREATE TABLE test_char (C_col CHAR(10), V_col VARCHAR2(10));
INSERT INTO test char VALUES ('MILLER', 'KING');
```

A precompiler program to select from this table contains the following embedded SQL:

```
char name1[10];
```



```
EXEC SQL SELECT C_col, V_col INTO :name1, :name2
    FROM test char;
```

If you precompile the program with CHAR\_MAP=VARCHAR2, name1 will contain:

```
"MILLER###"
```

that is, the name "MILLER" followed by 4 blanks, with no null-termination. (If *name1* had been declared with a size of 15, there are 9 blanks following the name.)

name2 will contain:

```
"KING#####" /* 6 trailing blanks */
```

If you precompile the program with DBMS=V7 or V8, name1 will contain:

```
"MILLER###\0" /* 3 trailing blanks, then a null-terminator */
```

that is, a string containing the name, blank-padded to the length of the column, followed by a null terminator. *name2* will contain:

```
"KING####\0"
```

In summary, if CHAR\_MAP=VARCHAR2, the output from either a CHARACTER column or a VARCHAR2 column is blank-padded to the length of the host variable array. If DBMS=V7 or V8, the output string is always null-terminated.

#### **Character Pointer**

The DBMS and CHAR\_MAP options do not affect the way character data are output to a pointer host variable.

When you output data to a character pointer host variable, the pointer must point to a buffer large enough to hold the output from the table, plus one extra byte to hold a null terminator.

The precompiler runtime environment calls <code>strlen()</code> to determine the size of the output buffer, so make sure that the buffer does not contain any embedded nulls ('\0'). Fill allocated buffers with some value other than '\0', then null-terminate the buffer, before fetching the data.



C pointers can be used in a Pro\*C/C++ program that is precompiled with DBMS=V7 or V8 and MODE=ANSI. However, pointers are not legal host variable types in a SQL standard compliant program. The FIPS flagger warns you if you use pointers as host variables.

The following code fragment uses the columns and table defined in the previous section, and shows how to declare and SELECT into character pointer host variables:

```
char *p_name1;
char *p_name2;
...
p_name1 = (char *) malloc(11);
p_name2 = (char *) malloc(11);
strcpy(p_name1, " ");
strcpy(p name2, "0123456789");
```



```
EXEC SQL SELECT C_col, V_col INTO :p_name1, :p_name2
   FROM test char;
```

When the SELECT statement mentioned earlier is executed with any DBMS or CHAR\_MAP setting, the value fetched is:

```
"MILLER####0" /* 4 trailing blanks and a null terminator */
"KING######0" /* 6 blanks and null */
```

### 5.1.4 VARCHAR Variables and Pointers

The following example shows how VARCHAR host variables are declared:

```
VARCHAR emp_name1[10];  /* VARCHAR variable */
VARCHAR *emp_name2;  /* pointer to VARCHAR */
```

### 5.1.4.1 On Input

#### **VARCHAR Variables**

When you use a VARCHAR variable as an input host variable, your program need only place the desired string in the array member of the expanded VARCHAR declaration (*emp\_name1.arr* in our example) and set the length member (*emp\_name1.len*). There is no need to blank-pad the array. Exactly *emp\_name1.len* characters are sent to Oracle, counting any blanks and nulls. In the following example, you set *emp\_name1.len* to 8:

```
strcpy((char *)emp_name1.arr, "VAN HORN");
emp_name1.len = strlen((char *)emp_name1.arr);
```

#### Pointer to a VARCHAR

When you use a pointer to a VARCHAR as an input host variable, you must allocate enough memory for the expanded VARCHAR declaration. Then, you must place the desired string in the array member and set the length member, as shown in the following example:

```
emp_name2 = malloc(sizeof(short) + 10)     /* len + arr */
strcpy((char *)emp_name2->arr, "MILLER");
emp_name2->len = strlen((char *)emp_name2->arr);
```

Or, to make *emp\_name2* point to an existing VARCHAR (*emp\_name1* in this case), you could code the assignment

```
emp_name2 = &emp_name1;
```

then use the VARCHAR pointer in the usual way, as in

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
    VALUES (:emp_number, :emp_name2, :dept_number);
```

### 5.1.4.2 On Output

#### **VARCHAR Variables**

When you use a VARCHAR variable as an output host variable, the program interface sets the length member but does *not* null-terminate the array member. As with character arrays, your program can null-terminate the *arr* member of a VARCHAR variable before passing it to a function such as printf() or strlen(). An example follows:



```
emp_name1.arr[emp_name1.len] = '\0';
printf("%s", emp_name1.arr);
```

Or, you can use the length member to limit the printing of the string, as in:

```
printf("%.*s", emp name1.len, emp name1.arr);
```

An advantage of VARCHAR variables over character arrays is that the length of the value returned by Oracle is available immediately. With character arrays, you might need to strip the trailing blanks yourself to get the actual length of the character string.

### **VARCHAR Pointers**

When you use a pointer to a VARCHAR as an output host variable, the program interface determines the variable's maximum length by checking the length member (*emp\_name2->len* in our example). So, your program must set this member before *every* fetch. The fetch then sets the length member to the actual number of characters returned, as the following example shows:

```
emp_name2->len = 10;  /* Set maximum length of buffer. */
EXEC SQL SELECT ENAME INTO :emp_name2 WHERE EMPNO = 7934;
printf("%d characters returned to emp name2", emp name2->len);
```

### 5.1.5 Unicode Variables

Pro\*C/C++ allows fixed-width Unicode data (character set Unicode Standard Version 3.0, known simply as UCS-16) in host char variables. UCS-16 uses 2 bytes for each character, so it is an unsigned 2-byte datatype. SQL statement text in UCS-16 is not supported yet.

In the following example code a host variable, <code>employee</code>, of the Unicode type *utext* is declared to be 20 Unicode characters long. A table <code>emp</code> is created containing the column <code>ename</code>, which is 60 bytes long, so that database character sets in Asian languages, where multibyte characters are up to three bytes long, will be supported.

A public header file, sqlucs2.h, must be included in your application code. It does the following:

Contains the statement:

```
#include <oratypes.h>
```

Defines a "Unicode varchar", uvarchar, as:

```
struct uvarchar
{
   ub2 len;
   utext arr[1];
};
typedef struct uvarchar uvarchar;
```

Defines a "Unicode long varchar", ulong\_varchar, as:

```
struct ulong_varchar
{
   ub4 len ;
   utext arr[1] ;
}
typedef struct ulong varchar ulong varchar ;
```

The default datatype of utext is the same as the default for any character variables, CHARZ, which is blank-padded and null-terminated.

Use the CHAR\_MAP precompiler option to change the default datatype, as follows:

## 5.1.5.1 Restrictions on Unicode Variable Usage

 Static and dynamic SQL cannot contain Unicode in the SQL statement text. The following is not permitted:

```
#include oratypes.h
utext sqlstmt[100];
...
/* If sqlstmt contains a SQL statement: */
EXEC SQL PREPARE s1 FROM :sqlstmt;
EXEC SQL EXECUTE IMMEDIATE :sqlstmt;
```

You cannot use type equivalencing for utext variables. The following code is not permitted:

```
typedef utext utext_5 ;
EXEC SQL TYPE utext 5 IS STRING ;
```

- CONVBUFSZ cannot be used as a conversion buffer size. Use the CHAR\_MAP option instead.
- Oracle dynamic SQL method 4 does not support Unicode.
- Object types do not support Unicode.

#### **Related Topics**

- CONVBUFSZ Clause in VAR
- ANSI Dynamic SQL



#### Objects

# 5.2 Datatype Conversion

At precompile time, a default external datatype is assigned to each host variable. For example, the precompiler assigns the INTEGER external datatype to host variables of type **short int** and **int**.

At run time, the datatype code of every host variable used in a SQL statement is passed to Oracle. Oracle uses the codes to convert between internal and external datatypes.

Before assigning a SELECTed column (or pseudocolumn) value to an output host variable, Oracle must convert the internal datatype of the source column to the datatype of the host variable. Likewise, before assigning or comparing the value of an input host variable to a column, Oracle must convert the external datatype of the host variable to the internal datatype of the target column.

Conversions between internal and external datatypes follow the usual data conversion rules. For example, you can convert a CHAR value of "1234" to a C *short* value. You cannot convert a CHAR value of "65543" (number too large) or "10F" (number not decimal) to a C *short* value. Likewise, you cannot convert a char[n] value that contains any alphabetic characters to a NUMBER value.

# 5.3 Datatype Equivalencing

Datatype equivalencing lets you control the way Oracle interprets input data, and the way Oracle formats output data. It provides the ability to override the default external datatypes that the precompiler assigns. On a variable-by-variable basis, you can map (or make equivalent) supported C host variable datatypes to Oracle external datatypes. You can also map user-defined datatypes to Oracle external datatypes.

# 5.3.1 Host Variable Equivalencing

By default, the Pro\*C/C++ Precompiler assigns a specific external datatype to every host variable.

Table 5-2 lists the default assignments:

**Table 5-2 Default Type Assignments** 

C Type, or Pseudotype	Oracle External Type	Notes
char	VARCHAR2	(CHAR_MAP=VARCHAR2)
char[n]	CHARZ	(DBMS=V7, V8 default)
char*	STRING	(CHAR_MAP=STRING)
	CHARF	(CHAR_MAP=CHARF)
int, int*	INTEGER	-
short, short*	INTEGER	-
long, long*	INTEGER	-
long long, long long*	INTEGER	-
float, float*	FLOAT	-
double, double*	FLOAT	-



Table 5-2 (Cont.) Default Type Assignments

C Type, or Pseudotype	Oracle External Type	Notes
VARCHAR*, VARCHAR[n]	VARCHAR	-
bool	BOOLEAN	-

With the VAR statement, you can override the default assignments by equivalencing host variables to Oracle external datatypes. The syntax you use is

```
EXEC SQL VAR host variable IS type name [ (length) ];
```

where *host\_variable* is an input or output host variable (or host array) declared earlier, *type\_name* is the name of a valid external datatype, and *length* is an integer literal specifying a valid length in bytes.

Host variable equivalencing is useful in several ways. For example, suppose you want to SELECT employee names from the EMP table, then pass them to a routine that expects null-terminated strings. You need not explicitly null-terminate the names. Simply equivalence a host variable to the STRING external datatype, as follows:

```
char emp_name[11];
EXEC SQL VAR emp name IS STRING(11);
```

The length of the ENAME column in the EMP table is 10 characters, so you allot the new *emp\_name* 11 characters to accommodate the null terminator. When you SELECT a value from the ENAME column into *emp\_name*, the program interface null-terminates the value for you.

You can use any external datatypes except NUMBER (for example, VARNUM).

#### **Related Topics**

External Datatypes

### 5.3.2 User-Defined Type Equivalencing

You can also map (or make equivalent) user-defined datatypes to Oracle external datatypes. First, define a new datatype structured like the external datatype that suits your needs. Then, map your new datatype to the external datatype using the TYPE statement.

With the TYPE statement, you can assign an Oracle external datatype to a whole class of host variables. The syntax you use is:

```
EXEC SQL TYPE user type IS type name [ (length) ] [REFERENCE];
```

Suppose you need a variable-length string datatype to hold graphics characters. First, declare a struct with a **short** length component followed by a 65533-byte data component. Second, use **typedef** to define a new datatype based on the struct. Then, equivalence your new user-defined datatype to the VARRAW external datatype, as shown in the following example:

```
struct screen
{
    short len;
    char buff[4000];
};
typedef struct screen graphics;
```



```
EXEC SQL TYPE graphics IS VARRAW(4000);
graphics crt; - host variable of type graphics
...
```

You specify a length of 4000 bytes for the new *graphics* type because that is the maximum length of the data component in your struct. The precompiler allows for the *len* component (and any padding) when it sends the length to the Oracle server.

### 5.3.2.1 REFERENCE Clause

You can declare a user-defined type to be a pointer, either explicitly, as a pointer to a scalar or struct type, or implicitly, as an array, and use this type in an EXEC SQL TYPE statement. In this case, you must use the REFERENCE clause at the end of the statement, as shown in the following example:

```
typedef unsigned char *my_raw;

EXEC SQL TYPE my_raw IS VARRAW(4000) REFERENCE;
my_raw graphics_buffer;
...
graphics buffer = (my raw) malloc(4004);
```

In this example, you allocated additional memory over the type length (4000). This is necessary because the precompiler also returns the length (the size of a *short*), and can add padding after the length due to word alignment restrictions on your system. If you do not know the alignment practices on your system, make sure to allocate sufficient extra bytes for the length and padding (9 should usually be sufficient).

#### **Related Topics**

Example Program: Using sqlvcp()

### 5.3.3 CHARF External Datatype

CHARF is a fixed-length character string. You can use this datatype in VAR and TYPE statements to equivalence C datatypes to the fixed-length SQL standard datatype CHAR, regardless of the setting of the DBMS or CHAR\_MAP option.

When DBMS=V7 or V8, specifying the external datatype CHARACTER in a VAR or TYPE statement equivalences the C datatype to the fixed-length datatype CHAR (datatype code 96). However, when CHAR\_MAP=VARCHAR2, the C datatype is equivalenced to the variable-length datatype VARCHAR2 (code 1).

Now, you can always equivalence C datatypes to the fixed-length SQL standard type CHARACTER by using the CHARF datatype in the VAR or TYPE statement. When you use CHARF, the equivalence is always made to the fixed-length character type, regardless of the setting of the DBMS or CHAR MAP option.

# 5.3.4 The EXEC SQL VAR and TYPE Directives

You can code an EXEC SQL VAR ... or EXEC SQL TYPE ... statement anywhere in your program. These statements are treated as executable statements that change the datatype of any variable affected by them from the point that the TYPE or VAR statement was made to the end of the scope of the variable. If you precompile with MODE=ANSI, you must use Declare Sections. In this case, the TYPE or VAR statement must be in a Declare Section.

#### **Related Topics**

TYPE (Oracle Embedded SQL Directive)



VAR (Oracle Embedded SQL Directive)

## 5.3.5 Example: Datatype Equivalencing (sample4.pc):

The demonstration program in this section shows you how you can use datatype equivalencing in your Pro\*C/C++ programs. This program is available as sample4.pc in the demo directory.It demonstrates the use of type equivalencing using the LONG VARRAW external datatype. In order to provide a useful example that is portable across different systems, the program inserts binary files into and retrieves them from the database.

This program uses LOB embedded SQL statements.

Please read the introductory comments for an explanation of the program's purpose.

```
/********************
sample4.pc
This program demonstrates the use of type equivalencing using the
LONG VARRAW external datatype. In order to provide a useful example
that is portable across different systems, the program inserts
binary files into and retrieves them from the database. For
example, suppose you have a file called 'hello' in the current
directory. You can create this file by compiling the following
source code:
#include <stdio.h>
int main()
  printf("Hello World!\n");
When this program is run, we get:
$hello
Hello World!
Here is some sample output from a run of sample4:
$sample4
Connected.
Do you want to create (or re-create) the EXECUTABLES table (y/n)? y
EXECUTABLES table successfully dropped. Now creating new table...
EXECUTABLES table created.
Sample 4 Menu. Would you like to:
(I)nsert a new executable into the database
(R) etrieve an executable from the database
(L) ist the executables stored in the database
(D) elete an executable from the database
(Q) uit the program
Enter i, r, l, or q: l
Executables
                   Length (bytes)
Total Executables: 0
Sample 4 Menu. Would you like to:
(I) nsert a new executable into the database
(R) etrieve an executable from the database
```



```
(L) ist the executables stored in the database
(D)elete an executable from the database
(Q) uit the program
Enter i, r, l, or q: i
Enter the key under which you will insert this executable: hello
Enter the filename to insert under key 'hello'.
If the file is not in the current directory, enter the full
path: hello
Inserting file 'hello' under key 'hello'...
Inserted.
Sample 4 Menu. Would you like to:
(I)nsert a new executable into the database
(R)etrieve an executable from the database
(L) ist the executables stored in the database
(D) elete an executable from the database
(Q) uit the program
Enter i, r, l, or q: l
Executables
               Length (bytes)
                       5508
hello
Total Executables: 1
Sample 4 Menu. Would you like to:
(I)nsert a new executable into the database
(R) etrieve an executable from the database
(L) ist the executables stored in the database
(D)elete an executable from the database
(Q) uit the program
Enter i, r, l, or q: r
Enter the key for the executable you wish to retrieve: hello
Enter the file to write the executable stored under key hello into. If you
don't want the file in the current directory, enter the
full path: h1
Retrieving executable stored under key 'hello' to file 'h1'...
Retrieved.
Sample 4 Menu. Would you like to:
(I) nsert a new executable into the database
(R) etrieve an executable from the database
(L) ist the executables stored in the database
(D)elete an executable from the database
(Q) uit the program
Enter i, r, l, or q: q
We now have the binary file 'h1' created, and we can run it:
$h1
Hello World!
#include <oci.h>
#include <string.h>
#include <stdio.h>
#include <sqlca.h>
#include <stdlib.h>
```

```
#include <sqlcpr.h>
/* Oracle error code for 'table or view does not exist'. */
#define NON EXISTENT -942
#define NOT_FOUND
                      1403
/* This is the definition of the long varraw structure.
* Note that the first field, len, is a long instead
 \star of a short. This is because the first 4
 * bytes contain the length, not the first 2 bytes.
typedef struct long varraw {
 ub4 len;
 text buf[1];
} long_varraw;
/* Type Equivalence long varraw to LONG VARRAW.
* All variables of type long varraw from this point
 * on in the file will have external type 95 (LONG VARRAW)
 * associated with them.
EXEC SQL TYPE long varraw IS LONG VARRAW REFERENCE;
/* This program's functions declared. */
#if defined( STDC )
 void do connect(void);
 void create table(void);
 void sql error(char *);
 void list executables(void);
 void print_menu(void);
 void do_insert(varchar *, char *);
 void do_retrieve(varchar *, char *);
 void do_delete(varchar *);
 ub4 read_file(char *, OCIBlobLocator *);
 void write_file(char *, OCIBlobLocator *);
#else
 void do connect(/* void */);
 void create table(/* void */);
 void sql_error(/*_ char * _*/);
 void list executables(/* void */);
 void print_menu(/*_ void _*/);
void do_insert(/*_ varchar *, char * _*/);
 void do_retrieve(/*_ varchar *, char * _*/);
 void do_delete(/*_ varchar * _*/);
ub4 read_file(/*_ char *, OCIBlobLocator * _*/);
 void write_file(/*_ char *, OCIBlobLocator * _ */);
#endif
void main()
 char reply[20], filename[100];
  varchar key[20];
  short ok = 1;
  /* Connect to the database. */
  do_connect();
 printf("Do you want to create (or re-create) the EXECUTABLES table (y/n)?");
  gets(reply);
```

```
if ((reply[0] == 'y') || (reply[0] == 'Y'))
   create table();
  /* Print the menu, and read in the user's selection. */
 print menu();
 gets(reply);
 while (ok)
   switch(reply[0]) {
   case 'I': case 'i':
      /\!\!\!^* User selected insert - get the key and file name. ^*/\!\!\!
     printf("Enter the key under which you will insert this executable: ");
     key.len = strlen(gets((char *)key.arr));
     printf("Enter the filename to insert under key '%.*s'.\n",
             key.len, key.arr);
     printf("If the file is not in the current directory, enter the full\n");
     printf("path: ");
     gets (filename);
     do insert((varchar *)&key, filename);
     break;
    case 'R': case 'r':
     /* User selected retrieve - get the key and file name. */
     printf("Enter the key for the executable you wish to retrieve: ");
     key.len = strlen(gets((char *)key.arr));
     printf("Enter the file to write the executable stored under key ");
     printf("%.*s into. If you\n", key.len, key.arr);
     printf("don't \ want \ the \ file \ in \ the \ current \ directory, \ enter \ the \verb|\n"|);
     printf("full path: ");
      gets(filename);
     do retrieve((varchar *)&key, filename);
    case 'L': case 'l':
      /* User selected list - just call the list routine. */
     list executables();
     break;
    case 'D': case 'd':
      /\star User selected delete - get the key for the executable to delete. \star/
     printf("Enter the key for the executable you wish to delete: ");
     key.len = strlen(gets((char *)key.arr));
     do delete((varchar *) &key);
     break;
    case 'Q': case 'q':
      /* User selected quit - just end the loop. */
     ok = 0;
     break:
    default:
     /* Invalid selection. */
     printf("Invalid selection.\n");
     break;
   if (ok)
      /* Print the menu again. */
     print menu();
     gets(reply);
 }
 EXEC SQL COMMIT WORK RELEASE;
}
```

```
/* Connect to the database. */
void do connect()
  /* Note this declaration: uid is a char * pointer, so Oracle
    will do a strlen() on it at runtime to determine the length.
 char *uid = "scott/tiger";
 EXEC SQL WHENEVER SQLERROR DO sql error("do connect():CONNECT");
 EXEC SQL CONNECT : uid;
 printf("Connected.\n");
}
/* Creates the executables table. */
void create table()
  /* We are going to check for errors ourselves for this statement. */
 EXEC SOL WHENEVER SOLERROR CONTINUE;
 EXEC SQL DROP TABLE EXECUTABLES;
 if (sqlca.sqlcode == 0)
     printf("EXECUTABLES table successfully dropped. ");
     printf("Now creating new table...\n");
  else if (sqlca.sqlcode == NON_EXISTENT)
   {
     printf("EXECUTABLES table does not exist. ");
     printf("Now creating new table...\n");
 else
   sql_error("create_table()");
  /* Reset error handler. */
 EXEC SQL WHENEVER SQLERROR DO sql error("create table():CREATE TABLE");
 EXEC SQL CREATE TABLE EXECUTABLES
    ( name VARCHAR2(30), length NUMBER(10), binary BLOB );
 printf("EXECUTABLES table created.\n");
/\star Opens the binary file identified by 'filename' for reading, and writes
  it into into a Binary LOB. Returns the actual length of the file read.
ub4 read file(filename, blob)
 char *filename;
 OCIBlobLocator *blob;
 long_varraw *lvr;
 ub4
       bufsize;
 ub4
          amt;
 ub4
         filelen, remainder, nbytes;
          offset = 1;
 boolean last = FALSE;
 FILE *in fd;
  /* Open the file for reading. */
```

```
in fd = fopen(filename, "r");
if (in_fd == (FILE *)0)
  return (ub4)0;
/* Determine Total File Length - Total Amount to Write to BLOB */
(void) fseek(in fd, OL, SEEK END);
amt = filelen = (ub4)ftell(in_fd);
/* Determine the Buffer Size and Allocate the LONG VARRAW Object */
bufsize = 2048;
lvr = (long varraw *)malloc(sizeof(ub4) + bufsize);
nbytes = (filelen > bufsize) ? bufsize : filelen;
/* Reset the File Pointer and Perform the Initial Read */
(void) fseek(in fd, OL, SEEK SET);
lvr->len = fread((void *)lvr->buf, (size t)1, (size t)nbytes, in fd);
remainder = filelen - nbytes;
EXEC SQL WHENEVER SQLERROR DO sql error("read file():WRITE");
if (remainder == 0)
    /* Write the BLOB in a Single Piece */
    EXEC SQL LOB WRITE ONE :amt
       FROM :lvr WITH LENGTH :nbytes INTO :blob AT :offset;
else
    /* Write the BLOB in Multiple Pieces using Standard Polling */
    EXEC SQL LOB WRITE FIRST :amt
       FROM : lvr WITH LENGTH : nbytes INTO : blob AT : offset;
    do {
      if (remainder > bufsize)
        nbytes = bufsize;
      else
          nbytes = remainder;
          last = TRUE;
      if ((lvr->len = fread(
            (void *)lvr->buf, (size_t)1, (size_t)nbytes, in_fd)) != nbytes)
        last = TRUE;
      if (last)
          /* Write the Final Piece */
          EXEC SQL LOB WRITE LAST :amt
            FROM :lvr WITH LENGTH :nbytes INTO :blob;
      else
          /* Write an Interim Piece - Still More to Write */
          EXEC SQL LOB WRITE NEXT :amt
            FROM : lvr WITH LENGTH : nbytes INTO : blob;
      remainder -= nbytes;
```

```
} while (!last && !feof(in fd));
    }
  /* Close the file, and return the total file size. */
  fclose(in fd);
  free(lvr);
 return filelen;
/\star Generic error handler. The 'routine' parameter should contain the name
  of the routine executing when the error occured. This would be specified
  in the 'EXEC SQL WHENEVER SQLERROR DO sql error()' statement.
void sql error(routine)
 char *routine;
 char message buffer[512];
 size t buffer size;
 size t message length;
  /* Turn off the call to sql error() to avoid a possible infinite loop */
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 printf("\nOracle error while executing %s!\n", routine);
  /* Use sqlglm() to get the full text of the error message. */
 buffer size = sizeof(message_buffer);
  sqlglm(message buffer, &buffer size, &message length);
 printf("%.*s\n", message length, message buffer);
 EXEC SQL ROLLBACK WORK RELEASE;
 exit(1);
/\star Opens the binary file identified by 'filename' for writing, and copies
   the contents of the Binary LOB into it.
void write file(filename, blob)
  char *filename;
  OCIBlobLocator *blob;
                            /* File descriptor for the output file */
 FILE
              *out fd;
 ub4
              amt:
 นb4
              bufsize;
 long varraw *lvr;
  /* Determine the Buffer Size and Allocate the LONG VARRAW Object */
 bufsize = 2048;
  lvr = (long varraw *)malloc(sizeof(ub4) + bufsize);
  /* Open the output file for Writing */
  out fd = fopen(filename, "w");
  if (out fd == (FILE *)0)
   return;
  amt = 0;
                       /* Initialize for Standard Polling (Possibly) */
 lvr->len = bufsize;
                                            /* Set the Buffer Length */
  EXEC SQL WHENEVER SQLERROR DO sql error("write file():READ");
```

```
/* READ the BLOB using a Standard Polling Loop */
 EXEC SQL WHENEVER NOT FOUND DO break;
 while (TRUE)
   {
     EXEC SQL LOB READ :amt FROM :blob INTO :lvr WITH LENGTH :bufsize;
      (void) fwrite((void *)lvr->buf, (size t)1, (size t)lvr->len, out fd);
 EXEC SQL WHENEVER NOT FOUND CONTINUE;
  ^{\prime\star} Write the Final Piece (or First and Only Piece if not Polling) ^{\star\prime}
  (void) fwrite((void *)lvr->buf, (size t)lvr->len, (size t)1, out fd);
 /* Close the Output File and Return */
 fclose(out fd);
 free(lvr);
 return;
/* Inserts the binary file identified by file into the
* executables table identified by key.
void do insert(key, file)
 varchar *key;
 char *file;
 OCIBlobLocator *blob;
 ub4 loblen, fillen;
 EXEC SQL ALLOCATE :blob;
 EXEC SQL WHENEVER SQLERROR DO sql error("do insert():INSERT/SELECT");
 EXEC SQL SAVEPOINT PREINSERT;
 EXEC SQL INSERT
   INTO executables (name, length, binary) VALUES (:key, 0, empty blob());
 EXEC SQL SELECT binary INTO :blob
             FROM executables WHERE name = :key FOR UPDATE;
 printf(
   "Inserting file '%s' under key '%.*s'...\n", file, key->len, key->arr);
 fillen = read_file(file, blob);
 EXEC SQL LOB DESCRIBE :blob GET LENGTH INTO :loblen;
 if ((fillen == 0) || (fillen != loblen))
     printf("Problem reading file '%s'\n", file);
     EXEC SQL ROLLBACK TO SAVEPOINT PREINSERT;
     EXEC SQL FREE :blob;
     return;
 EXEC SQL WHENEVER SQLERROR DO sql error("do insert():UPDATE");
 EXEC SQL UPDATE executables
   SET length = :loblen, binary = :blob WHERE name = :key;
 EXEC SQL COMMIT WORK;
```

```
EXEC SQL FREE :blob;
 EXEC SQL COMMIT;
 printf("Inserted.\n");
/* Retrieves the executable identified by key into file */
void do retrieve(key, file)
 varchar *key;
 char *file;
 OCIBlobLocator *blob;
 printf("Retrieving executable stored under key '%.*s' to file '%s'...\n",
        key->len, key->arr, file);
 EXEC SQL ALLOCATE :blob;
 EXEC SQL WHENEVER NOT FOUND continue;
 EXEC SQL SELECT binary INTO :blob FROM executables WHERE name = :key;
 if (sqlca.sqlcode == NOT FOUND)
   printf("Key '%.*s' not found!\n", key->len, key->arr);
 else
     write file(file, blob);
     printf("Retrieved.\n");
 EXEC SQL FREE :blob;
/* Delete an executable from the database */
void do_delete(key)
 varchar *key;
 EXEC SQL WHENEVER SQLERROR DO sql error("do delete():DELETE");
 EXEC SQL DELETE FROM executables WHERE name = :key;
 if (sqlca.sqlcode == NOT FOUND)
   printf("Key '%.*s' not found!\n", key->len, key->arr);
 else
   printf("Deleted.\n");
/* List all executables currently stored in the database */
void list executables()
 char key[21];
 ub4 length;
 EXEC SQL WHENEVER SQLERROR DO sql error("list executables");
 EXEC SQL DECLARE key cursor CURSOR FOR
   SELECT name, length FROM executables;
 EXEC SQL OPEN key cursor;
 printf("\nExecutables
                               Length (bytes) \n");
 printf("-----\n");
```

```
EXEC SQL WHENEVER NOT FOUND DO break;
 while (1)
   EXEC SQL FETCH key_cursor INTO :key, :length;
   printf("%s %10d\n", key, length);
  EXEC SQL WHENEVER NOT FOUND CONTINUE;
 EXEC SQL CLOSE key cursor;
 printf("\nTotal Executables: %d\n", sqlca.sqlerrd[2]);
/* Prints the menu selections. */
void print menu()
 printf("\nSample 4 Menu. Would you like to:\n");
 printf("(I)nsert a new executable into the database\n");
 printf("(R)etrieve an executable from the database\n");
 printf("(L)ist the executables stored in the database\n");
 printf("(D)elete an executable from the database\n");
 printf("(Q)uit the program\n'n");
 printf("Enter i, r, l, or q: ");
```

#### **Related Topics**

LOBs

# 5.4 The C Preprocessor

Pro\*C/C++ supports most C preprocessor directives. Some of the things that you can do using the Pro\*C/C++ preprocessor are:

- Define constants and macros using the #define directive, and use the defined entities to parameterize Pro\*C/C++ datatype declarations, such as VARCHAR
- Read files required by the precompiler, such as sqlca.h, using the #include directive
- Define constants and macros in a separate file, and have the precompiler read this file using the #include directive

### 5.4.1 How the Pro\*C/C++ Preprocessor Works

The Pro\*C/C++ preprocessor recognizes most C preprocessor commands, and effectively performs the required macro substitutions, file inclusions, and conditional source text inclusions or exclusions. The Pro\*C/C++ preprocessor uses the values obtained from preprocessing, and alters the source output text (the generated .c output file).

An example should clarify this point. Consider the following program fragment:

Suppose the file  $my\_header.h$  in the current directory contains, among other things, the line

#define VC LEN 20

The precompiler reads the file my\_header.h, and uses the defined value of VC\_LEN (20), declares the structure of *name* as VARCHAR[20].

**char** is a native type. The precompiler does not substitute 20 in the declaration of another\_name[VC\_LEN].

This does not matter, since the precompiler does not need to process declarations of C datatypes, even when they are used as host variables. It is left up to the C compiler's preprocessor to actually include the file <code>my\_header.h</code>, and perform the substitution of 20 for VC\_LEN in the declaration of <code>another\_name</code>.

### 5.4.2 Preprocessor Directives

The preprocessor directives that Pro\*C/C++ supports are:

- #define, to create macros for use by the precompiler and the C or C++ compiler
- #include, to read other source files for use by the precompiler
- #if, to precompile and compile source text based on evaluation of a constant expression to
- #ifdef, to precompile and compile source text conditionally, depending on the existence of a defined constant
- #ifndef, to exclude source text conditionally
- #endif, to end an #if or #ifdef or #ifndef command
- #else, to select an alternative body of source text to be precompiled and compiled, in case an #if or #ifdef or #ifndef condition is not satisfied
- #elif, to select an alternative body of source text to be precompiled and compiled, depending on the value of a constant or a macro argument

### 5.4.2.1 Directives Ignored

Some C preprocessor directives are not used by the Pro\*C/C++ preprocessor. Most of these directives are not relevant for the precompiler. For example, **#pragma** is a directive for the C compiler—the precompiler does not process it. The C preprocessor directives not processed by the precompiler are:

- #, to convert a preprocessor macro parameter to a string constant
- ##, to merge two preprocessor tokens in a macro definition
- #error, to produce a compile-time error message
- #pragma, to pass implementation-dependent information to the C compiler
- #line, to supply a line number for C compiler messages

While your C compiler preprocessor may support these directives, Pro\*C/C++ does not use them. Most of these directives are not used by the precompiler. You can use these directives in your Pro\*C/C++ program if your compiler supports them, but only in C or C++ code, not in embedded SQL statements or declarations of variables using datatypes supplied by the precompiler, such as VARCHAR.



## 5.4.3 ORA\_PROC Macro

Pro\*C/C++ predefines a C preprocessor macro called ORA\_PROC that you can use to avoid having the precompiler process unnecessary or irrelevant sections of code. Some applications include large header files, which provide information that is unnecessary when precompiling. By conditionally excluding such header files based on the ORA\_PROC macro, the precompiler never reads the file.

The following example uses the ORA PROC macro to exclude the irrelevant.h file:

```
#ifndef ORA_PROC
#include <irrelevant.h>
#endif
```

Because ORA\_PROC is defined during precompilation, the irrelevant.h file is never included.

The ORA\_PROC macro is available only for C preprocessor directives, such as **#ifdef** or **#ifndef**. The EXEC ORACLE conditional statements do *not* share the same namespaces as the C preprocessor macros. Therefore, the condition in the following example does *not* use the predefined ORA\_PROC macro:

```
EXEC ORACLE IFNDEF ORA_PROC;
    <section of code to be ignored>
EXEC ORACLE ENDIF;
```

ORA\_PROC, in this case, must be set using either the DEFINE option or an EXEC ORACLE DEFINE statement for this conditional code fragment to work properly.

## 5.4.4 Location of Header File Specification

The Pro\*C/C++ Precompiler for each system assumes a standard location for header files to be read by the preprocessor, such as sqlca.h, oraca.h, and sqlda.h. For example, on most UNIX systems, the standard location is \$ORACLE\_HOME/precomp/public. For the default location on your system, see your system-specific Oracle documentation. If header files that you need to include are not in the default location, you must use the INCLUDE= option, on the command line or as an EXEC ORACLE option.

To specify the location of system header files, such as stdio.h or iostream.h, where the location might be different from that hard-coded into Pro\*C/C++ use the SYS\_INCLUDE precompiler option.

#### **Related Topics**

Precompiler Options

### 5.4.5 Some Preprocessor Examples

You can use the **#define** command to create named constants, and use them in place of "magic numbers" in your source code. You can use **#define**d constants for declarations that the precompiler requires, such as VARCHAR[const]. For example, instead of code with bugs, such as:

```
VARCHAR emp_name[10];
VARCHAR dept_loc[14];
```



You can use preprocessor macros with arguments for objects that the precompiler must process, just as you can for C objects. For example:

```
#define ENAME_LEN 10
#define LOCATION_LEN 14
#define MAX(A,B) ((A) > (B) ? (A) : (B))

...
f43()
{
    /* need to declare a temporary variable to hold either an
        employee name or a department location */
    VARCHAR name_loc_temp[MAX(ENAME_LEN, LOCATION_LEN)];
    ...
}
```

You can use the **#include**, **#ifdef** and **#endif** preprocessor directives to conditionally include a file that the precompiler requires. For example:

```
#ifdef ORACLE_MODE
#include <sqlca.h>
#else
    int SQLCODE;
#endif
```

### 5.4.5.1 About Using #define

There are restrictions on the use of the **#define** preprocessor directive in Pro\*C/C++ You cannot use the **#define** directive to create symbolic constants for use in *executable* SQL statements. The following *invalid* example demonstrates this:

```
#define RESEARCH_DEPT 40
...
EXEC SQL SELECT empno, sal
    INTO :emp_number, :salary /* host arrays */
    FROM emp
    WHERE deptno = RESEARCH DEPT; /* INVALID! */
```

The only declarative SQL statements where you can legally use a #defined macro are TYPE and VAR statements. So, for example, the following uses of a macro are legal in Pro\*C/C++

### 5.4.5.2 Other Preprocessor Restrictions

The preprocessor ignores directives # and ## to create tokens that the precompiler must recognize. You can use these commands (if your compiler supports them) in pure C code that the precompiler does not have to process. Using the preprocessor command ## is *not* valid in this example:

```
#define MAKE_COL_NAME(A) col ## A
...
EXEC SQL SELECT MAKE_COL_NAME(1), MAKE_COL_NAME(2)
    INTO :x, :y
    FROM table1;
```

The example is incorrect because the precompiler ignores ##.

## 5.4.6 SQL Statements Not Allowed in #include

Because of the way the Pro\*C/C++ preprocessor handles the **#include** directive, as described in the previous section, you cannot use the **#include** directive to include files that contain embedded SQL statements. You use **#include** to include files that contain purely declarative statements and directives; for example, **#defines**, and declarations of variables and structures required by the precompiler, such as in sqlca.h.

## 5.4.7 Include the SQLCA, ORACA, and SQLDA

You can include the sqlca.h, oraca.h, and sqlda.h declaration header files in your Pro\*C/C++ program using either the C/C++ preprocessor #include command, or the precompiler EXEC SQL INCLUDE command. For example, you use the following statement to include the SQL Communications Area structure (SQLCA) in your program with the EXEC SQL option:

```
EXEC SQL INCLUDE sqlca;
```

To include the SQLCA using the C/C++ preprocessor directive, add the following code:

```
#include <sqlca.h>
```

When you use the preprocessor **#include** directive, you must specify the file extension (such as .h).



#### Note:

If you need to include the SQLCA in multiple places, using the **#include** directive, you should precede the **#include** with the directive **#undef** SQLCA. This is because sqlca.h starts with the lines

```
#ifn lef SQLCA
#define SQLCA 1
```

and then declares the SQLCA struct only in the case that SQLCA is not defined.

When you precompile a file that contains a **#include** directive or an EXEC SQL INCLUDE statement, you have to tell the precompiler the location of all files to be included. You can use the INCLUDE= option, either in the command line, or in the system configuration file, or in the user configuration file.

The default location for standard preprocessor header files, such as sqlca.h, oraca.h, and sqlda.h, is preset in the precompiler. The location varies from system to system. See your system-specific Oracle documentation for the default location on your system.

When you compile the .c output file that Pro\*C/C++ generates, you must use the option provided by your compiler and operating system to identify the location of included files.

For example, on most UNIX systems, you can compile the generated C source file using the command

```
cc -o progname -I$ORACLE HOME/sqllib/public ... filename.c ...
```

On VAX/OPENVMS systems, you pre-pend the include directory path to the value in the logical VAXC\$INCLUDE.

#### **Related Topics**

- Handling Runtime Errors
- Precompiler Options

# 5.4.8 EXEC SQL INCLUDE and #include Summary

When you use an EXEC SQL INCLUDE statement in your program, the precompiler includes the source text in the output (.c) file. Therefore, you can have declarative and executable embedded SQL statements in a file that is included using EXEC SQL INCLUDE.

When you include a file using **#include**, the precompiler merely reads the file, and keeps track of **#define**d macros.



VARCHAR declarations and SQL statements are *not* allowed in included (#include) files. For this reason, you cannot use SQL statements in files included using the Pro\*C/C++ preprocessor #include directive.



### 5.4.9 Defined Macros

If you define macros on the C compiler's command line, you might also have to define these macros on the precompiler command line, depending on the requirements of your application. For example, if you compile with a UNIX command line such as

```
cc -DDEBUG ...
```

you should precompile using the DEFINE= option, namely

```
proc DEFINE=DEBUG ...
```

### 5.4.10 Include Files

The location of all included files that need to be precompiled must be specified on the command line, or in a configuration file.

For example, if you are developing under UNIX, and your application includes files in the directory /home/project42/include, you must specify this directory both on the Pro\*C/C++ command line and on the cc command line. You use commands like these:

```
proc iname=my_app.pc include=/home/project42/include ...
cc -I/home/project42/include ... my app.c
```

or you include the appropriate macros in a *makefile*. For complete information about compiling and linking your Pro\*C/C++ application, see your system-specific Oracle documentation.

#### **Related Topics**

INCLUDE

# 5.5 Precompiled Header Files

Precompiled header files save time and resources by precompiling header files that contain many #include statements. The two steps in using this feature are:

- The precompiled header file is created first,
- The precompiled header is then automatically used in subsequent precompilation of your application.

Use this capability with large applications that have many modules.

The precompiler option, HEADER=hdr, specifies

- That precompiled headers are to be used,
- That the file extension for the output file to be generated is hdr.

This option can only be entered in a configuration file or on the command line. There is no default value for HEADER, but the input header must have a .h extension.

## 5.5.1 Precompiled Header File Creation

Assume that you have a header file called top.h. Then you can precompile it, specifying that HEADER=hdr:

```
proc HEADER=hdr INAME=top.h
```





You must provide the '.h' extension. You cannot use an absolute path element or relative path elements such as '/', '..', and so on., in the INAME value.

Pro\*C/C++ precompiles the given input file, top.h, and generates a new precompiled header file, top.hdr, in the same directory. The output file, top.hdr, can be moved to a directory that the #include statement will cause to be searched.



Do not use the ONAME option to name the output file; it is ignored when used with HEADER.

# 5.5.2 Use of the Precompiled Header Files

Use the same value of the HEADER option with an application file that is to be precompiled. If simple.pc contains:

```
#include <top.h>
```

#### and top.h contains:

```
#include <a.h>
#include <b.h>
#include <c.h>
```

#### then precompile this way:

```
proc HEADER=hdr INAME=simple.pc
```

When Pro\*C/C++ reads the #include top.h statement, it will search for a corresponding 'top.hdr' file and instantiate the data from that file instead of precompiling 'top.h' again.



A precompiled header file will always be used instead of its input header file even if the input (.h) file appears first in the standard search hierarchy of the include directories.

### 5.5.3 Examples

This section includes examples demonstrating several different cases.

### 5.5.3.1 Redundant File Inclusion

The following two cases illustrate two possibilities for redundant file inclusion.

### 5.5.3.1.1 Case 1: Top-Level Header File Inclusion

A precompiled header file will only be instantiated once regardless of how many times the file is included using a #include directive.

Suppose we precompile a top-level header file, top.h, with the value of HEADER set to 'hdr' as before. Next, we code multiple #include directives for that header file in a program:

```
#include <top.h>
#include <top.h>
main(){}
```

When the first #include for top.h is encountered, the precompiled header file, top.hdr, will be instantiated. The second inclusion of that same header file will be redundant and thus, will be ignored.

#### 5.5.3.1.2 Case 2: Nested Header File Inclusion

Suppose the file a.h contains the following statement:

```
#include <b.h>
```

and that we precompile that header file specifying HEADER as before. Pro\*C/C++ will precompile both a.h and b.h generating a.hdr as a result.

Now suppose we precompile this Pro\*C/C++ program:

```
#include <a.h>
#include <b.h>
main(){}
```

When the #include for a.h is encountered, the a.hdr precompiled header file will be instantiated instead of precompiling a.h again. This instantiation will also contain the entire contents of b.h.

Now, because b.h was included in the precompilation of a.h, and a.hdr was instantiated, the subsequent #include of b.h in our program is redundant and thus, will be ignored.

### 5.5.3.2 Multiple Precompiled Header Files

Pro\*C/C++ is capable of instantiating more than one different precompiled header file in a single precompilation. However, one pitfall to avoid occurs when two or more precompiled header files share common header files.

For example, suppose topA.h contains the following lines:

```
#include <a.h>
#include <c.h>
```

and that topB.h contains the following lines:

```
#include <b.h>
#include <c.h>
```

Notice how topA.h and topB.h both include the same common header file, c.h. Precompiling topA.h and topB.h with the same HEADER value will yield topA.hdr and topB.hdr. Both, however, will contain the entire contents of c.h.

Now suppose we have a Pro\*C/C++ program:

```
#include <topA.h>
#include <topB.h>
main(){}
```

Both precompiled header files, topA.hdr and topB.hdr will be instantiated as before. However, because each shares the common header file, c.h, the contents of that file will be instantiated twice.

Pro\*C/C++ cannot determine when such commonality is occurring among precompiled header files. Try to have each precompiled header file contain a unique set of included headers. Sharing headers should be avoided as much as possible because it will ultimately slow down precompilation and utilize more memory, thus undermining the basic intent of using precompiled header files.

### 5.5.4 List of Header Files

The <code>ORACLE\_BASE\ORACLE\_HOME\precomp\public</code> directory contains the Pro\*C/C++ header files. Table 5-3 lists and describes the header files.

Table 5-3 Header Files

Header Files	Description	
oraca.h	Contains the Oracle Communications Area (ORACA), which helps you to diagnose runtime errors and to monitor your program's use of various Oracle Database 10 <i>g</i> resources.	
sql2oci.h	Contains SQLLIB functions that enable the Oracle Call Interface (OCI) environment handle and OCI service context to be obtained in a Pro*C/C++ application.	
sqlapr.h	Contains ANSI prototypes for externalized functions that can be used in conjunction with OCI.	
sqlca.h	Contains the SQL Communications Area (SQLCA), which helps you to diagnose runtime errors. The SQLCA is updated after every executable SQL statement.	
sqlcpr.h	Contains platform-specific ANSI prototypes for SQLLIB functions that are generated by Pro*C/C++. By default, Pro*C/C++ does not support full-function prototyping of SQL programming calls. If you need this feature, include sqlcpr.h before any EXEC SQL statements in your application source file.	
oraca.h	Contains the Oracle Communications Area (ORACA), which helps you to diagnose runtime errors and to monitor your program's use of various Oracle Database 10 <i>g</i> resources.	
sql2oci.h	Contains SQLLIB functions that enable the Oracle Call Interface (OCI) environment handle and OCI service context to be obtained in a Pro*C/C++ application.	
sqlapr.h	Contains ANSI prototypes for externalized functions that can be used in conjunction with OCI.	

# 5.5.5 Effects of Options

The following precompiler options are used with the precompilation of the application.

### 5.5.5.1 DEFINE and INCLUDE Options

During any precompilation using precompiled headers, you must use the same values for DEFINE and INCLUDE as when you created the precompiled header files. If the values of DEFINE or INCLUDE change, you must re-create the precompiled header files.

If development environments change, you must also re-create the precompiled header files.

### 5.5.5.1.1 Single User Scenario

Consider a single user. If the values of either the DEFINE or the INCLUDE options were to change, then the contents of the precompiled header files may no longer be suitable for use in subsequent Pro\*C/C++ precompilations.

Because the values of the DEFINE and INCLUDE; DEFINE or INCLUDE options have changed, the contents of the precompiled header file may no longer be consistent with what a standard precompilation would result in had the corresponding .h file in the #include directive been processed normally.

In short, if the values of the DEFINE and INCLUDE; DEFINE or INCLUDE options change, any precompiled header files must be re-created and Pro\*C/C++ programs which use them reprecompiled.

#### **Related Topics**

- DEFINE
- INCLUDE

### 5.5.5.1.2 Multiple User Scenario

Consider two users, A and B, who develop in totally separate environments, thus having completely different values for their DEFINE and INCLUDE options.

User A precompiles a common header file, common.h, creating a precompiled header file common.hdrA. User B also precompiles the same header file creating common.hdrB. However, given that the two environments are different, specifically with respect to the values of the DEFINE and INCLUDE options used by both users, it is not guaranteed that both user A's and B's versions of common.hdr will be the same.

#### To summarize

```
A> proc HEADER=hdrA DEFINE=<A macros> INCLUDE=<A dirs> common.h
B> proc HEADER=hdrB DEFINE=<B macros> INCLUDE=<B dirs> common.h
```

The generated precompiled header files common.hdrA may not equal common.hdrB because of the different environments in which they where created. This means that neither user A nor user B would be guaranteed that using the common.hdr created by the other user would result in correct precompilation of the Pro\*C/C++ programs in their respective development environments.

Therefore, care should be taken when sharing or exchanging precompiled header files between different users and different users' development environments.

## 5.5.5.2 CODE and PARSE Options

Pro\*C/C++ does not search for C++ header files with extensions such as hpp or h++. So do not use CODE=CPP when precompiling header files. You may use the CPP value when precompiling the application, as long as the source code only includes .h header files.

You can only use the values FULL or PARTIAL for the option PARSE when creating the precompiled header files, or when precompiling the modules. The value FULL is considered to be of higher value than PARTIAL. The value of PARSE used should be the same or lower when precompiling modules as when you created the precompiled header files.





Precompiling the precompiled header file with PARSE=FULL and then precompiling modules with PARSE=PARTIAL requires that the host variables be declared inside a Declare Section. C++ code will only be understood when PARSE=PARTIAL.

Suppose we precompile a header file with PARSE set to PARTIAL as follows:

```
proc HEADER=hdr PARSE=PARTIAL file.h
```

and then try to precompile a program that includes that header file using PARSE set to FULL:

```
proc HEADER=hdr PARSE=FULL program.pc
```

Because file.h was precompiled using a PARTIAL setting for the PARSE option, not all of the header file would have been processed. It would therefore be possible for an error to occur during the precompilation of the Pro\*C/C++ program if a reference was made to something in the unprocessed portion.

To illustrate, suppose that file.h contained the following code:

```
#define LENGTH 10
typedef int myint;
```

and that our program.pc contained the following short program:

```
#include <file.h>
main()
{
    VARCHAR ename[LENGTH];
    myint empno = ...;
    EXEC SQL SELECT ename INTO :ename WHERE JOB = :empno;
}
```

Because PARSE was set to PARTIAL when precompiling file.h, only the LENGTH macro would have been processed leaving the typedef unseen.

The VARCHAR declaration and subsequent use as a host variable would succeed. However, the use of the empno host variable would not because the myint type declaration would never have been processed by Pro\*C/C++.

Precompiling the header file with the PARSE option set to FULL and then precompiling the program with PARSE set to PARTIAL would work. However, the host variables would have to be declared inside an explicit DECLARE SECTION.

#### **Related Topics**

- About Parsing Code
- CODE
- PARSE

### 5.5.6 Usage Notes

The file format of the generated output file of a precompiled header is not guaranteed to remain fixed from one release to the next. Pro\*C/C++ has no way of determining which version of the precompiler was used to generate the precompiled header file output.

Because of this, it is strongly recommended that, in order to avoid the possibility of errors or other strange behavior during a precompilation that uses precompiled header files, those files be regenerated by re-precompiling the corresponding header files when upgrading to newer releases of Pro\*C/C++.

The generated output from the precompilation of a header file is completely non-portable. This means that you cannot transfer the output file from the precompilation of a header file from one platform to another and use that file during the subsequent precompilation of another header file or Pro\*C/C++ program.

# 5.6 The Oracle Preprocessor

Conditional sections of code are marked by EXEC ORACLE directives that define the environment and actions to take. You can code C statements as well as embedded SQL statements and directives in these sections. The following EXEC ORACLE directives let you exercise conditional control over precompilation:

```
EXEC ORACLE DEFINE symbol; -- define a symbol

EXEC ORACLE IFDEF symbol; -- if symbol is defined

EXEC ORACLE IFNDEF symbol; -- if symbol is not defined

EXEC ORACLE ELSE; -- otherwise

EXEC ORACLE ENDIF; -- end this block
```

All EXEC ORACLE statements must be terminated with a semi-colon.

### 5.6.1 Symbol Definition

You can define a symbol in two ways. Either include the statement:

```
EXEC ORACLE DEFINE symbol;
```

in your host program or define the symbol on the command line using the syntax

```
... INAME=filename ... DEFINE=symbol
```

where symbol is not case-sensitive.



The #define preprocessor directive is not the same as the EXEC ORACLE DEFINE command.

Some port-specific symbols are predefined for you when the Pro\*C/C++ precompiler is installed on your system.

### 5.6.2 An Oracle Preprocessor Example

In the following example, the SELECT statement is precompiled only when the symbol *site2* is defined:

```
EXEC ORACLE IFDEF site2;

EXEC SQL SELECT DNAME

INTO :dept_name

FROM DEPT
```



```
WHERE DEPTNO = :dept_number;
EXEC ORACLE ENDIF;
```

Blocks of conditions can be nested as shown in the following example:

```
EXEC ORACLE IFDEF outer;

EXEC ORACLE IFDEF inner;

EXEC ORACLE ENDIF;

EXEC ORACLE ENDIF;
```

You can "Comment out" C or embedded SQL code by placing it between IFDEF and ENDIF and *not* defining the symbol.

### 5.7 Evaluation of Numeric Constants

Previously, Pro\*C/C++ allowed only numeric literals and simple constant expressions involving numeric literals to be used when declaring the sizes of host variables (such as char or VARCHAR), as in the following examples:

```
#define LENGTH 10
VARCHAR v[LENGTH];
char c[LENGTH + 1];
```

You can now also use numeric constant declarations such as:

```
const int length = 10;
VARCHAR v[length];
char c[length + 1];
```

This is highly desirable, especially for developers who use ANSI or C++ compilers that support such constant declarations.

Pro\*C/C++ has always determined the values of constant expressions that can be evaluated, but it has never allowed the use of a numeric constant declaration in any constant expression.

Pro\*C/C++ supports the use of numeric constant declarations anywhere that an ordinary numeric literal or macro is used, given the macro expands to some numeric literal.

This is used primarily for declaring the sizes of arrays for bind variables to be used in a SQL statement.

### 5.7.1 Numeric Constants in Pro\*C/C++

In Pro\*C/C++, normal C scoping rules are used to find and locate the declaration of a numeric constant declaration.

## 5.7.2 Numeric Constant Rules and Examples

Variables which are of specific static types need to be defined with **static** and initialized. The following rules must be kept in mind when declaring numeric constants in Pro\*C/C++:

- The const qualifier must be used when declaring the constant
- An initializer must be used to initialize the value of the constant. This initializer must be precompile-time evaluable.

Any attempt to use an identifier that does not resolve to a constant declaration with a valid initializer is considered an error.

The following shows examples of what is not permitted and why:

```
int a;
int b = 10;
volatile c;
volatile d = 10;
const e;
const f = b;

VARCHAR v1[a]; /* No const qualifier, missing initializer */
VARCHAR v2[b]; /* No const qualifier */
VARCHAR v3[c]; /* Not a constant, missing initializer */
VARCHAR v4[d]; /* Not a constant */
VARCHAR v5[e]; /* Bad initializer.. b is not a constant */
```

# 5.8 SQLLIB Extensions for OCI Release 8 Interoperability

An OCI environment handle will be tied to the Pro\*C/C++ runtime context, which is of the *sql\_context* type. That is, one Pro\*C/C++ runtime context maintained by SQLLIB during application execution will be associated with at most one OCI environment handle. Multiple database connections are allowed for each Pro\*C/C++ runtime context, which will be associated to the OCI environment handle for the runtime context.

### Note:

Precompiler applications can extract OCI handles and call OCI functions directly. However, non-blocking mode is not supported because the precompilers are unable to handle the "still executing" error that might be returned.

### 5.8.1 Runtime Context in the OCI Release 8 Environment

An EXEC SQL CONTEXT USE statement specifies a runtime context to be used in a Pro\*C/C+ + program. This context applies to all executable SQL statements that positionally follow it in a

given Pro\*C/C++ file until another EXEC SQL CONTEXT USE statement occurs. If no EXEC SQL CONTEXT USE appears in a source file, the default "global" context is assumed. Thus, the current runtime context, and therefore the current OCI environment handle, is known at any point in the program.

The runtime context and its associated OCI environment handle are initialized when a database logon is performed using EXEC SQL CONNECT in Pro\*C/C++.

When a Pro\*C/C++ runtime context is freed using the EXEC SQL CONTEXT FREE statement, the associated OCI environment handle is terminated and all of its resources, such as space allocated for the various OCI handles and LOB locators, are de-allocated. This command releases all other memory associated with the Pro\*C/C++ runtime context. An OCI environment handle that is established for the default "global" runtime remains allocated until the Pro\*C/C++ program terminates.

### 5.8.2 Parameters in the OCI Release 8 Environment Handle

An OCI environment established through Pro\*C/C++ will use the following parameters:

- The callback functions used by the environment for allocating memory, freeing memory, writing to a text file, and flushing the output buffer will be trivial functions that call malloc(), free(), fprintf(stderr, ...), and fflush(stderr) respectively.
- The language will be obtained from the Globalization Support variable NLS LANG.
- The error message buffer will be allocated in thread-specific storage.

## 5.9 Interface to OCI Release 8

SQLLIB library provides routines to obtain the OCI environment and service context handles for database connections established through a Pro\*C/C++ program. Once the OCI handles are obtained, the user can call various OCI routines, for example, to perform client-side DATE arithmetic, execute navigational operations on objects and so on. These SQLLIB functions are described later, and their prototypes are available in the public header file sql2oci.h.

A Pro\*C/C++ user who mixes embedded SQL and calls in the other Oracle programmatic interfaces must exercise reasonable care. For example, if a user terminates a connection directly using the OCI interface, SQLLIB state is out-of-sync; the behavior for subsequent SQL statements in the Pro\*C/C++ program is undefined in such cases.



Pro\*C/C++, the Oracle Call Interface (OCI) release 8, and XA are *not* compatible.

The new SQLLIB functions that provide interoperability with the Oracle OCI are declared in header file sql2oci.h:

- SQLEnvGet (), to return a pointer to an OCI environment handle associated with a given SQLLIB runtime context. Used for both single and shared server environments.
- SQLSvcCtxGet(), to return an OCI service context handle for a Pro\*C/C++ database connection. Used for both single and shared server environments.
- Pass the constant SQL\_SINGLE\_RCTX, defined as (dvoid \*) 0, when you include sql2oci.h,
  as the first parameter in either function, when using single threaded runtime contexts.

#### **Related Topics**

Objects

# 5.9.1 SQLEnvGet()

The SQLLIB library function SQLEnvGet() (SQLLIB OCI Environment Get) returns the pointer to the OCI environment handle associated with a given SQLLIB runtime context. The prototype for this function is:

sword SQLEnvGet(dvoid \*rctx, OCIEnv \*\*oeh);

#### where:

Terms	Description	
Description	Sets <i>oeh</i> to the OCIEnv corresponding to the runtime context	
Parameters	rctx (IN) pointer to a SQLLIB runtime context oeh (OUT) pointer to OCIEnv	
Returns	SQL_SUCCESS on success SQL_ERROR on failure	
Notes	The usual error status variables in Pro*C/C++ such as SQLCA and SQLSTATE will not be affected by a call to this function	

# 5.9.2 SQLSvcCtxGet()

The SQLLIB library function SQLSvcCtxGet() (SQLLIB OCI Service Context Get) returns the OCI service context for the Pro\*C/C++ database connection. The OCI service context can then be used in direct calls to OCI functions. The prototype for this function is:

#### where:

Terms	Description Sets svc to the OCI Service Context corresponding to the runtime context	
Description		
Parameters	rctx (IN) = pointer to a SQLLIB runtime context  dbname (IN) = buffer containing the "logical" name for this connection  dbnamelen (IN) = length of the dbname buffer  svc (OUT) = address of an OCISvcCtx pointer	
Returns	SQL_SUCCESS on success SQL_ERROR on failure	
Notes	The usual error status variables in Pro*C/C++ such as SQLCA and SQLSTATE will not be affected by a call to this function	
	2. dbname is the same identifier used in an AT clause in an embedded SQL statement.	
	3. If <i>dbname</i> is a null pointer or <i>dbnamelen</i> is 0, then the default database connection is assumed, as in a SQL statement with no AT clause.	
	4. A value of -1 for <i>dbnamelen</i> is used to indicate that <i>dbname</i> is a zero-terminated string.	



### 5.9.3 Embedded OCI Release 8 Calls

To embed OCI release 8 calls in your Pro\*C/C++ program:

- 1. Include the public header sql2oci.h
- 2. Declare an environment handle (type OCIEnv \*) in your Pro\*C/C++ program:

```
OCIEnv *oeh;
```

3. Optionally, declare a service context handle (type OCISvcCtx \*) in your Pro\*C/C++ program if the OCI function you wish to call requires the Service Context handle.

```
OCISvcCtx *svc;
```

4. Declare an error handle (type OCIError \*) in your Pro\*C/C++ program:

```
OCIError *err;
```

5. Connect to Oracle using the embedded SQL statement CONNECT. Do not connect using OCI.

```
EXEC SOL CONNECT ...
```

6. Obtain the OCI Environment handle that is associated with the desired runtime context using the SQLEnvGet function.

For single threaded applications:

```
retcode = SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
```

or for shared server applications:

```
sql_context ctx1;
...
EXEC SQL CONTEXT ALLOCATE :ctx1;
EXEC SQL CONTEXT USE :ctx1;
...
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
...
retcode = SQLEnvGet(ctx1, &oeh);
```

7. Allocate an OCI error handle using the retrieved environment handle:

8. Optionally, if needed by the OCI call you use, obtain the OCIServiceContext handle using the SQLSvcCtxGet call:

For single threaded applications:

```
retcode = SQLSvcCtxGet(SQL SINGLE RCTX, (text *)dbname, (ub4)dbnlen, &svc);
```

or, for shared server environment applications:

```
sql_context ctx1;
...
EXEC SQL ALLOCATE :ctx1;
EXEC SQL CONTEXT USE :ctx1;
...
EXEC SQL CONNECT :uid IDENTIFIED BY :pwd AT :dbname
```



```
USING :hst;
...
retcode = SQLSvcCtxGet(ctx1, (text *)dbname, (ub4)strlen(dbname), &svc);
```



A null pointer may be passed as the *dbname* if the Pro\*C/C++ connection is not named with an AT clause.

### 5.10 Embedded OCI Release 7 Calls



The Logon Data Area (LDA) is *no longer supported*. The ability to embed OCI Release 7 calls in your Pro\*C/C++ program is not supported.

To embed OCI release 7 calls in your Pro\*C/C++ program, take the following steps:

- Declare an OCI Logon Data Area (LDA) in your Pro\*C/C++ program (outside the Declare Section if you precompile with MODE=ANSI). The LDA is a structure defined in the OCI header file oci.h. For details, see the Oracle Call Interface Developer's Guide for Release 7.
- Connect to Oracle using the embedded SQL statement CONNECT, not the OCI orlon() or onblon() calls.
- Call the SQLLIB runtime library function sqllda () to set up the LDA.SQLLIB function

That way, the Pro\*C/C++ Precompiler and the OCI "know" that they are working together. However, there is no sharing of Oracle cursors.

You need not worry about declaring the OCI Host Data Area (HDA) because the Oracle runtime library manages connections and maintains the HDA for you.

## 5.10.1 Set Up the LDA

You set up the LDA by issuing the OCI call

```
sqllda(&lda);
```

where Ida identifies the LDA data structure.

If the setup fails, the *lda\_rc* field in the *lda* is set to 1012 to indicate the error.

## 5.10.2 Remote and Multiple Connections

A call to sqllda() sets up an LDA for the connection used by the most recently executed SQL statement. To set up the different LDAs needed for additional connections, you *must* call sqllda() with a different LDA *immediately* after each CONNECT. In the following example, you connect to two nondefault databases concurrently:

```
#include <ocidfn.h>
Lda Def lda1;
Lda Def 1da2;
char username[10], password[10], db_string1[20], dbstring2[20];
strcpy(username, "scott");
strcpy(password, "tiger");
strcpy(db string1, "NYNON");
strcpy(db_string2, "CHINON");
/* give each database connection a unique name */
EXEC SQL DECLARE DB NAME1 DATABASE;
EXEC SQL DECLARE DB NAME2 DATABASE;
/* connect to first nondefault database */
EXEC SQL CONNECT :username IDENTIFIED BY :password;
   AT DB NAME1 USING :db_string1;
/* set up first LDA */
sqllda(&lda1);
/* connect to second nondefault database */
EXEC SQL CONNECT : username IDENTIFIED BY :password;
   AT DB NAME2 USING :db string2;
/* set up second LDA */
sqllda(&lda2);
```

DB\_NAME1 and DB\_NAME2 are *not* C variables; they are SQL identifiers. You use them only to name the default databases at the two nondefault nodes, so that later SQL statements can refer to the databases by name.

# 5.11 New Names for SQLLIB Public Functions

The names of SQLLIB functions are listed in Table 5-4. You can use these SQLLIB functions for both threaded and nonthreaded applications. Previously, for example, sqlglm() was documented as the nonthreaded or default context version of this function, while sqlglmt() was the threaded or nondefault context version, with context as the first argument. The names sqlglm() and sqlglmt() are still available. The new function sqlglmt() requires the same arguments as sqlglmt(). For nonthreaded or default context applications, pass the defined constant SQL SINGLE RCTX as the context.

Each standard SQLLIB public function is thread-safe and accepts the runtime context as the first argument. For example, the syntax for SQLErrorGetText() is:

In summary, the old function names will continue to work in your existing applications. You can use the new function names in the new applications that you will write.

Table 5-4 lists all the SQLLIB public functions and their corresponding syntax. Cross-references to the nonthreaded or default-context usages are provided to help you find more complete descriptions.

Table 5-4 SQLLIB Public Functions -- New Names

Old Name	New Function Prototype	Cross-reference
sqlaldt()	<pre>struct SQLDA *SQLSQLDAAlloc(dvoid *context, unsigned int</pre>	See also "Allocating a SQLDA " .
sqlcdat()	<pre>void SQLCDAFromResultSetCursor(dvoid *context, Cda_Def *cda, void *cursor, sword *return_value);</pre>	See also "Cursor Variables with the OCI (Release 7 Only)".
sqlclut()	<pre>void SQLSQLDAFree(dvoid *context, struct SQLDA *descriptor_name);</pre>	See also"Deallocate Storage " .
sqlcurt()	<pre>void SQLCDAToResultSetCursor(dvoid *context, void *cursor, Cda_Def *cda, sword *return_value)</pre>	See also "Cursor Variables with the OCI (Release 7 Only)" .
sqlglmt()	<pre>void SQLErrorGetText(dvoid *context, unsigned char *message_buffer, size_t *buffer_size, size_t *message_length);</pre>	See also "About Getting the Full Text of Error Messages ".
sqlglst()	<pre>void SQLStmtGetText(dvoid *context,   char *statement_buffer,   size_t *statement_length,   size_t *sqlfc);</pre>	See also "About Obtaining the Text of SQL Statements ".
sqlld2t()	<pre>void SQLLDAGetName(dvoid *context, Lda_Def *lda, text *cname, int *cname_length);</pre>	See also "OCI Calls (Release 7 Only)"
sqlldat()	<pre>void SQLLDAGetCurrent(dvoid *context, Lda_Def *lda);</pre>	See also "Remote and Multiple Connections ".
sqlnult()	<pre>void SQLColumnNullCheck(dvoid *context, unsigned short *value_type, unsigned short *type_code, int</pre>	See also "Handling NULL/Not NULL Datatypes ".

Table 5-4 (Cont.) SQLLIB Public Functions -- New Names

Old Name	New Function Prototype	Cross-reference
sqlprct()	<pre>void SQLNumberPrecV6(dvoid *context, unsigned long *length, int *precision, int *scale);</pre>	See also "Extracting Precision and Scale ".
sqlpr2t()	<pre>void SQLNumberPrecV7(dvoid *context, unsigned long *length, int *precision, int *scale);</pre>	See also "Extracting Precision and Scale ".
sqlvcpt()	<pre>void SQLVarcharGetLength(dvoid *context, unsigned long *data_length, unsigned long *total_length);</pre>	See also "Find the Length of the VARCHAR Array Component ".
N/A	<pre>sword SQLEnvGet(dvoid *context, OCIEnv **oeh);</pre>	See "SQLEnvGet()".
N/A	<pre>sword SQLSvcCtxGet(dvoid *context, text</pre>	See "SQLSvcCtxGet()".
N/A	<pre>void SQLRowidGet(dvoid *context, OCIRowid **urid);</pre>	See "SQLRowidGet()".
N/A	<pre>void SQLExtProcError(dvoid *context, char *msg, size_t msglen);</pre>	See "SQLExtProcError()" for a discussion of its use in external procedures.



For the specific datatypes used in the argument lists for these functions, refer to your platform-specific version of the sqlepr.h header file.

#### **Related Topics**

Interface to OCI Release 8

# 5.12 X/Open Application Development

X/Open applications run in a distributed transaction processing (DTP) environment. In an abstract model, an X/Open application calls on *resource managers* (RMs) to provide a variety of services. For example, a database resource manager provides access to data in a database. Resource managers interact with a *transaction manager* (TM), which controls all transactions for the application.

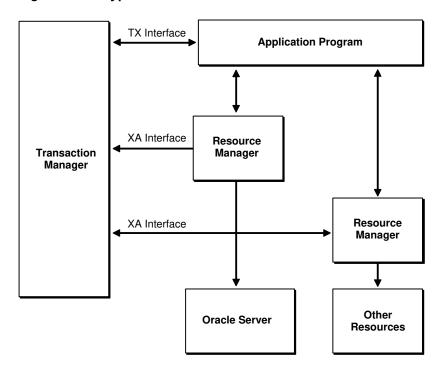


Figure 5-1 Hypothetical DTP Model

Figure 5-1 shows one way that components of the DTP model can interact to provide efficient access to data in an Oracle database. The DTP model specifies the *XA interface* between resource managers and the transaction manager. Oracle supplies an XA-compliant library, which you must link to your X/Open application. Also, you must specify the *native interface* between your application program and the resource managers.

The DTP model that specifies how a transaction manager and resource managers interact with an application program is described in the X/Open guide *Distributed Transaction Processing Reference Model* and related publications, which you can obtain by writing to

The Open Group 1010 El Camino Real, Suite 380, Menlo Park, CA 94025-4345 USA

http://www.opengroup.org/

For instructions on using the XA interface, see your Transaction Processing (TP) Monitor user's guide.

# 5.12.1 Oracle-Specific Issues

You can use the precompiler to develop applications that comply with the X/Open standards. However, you must meet the following requirements.

### 5.12.1.1 Connecting to Oracle

The X/Open application does not establish and maintain connections to a database. Instead, the transaction manager and the XA interface, which is supplied by Oracle, handle database connections and disconnections transparently. So, normally an X/Open-compliant application does not execute CONNECT statements.

### 5.12.1.2 Transaction Control

The X/Open application must not execute statements such as COMMIT, ROLLBACK, SAVEPOINT, and SET TRANSACTION that affect the state of global transactions. For example, the application must not execute the COMMIT statement because the transaction manager handles commits. Also, the application must not execute SQL data definition statements such as CREATE, ALTER, and RENAME because they issue an implicit COMMIT.

The application can execute an internal ROLLBACK statement if it detects an error that prevents further SQL operations. However, this might change in later releases of the XA interface.

### 5.12.1.3 OCI Calls (Release 7 Only)

OCI Calls Release 7 are no longer supported.



The Logon Data Area (LDA) is *no longer supported* in Oracle9*i*. The ability to embed OCI Release 7 calls in your Pro\*C/C++ program will be phased out by the next major Oracle release.

If you want your X/Open application to issue OCI calls, you must use the runtime library routine sqlld2(), which sets up an LDA for a specified connection established through the XA interface. For a description of the sqlld2() call, see the Oracle Call Interface Programmer's Guide for Release 7.

The following OCI calls cannot be issued by an X/Open application: OCOM, OCON, OCOF, ONBLON, ORLON, OLOGOF.

#### **Related Topics**

Interface to OCI Release 8

### 5.12.1.4 Linking

To get XA functionality, you must link the XA library to your X/Open application object modules. For instructions, see your system-specific Oracle documentation.



6

# Embedded SQL

This chapter helps you to understand and apply the basic techniques of embedded SQL programming. This chapter contains the following topics:

- Host Variables
- Indicator Variables
- The Basic SQL Statements
- Using Boolean Data Type
- The DML Returning Clause
- Cursors
- Scrollable Cursors
- Optimizer Hints
- Fix Execution Plan
- · The CURRENT OF Clause
- The Cursor Statements
- A Complete Example Using Non-Scrollable Cursor
- A Complete Example Using Scrollable Cursor
- Positioned Update
- Explicit Request Boundaries for Handling Planned Maintenance

## 6.1 Host Variables

Oracle uses host variables to pass data and status information to your program; your program uses host variables to pass data to Oracle.

### 6.1.1 Output versus Input Host Variables

Depending on how they are used, host variables are called output or input host variables.

Host variables in the INTO clause of a SELECT or FETCH statement are called *output* host variables because they hold column values output by Oracle. Oracle assigns the column values to corresponding output host variables in the INTO clause.

All other host variables in a SQL statement are called *input* host variables because your program inputs their values to Oracle. For example, you use input host variables in the VALUES clause of an INSERT statement and in the SET clause of an UPDATE statement. They are also used in the WHERE, HAVING, and FOR clauses. Input host variables can appear in a SQL statement wherever a value or expression is allowed.

#### Note:

In an ORDER BY clause, you *can* use a host variable, but it is treated as a constant or literal, and hence the contents of the host variable have no effect. For example, the SQL statement

```
EXEC SQL SELECT ename, empno INTO :name,:number FROM emp ORDER BY :ord;
```

appears to contain an input host variable *:ord*. However, the host variable in this case is treated as a constant, and regardless of the value of *:ord*, no ordering is done.

You cannot use input host variables to supply SQL keywords or the names of database objects. Thus, you cannot use input host variables in data definition statements such as ALTER, CREATE, and DROP. In the following example, the DROP TABLE statement is *invalid*:

```
char table_name[30];
printf("Table name? ");
gets(table_name);

EXEC SQL DROP TABLE :table name; -- host variable not allowed
```

If you need to change database object names at runtime, use dynamic SQL.

Before Oracle executes a SQL statement containing input host variables, your program must assign values to them. An example follows:

Notice that the input host variables in the VALUES clause of the INSERT statement are prefixed with colons.

#### **Related Topics**

Oracle Dynamic SQL

## 6.2 Indicator Variables

You can associate any host variable with an optional indicator variable. Each time the host variable is used in a SQL statement, a result code is stored in its associated indicator variable. Thus, indicator variables let you monitor host variables.



You cannot use multiple indicator variables with a single host variable within PL/SQL blocks. Doing so results in a "not all variables bound" error.

You use indicator variables in the VALUES or SET clauses to assign NULLs to input host variables. Use indicator variables in the INTO clause to detect NULLs or truncated values in output host variables.

#### On Input

The values your program can assign to an indicator variable have the following meanings:

Variable	Description
-1	Oracle will assign a NULL to the column, ignoring the value of the host variable.
>=0	Oracle will assign the value of the host variable to the column.

#### **On Output**

The values Oracle can assign to an indicator variable have the following meanings:

Variable	Description
-1	The column value is NULL, so the value of the host variable is indeterminate.
0	Oracle assigned an intact column value to the host variable.
>0	Oracle assigned a truncated column value to the host variable. The integer returned by the indicator variable is the original length of the column value, and SQLCODE in SQLCA is set to zero.
-2	Oracle assigned a truncated column variable to the host variable, but the original column value could not be determined (a LONG column, for example).

Remember, an indicator variable must be defined as a 2-byte integer and, in SQL statements, must be prefixed with a colon and must immediately follow its host variable.

## 6.2.1 Insert NULLs

You can use indicator variables to INSERT NULLs. Before the INSERT, for each column you want to be NULL, set the appropriate indicator variable to -1, as shown in the following example:

```
set ind_comm = -1;

EXEC SQL INSERT INTO emp (empno, comm)
     VALUES (:emp_number, :commission:ind_comm);
```

The indicator variable ind comm specifies that a NULL is to be stored in the COMM column.

You can hard code the NULL instead, as follows:

```
EXEC SQL INSERT INTO emp (empno, comm)
    VALUES (:emp number, NULL);
```



While this is less flexible, it might be more readable. Typically, you insert NULLs conditionally, as the next example shows:

```
printf("Enter employee number or 0 if not available: ");
scanf("%d", &emp_number);

if (emp_number == 0)
    ind_empnum = -1;
else
    ind_empnum = 0;

EXEC SQL INSERT INTO emp (empno, sal)
    VALUES (:emp number:ind empnum, :salary);
```

## 6.2.2 Returned NULLs

You can also use indicator variables to manipulate returned NULLs, as the following example shows:

```
EXEC SQL SELECT ename, sal, comm
   INTO :emp_name, :salary, :commission:ind_comm
   FROM emp
   WHERE empno = :emp_number;
if (ind_comm == -1)
   pay = salary;   /* commission is NULL; ignore it */
else
   pay = salary + commission;
```

## 6.2.3 Fetch NULLs

When DBMS=V7 or DBMS=V8, if you SELECT or FETCH NULLs into a host variable not associated with an indicator variable, Oracle issues the following error message:

```
ORA-01405: fetched column value is NULL
```

#### **Related Topics**

DBMS

## 6.2.4 Test for NULLs

You can use indicator variables in the WHERE clause to test for NULLs, as the following example shows:

```
EXEC SQL SELECT ename, sal
INTO :emp_name, :salary
FROM emp
WHERE :commission INDICATOR :ind comm IS NULL ...
```

However, you cannot use a relational operator to compare NULLs with each other or with other values. For example, the following SELECT statement fails if the COMM column contains one or more NULLs:

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

The next example shows how to compare values for equality when some of them might be NULLs:

## 6.2.5 Truncated Values

When DBMS=V7 or V8, if you SELECT or FETCH a truncated column value into a host variable not associated with an indicator variable, a warning is generated instead of an error.

## 6.3 The Basic SQL Statements

Executable SQL statements let you query, manipulate, and control Oracle data and create, define, and maintain Oracle objects such as tables, views, and indexes. This chapter focuses on the statements that guery and manipulate data.

When executing a data manipulation statement such as INSERT, UPDATE, or DELETE, your only concern, besides setting the values of any input host variables, is whether the statement succeeds or fails. To find out, you simply check the SQLCA. (Executing any SQL statement sets the SQLCA variables.) You can check in the following two ways:

- Implicit checking with the WHENEVER statement
- Explicit checking of SQLCA variables

When executing a SELECT statement (query), however, you must also deal with the rows of data it returns. Queries can be classified as follows:

- Queries that return no rows (that is, merely check for existence)
- · Queries that return only one row
- Oueries that return more than one row

Queries that return more than one row require explicitly declared cursors or the use of *host arrays* (host variables declared as arrays).



Host arrays let you process "batches" of rows.

This chapter assumes the use of scalar host variables.

The following embedded SQL statements let you query and manipulate Oracle data:

Embedded SQL Statements	Description
SELECT	Returns rows from one or more tables.
INSERT	Adds new rows to a table.
UPDATE	Modifies rows in a table.
DELETE	Removes unwanted rows from a table.

The following embedded SQL statements let you define and manipulate an explicit cursor:

Embedded SQL Statements	Description
DECLARE	Names the cursor and associates it with a query.
OPEN	Executes the query and identifies the active set.
FETCH	Advances the cursor and retrieves each row in the active set, one by one.
CLOSE	Disables the cursor (the active set becomes undefined).

The following sections, you first learn how to code INSERT, UPDATE, DELETE, and single-row SELECT statements. Then, you progress to multirow SELECT statements.

#### **Related Topics**

- Handling Runtime Errors
- Host Arrays
- Embedded SQL Statements and Directives
- Oracle Database SQL Language Reference

## 6.3.1 The SELECT Statement

Querying the database is a common SQL operation. To issue a query you use the SELECT statement. In the following example, you query the EMP table:

```
EXEC SQL SELECT ename, job, sal + 2000
INTO :emp_name, :job_title, :salary
FROM emp
WHERE empno = :emp number;
```

The column names and expressions following the keyword SELECT make up the *select list*. The select list in our example contains three items. Under the conditions specified in the WHERE clause (and following clauses, if present), Oracle returns column values to the host variables in the INTO clause.

The number of items in the select list should equal the number of host variables in the INTO clause, so there is a place to store every returned value.

In the simplest case, when a query returns one row, its form is that shown in the last example. However, if a query can return more than one row, you must FETCH the rows using a cursor or SELECT them into a host-variable array. Cursors and the FETCH statement are discussed later in this chapter.

If a query is written to return only one row but might actually return several rows, the result of the SELECT is indeterminate. Whether this causes an error depends on how you specify the SELECT\_ERROR option. The default value, YES, generates an error if more than one row is returned.

#### **Related Topics**

Host Arrays

## 6.3.1.1 Available Clauses

You can use all of the following standard SQL clauses in your

SELECT statements:



- INTO
- FROM
- WHERE
- CONNECT BY
- START WITH
- GROUP BY
- HAVING
- ORDER BY
- FOR UPDATE OF

Except for the INTO clause, the text of embedded SELECT statements can be executed and tested interactively using SQL\*Plus. In SQL\*Plus, you use substitution variables or constants instead of input host variables.

#### 6.3.2 The INSERT Statement

Use the INSERT statement to add rows to a table or view. In the following example, you add a row to the EMP table:

```
EXEC SQL INSERT INTO emp (empno, ename, sal, deptno) VALUES (:emp number, :emp name, :salary, :dept number);
```

Each column you specify in the *column list* must belong to the table named in the INTO clause. The VALUES clause specifies the row of values to be inserted. The values can be those of constants, host variables, SQL expressions, SQL functions such as USER and SYSDATE, or user-defined PL/SQL functions.

The number of values in the VALUES clause must equal the number of names in the column list. However, you can omit the column list if the VALUES clause contains a value for each column in the table, in the order that they are defined in the table.

#### **Related Topics**

• INSERT (Executable Embedded SQL)

## 6.3.2.1 About Using Subgueries

A *subquery* is a nested SELECT statement. Subqueries let you conduct multipart searches. They can be used to

- Supply values for comparison in the WHERE, HAVING, and START WITH clauses of SELECT, UPDATE, and DELETE statements
- Define the set of rows to be inserted by a CREATE TABLE or INSERT statement
- Define values for the SET clause of an UPDATE statement

The following example uses a subquery in an INSERT statement to copy rows from one table to another:

```
EXEC SQL INSERT INTO emp2 (empno, ename, sal, deptno)
SELECT empno, ename, sal, deptno FROM emp
WHERE job= :job title;
```

This INSERT statement uses the subquery to obtain intermediate results.

## 6.3.3 The UPDATE Statement

Use the UPDATE statement to change the values of specified columns in a table or view. In the following example, we update the SAL and COMM columns in the EMP table:

```
EXEC SQL UPDATE emp
SET sal = :salary, comm = :commission
WHERE empno = :emp number;
```

Use the optional WHERE clause to specify the conditions under which rows are updated.

The SET clause lists the names of one or more columns for which you must provide values. You can use a subquery to provide the values, as the following example shows:

```
EXEC SQL UPDATE emp
SET sal = (SELECT AVG(sal)*1.1 FROM emp WHERE deptno = 20)
WHERE empno = :emp_number;
```

The UPDATE statement has an optional returning clause, like the INSERT and DELETE statements. It is only allowed after the optional WHERE condition.

#### **Related Topics**

- The WHERE Clause
- UPDATE (Executable Embedded SQL)

## 6.3.4 The DELETE Statement

Use the DELETE statement to remove rows from a table or view. In the following example, you delete all employees in a given department from the EMP table:

```
EXEC SQL DELETE FROM emp
WHERE deptno = :dept number ;
```

We have used the optional WHERE clause to specify the condition under which rows are deleted.

The returning clause option can be used in DELETE statements also. It is allowed after the optional WHERE condition. In the earlier example, it is good practice to record the field values of each employee that is deleted.

#### **Related Topics**

DELETE (Executable Embedded SQL)

## 6.3.5 The WHERE Clause

Use the WHERE clause to SELECT, UPDATE, or DELETE only those rows in a table or view that meet your search condition. The WHERE-clause search condition is a Boolean expression, which can include scalar host variables, host arrays (not in SELECT statements), subqueries, and user-defined stored functions.

If you omit the WHERE clause, all rows in the table or view are processed. If you omit the WHERE clause in an UPDATE or DELETE statement, Oracle sets sqlwarn[4] in the SQLCA to 'W' to warn that all rows were processed.



# 6.4 Using Boolean Data Type

Oracle Database supports storing of native boolean data type values. Pro\*C applications can perform bind and fetch operations on boolean data in the database. Using the boolean data type makes it easier to represent boolean state in applications instead of using a character type column to indicate "Y" or "N."

An application can include the boolean data type with the EXEC SQL statements in the following ways:

Declare a variable of boolean data type in the EXEC SQL variable declaration statement.

```
EXEC SQL BEGIN DECLARE SECTION;
   EXEC SQL var variable_name IS boolean;
EXEC SQL END DECLARE SECTION;
```

Define an existing table with boolean data type columns.

```
EXEC SQL DECLARE table_name TABLE (column_name boolean);
```

Use boolean data type to define user-defined data type (equivalent of C typedef).

```
EXEC SQL TYPE myBoolType IS boolean;
```

Create a table.

```
EXEC SQL CREATE TABLE booleanTable (b1 boolean);
```

Insert a boolean value into a table.

```
myBoolType b1 = TRUE;
EXEC SQL INSERT INTO table_name values(:b1);
```

Fetch data from a boolean data type column.

If data is fetched in a boolean data type variable, the data from boolean data type column returns a true or false value. If the output variable type is INTEGER then a 1 or 0 integer value is returned. If the database column value is <code>NULL</code> then the declared variable is untouched.

The following example gives the output as a boolean type, true or false:

```
bool b1;
EXEC SQL SELECT col_name INTO :b1
FROM table name;
```

The following example gives the output as an integer type, 1 or 0:

```
int c1;
EXEC SQL SELECT col_name INTO :c1
FROM table name;
```

The following example gives the output as a char type, "true" or "false":

```
char[5] var1;
EXEC SQL SELECT col_name INTO :var1
FROM table_name;
```

# 6.5 The DML Returning Clause

The INSERT, UPDATE, and DELETE statements can have an optional *DML returning clause* which returns column value expressions *expr*, into host variables *hv*, with host indicator variables *iv*. The DML returning clause looks like this:

```
{RETURNING | RETURN} {expr [,expr]}
INTO {:hv [[INDICATOR]:iv] [, :hv [[INDICATOR]:iv]]}
```

The number of expressions must equal the number of host variables. This clause eliminates the need for selecting the rows after an INSERT or UPDATE, and before a DELETE when you need to record that information for your application. The returning clause eliminates inefficient network round trips, extra processing, and server memory.

Oracle Dynamic SQL Method 4 does not support the DML returning clause; but ANSI Dynamic SQL Method 4 does. Support for DML statements with a DML returning clause that affects more than a single row is not supported by ANSI DYNAMIC SQL.

#### **Related Topics**

ANSI Dynamic SQL

## 6.6 Cursors

When a query returns multiple rows, you can explicitly define a cursor to

- Process beyond the first row returned by the query
- Keep track of which row is currently being processed

Or, you can use host arrays.

A cursor identifies the current row in the set of rows returned by the query. This allows your program to process the rows one at a time. The following statements let you define and manipulate a cursor:

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

First you use the DECLARE CURSOR statement to name the cursor and associate it with a query.

The OPEN statement executes the query and identifies all the rows that meet the query search condition. These rows form a set called the active set of the cursor. After OPENing the cursor, you can use it to retrieve the rows returned by its associated query.

Rows of the active set are retrieved one by one (unless you use host arrays). You use a FETCH statement to retrieve the current row in the active set. You can execute FETCH repeatedly until all rows have been retrieved.

When done FETCHing rows from the active set, you disable the cursor with a CLOSE statement, and the active set becomes undefined.

The following sections show you how to use these cursor control statements in your application program.

#### **Related Topics**

Host Arrays



#### 6.6.1 The DECLARE CURSOR Statement

You use the DECLARE CURSOR statement to define a cursor by giving it a name and associating it with a query, as the following example shows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, empno, sal
    FROM emp
    WHERE deptno = :dept number;
```

The cursor name is an identifier used by the precompiler, *not* a host or program variable, and should not be defined in the Declare Section. Therefore, cursor names cannot be passed from one precompilation unit to another. Cursor names cannot be hyphenated. They can be any length, but only the first 31 characters are significant. For ANSI compatibility, use cursor names no longer than 18 characters.

The precompiler option CLOSE\_ON\_COMMIT is provided for use in the command line or in a configuration file. Any cursor not declared with the WITH HOLD clause is closed after a COMMIT or ROLLBACK when CLOSE ON COMMIT=YES.

If MODE is specified at a higher level than CLOSE\_ON\_COMMIT, then MODE takes precedence. The defaults are MODE=ORACLE and CLOSE\_ON\_COMMIT=NO. If you specify MODE=ANSI then any cursors not using the WITH HOLD clause will be closed on COMMIT. The application will run more slowly because cursors are closed and re-opened many times. Setting CLOSE\_ON\_COMMIT=NO when MODE=ANSI results in performance improvement. To see how macro options such as MODE affect micro options such as CLOSE\_ON\_COMMIT, see "Precedence of Option Values".

The SELECT statement associated with the cursor cannot include an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement.

Because it is declarative, the DECLARE CURSOR statement must physically (not just logically) precede all other SQL statements referencing the cursor. That is, forward references to the cursor are not allowed. In the following example, the OPEN statement is misplaced:

```
EXEC SQL OPEN emp_cursor;

* -- MISPLACED OPEN STATEMENT

EXEC SQL DECLARE emp_cursor CURSOR FOR

SELECT ename, empno, sal

FROM emp

WHERE ename = :emp name;
```

The cursor control statements (DECLARE, OPEN, FETCH, CLOSE) must all occur within the same precompiled unit. For example, you cannot DECLARE a cursor in file A, then OPEN it in file B.

Your host program can DECLARE as many cursors as it needs. However, in a given file, every DECLARE statement must be unique. That is, you cannot DECLARE two cursors with the same name in one precompilation unit, even across blocks or procedures, because the scope of a cursor is global within a file.

For users of MODE=ANSI or CLOSE\_ON\_COMMIT=YES, the WITH HOLD clause can be used in a DECLARE section to override the behavior defined by the two options. With these options set, the behavior will be for all cursors to be closed when a COMMIT is issued. This can have performance implications due to the overhead of re-opening the cursor to continue processing. The careful use of WITH HOLD can speed up programs that need to conform to the ANSI standard for precompilers in most respects.



If you will be using many cursors, you might want to specify the MAXOPENCURSORS option.

#### **Related Topics**

- WITH HOLD Clause in DECLARE CURSOR Statements
- CLOSE\_ON\_COMMIT
- Precompiler Options
- Performance Tuning

## 6.6.2 The OPEN Statement

You use the OPEN statement to execute the query and identify the active set. In the following example, you OPEN a cursor named *emp\_cursor*:

```
EXEC SQL OPEN emp cursor;
```

OPEN zeroes the rows-processed count kept by the third element of SQLERRD in the SQLCA. However, none of the rows are visible to the application at this point. That is handled by the FETCH statement.

OPEN positions the cursor just before the first row of the active set. It also zeroes the rowsprocessed count kept by the third element of SQLERRD in the SQLCA. However, none of the rows is actually retrieved at this point. That will be done by the FETCH statement.

Once you OPEN a cursor, the query's input host variables are not re-examined until you reOPEN the cursor. Thus, the active set does not change. To change the active set, you must reOPEN the cursor.

Generally, you should CLOSE a cursor before reOPENing it. However, if you specify MODE=ORACLE (the default), you need not CLOSE a cursor before reOPENing it. This can increase performance.

The amount of work done by OPEN depends on the values of three precompiler options: HOLD CURSOR, RELEASE CURSOR, and MAXOPENCURSORS.

#### **Related Topics**

About Using the Precompiler Options

## 6.6.3 The FETCH Statement

You use the FETCH statement to retrieve rows from the active set and specify the output host variables that will contain the results. Recall that the SELECT statement associated with the cursor cannot include an INTO clause. Rather, the INTO clause and list of output host variables are part of the FETCH statement. In the following example, you FETCH INTO three host variables:

```
EXEC SQL FETCH emp_cursor
INTO :emp_name, :emp_number, :salary;
```

The cursor must have been previously DECLAREd and OPENed. The first time you execute FETCH, the cursor moves from before the first row in the active set to the first row. This row becomes the current row. Each subsequent execution of FETCH advances the cursor to the next row in the active set, changing the current row. The cursor can only move forward in the active set. To return to a row that has already been FETCHed, you must reOPEN the cursor, then begin again at the first row of the active set.



If you want to change the active set, you must assign new values to the input host variables in the query associated with the cursor, then reOPEN the cursor. When MODE=ANSI, you must CLOSE the cursor before reOPENing it.

As the next example shows, you can FETCH from the same cursor using different sets of output host variables. However, corresponding host variables in the INTO clause of each FETCH statement must have the same datatype.

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, sal FROM emp WHERE deptno = 20;
...
EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name1, :salary1;
    EXEC SQL FETCH emp_cursor INTO :emp_name2, :salary2;
    EXEC SQL FETCH emp_cursor INTO :emp_name3, :salary3;
...
}
```

If the active set is empty or contains no more rows, FETCH returns the "no data found" error code to *sqlcode* in the SQLCA, or to the SQLCODE or SQLSTATE status variables. The status of the output host variables is indeterminate. (In a typical program, the WHENEVER NOT FOUND statement detects this error.) To reuse the cursor, you must reOPEN it.

It is an error to FETCH on a cursor under the following conditions:

- Before OPENing the cursor
- · After a "no data found" condition
- After CLOSEing it

## 6.6.4 The CLOSE Statement

When done FETCHing rows from the active set, you CLOSE the cursor to free the resources, such as storage, acquired by OPENing the cursor. When a cursor is closed, parse locks are released. What resources are freed depends on how you specify the HOLD\_CURSOR and RELEASE\_CURSOR options. In the following example, you CLOSE the cursor named *emp\_cursor*:

```
EXEC SQL CLOSE emp_cursor;
```

You cannot FETCH from a closed cursor because its active set becomes undefined. If necessary, you can reOPEN a cursor (with new values for the input host variables, for example).

When MODE=ORACLE, issuing a COMMIT or ROLLBACK closes cursors referenced in a CURRENT OF clause. Other cursors are unaffected by COMMIT or ROLLBACK and if open, remain open. However, when MODE=ANSI, issuing a COMMIT or ROLLBACK closes *all* explicit cursors.

#### **Related Topics**

Database Concepts



## 6.7 Scrollable Cursors

A scrollable cursor is a work area where Oracle executes SQL statements and stores information that is processed during execution.

When a cursor is executed, the results of the query are placed into a a set of rows called the result set. The result set can be fetched either sequentially or non-sequentially. Non-sequential result sets are called scrollable cursors.

A scrollable cursor enables users to access the rows of a database result set in a forward, backward, and random manner. This enables the program to fetch any row in the result set. See *Oracle Call Interface Programmer's Guide*, Release 9.2.0.

## 6.7.1 About Using Scrollable Cursors

The following statements let you define and manipulate a scrollable cursor.

#### 6.7.1.1 DECLARE SCROLL CURSOR

You can use the DECLARE <cursor name > SCROLL CURSOR statement to name the scrollable cursor and associate it with a query.

#### 6.7.1.2 OPEN for Scrollable Cursors

You can use the OPEN statement in the same way as in the case of a non-scrollable cursor.

#### 6.7.1.3 FETCH for Scrollable Cursors

You can use the FETCH statement to fetch required rows in a random manner. An application can fetch rows up or down, first or last row directly, or fetch any single row in a random manner.

The following options are available with the FETCH statement.

1. FETCH FIRST

Fetches the first row from the result set.

2. FETCH PRIOR

Fetches the row prior to the current row.

3. FETCH NEXT

Fetches the next row from the current position. This is same as the non-scrollable cursor FETCH.

4. FETCH LAST

Fetches the last row from the result set.

5. FETCH CURRENT

Fetches the current row.

6. FETCH RELATIVE n

Fetches the *n*th row relative to the current row, where *n* is the offset.

7. FETCH ABSOLUTE n



Fetches the *n*th row, where *n* is the offset from the start of the result set.

The following example describes how to FETCH the last record from a result set.

```
EXEC SQL DECLARE emp_cursor SCROLL CURSOR FOR SELECT ename, sal FROM emp WHERE deptno=20; ...

EXEC SQL OPEN emp_cursor;

EXEC SQL FETCH LAST emp_cursor INTO :emp_name, :sal;

EXEC SQL CLOSE emp cursor;
```

## 6.7.1.4 CLOSE for Scrollable Cursors

You can use the CLOSE statement in the same way as in the case of a non-scrollable cursor.



You cannot use scrollable cursors for REF cursors.

## 6.7.2 The CLOSE ON COMMIT Precompiler Option

The CLOSE\_ON\_COMMIT micro precompiler option provides the ability to choose whether or not to close all cursors when a COMMIT is executed and the macro option MODE=ANSI. When MODE=ANSI, CLOSE\_ON\_COMMIT has the default value YES. Explicitly setting CLOSE\_ON\_COMMIT=NO results in better performance because cursors will not be closed when a COMMIT is executed, removing the need to re-open the cursors and incur extra parsing.

## 6.7.3 The PREFETCH Precompiler Option

The precompiler option PREFETCH allows for more efficient queries by pre-fetching a given number of rows. This decreases the number of server round trips needed and reduces overall memory usage. The number of rows set by the PREFETCH option value is used for all queries involving explicit cursors, subject to the standard precedence rules. When used inline, the PREFETCH option must precede any of these cursor statements:

- EXEC SQL OPEN cursor
- EXEC SQL OPEN cursor USING host var list
- EXEC SQL OPEN cursor USING DESCRIPTOR desc name

When an OPEN is executed, the value of PREFETCH gives the number of rows to be prefetched when the query is executed. You can set the value from 0 (no pre-fetching) to 65535. The default value is 1.



#### **Note:**

The default value of the PREFETCH option is 1 - return a single row for each round-trip. If you choose not to use the PREFETCH option, using the command line, you must explicitly disable it by setting the PREFETCH option to 0.

PREFETCH is automatically disabled when LONG or LOB columns are being retrieved.

#### Note:

PREFETCH is used primarily to enhance the performance of single row fetches.

#### Note:

The PREFETCH option should be used wisely, and on a case-by-case basis. Select an appropriate PREFETCH value that will optimize performance of a specific FETCH statement. To accomplish this, use the inline prefetch option instead of the command line prefetch option.

#### Note:

The performance of many large applications can be improved simply by using indicator variables with host variables in FETCH statements.

To enable precompiler applications to obtain the maximum advantage from the use of the PREFETCH option on single row fetches, it is strongly recommended that you use indicator variables.

# 6.8 Optimizer Hints

The Pro\*C/C++ Precompiler supports optimizer hints in SQL statements. An *optimizer hint* is a suggestion to the Oracle SQL optimizer that can override the optimization approach that would normally be taken. You can use hints to specify the

- Optimization approach for a SQL statement
- Access path for each referenced table
- Join order for a join
- Method used to join tables

Hints allow you to choose between rule-based and cost-based optimization. With cost-based optimization, you can use further hints to maximize throughput or response time.



## 6.8.1 Issuing Hints

You can issue an optimizer hint inside a C or C++ style comment, immediately after a SELECT, DELETE, or UPDATE command. You indicate that the comment contains one or more hints by following the comment opener with a plus sign, leaving no space between the opener and the '+'. For example, the following statement uses the ALL\_ROWS hint to let the cost-based approach optimize the statement for the goal of best throughput:

```
EXEC SQL SELECT /*+ ALL_ROWS (cost-based) */ empno, ename, sal, job
   INTO :emp_rec FROM emp
   WHERE deptno = :dept number;
```

As shown in this statement, the comment can contain optimizer hints as well as other comments.

## 6.9 Fix Execution Plan

In application development environments where modules are developed in one environment, and then integrated and deployed into another, the performance of the applications are affected. At times, the performance of the precompiler applications are affected by changes in the database environment. These may include changes in the optimizer statistics, changes to the optimizer settings, or changes to parameters affecting the sizes of memory structures.

To fix execution plans for SQL's used in Pro\*C/C++ in the development environment, you need to use the outline feature of Oracle at the time of precompiling. An outline is implemented as a set of optimizer hints that are associated with the SQL statement. If you enable the use of the outline for the statement, Oracle automatically considers the stored hints and tries to generate an execution plan in accordance with those hints. In this way, you can ensure that the performance is not affected when the modules are integrated or deployed into different environments.

You can use the following SQL statements to create outlines in Pro\*C/C++:

- SELECT
- DELETE
- UPDATE
- INSERT ... SELECT
- CREATE TABLE ... AS SELECT

If the outline option is set, then the precompiler generates two files, a SQL file and a LOG file at the end of successful precompilation. Command line options outline and outlnprefix control the generation of the outlines.

Each generated outline name is unique. Because the file names used in the application are unique, this information is used in generating the outline name. In addition, the category name is also prefixed.



#### **Caution:**

Oracle allows only 128 bytes for the outline name. If you exceed the limit, the precompiler will flag an error. You can restrict the length of the outline name by using the outlnprefix option.

#### Example 6-1 Generating a SQL File Containing Outlines

You need to precompile the following program by using the outline option to generate SQL files containing the outlines for all the outline-supported SQL statements in this program.

```
* outlndemo.pc
 * Outlines will be created for the following SQL operations,
 * 1. CREATE ... SELECT
 * 2. INSERT ... SELECT
 * 3. UPDATE
 * 4. DELETE
 * 5. SELECT
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
#include <sqlca.h>
/* Error handling function. */
void sql_error(char *msg)
 exec sql whenever sqlerror continue;
 printf("\n%s\n", msq);
 printf("%.70s\n", sqlca.sqlerrm.sqlerrmc);
 exec sql rollback release;
  exit(EXIT FAILURE);
int main()
 varchar ename[10];
 varchar job[9];
 float sal, comm;
 exec sql begin declare section;
   char *uid = "scott/tiger";
  exec sql end declare section;
 exec sql whenever sqlerror do sql error("ORACLE error--\n");
 exec sql connect :uid;
  exec sql insert into bonus
           select ename, job, sal, comm from emp where job like 'SALESMAN';
  exec sql update bonus set sal = sal \star 1.1 where sal < 1500;
  exec sql declare c1 cursor for
          select ename, job, sal, comm from bonus order by sal;
  exec sql open c1;
 printf ("Contents of updated BONUS table\n'");
 printf ("ENAME
                    JOB
                            SALARY COMMISSION\n\n");
  exec sql whenever not found do break;
 while (1)
    exec sql fetch cl into :ename, :job, :sal, :comm;
    ename.arr[ename.len]='\0';
```



## 6.9.1 SQL File

The generated file name has the following format:

```
<filename>_<filetype>.sql
```

In Pro\*C, for the file "abc.pc", the generated SQL file will be abc\_pc.sql.

#### Generated file format

If the outInprefix option is not used, then the format of the unique identifier used as outline name and comment is:

```
<category_name>_<filename>_<filetype>_<sequence no.>
```

If the outInprefix option is used (outInprefix=cprefix\_name>), then the format of the unique identifier used as outline name and comment is:

```
fix name> <sequence no.>
```

If outline=yes, which is the default category, then <category\_name> will be DEFAULT and outline name will be:

```
DEFAULT_<filename>_<filetype>_<sequence no.>

or

<prefix_name>_<sequence no.>
```

The allowed range for <sequence no.> is 0000 to 9999.

SQL in the generated precompiled file will have the comment appended to it as it appears in the outline for that SQL.

## 6.9.1.1 Examples

Consider the following examples.

#### **Example 1**

If abc.pc has the statements

```
EXEC SQL select * from emp where empno=:var;
EXEC SQL select * from dept;
```

and if outline=mycat1 and outlnprefix is not used, then:

#### Contents of abc\_pc.sql

```
create or replace outline mycat1_abc_pc_0000 for category mycat1 on select * from emp
where empno=:b1 /* mycat1_abc_pc_0000 */;
```

create or replace outline mycat1\_abc\_pc\_0001 for category mycat1 on select \* from
dept /\* mycat1 abc pc 0001 \*/;

#### Contents of abc.c

```
sqlstm.stmt = select * from emp where empno=:b1 /* mycat1_abc_pc_0000 */;
sqlstm.stmt = select * from dept /* mycat1 abc pc 0001 */;
```

#### **Example 2**

#### If abc.pc has the statements

```
EXEC SQL select * from emp where empno=:var;
EXEC SQL select * from dept;
```

and if outline=mycat1 and outInprefix=myprefix, then:

#### Contents of abc\_pc.sql

create or replace outline myprefix\_0000 for category mycat1 on select \* from emp where
empno=:b1 /\* myprefix\_0000 \*/;

create or replace outline myprefix\_0001 for category mycat1 on select \* from dept /\*
myprefix\_0001 \*/;

#### Contents of abc.c

```
sqlstm.stmt = select * from emp where empno=:b1 /* myprefix_0000 */;
sqlstm.stmt = select * from dept /* myprefix 0001 */;
```

#### Example 3

#### If abc.pc has the statements

```
EXEC SQL select * from emp where empno=:var;
EXEC SQL select * from dept;
```

and if outline=yes and outInprefix=myprefix, then:

#### Contents of abc pc.sql

```
create or replace outline myprefix_0000 on select * from emp where empno=:b1 /* myprefix_0000 */;
```

create or replace outline myprefix\_0001 on select \* from dept /\* myprefix 0001 \*/;



#### Contents of abc.c

```
sqlstm.stmt = "select * from emp where empno=:b1 /* myprefix_0000 */;
sqlstm.stmt = "select * from dept /* myprefix 0001 */";
```

## 6.9.2 LOG File

The generated file name has the following format:

```
<filename> <filetype>.log
```

In Pro\*C, for the file "abc.pc", the generated LOG file will be abc pc.log.

Consider the following example.

#### **Example 1**

If abc.pc has the statements

```
EXEC SQL select * from emp;
```

#### Contents of abc\_pc.log

```
CATEGORY <Category_name>
Source SQL_0
SELECT * FROM emp
OUTLINE NAME
abc_pc_0000
OUTLINE SQL_0
Select * from emp /* abc pc 0000 */
```

## 6.10 The CURRENT OF Clause

You use the CURRENT OF *cursor\_name* clause in a DELETE or UPDATE statement to refer to the latest row FETCHed from the named cursor. The cursor must be open and positioned on a row. If no FETCH has been done or if the cursor is not open, the CURRENT OF clause results in an error and processes no rows.

The FOR UPDATE OF clause is optional when you DECLARE a cursor that is referenced in the CURRENT OF clause of an UPDATE or DELETE statement. The CURRENT OF clause signals the precompiler to add a FOR UPDATE clause if necessary.

In the following example, you use the CURRENT OF clause to refer to the latest row FETCHed from a cursor named *emp\_cursor*:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, sal FROM emp WHERE job = 'CLERK'
    FOR UPDATE OF sal;
...

EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND GOTO ...
for (;;) {
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    ...
    EXEC SQL UPDATE emp SET sal = :new_salary
        WHERE CURRENT OF emp_cursor;
}
```

#### **Related Topics**

Using FOR UPDATE OF

## 6.10.1 Restrictions (FOR UPDATE OF)

You cannot use CURRENT OF clause on an index-organized table.

Explicit FOR UPDATE OF clauses or implicit FOR UPDATE clauses acquire exclusive row locks. All rows are locked at the OPEN, not as they are FETCHed, and are released when you COMMIT or ROLLBACK. Therefore, you cannot FETCH from a FOR UPDATE cursor after a COMMIT. If you try to do this, Oracle returns a 1002 error code.

Also, you cannot use host arrays with the CURRENT OF clause. For an alternative, see also "About Mimicking CURRENT OF".

Furthermore, you cannot reference multiple tables in an associated FOR UPDATE OF clause, which means that you cannot do joins with the CURRENT OF clause.

Finally, you cannot use dynamic SQL with the CURRENT OF clause.

## 6.11 The Cursor Statements

The following example shows the typical sequence of cursor control statements in an application program:

```
/* define a cursor */
   EXEC SQL DECLARE emp cursor CURSOR FOR
       SELECT ename, job
       FROM emp
       WHERE empno = :emp number
       FOR UPDATE OF job;
/* open the cursor and identify the active set */
  EXEC SQL OPEN emp cursor;
/* break if the last row was already fetched */
   EXEC SQL WHENEVER NOT FOUND DO break;
/* fetch and process data in a loop */
   for (;;)
      EXEC SQL FETCH emp cursor INTO :emp name, :job title;
/* optional host-language statements that operate on
   the FETCHed data */
      EXEC SQL UPDATE emp
           SET job = :new job title
           WHERE CURRENT OF emp cursor;
/* disable the cursor */
  EXEC SQL CLOSE emp cursor;
  EXEC SQL COMMIT WORK RELEASE;
```



# 6.12 A Complete Example Using Non-Scrollable Cursor

The following complete program illustrates the use of a cursor and the FETCH statement. The program prompts for a department number, then displays the names of all employees in that department.

All FETCHes except the final one return a row and, if no errors were detected during the FETCH, a success status code. The final FETCH fails and returns the "no data found" Oracle error code to *sqlca.sqlcode*. The cumulative number of rows actually FETCHed is found in sqlerrd[2] in the SQLCA.

```
#include <stdio.h>
/* declare host variables */
char userid[12] = "SCOTT/TIGER";
char emp name[10];
int emp number;
int dept number;
char temp[32];
void sql error();
/* include the SQL Communications Area */
#include <sqlca.h>
main()
{ emp number = 7499;
/* handle errors */
   EXEC SQL WHENEVER SQLERROR do sql error ("Oracle error");
/* connect to Oracle */
   EXEC SQL CONNECT :userid;
   printf("Connected.\n");
/* declare a cursor */
   EXEC SQL DECLARE emp_cursor CURSOR FOR
   SELECT ename
     FROM emp
      WHERE deptno = :dept number;
   printf("Department number? ");
   gets (temp);
   dept number = atoi(temp);
/* open the cursor and identify the active set */
   EXEC SQL OPEN emp cursor;
   printf("Employee Name\n");
   printf("----\n");
/\star fetch and process data in a loop
   exit when no more data */
   EXEC SQL WHENEVER NOT FOUND DO break;
   while (1)
      EXEC SQL FETCH emp cursor INTO :emp name;
      printf("%s\n", emp_name);
   EXEC SQL CLOSE emp cursor;
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
```

```
void
sql_error(msg)
char *msg;
{
    char buf[500];
    int buflen, msglen;

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    buflen = sizeof (buf);
    sqlglm(buf, &buflen, &msglen);
    printf("%s\n", msg);
    printf("%*.s\n", msglen, buf);
    exit(1);
```

# 6.13 A Complete Example Using Scrollable Cursor

The following program illustrates the use of scrollable cursor and the various options used by the FETCH statement.

```
#include <stdio.h>
/* declare host variables */
char userid[12]="SCOTT/TIGER";
char emp_name[10];
void sql_error();
/* include the SQL Communications Area */
#include<sqlca.h>
main()
/* handle errors */
    EXEC SQL WHENEVER SQLERROR do sql error("Oracle error");
/* connect to Oracle */
    EXEC SQL CONNECT :userid;
    printf("Connected.\n");
/* declare a scrollable cursor */
    EXEC SQL DECLARE emp_cursor SCROLL CURSOR FOR
    SELECT ename FROM emp;
/* open the cursor and identify the active set */
     EXEC SQL OPEN emp cursor;
 /* Fetch the last row */
     EXEC SQL FETCH LAST emp_cursor INTO :emp_name;
 /* Fetch row number 5 */
     EXEC SQL FETCH ABSOLUTE 5 emp_cursor INTO :emp_name;
/* Fetch row number 10 */
      EXEC SQL FETCH RELATIVE 5 emp cursor INTO :emp name;
   Fetch row number 7 */
       EXEC SQl FETCH RELATIVE -3 emp cursor INTO :emp name;
/* Fetch the first row */
```

```
EXEC SQL FETCH FIRST emp cursor INTO :emp name;
   Fetch row number 2*/
       EXEC SQL FETCH my_cursor INTO :emp_name;
/* Fetch row number 3 */
       EXEC SQL FETCH NEXT my cursor INTO :emp name;
    Fetch row number 3 */
       EXEC SQL FETCH CURRENT my_cursor INTO :emp_name;
  Fetch row number 2 */
        EXEC SQL FETCH PRIOR my_cursor INTO :emp_name;
void
sql error(msg)
char *msg;
     char buf[500];
    int buflen , msglen;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK TRANSACTION;
    buflen = sizeof (buf);
     sqlglm(buf, &buflen, &mesglen);
     printf("%s\n", msg);
     printf("%*.s\n",msglen,buf);
     exit(1);
```

# 6.14 Positioned Update

The following skeletal example demonstrates positioned update using the universal ROWID.

```
#include <oci.h>
...
OCIRowid *urowid;
...
EXEC SQL ALLOCATE :urowid;
EXEC SQL DECLARE cur CURSOR FOR
        SELECT rowid, ... FROM my_table FOR UPDATE OF ...;
EXEC SQL OPEN cur;
EXEC SQL FETCH cur INTO :urowid, ...;
/* Process data */
...
EXEC SQL UPDATE my_table SET ... WHERE CURRENT OF cur;
EXEC SQL CLOSE cur;
EXEC SQL FREE :urowid;
...
```

#### **Related Topics**

Universal ROWIDs

# 6.15 Explicit Request Boundaries for Handling Planned Maintenance

Oracle high availability (HA) best practices define a strategy to ensure that applications can maximize uptime by shifting workloads to an available Oracle Real Application Cluster (RAC) instance or to an Active DataGuard (ADG) standby ahead of a planned maintenance. Applications are programmatically notified prior to the start of planned maintenance, thus enabling applications to shift the workloads without returning disruptive errors to the end user. Additionally, failover-enabled services ensure that new application requests are directed to a database instance or standby that is not at a risk of being taken down when maintenance activities commence.

#### See Also:

Configuring Continuous Availability for Applications in *Oracle Database High Availability Overview and Best Practices* for information about configuring an application for planned maintenance

To handle a planned maintenance in a transparent manner, Oracle Pro\*C precompiler applications can use explicit request boundaries coupled with a failover-enabled service. Applications can programmatically note request boundaries, which are tags that mark the beginning and end of a database request. When an application gets notified about a planned maintenance, the application can use the explicit request boundaries to transparently shift the work to an appropriate database instance or standby (away from the instance or database that is about to be patched) before proceeding with the application request.

#### Note:

Applications must connect to a Transparent Application Failover (TAF) service when using the embedded SQL statement syntax for explicit request boundaries.

#### EXEC SQL REQUEST BEGIN

The EXEC SQL REQUEST BEGIN statement notes the start of an application request. As with other embedded SQL statements, this call can operate on the default (unnamed) connection, on a connection specified by a precompiler constant, or on a connection name provided through a host variable. The EXEC SQL REQUEST BEGIN statement validates the connection by checking for the receipt of a planned maintenance notification. If the validation fails because the connection is not responding or because a planned maintenance notification has been received, the connection is replaced with a new connection. According to the guidelines in Oracle high availability best practices, when constructing the application's connect string, the new connection is established to an alternate instance or to the standby.





If your application relies on statistics maintained by the ORACA, a failed-over connection is considered to be a continuation of the same logical connection, hence the statistics are not reset.

The following are some examples of using the EXEC SQL REQUEST BEGIN statement.

#### **Example 6-2 Using the Default Connection**

```
EXEC SQL REQUEST BEGIN;
```

#### Example 6-3 Using the Connection in a Host Variable ":db\_name"

```
EXEC SQL AT : db name REQUEST BEGIN;
```

#### Example 6-4 Using the Connection Associated with a Constant "DB\_NAME"

```
EXEC SQL DECLARE DB_NAME DATABASE;

EXEC SQL CONNECT :username IDENTIFIED BY :password at DB_NAME;

EXEC SQL AT DB_NAME REQUEST BEGIN;
```

#### EXEC SQL REQUEST END

The EXEC SQL REQUEST END statement notes the end of an application request. As with other embedded SQL statements, this call can operate on the default (unnamed) connection, on a connection specified by a precompiler constant, or on a connection name provided through a host variable.

#### **Example 6-5 Using the Default Connection**

```
EXEC SQL REQUEST END;
```

#### Example 6-6 Using the Connection in a Host Variable ":db\_name"

```
EXEC SQL AT : db name REQUEST END;
```

#### Example 6-7 Using the Connection Associated with a Constant "DB\_NAME"

```
EXEC SQL DECLARE DB_NAME DATABASE;
EXEC SQL CONNECT :username IDENTIFIED BY :password at DB_NAME;
EXEC SQL AT DB NAME REQUEST END;
```

If an uncommitted transaction is detected when EXEC SQL REQUEST END is run, it is implicitly rolled back.

## 6.15.1 Explicit Request Boundaries Limitations

The embedded SQL statements exec sql request begin and exec sql request end cannot be used if your precompiler application is:

- Deployed as a service that is part of an XA application.
- Linked in as a SQL\*Forms user exit.
- Using connection pooling because when connection pooling is used planned outage is not supported.
- Connected to Application Continuity (AC) or Transparent Application Continuity (TAC) services.

## 6.15.2 Explicit Request Boundaries Example

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
/* Define constants for VARCHAR lengths. */
#define UNAME_LEN
#define PWD_LEN #define DB_LEN
                           40
                           20
/* Declare variables. No declare section is
   needed if MODE=ORACLE. */
void main(int argc, char* argv[])
 EXEC SQL BEGIN DECLARE SECTION;
   int beforeSessionId, beforeSerialNum, afterSessionId, afterSessionNum;
  EXEC SQL END DECLARE SECTION;
  /* Register sql error() as the error handler. */
  EXEC SQL WHENEVER SQLERROR DO sql error("ORACLE error--\n");
  /* Connect to the database with the default, unnamed connection omitted for
brevity. */
  EXEC SOL SELECT SID, SERIAL#
           INTO :beforeSessionId, :beforeSerialNum
           FROM v$session
           WHERE USERNAME='SCOTT';
  printf("Session ID and Serial # before failover\n");
  printf("session ID:%d \nserial num:%d\n", beforeSessionId, beforeSerialNum);
  EXEC SQL REQUEST BEGIN;
  EXEC SQL CREATE TABLE TPC2AC01 TAB(col1 INT);
  EXEC SQL REQUEST END;
  EXEC SOL REQUEST BEGIN;
  printf("EXEC SQL INSERT INTO TPC2AC01 TAB VALUES(3);\n");
  EXEC SQL INSERT INTO TPC2AC01 TAB VALUES(3);
  EXEC SQL COMMIT;
  EXEC SQL REQUEST END;
  EXEC SQL REQUEST BEGIN;
  printf("EXEC SQL SELECT COL1 INTO :COL1 FROM TPC2AC01 TAB; \n");
  EXEC SQL SELECT COL1 INTO :col1 FROM TPC2AC01 TAB;
  printf("number of rows returned:%d\n", sqlca.sqlerrd[2]);
  EXEC SQL REQUEST END;
```

```
/* stop the target service with a drain timeout */
printf("EXEC SQL REQUEST BEGIN; \n"); /* this should get a new connection */
EXEC SQL REQUEST BEGIN;
printf("EXEC SQL INSERT INTO TPC2AC01 TAB VALUES(1);\n");
EXEC SQL INSERT INTO TPC2AC01 TAB VALUES(1);
printf("EXEC SQL INSERT INTO TPC2AC01 TAB VALUES(2);\n");
EXEC SQL INSERT INTO TPC2AC01 TAB VALUES(2);
printf("EXEC SELECT SID, SERIAL# FROM v$session WHERE"
       " USERNAME='SCOTT'\n");
EXEC SQL SELECT SID, SERIAL#
        INTO :afterSessionId, :afterSerialNum
         FROM v$session
        WHERE USERNAME='SCOTT';
printf("Session ID and Serial # after failover \n");
printf("session ID:%d \nserial num:%d\n", afterSessionId, afterSerialNum);
printf("EXEC SQL REQUEST END\n");
EXEC SQL REQUEST END;
```



7

# Embedded PL/SQL

This chapter shows you how to improve performance by embedding PL/SQL transaction processing blocks in your program. This chapter contains the following topics:

- Advantages of PL/SQL
- Embedded PL/SQL Blocks
- Host Variables
- Indicator Variables
- Host Arrays
- Cursor Usage in Embedded PL/SQL
- Stored PL/SQL and Java Subprograms
- External Procedures
- About Using Dynamic SQL

#### **Related Topics**

Embedded PL/SQL

# 7.1 Advantages of PL/SQL

This section looks at some of the features and benefits offered by PL/SQL, such as:

- Better Performance
- · Integration with Oracle
- Cursor FOR Loops
- Procedures and Functions
- Packages
- PL/SQL Tables
- User-Defined Records

#### **Related Topics**

Oracle Database PL/SQL Language Reference

## 7.1.1 Better Performance

PL/SQL can help you reduce overhead, improve performance, and increase productivity. For example, without PL/SQL, Oracle must process SQL statements one at a time. Each SQL statement results in another call to the Server and higher overhead. However, with PL/SQL, you can send an entire block of SQL statements to the Server. This minimizes communication between your application and Oracle.

## 7.1.2 Integration with Oracle

PL/SQL is tightly integrated with the Oracle Server. For example, most PL/SQL datatypes are native to the Oracle data dictionary. Furthermore, you can use the %TYPE attribute to base variable declarations on column definitions stored in the data dictionary, as the following example shows:

```
job title emp.job%TYPE;
```

That way, you need not know the exact datatype of the column. Furthermore, if a column definition changes, the variable declaration changes accordingly and automatically. This provides data independence, reduces maintenance costs, and allows programs to adapt as the database changes.

## 7.1.3 Cursor FOR Loops

With PL/SQL, you need not use the DECLARE, OPEN, FETCH, and CLOSE statements to define and manipulate a cursor. Instead, you can use a cursor FOR loop, which implicitly declares its loop index as a record, opens the cursor associated with a given query, repeatedly fetches data from the cursor into the record, then closes the cursor. An example follows:

```
DECLARE
...

BEGIN

FOR emprec IN (SELECT empno, sal, comm FROM emp) LOOP

IF emprec.comm / emprec.sal > 0.25 THEN ...

END LOOP;

END;
```

Notice that you use dot notation to reference components in the record.

## 7.1.4 Procedures and Functions

PL/SQL has two types of subprograms called *procedures* and *functions*, which aid application development by letting you isolate operations. Generally, you use a procedure to perform an action and a function to compute a value.

Procedures and functions provide *extensibility*. That is, they let you tailor the PL/SQL language to suit your needs. For example, if you need a procedure that creates a new department, just write your own as follows:

```
PROCEDURE create_dept
  (new_dname IN CHAR(14),
    new_loc IN CHAR(13),
    new_deptno OUT NUMBER(2)) IS

BEGIN
   SELECT deptno_seq.NEXTVAL INTO new_deptno FROM dual;
   INSERT INTO dept VALUES (new_deptno, new_dname, new_loc);

END create_dept;
```

When called, this procedure accepts a new department name and location, selects the next value in a department-number database sequence, inserts the new number, name, and location into the *dept* table, then returns the new number to the caller.

You use *parameter modes* to define the behavior of formal parameters. There are three parameter modes: IN (the default), OUT, and IN OUT. An IN parameter lets you pass values to

the subprogram being called. An OUT parameter lets you return values to the caller of a subprogram. An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller.

The datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Table 7-1 shows the legal conversions between datatypes.

## 7.1.5 Packages

PL/SQL lets you bundle logically related types, program objects, and subprograms into a *package*. With the Procedural Database Extension, packages can be compiled and stored in an Oracle database, where their contents can be shared by many applications.

Packages usually have two parts: a specification and a body. The *specification* is the interface to your applications; it declares the types, constants, variables, exceptions, cursors, and subprograms available for use. The *body* defines cursors and subprograms; it implements the specification. In the following example, you "package" two employment procedures:

```
PACKAGE emp_actions IS -- package specification
PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...);

PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;

PACKAGE BODY emp_actions IS -- package body
PROCEDURE hire_employee (empno NUMBER, ename CHAR, ...) IS
BEGIN
INSERT INTO emp VALUES (empno, ename, ...);
END hire_employee;

PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
DELETE FROM emp WHERE empno = emp_id;
END fire_employee;
END emp_actions;
```

Only the declarations in the package specification are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible.

## 7.1.6 PL/SQL Tables

PL/SQL provides a composite datatype named TABLE. Objects of type TABLE are called *PL/SQL tables*, which are modeled as (but not the same as) database tables. PL/SQL tables have only one column and use a primary key to give you array-like access to rows. The column can belong to any scalar type (such as CHAR, DATE, or NUMBER), but the primary key must belong to type BINARY\_INTEGER, PLS\_INTEGER or VARCHAR2.

You can declare PL/SQL table types in the declarative part of any block, procedure, function, or package. In the following example, you declare a TABLE type called *NumTabTyp*:

```
DECLARE

TYPE NumTabTyp IS TABLE OF NUMBER

INDEX BY BINARY_INTEGER;

...

BEGIN

...

END;
```



Once you define type *NumTabTyp*, you can declare PL/SQL tables of that type, as the next example shows:

```
num_tab NumTabTyp;
```

The identifier *num\_tab* represents an entire PL/SQL table.

You reference rows in a PL/SQL table using array-like syntax to specify the primary key value. For example, you reference the ninth row in the PL/SQL table named *num\_tab* as follows:

```
num tab(9) ...
```

## 7.1.7 User-Defined Records

You can use the %ROWTYPE attribute to declare a record that represents a row in a table or a row fetched by a cursor. However, you cannot specify the datatypes of components in the record or define components of your own. The composite datatype RECORD lifts those restrictions.

Objects of type RECORD are called *records*. Unlike PL/SQL tables, records have uniquely named components, which can belong to different datatypes. For example, suppose you have different kinds of data about an employee such as name, salary, hire date, and so on. This data is dissimilar in type but logically related. A record that contains such components as the name, salary, and hire date of an employee would let you treat the data as a logical unit.

You can declare record types and objects in the declarative part of any block, procedure, function, or package. In the following example, you declare a RECORD type called <code>DeptRecTyp</code>:

```
DECLARE

TYPE DeptRecTyp IS RECORD

(deptno NUMBER(4) NOT NULL, -- default is NULL allowed dname CHAR(9),
loc CHAR(14));
```

Notice that the component declarations are like variable declarations. Each component has a unique name and specific datatype. You can add the NOT NULL option to any component declaration and so prevent the assigning of NULLs to that component.

Once you define type *DeptRecTyp*, you can declare records of that type, as the next example shows:

```
dept_rec DeptRecTyp;
```

The identifier *dept\_rec* represents an entire record.

You use dot notation to reference individual components in a record. For example, you reference the *dname* component in the *dept\_rec* record as follows:

```
dept rec.dname ...
```

# 7.2 Embedded PL/SQL Blocks

The Pro\*C/C++ Precompiler treats a PL/SQL block like a single embedded SQL statement. So, you can place a PL/SQL block anywhere in a program that you can place a SQL statement.

To embed a PL/SQL block in your Pro\*C/C++ program, simply bracket the PL/SQL block with the keywords EXEC SQL EXECUTE and END-EXEC as follows:

```
EXEC SQL EXECUTE
DECLARE
...
BEGIN
...
END;
END-EXEC;
```

The keyword END-EXEC must be followed by a semicolon.

After writing your program, you precompile the source file in the usual way.

When the program contains embedded PL/SQL, you must use the SQLCHECK=SEMANTICS command-line option, since the PL/SQL must be parsed by the Oracle Server. SQLCHECK=SEMANTICS requires the USERID option also, to connect to a server.

#### **Related Topics**

About Using the Precompiler Options

## 7.3 Host Variables

Host variables are the key to communication between a host language and a PL/SQL block. Host variables can be shared with PL/SQL, meaning that PL/SQL can set and reference host variables.

For example, you can prompt a user for information and use host variables to pass that information to a PL/SQL block. Then, PL/SQL can access the database and use host variables to pass the results back to your host program.

Inside a PL/SQL block, host variables are treated as global to the entire block and can be used anywhere a PL/SQL variable is allowed. Like host variables in a SQL statement, host variables in a PL/SQL block must be prefixed with a colon. The colon sets host variables apart from PL/SQL variables and database objects.



To use VARCHAR, CHARZ, or STRING types as output host variables in PL/SQL blocks, you must initialize the length *before* entering the block. Set the length to the declared (maximum) length of the VARCHAR, CHARZ, or STRING.

#### **Related Topics**

Pointer Variables

## 7.3.1 Example: Using Host Variables with PL/SQL

The following example illustrates the use of host variables with PL/SQL. The program prompts the user for an employee number, then displays the job title, hire date, and salary of that employee.

```
char username[100], password[20];
char job_title[20], hire_date[9], temp[32];
int emp_number;
float salary;
#include <sqlca.h>
```



```
printf("Username? \n");
gets(username);
printf("Password? \n");
gets (password);
EXEC SQL WHENEVER SQLERROR GOTO sql error;
EXEC SQL CONNECT :username IDENTIFIED BY :password;
printf("Connected to Oracle\n");
for (;;)
  printf("Employee Number (0 to end)? ");
  gets(temp);
  emp number = atoi(temp);
  if (emp number == 0)
     EXEC SQL COMMIT WORK RELEASE;
     printf("Exiting program\n");
     break;
/*----*/
  EXEC SQL EXECUTE
  BEGIN
     SELECT job, hiredate, sal
       INTO :job title, :hire date, :salary
       FROM emp
       WHERE empno = :emp number;
  END;
  END-EXEC;
/*----*/
  printf("Number Job Title Hire Date Salary\n");
  printf("-----\n");
  printf("%6d %8.8s %9.9s %6.2f\n",
  emp_number, job_title, hire_date, salary);
}
exit(0);
sql error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SOL ROLLBACK WORK RELEASE;
printf("Processing error\n");
exit(1);
```

Notice that the host variable *emp\_number* is set before the PL/SQL block is entered, and the host variables *job\_title*, *hire\_date*, and *salary* are set inside the block.

## 7.3.2 Complex Example

In the example later, you prompt the user for a bank account number, transaction type, and transaction amount, then debit or credit the account. If the account does not exist, you raise an exception. When the transaction is complete, you display its status.

```
#include <stdio.h>
#include <sqlca.h>
char username[20];
char password[20];
```



```
char status[80];
char temp[32];
int acct_num;
double trans amt;
void sql_error();
main()
  char trans_type;
  strcpy(password, "TIGER");
  strcpy(username, "SCOTT");
  EXEC SQL WHENEVER SQLERROR DO sql error();
  EXEC SQL CONNECT :username IDENTIFIED BY :password;
  printf("Connected to Oracle\n");
  for (;;)
     printf("Account Number (0 to end)? ");
     gets (temp);
     acct num = atoi(temp);
     if(acct num == 0)
        EXEC SQL COMMIT WORK RELEASE;
        printf("Exiting program\n");
        break;
     printf("Transaction Type - D)ebit or C)redit? ");
     gets(temp);
     trans_type = temp[0];
     printf("Transaction Amount? ");
     gets(temp);
     trans_amt = atof(temp);
/*-----*/
     EXEC SQL EXECUTE
     DECLARE
        old bal
                  NUMBER (9, 2);
        err_msg CHAR(70);
        nonexistent EXCEPTION;
     BEGIN
        :trans type := UPPER(:trans type);
        UPDATE accts SET bal = bal + :trans amt
           WHERE acctid = :acct num;
           IF SQL%ROWCOUNT = 0 THEN
                                    -- no rows affected
             RAISE nonexistent;
             :status := 'Credit applied';
           END IF;
        ELSIF :trans_type = 'D' THEN -- debit the account
           SELECT bal INTO old_bal FROM accts
             WHERE acctid = :acct_num;
           IF old bal >= :trans amt THEN     -- enough funds
             UPDATE accts SET bal = bal - :trans amt
```



```
WHERE acctid = :acct num;
              :status := 'Debit applied';
           ELSE
              :status := 'Insufficient funds';
           END IF;
        ELSE
           :status := 'Invalid type: ' || :trans type;
        COMMIT;
     EXCEPTION
        WHEN NO_DATA_FOUND OR nonexistent THEN
          :status := 'Nonexistent account';
        WHEN OTHERS THEN
           err msg := SUBSTR(SQLERRM, 1, 70);
           :status := 'Error: ' || err msg;
     END;
     END-EXEC:
/*----* end PL/SQL block -----*/
     printf("\nStatus: %s\n", status);
  }
  exit(0);
}
void
sql error()
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL ROLLBACK WORK RELEASE;
   printf("Processing error\n");
   exit(1);
```

## 7.3.3 VARCHAR Pseudotype

You can use the VARCHAR datatype to declare variable-length character strings. If the VARCHAR is an input host variable, you must tell Oracle what length to expect. Therefore, set the length component to the actual length of the value stored in the string component.

If the VARCHAR is an output host variable, Oracle automatically sets the length component. However, to use a VARCHAR (as well as CHARZ and STRING) output host variable in your PL/SQL block, you must initialize the length component *before* entering the block. So, set the length component to the declared (maximum) length of the VARCHAR, as shown here:

```
int emp_number;
varchar emp_name[10];
float salary;
...
emp_name.len = 10; /* initialize length component */

EXEC SQL EXECUTE

BEGIN
    SELECT ename, sal INTO :emp_name, :salary
        FROM emp
        WHERE empno = :emp_number;
...
END;
END-EXEC;
...
```



### 7.3.4 Restriction

Do not use C pointer or array syntax in PL/SQL blocks. The PL/SQL compiler does not understand C host-variable expressions and is, therefore, unable to parse them. For example, the following is *invalid*:

To avoid syntax errors, use a place-holder (a temporary variable), to hold the address of the structure field to populate structures as shown in the following *valid* example:

### 7.4 Indicator Variables

PL/SQL does not need indicator variables because it can manipulate NULLs. For example, within PL/SQL, you can use the IS NULL operator to test for NULLs, as follows:

```
IF variable IS NULL THEN ...
```

And, you can use the assignment operator (:=) to assign NULLs, as follows:

```
variable := NULL;
```

However, a host language such as C needs indicator variables because it cannot manipulate NULLs. Embedded PL/SQL meets this need by letting you use indicator variables to

- Accept NULLs input from a host program
- Output NULLs or truncated values to a host program

When used in a PL/SQL block, indicator variables are subject to the following rules:

- You cannot refer to an indicator variable by itself; it must be appended to its associated host variable.
- If you refer to a host variable with its indicator variable, you must always refer to it that way
  in the same block.

In the following example, the indicator variable *ind\_comm* appears with its host variable *commission* in the SELECT statement, so it must appear that way in the IF statement:

```
EXEC SQL EXECUTE

BEGIN

SELECT ename, comm

INTO :emp_name, :commission :ind_comm

FROM emp

WHERE empno = :emp_number;

IF :commission :ind comm IS NULL THEN ...
```



```
END;
END-EXEC;
```

Notice that PL/SQL treats :commission :ind\_comm like any other simple variable. Though you cannot refer directly to an indicator variable inside a PL/SQL block, PL/SQL checks the value of the indicator variable when entering the block and sets the value correctly when exiting the block.

## 7.4.1 NULLs Handling

When entering a block, if an indicator variable has a value of -1, PL/SQL automatically assigns a NULL to the host variable. When exiting the block, if a host variable is NULL, PL/SQL automatically assigns a value of -1 to the indicator variable. In the next example, if *ind\_sal* had a value of -1 before the PL/SQL block was entered, the *salary\_missing* exception is raised. An *exception* is a named error condition.

```
EXEC SQL EXECUTE
BEGIN
    IF :salary :ind_sal IS NULL THEN
    RAISE salary_missing;
END IF;
...
END;
END-EXEC;
```

### 7.4.2 Truncated Values

PL/SQL does not raise an exception when a truncated string value is assigned to a host variable. However, if you use an indicator variable, PL/SQL sets it to the original length of the string. In the following example, the host program will be able to tell, by checking the value of *ind name*, if a truncated value was assigned to *emp\_name*:

```
EXEC SQL EXECUTE
DECLARE
...
new_name CHAR(10);
BEGIN
...
:emp_name:ind_name := new_name;
...
END;
END-EXEC;
```

# 7.5 Host Arrays

You can pass input host arrays and indicator arrays to a PL/SQL block. They can be indexed by a PL/SQL variable of type BINARY\_INTEGER or PLS\_INTEGER; VARCHAR2 key types are not permitted. Normally, the entire host array is passed to PL/SQL, but you can use the ARRAYLEN statement (discussed later) to specify a smaller array dimension.

Furthermore, you can use a procedure call to assign all the values in a host array to rows in a PL/SQL table. Given that the array subscript range is m ... n, the corresponding PL/SQL table index range is always 1 ... n - m + 1. For example, if the array subscript range is 5 ... 10, the corresponding PL/SQL table index range is 1 ... (10 - 5 + 1) or 1 ... 6.

In the following example, you pass an array named *salary* to a PL/SQL block, which uses the array in a function call. The function is named *median* because it finds the middle value in a series of numbers. Its formal parameters include a PL/SQL table named *num\_tab*. The function call assigns all the values in the actual parameter *salary* to rows in the formal parameter *num\_tab*.

```
float salary[100];
/* populate the host array */
EXEC SQL EXECUTE
 DECLARE
   TYPE NumTabTyp IS TABLE OF REAL
       INDEX BY BINARY INTEGER;
   median salary REAL;
   n BINARY INTEGER;
  FUNCTION median (num tab NumTabTyp, n INTEGER)
    RETURN REAL IS
  BEGIN
   -- compute median
 END;
 BEGIN
   n := 100;
   median salary := median(:salary, n);
 END;
END-EXEC;
```

#### Note:

In dynamic SQL Method 4, you cannot bind a host array to a PL/SQL procedure with a parameter of type "table."

You can also use a procedure call to assign all row values in a PL/SQL table to corresponding elements in a host array.

Table 7-1 shows the legal conversions between row values in a PL/SQL table and elements in a host array. For example, a host array of type LONG is compatible with a PL/SQL table of type VARCHAR2, LONG, RAW, or LONG RAW. Notably, it is not compatible with a PL/SQL table of type CHAR.

Table 7-1 Legal Datatype Conversions

PL/SQL Table-> Host Array	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR 2
CHARF	X	-	-	-	-	-	-	-
CHARZ	Χ	-	-	-	-	-	-	-
DATE	-	Χ	-	-	-	-	-	-
DECIMAL	-	-	-	-	Χ	-	-	-
DISPLAY	-	-	-	-	Χ	-	-	-



Table 7-1 (Cont.) Legal Datatype Conversions

PL/SQL Table-> Host Array	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR 2
FLOAT	-	-	-	-	Х	-	-	-
INTEGER	-	-	-	-	Χ	-	-	-
LONG	Χ	-	Χ	-	-	-	-	-
LONG VARCHAR	-	-	Χ	Χ	-	Χ	-	Х
LONG VARRAW	-	-	-	X	-	X	-	-
NUMBER	-	-	-	-	Χ	-	-	-
RAW	-	-	-	Χ	-	Χ	-	-
ROWID	-	-	-	-	-	-	Χ	-
STRING	-	-	Χ	Χ	-	Χ	-	Χ
UNSIGNED	-	-	-	-	Χ	-	-	-
VARCHAR	-	-	Χ	Χ	-	Χ	-	Χ
VARCHAR2	-	-	Χ	Χ	-	Χ	-	Χ
VARNUM	-	-	-	-	Χ	-	-	-
VARRAW	-	-	-	Χ	-	Χ	-	-



The Pro\*C/C++ Precompiler does not check your usage of host arrays. For instance, no index range-checking is done.

#### **Related Topics**

- Using Method 4
- Stored PL/SQL and Java Subprograms

## 7.5.1 ARRAYLEN Statement

Suppose you must pass an input host array to a PL/SQL block for processing. By default, when binding such a host array, the Pro\*C/C++ Precompiler uses its declared dimension. However, you might not want to process the entire array. In that case, you can use the ARRAYLEN statement to specify a smaller array dimension. ARRAYLEN associates the host array with a host variable, which stores the smaller dimension. The statement syntax is

EXEC SQL ARRAYLEN host\_array (dimension) [EXECUTE];

where *dimension* is a 4-byte integer host variable, *not* a literal or expression.

EXECUTE is an optional keyword.

The ARRAYLEN statement must appear along with, but somewhere after, the declarations of host\_array and dimension. You cannot specify an offset into the host array. However, you might

be able to use C features for that purpose. The following example uses ARRAYLEN to override the default dimension of a C host array named *bonus*:

```
float bonus[100];
int dimension;
EXEC SQL ARRAYLEN bonus (dimension);
/* populate the host array */
dimension = 25; /* set smaller array dimension */
EXEC SQL EXECUTE
DECLARE
   TYPE NumTabTyp IS TABLE OF REAL
   INDEX BY BINARY INTEGER;
   median bonus REAL;
   FUNCTION median (num tab NumTabTyp, n INTEGER)
       RETURN REAL IS
 BEGIN
   -- compute median
 END;
 BEGIN
   median bonus := median(:bonus, :dimension);
 END;
END-EXEC;
```

Only 25 array elements are passed to the PL/SQL block because ARRAYLEN reduces the array from 100 to 25 elements. As a result, when the PL/SQL block is sent to Oracle for execution, a much smaller host array is sent along. This saves time and, in a networked environment, reduces network traffic.

# 7.5.2 Optional Keyword EXECUTE

Host arrays used in a dynamic SQL method 2 EXEC SQL EXECUTE statement may have two different interpretations based on the presence or absence of the optional keyword EXECUTE.

By default (if the EXECUTE keyword is absent on an ARRAYLEN statement):

- The host array is considered when determining the number of times a PL/SQL block will be executed. (The minimum array dimension is used.)
- The host array must not be bound to a PL/SQL index table.

If the keyword EXECUTE is present:

- The host array must be bound to an index table.
- The PL/SQL block will be executed one time.
- All host variables specified in the EXEC SQL EXECUTE statement must either
  - Be specified in an ARRAYLEN ... EXECUTE statement
  - Be scalar.

For example, given the following PL/SQL procedure:

```
CREATE OR REPLACE PACKAGE pkg AS

TYPE tab IS TABLE OF NUMBER(5) INDEX BY BINARY_INTEGER;

PROCEDURE proc1 (parm1 tab, parm2 NUMBER, parm3 tab);

END;
```



The following Pro\*C/C++ function demonstrates how host arrays can be used to determine how many times a given PL/SQL block is executed. In this case, the PL/SQL block will be executed 3 times resulting in 3 new rows in the emp table.

```
func1()
{
  int empno_arr[5] = {1111, 2222, 3333, 4444, 5555};
  char *ename_arr[3] = {"MICKEY", "MINNIE", "GOOFY"};
  char *stmt1 = "BEGIN INSERT INTO emp(empno, ename) VALUES :b1, :b2; END;";

  EXEC SQL PREPARE s1 FROM :stmt1;
  EXEC SQL EXECUTE s1 USING :empno_arr, :ename_arr;
}
```

The following Pro\*C/C++ function demonstrates how to bind a host array to a PL/SQL index table through dynamic method 2. Note the presence of the ARRAYLEN...EXECUTE statement for all host arrays specified in the EXEC SQL EXECUTE statement.

```
func2()
{
  int ii = 2;
  int int_tab[3] = {1,2,3};
  int dim = 3;
  EXEC SQL ARRAYLEN int_tab (dim) EXECUTE;
  char *stmt2 = "begin pkg.proc1(:v1, :v2, :v3); end; ";
  EXEC SQL PREPARE s2 FROM :stmt2;
  EXEC SQL EXECUTE s2 USING :int_tab, :ii, :int_tab;
}
```

However the following Pro\*C/C++ function will result in a precompile-time warning because there is no ARRAYLEN...EXECUTE statement for int arr.

```
func3()
{
  int int_arr[3];
  int int_tab[3] = {1,2,3};
  int dim = 3;
  EXEC SQL ARRAYLEN int_tab (dim) EXECUTE;
  char *stmt3 = "begin pkg.proc1(:v1, :v2, :v3); end; ";
  EXEC SQL PREPARE s3 FROM :stmt3;
  EXEC SQL EXECUTE s3 USING :int_tab, :int_arr, :int_tab;
}
```

#### **Related Topics**

- Using Method 2
- Host Arrays

# 7.6 Cursor Usage in Embedded PL/SQL

The maximum number of cursors your program can use simultaneously is determined by the database initialization parameter OPEN\_CURSORS. While executing an embedded PL/SQL block, one cursor, the parent cursor, is associated with the entire block and one cursor, the

child cursor, is associated with each SQL statement in the embedded PL/SQL block. Both parent and child cursors count toward the OPEN CURSORS limit.

The following calculation shows how to determine the maximum number of cursors used. The sum of the cursors used must not exceed OPEN CURSORS.

If your program exceeds the limit imposed by OPEN CURSORS, Oracle gives you an error.

#### **Related Topics**

Embedded PL/SQL Considerations

# 7.7 Stored PL/SQL and Java Subprograms

Unlike anonymous blocks, PL/SQL subprograms (procedures and functions) and Java methods can be compiled separately, stored in an Oracle database, and invoked.

A subprogram explicitly created using an Oracle tool such as SQL\*Plus is called a *stored* subprogram. Once compiled and stored in the data dictionary, it is a database object, which can be re-executed without being recompiled.

When a subprogram within a PL/SQL block or stored procedure is sent to Oracle by your application, it is called an *inline* subprogram. Oracle compiles the inline subprogram and caches it in the System Global Area (SGA) but does not store the source or object code in the data dictionary.

Subprograms defined within a package are considered part of the package, and thus are called *packaged* subprograms. Stored subprograms not defined within a package are called *standalone* subprograms.

#### **Related Topics**

Java Applications on Oracle Database

## 7.7.1 About Creating Stored Subprograms

You can embed the SQL statements CREATE FUNCTION, CREATE PROCEDURE, and CREATE PACKAGE in a host program, as the following example shows:

Notice that the embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement is a hybrid. Like all other embedded CREATE statements, it begins with the keywords EXEC SQL (not EXEC SQL EXECUTE). But, unlike other embedded CREATE statements, it ends with the PL/SQL terminator END-EXEC.

In the example later, you create a package that contains a procedure named *get\_employees*, which fetches a batch of rows from the EMP table. The batch size is determined by the caller of the procedure, which might be another stored subprogram or a client application.

The procedure declares three PL/SQL tables as OUT formal parameters, then fetches a batch of employee data into the PL/SQL tables. The matching actual parameters are host arrays. When the procedure finishes, it automatically assigns all row values in the PL/SQL tables to corresponding elements in the host arrays.

```
EXEC SQL CREATE OR REPLACE PACKAGE emp actions AS
   TYPE CharArrayTyp IS TABLE OF VARCHAR2(10)
      INDEX BY BINARY INTEGER;
   TYPE NumArrayTyp IS TABLE OF FLOAT
      INDEX BY BINARY INTEGER;
 PROCEDURE get employees (
   batch size IN INTEGER,
   found IN OUT INTEGER,
   done fetch OUT INTEGER,
   emp name OUT CharArrayTyp,
   job title OUT CharArrayTyp,
   salary OUT NumArrayTyp);
 END emp actions;
END-EXEC;
EXEC SQL CREATE OR REPLACE PACKAGE BODY emp actions AS
   CURSOR get emp (dept number IN INTEGER) IS
       SELECT ename, job, sal FROM emp
          WHERE deptno = dept number;
 PROCEDURE get employees(
   found IN OUT INTEGER,
   done fetch OUT INTEGER,
   emp name OUT CharArrayTyp,
   job title OUT CharArrayTyp,
   salary OUT NumArrayTyp) IS
   IF NOT get emp%ISOPEN THEN
       OPEN get emp(dept number);
   END IF;
   done fetch := 0;
   found := 0;
   FOR i IN 1..batch size LOOP
       FETCH get emp INTO emp name(i),
       job title(i), salary(i);
       IF get emp%NOTFOUND THEN
          CLOSE get emp;
           done fetch := 1;
          EXIT;
       ELSE
           found := found + 1;
       END IF;
   END LOOP;
 END get employees;
```

```
END emp_actions;
END-EXEC;
```

You specify the REPLACE clause in the CREATE statement to redefine an existing package without having to drop the package, re-create it, and re-grant privileges on it. For the full syntax of the CREATE statement see SQL Statements: COMMIT to CREATE JAVA.

If an embedded CREATE {FUNCTION | PROCEDURE | PACKAGE} statement fails, Oracle generates a warning, not an error.

## 7.7.2 About Calling a Stored PL/SQL or Java Subprogram

To call a stored subprogram from your host program, you can use either an anonymous PL/SQL block, or the CALL embedded SQL statement.

### 7.7.2.1 Anonymous PL/SQL Block

In the following example, you call a standalone procedure named raise salary:

```
EXEC SQL EXECUTE
  BEGIN
    raise_salary(:emp_id, :increase);
  END;
END-EXEC;
```

Notice that stored subprograms can take parameters. In this example, the actual parameters *emp\_id* and *increase* are C host variables.

In the next example, the procedure *raise\_salary* is stored in a package named *emp\_actions*, so you must use dot notation to fully qualify the procedure call:

```
EXEC SQL EXECUTE
BEGIN
    emp_actions.raise_salary(:emp_id, :increase);
END;
END-EXEC;
```

An actual IN parameter can be a literal, scalar host variable, host array, PL/SQL constant or variable, PL/SQL table, PL/SQL user-defined record, procedure call, or expression. However, an actual OUT parameter cannot be a literal, procedure call, or expression.

You must use precompiler option SQLCHECK=SEMANTICS with an embedded PL/SQL block.

In the following example, three of the formal parameters are PL/SQL tables, and the corresponding actual parameters are host arrays. The program calls the stored procedure get\_employees repeatedly, displaying each batch of employee data, until no more data is found. This program is available on-line in the demo directory, in the file sample9.pc. A SQL script to create the CALLDEMO stored package is available in the file calldemo.sql.



```
retrieved. GET EMPLOYEES sets the done flag to indicate "no
more data."
#include <stdio.h>
#include <string.h>
EXEC SQL INCLUDE sqlca.h;
typedef char asciz[20];
typedef char vc2 arr[11];
EXEC SQL BEGIN DECLARE SECTION;
/* User-defined type for null-terminated strings */
EXEC SQL TYPE asciz IS STRING(20) REFERENCE;
/* User-defined type for a VARCHAR array element. */
EXEC SQL TYPE vc2 arr IS VARCHAR2(11) REFERENCE;
asciz
        username;
asciz password;
      dept no;
                            /* which department to query? */
vc2 arr emp name[10];
                               /* array of returned names */
vc2 arr job[10];
float salary[10];
       done_flag;
int
int
        array size;
     num_ret;
int
                                /* number of rows returned */
EXEC SQL END DECLARE SECTION;
int
      SQLCODE;
void print_rows();
                       /* produces program output */
                           /* handles unrecoverable errors */
void sql error();
main()
  char temp_buf[32];
/* Connect to ORACLE. */
  EXEC SQL WHENEVER SQLERROR DO sql_error();
  strcpy(username, "scott");
  strcpy(password, "tiger");
  EXEC SQL CONNECT :username IDENTIFIED BY :password;
  printf("\nConnected to ORACLE as user: %s\n\n", username);
  printf("Enter department number: ");
  gets(temp buf);
  dept_no = atoi(temp_buf);/* Print column headers. */
  printf("\n\n");
  printf("%-10.10s%-10.10s%s\n", "Employee", "Job", "Salary");
  printf("%-10.10s%-10.10s%s\n", "-----", "---", "----");
/* Set the array size. */
  array_size = 10;
  done flag = 0;
  num ret = 0;
```

```
/* Array fetch loop.
 * The loop continues until the OUT parameter done_flag is set.
 ^{\star} \, Pass in the department number, and the array size--
 * get names, jobs, and salaries back.
   for (;;)
      EXEC SQL EXECUTE
         BEGIN calldemo.get_employees
            (:dept_no, :array_size, :num_ret, :done_flag,
             :emp_name, :job, :salary);
         END;
      END-EXEC;
      print rows(num ret);
      if (done flag)
        break;
/* Disconnect from the database. */
   EXEC SQL COMMIT WORK RELEASE;
   exit(0);
}
void
print rows(n)
int n;
   int i;
    if (n == 0)
        printf("No rows retrieved.\n");
        return;
    for (i = 0; i < n; i++)
       printf("%10.10s%10.10s%6.2f\n",
               emp name[i], job[i], salary[i]);
/* Handle errors. Exit on any error. */
void
sql_error()
   char msg[512];
   int buf_len, msg_len;
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   buf len = sizeof(msg);
   sqlglm(msg, &buf len, &msg len);
   printf("\nORACLE error detected:");
   printf("\n%.*s \n", msg len, msg);
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(1);
}
```

Remember, the datatype of each actual parameter must be convertible to the datatype of its corresponding formal parameter. Also, before a stored procedure is exited, all OUT formal parameters must be assigned values. Otherwise, the values of corresponding actual parameters are indeterminate.

SQLCHECK=SEMANTICS is required when using an anonymous PL/SQL block.

#### **Related Topics**

About Creating Stored Subprograms

### 7.7.2.2 Remote Access

PL/SQL lets you access remote databases using *database links*. Typically, database links are established by your DBA and stored in the Oracle data dictionary. A database link tells Oracle where the remote database is located, the path to it, and what Oracle username and password to use. In the following example, you use the database link *dallas* to call the *raise\_salary* procedure:

```
EXEC SQL EXECUTE
BEGIN
    raise_salary@dallas(:emp_id, :increase);
END;
END-EXEC;
```

You can create synonyms to provide location transparency for remote subprograms, as the following example shows:

```
CREATE PUBLIC SYNONYM raise_salary
FOR raise salary@dallas;
```

### 7.7.2.3 The CALL Statement

The concepts presented earlier for the embedded PL/SQL block also hold true for the CALL statement. The CALL embedded SQL statement has the form:

```
CALL [schema.] [package.]stored_proc[@db_link] (argl, ...)
[INTO:ret_var [[INDICATOR]:ret_ind]];

where
schema
the schema containing the procedure
package
the package containing the procedure
stored_proc
is the Java or PL/SQL stored procedure to be called
db_link
is the optional remote database link
arg1...
is the list of arguments (variables, literals, or expressions) passed,
ret_var
```



is the optional host variable which receives the result

ind\_var

the optional indicator variable for ret\_var.

You can use either SQLCHECK=SYNTAX, or SEMANTICS with the CALL statement.

### 7.7.2.4 CALL Example

If you have created a PL/SQL function fact (stored in the package mathpkg) that takes an integer as input and returns its factorial in an integer:

```
EXEC SQL CREATE OR REPLACE PACKAGE BODY mathpkg as
function fact(n IN INTEGER) RETURN INTEGER AS
BEGIN
    IF (n <= 0) then return 1;
    ELSE return n * fact(n - 1);
    END IF;
    END fact;
END mathpkge;
END-EXEC.</pre>
```

then to use fact in a Pro\*C/C++ application using the CALL statement:

```
int num, fact;
...
EXEC SQL CALL mathpkge.fact(:num) INTO :fact;
```

### See Also:

- "CALL (Executable Embedded SQL)" for more information about the CALL statement.
- Oracle Database Advanced Application Developer's Guide, for a complete explanation of passing arguments and other issues.

# 7.7.3 About Getting Information about Stored Subprograms



The Logon Data Area (LDA) is *no longer supported* in Oracle. The ability to embed OCI Release 7 calls in your Pro\*C/C++ program will be phased out by the next major Oracle release.

Datatypes and Host Variables described how to embed OCI calls in your host program. After calling the library routine SQLLDA to set up the LDA, use the OCI call *odessp* to get useful information about a stored subprogram. When you call *odessp*, you must pass it a valid LDA and the name of the subprogram. For packaged subprograms, you must also pass the name of the package. *odessp* returns information about each subprogram parameter such as its datatype, size, position, and so on.

You can also use the DESCRIBE\_PROCEDURE stored procedure, in the DBMS\_DESCRIBE package.

## 7.8 External Procedures

PL/SQL can call C functions which are external procedures. External procedures (also known as external procedures) are stored in dynamic link libraries (DLL) or, for example, in .so libraries under Solaris.

If the external procedure executes on the server-side, it can call back into the server to execute SQL and PL/SQL in the same transaction. External procedures on the server execute faster than on the client and can interface the database server with external systems and data sources.

In order to execute a server-side external C function, the REGISTER CONNECT embedded SQL statement must be used inside that function. The syntax of the statement is:

```
EXEC SQL REGISTER CONNECT USING :epctx [RETURNING :host_context] ;
```

where epctx is the external procedure context (of type pointer to OCIExtProcContext). epctx is passed to the procedure by PL/SQL.

host\_context is a runtime context returned by the external procedure. Currently, it is the default (global) context.

The REGISTER CONNECT statement will return the set of OCI handles (OCIEnv, OCISvcCtx, and OCIError) that are associated with the current Oracle connection and transaction. These handles are then used to define the Pro\*C/C++ default unnamed connection for the global SQLLIB runtime context. The REGISTER CONNECT statement is therefore used *instead* of a CONNECT statement.

Subsequent embedded SQL statements will use this set of OCI handles. They execute against the global SQLLIB runtime context and the unnamed connection, even those that are in separately precompiled units. Uncommitted changes are not seen. In the future, a (nondefault) runtime context can be returned in the optional RETURNING clause.

There cannot already be any active default connection for the global runtime context. A runtime error is returned if you try to use REGISTER CONNECT when a connection already exists.

In real-world cases, the external procedure should be one that you can reuse from many different applications.

#### **Related Topics**

Oracle Call Interface Programmer's Guide

### 7.8.1 Restrictions on External Procedures

Follow these rules for external procedures:

- External procedures can only be in C. C++ external procedures are not supported.
- When you are connected to the external procedure context, any additional connection is not permitted and results in a runtime error.
- Multithreaded external procedures are not supported. Executing an EXEC SQL ENABLE THREADS statement is not permitted and will return a runtime error. Pro\*C/C++ does support multithreading in an application not using the external procedure method we are describing.



- You cannot use DDL statements. They result in runtime errors.
- You cannot use transaction control statements, such as EXEC SQL COMMIT, and EXEC SQL ROLLBACK.
- You cannot use object navigation statements such as EXEC SQL OBJECT ....
- You cannot use polling EXEC SQL LOB statements.
- You cannot use EXEC TOOLS statements. They will result in runtime errors.

## 7.8.2 About Creating the External Procedure

Here is a simple example to create the external procedure extp1.

To store an external C procedure, compile and link the code to a library such as a DLL.

Reference to NT removed user comment 9561

Then use the following SQL command once to register the external procedure extp1:

```
CREATE OR REPLACE PROCEDURE extp1
AS EXTERNAL NAME "extp1"
LIBRARY mylib
WITH CONTEXT
PARAMETERS (CONTEXT);
```

Where *mylib* is the name of the library storing procedure *extp1*. WITH CONTEXT means to implicitly call this procedure with argument of type <code>OCIExtProcContext\*</code>. The context is omitted in your call, but is passed to the procedure anyway. The keyword CONTEXT appears in the CREATE statement, however, as a place marker.

This context parameter is the one referenced in the EXEC SQL REGISTER CONNECT statement inside extp1.

The external procedure is called from SQL\*Plus this way:

#### **Related Topics**

External Subprograms

## 7.8.3 SQLExtProcError()

The SQLLIB function SQLExtProcError() provides the ability to return control to PL/SQL when an error occurs in an external C procedure. The function and its arguments are:

```
SQLExtProcError (ctx, msg, msglen)
```

where:

```
ctx (IN) sql context *
```

This is the target SQLLIB runtime context of the REGISTER CONNECT statement, which has to be executed before this function is invoked. Only the global runtime context is supported now.

```
msg (OUT) char *
```

The text of the error message.

```
msglen (OUT) size_t
```

The length in bytes of the message.

SQLLIB calls the OCI service function OCIExtProcRaiseExcpWithMsg when this function is executed.

The message is from the structure sqlerrm in the SQLCA.

Here is an example showing use of SQLExtProcError():

```
void extp1 (epctx)
OCIExtProcContext *epctx;
{
   char name[15];
   EXEC SQL REGISTER CONNECT USING :epctx;
   EXEC SQL WHENEVER SQLERROR goto err;
   EXEC SQL SELECT ename INTO :name FROM emp WHERE smpno = 9999;
   return;
err:
   SQLExtProcError (SQL_SINGLE_RCTX, sqlca.sqlerrm.sqlerrmc,
        sqlca.sqlerrm.sqlerrml);
   printf("\n%*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
   return;
}
```

#### **Related Topics**

SQLCA Structure

# 7.9 About Using Dynamic SQL

Recall that the precompiler treats an entire PL/SQL block like a single SQL statement. Therefore, you can store a PL/SQL block in a string host variable. Then, if the block contains no host variables, you can use dynamic SQL Method 1 to EXECUTE the PL/SQL string. Or, if the block contains a known number of host variables, you can use dynamic SQL Method 2 to PREPARE and EXECUTE the PL/SQL string. If the block contains an unknown number of host variables, you must use dynamic SQL Method 4.

Note:

In dynamic SQL Method 4, you cannot bind a host array to a PL/SQL procedure with a parameter of type "table."

### **Related Topics**

- ANSI Dynamic SQL
- Oracle Dynamic SQL
- Oracle Dynamic SQL: Method 4
- Using Method 4



# **Host Arrays**

This chapter looks at using arrays to simplify coding and improve program performance. You will learn how to manipulate Oracle data using arrays, how to operate on all the elements of an array with a single SQL statement, and how to limit the number of array elements processed. The chapter contains the following topics:

- · Why Use Arrays?
- About Declaring Host Arrays
- About Using Arrays in SQL Statements
- About Selecting into Arrays
- About Inserting with Arrays
- About Updating with Arrays
- About Deleting with Arrays
- · About Using the FOR Clause
- About Using the WHERE Clause
- Arrays of Structs
- About Mimicking CURRENT OF
- About Using sqlca.sqlerrd[2]
- About Using Additional Array Insert/Select Syntax
- About Using Implicit Buffered Insert

## 8.1 Why Use Arrays?

Arrays reduce programming time and result in improved performance.

With arrays, you manipulate an entire array with a single SQL statement. Thus, Oracle communication overhead is reduced markedly, especially in a networked environment. A major portion of runtime is spent on network round trips between the client program and the server database. Arrays reduce the round trips.

For example, suppose you want to insert information about 300 employees into the EMP table. Without arrays your program must do 300 individual INSERTs—one for each employee. With arrays, only one INSERT needs to be done.

# 8.2 About Declaring Host Arrays

The following example declares three host arrays, each with a maximum of 50 elements:

```
char emp_name[50][10];
int emp_number[50];
float salary[50];
```

Arrays of VARCHARs are also allowed. The following declaration is a valid host language declaration:

```
VARCHAR v_array[10][30];
```

## 8.2.1 Restrictions (Declaring Host Arrays)

You cannot declare host arrays of pointers, except for object types.

Except for character arrays (strings), host arrays that might be referenced in a SQL statement are limited to one dimension. So, the two-dimensional array declared in the following example is *invalid*:

```
int hi_lo_scores[25][25];  /* not allowed */
```

## 8.2.2 Maximum Size of Arrays

The maximum number of array elements in a SQL statement that is accessible in one fetch is 4GB (or possibly greater, depending on the platform and the available memory). If you try to access a number that exceeds the maximum, you get a "parameter out of range" runtime error. If the statement is an anonymous PL/SQL block, the number of array elements accessible is limited to 4GB divided by the size of the datatype.

# 8.3 About Using Arrays in SQL Statements

You can use host arrays as input variables in the INSERT, UPDATE, and DELETE statements and as output variables in the INTO clause of SELECT and FETCH statements.

The embedded SQL syntax used for host arrays and simple host variables is nearly the same. One difference is the optional FOR clause, which lets you control array processing. Also, there are restrictions on mixing host arrays and simple host variables in a SQL statement.

The following sections illustrate the use of host arrays in data manipulation statements.

## 8.3.1 About Referencing Host Arrays

If you use multiple host arrays in a single SQL statement, their number of elements should be the same. Otherwise, an "array size mismatch" warning message is issued at precompile time. If you ignore this warning, the precompiler uses the *smallest* number of elements for the SQL operation.

In this example, only 25 rows are Inserted

```
int emp_number[50];
char emp_name[50][10];
int dept_number[25];
/* Populate host arrays here. */

EXEC SQL INSERT INTO emp (empno, ename, deptno)
    VALUES (:emp number, :emp name, :dept number);
```

It is possible to subscript host arrays in SQL statements, and use them in a loop to INSERT or fetch data. For example, you could INSERT every fifth element in an array using a loop such as:

```
for (i = 0; i < 50; i += 5)
    EXEC SQL INSERT INTO emp (empno, deptno)
    VALUES (:emp_number[i], :dept_number[i]);</pre>
```

However, if the array elements that you need to process are contiguous, you should not process host arrays in a loop. Simply use the non-scripted array names in your SQL statement. Oracle treats a SQL statement containing host arrays of element number n like the same statement executed n times with n different scalar variables.

## 8.3.2 About Using Indicator Arrays

You can use indicator arrays to assign NULLs to input host arrays, and to detect NULL or truncated values (character columns only) in output host arrays. The following example shows how to INSERT with indicator arrays:

```
int emp_number[50];
int dept_number[50];
float commission[50];
short comm_ind[50]; /* indicator array */

/* Populate the host and indicator arrays. To insert a null into the comm column, assign -1 to the appropriate element in the indicator array. */
    EXEC SQL INSERT INTO emp (empno, deptno, comm)
        VALUES (:emp_number, :dept_number, :commission INDICATOR :comm ind);
```

## 8.3.3 Oracle Restrictions (for Host Arrays)

Mixing scalar host variables with host arrays in the VALUES, SET, INTO, or WHERE clause is *not* allowed. If any of the host variables is an array, all must be arrays.

You cannot use host arrays with the CURRENT OF clause in an UPDATE or DELETE statement.

## 8.3.4 ANSI Restriction and Requirements

The array interface is an Oracle extension to the ANSI/ISO embedded SQL standard. However, when you precompile with MODE=ANSI, array SELECTs and FETCHes are still allowed. The use of arrays can be flagged using the FIPS flagger precompiler option, if desired.

When doing array SELECTs and FETCHes, always use indicator arrays. That way, you can test for NULLs in the associated output host array.

If DBMS=V7 or DBMS=v8 and you SELECT or FETCH a NULL column value into a host array that is not associated with an indicator array, then Oracle stops processing, sets sqlerrd[2] to the number of rows processed, and returns an error message. When DBMS=V7 or DBMS=v8, Oracle does not consider truncation to be an error.

Also, if your SELECT or FETCH results in any warning such as ORA-24347 due to usage of NULL, and if any column does not have an indicator array, Oracle stops processing.



Use indicator variables for all the columns in the SELECT or FETCH. If all columns do not have indicators, then the precompiler option <code>unsafe\_null=yes</code> can be used as an alternative.

# 8.4 About Selecting into Arrays

You can use host arrays as output variables in the SELECT statement. If you know the maximum number of rows the SELECT will return, simply declare the host arrays with that number of elements. In the following example, you select directly into three host arrays. Knowing the SELECT will return no more than 50 rows, you declare the arrays with 50 elements:

```
char emp_name[50][20];
int emp_number[50];
float salary[50];

EXEC SQL SELECT ENAME, EMPNO, SAL
    INTO :emp_name, :emp_number, :salary
    FROM EMP
    WHERE SAL > 1000;
```

In the preceding example, the SELECT statement returns up to 50 rows. If there are fewer than 50 eligible rows or you want to retrieve only 50 rows, this method will suffice. However, if there are more than 50 eligible rows, you cannot retrieve all of them this way. If you reexecute the SELECT statement, it just returns the first 50 rows again, even if more are eligible. You must either declare a larger array or declare a cursor for use with the FETCH statement.

If a SELECT INTO statement returns more rows than the number of elements you declared, Oracle issues an error message unless you specify SELECT\_ERROR=NO.

### ✓ See Also:

"Precompiler Options " for more information about the SELECT\_ERROR option.

### 8.4.1 Cursor Fetches

If you do not know the maximum number of rows a SELECT will return, you can declare and open a cursor, then fetch from it in "batches."

Batch fetches within a loop let you retrieve a large number of rows with ease. Each FETCH returns the next batch of rows from the current active set. In the following example, you fetch in 20-row batches:



```
EXEC SQL OPEN emp_cursor;

EXEC SQL WHENEVER NOT FOUND do break;
for (;;)
{
    EXEC SQL FETCH emp_cursor
        INTO :emp_number, :salary;
    /* process batch of rows */
    ...
}
...
```

Do not forget to check how many rows were actually returned in the last fetch, and process them

#### **Related Topics**

Number of Rows Fetched

## 8.4.2 About Using sqlca.sqlerrd[2]

For INSERT, UPDATE, DELETE, and SELECT INTO statements, sqlca.sqlerrd[2] records the number of rows processed. For FETCH statements, it records the cumulative sum of rows processed.

When using host arrays with FETCH, to find the number of rows returned by the most recent iteration, subtract the current value of sqlca.sqlerrd[2] from its previous value (stored in another variable). In the following example, you determine the number of rows returned by the most recent fetch:

```
int emp number[100];
char emp name[100][20];
int rows_to_fetch, rows_before, rows_this_time;
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT empno, ename
    FROM emp
    WHERE deptno = 30;
EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND CONTINUE;
/* initialize loop variables */
rows to fetch = 20; /* number of rows in each "batch" */
rows before = 0;
                     /* previous value of sqlerrd[2] */
rows this time = 20;
while (rows_this_time == rows_to_fetch)
    EXEC SQL FOR :rows_to_fetch
    FETCH emp_cursor
       INTO :emp number, :emp name;
    rows this time = sqlca.sqlerrd[2] - rows before;
    rows before = sqlca.sqlerrd[2];
```

sqlca.sqlerrd[2] is also useful when an error occurs during an array operation. Processing stops at the row that caused the error, so sqlerrd[2] gives the number of rows processed successfully.

### 8.4.3 Number of Rows Fetched

Each FETCH returns, at most, the total number of rows in the array. Fewer rows are returned in the following cases:

- The end of the active set is reached. The "no data found" Oracle error code is returned to SQLCODE in the SQLCA. For example, this happens if you fetch into an array of number of elements 100 but only 20 rows are returned.
- Fewer than a full batch of rows remain to be fetched. For example, this happens if you fetch 70 rows into an array of 20 number elements because after the third FETCH, only 10 rows remain to be fetched.
- An error is detected while processing a row. The FETCH fails and the applicable Oracle error code is returned to SQLCODE.

The cumulative number of rows returned can be found in the third element of *sqlerrd* in the SQLCA, called sqlerrd[2] in this guide. This applies to each open cursor. In the following example, notice how the status of each cursor is maintained separately:

```
EXEC SQL OPEN cursor1;
EXEC SQL OPEN cursor2;
EXEC SQL FETCH cursor1 INTO :array_of_20;
/* now running total in sqlerrd[2] is 20 */
EXEC SQL FETCH cursor2 INTO :array_of_30;
/* now running total in sqlerrd[2] is 30, not 50 */
EXEC SQL FETCH cursor1 INTO :array_of_20;
/* now running total in sqlerrd[2] is 40 (20 + 20) */
EXEC SQL FETCH cursor2 INTO :array_of_30;
/* now running total in sqlerrd[2] is 60 (30 + 30) */
```

### 8.4.4 Scrollable Cursor Fetches

You can also use host arrays with scrollable cursors. With scrollable cursors sqlca.sqlerrd[2] represents the maximum (absolute) row number processed. Since an application can arbitrarily position the fetches in scrollable mode, it need not be the total number of rows processed.

While using host arrays with the FETCH statement in scrollable mode, you cannot subtract the current value of sqlca.sqlerrd[2] from its previous value to find the number of rows returned by the most recent iteration. The application program determines the total number of rows in the result set by executing a FETCH LAST. The value of sqlca.sqlerrd[2] provides the total number of rows in the result set.

#### **Related Topics**

Sample Program: Host Arrays Using Scrollable Cursor

### 8.4.5 Sample Program 3: Host Arrays

The demonstration program in this section shows how you can use host arrays when writing a query in Pro\*C/C++. Pay particular attention to the use of the "rows processed count" in the SQLCA (sqlca.sqlerrd[2]). This program is available on-line in the file sample3.pc in your demo directory.

```
/*
    * sample3.pc
    * Host Arrays
```



```
* This program connects to ORACLE, declares and opens a cursor,
 * fetches in batches using arrays, and prints the results using
 * the function print_rows().
#include <stdio.h>
#include <string.h>
#include <sqlca.h>
#define NAME_LENGTH 20
#define ARRAY LENGTH 5
/* Another way to connect. */
char *username = "SCOTT";
char *password = "TIGER";
/* Declare a host structure tag. */
struct
         emp number[ARRAY LENGTH];
   char emp name[ARRAY LENGTH][NAME LENGTH];
   float salary[ARRAY LENGTH];
} emp rec;
/* Declare this program's functions. */
main()
                           /* number of rows returned */
   int num ret;
/* Connect to ORACLE. */
   EXEC SQL WHENEVER SQLERROR DO sql error("Connect error:");
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
   printf("\nConnected to ORACLE as user: %s\n", username);
   EXEC SQL WHENEVER SQLERROR DO sql error("Oracle error:");
/* Declare a cursor for the FETCH. */
   EXEC SQL DECLARE c1 CURSOR FOR
       SELECT empno, ename, sal FROM emp;
   EXEC SOL OPEN c1;
/* Initialize the number of rows. */
   num ret = 0;
/\!\!^{\star} Array fetch loop - ends when NOT FOUND becomes true. ^{\star}/\!\!^{}
   EXEC SQL WHENEVER NOT FOUND DO break;
   for (;;)
       EXEC SQL FETCH c1 INTO :emp rec;
/* Print however many rows were returned. */
       print_rows(sqlca.sqlerrd[2] - num_ret);
       }
```

```
/* Print remaining rows from last fetch, if any. */
    if ((sqlca.sqlerrd[2] - num ret) > 0)
        print_rows(sqlca.sqlerrd[2] - num_ret);
    EXEC SQL CLOSE c1;
    printf("\nAu revoir.\n\n\n");
/* Disconnect from the database. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}
void
print_rows(n)
int n;
    int i;
   printf("\nNumber Employee
   printf("\n-----
                                       ----\n");
    for (i = 0; i < n; i++)
        printf("%-9d%-15.15s%9.2f\n", emp_rec.emp_number[i],
              emp_rec.emp_name[i], emp_rec.salary[i]);
sql error(msg)
char *msg;
   EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\n%s", msg);
    printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
```

### See Also:

" Handling Runtime Errors" for more information about SQLCA

# 8.4.6 Sample Program: Host Arrays Using Scrollable Cursor

This program describes how to use host arrays with scrollable cursors. This program is available on-line in the file scdemo2.pc in your demo directory.

### Note:

Note that we do a FETCH LAST to determine the number of rows in the result set.

### 8.4.6.1 scdemo2.pc

```
A Sample program to demonstrate the use of scrollable
   cursors with host arrays.
 * This program uses the hr/hr schema.Make sure
 ^{\star} that this schema exists before executing this program
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#define ARRAY_LENGTH 4
/* user and passwd */
char *username = "hr";
char *password = "hr";
/* Declare a host structure tag. */
struct emp rec array
   int
         emp_number;
   char emp_name[20];
   float salary;
} emp rec[ARRAY LENGTH];
/* Print the result of the query */
void print rows()
   int i;
    for (i=0; i<ARRAY LENGTH; i++)</pre>
       emp_rec[i].emp_name, emp_rec[i].salary);
/* Oracle error handler */
void sql error(char *msg)
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   printf("\n%s", msg);
   printf("\n% .70s \n", sqlca.sqlerrm.sqlerrmc);
   EXEC SQL ROLLBACK WORK RELEASE;
   exit(EXIT FAILURE);
void main()
   int noOfRows; /* Number of rows in the result set */
   /* Error handler */
```



```
EXEC SQL WHENEVER SQLERROR DO sql error("Connect error:");
   /* Connect to the data base */
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
    /* Error handle */
   EXEC SQL WHENEVER SQLERROR DO sql error("Oracle error:");
    /* declare the cursor in scrollable mode */
   EXEC SQL DECLARE c1 SCROLL CURSOR FOR
       SELECT employee id, first name, salary FROM employees;
   EXEC SQL OPEN c1;
   EXEC SQL WHENEVER SQLERROR DO sql error("Fetch Error:");
   /* This is a dummy fetch to find out the number of rows
      in the result set */
   EXEC SQL FETCH LAST c1 INTO :emp rec;
   /* The number of rows in the result set is given by
      the value of sqlca.sqlerrd[2] */
   noOfRows = sqlca. sqlerrd[2];
   printf("Total number of rows in the result set %d:\n",
            noOfRows);
   /* Fetch the first ARRAY LENGTH number of rows */
   EXEC SQL FETCH FIRST c1 INTO :emp rec;
   printf("************* DEFAULT : \n");
   print rows();
    /* Fetch the next set of ARRAY LENGTH rows */
   EXEC SQL FETCH NEXT c1 INTO :emp rec;
   printf("************************** NEXT : \n");
   print rows();
   /* Fetch a set of ARRAY LENGTH rows from the 3rd row onwards */
   EXEC SQL FETCH ABSOLUTE 3 c1 INTO :emp_rec;
   printf("************* ABSOLUTE 3 : \n");
   print rows();
   /* Fetch the current ARRAY LENGTH set of rows */
   EXEC SOL FETCH CURRENT c1 INTO :emp rec;
   printf("***************** CURRENT : \n");
   print rows();
   /* Fetch a set of ARRAY LENGTH rows from the 2nd offset
      from the current cursor position */
   EXEC SQL FETCH RELATIVE 2 c1 INTO :emp_rec;
   printf("********** RELATIVE 2 : \n");
   print rows();
   /* Again Fetch the first ARRAY LENGTH number of rows */
   EXEC SQL FETCH ABSOLUTE 0 c1 INTO :emp rec;
   printf("************** ABSOLUTE 0 : \n");
   print rows();
    /* close the cursor */
   EXEC SQL CLOSE c1;
/* Disconnect from the database. */
```

```
EXEC SQL COMMIT WORK RELEASE;
exit(EXIT_SUCCESS);
```

## 8.4.7 Host Array Restrictions

Using host arrays in the WHERE clause of a SELECT statement is *not* allowed except in a subquery. For an example, see "About Using the WHERE Clause".

Also, you cannot mix simple host variables with host arrays in the INTO clause of a SELECT or FETCH statement. If any of the host variables is an array, all must be arrays.

Table 8-1 shows which uses of host arrays are valid in a SELECT INTO statement:

Table 8-1 Valid Host Arrays for SELECT INTO

INTO Clause	WHERE Clause	Valid?
array	array	no
scalar	scalar	yes
array	scalar	yes
scalar	array	no

## 8.4.8 About Fetching NULLs

When doing array SELECTs and FETCHes, always use indicator arrays. That way, you can test for NULLs in the associated output host array.

When DBMS = V7 or DBMS=v8, if you SELECT or FETCH a NULL column value into a host array that is not associated with an indicator array, Oracle stops processing, sets sqlerrd[2] to the number of rows processed, and issues an error message.

Also, if your SELECT or FETCH results in any warning such as *ORA-24347* due to usage of NULL, and if any column does not have an indicator array, Oracle stops processing. Use indicator variables in all the columns in the SELECT or FETCH.If all columns do not have indicators, the precompiler option unsafe null=yes could be used as an alternative.

## 8.4.9 About Fetching Truncated Values

When DBMS=V7, truncation results in a warning message, but Oracle continues processing.

Again, when doing array SELECTs and FETCHes, always use indicator arrays. That way, if Oracle assigns one or more truncated column values to an output host array, you can find the original lengths of the column values in the associated indicator array.

# 8.5 About Inserting with Arrays

You can use host arrays as input variables in an INSERT statement. Just make sure your program populates the arrays with data before executing the INSERT statement.

If some elements in the arrays are irrelevant, you can use the FOR clause to control the number of rows inserted.

An example of inserting with host arrays follows:

The cumulative number of rows inserted can be found in the rows-processed count, sqlca.sqlerrd[2].

In the following example, the INSERT is done one row at a time. This is much less efficient than the previous example, since a call to the server must be made for each row inserted.

```
for (i = 0; i < array_size; i++)
    EXEC SQL INSERT INTO emp (ename, empno, sal)
    VALUES (:emp name[i], :emp number[i], :salary[i]);</pre>
```

#### **Related Topics**

About Using the FOR Clause

### 8.5.1 About Inserting with Arrays Restrictions

You cannot use an array of pointers in the VALUES clause of an INSERT statement; all array elements must be data items.

Mixing scalar host variables with host arrays in the VALUES clause of an INSERT statement is *not* allowed. If any of the host variables is an array, all must be arrays.

# 8.6 About Updating with Arrays

You can also use host arrays as input variables in an UPDATE statement, as the following example shows:

```
int emp_number[50];
float salary[50];
/* populate the host arrays */
EXEC SQL UPDATE emp SET sal = :salary
    WHERE EMPNO = :emp_number;
```

The cumulative number of rows updated can be found in sqlerrd[2]. This number does *not* include rows processed by an update cascade.

If some elements in the arrays are irrelevant, you can use the embedded SQL FOR clause to limit the number of rows updated.

The last example showed a typical update using a unique key (EMP\_NUMBER). Each array element qualified just one row for updating. In the following example, each array element qualifies multiple rows:

```
char job_title [10][20];
float commission[10];
...

EXEC SQL UPDATE emp SET comm = :commission
   WHERE job = :job title;
```



## 8.6.1 About Updating with Arrays Restrictions

Mixing simple host variables with host arrays in the SET or WHERE clause of an UPDATE statement is *not* recommended. If any of the host variables is an array, all should be arrays. Furthermore, if you use a host array in the SET clause, use one of equal number of elements in the WHERE clause.

You cannot use host arrays with the CURRENT OF clause in an UPDATE statement.

Table 8-2 shows which uses of host arrays are valid in an UPDATE statement:

Table 8-2 Host Arrays Valid in an UPDATE

SET Clause	WHERE Clause	Valid?
array	array	yes
scalar	scalar	yes
array	scalar	no
scalar	array	no

#### **Related Topics**

About Mimicking CURRENT OF

# 8.7 About Deleting with Arrays

You can also use host arrays as input variables in a DELETE statement. It is like executing the DELETE statement repeatedly using successive elements of the host array in the WHERE clause. Thus, each execution might delete zero, one, or more rows from the table.

An example of deleting with host arrays follows:

```
int emp_number[50];

/* populate the host array */
...

EXEC SQL DELETE FROM emp
   WHERE empno = :emp number;
```

The cumulative number of rows deleted can be found in sqlerrd[2]. The number does *not* include rows processed by a delete cascade.

The last example showed a typical delete using a unique key (EMP\_NUMBER). Each array element qualified just one row for deletion. In the following example, each array element qualifies multiple rows:

```
char job_title[10][20];

/* populate the host array */
...

EXEC SQL DELETE FROM emp
   WHERE job = :job_title;
...
```



## 8.7.1 About Deleting with Arrays Restrictions

Mixing simple host variables with host arrays in the WHERE clause of a DELETE statement is *not* allowed. If any of the host variables is an array, all must be arrays.

You cannot use host arrays with the CURRENT OF clause in a DELETE statement.



# 8.8 About Using the FOR Clause

You can use the optional embedded SQL FOR clause to set the number of array elements processed by any of the following SQL statements:

- DELETE
- EXECUTE
- FETCH
- INSERT
- OPEN
- UPDATE

The FOR clause is especially useful in UPDATE, INSERT, and DELETE statements. With these statements you might not want to use the entire array. The FOR clause lets you limit the elements used to just the number you need, as the following example shows:

The FOR clause can use an integer host variable to count array elements, or an integer literal. A complex C expression that resolves to an integer *cannot* be used. For example, the following statement that uses an integer expression is illegal:

The FOR clause variable specifies the number of array elements to be processed. Make sure the number does not exceed the smallest array dimension. Internally, the value is treated as an unsigned quantity. An attempt to pass a negative value through the use of a signed host variable will result in unpredictable behavior.

### 8.8.1 FOR Clause Restrictions

Two restrictions keep FOR clause semantics clear: you cannot use the FOR clause in a SELECT statement or with the CURRENT OF clause.

### 8.8.1.1 In a SELECT Statement

If you use the FOR clause in a SELECT statement, you get an error message.

The FOR clause is not allowed in SELECT statements because its meaning is unclear. Does it mean "execute this SELECT statement *n* times"? Or, does it mean "execute this SELECT statement once, but return *n* rows"? The problem in the former case is that each execution might return multiple rows. In the latter case, it is better to declare a cursor and use the FOR clause in a FETCH statement, as follows:

```
EXEC SQL FOR : limit FETCH emp cursor INTO ...
```

### 8.8.1.2 With the CURRENT OF Clause

You can use the CURRENT OF clause in an UPDATE or DELETE statement to refer to the latest row returned by a FETCH statement, as the following example shows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, sal FROM emp WHERE empno = :emp_number;
...
EXEC SQL OPEN emp_cursor;
...
EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
...
EXEC SQL UPDATE emp SET sal = :new_salary
WHERE CURRENT OF emp cursor;
```

However, you cannot use the FOR clause with the CURRENT OF clause. The following statements are invalid because the only logical value of *limit* is 1 (you can only update or delete the current row once):

```
EXEC SQL FOR :limit UPDATE emp SET sal = :new_salary
WHERE CURRENT OF emp_cursor;
...
EXEC SQL FOR :limit DELETE FROM emp
WHERE CURRENT OF emp cursor;
```

# 8.9 About Using the WHERE Clause

Oracle treats a SQL statement containing host arrays of number of elements n like the same SQL statement executed n times with n different scalar variables (the individual array elements). The precompiler issues an error message only when such treatment would be ambiguous.

For example, assuming the declarations

```
int mgr_number[50];
char job_title[50][20];
```

it would be ambiguous if the statement

```
EXEC SQL SELECT mgr INTO :mgr_number FROM emp
WHERE job = :job_title;
```

#### were treated like the imaginary statement

```
for (i = 0; i < 50; i++)
    SELECT mgr INTO :mgr_number[i] FROM emp
    WHERE job = :job title[i];</pre>
```

because multiple rows might meet the WHERE-clause search condition, but only one output variable is available to receive data. Therefore, an error message is issued.

On the other hand, it would not be ambiguous if the statement

```
EXEC SQL UPDATE emp SET mgr = :mgr_number
WHERE empno IN (SELECT empno FROM emp
WHERE job = :job_title);
```

were treated like the imaginary statement

```
for (i = 0; i < 50; i++)
    UPDATE emp SET mgr = :mgr_number[i]
    WHERE empno IN (SELECT empno FROM emp
    WHERE job = :job_title[i]);</pre>
```

because there is a *mgr\_number* in the SET clause for each row matching *job\_title* in the WHERE clause, even if each *job\_title* matches multiple rows. All rows matching each *job\_title* can be SET to the same *mgr\_number*. Therefore, no error message is issued.

# 8.10 Arrays of Structs

Using arrays of scalars, you can perform multirow operations involving a single column only. Using structs of scalars allows users to perform single row operations involving multiple columns.

In order to perform multirow operations involving multiple columns, however, you previously needed to allocate several parallel arrays of scalars either separately or encapsulated within a single struct. In many cases, it is easier to reorganize this data structure more conveniently as a single array of structs instead.

Pro\*C/C++ supports the use of *arrays of structs* which enable an application developer to perform multirow, multicolumn operations using an array of C structs. With this enhancement, Pro\*C/C++ can handle simple arrays of structs of scalars as bind variables in embedded SQL statements for easier processing of user data. This makes programming more intuitive, and allows users greater flexibility in organizing their data.

In addition to supporting arrays of structs as bind variables, Pro\*C/C++ also supports arrays of indicator structs when used in conjunction with an array of structs declaration.



Binding structs to PL/SQL records and binding arrays of structs to PL/SQL tables of records are *not* part of this new functionality. Arrays of structs may also not be used within an embedded PL/SQL block.

Since arrays of structs are intended to be used when performing multirow operations involving multiple columns, it is generally anticipated that they will be used in the following ways.

As output bind variables in SELECT statements or FETCH statements.

As input bind variables in the VALUES clause of an INSERT statement.

#### **Related Topics**

Restrictions on Arrays of Structs

## 8.10.1 Arrays of Structs Usage

The notion of an array of structs is not new to C developers. It does, however, present a conceptual difference for data storage when it is compared to a struct of parallel arrays.

In a struct of parallel arrays, the data for the individual columns is stored contiguously. In an array of structs, on the other hand, the column data is *interleaved*, whereby each occurrence of a column in the array is separated by the space required by the other columns in the struct. This space is known as a *stride*.

## 8.10.2 Restrictions on Arrays of Structs

The following restrictions apply to the use of arrays of structs in Pro\*C/C++:

- Arrays of structs (just as with ordinary structs) are not permitted inside an embedded PL/SQL block.
- Use of arrays of structs in WHERE or FROM clauses is prohibited.
- Arrays of structs are not permitted with Oracle Dynamic SQL Method 4. They are permitted with ANSI Dynamic SQL.
- Arrays of structs are not permitted in the SET clause of an UPDATE statement.

The syntax for declaring an array of structs does not change. There are, however, a few things to keep in mind when using an array of structs.

#### **Related Topics**

ANSI Dynamic SQL

## 8.10.3 About Declaring an Array of Structs

When declaring an array of structs which will be used in a Pro\*C/C++ application, a developer must keep in mind the following important points:

The struct must have a structure tag. For example, in the following code segment

```
struct person {
   char name[15];
   int age;
} people[10];
```

the person variable is the structure tag. This is so the precompiler can use the name of the struct to compute the size of the stride.

- The members of the struct must not be arrays. The only exception to this rule is for character types such as **char** or **VARCHAR** since array syntax is used when declaring variables of these types.
- char and VARCHAR members may not be two-dimensional.
- Nested structs are not permitted as members of an array of structs. This is not a new restriction, since nested structs have not been supported by previous releases of Pro\*C/C+ +.



 The size of just the struct may not exceed the maximum value that a signed 4-byte quantity may represent. This is typically two gigabytes.

Given these restrictions regarding the use of arrays of structs, the following declaration is legal in Pro\*C/C++

```
struct department {
   int deptno;
   char dname[15];
   char loc[14];
} dept[4];
```

while the following declaration is illegal.

It is also important to note that you cannot apply datatype equivalencing to either the array of structs itself or to any of the individual fields within the struct. For example, assuming empno is not declared as an array in the earlier illegal struct, the following is illegal:

```
exec sql var bad[3].empno is integer(4);
```

The precompiler has no way to keep track of individual structure elements within the array of structs. One could do the following, on the other hand, to achieve the desired effect.

```
typedef int myint;
exec sql type myint is integer(4);
struct equiv {
  myint empno; /* now legally considered an integer(4) datatype */
    ...
} ok[15];
```

This should come as no surprise since equivalencing individual array items has not been supported by previous releases of Pro\*C/C++. For example, the following scalar array declarations illustrate what is legal and what is not.

```
int empno[15];
exec sql var empno[3] is integer(4); /* illegal */
myint empno[15]; /* legal */
```

In summary, you may not equivalence any individual array item.

### 8.10.4 Variables Guidelines

Indicator variables for an array of structs declaration work in much the same way as a normal struct declaration. An indicator array of structs declaration must abide by the rules for an array of structs as follows:

- The number of fields in the indicator struct must be less than or equal to the number of fields in the corresponding array of structs.
- The order of the fields must match the order of the corresponding members of the array of structs.



- The datatype for all elements in the indicator struct must be short.
- The size of the indicator array must be at least the same size as the host variable declaration. It may be larger, but it may not be smaller.

These rules generally reflect the rules for using structs as implemented in prior releases of Pro\*C/C++. The array restriction is also the same as that previously used for arrays of scalars.

Given these rules, assume the following struct declaration:

```
struct department {
  int deptno;
  char dname[15];
  char loc[14];
} dept[4];
```

The following is a legal indicator variable struct declaration:

```
struct department_ind {
    short deptno_ind;
    short dname_ind;
    short loc_ind;
} dept ind[4];
```

while the following is illegal as an indicator variable

#### **Related Topics**

ANSI Dynamic SQL

## 8.10.5 About Declaring a Pointer to an Array of Structs

In some cases, it may be desirable to declare a pointer to an array of structs. This allows pointers to arrays of structs to be passed to other functions or used directly in an embedded SQL statement.



The length of the array referenced by a pointer to an array of structs cannot be known during precompilation. For this reason, an explicit FOR clause must be used when a bind variable whose type is a pointer to an array of structs is used in any embedded SQL statement.

Remember that FOR clauses may not be used in an embedded SQL SELECT statement. Therefore, to retrieve data into a pointer to an array of structs, an explicit cursor and FETCH statement must be used with the FOR clause.

## 8.10.6 Examples

The following examples demonstrate different uses of the array of structs functionality in Pro\*C/C++.

#### 8.10.6.1 Example 1: A Simple Array of Structs of Scalars

Given the following structure declaration,

```
struct department {
   int deptno;
   char dname[15];
   char loc[14];
} my dept[4];
```

a user could then select the dept data into my dept as follows:

```
exec sql select * into :my_dept from dept;
```

or the user could populate my dept first and then bulk insert it into the dept table:

```
exec sql insert into dept values (:my dept);
```

To use an indicator variable, a parallel indicator array of structs could be declared.

```
struct deptartment_ind {
    short deptno_ind;
    short dname_ind;
    short loc_ind;
} my dept ind[4];
```

Data is then be selected using the same query except for the addition of the indicator variable:

```
exec sql select * into :my_dept indicator :my_dept_ind from dept;
```

Similarly, the indicator could be used when inserting the data as well:

```
exec sql insert into dept values (:my dept indicator :my dept ind);
```

### 8.10.6.2 Example 2: Using Mixed Scalar Arrays with An Array of Structs

As in prior releases of Pro\*C/C++, when using multiple arrays for bulk handling of user data, the size of the arrays must be the same. If they are not, the smallest array size is chosen leaving the remaining portions of the arrays unaffected.

Given the following declarations,

```
struct employee {
   int empno;
   char ename[11];
} emp[14];

float sal[14];
float comm[14];
```

it is possible to select multiple rows for all columns in one simple query:

```
exec sql select empno, ename, sal, comm into :emp, :sal, :comm from emp;
```

We also want to know whether the column values for the commissions are NULL or not. A single indicator array could be used given the following declaration:

```
short comm_ind[14];
...
exec sql select empno, ename, sal, comm
  into :emp, :sal, :comm indicator :comm ind from emp;
```



You cannot declare a single indicator array of structs that encapsulate all indicator information from the query. Therefore:

```
struct employee_ind {    /* example of illegal usage */
    short empno_ind;
    short ename_ind;
    short sal_ind;
    short comm_ind;
} illegal_ind[15];

exec sql select empno, ename, sal, comm
    into :emp, :sal, :comm indicator :illegal ind from emp;
```

is illegal (as well as undesirable). The earlier statement associates the indicator array with the comm column only, not the entire SELECT...INTO list.

Assuming the array of structs and the sal, comm and comm\_ind arrays were populated with the desired data, insertion is straightforward:

```
exec sql insert into emp (empno, ename, sal, comm)
values (:emp, :sal, :comm indicator :comm ind);
```

### 8.10.6.3 Example 3: Using Multiple Arrays of Structs with a Cursor

For this example, we make the following declarations:

```
struct employee {
   int empno;
   char ename[11];
   char job[10];
} emp[14];

struct compensation {
   int sal;
   int comm;
} wage[14];

struct compensation_ind {
   short sal_ind;
   short comm_ind;
} wage_ind[14];
```

Our program could then make use of these arrays of structs as follows:

```
exec sql declare c cursor for
    select empno, ename, job, sal, comm from emp;

exec sql open c;

exec sql whenever not found do break;
while(1)
{
    exec sql fetch c into :emp, :wage indicator :wage_ind;
    ... process batch rows returned by the fetch ...
}

printf("%d rows selected.\n", sqlca.sqlerrd[2]);
exec sql close c;
```



#### 8.10.6.3.1 About Using the FOR clause

Alternatively, we could have used the FOR clause to instruct the fetch on how many rows to retrieve. Recall that the FOR clause is prohibited when using the SELECT statement, but not the INSERT or FETCH statements.

We add the following to our original declarations

```
int limit = 10;
```

and code our example accordingly.

```
exec sql for :limit
  fetch c into :emp, :wage indicator :wage ind;
```

#### 8.10.6.4 Example 4: Individual Array and Struct Member Referencing

Prior releases of Pro\*C/C++ allowed array references to single structures in an array of structs. The following is therefore legal since the bind expression resolves to a simple struct of scalars.

```
exec sql select * into :dept[3] from emp;
```

Users can reference an individual scalar member of a specific struct in an array of structs as the following example shows.

```
exec sql select dname into :dept[3].dname from dept where ...;
```

Naturally, this requires that the query be a single row query so only one row is selected into the variable represented by this bind expression.

### 8.10.6.5 Example 5: Using Indicator Variables, a Special Case

Prior releases of Pro\*C/C++ required that an indicator struct have the same number of fields as its associated bind struct. This restriction has been relaxed when using structs in general. By following the previously mentioned guidelines for indicator arrays of structs it is possible to construct the following example.

```
struct employee {
    float comm;
    float sal;
    int empno;
    char ename[10];
} emp[14];

struct employee_ind {
    short comm;
} emp_ind[14];

exec sql select comm, sal, empno, ename
    into :emp indicator :emp ind from emp;
```

The mapping of indicator variables to bind values is one-to-one. They map in associative sequential order starting with the first field.

Be aware, however, that if any of the other fields has a fetched value of NULL and no indicator is provided, the following error is raised:

```
ORA-1405: fetched column value is NULL
```

As an example, such is the case if sal was nullable because there is no indicator for sal.

Suppose we change the array of structs as follows,

```
struct employee {
   int empno;
   char ename[10];
   float sal;
   float comm;
} emp[15];
```

but still used the same indicator array of structs.

Because the indicators map in associative sequential order, the comm indicator maps to the empno field leaving the comm bind variable without an indicator once again leading to the ORA-1405 error.

To avoid the ORA-1405 when using indicator structs that have fewer fields than their associative bind variable structs, the nullable attributes should appear first and in sequential order.

We could easily change this into a single-row fetch involving multiple columns by using nonarray structs and expect it to work as though the indicator struct was declared as follows.

```
struct employee_ind {
    short comm;
    short sal;
    short empno;
    short ename;
} emp ind;
```

Because Pro\*C/C++ no longer requires that the indicator struct have the same number of fields as its associated value struct, the earlier example is now legal in Pro\*C/C++ whereas previously it was not.

Our indicator struct could now look like the following simple struct.

```
struct employee_ind {
    short comm;
} emp ind;
```

Using the non-array emp and  $emp\_ind$  structs we are able to perform a single row fetch as follows.

```
exec sql fetch comm, sal, empno, ename
  into :emp indicator :emp_ind from emp;
```

Note once again how the comm indicator maps to the comm bind variable in this case as well.

### 8.10.6.6 Example 6: Using a Pointer to an Array of Structs

This example demonstrates how to use a pointer to an array of structs.

Given the following type declaration:

```
typedef struct dept {
  int deptno;
  char dname[15];
  char loc[14];
} dept;
```

we can perform a variety of things, manipulating a pointer to an array of structs of that type. For example, we can pass pointers to arrays of structs to other functions.

```
void insert_data(d, n)
  dept *d;
  int n;
{
    exec sql for :n insert into dept values (:d);
}

void fetch_data(d, n)
  dept *d;
  int n;
{
    exec sql declare c cursor for select deptno, dname, loc from dept;
    exec sql open c;
    exec sql for :n fetch c into :d;
    exec sql close c;
}
```

Such functions are invoked by passing the address of the array of structs as these examples indicate.

```
dept d[4];
dept *dptr = &d[0];
const int n = 4;

fetch_data(dptr, n);
insert data(d, n); /* We are treating '&d[0]' as being equal to 'd' */
```

Or we can simply use such pointers to arrays of structs directly in some embedded SQL statement.

```
exec sql for :n insert into dept values (:dptr);
```

The most important thing to remember is the use of the FOR clause.

# 8.11 About Mimicking CURRENT OF

You use the CURRENT OF *cursor* clause in a DELETE or UPDATE statement to refer to the latest row FETCHed from the cursor. However, you cannot use CURRENT OF with host arrays. Instead, select the ROWID of each row, then use that value to identify the current row during the update or delete.

#### For example:

```
char emp_name[20][10];
char job_title[20][10];
char old_title[20][10];
char row_id[20][19];
...

EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, job, rowid FROM emp FOR UPDATE;
...

EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND do break;
for (;;)
{
    EXEC SQL FETCH emp_cursor
        INTO :emp_name, :job_title, :row_id;
    ...
    EXEC SQL DELETE FROM emp
    WHERE job = :old title AND rowid = :row id;
```



```
EXEC SQL COMMIT WORK;
```

#### **Related Topics**

The CURRENT OF Clause

# 8.12 About Using Additional Array Insert/Select Syntax

The Oracle precompiler also supports the DB2 insert and fetch syntax for the host tables. The supported additional array insert and fetch syntax are shown in the following figures, respectively.

Figure 8-1 Additional Insert Syntax

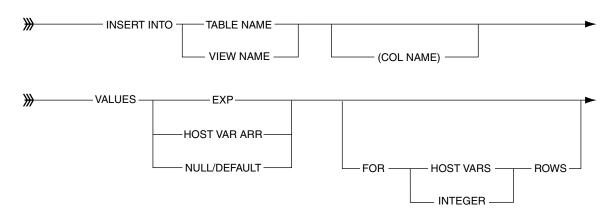
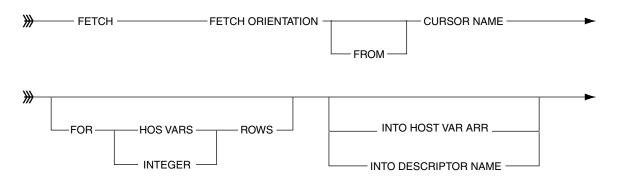


Figure 8-2 Additional Fetch Syntax



The optional ROWSET and ROWSET STARTING AT clauses are used in the fetch-orientation (FIRST, PRIOR, NEXT, LAST, CURRENT, RELATIVE and ABSOLUTE). Consider the following examples:

- FIRST ROWSET
- PRIOR ROWSET
- NEXT ROWSET
- LAST ROWSET

- CURRENT ROWSET
- ROWSET STARTING AT RELATIVE
- ROWSET STARTING AT ABSOLUTEn

Examples of the DB2 array insert/fetch syntax and their comparison with the corresponding Oracle precompiler syntax are shown in Table 8-3:

Table 8-3 DB2 Array Syntax vs. Oracle Precompiler Syntax

DB2 Array Syntax	Oracle Precompiler Syntax
<pre>EXEC SQL   INSERT INTO dsn8810.act   (actno, actkwd, actdesc)   VALUES (:hva1, :hva2, :hva3)   FOR :NUM_ROWS ROWS;</pre>	<pre>EXEC SQL FOR :num_rows   INSERT INTO dsn8810.act   (actno, actkwd, actdesc)   VALUES (:hva1, :hva2, :hva3);</pre>
<pre>EXEC SQL   FETCH NEXT ROWSET FROM c1   FOR 20 ROWS   INTO :hva_empno, :hva_lastname,       :hva_salary;</pre>	<pre>EXEC SQL FOR :twenty FETCH c1 INTO :hva_empno, :hva_lastname,     :hva_salary;</pre>

In DB2 syntax, a row-set positioned cursor should be first declared before retrieving row sets of data. To enable a cursor to fetch row sets, the 'WITH ROWSET POSITIONING' clause has to be used in the DECLARE CURSOR statement, which is not required and relevant in the Oracle precompiler syntax, as shown in the following table.

DB2 Array Syntax	Oracle Precompiler Syntax
EXEC SQL  DECLARE c1 CURSOR  WITH ROWSET POSITIONING FOR  SELECT empno, lastname, salary  FROM dsn8810.emp;	EXEC SQL  DECLARE c1 CURSOR FOR  SELECT empno, lastname, salary  FROM dsn8810.emp;

This DB2 array syntax support can be enabled with the precompiler option  $db2\_array$ , whose default option is no. The DB2 array syntax support cannot be used together with the Oracle precompiler syntax; only one of the syntax, only on of the syntax, either Oracle precompiler or DB2 syntax, is supported at a time.

#### Example 8-1 Inserting and Fetching Rows by Using the DB2 Array Syntax

This program inserts INSCNT rows into the EMP table by using the DB2 array insert syntax, and then fetches the inserted rows by using the DB2 array fetch syntax.

```
/*
    * db2arrdemo.pc
    */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
#include <sqlcpr.h>
```



```
/* Number of rows to be inserted in one shot */
#define INSCNT 100
/* Number of rows to be fetched in one shot */
#define FETCHCNT 20
/* Define a host structure
   for inserting data into the table
   and for fetching data from the table */
struct emprec
  int
          empno;
  varchar ename[10];
  varchar job[9];
  int mgr;
  char hiredate[10];
  float sal;
  float comm;
  int deptno;
};
typedef struct emprec empdata;
/* Function prototypes */
void sql error(char *);
void insertdata();
void fetchdata();
void printempdata(empdata *);
void main()
  exec sql begin declare section;
   char *uid = "scott/tiger";
  exec sql end declare section;
  exec sql whenever sqlerror do sql_error("ORACLE error--\n");
  exec sql connect :uid;
  printf("Inserting %d rows into EMP table using DB2 array insert syntax.\n",
         INSCNT);
  insertdata();
  printf("\nFetching data using DB2 array fetch syntax.\n");
  fetchdata();
  exec sql rollback work release;
  exit(EXIT SUCCESS);
/* Inserting data into the table using DB2 array insert syntax*/
void insertdata()
  int i, cnt;
  char *str;
  empdata emp in[INSCNT];
  /* To store temporary strings */
  str = (char *)malloc (25 * sizeof(char));
  /* Fill the array elements to insert */
  for (i = 0; i < INSCNT; i++)
    emp in[i].empno = i+1;
```

```
sprintf(str, "EMP %03d", i+1);
    strcpy (emp in[i].ename.arr, str);
    emp_in[i].ename.len = strlen (emp_in[i].ename.arr);
    sprintf(str, "JOB %03d", i+1);
    strcpy (emp_in[i].job.arr, str);
    emp in[i].job.len = strlen (emp in[i].job.arr);
    emp in[i].mgr = i+1001;
    sprintf(str, "%02d-MAY-06", (i%30)+1);
    strcpy (emp in[i].hiredate, str);
    emp_in[i].sal = (i+1) * 10;
    emp in[i].comm = (i+1) * 0.1;
    emp in[i].deptno = 10;
  free (str);
  /* Inserting data using DB2 array insert syntax */
  exec sql insert into emp values (:emp in) FOR :INSCNT rows;
  exec sql select count(*) into :cnt from emp where ename like 'EMP %';
 printf ("Number of rows successfully inserted into emp table: %d\n", cnt);
/* Fetches data from the table using DB2 array fetch syntax*/
void fetchdata()
  empdata emp out[FETCHCNT];
  /* Declares scrollable cursor to fetch data */
  exec sql declare c1 scroll cursor with rowset positioning for
         select empno, ename, job, mgr, hiredate, sal, comm, deptno
         from emp where ename like 'EMP %' order by empno;
  exec sql open c1;
  exec sql whenever not found do break;
  while(1)
    /* Fetches data using DB2 array fetch syntax */
   exec sql fetch next rowset from cl for :FETCHCNT rows into :emp out;
   printempdata(emp out);
  exec sql whenever not found do sql error("ORACLE ERROR");
 exec sql close c1;
}
/* Prints the fetched employee data */
void printempdata(empdata *emp out)
  int i;
  for (i=0; i<FETCHCNT; i++)
    emp out[i].ename.arr[emp out[i].ename.len] = '\0';
    emp out[i].job.arr[emp out[i].job.len] = '\0';
    printf("Empno=%d, Ename=%s, Job=%s, Mgr=%d, Hiredate=%s, Sal=%6.2f,\n"
           "Comm=%5.2f, Deptno=%d\n", emp out[i].empno, emp out[i].ename.arr,
 emp_out[i].job.arr, emp_out[i].mgr, emp_out[i].hiredate,
 emp_out[i].sal, emp_out[i].comm, emp_out[i].deptno);
 }
}
```

```
/* Error handling function. */
void sql_error(char *msg)
{
   exec sql whenever sqlerror continue;
   printf("\n%s\n", msg);
   printf("%.70s\n", sqlca.sqlerrm.sqlerrmc);
   exec sql rollback release;
   exit(EXIT_FAILURE);
}
```

# 8.13 About Using Implicit Buffered Insert

For improved performance, Pro\*C/C++ application developers can reference host arrays in their embedded SQL statements. This provides a means to execute an array of SQL statements with a single round-trip to the database. Despite the significant performance improvements afforded by array execution, some developers choose not to use this capability because it is not ANSI standard. For example, an application written to exploit array execution in Oracle cannot be precompiled using IBM's precompiler.

One workaround is to use buffered INSERT statements, which enable you to gain performance benefits while retaining ANSI standard embedded SQL syntax.

The command line option "max\_row\_insert" controls the number of rows to be buffered before executing the INSERT statement. By default it is zero and the feature is disabled. To enable this feature, specify any number greater than zero.

If insert bufering is enabled, precompiler runtime will flag the corresponding cursor and:

- Allocate or re-allocate extra memory to hold bind values (first execute only).
- Copy bind values from program host variables to internal runtime bind structures.
- Increment the rows buffered count.
- Flush the buffered INSERT statements if MAX INSERT ROWS have been buffered.
- If MAX\_INSERT\_ROWS has not been hit, then return after copying the values to the internal bind buffers without flushing.

If a new embedded SQL statement is executed and results in a flush of the buffered insert statements:

- Flush the buffers.
- Continue with the call that prompted the flush.

The application is informed of the error through the standard precompiler error mechanisms such as the sqlca in Pro\*C.

The "implicit\_svpt" option controls whether an implicit savepoint is taken prior to the start of a new batched insert.

- If yes, a savepoint is taken prior to the start of a new batch of rows. If an error occurs on the insert, an implicit "rollback to savepoint" is executed.
- If no, there is no implicit savepoint taken. If an error occurs on the buffered insert, then it is
  reported back to the application, but no rollback is executed. Errors are reported
  asynchronously for buffer inserts. Errors for inserted rows are not reported when the
  INSERT statement is executed in the application.



Some errors for inserted rows are reported later, when the first statement other than the INSERT is executed. This may include DELETE, UPDATE, INSERT (into different tables), COMMIT, and ROLLBACK. Any statement that closes the buffered insert statement can report an error. In such cases, the statement that reports the error is not executed. You need to first handle the error and also reexecute the statement on which the buffered insert error is reported. Otherwise, you may rollback the transaction and reexecute it.

For example, consider using a COMMIT statement to close a buffered insert loop. COMMIT can report an error because of a duplicate key from an earlier insert. In this scenario, the commit is not executed. You should first handle the error and then reexecute COMMIT. Otherwise, you can rollback the transaction and reexecute it.

Some errors are reported on the insert itself, and may reflect an error of a previously inserted row. In such cases, no further inserts are executed. You need to handle the errors of the previously inserted row and continue inserting the current insert, which is a long process. Instead, you may rollback and reexecute the transaction.

For example, consider that the limit of internal buffer is 10 rows and the application is inserting 15 rows in a loop. Suppose there is an error on the 8th row. The error is reported when the 11th row insert happens and the insert is no more executed further.

The following are some of the possible errors that you might face during buffered insert:

- ORA-00001: duplicate key in index
- ORA-01400: mandatory (not null) column is missing or Null during insert
- ORA-01401: inserted value too large for column
- ORA-01438: value larger than specified precision allows for this column

#### **Example 8-2** Inserting Buffered Rows into a Table

This program inserts LOOPCNT number of rows into the EMP table. At loop counter=5, this program attempts to insert an invalid empno. Without the max\_row\_insert option, the program inserts all rows except the invalid row. When the max\_row\_insert option is set to LOOPCNT, only the first four rows are inserted.

Using the max\_row\_insert option, when the erroneous statement is removed, the program performs the same way an array insert program would.

```
* bufinsdemo.pc
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>
#include <sqlca.h>
/* Number of rows to be inserted into the table */
#define LOOPCNT 100
/* Define a host structure
  for inserting data into the table
  and for fetching data from the table */
struct emprec
 int empno;
 varchar ename[10];
 varchar job[9];
 int mgr;
```

```
char hiredate[10];
  float sal;
  float comm;
  int deptno;
typedef struct emprec buffinstyp;
/* Function prototypes */
void sql error();
void insertdata();
void fetchdata();
void printempdata(buffinstyp);
void main()
{
  exec sql begin declare section;
   char *uid = "scott/tiger";
  exec sql end declare section;
  exec sql whenever sqlerror do sql_error();
  exec sql connect :uid;
  printf("\nInserting %d rows into EMP table.\n", LOOPCNT);
  insertdata();
  \label{lem:printf("nFetching inserted data from EMP table.\n");}
  fetchdata();
  exec sql delete from emp where empno < 1000;
  exec sql commit work release;
  exit(EXIT_SUCCESS);
/* Inserting data into the table */
void insertdata()
{
  int i, cnt;
  char *str;
  buffinstyp emp in;
  /* To store temporary strings */
  str = (char *)malloc (25 * sizeof(char));
   ^{\star} When max row insert option is set to LOOPCNT and when the errorneous
   ^{\star} statement is removed, all the rows will be inserted into the database in
   * one stretch and hence maximum performance gain will be achieved.
  for (i = 1; i <= LOOPCNT; i++)
    if (i != 5)
      emp in.empno = i;
    /* Errorneous statement. In emp table, empno is defined as number(4). */
     emp in.empno = 10000;
    sprintf(str, "EMP_%03d", i);
    strcpy (emp_in.ename.arr, str);
    emp_in.ename.len = strlen (emp_in.ename.arr);
    sprintf(str, "JOB %03d", i);
    strcpy (emp in.job.arr, str);
```

```
emp in.job.len = strlen (emp in.job.arr);
    emp in.mgr = i+1001;
    sprintf(str, "%02d-MAY-06", (i%30));
    strcpy (emp_in.hiredate, str);
    emp_in.sal = (i) * 10;
    emp in.comm = (i) * 0.1;
    emp in.deptno = 10;
    exec sql insert into emp values (:emp in);
  free (str);
 exec sql commit;
 exec sql select count(*) into :cnt from emp where ename like 'EMP %';
 printf ("Number of rows successfully inserted into emp table: %d\n", cnt);
/* Fetches data from the table*/
void fetchdata()
 buffinstyp emp out;
  /* Declares cursor to fetch only the rows that are inserted */
  exec sql declare c1 cursor for
         select empno, ename, job, mgr, hiredate, sal, comm, deptno
         from emp where ename like 'EMP %' order by empno;
  exec sql open c1;
  exec sql whenever not found do break;
  while(1)
    /* Fetches single row at each call */
   exec sql fetch cl into :emp out;
   printempdata(emp_out);
  exec sql whenever not found do sql error();
  exec sql close c1;
/* Prints the fetched employee data */
void printempdata(buffinstyp emp out)
 emp out.ename.arr[emp out.ename.len] = '\0';
  emp_out.job.arr[emp_out.job.len] = '\0';
 printf("Empno=%d, Ename=%s, Job=%s, Mgr=%d, Hiredate=%s, Sal=%6.2f, \n"
         "Comm=%5.2f, Deptno=%d\n", emp out.empno, emp out.ename.arr,
 emp out.job.arr, emp out.mgr, emp out.hiredate, emp out.sal,
 emp out.comm, emp out.deptno);
/* Error handling function. */
void sql error()
  printf("Error %s\n", sqlca.sqlerrm.sqlerrmc);
  printf(" Rows Processed: %d\n", sqlca.sqlerrd[2]);
  printf(" Rows Rolled Back: %d\n", sqlca.sqlerrd[0]);
```

# 8.14 Scrollable Cursors

A scrollable cursor is a work area where Oracle executes SQL statements and stores information that is processed during execution. When a cursor is executed, the results of the query are placed into a a set of rows called the result set. The result set can be fetched either sequentially or non-sequentially. Non-sequential result sets are called scrollable cursors. A scrollable cursor enables users to access the rows of a database result set in a forward, backward, and random manner. This enables the program to fetch any row in the result set.



9

# **Handling Runtime Errors**

An application program must anticipate runtime errors and attempt to recover from them. This chapter provides an in-depth discussion of error reporting and recovery. You learn how to handle errors and status changes using the SQLSTATE status variable, as well as the SQL Communications Area (SQLCA) and the WHENEVER directive. You also learn how to diagnose problems using the Oracle Communications Area (ORACA). This chapter contains the following topics:

- The Need for Error Handling
- Error Handling Alternatives
- The SQLSTATE Status Variable
- About Declaring SOLCODE
- Key Components of Error Reporting Using the SQLCA
- Using the SQL Communications Area (SQLCA)
- · About Getting the Full Text of Error Messages
- About Using the WHENEVER Directive
- About Obtaining the Text of SQL Statements
- About Using the Oracle Communications Area (ORACA)

# 9.1 The Need for Error Handling

A significant part of every application program must be devoted to error handling. The main reason for error handling is that it allows your program to continue operating in the presence of errors. Errors arise from design faults, coding mistakes, hardware failures, invalid user input, and many other sources.

You cannot anticipate all possible errors, but you can plan to handle certain kinds of errors that are meaningful to your program. For the Pro\*C/C++ Precompiler, error handling means detecting and recovering from SQL statement execution errors. You can also prepare to handle warnings such as "value truncated" and status changes such as "end of data." It is especially important to check for error and warning conditions after every SQL data manipulation statement, because an INSERT, UPDATE, or DELETE statement might fail before processing all eligible rows in a table.

# 9.2 Error Handling Alternatives

There are several alternatives that you can use to detect errors and status changes in the application. This chapter describes these alternatives, however, no specific recommendations are made about what method you should use. The method is, after all, dictated by the design of the application program or tool that you are building.

#### 9.2.1 Status Variables

You can declare a separate status variable, SQLSTATE or SQLCODE, examine its value after each executable SQL statement, and take appropriate action. The action might be calling an error-reporting function, then exiting the program if the error is unrecoverable. Or, you might be able to adjust data or control variables and retry the action.

#### **Related Topics**

- The SQLSTATE Status Variable
- About Declaring SQLCODE

## 9.2.2 The SQL Communications Area

Another alternative that you can use is to include the SQL Communications Area structure (*sqlca*) in your program. This structure contains components that are filled in at runtime after the SQL statement is processed by Oracle.



In this guide, the *sqlca* structure is commonly referred to using the acronym for *SQL Communications Area* (SQLCA). When this guide refers to a specific component in the C **struct**, the structure name (*sqlca*) is used.

The SQLCA is defined in the header file sqlca.h, which you include in your program using either of the following statements:

- EXEC SQL INCLUDE SQLCA;
- #include <sqlca.h>

Oracle updates the SQLCA after every *executable* SQL statement. (SQLCA values are unchanged after a declarative statement.) By checking Oracle return codes stored in the SQLCA, your program can determine the outcome of a SQL statement. This can be done in the following two ways:

- Implicit checking with the WHENEVER directive
- Explicit checking of SQLCA components

You can use WHENEVER directives, code explicit checks on SQLCA components, or do both.

The most frequently-used components in the SQLCA are the status variable (sqlca.sqlcode), and the text associated with the error code (sqlca.sqlerrm.sqlerrmc). Other components contain warning flags and miscellaneous information about the processing of the SQL statement.



SQLCODE (upper case) always refers to a separate status variable, not a component of the SQLCA. SQLCODE is declared as a **integer**. When referring to the component of the SQLCA named *sqlcode*, the fully-qualified name *sqlca.sqlcode* is always used.



When more information is needed about runtime errors than the SQLCA provides, you can use the ORACA. The ORACA is a C **struct** that handles Oracle communication. It contains cursor statistics, information about the current SQL statement, option settings, and system statistics.

#### **Related Topics**

- About Using the Oracle Communications Area (ORACA)
- Using the SQL Communications Area (SQLCA)

## 9.3 The SQLSTATE Status Variable

The precompiler command line option MODE governs ANSI/ISO compliance. When MODE=ANSI, declaring the SQLCA data structure is optional. However, you must declare a separate status variable named SQLCODE. The SQL standard specifies a similar status variable named SQLSTATE, which you can use with or without SQLCODE.

After executing a SQL statement, the Oracle Server returns a status code to the SQLSTATE variable currently in scope. The status code indicates whether the SQL statement executed successfully or raised an exception (error or warning condition). To promote *interoperability* (the ability of systems to exchange information easily), the SQL standard predefines all the common SQL exceptions.

Unlike SQLCODE, which stores only error codes, SQLSTATE stores error and warning codes. Furthermore, the SQLSTATE reporting mechanism uses a standardized coding scheme. Thus, SQLSTATE is the preferred status variable. SQLCODE was a deprecated feature of SQL-92 that was retained only for compatibility with SQL-89. SQLCODE has been removed from all editions of the SQL standard subsequent to SQL-92.

## 9.3.1 About Declaring SQLSTATE

When MODE=ANSI, you must declare SQLSTATE or SQLCODE. Declaring the SQLCA is optional. When MODE=ORACLE, if you declare SQLSTATE, it is not used.

Unlike SQLCODE, which stores signed integers and can be declared outside the Declare Section, SQLSTATE stores 5-character null-terminated strings and must be declared inside the Declare Section. You declare SQLSTATE as

char SQLSTATE[6]; /\* Upper case is required. \*/



SQLSTATE must be declared with a dimension of exactly 6 characters.

## 9.3.2 SQLSTATE Values

SQLSTATE status codes consist of a 2-character *class code* immediately followed by a 3-character *subclass code*. Aside from class code 00 ("successful completion",) the class code denotes a category of exceptions. And, aside from subclass code 000 ("not applicable",) the subclass code denotes a specific exception within that category. For example, the SQLSTATE value '22012' consists of class code 22 ("data exception") and subclass code 012 ("division by zero").

Each of the five characters in a SQLSTATE value is a digit (0..9) or an uppercase Latin letter (A..Z). Class codes that begin with a digit in the range 0..4 or a letter in the range A..H are

reserved for predefined conditions (those defined in the SQL standard). All other class codes are reserved for implementation-defined conditions. Within predefined classes, subclass codes that begin with a digit in the range 0..4 or a letter in the range A..H are reserved for predefined subconditions. All other subclass codes are reserved for implementation-defined subconditions. Figure 9-1 shows the coding scheme.

Figure 9-1 SQLSTATE Coding Scheme

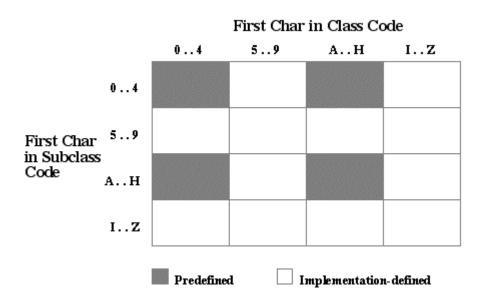


Table 9-1 shows the classes predefined by SQL92.

Table 9-1 Predefined Class Codes

Class	Condition
00	success completion
01	warning
02	no data
07	dynamic SQL error
08	connection exception
09	triggered action exception
0A	feature not supported
0D	invalid target type specification
0E	invalid schema name list specification
0F	locator exception
0L	invalid grantor
OM	invalid SQL-invoked procedure reference
0P	invalid role specification
0S	invalid transform group name specification
0T	target table disagrees with cursor specification



Table 9-1 (Cont.) Predefined Class Codes

Class	Condition
0U	attempt to assign to non-updatable column
0V	attempt to assign to ordering column
0W	prohibited statement encountered during trigger execution
0Z	diagnostics exception
21	cardinality violation
22	data exception
23	integrity constraint violation
24	invalid cursor state
25	invalid transaction state
26	invalid SQL statement name
27	triggered data change violation
28	invalid authorization specification
2A	direct SQL syntax error or access rule violation
2B	dependent privilege descriptors still exist
2C	invalid character set name
2D	invalid transaction termination
2E	invalid connection name
2F	SQL routine exception
2H	invalid collation name
30	invalid SQL statement identifier
33	invalid SQL descriptor name
34	invalid cursor name
35	invalid condition number
36	cursor sensitivity exception
37	dynamic SQL syntax error or access rule violation
38	external routine exception
39	external routine invocation exception
3B	savepoint exception
3C	ambiguous cursor name
3D	invalid catalog name
3F	invalid schema name
40	transaction rollback
42	syntax error or access rule violation
44	with check option violation
HZ	remote database access



#### Note:

The class code HZ is reserved for conditions defined in International Standard ISO/IEC DIS 9579-2, *Remote Database Access*.

Table 9-2 shows how SQLSTATE status codes and conditions are mapped to Oracle errors. Status codes in the range 60000 to 99999 are implementation-defined.

Table 9-2 SQLSTATE Status Codes

Code	Condition	Oracle Error(s)
00000	successful completion	ORA-00000
01000	warning	
01001	cursor operation conflict	
01002	disconnect error	
01003	NULL value eliminated in set function	
01004	string data-right truncation	
01005	insufficient item descriptor areas	
01006	privilege not revoked	
01007	privilege not granted	
01008	implicit zero-bit padding	
01009	search condition too long for info schema	
0100A	query expression too long for info schema	
02000	no data	ORA-01095
		ORA-01403
07000	dynamic SQL error	
07001	using clause does not match parameter specs	
07002	using clause does not match target specs	
07003	cursor specification cannot be executed	
07004	using clause required for dynamic parameters	
07005	prepared statement not a cursor specification	
07006	restricted datatype attribute violation	
07007	using clause required for result components invalid descriptor count	
07008	invalid descriptor count	SQL-02126
07009	invalid descriptor index	
08000	connection exception	
08001	SQL-client unable to establish SQL-connection	
08002	connection name is use	
08003	connection does not exist	SQL-02121
08004	SQL-server rejected SQL-connection	
08006	connection failure	
08007	transaction resolution unknown	



Table 9-2 (Cont.) SQLSTATE Status Codes

Code	Condition	Oracle Error(s)
0A000	feature not supported	ORA-0300003099
0A001	multiple server transactions	
21000	cardinality violation	ORA-01427
		SQL-02112
22000	data exception	
22001	string data - right truncation	ORA-01406
22002	NULL value-no indicator parameter	SQL-02124
22003	numeric value out of range	ORA-01426
22005	error in assignment	
22007	invalid datetime format	
22008	datetime field overflow	ORA-0180001899
22009	invalid time zone displacement value	
22011	substring error	
22012	division by zero	ORA-01476
22015	interval field overflow	
22018	invalid character value for cast	
22019	invalid escape character	ORA-00911
22021	character not in repertoire	
22022	indicator overflow	ORA-01411
22023	invalid parameter value	ORA-01025
		ORA-0400004019
22024	unterminated C string	ORA-01479
		ORA-01480
22025	invalid escape sequence	ORA-01424
		ORA-01425
22026	string data-length mismatch	ORA-01401
22027	trim error	-
23000	integrity constraint violation	ORA-1400, ORA-0229002299
24000	invalid cursor state	ORA-001002
		ORA-001003
		SQL-02114
25000	invalid transaction state	SQL-02117
25000		SQL-02118
26000	invalid SQL statement name	<del></del>
27000	triggered data change violation	
28000	invalid authorization specification	
2A000	direct SQL syntax error or access rule violation	
2B000	dependent privilege descriptors still exist	
2C000	invalid character set name	
2D000	invalid transaction termination	

Table 9-2 (Cont.) SQLSTATE Status Codes

Code	Condition	Oracle Error(s)
2E000	invalid connection name	
33000	invalid SQL descriptor name	
34000	invalid cursor name	
35000	invalid condition number	
37000	dynamic SQL syntax error or access rule violation	
3C000	ambiguous cursor name	
3D000	invalid catalog name	
3F000	invalid schema name	
40000	transaction rollback	ORA-02091
		ORA-02092
40001	serialization failure	<del></del>
40002	integrity constraint violation	<del></del>
40003	statement completion unknown	
42000	syntax error or access rule violation	ORA-00022
000	oyman oner or access rais notation	ORA-00251
		ORA-0090000999
		ORA-01031
		ORA-0149001493
		ORA-0170001799
		ORA-0190002099
		ORA-0214002289
		ORA-0242002424
		ORA-0245002499 ORA-0327603299
		ORA-0404004059
		ORA-0407004099
44000	with check option violation	ORA-01402
60000	system error	ORA-0037000429
00000	oyotom on on	ORA-0060000899
		ORA-0643006449
		ORA-0720007999
		ORA-0970009999
61000	shared server and detached process errors	ORA-0001800035
		ORA-0005000068
		ORA-0237602399
		ORA-0402004039
62000	shared server and detached process errors	ORA-0010000120
		ORA-0044000569

Table 9-2 (Cont.) SQLSTATE Status Codes

Code	Condition	Oracle Error(s)
63000	Oracle*XA and two-task interface errors	ORA-0015000159
		ORA-0270002899
		ORA-0310003199
		ORA-0620006249
		SQL-02128
64000	control file, database file, and redo file errors; archival	ORA-0020000369
	and media recovery errors	ORA-0110001250
65000	PL/SQL errors	ORA-0650006599
66000	Oracle Net driver errors	ORA-0600006149
		ORA-0625006429
		ORA-0660006999
		ORA-1210012299
		ORA-1250012599
67000	licensing errors	ORA-0043000439
69000	SQL*Connect errors	ORA-0057000599
		ORA-0700007199
72000	SQL execute phase errors	ORA-00001
		ORA-0100001099
		ORA-0140101489
		ORA-0149501499
		ORA-0150001699
		ORA-0240002419
		ORA-0242502449
		ORA-0406004069 ORA-0800008190
		ORA-0800008190 ORA-1200012019
		ORA-1230012499
		ORA-1270021999
82100	out of memory (could not allocate)	SQL-02100
82101	inconsistent cursor cache (UCE/CUC mismatch)	SQL-02101
82102	inconsistent cursor cache (no CUC entry for UCE)	SQL-02102
82103	inconsistent cursor cache (out-or-range CUC ref)	SQL-02103
82104	inconsistent cursor cache (no CUC available)	SQL-02104
82105	inconsistent cursor cache (no CUC entry in cache)	SQL-02105
82106	inconsistent cursor cache (invalid cursor number)	SQL-02106
82107	program too old for runtime library; re-precompile	SQL-02107
82108	invalid descriptor passed to runtime library	SQL-02108
82109	inconsistent host cache (out-or-range SIT ref)	SQL-02109
82110	inconsistent host cache (invalid SQL type)	SQL-02110
82111	heap consistency error	SQL-02111
02111		

Table 9-2 (Cont.) SQLSTATE Status Codes

Code	Condition	Oracle Error(s)
82114	reentrant code generator gave invalid context	SQL-02116
82117	invalid OPEN or PREPARE for this connection	SQL-02122
82118	application context not found	SQL-02123
82119	unable to obtain error message text	SQL-02125
82120	Precompiler/SQLLIB version mismatch	SQL-02127
82121	NCHAR error; fetched number of bytes is odd	SQL-02129
82122	EXEC TOOLS interface not available	SQL-02130
82123	runtime context in use	SQL-02131
82124	unable to allocate runtime context	SQL-02132
82125	unable to initialize process for use with threads	SQL-02133
82126	invalid runtime context	SQL-02134
HZ000	remote database access	
-		

## 9.3.3 About Using SQLSTATE

The following rules apply to using SQLSTATE with SQLCODE or the SQLCA when you precompile with the option setting MODE=ANSI. SQLSTATE must be declared inside a Declare Section; otherwise, it is ignored.

### 9.3.3.1 If You Declare SQLSTATE

- Declaring SQLCODE is optional. If you declare SQLCODE inside the Declare Section, the Oracle Server returns status codes to SQLSTATE and SQLCODE after every SQL operation. However, if you declare SQLCODE outside of the Declare Section, Oracle returns a status code only to SQLSTATE.
- Declaring the SQLCA is optional. If you declare the SQLCA, Oracle returns status codes to SQLSTATE and the SQLCA. In this case, to avoid compilation errors, do not declare SQLCODE.

#### 9.3.3.2 If You Do not Declare SQLSTATE

- You must declare SQLCODE inside or outside the Declare Section. The Oracle Server returns a status code to SQLCODE after every SQL operation.
- Declaring the SQLCA is optional. If you declare the SQLCA, Oracle returns status codes to SQLCODE and the SQLCA.

You can learn the outcome of the most recent executable SQL statement by checking SQLSTATE explicitly with your own code or implicitly with the WHENEVER SQLERROR directive. Check SQLSTATE only after executable SQL statements and PL/SQL statements.

# 9.4 About Declaring SQLCODE

When MODE=ANSI, and you have not declared a SQLSTATE status variable, you must declare an **int** integer variable named SQLCODE inside or outside the Declare Section. An example follows:

```
/* declare host variables */
EXEC SQL BEGIN DECLARE SECTION;
int emp_number, dept_number;
char emp_name[20];
EXEC SQL END DECLARE SECTION;
/* declare status variable--must be upper case */
int SQLCODE;
```

When MODE=ORACLE, if you declare SQLCODE, it is not used.

You can declare more than one SQLCODE. Access to a local SQLCODE is limited by its scope within your program.

After every SQL operation, Oracle returns a status code to the SQLCODE currently in scope. So, your program can learn the outcome of the most recent SQL operation by checking SQLCODE explicitly, or implicitly with the WHENEVER directive.

When you declare SQLCODE instead of the SQLCA in a particular compilation unit, the precompiler allocates an internal SQLCA for that unit. Your host program cannot access the internal SQLCA. If you declare the SQLCA and SQLCODE, Oracle returns the same status code to both after every SQL operation.

# 9.5 Key Components of Error Reporting Using the SQLCA

Error reporting depends on variables in the SQLCA. This section highlights the key components of error reporting. The next section takes a close look at the SQLCA.

#### 9.5.1 Status Codes

Every executable SQL statement returns a status code to the SQLCA variable *sqlcode*, which you can check implicitly with the WHENEVER directive or explicitly with your own code.

A zero status code means that Oracle executed the statement without detecting an error or exception. A positive status code means that Oracle executed the statement but detected an exception. A negative status code means that Oracle did not execute the SQL statement because of an error.

## 9.5.2 Warning Flags

Warning flags are returned in the SQLCA variables sqlwarn[0] through sqlwarn[7], which you can check implicitly or explicitly. These warning flags are useful for runtime conditions not considered errors by Oracle. If no indicator variable is available, Oracle issues an error message.

### 9.5.3 Rows-Processed Count

The number of rows processed by the most recently executed SQL statement is returned in the SQLCA variable sqlca.sqlerrd[2], which you can check explicitly.

Strictly speaking, this variable is not for error reporting, but it can help you avoid mistakes. For example, suppose you expect to delete about ten rows from a table. After the deletion, you check sqlca.sqlerrd[2] and find that 75 rows were processed. To be safe, you might want to roll back the deletion and examine your WHERE-clause search condition.

#### 9.5.4 Parse Error Offsets

Before executing a SQL statement, Oracle must *parse* it to make sure it follows syntax rules and refers to valid database objects. If Oracle finds an error, an offset is stored in the SQLCA variable sqlca.sqlerrd[4], which you can check explicitly. The offset specifies the character position in the SQL statement at which the parse error begins. As in a normal C string, the first character occupies position zero. For example, if the offset is 9, the parse error begins at the 10th character.

The parse error offset is used for situations where a separate prepare/parse is performed. This is typical for dynamic SQL statements.

Parse errors may arise from missing, misplaced, or misspelled keywords, invalid options, and the like. For example, the dynamic SQL statement:

```
"UPDATE emp SET jib = :job title WHERE empno = :emp number"
```

#### causes the parse error

```
ORA-00904: invalid column name
```

because the column name JOB is misspelled. The value of sqlca.sqlerrd[4] is 15 because the erroneous column name JIB begins at the 16th character.

If your SQL statement does not cause a parse error, Oracle sets sqlca.sqlerrd[4] to zero. Oracle also sets sqlca.sqlerrd[4] to zero if a parse error begins at the first character (which occupies position zero). So, check sqlca.sqlerrd[4] only if sqlca.sqlcode is negative, which means that an error has occurred.

## 9.5.5 Error Message Text

The error code and message for Oracle errors are available in the SQLCA variable SQLERRMC. At most, the first 70 characters of text are stored. To get the full text of messages longer than 70 characters, you use the sqlglm() function.

#### **Related Topics**

About Getting the Full Text of Error Messages

# 9.6 Using the SQL Communications Area (SQLCA)

The SQLCA is a data structure. Its components contain error, warning, and status information updated by Oracle whenever a SQL statement is executed. Thus, the SQLCA always reflects the outcome of the most recent SQL operation. To determine the outcome, you can check variables in the SQLCA.

Your program can have more than one SQLCA. For example, it might have one global SQLCA and several local ones. Access to a local SQLCA is limited by its scope within the program. Oracle returns information only to the SQLCA that is in scope.





When your application uses Oracle Net to access a combination of local and remote databases concurrently, all the databases write to one SQLCA. There is *not* a different SQLCA for each database.

#### **Related Topics**

Advanced Connection Options

## 9.6.1 About Declaring the SQLCA

When MODE=ORACLE, declaring the SQLCA is required. To declare the SQLCA, you should copy it into your program with the INCLUDE or **#include** statement, as follows:

EXEC SQL INCLUDE SQLCA;

or

#include <sqlca.h>

If you use a Declare Section, the SQLCA must be declared *outside* the Declare Section. Not declaring the SQLCA results in compile-time errors.

When you precompile your program, the INCLUDE SQLCA statement is replaced by several variable declarations that allow Oracle to communicate with the program.

When MODE=ANSI, declaring the SQLCA is optional. But in this case you must declare a SQLCODE or SQLSTATE status variable. The type of SQLCODE (upper case is required) is **int**. If you declare SQLCODE or SQLSTATE instead of the SQLCA in a particular compilation unit, the precompiler allocates an internal SQLCA for that unit. Your Pro\*C/C++ program cannot access the internal SQLCA. If you declare the SQLCA and SQLCODE, Oracle returns the same status code to both after every SQL operation.

#### Note:

Declaring the SQLCA is optional when MODE=ANSI, but you cannot use the WHENEVER SQLWARNING directive without the SQLCA. So, if you want to use the WHENEVER SQLWARNING directive, you must declare the SQLCA.

This Guide uses SQLCODE when referring to the SQLCODE status variable, and *sqlca.sqlcode* when explicitly referring to the component of the SQLCA structure.

## 9.6.2 SQLCA Contents

The SQLCA contains the following runtime information about the outcome of SQL statements:

- Oracle error codes
- Warning flags
- Event information
- Rows-processed count



#### Diagnostics

```
The sqlca.h header file is:
```

```
NAME
  SQLCA: SQL Communications Area.
FUNCTION
 Contains no code. Oracle fills in the SQLCA with status info
  during the execution of a SQL stmt.
NOTES
  *************
  *** This file is SOSD. Porters must change the data types ***
  *** appropriately on their platform. See notes/pcport.doc ***
  *** for more information.
  *****************
 If the symbol SQLCA STORAGE CLASS is defined, then the SQLCA
  will be defined to have this storage class. For example:
    #define SQLCA STORAGE CLASS extern
 will define the SQLCA as an extern.
  If the symbol SQLCA INIT is defined, then the SQLCA will be
  statically initialized. Although this is not necessary in order
  to use the SQLCA, it is a good programing practice not to have
  unitialized variables. However, some C compilers/operating systems
  don't allow automatic variables to be initialized in this manner.
  Therefore, if you are INCLUDE'ing the SQLCA in a place where it
 would be an automatic AND your C compiler/operating system doesn't
  allow this style of initialization, then SQLCA INIT should be left
 undefined -- all others can define SQLCA INIT if they wish.
 If the symbol SQLCA NONE is defined, then the SQLCA
 variable will not be defined at all. The symbol SQLCA NONE
  should not be defined in source modules that have embedded SQL.
 However, source modules that have no embedded SQL, but need to
 manipulate a sqlca struct passed in as a parameter, can set the
 SQLCA NONE symbol to avoid creation of an extraneous sqlca
 variable.
#ifndef SQLCA
#define SQLCA 1
struct sqlca
        /* ub1 */ char
                       sqlcaid[8];
        /* b4 */ long
                       sqlabc;
        /* b4 */ int
                        sqlcode;
        struct
          /* ub2 */ unsigned short sqlerrml;
          /* ub1 */ char sqlerrmc[70];
          } sqlerrm;
        /* ub1 */ char
                        sqlerrp[8];
        /* b4 */ int sqlerrd[6];
        /* ub1 */ char sqlwarn[8];
        /* ub1 */ char
                         sqlext[8];
        };
#ifndef SQLCA NONE
#ifdef SQLCA_STORAGE_CLASS
```

## 9.6.3 SQLCA Structure

This section describes the structure of the SQLCA, its components, and the values they can store.

#### 9.6.3.1 sqlcaid

This string component is initialized to "SQLCA" to identify the SQL Communications Area.

### 9.6.3.2 *sqlcabc*

This integer component holds the length, in bytes, of the SQLCA structure.

### 9.6.3.3 sqlcode

This integer component holds the status code of the most recently executed SQL statement. The status code, which indicates the outcome of the SQL operation, can be any of the following numbers:

Status Codes	Description
0	Means that Oracle executed the statement without detecting an error or exception.
>0	Means that Oracle executed the statement but detected an exception. This occurs when Oracle cannot find a row that meets your WHERE-clause search condition or when a SELECT INTO or FETCH returns no rows.

When MODE=ANSI, +100 is returned to *sqlcode* after an INSERT of no rows. This can happen when a subquery returns no rows to process.

 <0 - Means that Oracle did not execute the statement because of a database, system, network, or application error. Such errors can be fatal. When they occur, the current transaction should, in most cases, be rolled back.

Negative return codes correspond to error codes listed in Oracle Database Error Messages

#### 9.6.3.4 sqlerrm

This embedded struct contains the following two components:

Components	Description
sqlerrml	This integer component holds the length of the message text stored in sqlerrmc.
sqlerrmc	This string component holds the message text corresponding to the error code stored in sqlcode. The string is not null terminated. Use the sqlerrml component to determine the length.

This component can store up to 70 characters. To get the full text of messages longer than 70 characters, you must use the sqlglm() function (discussed later).

Make sure *sqlcode* is negative *before* you reference *sqlerrmc*. If you reference *sqlerrmc* when *sqlcode* is zero, you get the message text associated with a prior SQL statement.

#### 9.6.3.5 sqlerrp

This string component is reserved for future use.

### 9.6.3.6 *sqlerrd*

This array of binary integers has six elements. Descriptions of the components in *sqlerrd* follow:

Components	Description
sqlerrd[0]	This component is reserved for future use.
sqlerrd[1]	This component is reserved for future use.
sqlerrd[2]	This component holds the number of rows processed by the most recently executed SQL statement. However, if the SQL statement failed, the value of sqlca.sqlerrd[2] is undefined, with one exception. If the error occurred during an array operation, processing stops at the row that caused the error, so sqlca.sqlerrd[2] gives the number of rows processed successfully.

The rows-processed count is zeroed after an OPEN statement and incremented after a FETCH statement. For the EXECUTE, INSERT, UPDATE, DELETE, and SELECT INTO statements, the count reflects the number of rows processed successfully. The count does *not* include rows processed by an UPDATE or DELETE CASCADE. For example, if 20 rows are deleted because they meet WHERE-clause criteria, and 5 more rows are deleted because they now (after the primary delete) violate column constraints, the count is 20 not 25.

Components	Description
sqlerrd[3]	This component is reserved for future use.
sqlerrd[4]	This component holds an offset that specifies the character position at which a parse error begins in the most recently executed SQL statement. The first character occupies position zero.
sqlerrd[5]	This component is reserved for future use.



#### 9.6.3.7 sqlwarn

This array of single characters has eight elements. They are used as warning flags. Oracle sets a flag by assigning it a "W" (for warning) character value.

The flags warn of exceptional conditions. For example, a warning flag is set when Oracle assigns a truncated column value to an output host variable.

Descriptions of the components in *sqlwarn* follow:

Components	Description
sqlwarn[0]	This flag is set if another warning flag is set.
sqlwarn[1]	This flag is set if a truncated column value was assigned to an output host variable. This applies only to character data. Oracle truncates certain numeric data without setting a warning or returning a negative sqlcode.

To find out if a column value was truncated and by how much, check the indicator variable associated with the output host variable. The (positive) integer returned by an indicator variable is the original length of the column value. You can increase the length of the host variable accordingly.

Components	Description
sqlwarn[2]	This flag is set if a NULL column is not used in the result of a SQL group function, such as AVG() or SUM().
sqlwarn[3]	This flag is set if the number of columns in a query select list does not equal the number of host variables in the INTO clause of the SELECT or FETCH statement. The number of items returned is the lesser of the two.
sqlwarn[4]	This flag is no longer in use.
sqlwarn[5]	This flag is set when an EXEC SQL CREATE {PROCEDURE   FUNCTION   PACKAGE   PACKAGE BODY} statement fails because of a PL/SQL compilation error.
sqlwarn[6]	This flag is no longer in use.
sqlwarn[7]	This flag is no longer in use.

#### 9.6.3.8 sqlext

This string component is reserved for future use.

## 9.6.4 PL/SQL Considerations

When the precompiler application executes an embedded PL/SQL block, not all components of the SQLCA are set. For example, if the block fetches several rows, the rows-processed count (sqlerrd[2]) is set to only 1. You should depend only on the sqlcode and sqlerrm components of the SQLCA after execution of a PL/SQL block.

# 9.7 About Getting the Full Text of Error Messages

The SQLCA can accommodate error messages up to 70 characters long. To get the full text of longer (or nested) error messages, you need to use the sqlglm() function. The syntax is

#### where:

Syntax	Description
message_buffer	Is the text buffer in which you want Oracle to store the error message (Oracle blank-pads to the end of this buffer).
buffer_size	Is a scalar variable that specifies the maximum size of the buffer in bytes.
message_length	Is a scalar variable in which Oracle stores the actual length of the error message, if not truncated.

#### Note:

The types of the last two arguments for the sqlglm() function are shown here generically as  $size_t$  pointers. However on your platform they might have a different type. For example, on many UNIX workstation ports, they are unsigned int \*.

You should check the file sqlcpr.h, which is in the standard include directory on your system, to determine the datatype of these parameters.

The maximum length of an Oracle error message is 512 characters including the error code, nested messages, and message inserts such as table and column names. The maximum length of an error message returned by sqlglm() depends on the value you specify for buffer size.

The following example calls sqlglm() to get an error message of up to 200 characters in length:

```
EXEC SQL WHENEVER SQLERROR DO sql_error();
...
/* other statements */
...
sql_error()
{
    char msg[200];
    size_t buf_len, msg_len;

    buf_len = sizeof (msg);
    sqlglm(msg, &buf_len, &msg_len); /* note use of pointers */
    if (msg_len > buf_len)
    msg_len = buf_len;
    printf("%.*s\n\n", msg_len, msg);
    exit(1);
}
```

Notice that sqlglm() is called only when a SQL error has occurred. Always make sure SQLCODE (or sqlca.sqlcode) is nonzero before calling sqlglm. If you call sqlglm() when SQLCODE is zero, you get the message text associated with a prior SQL statement.



In cases where multiple runtime contexts are used, use the version of sqlglmt() that takes a context to get the correct error message.

#### **Related Topics**

Multithreaded Applications

## 9.8 About Using the WHENEVER Directive

By default, precompiled programs ignore Oracle error and warning conditions and continue processing if possible. To do automatic condition checking and error handling, you need the WHENEVER directive.

With the WHENEVER directive you can specify actions to be taken when Oracle detects an error, warning condition, or "not found" condition. These actions include continuing with the next statement, calling a routine, branching to a labeled statement, or stopping.

You code the WHENEVER directive using the following syntax:

EXEC SQL WHENEVER <condition> <action>;

#### 9.8.1 WHENEVER Conditions

You can have Oracle automatically check the SQLCA for any of the following conditions.

### 9.8.1.1 SQLWARNING

sqlwarn[0] is set because Oracle returned a warning (one of the warning flags, sqlwarn[1] through sqlwarn[7], is also set) or SQLCODE has a positive value other than +1403. For example, sqlwarn[0] is set when Oracle assigns a truncated column value to an output host variable.

Declaring the SQLCA is optional when MODE=ANSI. To use WHENEVER SQLWARNING, however, you *must* declare the SQLCA.

### **9.8.1.2 SQLERROR**

SQLCODE has a negative value because Oracle returned an error.

#### 9.8.1.3 NOT FOUND

SQLCODE has a value of +1403 (+100 when MODE=ANSI) because Oracle could not find a row that meets your WHERE-clause search condition, or a SELECT INTO or FETCH returned no rows.

When MODE=ANSI, +100 is returned to SQLCODE after an INSERT of no rows.

### 9.8.2 WHENEVER Actions

When Oracle detects one of the preceding *conditions*, you can have your program take any of the following actions.

#### 9.8.2.1 CONTINUE

Your program continues to run with the next statement if possible. This is the default action, equivalent to not using the WHENEVER directive. You can use it to turn off condition checking.

#### 9.8.2.2 DO

Your program transfers control to an error handling function in the program. When the end of the routine is reached, control transfers to the statement that follows the failed SQL statement.

The usual rules for entering and exiting a function apply. You can pass parameters to the error handler invoked by an EXEC SQL WHENEVER ... DO ... directive, and the function can return a value.

#### 9.8.2.3 DO BREAK

An actual "break" statement is placed in your program. Use this action in loops. When the WHENEVER condition is met, your program exits the loop it is inside.

#### **9.8.2.4 DO CONTINUE**

An actual "continue" statement is placed in your program. Use this action in loops. When the WHENEVER condition is met, your program continues with the next iteration of the loop it is inside.

## 9.8.2.5 GOTO label\_name

Your program branches to a labeled statement. Label names can be any length, but only the first 31 characters are significant. Your C compiler might require a different maximum length. Check your C compiler user's guide.

#### 9.8.2.6 STOP

Your program stops running and uncommitted work is rolled back.

STOP in effect just generates an exit() call whenever the condition occurs. Be careful. The STOP action displays no messages before disconnecting from Oracle.

### 9.8.3 WHENEVER Examples

If you want your program to

- Go to close\_cursor if a "no data found" condition occurs
- Continue with the next statement if a warning occurs
- Go to error handler if an error occurs

you must code the following WHENEVER directives before the first executable SQL statement:

```
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;
EXEC SQL WHENEVER SQLWARNING CONTINUE;
EXEC SQL WHENEVER SQLERROR GOTO error handler;
```

In the following example, you use WHENEVER...DO directives to handle specific errors:

```
EXEC SQL WHENEVER SQLERROR DO handle insert error("INSERT error");
EXEC SQL INSERT INTO emp (empno, ename, deptno)
    VALUES (:emp number, :emp_name, :dept_number);
EXEC SQL WHENEVER SQLERROR DO handle_delete_error("DELETE error");
EXEC SQL DELETE FROM dept WHERE deptno = :dept number;
handle insert error(char *stmt)
{ switch(sqlca.sqlcode)
   case -1:
    /* duplicate key value */
       break;
    case -1401:
    /* value too large */
        break;
    default:
    /* do something here too */
        . . .
        break;
}
handle delete error(char *stmt)
    printf("%s\n\n", stmt);
   if (sqlca.sqlerrd[2] == 0)
        /* no rows deleted */
        . . .
    }
    else
    }
```

Notice how the procedures check variables in the SQLCA to determine a course of action.

#### 9.8.4 Use of DO BREAK and DO CONTINUE

This example illustrates how to display employee name, salary, and commission for only those employees who receive commissions:

```
#include <sqlca.h>
#include <stdio.h>

main()
{
    char *uid = "scott/tiger";
    struct { char ename[12]; float sal; float comm; } emp;

    /* Trap any connection error that might occur. */
    EXEC SQL WHENEVER SQLERROR GOTO whoops;
    EXEC SQL CONNECT :uid;

EXEC SQL DECLARE c CURSOR FOR
    SELECT ename, sal, comm FROM EMP ORDER BY ENAME ASC;
```

```
EXEC SQL OPEN c;
    /* Set up 'BREAK' condition to exit the loop. */
   EXEC SQL WHENEVER NOT FOUND DO BREAK;
   /* The DO CONTINUE makes the loop start at the next iteration when an error occurs.*/
   EXEC SQL WHENEVER SQLERROR DO CONTINUE;
    while (1)
      {
          EXEC SQL FETCH c INTO :emp;
   ^{\prime\prime} An ORA-1405 would cause the 'continue' to occur. So only employees with ^{\star\prime}
   /* non-NULL commissions will be displayed. */
          printf("%s %7.2f %9.2f\n", emp.ename, emp.sal, emp.comm);
/* This 'CONTINUE' shuts off the 'DO CONTINUE' allowing the program to
  proceed if any further errors do occur, specifically, with the CLOSE */
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL CLOSE c;
    exit(EXIT SUCCESS);
whoops:
   printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
   exit(EXIT FAILURE);
```

## 9.8.5 Scope of WHENEVER

Because WHENEVER is a declarative statement, its scope is positional, not logical. That is, it tests all executable SQL statements that *physically* follow it in the source file, not in the flow of program logic. So, code the WHENEVER directive before the first executable SQL statement you want to test.

A WHENEVER directive stays in effect until superseded by another WHENEVER directive checking for the same condition.

In the following example, the first WHENEVER SQLERROR directive is superseded by a second, and so applies only to the CONNECT statement. The second WHENEVER SQLERROR directive applies to both the UPDATE and DROP statements, despite the flow of control from *step1* to *step3*.

```
step1:
    EXEC SQL WHENEVER SQLERROR STOP;
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    ...
    goto step3;
step2:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL UPDATE emp SET sal = sal * 1.10;
    ...
step3:
    EXEC SQL DROP INDEX emp_index;
    ...
```

### 9.8.6 Guidelines for WHENEVER

The following guidelines will help you avoid some common pitfalls.

## 9.8.6.1 Placing the Statements

In general, code a WHENEVER directive before the first executable SQL statement in your program. This ensures that all ensuing errors are trapped because WHENEVER directives stay in effect to the end of a file.

# 9.8.6.2 Handling End-of-Data Conditions

Your program should be prepared to handle an end-of-data condition when using a cursor to fetch rows. If a FETCH returns no data, the program should exit the fetch loop, as follows:

```
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
    EXEC SQL FETCH...
}
EXEC SQL CLOSE my_cursor;
...
```

An INSERT can return NOT FOUND if no rows have been inserted. If you do not want to catch that condition, use the EXEC SQL WHENEVER NOT FOUND CONTINUE statement before the INSERT:

```
EXEC SQL WHENEVER NOT FOUND DO break;
for(;;)
{
    EXEC SQL FETCH ...
    EXEC SQL WHENEVER NOT FOUND CONTINUE;
    EXEC SQL INSERT INTO ...
}
EXEC SQL CLOSE my_cursor;
```

## 9.8.6.3 About Avoiding Infinite Loops

If a WHENEVER SQLERROR GOTO directive branches to an error handling routine that includes an executable SQL statement, your program might enter an infinite loop if the SQL statement fails with an error. You can avoid this by coding WHENEVER SQLERROR CONTINUE before the SQL statement, as shown in the following example:

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...
sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
...
```

Without the WHENEVER SQLERROR CONTINUE statement, a ROLLBACK error would invoke the routine again, starting an infinite loop.

Careless use of WHENEVER can cause problems. For example, the following code enters an infinite loop if the DELETE statement sets NOT FOUND because no rows meet the search condition:

```
/* improper use of WHENEVER */
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
for (;;)
{
```

```
EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
...
}
no_more:
    EXEC SQL DELETE FROM emp WHERE empno = :emp_number;
...
```

The next example handles the NOT FOUND condition properly by resetting the GOTO target:

```
/* proper use of WHENEVER */
...
EXEC SQL WHENEVER NOT FOUND GOTO no_more;
for (;;)
{
    EXEC SQL FETCH emp_cursor INTO :emp_name, :salary;
    ...
}
no_more:
    EXEC SQL WHENEVER NOT FOUND GOTO no_match;
    EXEC SQL DELETE FROM emp WHERE empno = :emp_number;
...
no_match:
```

## 9.8.6.4 About Maintaining Addressability

Make sure all SQL statements governed by a WHENEVER GOTO directive can branch to the GOTO label. The following code results in a compile-time error because *labelA* in *func1* is not within the scope of the INSERT statement in *func2*:

```
func1()
{
    EXEC SQL WHENEVER SQLERROR GOTO labelA;
    EXEC SQL DELETE FROM emp WHERE deptno = :dept_number;
    ...
labelA:
    ...
}
func2()
{
    EXEC SQL INSERT INTO emp (job) VALUES (:job_title);
    ...
}
```

The label to which a WHENEVER GOTO directive branches must be in the same precompilation file as the statement.

## 9.8.6.5 About Returning After an Error

If your program must return after handling an error, use the DO *routine\_call* action. Alternatively, you can test the value of *sqlcode*, as shown in the following example:

```
EXEC SQL UPDATE emp SET sal = sal * 1.10;
if (sqlca.sqlcode < 0)
{    /* handle error */

EXEC SQL DROP INDEX emp index;</pre>
```



Just make sure no WHENEVER GOTO or WHENEVER STOP directive is active.

# 9.9 About Obtaining the Text of SQL Statements

In many precompiler applications it is convenient to know the text of the statement being processed, its length, and the SQL command (such as INSERT or SELECT) that it contains. This is especially true for applications that use dynamic SQL.

The SQLStmtGetText() function (old name:sqlgls() function)—part of the SQLLIB runtime library—returns the following information:

- The text of the most recently parsed SQL statement
- The effective length of the statement
- A function code for the SQL command used in the statement

SQLStmtGetText() is thread-safe. You can call SQLStmtGetText() after issuing a static SQL statement. For dynamic SQL Method 1, call SQLStmtGetText() after the SQL statement is executed. For dynamic SQL Methods 2, 3, and 4, you can call SQLStmtGetText() as soon as the statement has been PREPAREd.

For the new names of all the SQLLIB functions, see also "New Names for SQLLIB Public Functions".

The prototype for SQLStmtGetText() is

```
void SQLStmtGetText(dvoid *context, char *sqlstm, size t *stmlen, size t *sqlfc);
```

The context parameter is the runtime context. For definition and use of contexts, see "CONTEXT Variables".

The *sqlstm* parameter is a character buffer that holds the returned text of the SQL statement. Your program must statically declare the buffer or dynamically allocate memory for the buffer.

The stmlen parameter is a size\_t variable. Before calling sqlstmtGetText(), set this parameter to the actual size, in bytes, of the sqlstm buffer. When sqlstmtGetText() returns, the sqlstm buffer contains the SQL statement text, blank padded to the length of the buffer. The stmlen parameter returns the actual number of bytes in the returned statement text, not counting blank padding. The maximum value of stmlen is port-specific and generally will be the maximum integer size.

The *sqlfc* parameter is a size\_t variable that returns the SQL function code for the SQL command in the statement. Table 9-3 shows the SQL function codes for the commands.

Table 9-3 SQL Function Codes

Code	SQL Function	Code	SQL Function	Code	SQL Function
01	CREATE TABLE	26	ALTER TABLE	51	DROP TABLESPACE
02	SET ROLE	27	EXPLAIN	52	ALTER SESSION
03	INSERT	28	GRANT	53	ALTER USER
04	SELECT	29	REVOKE	54	COMMIT
05	UPDATE	30	CREATE SYNONYM	55	ROLLBACK
06	DROP ROLE	31	DROP SYNONYM	56	SAVEPOINT
07	DROP VIEW	32	ALTER SYSTEM SWITCH LOG	57	CREATE CONTROL FILE



Table 9-3 (Cont.) SQL Function Codes

Code	SQL Function	Code	SQL Function	Code	SQL Function
08	DROP TABLE	33	SET TRANSACTION	58	ALTER TRACING
09	DELETE	34	PL/SQL EXECUTE	59	CREATE TRIGGER
10	CREATE VIEW	35	LOCK TABLE	60	ALTER TRIGGER
11	DROP USER	36	(NOT USED)	61	DROP TRIGGER
12	CREATE ROLE	37	RENAME	62	ANALYZE TABLE
13	CREATE SEQUENCE	38	COMMENT	63	ANALYZE INDEX
14	ALTER SEQUENCE	39	AUDIT	64	ANALYZE CLUSTER
15	(NOT USED)	40	NOAUDIT	65	CREATE PROFILE
16	DROP SEQUENCE	41	ALTER INDEX	66	DROP PROFILE
17	CREATE SCHEMA	42	CREATE EXTERNAL DATABASE	67	ALTER PROFILE
18	CREATE CLUSTER	43	DROP EXTERNAL DATABASE	68	DROP PROCEDURE
19	CREATE USER	44	CREATE DATABASE	69	(NOT USED)
20	CREATE INDEX	45	ALTER DATABASE	70	ALTER RESOURCE COST
21	DROP INDEX	46	CREATE ROLLBACK SEGMENT	71	CREATE SNAPSHOT LOG
22	DROP CLUSTER	47	ALTER ROLLBACK SEGMENT	72	ALTER SNAPSHOT LOG
23	VALIDATE INDEX	48	DROP ROLLBACK SEGMENT	73	DROP SNAPSHOT LOG
24	CREATE PROCEDURE	49	CREATE TABLESPACE	74	CREATE SNAPSHOT
25	ALTER PROCEDURE	50	ALTER TABLESPACE	75	ALTER SNAPSHOT
				76	DROP
					SNAPSHOT

The length parameter (*stmlen*) returns a zero if an error occurred. Possible error conditions are:

- No SQL statement has been parsed.
- You passed an invalid parameter (for example, a negative length parameter).
- An internal exception occurred in SQLLIB.

# 9.9.1 Restrictions (using SQLStmtGetText())

 ${\tt SQLStmtGetText} \ () \ \ \textbf{does not return the text for statements that contain the following commands:}$ 

- CONNECT
- COMMIT
- ROLLBACK

#### FETCH

There are no SQL function codes for these commands.

# 9.9.2 Example Program

The example program sqlvcp.pc, is available in the demo directory. It demonstrates how you can use the sqlgls() function.

#### **Related Topics**

Datatypes and Host Variables

# 9.10 About Using the Oracle Communications Area (ORACA)

The SQLCA handles standard SQL communications The ORACA handles Oracle communications. When you need more information about runtime errors and status changes than the SQLCA provides, use the ORACA. It contains an extended set of diagnostic tools. However, use of the ORACA is optional because it adds to runtime overhead.

Besides helping you to diagnose problems, the ORACA lets you monitor your program's use of Oracle resources such as the SQL Statement Executor and the cursor cache.

Your program can have more than one ORACA. For example, it might have one global ORACA and several local ones. Access to a local ORACA is limited by its scope within the program. Oracle returns information only to the ORACA that is in scope.

# 9.10.1 About Declaring the ORACA

To declare the ORACA, copy it into your program with the INCLUDE statement or the **#include** preprocessor directive, as follows:

```
EXEC SQL INCLUDE ORACA;

Or
#include <oraca.h>
```

If your ORACA must be of the **extern** storage class, define ORACA\_STORAGE\_CLASS in your program as follows:

```
#define ORACA_STORAGE_CLASS extern
```

If the program uses a Declare Section, the ORACA must be defined outside it.

# 9.10.2 About Enabling the ORACA

To enable the ORACA, you must specify the ORACA option, either on the command line with

ORACA=YES

#### or inline with

```
EXEC ORACLE OPTION (ORACA=YES);
```

Then, you must choose appropriate runtime options by setting flags in the ORACA.

## 9.10.3 ORACA Contents

The ORACA contains option settings, system statistics, and extended diagnostics such as

- SQL statement text (you can specify when to save the text)
- The name of the file in which an error occurred (useful when using subroutines)
- Location of the error in a file
- Cursor cache errors and statistics

A partial listing of oraca.h is

```
/*
NAME
 ORACA: Oracle Communications Area.
  If the symbol ORACA NONE is defined, then there will be no ORACA
  *variable*, although there will still be a struct defined. This
 macro should not normally be defined in application code.
 If the symbol ORACA_INIT is defined, then the ORACA will be
  statically initialized. Although this is not necessary in order
  to use the ORACA, it is a good pgming practice not to have
 unitialized variables. However, some C compilers/operating systems
 don't allow automatic variables to be init'd in this manner. Therefore,
 if you are INCLUDE'ing the ORACA in a place where it would be
 an automatic AND your C compiler/operating system doesn't allow this style
  of initialization, then ORACA INIT should be left undefined --
 all others can define ORACA INIT if they wish.
#ifndef ORACA
#define ORACA
struct
        oraca
   char oracaid[8]; /* Reserved
                    /* Reserved
   long oracabc;
   Flags which are setable by User. */
  long orastxtf; /* SQL stmt text flag
#define ORASTFNON 0 /* = don't save text of SQL stmt
#define ORASTFERR 1 /* = only save on SQLERROR */
\#define ORASTFWRN 2 /* = only save on SQLWARNING/SQLERROR */
#define ORASTFANY 3 /* = always save
   struct
     {
 unsigned short orastxtl;
  char orastxtc[70];
    } orastxt;
                     /* text of last SQL stmt
   struct
    {
  unsigned short orasfnml;
  char orasfnmc[70];
   } orasfnm; /* name of file containing SQL stmt
  long oraslnr;
                     /* line nr-within-file of SQL stmt
```

```
/* highest max open OraCurs requested */
 long orahoc;
                     /* max open OraCursors required
 long oramoc;
 long oracoc;
                      /* current OraCursors open
 long oranor;
                      /* nr of OraCursor re-assignments
                      /* nr of parses
 long oranpr;
                                          */
                      /* nr of executes
 long oranex;
   };
#ifndef ORACA NONE
#ifdef ORACA STORAGE CLASS
ORACA STORAGE CLASS struct oraca oraca
#else
struct oraca oraca
#endif
#ifdef ORACA INIT
   {'O', 'R', 'A', 'C', 'A', ' ', ' ', ' '},
   sizeof(struct oraca),
   0,0,0,0,
   {0,{0}},
   {0,{0}},
   Ο,
   0,0,0,0,0,0
   }
#endif
   ;
#endif
#endif
/* end oraca.h */
```

# 9.10.4 About Choosing Runtime Options

The ORACA includes several option flags. Setting these flags by assigning them nonzero values provides the ability to

- Save the text of SQL statements
- Enable DEBUG operations
- Check cursor cache consistency (the cursor cache is a continuously updated area of memory used for cursor management)
- Check heap consistency (the heap is an area of memory reserved for dynamic variables)
- Gather cursor statistics

The following descriptions will help you choose the options you need.

## 9.10.5 Structure of the ORACA

This section describes the structure of the ORACA, its components, and the values they can store.

## 9.10.5.1 oracaid

This string component is initialized to "ORACA" to identify the Oracle Communications Area.

## 9.10.5.2 oracabc

This integer component holds the length, in bytes, of the ORACA data structure.

### 9.10.5.3 oracchf

If the master DEBUG flag (*oradbgf*) is set, this flag enables the gathering of cursor cache statistics and lets you check the cursor cache for consistency before every cursor operation.

The Oracle runtime library does the consistency checking and might issue error messages, which are listed in the manual *Oracle Database Error Messages*. They are returned to the SQLCA just like Oracle error messages.

This flag has the following settings:

- Disable cache consistency checking (the default).
- Enable cache consistency checking.

## 9.10.5.4 *oradbgf*

This master flag lets you choose all the DEBUG options. It has the following settings:

Disable all DEBUG operations (the default).

Enable all DEBUG operations.

### 9.10.5.5 orahchf

If the master DEBUG flag (*oradbgf*) is set, this flag tells the Oracle runtime library to check the heap for consistency every time the precompiler dynamically allocates or frees memory. This is useful for detecting program bugs that upset memory.

This flag must be set before the CONNECT command is issued and, once set, cannot be cleared; subsequent change requests are ignored. It has the following settings:

- Disable heap consistency checking (the default).
- Enable heap consistency checking.

## 9.10.5.6 orastxtf

This flag lets you specify when the text of the current SQL statement is saved. It has the following settings:

- Never save the SQL statement text (the default).
- · Save the SQL statement text on SQLERROR only.
- Save the SQL statement text on SQLERROR or SQLWARNING.
- Always save the SQL statement text.

The SQL statement text is saved in the ORACA embedded struct named orastxt.

## 9.10.5.7 Diagnostics

The ORACA provides an enhanced set of diagnostics; the following variables help you to locate errors quickly:



## 9.10.5.8 orastxt

This embedded struct helps you find faulty SQL statements. It lets you save the text of the last SQL statement parsed by Oracle. It contains the following two components:

Components	Description
orastxtl	This integer component holds the length of the current SQL statement.
orastxtc	This string component holds the text of the current SQL statement. At most, the first 70 characters of text are saved. The string is not null terminated. Use the oratxtl length component when printing the string.

Statements parsed by the precompiler, such as CONNECT, FETCH, and COMMIT, are *not* saved in the ORACA.

## 9.10.5.9 orasfnm

This embedded struct identifies the file containing the current SQL statement and so helps you find errors when multiple files are precompiled for one application. It contains the following two components:

Components	Description
orasfnml	This integer component holds the length of the filename stored in orasfnmc.
orasfnmc	This string component holds the filename. At most, the first 70 characters are stored.

### 9.10.5.10 orasInr

This integer component identifies the line at (or near) which the current SQL statement can be found.

### 9.10.5.11 Cursor Cache Statistics

If the master DEBUG flag (*oradbgf*) and the cursor cache flag (*oracchf*) are set, the following variables let you gather cursor cache statistics. They are automatically set by every COMMIT or ROLLBACK command your program issues.

Internally, there is a set of these variables for each CONNECTed database. The current values in the ORACA pertain to the database against which the last COMMIT or ROLLBACK was executed:

### 9.10.5.12 orahoc

This integer component records the highest value to which MAXOPENCURSORS was set during program execution.

### 9.10.5.13 oramoc

This integer component records the maximum number of open Oracle cursors required by your program. This number can be higher than *orahoc* if MAXOPENCURSORS was set too low, which forced the precompiler to extend the cursor cache.

## 9.10.5.14 oracoc

This integer component records the current number of open Oracle cursors required by your program.

## 9.10.5.15 oranor

This integer component records the number of cursor cache reassignments required by your program. This number shows the degree of "thrashing" in the cursor cache and should be kept as low as possible.

## 9.10.5.16 oranpr

This integer component records the number of SQL statement parses required by your program.

### 9.10.5.17 oranex

This integer component records the number of SQL statement executions required by your program. The ratio of this number to the *oranpr* number should be kept as high as possible. In other words, avoid unnecessary re-parsing.

#### **Related Topics**

Performance Tuning

# 9.10.6 ORACA Example

The following program prompts for a department number, inserts the name and salary of each employee in that department into one of two tables, then displays diagnostic information from the ORACA. This program is available online in the demo directory, as oraca.pc.

```
/* oraca.pc
 * This sample program demonstrates how to
 * use the ORACA to determine various performance
 * parameters at runtime.
#include <stdio.h>
#include <string.h>
#include <sqlca.h>
#include <oraca.h>
EXEC SQL BEGIN DECLARE SECTION;
char *userid = "SCOTT/TIGER";
char emp name[21];
int dept number;
float salary;
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
void sql error();
main()
    char temp buf[32];
```



```
EXEC SQL WHENEVER SQLERROR DO sql error("Oracle error");
    EXEC SQL CONNECT :userid;
    EXEC ORACLE OPTION (ORACA=YES);
    oraca.oradbgf = 1;
                                  /* enable debug operations */
                         /* gather cursor cache statistics */
    oraca.oracchf = 1;
    oraca.orastxtf = 3;
                           /* always save the SQL statement */
   printf("Enter department number: ");
    gets(temp buf);
   dept number = atoi(temp buf);
   EXEC SQL DECLARE emp cursor CURSOR FOR
     SELECT ename, sal + NVL(comm,0) AS sal comm
       FROM emp
       WHERE deptno = :dept number
       ORDER BY sal comm DESC;
    EXEC SQL OPEN emp cursor;
    EXEC SQL WHENEVER NOT FOUND DO sql error ("End of data");
    for (;;)
       EXEC SQL FETCH emp cursor INTO :emp name, :salary;
       printf("%.10s\n", emp_name);
       if (salary < 2500)
           EXEC SQL INSERT INTO pay1 VALUES (:emp name, :salary);
       else
           EXEC SQL INSERT INTO pay2 VALUES (:emp name, :salary);
void
sql_error(errmsg)
char *errmsg;
{
   char buf[6];
    strcpy(buf, SQLSTATE);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL COMMIT WORK RELEASE;
    if (strncmp(errmsg, "Oracle error", 12) == 0)
       printf("\n%s, sqlstate is %s\n\n", errmsg, buf);
    else
       printf("\n%s\n\n", errmsg);
   printf("Last SQL statement: %.*s\n",
   oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf("\nAt or near line number %d\n", oraca.oraslnr);
   printf
("\nCursor Cache Statistics\n----\n");
   printf
("Maximum value of MAXOPENCURSORS:
                                    %d\n", oraca.orahoc);
   printf
                                    %d\n", oraca.oramoc);
("Maximum open cursors required:
   printf
("Current number of open cursors:
                                    %d\n", oraca.oracoc);
   printf
("Number of cache reassignments:
                                    %d\n", oraca.oranor);
   printf
```

```
("Number of SQL statement parses: %d\n", oraca.oranpr);
   printf
("Number of SQL statement executions: %d\n", oraca.oranex);
   exit(1);
}
```



10

# **Precompiler Options**

This chapter tells you how to run the Pro\*C/C++ precompiler, and describes the extensive set of precompiler options in detail. This chapter contains the following topics:

- The Precompiler Command
- · Precompiler Options
- Quick Reference
- Entering Options
- About Using Domain Metadata
- About Using the Precompiler Options

# 10.1 The Precompiler Command

The location of the precompiler differs from system to system. The system or database administrator usually defines logicals or aliases, or uses other system-specific means to make the Pro\*C/C++ executable accessible.

To run the Pro\*C/C++ precompiler, you issue the following command:

proc option=value...



The option value is always separated from the option name by an equals sign, with no whitespace around the equals sign.

#### For example, the command

proc INAME=test proc

precompiles the file  $test\_proc.pc$  in the current directory, since the precompiler assumes that the filename extension is pc. The INAME=argument specifies the source file to be precompiled. The INAME option does not have to be the first option on the command line, but if it is, you can omit the option specification. So, the command

proc myfile

#### is equivalent to

proc INAME=myfile



The option names, and option values that do not name specific operating system objects, such as filenames, are not case-sensitive. In the examples in this guide, option names are written in upper case, and option values are usually in lower case. When you enter filenames, including the name of the Pro\*C/C++ precompiler executable itself, always follow the case conventions used by your operating system.

Some platforms, such as UNIX, require "escape characters" before certain characters in value strings. Consult your platform-specific documentation.

# 10.1.1 Case Sensitivity

In general, you can use either uppercase or lowercase for precompiler option names and values. However, if your operating system is case sensitive, like UNIX, you must specify filename values, including the name of the Pro\*C/C++ executable, using the correct combination of uppercase and lowercase letters.

# 10.2 Precompiler Options

Precompiler options enable you to control how resources are used, how errors are reported, how input and output are formatted, and how cursors are managed.

The value of an option is a literal, which represents text or numeric values. For example, for the option

```
... INAME=my test
```

the value is a string literal that specifies a filename.

For the option MAXOPENCURSORS

```
...MAXOPENCURSORS=20
```

the value is numeric.

Some options take Boolean values, and you can represent these with the strings *yes* or *no*, *true* or *false*, or with the integer literals 1 or 0 respectively. For example, the option

```
... SELECT_ERROR=yes
```

#### is equivalent to

```
... SELECT_ERROR=true
```

or

```
... SELECT_ERROR=1
```

all of which mean that SELECT errors should be flagged at run time.

## 10.2.1 Environment Variables

You can use environment variables in SYS\_INCLUDE and INCLUDE precompiler options. Environment variables like <code>ORACLE\_HOME</code> can be used in SYS\_INCLUDE and INCLUDE directory paths while precompiling a PROC application. SYS\_INCLUDE and INCLUDE option

values can also come from the config file, pcscfg.cfg. The following usages of environment variables are supported.

#### In Linux

```
$ENV_VAR

sys_include=$ORACLE_HOME/precomp/public
include=$ORACLE_HOME/precomp/public

$(ENV_VAR)

sys_include=$(ORACLE_HOME)/precomp/public
include=$(ORACLE_HOME)/precomp/public

${ENV_VAR}

sys_include=${ORACLE_HOME}/precomp/public
include=${ORACLE_HOME}/precomp/public
include=${ORACLE_HOME}/precomp/public

In Windows

%ENV_VAR%

sys_include=%ORACLE_HOME%\precomp\public
include=%ORACLE_HOME%\precomp\public
```

# 10.2.2 Configuration Files

A configuration file is a text file that contains precompiler options. Each record (line) in the file contains only one option, with its associated value or values. *Any options entered on a line after the first option are ignored.* For example, the following configuration file contains the lines:

```
FIPS=YES
MODE=ANSI
CODE=ANSI C
```

to set defaults for the FIPS, MODE, and CODE options.

There is a restriction of 300 characters per line for each entry in <code>pcscfg.cfg</code>. To set a value longer than 300 characters, for example, the SYS\_INCLUDE path), create entries in multiple lines. For example,

```
sys_include=/ade/aime_rdbms_9819/oracle/precomp/public
sys_include=/usr/include,/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include
sys_include=/usr/lib/gcc-lib/i386-redhat-linux/3.2.3/include
sys_include=/usr/lib/gcc-lib/i386-redhat-linux7/2.96/include
sys_include=/usr/include
```

Do not use brackets at the end of a line. A bracket at the right hand end of a line nullifies all previous lines. For example, a bracket at the end of the third line,

```
sys_include=/ade/aime_rdbms_9819/oracle/precomp/public
sys_include=/usr/include,/usr/lib/gcc-lib/i486-suse-linux/2.95.3/include
sys_include=/usr/lib/gcc-lib/i386-redhat-linux/3.2.3/include)
sys_include=/usr/lib/gcc-lib/i386-redhat-linux7/2.96/include
sys_include=/usr/include
```

#### sets SYS\_INCLUDE to

/usr/lib/gcc-lib/i386-redhat-linux/3.2.3/include, /usr/lib/gcc-lib/i386-redhat-linux7/2.96/include, /usr/include

There is a single system configuration file for each installation. The name of the system configuration file is pesefg.cfg. The location of the file is system specific.

#### Note:

pcscfg.cfg contains default path settings for variables like include or LIBPATH. These paths may be computer or operating system dependent. You must verify that default paths are valid for your computer and operating system. If not valid, replace the path with 8dot3 notation.

In the pcscfg.cfg file, you cannot use spaces. For example, if the file contains the following line:

include="D:\Program Files\Microsoft Visual Studio\VC98\include"

the precompilation will fail. You can replace it with the following:

include=D:\Progra~1\Microa~4\VC98\include

Each Pro\*C/C++ user can have one or more private configuration files. The name of the configuration file must be specified using the CONFIG= precompiler option.



You cannot nest configuration files. This means that CONFIG= is not a valid option inside a configuration file.

#### **Related Topics**

About Using the Precompiler Options

## 10.2.3 Precedence of Option Values

The value of an option is determined, in increasing precedence, by:

- A value built in to the precompiler
- A value set in the Pro\*C/C++ system configuration file
- A value set in a Pro\*C/C++ user configuration file
- A value set in the command line
- A value set inline

For example, the option MAXOPENCURSORS specifies the maximum number of cached open cursors. The built-in precompiler default value for this option is 10. However, if MAXOPENCURSORS=32 is specified in the system configuration file, the default now becomes 32. The user configuration file could set it to yet another value, which then overrides the system configuration value. Finally, an inline specification takes precedence over all preceding defaults.

If a PROC command-line option is used a multiple number of times, the last value assigned in the last occurrence is the value used by PROC for precompilation. For example,

```
$ proc iname=sample.pc ... oname=output1.c ... oname=output2.c ... oname=output3.c
```

In the example, output3.c is the ONAME value used by PROC and the generated output filename is output3.c.

If an option is specified both inside the config file (system default or user specified) and on the command line, then the value specified in the command line takes precedence.

In the case of SYS\_INCLUDE and INCLUDE options, the behavior is as defined in Environment Variables. The values are appended unless interrupted by a bracket at the end of a line.

If you specify a private configuration file with CONFIG=*filename*, then the first value takes precedence and subsequent occurences in the command line are ignored. This is an exception to command-line last-value precedence.

Some options, such as USERID, do not have a precompiler default value. The built-in default values for options that do have them are listed in Table 10-2.



Check your system-specific documentation for the precompiler default values; they may have been changed from the values in this chapter for your platform.

#### **Determining Current Values**

You can interactively determine the current value for one or more options by using a question mark on the command line. For example, if you issue the command

```
proc ?
```

the complete set of options, along with their current values, is printed to your terminal. (On a UNIX system running the C shell, escape the '?' with a backslash.) In this case, the values are those built into the precompiler, overridden by any values in the system configuration file. But if you issue the command

```
proc config=my_config_file.h ?
```

and there is a file named my\_config\_file.h in the current directory, all options are listed. Values in the user configuration file supply missing values, and supersede values built-in to the Pro\*C/C++ precompiler, or values specified in the system configuration file.

You can also determine the current value of a single option, by simply specifying that option name, followed by =?. For example:

```
proc maxopencursors=?
```

prints the current default value for the MAXOPENCURSORS option.

#### Entering:

proc

will give a short summary that resembles "Table 10-2".

#### **Related Topics**

- What Occurs During Precompilation?
- About Using the Precompiler Options



# 10.2.4 Macro and Micro Options

The option MODE controls several options at once. MODE is known as a *macro* option. Some newer options such as CLOSE\_ON\_COMMIT, DYNAMIC and TYPE\_CODE control only one function and are known as *micro* options. A macro option has precedence over micro options only if the macro option is at a higher level of precedence.

The following table lists the values of micro options set by the macro option values:

Table 10-1 How Macro Option Values Set Micro Option Values

Macro Option	Micro Option
MODE=ANSI   ISO	CLOSE_ON_COMMIT=YES
	DYNAMIC=ANSI
	TYPE_CODE=ANSI
MODE=ORACLE	CLOSE_ON_COMMIT=NO
	DYNAMIC=ORACLE
	TYPE_CODE=ORACLE

If you specify both MODE=ANSI and CLOSE\_ON\_COMMIT=NO in the user configuration file, then cursors will not be closed after a COMMIT. If you specify MODE=ORACLE in your configuration file and CLOSE\_ON\_COMMIT=YES on the command line, then the cursors will be closed.

#### **Related Topics**

Precedence of Option Values

# 10.2.5 What Occurs During Precompilation?

During precompilation, Pro\*C/C++ generates C or C++ code that replaces the SQL statements embedded in your host program. The generated code contains data structures that indicate the datatype, length, and address of host variables, as well as other information required by the runtime library, SQLLIB. The generated code also contains the calls to SQLLIB routines that perform the embedded SQL operations.



The precompiler does not generate calls to Oracle Call Interface (OCI) routines.

Table 10-2 is a quick reference to the major precompiler options. The options that are accepted, but do not have any affect, are not included in this table.

#### **Related Topics**

- Oracle Database Error Messages Reference
- About Using the Precompiler Options



# 10.2.6 Scope of Options

A precompilation unit is a file containing C code and one or more embedded SQL statements. The options specified for a given precompilation unit affect only that unit; they have no effect on other units. For example, if you specify HOLD\_CURSOR=YES and RELEASE\_CURSOR=YES for unit A, but not for unit B, SQL statements in unit A run with these HOLD\_CURSOR and RELEASE\_CURSOR values, but SQL statements in unit B run with the default values.

# 10.2.7 Pro\*C/C++ Precompiler Issues for Windows Platforms

This section highlights issues related to Pro\*C/C++ for Windows platforms.

# 10.2.7.1 Configuration File

For this release, the system configuration file is called pcscfg.cfg. This file is located in the ORACLE HOME\precomp\admin directory.

## 10.2.7.2 CODE

The CODE option has a default setting of ANSI\_C. Pro\*C/C++ for other operating systems may have a default setting of KR C.

### 10.2.7.3 DBMS

DBMS=V6 CHAR is not supported when using CHAR MAP=VARCHAR2. Instead, use DBMS=V7.

### 10.2.7.4 INCLUDE

For sample programs that precompile with PARSE=PARTIAL or PARSE=FULL, an include path of c:\program files\devstudio\vc\include has been added. If Microsoft Visual Studio has been installed in a different location, modify the Include Directories field accordingly for the sample programs to precompile correctly.

## 10.2.7.5 PARSE

The PARSE option has a default setting of NONE. Pro\*C/C++ for other operating systems may have a default setting of FULL.

# 10.3 Quick Reference

Table 10-2 is a quick reference to the Pro\*C/C++ options. Options marked with an asterisk can be entered inline.

**Table 10-2** Precompiler Options

Syntax	Default	Specifics
AUTO_CONNECT={YES   NO}	NO	Automatic CLUSTER\$ account connection before the first executable statement.



Table 10-2 (Cont.) Precompiler Options

Syntax	Default	Specifics
CHAR_MAP={VARCHAR2   CHARZ   STRING   CHARF} *	CHARZ	Mapping of character arrays and strings.
CINCR	1	Allows the application to set the next increment for physical connections to be opened to the database, if the current number of physical connections is less than CMAX.
CLOSE_ON_COMMIT={YES   NO}	NO	Close all cursors on COMMIT.
CODE={ANSI_C   KR_C   CPP}	KR_C	Kind of C code generated.
COMP_CHARSET={MULTI_BYTE   SINGLE_BYTE}	MULTI_BYTE	The character set type the C/C++ compiler supports.
CONFIG=filename	none	User's private configuration file.
CMIN	2	Specifies the minimum number of physical connections in the connection pool.
CMAX	100	Specifies the maximum number of physical connections that can be opened for the database.
CNOWAIT	0 which means not set.	This attribute determines if the application must repeatedly try for a physical connection when all other physical connections in the pool are busy, and the total number of physical connections has already reached its maximum.
CPOOL	NO	Based on this option, the precompiler generates the appropriate code that directs SQLLIB to enable or disable the connection pool feature.
CPP_SUFFIX=extension	none	Specify the default filename extension for C+ + output files.
CTIMEOUT	0 which means not set.	Physical connections that are idle for more than the specified time (in seconds) are terminated to maintain an optimum number of open physical connections.
DBMS={V7   NATIVE   V8}	NATIVE	Compatibility (Oracle7, Oracle8, Oracle8 <i>i</i> , Oracle9 <i>i</i> , or the database version to which you are connected at precompile time).
DEF_SQLCODE={YES   NO}	NO	Generate a macro to #define SQLCODE.
DEFINE=name *	none	Define a name for use by the Pro*C/C++ precompiler.
DURATION={TRANSACTION   SESSION}	TRANSACTION	Set pin duration for objects in the cache.
DYNAMIC={ANSI   ORACLE}	ORACLE	Specifies Oracle or ANSI SQL semantics.
ERRORS={YES   NO}	YES	Where to direct error messages (NO means only to listing file, and not to terminal).
ERRTYPE=filename	none	Name of the listing file for intype file error messages.
FIPS={NO   SQL89   SQL2   YES} *	none	Whether to flag ANSI/ISO non-compliance.



Table 10-2 (Cont.) Precompiler Options

Syntax	Default	Specifics
HEADER=extension	none	Specify file extension for precompiled header files.
HOLD_CURSOR={YES   NO} *	NO	How cursor cache handles SQL statement.
INAME=]filename	none	Name of the input file.
INCLUDE=pathname *	none	Directory path for EXEC SQL INCLUDE or #include statements.
INTYPE=filename	none	Name of the input file for type information.
LINES={YES   NO}	NO	Whether #line directives are generated.
LNAME=filename	none	Name of listing file.
LTYPE={NONE   SHORT   LONG}	none	Type of listing file to be generated, if any.
MAXLITERAL=101024	1024	Maximum length (bytes) of string literals in generated C code.
MAXOPENCURSORS=5255 *	10	Maximum number of concurrent cached open cursors.
MODE={ANSI   ISO   ORACLE}	ORACLE	ANSI/ISO or Oracle behavior.
NATIVE_TYPES	NO	Support for native float/double.
NLS_CHAR=(var1,, varn)	none	Specify multibyte character variables.
NLS_LOCAL={YES   NO}	NO	Control multibyte character semantics.
OBJECTS={YES   NO}	YES	Support of object types.
ONAME=]filename	iname.c	Name of the output (code) file.
ORACA={YES   NO} *	NO	Whether to use the ORACA.
PAGELEN=30256	80	Page length of the listing file.
PARSE={NONE   PARTIAL   FULL}	FULL	Whether Pro*C/C++ parses (with a C parser) the.pc source.
PLAN_BASELINE={module_name  YES  NO}	NO	Specify a module name to create a SQL Plan Baseline.
PLAN_PREFIX={prefix_name   none}	none	Ensures that the plan name does not exceed 128 bytes.
PLAN_RUN={YES   NO}	NO	Executes the generated SQL file
PLAN_FIXED={YES   NO}	YES	Specifies whether or not the created plan baseline is fixed or non-fixed
PLAN_ENABLED={YES   NO}	YES	Enables the Plan Baseline that is created.
MEMFORPREFETCH=04294967294	none	Speed up queries by pre-fetching rows that fill the specified memory.
PREFETCH=065535	1	Speed up queries by pre-fetching a given number of rows.
RELEASE_CURSOR={YES   NO} *	NO	Control release of cursors from cursor cache.
SELECT_ERROR={YES   NO} *	YES	Flagging of SELECT errors.
SQLCHECK={SEMANTICS   SYNTAX} *	SYNTAX	Amount of precompile time SQL checking.
SYS_INCLUDE=pathname	none	Directory where system header files, such as iostream.h, are found.
THREADS={YES   NO}	NO	Indicates a shared server application.

Table 10-2 (Cont.) Precompiler Options

Syntax	Default	Specifics
TYPE_CODE={ORACLE   ANSI}	ORACLE	Use of Oracle or ANSI type codes for dynamic SQL.
UNSAFE_NULL={YES   NO}	NO	UNSAFE_NULL=YES disables the ORA-01405 message.
USERID=username/password[@dbname]	none	Username/password[@dbname] connect string.
UTF16_CHARSET={NCHAR_CHARSET   DB_CHARSET}	NCHAR_CHARSET	Specify the character set form used by UNICODE(UTF16).
VARCHAR={YES   NO}	NO	Allow the use of implicit VARCHAR structures.
VERSION={ANY   LATEST   RECENT} *	RECENT	Which version of an object is to be returned.

# **10.4 Entering Options**

You can enter any precompiler option in the command line. Many can also be entered inline in the precompiler program source file, using the EXEC ORACLE OPTION statement.

## 10.4.1 On the Command Line

You enter precompiler options in the command line using the following syntax:

```
... [OPTION NAME=value] [OPTION NAME=value] ...
```

Separate each option=value specification with one or more spaces. For example, you might enter the following:

```
... CODE=ANSI C MODE=ANSI
```

## 10.4.2 Inline

You enter options inline by coding EXEC ORACLE statements, using the following syntax:

```
EXEC ORACLE OPTION (OPTION NAME=value);
```

For example, you might code the following:

EXEC ORACLE OPTION (RELEASE CURSOR=yes);

## 10.4.2.1 Uses for EXEC ORACLE

The EXEC ORACLE feature is especially useful for changing option values during precompilation. For example, you might want to change HOLD\_CURSOR and RELEASE\_CURSOR on a statement-by-statement basis.

Specifying options inline or in a configuration file is also helpful if your operating system limits the number of characters you can enter on the command line.



Performance Tuning shows you how to optimize runtime performance using inline options.

## 10.4.2.2 Scope of EXEC ORACLE

An EXEC ORACLE statement stays in effect until textually superseded by another EXEC ORACLE statement specifying the same option. In the following example, HOLD\_CURSOR=NO stays in effect until superseded by HOLD\_CURSOR=YES:

```
char emp name[20];
int emp_number, dept_number;
float salary;
EXEC SQL WHENEVER NOT FOUND DO break;
EXEC ORACLE OPTION (HOLD CURSOR=NO);
EXEC SQL DECLARE emp cursor CURSOR FOR
SELECT empno, deptno FROM emp;
EXEC SQL OPEN emp_cursor;
printf(
"Employee Number Department\n-----\n");
for (;;)
  EXEC SQL FETCH emp_cursor INTO :emp_number, :dept_number;
  printf("%d\t%d\n", emp_number, dept_number);
EXEC SQL WHENEVER NOT FOUND CONTINUE;
for (;;)
  printf("Employee number: ");
  scanf("%d", &emp_number);
  if (emp number == 0)
     break;
  EXEC ORACLE OPTION (HOLD CURSOR=YES);
  EXEC SQL SELECT ename, sal
     INTO :emp name, :salary
     FROM emp WHERE empno = :emp number;
  printf("Salary for %s is %6.2f.\n", emp name, salary);
```

# 10.4.3 Column Properties Support

Column properties are returned in an 8-byte value, where each bit indicates a column property. Three column properties are supported:



You can get the column properties through dynamic statements using the new SQLDA member (sqlda->CP[]).

This member is updated as part of a metadata DESCRIBE.

You can get the column properties through static statements using the function SQLGetColProp(), which gets the column properties from the last executed statement.

SQLGetColProp() returns values determined by the column attribute colatr.

- SQL\_ATTR\_COL\_PROPERTIES: Returns an 8 byte value (colprop) containing the column properties of the named column
- SQL\_ATTR\_COL\_PROPERTY\_IS\_IDENTITY colprop is true if the named column is an identity column
- SQL\_ATTR\_COL\_PROPERTY\_IS\_GEN\_ALWAYS colprop is true if the named column always generates an auto increment value
- SQL\_ATTR\_COL\_PROPERTY\_IS\_GEN\_BY\_DEF\_ON\_NULL colprop is true if the named column generates an auto increment value in the case of a default-null column constraint.
- SQL\_ATTR\_COL\_PROPERTY\_HAS\_DOMAIN returns colprop is true if the named column has a DOMAIN constraint.

# 10.5 About Data Use Case Domain Metadata Support

A data use case domain (also called use case domain or domain) is a database object that maintains properties and annotations associated with an existing Oracle Database data type. Using domains, application developers can indicate intended usage of a column of a given type. Oracle Database enables applications to create a domain and associate it with columns.

## See Also:

- Data Use Case Domains in Oracle Database Concepts
- Data Use Case Domains in Oracle Database Development Guide

Precompilers can use predefined functions to obtain domain metadata or annotation information from columns associated with a domain. Applications with references to the DOMAIN syntax must be precompiled using the common front-end parser, for example:

```
proc common parser=yes userid=scott/tiger sample10.p
```

The following API function gets the domain name associated with a column (in the domainName argument), and returns the length of the domain (in the domainLen argument).

```
void SQLGetDomainName(
  void *uga,
  text *coln,
  text *domainName
  ub4 *domainLen);
```

#### Parameters:

- uga (IN) runtime context
- coln (IN) column name
- domainName (OUT) domain name
- domainLen (OUT) length of the domain name

#### The following is an example:

```
EXEC SQL SELECT empId, empEmail FROM empTable;
strcpy("empEMail", colName);
cp = SQL_ATTR_COL_PROPERTY_HAS_DOMAIN;
SQLGetColProp(NULL, colName, &cp, colProp);
SQLColumnHasDomain(NULL, &colProp, &hasDomain);

if (hasDomain)
{
    SQLGetDomainName(NULL, colName, domainName, &domainLen);
    printf("column %s has domain name:%.*s\n", colName, domainLen, domainName);
}

printf("domainName:%.*s\n", domainLen, domainName);
printf("domainLen:%d\n", domainLen);
```

## Note:

The CAST AS DOMAIN feature is not supported. For example:

```
EXEC SQL SELECT First_Name, CAST (Score AS domain domain_1) Int_Score
into :nfname, :nscore FROM Student_Score;
```

#### returns the following error:

ORA-11537 CAST AS DOMAIN feature is not supported.

# 10.6 About Using the Precompiler Options

This section is organized for easy reference. It lists precompiler options alphabetically, and for each option gives its purpose, syntax, and default value. Usage notes help you understand how the option works.

# 10.6.1 AUTO\_CONNECT

#### **Purpose**

Allows automatic connection to the CLUSTER\$ account.

#### **Syntax**

AUTO CONNECT={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Can be entered only on the command line or in a configuration file.

If AUTO\_CONNECT=YES, and the application is not already connected to a database when it processes the first executable SQL statement, it attempts to connect using the userid

CLUSTER\$username

where *username* is your current operating system user or task name and CLUSTER\$username is a valid Oracle userid.

When AUTO\_CONNECT=NO, you must use the CONNECT statement in your program to connect to Oracle.

# 10.6.2 CHAR\_MAP

#### **Purpose**

Specifies the default mapping of C host variables of type char or char[n], and pointers to them, into SQL.

#### **Syntax**

CHAR MAP={VARCHAR2 | CHARZ | STRING | CHARF}

#### **Default**

CHARZ

#### **Usage Note**

Before release 8.0, you had to declare char or char[n] host variables as CHAR, using the SQL DECLARE statement. The external datatypes VARCHAR2 and CHARZ were the default character mappings of Oracle7.



### See Also:

- "VARCHAR Variables" for a table of CHAR\_MAP settings, descriptions of the datatype, and where they are the default.
- "Inline Usage of the CHAR\_MAP Option" for an example of usage of CHAR\_MAP in Pro\*C/C++.

## 10.6.3 CINCR

#### **Purpose**

Allows the application to set the next increment for physical connections to be opened to the database.

#### **Syntax**

CINCR = Range is 1 to (CMAX-CMIN).

#### **Default**

1

#### **Usage Notes**

Initially, all physical connections as specified through CMIN are opened to the server. Subsequently, physical connections are opened only when necessary. Users should set CMIN to the total number of planned or expected concurrent statements to be run by the application to get optimum performance. The default value is set to 2.

# 10.6.4 CLOSE\_ON\_COMMIT

#### **Purpose**

Specifies whether or not all cursors declared without the WITH HOLD clause are closed on commit.

#### **Syntax**

CLOSE\_ON\_COMMIT={YES | NO}

#### Default

NO

#### **Usage Notes**

Can be used only on the command line or in a configuration file.

This option will only have an effect when a cursor is not coded using the WITH HOLD clause in a DECLARE CURSOR statement, since that will override both the new option and the existing behavior which is associated with the MODE option. If MODE is specified at a higher level than CLOSE\_ON\_COMMIT, then MODE takes precedence. For example, the defaults are MODE=ORACLE and CLOSE\_ON\_COMMIT=NO. If the user specifies MODE=ANSI on the command line, then any cursors not using the WITH HOLD clause will be closed on commit.

When CLOSE\_ON\_COMMIT=NO (when MODE=ORACLE), issuing a COMMIT or ROLLBACK will close only cursors that are declared using the FOR UPDATE clause or are referenced in a CURRENT OF clause. Other cursors that are not affected by the COMMIT or ROLLBACK statement, remain open, if they are open already. However, when CLOSE\_ON\_COMMIT=YES (when MODE=ANSI), issuing a COMMIT or ROLLBACK closes all cursors.

#### **Related Topics**

- Scrollable Cursors
- Macro and Micro Options

### 10.6.5 CMAX

#### **Purpose**

Specifies the maximum number of physical connections that can be opened for the database.

#### **Syntax**

CINCR = Range is 1 to 65535

#### Default

100

#### **Usage Notes**

CMAX value must be at least CMIN+CINCR.Once this value is reached, more physical connections cannot be opened.In a typical application, running 100 concurrent database operations is more than sufficient. The user can set an appropriate value.

## 10.6.6 CMIN

#### **Purpose**

Specifies the minimum number of physical connections that can be opened for the database.

#### **Syntax**

CINCR = Range is 1 to (CMAX-CINCR).

#### Default

2

#### **Usage Notes**

CMAX value must be at least CMIN+CINCR.Once this value is reached, more physical connections cannot be opened.In a typical application, running 100 concurrent database operations is more than sufficient. The user can set an appropriate value.



## **10.6.7 CNOWAIT**

#### **Purpose**

This attribute determines if the application must repeatedly try for a physical connection when all other physical connections in the pool are busy, and the total number of physical connections has already reached its maximum.

#### **Syntax**

CNOWAIT = Range is 1 to 65535.

#### Default

0 which means not set.

#### **Usage Notes**

If physical connections are not available and no more physical connections can be opened, an error is thrown when this attribute is set. Otherwise, the call waits until it acquires another connection. By default, CNOWAIT is not to be set so a thread will wait until it can acquire a free connection, instead of returning an error.

## 10.6.8 CODE

#### **Purpose**

Specifies the format of C function prototypes generated by the Pro\*C/C++ precompiler. (A *function prototype* declares a function and the datatypes of its arguments.) The precompiler generates function prototypes for SQL library routines, so that your C compiler can resolve external references. The CODE option lets you control the prototyping.

#### **Syntax**

CODE={ANSI\_C | KR\_C | CPP}

#### **Default**

KR\_C

#### **Usage Notes**

Can be entered on the command line, but not inline.

ANSI C standard X3.159-1989 provides for function prototyping. When CODE=ANSI\_C, Pro\*C/C++ generates full function prototypes, which conform to the ANSI C standard. An example follows:

```
extern void sqlora(long *, void *);
```

The precompiler can also generate other ANSI-approved constructs such as the **const** type qualifier.

When CODE=KR\_C (the default), the precompiler comments out the argument lists of generated function prototypes, as shown here:

```
extern void sqlora(/* long *, void * */);
```



Specify CODE=KR\_C if your C compiler is not compliant with the X3.159 standard.

When CODE=CPP, the precompiler generates C++ compatible code.



"Code Generation" for all of the consequences of using this option value.

# 10.6.9 COMMON PARSER

#### **Purpose**

Specifies that the SQL99 syntax for SELECT, INSERT, DELETE, UPDATE and body of the cursor in a DECLARE CURSOR statement will be supported.

#### **Syntax**

COMMON PARSER={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Can be entered in the command line.

# 10.6.10 COMP\_CHARSET

#### **Purpose**

Indicates to the Pro\*C/C++ Precompiler whether multibyte character sets are (or are not) supported by the compiler to be used. It is intended for use by developers working in a multibyte client-side environment (for example, when NLS\_LANG is set to a multibyte character set).

#### **Syntax**

COMP\_CHARSET={MULTI\_BYTE | SINGLE\_BYTE}

#### **Default**

MULTI\_BYTE

#### **Usage Notes**

Can be entered only on the command line.

With COMP\_CHARSET=MULTI\_BYTE (default), Pro\*C/C++ generates C code that is to be compiled by a compiler that supports multibyte character sets.

With COMP\_CHARSET=SINGLE\_BYTE, Pro\*C/C++ generates C code for single-byte compilers that addresses a complication that *may* arise from the ASCII equivalent of a backslash (\) character in the second byte of a double-byte character in a multibyte string. In this case, the backslash (\) character is "escaped" with another backslash character preceding it





The need for this feature is common when developing in a Shift-JIS environment with older C compilers.

This option has no effect when NLS\_LANG is set to a single-byte character set.

## 10.6.11 CONFIG

#### **Purpose**

Specifies the name of a user configuration file.

#### **Syntax**

CONFIG=filename

#### Default

None

#### **Usage Notes**

Can be entered only on the command line.

This option is the only way you can inform Pro\*C/C++ of the name and location of user configuration files.

## 10.6.12 CPOOL

#### **Purpose**

Based on this option, the precompiler generates the appropriate code that directs SQLLIB to enable or disable the connection pool feature.

#### **Syntax**

CPOOL = {YES|NO}

#### **Default**

NO

#### **Usage Notes**

If this option is set to NO, other connection pooling options will be ignored by the precompiler.

# 10.6.13 CPP\_SUFFIX

#### **Purpose**

The CPP\_SUFFIX option provides the ability to specify the filename extension that the precompiler appends to the C++ output file generated when the CODE=CPP option is specified.



#### **Syntax**

CPP SUFFIX=filename\_extension

#### **Default**

System-specific.

#### **Usage Notes**

Most C compilers expect a default extension of ".c" for their input files. Different C++ compilers, however, can expect different filename extensions. The CPP\_SUFFIX option provides the ability to specify the filename extension that the precompiler generates. The value of this option is a string, without the quotes or the period. For example, CPP\_SUFFIX=cc, or CPP\_SUFFIX=C.

## **10.6.14 CTIMEOUT**

#### **Purpose**

Physical connections that are idle for more than the specified time (in seconds) are terminated to maintain an optimum number of open physical connections

#### **Syntax**

CTIMEOUT = Range is 1 to 65535.

#### **Default**

0 which means not set.

#### **Usage Notes**

Physical connections will not be closed until the connection pool is terminated. Creating a new physical connection will cost a round trip to the server.

# 10.6.15 DB2\_ARRAY

#### **Purpose**

Based on this option, the precompiler activates the additional array insert/select syntax.

#### **Syntax**

DB2\_ARRAY={YES |NO}

#### Default

NO

#### **Usage Notes**

If this option is set to NO, the Oracle precompiler syntax is supported, otherwise the DB2 insert/select array syntax is supported.



## 10.6.16 DBMS

#### **Purpose**

Specifies whether Oracle follows the semantic and syntactic rules of Oracle9*i*, Oracle8*i*, Oracle8, Oracle7, or the native version of Oracle (that is, the version to which the application is connected).

#### **Syntax**

DBMS={NATIVE | V7 | V8}

#### Default

**NATIVE** 

#### **Usage Notes**

Can be entered only on the command line, or in a configuration file.

The DBMS option lets you control the version-specific behavior of Oracle. When DBMS=NATIVE (the default), Oracle follows the semantic and syntactic rules of the database version to which the application is connected.

When DBMS=V8, or DBMS=V7, Oracle follows the respective rules for Oracle9*i* (which remain the same as for Oracle7, Oracle8, and Oracle8*i*).

V6\_CHAR is not supported in Oracle and its functionality is provided by the precompiler option CHAR\_MAP.

Table 10-3 DBMS and MODE Interaction

Situation	DBMS=V7   V8 MODE=ANSI	DBMS=V7   V8 MODE=ORACLE
"no data found" warning code	+100	+1403
fetch NULLs without using indicator variables	error -1405	error -1405
fetch truncated values without using indicator variables	no error but sqlwarn[1] is set	no error but sqlwarn[1]) is set
cursors closed by COMMIT or ROLLBACK	all explicit	CURRENT OF only
open an already OPENed cursor	error -2117	no error
close an already CLOSEd cursor	error -2114	no error
SQL group function ignores NULLs	no warning	no warning
when SQL group function in multirow query is called	FETCH time	FETCH time
declare SQLCA structure	optional	required
declare SQLCODE or SQLSTATE status variable	required	optional, but Oracle ignores
integrity constraints	enabled	enabled
PCTINCREASE for rollback segments	not allowed	not allowed
MAXEXTENTS storage parameters	not allowed	not allowed

#### **Related Topics**

CHAR\_MAP



# 10.6.17 DEF\_SQLCODE

#### **Purpose**

Controls whether the Pro\*C/C++ precompiler generates #define's for SQLCODE.

#### **Syntax**

DEF SQLCODE={NO | YES}

#### Default

NO

### **Usage Notes**

Can be used only on the command line or in a configuration file.

When DEF\_SQLCODE=YES, the precompiler defines SQLCODE in the generated source code as follows:

```
#define SQLCODE sqlca.sqlcode
```

You can then use SQLCODE to check the results of executable SQL statement. The DEF\_SQLCODE option is supplied for compliance with standards that require the use of SQLCODE.

In addition, you must also include the SQLCA using one of the following entries in your source code:

```
#include <sqlca.h>
```

or

EXEC SQL INCLUDE SQLCA;

If the SQLCA is not included, using this option causes a precompile time error.

## 10.6.18 DEFINE

#### **Purpose**

Defines a name that can be used in **#ifdef** and **#ifndef** Pro\*C/C++ precompiler directives. The defined name can also be used by the EXEC ORACLE IFDEF and EXEC ORACLE IFNDEF statements.

#### **Syntax**

DEFINE=name

#### Default

None

#### **Usage Notes**

Can be entered on the command line or inline. You can only use DEFINE to define a name—you cannot define macros with it. For example, the following use of define is not valid:

```
proc my prog DEFINE=LEN=20
```

#### Using DEFINE in the correct way, you could do

```
proc my_prog DEFINE=XYZZY
```

#### And then in my\_prog.pc, code

```
#ifdef XYZZY
...
#else
...
#endif
```

#### Or, you could just as well code

```
EXEC ORACLE IFDEF XYZZY;
...
EXEC ORACLE ELSE;
...
EXEC ORACLE ENDIF;
```

#### The following example is invalid:

```
#define XYZZY
...
EXEC ORACLE IFDEF XYZZY
...
EXEC ORACLE ENDIF;
```

EXEC ORACLE conditional statements are *valid* only if the macro is defined using EXEC ORACLE DEFINE or the DEFINE option.

If you define a name using DEFINE=, and then conditionally include (or exclude) a code section using the Pro\*C/C++ precompiler #ifdef (or #ifndef) directives, you must also make sure that the name is defined when you run the C compiler. For example, for UNIX cc, you must use the -D option to define the name.

## **10.6.19 DURATION**

#### **Purpose**

Sets the pin duration used by subsequent EXEC SQL OBJECT CREATE and EXEC SQL OBJECT DEREF statements. Objects in the cache are implicitly unpinned at the end of the duration.

#### **Syntax**

DURATION={TRANSACTION | SESSION}

#### **Default**

**TRANSACTION** 

#### **Usage Notes**

Can be entered inline by use of the EXEC ORACLE OPTION statement.

TRANSACTION means that objects are implicitly unpinned when the transaction completes.

SESSION means that objects are implicitly unpinned when the connection is terminated.

## **10.6.20 DYNAMIC**

#### **Purpose**

This micro option specifies the descriptor behavior in dynamic SQL Method 4. The setting of MODE determines the setting of DYNAMIC.

#### **Syntax**

DYNAMIC={ORACLE | ANSI}

**Default** 

**ORACLE** 

#### **Usage Notes**

Cannot be entered inline by use of the EXEC ORACLE OPTION statement.

See the DYNAMIC option settings in ANSI Dynamic SQL Precompiler Options.

## 10.6.21 ERRORS

#### **Purpose**

Specifies whether error messages are sent to the terminal as well as the listing file (YES), or just to the listing file (NO).

#### **Syntax**

ERRORS={YES | NO}

**Default** 

YES

#### **Usage Notes**

Can be entered only on the command line, or in a configuration file.

## 10.6.22 ERRTYPE

#### **Purpose**

Specifies an output file in which errors generated in processing type files are written. If omitted, errors are output to the screen.

#### **Syntax**

ERRTYPE=filename

**Default** 

None



#### **Usage Notes**

Only one error file will be produced. If multiple values are entered, the last one is used by the precompiler.

#### **Related Topics**

INTYPE

### 10.6.23 EVENTS

#### **Purpose**

Specifies that the application is interested in registering for and receiving notifications.

#### **Syntax**

EVENTS={YES | NO}

#### Default

NO

#### **Usage Notes**

Can only be entered in the command line.

### 10.6.24 FIPS

#### **Purpose**

Specifies whether extensions to ANSI SQL are flagged (by the FIPS Flagger). An extension is any SQL element that violates ANSI format or syntax rules, except privilege enforcement rules.

#### **Syntax**

FIPS={SQL89 | SQL2 | YES | NO}

#### **Default**

None

#### **Usage Notes**

Can be entered inline or on the command line.

When FIPS=YES, the FIPS Flagger is enabled, and warning (not error) messages are issued if you use an Oracle extension to ANSI SQL, or use an ANSI SQL feature in a nonconforming manner. Extensions to ANSI SQL that are flagged at precompile time include the following:

- Array interface including the FOR clause
- SQLCA, ORACA, and SQLDA data structures
- Dynamic SQL including the DESCRIBE statement
- Embedded PL/SQL blocks
- Automatic datatype conversion



- DATE, NUMBER, RAW, LONGRAW, VARRAW, ROWID, VARCHAR2, and VARCHAR datatypes
- Pointer host variables
- Oracle OPTION statement for specifying runtime options
- TOOLS statements in user exits
- CONNECT statement
- TYPE and VAR datatype equivalence statements
- AT db name clause
- DECLARE...DATABASE, ...STATEMENT, and ...TABLE statements
- SQLWARNING condition in WHENEVER statement
- DO function\_name() and "do break" and "do continue" actions in WHENEVER statement
- COMMENT and FORCE TRANSACTION clauses in COMMIT statement
- FORCE TRANSACTION and TO SAVEPOINT clauses in ROLLBACK statement
- RELEASE parameter in COMMIT and ROLLBACK statements
- Optional colon-prefixing of WHENEVER...GOTO labels, and of host variables in the INTO clause

### 10.6.25 HEADER

#### **Purpose**

Permits precompiled header files. Specifies the file extension for precompiled header files.

#### **Syntax**

**HEADER**=extension

#### **Default**

NONE

#### **Usage Notes**

When precompiling a header file, this option is required and is used to specify the file extension for the output file that is created by precompiling that header file.

When precompiling an ordinary Pro\*C/C++ program this option is optional. When given, it enables the use of the precompiled header mechanism during the precompilation of that Pro\*C/C++ program.

In both cases, this option also specifies the file extension to use when processing a #include directive. If an #include file exists with the specified extension, Pro\*C/C++ assumes the file is a precompiled header file previously generated by Pro\*C/C++. Pro\*C/C++ will then instantiate the data from that file rather than process the #include directive and precompile the included header file.

This option is only allowed on the command line or in a configuration file. It is not allowed inline. When using this option, specify the file extension only. Do not include any file separators. For example, do not include a period '.' in the extension.



#### **Related Topics**

Precompiled Header Files

# 10.6.26 HOLD\_CURSOR

#### **Purpose**

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

#### **Syntax**

HOLD\_CURSOR={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Can be entered inline or on the command line.

You can use HOLD CURSOR to improve the performance of your program.

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. HOLD\_CURSOR controls what happens to the link between the cursor and cursor cache.

When HOLD\_CURSOR=NO, after Oracle executes the SQL statement and the cursor is closed, the precompiler marks the link as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks.

When HOLD\_CURSOR=YES, the link is maintained; the precompiler does not reuse it. This is useful for SQL statements that are often executed because it speeds up subsequent executions and there is no need to re-parse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set HOLD\_CURSOR before executing the SQL statement. For inline use with explicit cursors, set HOLD\_CURSOR before CLOSEing the cursor.

RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES and HOLD\_CURSOR=NO overrides RELEASE\_CURSOR=NO. For information showing how these two options interact, see Table B-1.

#### **Related Topics**

Performance Tuning

## 10.6.27 IMPLICIT SVPT

#### **Purpose**

Controls whether an implicit savepoint is taken prior to the start of a new batched insert.

#### **Syntax**

implicit\_svpt={YES|NO}



#### Default

NO

#### **Usage Notes**

If implict\_svpt=yes, a savepoint is taken prior to the start of a new batch of rows. If an error occurs on the insert, an implicit "rollback to savepoint" is executed. This option exists for DB2 compatibility, the obvious downside being the extra round-trip.

If implict\_svpt=no, there is no implicit savepoint taken. If an error occurs on the buffered insert, then it is reported back to the application, but no rollback is executed.

### 10.6.28 INAME

#### **Purpose**

Specifies the name of the input file.

#### **Syntax**

INAME=path\_and\_filename

#### Default

None

#### **Usage Notes**

Can be entered only on the command line.

All input file names must be unique at precompilation time.

You can omit the filename extension if it is .pc. If the input filename is the first option on the command line, you can omit the INAME= part of the option. For example:

```
proc sample1 MODE=ansi
```

to precompile the file sample1.pc, using ANSI mode. This command is the same as

proc INAME=sample1 MODE=ansi



The sqlctx hash value is generated based on the INAME parameter passed to the Pro\*C/C++ command. This can cause issues in the applications where files having the same name are stored in different directories containing different functions and the build scripts are sent to the physical directory to precompile the program. As a result, there is no need to place the makefiles at a higher level and precompile files using their pathnames.



### 10.6.29 INCLUDE

#### **Purpose**

Specifies a directory path for files included using the **#include** or EXEC SQL INCLUDE directives.

#### **Syntax**

INCLUDE=pathname or INCLUDE=(path\_1,path\_2,...,path\_n)

#### **Default**

Current directory and paths built into Pro\*C/C++.

#### **Usage Notes**

Can be entered inline or on the command line.

You use INCLUDE to specify a directory path for included files. The precompiler searches directories in the following order:

- 1. the current directory
- 2. the system directory specified in a SYS\_INCLUDE precompiler option
- 3. the directories specified by the INCLUDE option, in the order they are entered
- 4. the built-in directories for standard header files

You normally do not need to specify a directory path for Oracle-specific header files such as sqlca.h and sqlda.h.



If you specify an Oracle-specific filename without an extension for inclusion, Pro\*C/C++ assumes an extension of .h. So, included files should have an extension, even if it is not .h.

For all other header files, the precompiler does *not* assume a .h extension.

You must still use INCLUDE to specify directory paths for non-standard files, unless they are stored in the current directory. You can specify more than one path on the command line, as follows:

```
... INCLUDE=path 1 INCLUDE=path 2 ...
```



If the file you want to include resides in another directory, make sure that there is no file with the same name in the current directory.

The syntax for specifying a directory path using the INCLUDE option is system specific. Follow the conventions used for your operating system

### Note:

For the INCLUDE option, the precedence of option values gets reversed. Unlike other options in which the values get overwritten, INCLUDE appends all the directory files that are mentioned in:

- Precompiler
- Pro\*C/C++ system configuration file
- Pro\*C/C++ user configuration file
- Command line
- Inline

However, there is a difference between passing values within or without brackets. If you pass a single value or directory list within brackets, then the existing value of INCLUDE is overwritten. If you pass the list as a simple value without brackets, it will supplement any existing value.

### 10.6.30 INTYPE

#### **Purpose**

Specifies one or more OTT-generated type files (only needed if Object types are used in the application).

#### **Syntax**

INTYPE=(file 1,file 2,...,file n)

#### **Default**

None

#### **Usage Notes**

There will be one type file for each Object type in the Pro\*C/C++ code.

## 10.6.31 LINES

#### **Purpose**

Specifies whether the Pro\*C/C++ precompiler adds #line preprocessor directives to its output file.

#### **Syntax**

LINES={YES | NO}

#### Default

NO

#### **Usage Notes**

Can be entered only on the command line.

The LINES option helps with debugging.

When LINES=YES, the Pro\*C/C++ precompiler adds #line preprocessor directives to its output file.

Normally, your C compiler increments its line count after each input line is processed. The **#line** directives force the compiler to reset its input line counter so that lines of precompiler-generated code are not counted. Moreover, when the name of the input file changes, the next **#line** directive specifies the new filename.

The C compiler uses the line numbers and filenames to show the location of errors. Thus, error messages issued by the C compiler always refer to your original source files, not the modified (precompiled) source file. This also enables stepping through the original source code using most debuggers.

When LINES=NO (the default), the precompiler adds no #line directives to its output file.



The Pro\*C/C++ precompiler does not support the #line directive. This means that you cannot directly code #line directives in the precompiler source. But you can still use the LINES= option to have the precompiler insert #line directives for you.

#### **Related Topics**

Directives Ignored

## 10.6.32 LNAME

#### **Purpose**

Specifies the name of the listing file.

#### **Syntax**

LNAME=filename

#### Default

None

#### **Usage Notes**

Can be entered only on the command line.

The default filename extension for the listing file is .lis.

## 10.6.33 LTYPE

#### **Purpose**

Specifies the type of listing file generated.

#### **Syntax**

LTYPE={NONE | SHORT | LONG}

#### **Default**

**SHORT** 

#### **Usage Notes**

Can be entered on the command line or in a configuration file.

When a listing file is generated, the LONG format is the default. With LTYPE=LONG specified, all of the source code is listed as it is parsed and messages listed as they are generated. In addition, the Pro\*C/C++ options currently in effect are listed.

With LTYPE=SHORT specified, only the generated messages are listed—no source code—with line references to the source file to help you locate the code that generated the message condition.

With LTYPE=NONE specified, no list file is produced *unless* the LNAME option explicitly specifies a name for a list file. Under the latter condition, the list file *is* generated with LTYPE=LONG assumed.

## 10.6.34 MAX\_ROW\_INSERT

#### **Purpose**

Controls the number of rows that need to be buffered before executing the INSERT statement.

#### **Syntax**

max\_row\_insert={0...1000}

#### Default

0

#### **Usage Notes**

Any number greater than zero enables buffered insert feature and buffers that many rows before executing the INSERT statement.

### 10.6.35 MAXLITERAL

#### **Purpose**

Specifies the maximum length of string literals generated by the precompiler, so that compiler limits are not exceeded.

#### **Syntax**

MAXLITERAL=integer, range is 10 to 1024

#### **Default**

1024

#### **Usage Notes**

Cannot be entered inline.



The maximum value of MAXLITERAL is compiler dependent. For example, some C compilers cannot handle string literals longer than 512 characters, so you would specify MAXLITERAL=512.

Strings that exceed the length specified by MAXLITERAL are divided during precompilation, then recombined (concatenated) at run time.

You can enter MAXLITERAL inline but your program can set its value only once, and the EXEC ORACLE statement must precede the first EXEC SQL statement. Otherwise, Pro\*C/C++ issues a warning message, ignores the extra or misplaced EXEC ORACLE statement, and continues processing.

### 10.6.36 MAXOPENCURSORS

#### **Purpose**

Specifies the number of concurrently open cursors that the precompiler tries to keep cached.

#### **Syntax**

MAXOPENCURSORS=integer

#### **Default**

10

#### **Usage Notes**

Can be entered inline or on the command line.

You can use MAXOPENCURSORS to improve the performance of your program. When precompiling separately, use MAXOPENCURSORS. MAXOPENCURSORS specifies the *initial* size of the SQLLIB cursor cache.

When an implicit statement is executed and HOLD\_CURSOR=NO, or an explicit cursor is closed, the cursor entry is marked as reusable. If this statement is issued again and the cursor entry has not been used for another statement, it is reused.

If a new cursor is needed and the number of cursors allocated is less than MAXOPENCURSORS, then the next one in the cache is allocated. Once MAXOPENCURSORS has been exceeded Oracle first tries to reuse a previous entry. If there are no free entries, then an additional cache entry will be allocated. Oracle will continue to do this until the program runs out of memory or the database parameter OPEN\_CURSORS is exceeded.

During normal processing, when using HOLD\_CURSOR=NO and RELEASE\_CURSOR=NO (the default), it is advisable to set MAXOPENCURSORS to no more than 6 less than the database parameter OPEN\_CURSORS to allow for the cursors used by the data dictionary to process statements.

As your program's need for concurrently open cursors grows, you might want to respecify MAXOPENCURSORS to match the need. A value of 45 to 50 is not uncommon, but remember that each cursor requires another private SQL area in the user process memory space. The default value of 10 is adequate for most programs.

### **Related Topics**

- Performance Tuning
- Guidelines for Programming



### 10.6.37 MODE

#### **Purpose**

Specifies whether your program observes Oracle practices or complies with the current ANSI/ISO SQL standards.

#### **Syntax**

MODE={ANSI | ISO | ORACLE}

#### Default

**ORACLE** 

#### **Usage Notes**

Can be entered only on the command line or in a configuration file.

In the context of this option ISO is equivalent to ANSI.

When MODE=ORACLE (the default), your embedded SQL program observes Oracle practices. For example, a Declare Section is optional, and blanks are stripped.

When MODE=ANSI, your program complies *fully* with the ANSI SQL standard, and the following changes go into effect:

- Issuing a COMMIT or ROLLBACK closes all explicit cursors.
- You cannot OPEN an already open cursor or CLOSE an already closed cursor. (When MODE=ORACLE, you can reOPEN an open cursor to avoid re-parsing.)
- You must declare either an int variable named SQLCODE or a char SQLSTATE[6]
  variable (uppercase is required for both variables) that is in the scope of every EXEC SQL
  statement. The same SQLCODE or SQLSTATE variable need not be used in each case;
  that is, the variable need not be global.
- Declaring the SQLCA is optional. You need not include the SQLCA.
- The "no data found" Oracle warning code returned to SQLCODE becomes +100 instead of +1403. The message text does not change.
- You must have a Declare Section for host variables.

## 10.6.38 NATIVE TYPES

#### **Purpose**

Support for native float/double.

#### **Syntax**

NATIVE TYPES = {YES|NO}

#### **Default**

NO



#### **Usage Notes**

The native float and native double datatypes represent the single-precision and double-precision floating point values. They are represented natively, that is, in the host system's floating point format.

# 10.6.39 NLS\_CHAR

#### **Purpose**

Specifies which C host character variables are treated by the precompiler as multibyte character variables.

#### **Syntax**

NLS CHAR=varname or NLS CHAR=(var\_1,var\_2,...,var\_n)

#### **Default**

None.

#### **Usage Notes**

Can be entered only on the command line, or in a configuration file.

This option provides the ability to specify at precompile time a list of the names of one or more host variables that the precompiler must treat as multibyte character variables. You can specify only C *char* variables or Pro\*C/C++ VARCHARs using this option.

If you specify in the option list a variable that is not declared in your program, then the precompiler generates no error.

## 10.6.40 NLS\_LOCAL

#### **Purpose**

Determines whether multibyte character set conversions are performed by the precompiler runtime library, SQLLIB, or by the database server.

#### **Syntax**

NLS\_LOCAL={NO | YES}

#### **Default**

NO

#### **Usage Notes**

When set to YES, local multibyte support is provided by Pro\*C/C++ and the SQLLIB library. The option NLS\_CHAR must be used to indicate which C host variables are multibyte.

When set to NO, Pro\*C/C++ will use the database server support for multibyte objects. Set NLS LOCAL to NO for all new applications.

Environment variable NLS\_NCHAR must be set to a valid fixed-width National Character Set. Variable-width National Character Sets are not supported.

Can be entered only on the command line, or in a configuration file.

### 10.6.41 OBJECTS

#### **Purpose**

Requests support for object types.

#### **Syntax**

OBJECTS={YES | NO}

#### Default

YES

#### **Usage Notes**

Can only be entered in the command line.

### 10.6.42 ONAME

#### **Purpose**

Specifies the name of the output file. The output file is the C code file that the precompiler generates.

#### **Syntax**

ONAME=path and filename

#### **Default**

INAME with a .c extension.

#### **Usage Notes**

Can be entered only on the command line. Use this option to specify the full path name of the output file, where the path name differs from that of the input (.pc) file. For example, if you issue the command:

```
proc iname=my_test
```

the default output filename is  $my\_test.c.$  If you want the output filename to be  $my\_test\_1.c.$  issue the command

```
proc iname=my_test oname=my_test_1.c
```

You should add the .c extension to files specified using ONAME because one is not added by default.

### Note:

Oracle recommends that you not let the output filename default, but rather name it explicitly using ONAME. If you specify an ONAME value without an extension, the name of the generated file will not have one.

### 10.6.43 ORACA

#### **Purpose**

Specifies whether a program can use the Oracle Communications Area (ORACA).

#### **Syntax**

ORACA={YES | NO}

#### Default

NO

#### **Usage Notes**

Can be entered inline or on the command line.

When ORACA=YES, you must place either the EXEC SQL INCLUDE ORACA or #include oraca.h statement in your program.

### 10.6.44 OUTLINE

#### **Purpose**

Indicates that the outline SQL file needs to be generated for the SQL statements.

#### **Syntax**

outline={yes | no | category\_name}

#### Default

no

#### **Usage Notes**

The outline SQL file should be in the DEFAULT category if the value is yes and the generated outline format is

```
DEFAULT_<filename>_<filetype>_<sequence_no>
```

If the category name is mentioned, then the SQL file should be generated in the category mentioned. The generated outline format for this is

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

The outline SQL file is not generated if the value is no.

Semantic check should be full when this option is turned on, which means option sqlcheck=full/semantics. If sqlcheck=syntax/limited/none, then error will be generated.

## 10.6.45 OUTLNPREFIX

#### **Purpose**

Controls the generation of the outline names.

#### **Syntax**

outInprefix={none | prefix name}

#### **Default**

no

#### **Usage Notes**

If outInprefix=prefix\_name, then the outline format

```
<category_name>_<filename>_<filetype>
```

is replaced with <prefix name> for the outline names.

If the length of the outline name exceeds 128 bytes, then this option is helpful for the user who can just specify the prefix name.

If outInprefix=none, then the outline names are generated by the system. The generated format is

```
<category_name>_<filename>_<filetype>_<sequence_no>
```

Semantic check should be full when this option is turned on, which means option sqlcheck=full/semantics. If sqlcheck=syntax/limited/none, and/or outline=false, then error will be generated.

### 10.6.46 PAGELEN

#### **Purpose**

Specifies the number of lines for each physical page of the listing file.

#### **Syntax**

PAGELEN=integer

#### Default

80

#### **Usage Notes**

Cannot be entered inline. The value range allowed is 30..256..

## 10.6.47 PARSE

#### **Purpose**

Specifies the way that the Pro\*C/C++ precompiler parses the source file.

#### **Syntax**

PARSE={FULL | PARTIAL | NONE}

#### **Default**

**FULL** 



#### **Usage Notes**

To generate C++ compatible code, the PARSE option must be either NONE or PARTIAL.

If PARSE=NONE or PARSE=PARTIAL, all host variables must be declared inside a Declare Section.

The variable SQLCODE must also be declared inside a declare section, or it cannot be relied on to detect errors. Check the default value of PARSE for your platform.

If PARSE=FULL, the C parser is used, and it does not understand C++ constructs, such as classes, in your code.

With PARSE=FULL or PARSE=PARTIAL Pro\*C/C++ fully supports C preprocessor directives, such as **#define**, **#ifdef**, and so on. However, with PARSE=NONE conditional preprocessing is supported by EXEC ORACLE statements.



Some platforms have the default value of PARSE as other than FULL. See your system-dependent documentation.

#### **Related Topics**

- About Parsing Code
- Declare Section
- Conditional Precompilation

# 10.6.48 PLAN\_BASELINE

#### **Purpose**

Creates a SQL Plan Baseline by specifying the module name.

#### **Syntax**

PLAN BASELINE={module\_name | YES | NO}

#### Default

NO

#### **Usage Notes**

The module name will become a part of the unique plan name generated by appending the filename, file type, and sequence number.



### Note:

To generate and execute the plan baseline SQL statements, you must have the Execute privilege on the DBMS\_SPM\_INTERNAL package and the Administer SQL Management Object privilege.

The plan baseline options are valid only with SQLCHECK=FULL. If you specify these options without this, then the options are ignored and a warning is thrown informing that the option should be used with SQLCHECK=FULL only.

## 10.6.49 PLAN\_PREFIX

#### **Purpose**

Ensures that the plan name does not exceed 128 bytes.

#### **Syntax**

PLAN\_PREFIX={prefix\_name | none}

#### **Default**

none

#### **Usage Notes**

It is an optional command. The default is none which means that no prefix name is used, and if the plan name exceeds 128 bytes an error message is generated.

## 10.6.50 PLAN RUN

#### **Purpose**

Executes the generated SQL file.

#### **Syntax**

PLAN RUN={YES | NO}

#### **Default**

NO

#### **Usage Notes**

If the PLAN\_RUN option is not set then the generated SQL file is not executed.

## 10.6.51 PLAN\_FIXED

#### **Purpose**

Specifies whether or not the created plan baseline is fixed or non-fixed.



#### **Syntax**

PLAN\_FIXED={ YES | NO }

#### **Default**

YES

#### **Usage Notes**

When set to NO, a non-fixed Plan Baseline is created.

## 10.6.52 PLAN ENABLED

#### **Purpose**

Enables the use of the Plan Baseline that is created.

#### **Syntax**

PLAN\_ENABLED={ YES | NO }

#### **Default**

YES

#### **Usage Notes**

When set to the default, YES, the created Plan Baseline is used for plan selection. When set to NO, the Plan Baseline is created, but is not used until enabled manually.

## 10.6.53 MEMFORPREFETCH

#### **Purpose**

Use this option to speed up queries by pre-fetching the number of rows that can be accommodated in the specified memory.

### Syntax

MEMFORPREFETCH=integer

#### **Default**

No value set.

#### **Usage Notes**

This option can be used in configuration file or on the command line. The value of the integer is used for execution of all queries using explicit cursors, subject to the rules of precedence.

When used inline it must be placed before OPEN statements with explicit cursors. Then the number of rows pre-fetched when that OPEN is done is determined by the last in-line MEMFORPREFETCH option in effect.

The MEMFORPREFETCH default is no-value-is-set. To turn off prefetching, use MEMFORPREFETCH=0 on the command line.



Prefetching is turned off when LONG or LOB columns are being accessed.

MEMFORPREFETCH is used to enhance the performance of single row fetches.

MEMFORPREFETCH values have no effect when doing array fetches, regardless of which value is assigned.

There is no single perfect prefetch memory value that can be used to assist all the fetches in an application.

Therefore, when using the MEMFORPREFETCH option, you should test different values to give a general improvement across all statements in the program. Note that if certain statements need to be tuned individually, the MEMFORPREFETCH option can be specified inline using EXEC ORACLE OPTION. Note that this will affect all fetch statements that follow the command in your program. Select the appropriate prefetch memory to enhance the performance of any particular FETCH statement. To achieve this individual prefetch count, you should use the inline prefetch option rather than from the command line.

### 10.6.54 PREFETCH

#### **Purpose**

Use this option to speed up queries by pre-fetching a number of rows.

#### **Syntax**

PREFETCH=integer

#### **Default**

1

#### **Usage Notes**

Can be used in a configuration file or on the command-line. The value of the integer is used for execution of all queries using explicit cursors, subject to the rules of precedence.

When used in-line it must placed before OPEN statements with explicit cursors. Then the number of rows pre-fetched when that OPEN is done is determined by the last in-line PREFETCH option in effect.

The value range allowed is 0.. 65535.

## 10.6.55 RELEASE\_CURSOR

#### **Purpose**

Specifies how the cursors for SQL statements and PL/SQL blocks are handled in the cursor cache.

#### **Syntax**

RELEASE\_CURSOR={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Can be entered inline or on the command line.

You can use RELEASE CURSOR to improve the performance of your program.

When a SQL data manipulation statement is executed, its associated cursor is linked to an entry in the cursor cache. The cursor cache entry is in turn linked to an Oracle private SQL area, which stores information needed to process the statement. RELEASE\_CURSOR controls what happens to the link between the cursor cache and private SQL area.

When RELEASE\_CURSOR=YES, after Oracle executes the SQL statement and the cursor is closed, the precompiler immediately removes the link. This frees memory allocated to the private SQL area and releases parse locks. To make sure that associated resources are freed when you CLOSE a cursor, you must specify RELEASE\_CURSOR=YES.

When RELEASE\_CURSOR=NO, the link is maintained. The precompiler does not reuse the link unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are often executed because it speeds up subsequent executions. There is no need to re-parse the statement or allocate memory for an Oracle private SQL area.

For inline use with implicit cursors, set RELEASE\_CURSOR before executing the SQL statement. For inline use with explicit cursors, set RELEASE\_CURSOR before CLOSEing the cursor.

RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES.

#### **Related Topics**

Performance Tuning

### 10.6.56 RUNOUTLINE

#### **Purpose**

Provides the developer with the option of executing "create outline" statements either by using precompiler or by the developer manually at a later time.

#### **Syntax**

runoutline={yes | no}

#### Default

no

#### **Usage Notes**

If runoutline=yes, then the generated 'create outline' statements are executed by the precompiler/translator at the end of a successful precompilation.

The outline option should be set to true or category\_name when runoutline is used. Semantic check should be full when this option is turned on, which means option sqlcheck=full/ semantics. If sqlcheck=syntax/limited/none, then error will be generated.

## 10.6.57 SELECT\_ERROR

#### **Purpose**

Specifies whether your program generates an error when a SELECT statement returns more than one row, or more rows than a host array can accommodate.



#### **Syntax**

SELECT\_ERROR={YES | NO}

#### Default

YES

#### **Usage Notes**

Can be entered inline or on the command line.

When SELECT\_ERROR=YES, an error is generated when a single-row SELECT returns too many rows, or when an array SELECT returns more rows than the host array can accommodate. The result of the SELECT is indeterminate.

When SELECT\_ERROR=NO, no error is generated when a single-row SELECT returns too many rows, or when an array SELECT returns more rows than the host array can accommodate.

Whether you specify YES or NO, a random row is selected from the table. The only way to ensure a specific ordering of rows is to use the ORDER BY clause in your SELECT statement. When SELECT\_ERROR=NO and you use ORDER BY, Oracle returns the first row, or the first n rows when you are SELECTing into an array. When SELECT\_ERROR=YES, whether or not you use ORDER BY, an error is generated when too many rows are returned.

## 10.6.58 STMT\_CACHE

#### **Purpose**

Denotes the Statement cache size for the dynamic SQL statements.

#### **Syntax**

STMT\_CACHE = Range is 0 to 65535

#### Default

0

#### **Usage Notes**

The stmt\_cache option can be set to hold the anticipated number of distinct dynamic SQL statements in the application.

# 10.6.59 SYS\_INCLUDE

#### **Purpose**

Specifies the location of system header files.

#### **Syntax**

SYS INCLUDE=pathname | (path1, ..., pathn)

#### **Default**

System-specific.



#### **Usage Notes**

Pro\*C/C++ searches for standard system header files, such as stdio.h, in standard locations that are platform specific. For example, on almost all UNIX systems, the file stdio.h has the full path name /usr/include/stdio.h.

But C++ compilers can have system header files, such as stdio.h, that are not in the standard system locations. You can use the SYS\_INCLUDE command line option to specify a list of directory paths that Pro\*C/C++ searches to look for system header files. For example:

SYS INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)

The search path that you specify using SYS INCLUDE overrides the default header location.

If PARSE=NONE, the value specified in SYS\_INCLUDE is irrelevant for the precompilation, since there is no need for Pro\*C/C++ to include system header files in the precompilation. (You must, of course, still include Oracle-specific headers, such as sqlca.h. and system header files, with #include directives for pre-processing by the compiler.)

The precompiler searches directories in the following order:

- 1. The current directory
- 2. The system directory specified in the SYS\_INCLUDE precompiler option
- 3. The directories specified by the INCLUDE option, in the order entered
- 4. The built-in directory for standard header files

Because of step 3, you normally do not need to specify a directory path for standard header files such as sglca.h and sglda.h.

The syntax for specifying a directory path using the SYS\_INCLUDE option is system specific. Follow the conventions used for your operating system

#### Note:

For the SYS\_INCLUDE option, the precedence of option values gets reversed. Unlike other options in which the values get overwritten, SYS\_INCLUDE appends all the directory files that are mentioned in:

- Precompiler
- Pro\*C/C++ system configuration file
- Pro\*C/C++ user configuration file
- Command line
- Inline

However, there is a difference between passing values within or without brackets. If you pass a single value or directory list within brackets, then the existing value of SYS\_INCLUDE is overwritten. If you pass the list as a simple value without brackets, it will supplement any existing value.



### 10.6.60 THREADS

#### **Purpose**

When THREADS=YES, the precompiler searches for context declarations.

#### **Syntax**

THREADS={YES | NO}

#### Default

NO

#### **Usage Notes**

Cannot be entered inline.

This precompiler option is required for any program that requires multithreading support.

With THREADS=YES, the precompiler generates an error if no EXEC SQL CONTEXT USE directive is encountered before the first context is visible and an executable SQL statement is found.

#### **Related Topics**

Multithreaded Applications

## 10.6.61 TYPE\_CODE

#### **Purpose**

This micro option specifies whether ANSI or Oracle datatype codes are used in dynamic SQL Method 4. Its setting is the same as the setting of MODE option.

#### **Syntax**

TYPE\_CODE={ORACLE | ANSI}

### Default

**ORACLE** 

#### **Usage Notes**

Cannot be entered inline.

See the possible option settings in Table 14-3.

## 10.6.62 UNSAFE\_NULL

#### **Purpose**

Specifying UNSAFE\_NULL=YES prevents generation of ORA-01405 messages when fetching NULLs without using indicator variables.



#### **Syntax**

UNSAFE\_NULL={YES | NO}

#### **Default**

NO

#### **Usage Notes**

Cannot be entered inline.

The UNSAFE\_NULL=YES is allowed only when MODE=ORACLE.

The UNSAFE\_NULL option has no effect on host variables in an embedded PL/SQL block. You *must* use indicator variables to avoid ORA-01405 errors.

### 10.6.63 USERID

#### **Purpose**

Specifies an Oracle username and password.

#### **Syntax**

USERID=username/password[@dbname]

#### **Default**

None

#### **Usage Notes**

Can be entered only on the command line.

Do not specify this option when using the automatic connect feature, which accepts your Oracle username prefixed with CLUSTER\$. The actual value of the "CLUSTER\$" string is set as a parameter in the INIT.ORA file.

When SQLCHECK=SEMANTICS, if you want the precompiler to get needed information by connecting to Oracle and accessing the data dictionary, you must also specify USERID.

## 10.6.64 UTF16\_CHARSET

#### **Purpose**

Specify the character set form used by UNICODE(UTF16) variables.

#### **Syntax**

UTF16\_CHARSET={NCHAR\_CHARSET | DB\_CHARSET}

#### Default

NCHAR\_CHARSET

#### **Usage Notes**

Can be used only on the command line or in a configuration file, but not inline.



If UTF16 CHARSET=NCHAR CHARSET (the default), the UNICODE(UTF16) bind / define buffer is converted according to the server side National Character Set. There may be a performance impact when the target column is CHAR.

If UTF16\_CHAR=DB\_CHARSET, the UNICODE(UTF16) bind / define buffer is converted according to the database character set.



#### WARNING:

There may be data loss when the target column is NCHAR.

### 10.6.65 VARCHAR

#### **Purpose**

Instructs the Pro\*C/C++ precompiler to interpret some structs as VARCHAR host variables.

#### **Syntax**

VARCHAR={NO | YES}

#### **Default**

NO

#### **Usage Notes**

Can be entered only on the command line.

When VARCHAR=YES, a C struct that you code as

```
struct {
   short len;
   char arr[n];
```

is interpreted by the precompiler as a VARCHAR[n] host variable.

VARCHAR can be used in conjunction with the NLS\_CHAR option to designate a multibyte character variable.

## 10.6.66 VERSION

### **Purpose**

Determines which version of the object will be returned by the EXEC SQL OBJECT DEREF statement.

#### **Syntax**

VERSION={RECENT | LATEST | ANY}

#### **Default**

RECENT

#### **Usage Notes**

Can be entered inline by use of the EXEC ORACLE OPTION statement.

RECENT means that if the object has been selected into the object cache in the current transaction, then that object is returned. For transactions running in serializable mode, this option has the same effect as LATEST without incurring as many network round trips. Most applications should use RECENT.

LATEST means that if the object does not reside in the object cache, it is retrieved from the database. If It does reside in the object cache, it is refreshed from the server. Use LATEST with caution because it incurs the greatest number of network round trips. Use LATEST only when it is imperative that the object cache is kept as coherent as possible with the server buffer cache

ANY means that if the object already resides in the object cache, return that object. If not, retrieve the object from the server. ANY incurs the fewest network round trips. Use in applications that access read-only objects or when a user will have exclusive access to the objects.



# **Multithreaded Applications**

If your development platform does not support threads, ignore this chapter. This chapter contains the following topics:

- What are Threads?
- Runtime Contexts in Pro\*C/C++
- Runtime Context Usage Models
- User Interface Features for Multithreaded Applications
- Multithreaded Example
- Connection Pooling



When using XA with the Pro\*C/C++ Precompiler, you must use multithreading provided by XA. Use of multithreading provided by Pro\*C/C++ using the statement EXEC SQL ENABLE THREADS will result in an error.

## 11.1 What are Threads?

Multithreaded applications have multiple threads executing in a shared address space. Threads are "lightweight" subprocesses that execute within a process. They share 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 is often required to manage access to these variables from multiple threads within an application. Mutexes are the synchronization mechanism to insure that data integrity is preserved.

For further discussion of mutexes, see texts on multithreading. For more detailed information about multithreaded applications, see the documentation of your threads functions.

Pro\*C/C++ supports development of multithreaded Oracle Server applications (on platforms that support multithreaded applications) using the following:

- A command-line option to generate thread-safe code
- Embedded SQL statements and directives to support multithreading
- Thread-safe SQLLIB and other client-side Oracle libraries



While your platform may support a particular thread package, see your platformspecific Oracle documentation to determine whether Oracle supports it. The chapter's topics discuss how to use the preceding features to develop multithreaded Pro\*C/C++ applications:

- Runtime contexts for multithreaded applications
- Two models for using runtime contexts
- User-interface features for multithreaded applications
- Programming considerations for writing multithreaded applications with Pro\*C/C++
- An example multithreaded Pro\*C/C++ application

## 11.2 Runtime Contexts in Pro\*C/C++

To loosely couple a thread and a connection, Pro\*C/C++ introduces the notion of a runtime context. The runtime context includes the following resources and their current states:

- Zero or more connections to one or more Oracle Servers
- Zero or more cursors used for the server connections
- Inline options, such as MODE, HOLD\_CURSOR, RELEASE\_CURSOR, and SELECT\_ERROR

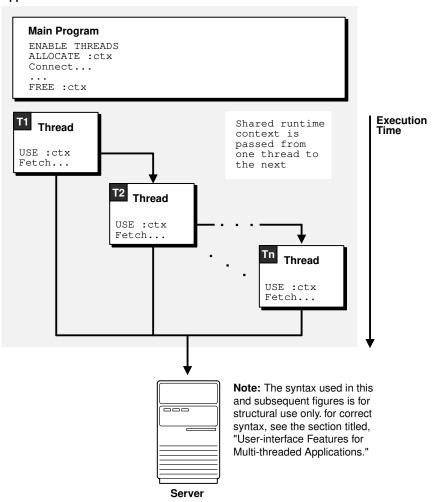
Rather than simply supporting a loose coupling between threads and connections, Pro\*C/C++ provides the ability to loosely couple threads with runtime contexts. Pro\*C/C++ allows your application to define a handle to a runtime context, and pass that handle from one thread to another.

For example, an interactive application spawns a thread, T1, to execute a query and return the first 10 rows to the application. T1 then terminates. After obtaining the necessary user input, another thread, T2, is spawned (or an existing thread is used) and the runtime context for T1 is passed to T2 so it can fetch the next 10 rows by processing the same cursor. See Figure 11-1.



Figure 11-1 Loosely Coupling Connections and Threads

#### **Application**



# 11.3 Runtime Context Usage Models

Two possible models for using runtime contexts in multithreaded Pro\*C/C++ applications are shown here:

- Multiple threads sharing a single runtime context
- Multiple threads using separate runtime contexts

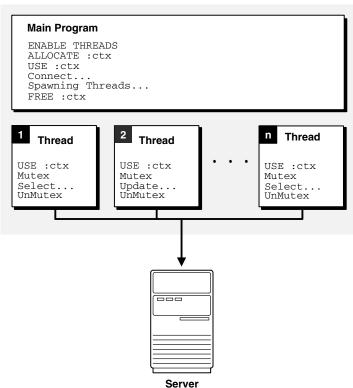
Regardless of the model you use for runtime contexts, you *cannot* share a runtime context between multiple threads *at the same time*. If two or more threads attempt to use the same runtime context simultaneously, a runtime error occurs.

## 11.3.1 Multiple Threads Sharing a Single Runtime Context

Figure 11-2 shows an application running in a multithreaded environment. The various threads share a single runtime context to process one or more SQL statements. Again, runtime contexts cannot be shared by multiple threads at the same time. The mutexes in Figure 11-2 show how to prevent concurrent usage.

Figure 11-2 Context Sharing Among Threads

#### **Application**

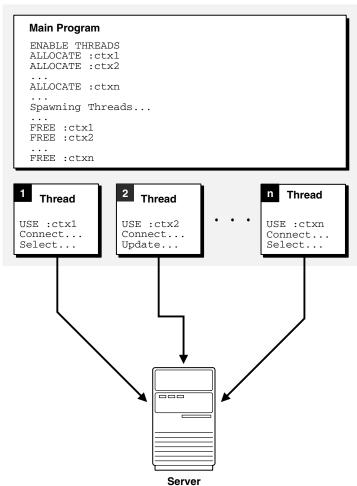


# 11.3.2 Multiple Threads Sharing Multiple Runtime Contexts

Figure 11-3 shows an application that executes multiple threads using multiple runtime contexts. In this situation, the application does not require mutexes, because each thread has a dedicated runtime context.

Figure 11-3 No Context Sharing Among Threads

#### **Application**



# 11.4 User Interface Features for Multithreaded Applications

The Pro\*C/C++ Precompiler provides the following user-interface features to support multithreaded applications:

- Command-line option, THREADS=YES|NO
- Embedded SQL statements and directives
- Thread-safe SQLLIB public functions

## 11.4.1 THREADS Option

With THREADS=YES specified on the command line, the Pro\*C/C++ Precompiler ensures that the generated code is thread-safe, given that you follow the guidelines. With THREADS=YES specified, Pro\*C/C++ verifies that all SQL statements execute within the scope of a user-defined runtime context. If your program does not meet this requirement, a precompiler error is returned.

#### **Related Topics**

Programming Considerations

## 11.4.2 Embedded SQL Statements and Directives

The following embedded SQL statements and directives support the definition and usage of runtime contexts and threads:

- EXEC SQL ENABLE THREADS;
- EXEC SQL CONTEXT ALLOCATE :context var;
- EXEC SQL CONTEXT USE { :context\_var | DEFAULT};
- EXEC SQL CONTEXT FREE :context\_var;

For these EXEC SQL statements, *context\_var* is the handle to the runtime context and must be declared of type **sql context** as follows:

```
sql context <context variable>;
```

Using DEFAULT means that the default (global) runtime context will be used in all embedded SQL statements that lexically follow until another CONTEXT USE statement overrides it.

### 11.4.2.1 EXEC SQL ENABLE THREADS

This executable SQL statement initializes a process that supports multiple threads. This must be the first executable SQL statement in your multithreaded application.



When using XA with the Pro\*C/C++ Precompiler, you must use multithreading provided by XA. Use of multithreading provided by Pro\*C using the statement EXEC SQL ENABLE THREADS will result in an error.

#### **Related Topics**

ENABLE THREADS (Executable Embedded SQL Extension)

## 11.4.2.2 EXEC SQL CONTEXT ALLOCATE

This executable SQL statement allocates and initializes memory for the specified runtime context; the runtime-context variable must be declared of type sql\_context.

#### **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension)

## 11.4.2.3 EXEC SQL CONTEXT USE

This directive instructs the precompiler to use the specified runtime context for subsequent executable SQL statements. The runtime context specified must be previously allocated using an EXEC SQL CONTEXT ALLOCATE statement.

The EXEC SQL CONTEXT USE directive works similarly to the EXEC SQL WHENEVER directive in that it affects all executable SQL statements which positionally follow it in a given

source file without regard to standard C scope rules. In the following example, the UPDATE statement in function2() uses the global runtime context, ctx1:

To use the global context after using a local context, add this code to function1():

In the next example, there is no global runtime context. The precompiler refers to the ctx1 runtime context in the generated code for the UPDATE statement. However, there is no context variable in scope for function2(), so errors are generated at compile time.

#### **Related Topics**

- CONTEXT OBJECT OPTION GET (Executable Embedded SQL Extension)
- CONTEXT ALLOCATE (Executable Embedded SQL Extension)

## 11.4.2.4 EXEC SQL CONTEXT FREE

This executable SQL statement frees the memory associated with the specified runtime context and places a null pointer in the host program variable.

#### **Related Topics**

CONTEXT FREE (Executable Embedded SQL Extension)

## 11.4.3 CONTEXT USE Examples

The following code fragments show how to use embedded SQL statements and precompiler directives for two typical programming models; they use *thread\_create()* to create threads.

The first example showing multiple threads using multiple runtime contexts:

```
main()
   sql context ctx1,ctx2;
                                    /* declare runtime contexts */
  EXEC SQL ENABLE THREADS;
  EXEC SQL CONTEXT ALLOCATE :ctx1;
  EXEC SQL CONTEXT ALLOCATE :ctx2;
/* spawn thread, execute function1 (in the thread) passing ctx1 */
  thread_create(..., function1, ctx1);
/* spawn thread, execute function2 (in the thread) passing ctx2 */
  thread create(..., function2, ctx2);
   EXEC SQL CONTEXT FREE :ctx1;
  EXEC SQL CONTEXT FREE :ctx2;
void function1(sql context ctx)
  EXEC SOL CONTEXT USE :ctx;
/* execute executable SOL statements on runtime context ctx1!!! */
}
void function2(sql context ctx)
  EXEC SQL CONTEXT USE :ctx;
/* execute executable SQL statements on runtime context ctx2!!! */
}
```

The next example shows how to use multiple threads that share a common runtime context. Because the SQL statements executed in function1() and function2() potentially execute at the same time, you must place mutexes around every executable EXEC SQL statement to ensure serial, therefore safe, manipulation of the data.



# 11.4.4 Programming Considerations

While Oracle ensures that the SQLLIB code is thread-safe, you are responsible for ensuring that your Pro\*C/C++ source code is designed to work properly with threads; for example, carefully consider your use of static and global variables.

In addition, multithreaded applications require design decisions regarding the following:

- Declaring the SQLCA as a thread-safe struct, typically an auto variable and one for each runtime context
- Declaring the SQLDA as a thread-safe struct, like the SQLCA, typically an auto variable and one for each runtime context
- Declaring host variables in a thread-safe fashion, in other words, carefully consider your use of static and global host variables.
- Avoiding simultaneous use of a runtime context in multiple threads
- Whether or not to use default database connections or to explicitly define them using the AT clause

Also, no more than one executable embedded SQL statement, for example, EXEC SQL UPDATE, may be outstanding on a runtime context at a given time.

Existing requirements for precompiled applications also apply. For example, all references to a given cursor must appear in the same source file.

# 11.5 Multithreaded Example

The following program is one approach to writing a multithreaded embedded SQL application. The program creates as many sessions as there are threads. Each thread executes zero or more transactions, that are specified in a transient structure called "records."

### Note:

This program was developed specifically for a Sun workstation running Solaris. Either the DCE or Solaris threads package is usable with this program. See your platform-specific documentation for the availability of threads packages.



```
* Requirements:
 * The program requires a table 'ACCOUNTS' to be in the schema
      scott/tiger. The description of ACCOUNTS is:
 * SQL> desc accounts
                                   Null?
 * ______ _______
 * ACCOUNT
                                           NUMBER (36)
 * BALANCE
                                           NUMBER (36,2)
 * For proper execution, the table should be filled with the accounts
       10001 to 10008.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlca.h>
                        _EXC__UNIX
            EXC OS
#define
#define
           CMA OS
                           CMA UNIX
#ifdef DCE THREADS
  #include <pthread.h>
#else
  #include <thread.h>
#endif
/* Function prototypes */
void err report();
#ifdef DCE_THREADS
void do_transaction();
#else
void *do_transaction();
#endif
void get_transaction();
void logon();
void logoff();
#define CONNINFO "scott/tiger"
#define THREADS 3
struct parameters
{ sql context * ctx;
 int thread id;
};
typedef struct parameters parameters;
struct record log
{ char action;
  unsigned int from account;
  unsigned int to account;
  float amount;
typedef struct record log record log;
record_log records[]= { 'M', 10001, 10002, 12.50 },
                       { 'M', 10001, 10003, 25.00 },
                       { 'M', 10001, 10003, 123.00 },
                       { 'M', 10001, 10003, 125.00 },
                       { 'M', 10002, 10006, 12.23 },
```

```
{ 'M', 10007, 10008, 225.23 },
                       { 'M', 10002, 10008, 0.70 },
                       { 'M', 10001, 10003, 11.30 },
                       { 'M', 10003, 10002, 47.50 },
                       { 'M', 10002, 10006, 125.00 },
                       { 'M', 10007, 10008, 225.00 },
                       { 'M', 10002, 10008, 0.70 },
                       { 'M', 10001, 10003, 11.00 },
                       { 'M', 10003, 10002, 47.50 },
                       { 'M', 10002, 10006, 125.00 },
                       { 'M', 10007, 10008, 225.00 },
                       { 'M', 10002, 10008, 0.70 },
                       { 'M', 10001, 10003, 11.00 },
                       { 'M', 10003, 10002, 47.50 },
                       { 'M', 10008, 10001, 1034.54}};
static unsigned int trx nr=0;
#ifdef DCE THREADS
pthread mutex t mutex;
#else
mutex t mutex;
#endif
/***********************************
 * Main
 *************************
main()
  sql context ctx[THREADS];
#ifdef DCE_THREADS
  pthread_t thread_id[THREADS];
  pthread_addr_t status;
#else
  thread t thread id[THREADS];
  int status;
#endif
  parameters params[THREADS];
  int i;
  EXEC SQL ENABLE THREADS;
  EXEC SQL WHENEVER SQLERROR DO err report(sqlca);
  /* Create THREADS sessions by connecting THREADS times */
  for(i=0;i<THREADS;i++)</pre>
   printf("Start Session %d....",i);
   EXEC SQL CONTEXT ALLOCATE :ctx[i];
    logon(ctx[i],CONNINFO);
  /*Create mutex for transaction retrieval */
#ifdef DCE THREADS
  if (pthread_mutex_init(&mutex,pthread_mutexattr_default))
#else
  if (mutex_init(&mutex, USYNC_THREAD, NULL))
#endif
    printf("Can't initialize mutex\n");
    exit(1);
```

```
/*Spawn threads*/
  for(i=0;i<THREADS;i++)</pre>
   params[i].ctx=ctx[i];
   params[i].thread id=i;
   printf("Thread %d... ",i);
#ifdef DCE THREADS
   if (pthread_create(&thread_id[i],pthread_attr_default,
        (pthread_startroutine_t)do_transaction,
        (pthread_addr_t) &params[i]))
#else
   if (status = thr create
    (NULL, 0, do transaction, &params[i], 0, &thread id[i]))
#endif
     printf("Cant create thread %d\n",i);
   else
     printf("Created\n");
 /* Logoff sessions....*/
 for(i=0;i<THREADS;i++)</pre>
    /*wait for thread to end */
    printf("Thread %d ....",i);
#ifdef DCE THREADS
    if (pthread_join(thread_id[i],&status))
      printf("Error when waiting for thread % to terminate\n", i);
     printf("stopped\n");
    printf("Detach thread...");
    if (pthread detach(&thread id[i]))
      printf("Error detaching thread! \n");
    else
      printf("Detached!\n");
#else
    if (thr join(thread id[i], NULL, NULL))
      printf("Error waiting for thread to terminate\n");
#endif
    printf("Stop Session %d....",i);
    logoff(ctx[i]);
    EXEC SQL CONTEXT FREE :ctx[i];
 /*Destroys mutex*/
#ifdef DCE THREADS
 if (pthread mutex destroy(&mutex))
 if (mutex_destroy(&mutex))
#endif
   printf("Can't destroy mutex\n");
   exit(1);
 }
}
/************************
```

```
* Function: do transaction
 ^{\star} Description: This functions executes one transaction out of the
               records array. The records array is 'managed' by
               the get_transaction function.
 **********************
#ifdef DCE THREADS
void do transaction (params)
#else
void *do_transaction(params)
#endif
parameters *params;
 struct sqlca sqlca;
 record log *trx;
 sql context ctx=params->ctx;
 /* Done all transactions ? */
 while (trx nr < (sizeof(records)/sizeof(record log)))</pre>
   get transaction(&trx);
   EXEC SQL WHENEVER SQLERROR DO err report(sqlca);
   EXEC SQL CONTEXT USE :ctx;
   printf("Thread %d executing transaction\n",params->thread id);
   switch(trx->action)
     case 'M': EXEC SQL UPDATE ACCOUNTS
                       SET BALANCE=BALANCE+:trx->amount
                       WHERE ACCOUNT=:trx->to account;
               EXEC SQL UPDATE ACCOUNTS
                       SET BALANCE=BALANCE-:trx->amount
                       WHERE ACCOUNT=:trx->from account;
               break:
      default: break;
   EXEC SQL COMMIT;
 }
}
/***********************
* Function: err_report
 ^{\star} Description: This routine prints out the most recent error
 *************************
        err report(sqlca)
struct sqlca sqlca;
 if (sqlca.sqlcode < 0)
  printf("\n%.*s\n\n",sqlca.sqlerrm.sqlerrml,sqlca.sqlerrm.sqlerrmc);
 exit(1);
/*********************
 * Function: logon
 ^{\star} Description: Logs on to the database as USERNAME/PASSWORD
```

```
logon(ctx,connect_info)
void
sql context ctx;
char * connect_info;
   EXEC SQL WHENEVER SQLERROR DO err_report(sqlca);
  EXEC SQL CONTEXT USE :ctx;
  EXEC SQL CONNECT : connect info;
   printf("Connected!\n");
/***********************
* Function: logoff
^{\star} Description: This routine logs off the database
************************
       logoff(ctx)
sql context ctx;
  EXEC SQL WHENEVER SQLERROR DO err report(sqlca);
  EXEC SQL CONTEXT USE :ctx;
  EXEC SQL COMMIT WORK RELEASE;
  printf("Logged off!\n");
}
/*********************
* Function: get_transaction
* Description: This routine returns the next transaction to process
**************************
void get transaction(trx)
record log ** trx;
#ifdef DCE THREADS
 if (pthread mutex lock(&mutex))
 if (mutex lock(&mutex))
#endif
  printf("Can't lock mutex\n");
 *trx=&records[trx_nr];
 trx_nr++;
#ifdef DCE THREADS
 if (pthread mutex unlock(&mutex))
#else
 if (mutex unlock(&mutex))
  printf("Can't unlock mutex\n");
```

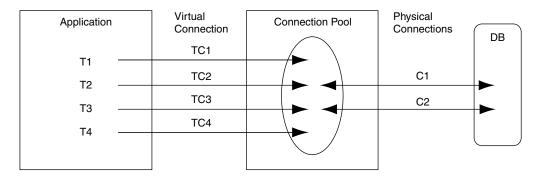
# 11.6 Connection Pooling

A connection pool is a group of physical connections to a database that can be re-used by several connections. The objective of the connection pooling feature is to improve

performance, and reduce resource use by avoiding usage of dedicated connections by each connection.

Figure 11-4 illustrates functionality of the connection pooling feature. In this example, four threads of the application are interacting with the database using the connection pool. The connection pool has two physical connections. The connection pool handle is used by four threads using different runtime contexts.

Figure 11-4 Connection Pooling



```
thread1()
{
EXEC SQL CONTEXT ALLOCATE :ctx1;
EXEC SQL CONTEXT USE:ctx1;
EXEC SQL CONNECT :uid AT :TC1 USING :db_string;
...
}
thread2()
{
EXEC SQL CONTEXT ALLOCATE :ctx2;
EXEC SQL CONNECT :uid AT :TC2 USING :db_string;
...
}
thread3()
{
EXEC SQL CONTEXT ALLOCATE :ctx3;
EXEC SQL CONNECT :uid AT :TC3 USING :db_string;
EXEC SQL CONNECT :uid AT :TC3 USING :db_string;
EXEC SQL AT :TC3 SELECT count(*) into :count FROM emp;
...
}
thread4()
{
EXEC SQL CONTEXT ALLOCATE :ctx4;
EXEC SQL CONNECT :uid AT :TC4 USING :db_string;
...
}
```

In this example, four named connections TC1, TC2, TC3, and TC4 are virtual connections created by threads T1, T2, T3, and T4 respectively. Named connections TC1, TC2, TC3, and TC4 from different runtime contexts share the same connection pool, and share physical

database connections available in the connection pool. Two physical connections, C1 and C2, serve four named connections and connect to the same database.

When the first connect request TC1 from thread T1 is received, SQLLIB creates a connection pool with one physical connection C1 to the database. When another connect request TC2 from thread T2 is sent to the same database, C1 serves the TC2 request to the database, if it is free. Otherwise, a new physical connection C2 is created to serve the request. If another connect request from thread T3 named TC3 comes in, TC3 either waits for a specified time or returns an error message, if both physical connections C1 and C2 are busy.

When thread T2 needs to select data using the TC2 named connection, it acquires any free physical connection, C1 or C2. After the request is served, the chosen connection will again be available in the connection pool, so that another named or virtual connection can utilize the same physical connection.



Usage of EXEC SQL CONNECT is not supported with the default context when connection pooling (CPOOL = YES) feature is enabled.

## 11.6.1 About Using the Connection Pooling Feature

This section comprises the following topics:

- How to Enable Connection Pooling
- Command Line Options for Connection Pooling
- Example
- Performance Tuning

### 11.6.1.1 How to Enable Connection Pooling

To enable connection pooling while precompiling an application, the user must set the command line option <code>CPOOL=YES</code>. Based on <code>CPOOL=YES/NO</code>, the connection pool feature is enabled or disabled.

#### Note:

- By default, CPOOL is set to NO and hence the connection pool feature is disabled. This feature cannot be enabled or disabled inline.
- Connection pool will not be created with external operating system authentication, even if CPOOL is set to YES.

## 11.6.1.2 Command Line Options for Connection Pooling

Table 11-1 lists the command line options for connection pooling:

Table 11-1 Command Line Options for Connection Pooling

Option	Valid Values	Default	Remarks
CPOOL	YES/NO	NO	Based on this option, the precompiler generates the appropriate code that directs SQLLIB to enable or disable the connection pool feature.
			<b>Note</b> : If this option is set to NO, other connection pooling options will be ignored by the precompiler.
CMAX	Valid values are 1 to 65535.	100	Specifies the maximum number of physical connections that can be opened for the database. CMAX value must be at least CMIN+CINCR.
			<b>Note</b> : Once this value is reached, more physical connections cannot be opened.
			In a typical application, running 100 concurrent database operations is more than sufficient. The user can set an appropriate value.
CMIN	Valid values are 1 to (CMAX- CINCR).	2	Specifies the minimum number of physical connections in the connection pool. Initially, all physical connections as specified through CMIN are opened to the server. Subsequently, physical connections are opened only when necessary. Users should set CMIN to the total number of planned or expected concurrent statements to be run by the application to get optimum performance. The default value is set to 2.
CINCR	Valid values are 1 to (CMAX- CMIN).	1	Allows the application to set the next increment for physical connections to be opened to the database, if the current number of physical connections is less than CMAX. To avoid creating unnecessary extra connections, the default value is set to 1.
CTIMEOUT	Valid values are 1 to 65535.	0 which means not set; hence will not time out.	Physical connections that are idle for more than the specified time (in seconds) are terminated to maintain an optimum number of open physical connections. If this attribute is not set, the physical connections are never timed out. Hence, physical connections will not be closed until the connection pool is terminated.
			<b>Note</b> : Creating a new physical connection will cost a round trip to the server.
CNOWAIT	Valid values are 1 to 65535.	0 which means not set; hence waits for a free connection.	This attribute determines if the application must repeatedly try for a physical connection when all other physical connections in the pool are busy, and the total number of physical connections has already reached its maximum. If physical connections are not available and no more physical connections can be opened, an error is thrown when this attribute is set. Otherwise, the call waits until it acquires another connection. By default, CNOWAIT is not to be set so a thread will wait until it can acquire a free connection, instead of returning an error.

A typical multithreaded application creates a pool of 'n' physical connections. The 'n' value needs to be specified by providing the CMIN value during precompilation. A minimum number of physical connections (CMIN) to the database are created on the first connect call. For new incoming requests, the mapping from a virtual connection (named connection) to a physical connection is carried out as described in the following section:

**Case 1:** If a physical connection is available (among the already opened connections), a new request will be served by this connection.

Case 2: If all physical connections are in use then,

**Case 2a:** If the number of opened connections has not reached the maximum limit (CMAX), new CINCR connections are created, and one of these connections is used to serve the request.

**Case 2b:** If the number of opened connections has reached the maximum limit (CMAX) without the CNOWAIT being set, the request waits until it acquires a connection. Otherwise, the application displays an error message *ORA 24401: cannot open further connections*.

### 11.6.1.3 Example

Refer to Figure 11-4 for an illustration of the following example.

```
Let CMIN be 1, CMAX be 2, and CINCR be 1.
```

Consider the following scenario. When the first request TC1 comes in, SQLLIB creates the connection pool with one physical connection C1. When another request TC2 comes in, the application checks if C1 is free. As C1 is used to serve the first request (Case 1), a new physical connection C2 is created to serve the request (Case 2a). If another request TC3 comes in, and if both C1 and C2 are busy, then TC3 either waits for a specified time or returns with an error message (Case 2b).

### 11.6.1.4 Performance Tuning

Users can set the connection pooling parameters to get better performance, based on the application. The Performance Graph in Figure 11-5 illustrates performance gain by changing the CMIN value for the Pro\*C/C++ Demo Program:1. Demo Program:2 illustrates performance gain by changing the CMAX parameter.

### 11.6.2 Demo Program:1

The following connection pool parameters are used while precompiling the Demo Program:1.

```
CMAX = 40
CINCR = 3
CMIN = varying values between 1 to 40
CPOOL = YES
CTIMEOUT - Do not set
```

(indicates that physical connections never time out)

```
CNOWAIT - Do not set
```

(indicates that a thread waits until it gets a free connection; see Table 11-1, for more details)

Other command line options required for this example are provided in the following section:

```
threads = yes
```



In this example, there are 40 threads and database operations are performed against the local database.

It was observed that with CPOOL=NO (without connection pooling), the time taken by the application was 6.1 seconds, whereas, with CPOOL=YES (with connection pooling), the minimum time taken by the application was 1.3 seconds (when CMIN was 2).

In both cases, the time taken for database query operations should remain the same since the connection pool only reduces the time taken for CONNECT statements. When CPOOL=NO the application will create 40 dedicated connections. When CPOOL=YES and CMIN=2 it will create 2 connections initially and, only if 2 threads access the connections concurrently, will it create more connections. Otherwise all threads will share those 2 connections. So the application potentially avoids 38 connections which in turn avoids 38 round trips to the server to establish those connections. This is where the three fold performance gain is seen.



These results were observed on a Sparc Ultra60 single CPU, 256 MB RAM machine, running one Oracle server on a Solaris 2.6 operating system; the server and client were running on the same machine.

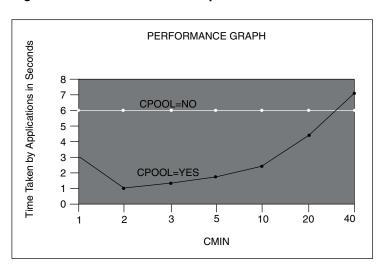


Figure 11-5 Performance Graph

The CPOOL=YES curve represents the time taken by the application when connection pooling is enabled. The CPOOL=NO curve represents the time taken by the application when connection pooling is disabled.

### 11.6.2.1 Example



```
#define
            EXC OS
                           EXC UNIX
                           _CMA_UNIX
#define
            CMA OS
#ifdef DCE THREADS
#include <pthread.h>
#else
#include <pthread.h>
typedef void*
              pthread addr t;
                (*pthread_startroutine_t) (void*);
typedef void*
#define pthread_attr_default (const pthread_attr_t *)NULL
#endif
/* Function prototypes */
void err_report();
void do_transaction();
void get transaction();
void logon();
void logoff();
#define CONNINFO "hr/hr"
#define THREADS 40
struct parameters
  sql context * ctx;
  int thread id;
 } ;
typedef struct parameters parameters;
struct timeval tp1;
struct timeval tp2;
/***********
* Main
************
main()
 sql context ctx[THREADS];
 pthread t thread id[THREADS];
 pthread addr t status;
 parameters params[THREADS];
 int i;
 EXEC SQL ENABLE THREADS;
 EXEC SQL WHENEVER SQLERROR DO err report(sqlca);
 if(gettimeofday(&tp1, (void*)NULL) == -1)
   perror("First: ");
   exit(0);
  /* Create THREADS sessions by connecting THREADS times */
 for(i=0;i<THREADS;i++)</pre>
   printf("Start Session %d....",i);
   EXEC SQL CONTEXT ALLOCATE :ctx[i];
   logon(ctx[i],CONNINFO);
  /*Spawn threads*/
```

```
for(i=0;i<THREADS;i++)</pre>
   params[i].ctx=ctx[i];
   params[i].thread id=i;
    if (pthread create(&thread id[i],pthread attr default,
      (pthread startroutine t) do transaction,
      (pthread addr t) &params[i]))
     printf("Cant create thread %d\n",i);
     printf("Created Thread %d\n", i);
  /* Logoff sessions....*/
  for(i=0;i<THREADS;i++)</pre>
    /*wait for thread to end */
   if (pthread join(thread id[i], &status))
     printf("Error when waiting for thread % to terminate\n", i);
     printf("stopped\n");
   if(i==THREADS-1)
     logoff(ctx[i]);
     EXEC SQL CONTEXT FREE :ctx[i];
  if(gettimeofday(&tp2, (void*)NULL) == -1)
   perror("Second: ");
   exit(0);
   printf(" \n THE TOTAL TIME TAKEN FOR THE PROGRAM EXECUTION = %f \n",
    (float)(tp2.tv_sec - tp1.tv_sec) + ((float)(tp2.tv_usec -
   tp1.tv_usec)/1000000.0));
}
/**************************
* Function: do transaction
* Description: This functions executes SELECT 5 times and calls COMMIT.
void do transaction(params)
parameters *params;
 struct sqlca sqlca;
 int src count;
  sql context ctx=params->ctx;
 EXEC SQL WHENEVER SQLERROR DO err report(sqlca);
 EXEC SQL CONTEXT USE :ctx;
 printf("Thread %d executing transaction\n",params->thread id);
 EXEC SQL COMMIT;
 EXEC SQL SELECT count(*) into :src_count from EMPLOYEES;
 EXEC SQL SELECT count(*) into :src count from EMPLOYEES;
 EXEC SQL SELECT count(*) into :src_count from EMPLOYEES;
 EXEC SQL SELECT count(*) into :src count from EMPLOYEES;
  EXEC SQL SELECT count(*) into :src count from EMPLOYEES;
```

```
/*****************
* Function: err_report
^{\star} Description: This routine prints out the most recent error
******************
       err report(sqlca)
struct sqlca sqlca;
 if (sqlca.sqlcode < 0)
   printf("\n%.*s\n\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
 exit(1);
/*****************
* Function: logon
^{\star} Description: Logs on to the database as USERNAME/PASSWORD
*************************************
       logon(ctx,connect info)
sql context ctx;
char * connect info;
 EXEC SQL WHENEVER SQLERROR DO err report(sqlca);
 EXEC SQL CONTEXT USE :ctx;
 EXEC SQL CONNECT : connect info;
 printf("Connected!\n");
/*************
* Function: logoff
* Description: This routine logs off the database
void
     logoff(ctx)
sql_context ctx;
 EXEC SQL WHENEVER SQLERROR DO err report(sqlca);
 EXEC SQL CONTEXT USE :ctx;
 EXEC SQL COMMIT WORK RELEASE;
 printf("Logged off!\n");
```

# 11.6.3 Demo Program:2

The following connection pool parameters are used while precompiling the Demo Program:2.

CMAX = varying values between 5 to 40

CINCR = 3

CMIN = varying values between 1 to 40

CPOOL = YES

CTIMEOUT - Do not set

(indicates that physical connections never time out)

CNOWAIT - Do not set

(indicates that a thread waits until it gets a free connection; see Table 11-1, for more details)

Other command line options required for this example are provided in the following section:

threads = yes

The following figure illustrates the performance graph for cpdemo2.



In this example there are 40 threads and database operations are performed against the local database.

In this example the best performance is observed with CMIN=5 and CMAX=14 when the program runs approximately 2.3 times faster compared to using CPOOL=NO. This is less of an improvement than "cpdemo1" though which ran faster with connection pooling enabled. The reason for this is because "cpdemo1" performs only simple SELECT statements whereas "cpdemo2" performs both UPDATE AND SELECT statements. Therefore "cpdemo1" spends more time creating connections than performing database operations. When connection pooling is enabled, time is saved as fewer connections are created. Hence overall performance improves. Since "cpdemo2" spends less time creating connections compared to performing database operations, the overall performance gain is less.

In the following graphs, the CPOOL=YES curve represents the time taken by the application when connection pooling is enabled. The CPOOL=NO curve represents the time taken by the application when connection pooling is disabled. The demo program "cpdemo2" creates 40 threads. With CPOOL=NO option, each thread establishes its own dedicated connection to the server. Hence 40 connections are created. The same demo program, when built with CPOOL=YES and CMAX=14, creates a maximum of 14 connections. These connections are shared across the 40 threads thus saving at least 26 connections and so avoiding 26 round-trips to the server.

The following two graphs show performance against varying CMIN and CMAX values respectively.

### 11.6.3.1 Case 1: By varying CMIN

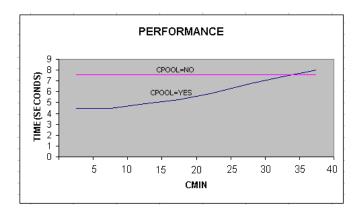


Figure 11-6 Performance Graph Case 1

The application takes around 7.5 seconds for execution with CPOOL=NO. With CPOOL=YES, and CMIN=8 and CMAX=14, the execution time reduces to 4.5 seconds. So, the performance improvement is about 1.7 times. The difference in performance is because of different

database operations (SELECT vs UPDATE) which is purely a server side activity and beyond the scope of connection pool feature which is a client side feature.

### 11.6.3.2 Case 2: By varying CMAX

PERFORMANCE CPOOL=NO TIME (SECONDS) в 5 CPOOL=YES 4 3 0 5 18 22 28 32 36 40 14

CMAX

Figure 11-7 Performance Graph Case 2

For the preceding graph the demo program was run with CMIN=5 and CINCR=3. The best performance was observed with CMAX=14. Execution takes around 7.4 seconds with CPOOL=NO. With CPOOL=YES, when CMAX=14 the execution time reduces to around 3.1 seconds, resulting in a 2.3 fold performance gain.

The performance improvement varies with CMAX. Therefore to obtain the best performance for a given application, the user should vary CMIN and CMAX until the optimal performance is achieved.

### 11.6.3.3 Example

```
cpdemo2.pc
  * Program to show the performance imcrement when the cpool option is used
  * Run this program with cpool=no. Record the time taken for the program to
  * execute
  * Compare the execution time
  ^{\star} This program also demonstrates the impact of a properly tuned CMAX
  * parameter on the performance
  ^{\star} Run the program with the following parameter values
  * CMIN=5
  * CINCR=2
  * CMAX=20
#include <stdio.h>
#include <sqlca.h>
#ifdef DCE THREADS
#include <pthread.h>
#else
#include <sys/time.h>
```

```
#include <pthread.h>
typedef void*
                  pthread addr t;
typedef void*
                (*pthread_startroutine_t) (void*);
#define pthread_attr_default (const pthread_attr_t *)NULL
#endif
#define CONNINFO "hr/hr"
#define THREADS 40
/**** prototypes ******** */
void selectFunction();
void updateFunction();
void err report(struct sqlca sqlca);
/***** parameter to the function selectFunction, updateFunction */
struct parameters
 sql context ctx;
 char connName[20];
 char dbName[20];
 int thread id;
};
typedef struct parameters parameters;
parameters params[THREADS];
struct timeval tp1;
struct timeval tp2;
int main()
{
 int i;
 pthread_t thread_id[THREADS];
 pthread_addr_t status;
 int thrNos[THREADS];
  for(i=0; i<THREADS; i++)</pre>
   thrNos[i] = i;
 EXEC SQL ENABLE THREADS;
  /* Time before executing the program */
 if(gettimeofday(&tp1, (void*)NULL) == -1){
   perror("First: ");
   exit(0);
 EXEC SQL WHENEVER SQLERROR DO err report(sqlca);
  /* connect THREADS times to the data base */
 for(i=0; i<THREADS; i++)</pre>
   strcpy(params[i].dbName, "");
   sprintf(params[i].connName,"conn%d", i);
   params[i].thread id = i;
    /* logon to the data base */
   EXEC SQL CONTEXT ALLOCATE :params[i].ctx;
```

```
EXEC SQL CONTEXT USE :params[i].ctx;
    EXEC SQL CONNECT : CONNINFO
      AT :params[i].connName USING :params[i].dbName;
  /* create THREADS number of threads */
  for(i=0;i<THREADS;i++)</pre>
      printf("Creating thread %d \n", i);
      if(i%2)
        /* do a select operation if the thread id is odd */
        if(pthread create(&thread id[i],pthread attr default,
            (pthread_startroutine_t) selectFunction,
             (pthread addr t) &params[i]))
        printf("Cant create thread %d \n", i);
      else
        /* otherwise do an update operation */
        if (pthread create (&thread id[i], pthread attr default,
             (pthread startroutine t) updateFunction,
             (pthread addr t) &params[i]))
        printf("Cant create thread %d \n", i);
  }
  for(i=0; i<THREADS; i++)</pre>
    if(pthread join(thread id[i],&status))
      printf("Error when waiting for thread % to terminate\n", i);
  if(gettimeofday(&tp2, (void*)NULL) == -1){
  perror("Second: ");
        exit(0);
  printf(" \n\nTHE TOTAL TIME TAKEN FOR THE PROGRAM EXECUTION = %f \n\n",
      (float)(tp2.tv sec - tp1.tv sec) + ((float)(tp2.tv usec -
      tp1.tv usec)/1000000.0));
  /* free the context */
  for(i=0; i<THREADS; i++)</pre>
    EXEC SQL CONTEXT USE :params[i].ctx;
    EXEC SQL AT :params[i].connName COMMIT WORK RELEASE;
    EXEC SQL CONTEXT FREE :params[i].ctx;
  return 0;
void selectFunction(parameters *params)
{
  struct sqlca sqlca;
  char empName[110][21];
  printf("Thread %d selecting .... \n", params->thread id);
```

```
EXEC SQL CONTEXT USE :params->ctx;
 EXEC SQL AT : params->connName
   SELECT FIRST_NAME into empName from EMPLOYEES;
 printf("Thread %d selected ....\n", params->thread id);
 return 0;
void updateFunction(parameters *params)
 struct sqlca sqlca;
 printf(" Thread %d Updating ... \n", params->thread_id);
 EXEC SQL CONTEXT USE :params->ctx;
 EXEC SQL AT :params->connName update EMPLOYEES
   set SALARY = 4000 where DEPARTMENT_ID = 10;
  /* commit the changes */
 EXEC SQL AT :params->connName COMMIT;
 printf(" Thread %d Updated ... \n", params->thread_id);
 return 0;
/******* Oracle error *******/
void err_report(struct sqlca sqlca)
 if (sqlca.sqlcode < 0)
  printf("\n%.*s\n\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
 exit(0);
```



# Part II

# **Applications**

#### Part II contains the following chapters:

- C++ Applications
- Oracle Dynamic SQL
- ANSI Dynamic SQL
- Oracle Dynamic SQL: Method 4
- LOBs
- Objects
- Collections
- The Object Type Translator
- User Exits



# C++ Applications

This chapter describes how you can use the Pro\*C/C++ Precompiler to precompile your C++ embedded SQL application, and how Pro\*C/C++ generates C++ compatible code. This chapter contains the following topics:

- · Understanding C++ Support
- Precompiling for C++
- Example Programs

# 12.1 Understanding C++ Support

To understand how Pro\*C/C++ supports C++, you must understand the basic functional capabilities of Pro\*C/C++. In particular, you must be aware of how Pro\*C/C++ differs from Pro\*C Version 1.

The basic capabilities of Pro\*C/C++ are:

- Full C preprocessor support. You can use #define, #include, #ifdef, and other
  preprocessor directives in your Pro\*C/C++ program, to handle constructs that the
  precompiler itself must process.
- Use of native C structures as host variables, including the ability to pass structs (or pointers to structs) as host variables to functions, and write functions that return host structures or struct pointers.

To support its C preprocessor capabilities and to enable host variables to be declared outside a special Declare Section, Pro\*C/C++ incorporates a complete C parser. The Pro\*C/C++ parser is a C parser; it cannot parse C++ code.

This means that for C++ support, you must be able to disable the C parser, or at least partially disable it. To disable the C parser, the Pro\*C/C++ Precompiler includes command-line options to give you control over the extent of C parsing that Pro\*C/C++ performs on your source code.

#### **Related Topics**

- Oracle Datatypes
- Precompiling for C++

### 12.1.1 No Special Macro Processing

Using C++ with Pro\*C/C++ does not require any special preprocessing or special macro processors that are external to Pro\*C/C++. There is no need to run a macro processor on the output of the precompiler to achieve C++ compatibility.

If you are a user of a release of Pro\*C/C++ Precompiler before this one, and you did use macro processors on the precompiler output, you should be able to precompile your C++ applications using Pro\*C/C++ with no changes to your code.

# 12.2 Precompiling for C++

To control precompilation so that it accommodates C++, there are four considerations:

- Code emission by the precompiler
- Parsing capability
- The output filename extension
- The location of system header files

#### 12.2.1 Code Generation

You must be able to specify what kind of code, C compatible code or C++ compatible code, the precompiler generates. Pro\*C/C++ by default generates C code. C++ is not a perfect superset of C. Some changes are required in generated code so that it can be compiled by a C++ compiler.

For example, in addition to emitting your application code, the precompiler interposes calls to its runtime library, SQLLIB. The functions in SQLLIB are C functions. There is no special C++ version of SQLLIB. For this reason, if you want to compile the generated code using a C++ compiler, Pro\*C/C++ must declare the functions called in SQLLIB as C functions.

For C output, the precompiler would generate a prototype such as

```
void sqlora(unsigned long *, void *);
```

But for C++ compatible code, the precompiler must generate

```
extern "C" {
void sqlora(unsigned long *, void *);
};
```

You control the kind of code Pro\*C/C++ generates using the precompiler option CODE. There are three values for this option: CPP, KR\_C, and ANSI\_C. The differences between these options can be illustrated by considering how the declaration of the SQLLIB function *sqlora* differs among the three values for the CODE option:

When you specify CODE=CPP, the precompiler

- Generates C++ compilable code.
- Gives the output file a platform-specific file extension (suffix), such as ".C" or ".cc", rather than the standard ".c" extension. (You can override this by using the CPP SUFFIX option.)
- Causes the value of the PARSE option to default to PARTIAL. You can also specify PARSE=NONE. If you specify PARSE=FULL, an error is issued at precompile time.
- Allows the use of the C++ style // Comments in your code. This style of Commenting is also permitted inside SQL statements and PL/SQL blocks when CODE=CPP.
- Pro\*C/C++ recognizes SQL optimizer hints that begin with //+.



 Requires that header files generated by OTT (Object Type Translator) must be included inside a declare section.

See Also:

"CODE" for information about the KR\_C and ANSI\_C values for the CODE option.

# 12.2.2 About Parsing Code

You must be able to control the effect of the Pro\*C/C++ C parser on your code. You do this by using the PARSE precompiler option, which controls how the precompiler's C parser treats your code.

The values and effects of the PARSE option are:

Table 12-1 Values and Effects of the PARSE Option

Values	Effects
PARSE=NONE	The value NONE has the following effects:
	<ul> <li>C preprocessor directives are understood only inside a declare section.</li> </ul>
	<ul> <li>You must declare all host variables inside a Declare Section.</li> </ul>
	<ul> <li>Precompiler release 1.x behavior</li> </ul>
PARSE=PARTIAL	The value PARTIAL has the following effects:
	<ul> <li>All preprocessor directives are understood</li> </ul>
	<ul> <li>You must declare all host variables inside a Declare Section</li> </ul>
	This option value is the default if CODE=CPP
PARSE=FULL	The value FULL has the following effects:
	The precompiler C parser runs on your code.
	<ul> <li>All Preprocessor directives are understood.</li> </ul>
	<ul> <li>You can declare host variables at any place that they can be declared legally in C.</li> </ul>

This option value is the default if the value of the CODE option is anything other than CPP. It is an error to specify PARSE=FULL when CODE=CPP.

To generate C++ compatible code, the PARSE option must be either NONE or PARTIAL. If PARSE=FULL, the C parser runs, and it does not understand C++ constructs in your code, such as classes.

# 12.2.3 Output Filename Extension

Most C compilers expect a default extension of ".c" for their input files. Different C++ compilers, however, can expect different filename extensions. The CPP\_SUFFIX option provides the ability to specify the filename extension that the precompiler generates. The value of this option is a string, without the quotes or the period. For example, CPP\_SUFFIX=cc, or CPP\_SUFFIX=C.



## 12.2.4 System Header Files

Pro\*C/C++ searches for standard system header files, such as stdio.h, in standard locations that are platform specific. Pro\*C/C++ does not search for header files with extensions such as hpp or h++. For example, on almost all UNIX systems, the file stdio.h has the full path name /usr/include/stdio.h.

But a C++ compiler has its own version of stdio.h that is not in the standard system location. When you are precompiling for C++, you must use the SYS\_INCLUDE precompiler option to specify the directory paths that Pro\*C/C++ searches to look for system header files. For example:

```
SYS INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)
```

Use the INCLUDE precompiler option to specify the location of non-system header files. The directories specified by the SYS\_INCLUDE option are searched before directories specified by the INCLUDE option.

If PARSE=NONE, the values specified in SYS\_INCLUDE and INCLUDE for system files are not relevant, since there is no need for Pro\*C/C++ to include system header files. (You can, of course, still include Pro\*C/C++-specific headers, such sqlca.h, using the EXEC SQL INCLUDE statement.)

#### **Related Topics**

INCLUDE

# 12.3 Example Programs

This section includes three example Pro\*C/C++ programs that include C++ constructs. Each of these programs is available on-line, in your demo directory.

### 12.3.1 cppdemo1.pc

```
cppdemo1.pc
* Prompts the user for an employee number, then queries the
* emp table for the employee's name, salary and commission.
* Uses indicator variables (in an indicator struct) to
   determine if the commission is NULL.
#include <iostream.h>
#include <stdio.h>
#include <string.h>
// Parse=partial by default when code=cpp,
// so preprocessor directives are recognized and parsed fully.
#define UNAME LEN
                           20
          PWD LEN
#define
                           40
// Declare section is required when CODE=CPP or
// PARSE={PARTIAL|NONE} or both.
EXEC SQL BEGIN DECLARE SECTION;
 {\tt VARCHAR\ username[UNAME\_LEN];\ //\ VARCHAR\ is\ an\ ORACLE\ pseudotype}
 varchar password[PWD LEN];  // can be in lower case also
```



```
// Define a host structure for the output values
  // of a SELECT statement
  struct empdat {
      VARCHAR emp_name[UNAME_LEN];
      float
                salary;
      float
               commission;
  } emprec;
  // Define an indicator struct to correspond to the
  // host output struct
  struct empind {
      short emp_name_ind;
      short
             sal ind;
      short
               comm ind;
  } emprec ind;
  // Input host variables
  int emp number;
  int total queried;
EXEC SOL END DECLARE SECTION;
// Define a C++ class object to match the desired
// struct from the preceding declare section.
class emp {
  char ename[UNAME_LEN];
  float salary;
  float commission;
public:
  // Define a constructor for this C++ object that
  // takes ordinary C objects.
  emp(empdat&, empind&);
  friend ostream& operator<<(ostream&, emp&);</pre>
};
emp::emp(empdat& dat, empind& ind)
  strncpy(ename, (char *)dat.emp name.arr, dat.emp name.len);
  ename[dat.emp name.len] = '\0';
  this->salary = dat.salary;
  this->commission = (ind.comm ind < 0) ? 0 : dat.commission;
ostream& operator<<(ostream& s, emp& e)
  return s << e.ename << " earns " << e.salary <<
              " plus " << e.commission << " commission."
           << endl << endl;
// Include the SQL Communications Area
// You can use #include or EXEC SQL INCLUDE
#include <sqlca.h>
// Declare error handling function
void sql error(char *msg);
main()
  char temp char[32];
```



```
// Register sql error() as the error handler
 EXEC SQL WHENEVER SQLERROR DO sql error("ORACLE error:");
 // Connect to ORACLE. Program calls sql error()
 // if an error occurs
 // when connecting to the default database.
 // Note the (char ^{\star}) cast when
 // copying into the VARCHAR array buffer.
 username.len = strlen(strcpy((char *)username.arr, "SCOTT"));
 password.len = strlen(strcpy((char *)password.arr, "TIGER"));
 EXEC SQL CONNECT :username IDENTIFIED BY :password;
 // Here again, note the (char *) cast when using VARCHARs
 cout << "\nConnected to ORACLE as user: "</pre>
       << (char *)username.arr << endl << endl;
 // Loop, selecting individual employee's results
 total queried = 0;
 while (1)
      emp number = 0;
     printf("Enter employee number (0 to quit): ");
     gets(temp char);
      emp number = atoi(temp char);
      if (emp number == 0)
       break;
      // Branch to the notfound label when the
      // 1403 ("No data found") condition occurs
      EXEC SQL WHENEVER NOT FOUND GOTO notfound;
     EXEC SQL SELECT ename, sal, comm
         INTO :emprec INDICATOR :emprec_ind // You can also use
                                             // C++ style
                                    // Comments in SQL statemtents.
         FROM EMP
         WHERE EMPNO = :emp number;
        // Basic idea is to pass C objects to
        // C++ constructors thus
        // creating equivalent C++ objects used in the
        // usual C++ way
        emp e(emprec, emprec ind);
        cout << e;
      total queried++;
     continue;
notfound:
     cout << "Not a valid employee number - try again."</pre>
           << endl << endl;
  } // end while(1)
 cout << endl << "Total rows returned was "</pre>
     << total queried << endl;
 cout << "Have a nice day!" << endl << endl;</pre>
  // Disconnect from ORACLE
 EXEC SQL COMMIT WORK RELEASE;
 exit(0);
}
```

```
void sql_error(char *msg)
{
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    cout << endl << msg << endl;
    cout << sqlca.sqlerrm.sqlerrmc << endl;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}</pre>
```

## 12.3.2 cppdemo2.pc

The next application is a simple modular example. First, execute the following SQL script, cppdemo2.sql, in SQL\*Plus:

```
Rem This is the SQL script that accompanies the cppdemo2 C++ Demo Rem Program. Run this prior to Precompiling the empclass.pc file.

/ CONNECT SCOTT/TIGER

/ CREATE OR REPLACE VIEW emp_view AS SELECT ename, empno FROM EMP

/ CREATE OR REPLACE PACKAGE emp_package AS

TYPE emp_cursor_type IS REF CURSOR RETURN emp_view%ROWTYPE;

PROCEDURE open_cursor(curs IN OUT emp_cursor_type);

END emp_package;

/ CREATE OR REPLACE PACKAGE BODY emp_package AS

PROCEDURE open_cursor(curs IN OUT emp_cursor_type) IS

BEGIN

OPEN curs FOR SELECT ename, empno FROM emp_view ORDER BY ename ASC;

END;

END emp_package;

/ EXIT

/ EXIT
```

The header file empclass.h defines the class emp:

```
// This class definition may be included in a Pro*C/C++ application
// program using the EXEC SQL INCLUDE directive only. Because it
// contains EXEC SQL syntax, it may not be included using a #include
// directive. Any program that includes this header must be
// precompiled with the CODE=CPP option. This emp class definition
// is used when building the cppdemo2 C++ Demo Program.
class emp
 public:
            // Constructor: ALLOCATE Cursor Variable
   emp();
   ~emp(); // Desctructor: FREE Cursor Variable
   void open();
                             // Open Cursor
   void fetch() throw (int); // Fetch (throw NOT FOUND condition)
    void close();
                             // Close Cursor
    void emp error();
                             // Error Handler
    EXEC SQL BEGIN DECLARE SECTION;
     // When included using EXEC SQL INCLUDE, class variables have
```

```
// global scope and are thus basically treated as ordinary
     // global variables by Pro*C/C++ during precompilation.
     char ename[10];
     int empno;
    EXEC SQL END DECLARE SECTION;
 private:
    EXEC SQL BEGIN DECLARE SECTION;
     // Pro*C/C++ treats this as a simple global variable also.
     SQL CURSOR emp cursor;
    EXEC SQL END DECLARE SECTION;
};
The code in empclass.pc contains the emp methods:
#include <stdio.h>
#include <stdlib.h>
// This example uses a single (global) SQLCA that is shared by the
// emp class implementation as well as the main program for this
// application.
#define SQLCA STORAGE CLASS extern
#include <sqlca.h>
// Include the emp class specification in the implementation of the
// class body as well as the application program that makes use of it.
EXEC SQL INCLUDE empclass.h;
emp::emp()
  // The scope of this WHENEVER statement spans the entire module.
 // Note that the error handler function is really a member function
 // of the emp class.
 EXEC SQL WHENEVER SQLERROR DO emp error();
 EXEC SQL ALLOCATE :emp cursor; // Constructor - ALLOCATE Cursor.
emp::~emp()
{
 EXEC SQL FREE :emp cursor;
                                  // Destructor - FREE Cursor.
void emp::open()
 EXEC SQL EXECUTE
     emp package.open cursor(:emp cursor);
   END;
 END-EXEC;
void emp::close()
 EXEC SQL CLOSE :emp cursor;
void emp::fetch() throw (int)
 EXEC SQL FETCH :emp_cursor INTO :ename, :empno;
 if (sqlca.sqlcode == 1403)
   throw sqlca.sqlcode; // Like a WHENEVER NOT FOUND statement.
}
```

```
void emp::emp error()
 printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 exit(1);
The main program, cppdemo2.pc, uses the cursor variable:
// Pro*C/C++ sample program demonstrating a simple use of Cursor Variables
// implemented within a C++ class framework. Build this program as follows
//
//
    1. Execute the cppdemo2.sql script within SQL*Plus
//
   2. Precompile the empclass.pc program as follows
//
       > proc code=cpp sqlcheck=full user=scott/tiger lines=yes empclass
//
   3. Precompile the cppdemo2.pc program as follows
       > proc code=cpp lines=yes cppdemo2
//
//
    4. Compile and Link
//
// Note that you may have to specify various include directories using the
// include option when precompiling.
#include <stdio.h>
#include <stdlib.h>
#include <sqlca.h>
static void sql error()
 printf("%.*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 EXEC SQL ROLLBACK WORK RELEASE;
 exit(1);
// Physically include the emp class definition in this module.
EXEC SQL INCLUDE empclass.h;
int main()
 EXEC SQL BEGIN DECLARE SECTION;
   char *uid = "scott/tiger";
 EXEC SQL END DECLARE SECTION;
 EXEC SQL WHENEVER SQLERROR DO sql error();
 EXEC SQL CONNECT : uid;
  emp *e = new emp(); // Invoke Constructor - ALLOCATE Cursor Variable.
  e->open();
                     // Open the Cursor.
  while (1)
     // Fetch from the Cursor, catching the NOT FOUND condition
     // thrown by the fetch() member function.
     try { e->fetch(); } catch (int code)
        { if (code == 1403) break; }
     printf("Employee: %s[%d]\n", e->ename, e->empno);
  e->close();
                    // Close the Cursor.
```



# 12.3.3 cppdemo3.pc

```
* cppdemo3.pc : An example of C++ Inheritance
 ^{\star} This program finds all salesman and prints their names
 * followed by how much they earn in total (ie; including
 * any commissions).
#include <iostream.h>
#include <stdio.h>
#include <sqlca.h>
#include <string.h>
#define NAMELEN 10
class employee {
                    // Base class is a simple employee
public:
  char ename[NAMELEN];
 int sal;
  employee(char *, int);
};
employee::employee(char *ename, int sal)
  strcpy(this->ename, ename);
  this->sal = sal;
// A salesman is a kind of employee
class salesman : public employee
  int comm;
public:
  salesman(char *, int, int);
  friend ostream& operator<<(ostream&, salesman&);</pre>
};
// Inherits employee attributes
salesman::salesman(char *ename, int sal, int comm)
  : employee(ename, sal), comm(comm) {}
ostream& operator<<(ostream& s, salesman& m)
  return s << m.ename << m.sal + m.comm << endl;</pre>
void print(char *ename, int sal, int comm)
  salesman man(ename, sal, comm);
  cout << man;</pre>
main()
```



```
EXEC SQL BEGIN DECLARE SECTION;
   char *uid = "scott/tiger";
   char ename[NAMELEN];
   int sal, comm;
   short comm ind;
 EXEC SQL END DECLARE SECTION;
 EXEC SQL WHENEVER SQLERROR GOTO error;
 EXEC SQL CONNECT : uid;
 EXEC SQL DECLARE c CURSOR FOR
   SELECT ename, sal, comm FROM emp WHERE job = 'SALESMAN'
    ORDER BY ename;
 EXEC SQL OPEN c;
 cout << "Name
                Salary" << endl << "----" << endl;
 EXEC SQL WHENEVER NOT FOUND DO break;
 while(1)
  {
    EXEC SQL FETCH c INTO :ename, :sal, :comm:comm ind;
   print(ename, sal, (comm_ind < 0) ? 0 : comm);</pre>
 EXEC SQL CLOSE c;
 exit(0);
error:
 cout << endl << sqlca.sqlerrm.sqlerrmc << endl;</pre>
 exit(1);
```



# Oracle Dynamic SQL

This chapter shows you how to use Oracle Dynamic SQL, an advanced programming technique that adds flexibility and functionality to your applications. You will learn four methods for writing programs that accept and process SQL statements at run time. This chapter contains the following topics:

#### Note:

Oracle Dynamic SQL does not support object types, cursor variables, arrays of structs, DML returning clauses, Unicode variables, and LOBs. Use ANSI Dynamic SQL method 4 instead.

- What is Dynamic SQL?
- Advantages and Disadvantages of Dynamic SQL
- When to Use Dynamic SQL
- Requirements for Dynamic SQL Statements
- · How Dynamic SQL Statements are Processed
- Methods for Using Dynamic SQL
- Using Method 1
- Using Method 2
- Using Method 3
- Using Method 4
- About Using the DECLARE STATEMENT Statement
- About Using PL/SQL
- Dynamic SQL Statement Caching
- About Boolean Data Type Support

# 13.1 What is Dynamic SQL?

Most database applications do a specific job. For example, a simple program might prompt the user for an employee number, then update rows in the EMP and DEPT tables. In this case, you know the makeup of the UPDATE statement at precompile time. That is, you know which tables might be changed, the constraints defined for each table and column, which columns might be updated, and the datatype of each column.

However, some applications must accept (or build) and process a variety of SQL statements at run time. For example, a general-purpose report writer must build different SELECT statements for the various reports it generates. In this case, the statement's makeup is unknown until run time. Such statements can, and probably will, change from execution to execution. They are aptly called *dynamic* SQL statements.

Unlike static SQL statements, dynamic SQL statements are not embedded in your source program. Instead, they are stored in character strings input to or built by the program at run time. They can be entered interactively or read from a file.

# 13.2 Advantages and Disadvantages of Dynamic SQL

Host programs that accept and process dynamically defined SQL statements are more versatile than plain embedded SQL programs. Dynamic SQL statements can be built interactively with input from users having little or no knowledge of SQL.

For example, your program might simply prompt users for a search condition to be used in the WHERE clause of a SELECT, UPDATE, or DELETE statement. A more complex program might allow users to choose from menus listing SQL operations, table and view names, column names, and so on. Thus, dynamic SQL lets you write highly flexible applications.

However, some dynamic queries require complex coding, the use of special data structures, and more runtime processing. While you might not notice the added processing time, you might find the coding difficult unless you fully understand dynamic SQL concepts and methods.

# 13.3 When to Use Dynamic SQL

In practice, static SQL will meet nearly all your programming needs. Use dynamic SQL only if you need its open-ended flexibility. Its use is suggested when one of the following items is unknown at precompile time:

- Text of the SQL statement (commands, clauses, and so on)
- The number of host variables
- The datatypes of host variables
- References to database objects such as columns, indexes, sequences, tables, usernames, and views

# 13.4 Requirements for Dynamic SQL Statements

To represent a dynamic SQL statement, a character string must contain the text of a valid SQL statement, but *not* contain the EXEC SQL clause, or the statement terminator, or any of the following embedded SQL commands:

- ALLOCATE
- CLOSE
- DECLARE
- DESCRIBE
- EXECUTE
- FETCH
- FREE
- GET
- INCLUDE
- OPEN
- PREPARE



- SET
- WHENEVER

In most cases, the character string can contain *dummy* host variables. They hold places in the SQL statement for actual host variables. Because dummy host variables are just placeholders, you do not declare them and can name them anything you like. For example, Oracle makes no distinction between the following two strings:

```
'DELETE FROM EMP WHERE MGR = :mgr_number AND JOB = :job_title'
'DELETE FROM EMP WHERE MGR = :m AND JOB = :j'
```

# 13.5 How Dynamic SQL Statements are Processed

Typically, an application program prompts the user for the text of a SQL statement and the values of host variables used in the statement. Oracle then parses the SQL statement to ensure it meets syntax rules.

Next, Oracle *binds* the host variables to the SQL statement. That is, Oracle gets the addresses of the host variables so that it can read or write their values.

Then Oracle *executes* the SQL statement. That is, Oracle does what the SQL statement requested, such as deleting rows from a table.

The SQL statement can be executed repeatedly using new values for the host variables.

# 13.6 Methods for Using Dynamic SQL

This section introduces four methods you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, then offers guidelines for choosing the right method. Later sections show you how to use the methods, and include example programs that you can study.

The four methods are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and so on. However, each method is most useful for handling a certain kind of SQL statement, as Table 13-1 shows:

Table 13-1 Methods for Using Dynamic SQL

Method	Kind of SQL Statement
1	non-query without host variables
2	non-query with known number of input host variables
3	query with known number of select-list items and input host variables
4	query with unknown number of select-list items or input host variables



The term select-list item includes column names and expressions such as SAL \* 1.10 and MAX(SAL).



### 13.6.1 Method 1

This method lets your program accept or build a dynamic SQL statement, then immediately execute it using the EXECUTE IMMEDIATE command. The SQL statement must not be a query (SELECT statement) and must not contain any placeholders for input host variables. For example, the following host strings qualify:

```
'DELETE FROM EMP WHERE DEPTNO = 20'
'GRANT SELECT ON EMP TO scott'
```

With Method 1, the SQL statement is parsed every time it is executed.

### 13.6.2 Method 2

This method lets your program accept or build a dynamic SQL statement, then process it using the PREPARE and EXECUTE commands. The SQL statement must not be a query. The number of placeholders for input host variables and the datatypes of the input host variables must be known at precompile time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:emp_name, :job_title)'
'DELETE FROM EMP WHERE EMPNO = :emp number'
```

With Method 2, the SQL statement is parsed just once, but can be executed many times with different values for the host variables. SQL data definition statements such as CREATE and GRANT are executed when they are PREPAREd.

### 13.6.3 Method 3

This method lets your program accept or build a dynamic query, then process it using the PREPARE command with the DECLARE, OPEN, FETCH, and CLOSE cursor commands. The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables must be known at precompile time. For example, the following host strings qualify:

```
'SELECT DEPTNO, MIN(SAL), MAX(SAL) FROM EMP GROUP BY DEPTNO'
'SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :dept number'
```

### 13.6.4 Method 4

This method lets your program accept or build a dynamic SQL statement, then process it using descriptors. The number of select-list items, the number of placeholders for input host variables, and the datatypes of the input host variables can be unknown until run time. For example, the following host strings fall into this category:

```
'INSERT INTO EMP (<unknown>) VALUES (<unknown>)'
'SELECT <unknown> FROM EMP WHERE DEPTNO = 20'
```

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables.

#### **Related Topics**

Using Method 4



### 13.6.5 Guidelines

With all four methods, you must store the dynamic SQL statement in a character string, which must be a host variable or quoted literal. When you store the SQL statement in the string, omit the keywords EXEC SQL and the ';' statement terminator.

With Methods 2 and 3, the number of placeholders for input host variables and the datatypes of the input host variables must be known at precompile time.

Each succeeding method imposes fewer constraints on your application, but is more difficult to code. As a rule, use the simplest method you can. However, if a dynamic SQL statement will be executed repeatedly by Method 1, use Method 2 instead to avoid reparsing for each execution.

Method 4 provides maximum flexibility, but requires complex coding and a full understanding of dynamic SQL concepts. In general, use Method 4 only if you cannot use Methods 1, 2, or 3.

The decision logic in Figure 13-1 will help you choose the right method.

### 13.6.5.1 About Avoiding Common Errors

If you precompile using the command-line option DBMS=V6\_CHAR, blank-pad the array before storing the SQL statement. That way, you clear extraneous characters. This is especially important when you reuse the array for different SQL statements. As a rule, always initialize (or re-initialize) the host string before storing the SQL statement. Do *not* null-terminate the host string. Oracle does not recognize the null terminator as an end-of-string sentinel. Instead, Oracle treats it as part of the SQL statement.

If you precompile with the command-line option DBMS=V8, make sure that the string is null terminated before you execute the PREPARE or EXECUTE IMMEDIATE statement.

Regardless of the value of DBMS, if you use a VARCHAR variable to store the dynamic SQL statement, make sure the length of the VARCHAR is set (or reset) correctly before you execute the PREPARE or EXECUTE IMMEDIATE statement.



About the SQL statement... Does its select list yes yes Is it a query? contain an unknown number of items? no no Does it contain an Does it contain yes yes unknown number of input host input host variables? variables? no no Does it yes contain an unknown number of input host variables? no yes Will it be executed repeatedly? no Method 1 Method 2 Method 3 Method 4

Figure 13-1 Choosing the Right Method

# 13.7 Using Method 1

The simplest kind of dynamic SQL statement results only in "success" or "failure" and uses no host variables. Some examples follow:

```
'DELETE FROM table_name WHERE column_name = constant'
'CREATE TABLE table_name ...'
'DROP INDEX index_name'
'UPDATE table_name SET column_name = constant'
'GRANT SELECT ON table_name TO username'
'REVOKE RESOURCE FROM username'
```



Method 1 parses, then immediately executes the SQL statement using the EXECUTE IMMEDIATE command. The command is followed by a character string (host variable or literal) containing the SQL statement to be executed, which cannot be a query.

The syntax of the EXECUTE IMMEDIATE statement follows:

```
EXEC SQL EXECUTE IMMEDIATE { :host_string | string_literal };
```

In the following example, you use the host variable *dyn\_stmt* to store SQL statements input by the user:

```
char dyn_stmt[132];
...
for (;;)
{
    printf("Enter SQL statement: ");
    gets(dyn_stmt);
    if (*dyn_stmt == '\0')
        break;
    /* dyn_stmt now contains the text of a SQL statement */
    EXEC SQL EXECUTE IMMEDIATE :dyn_stmt;
}
...
```

You can also use string literals, as the following example shows:

```
EXEC SQL EXECUTE IMMEDIATE 'REVOKE RESOURCE FROM MILLER';
```

Because EXECUTE IMMEDIATE parses the input SQL statement before every execution, Method 1 is best for statements that are executed only once. Data definition language statements usually fall into this category.

# 13.7.1 Example Program: Dynamic SQL Method 1

The following program uses dynamic SQL Method 1 to create a table, insert a row, commit the insert, then drop the table. This program is available on-line in your demo directory in the file sample6.pc.

```
/*
  * sample6.pc: Dynamic SQL Method 1
  *
  * This program uses dynamic SQL Method 1 to create a table,
  * insert a row, commit the insert, then drop the table.
  */

#include <stdio.h>
#include <string.h>

/* Include the SQL Communications Area, a structure through
  * which ORACLE makes runtime status information such as error
  * codes, warning flags, and diagnostic text available to the
  * program.
  */
#include <sqlca.h>

/* Include the ORACLE Communications Area, a structure through
  * which ORACLE makes additional runtime status information
  * available to the program.
  */
#include <oraca.h>
```



```
/* The ORACA=YES option must be specified to enable you
 * to use the ORACA.
EXEC ORACLE OPTION (ORACA=YES);
/* Specifying the RELEASE CURSOR=YES option instructs Pro*C
 * to release resources associated with embedded SQL
 * statements after they are executed. This ensures that
 * ORACLE does not keep parse locks on tables after data
 ^{\star} manipulation operations, so that subsequent data definition
 * operations on those tables do not result in a parse-lock
 * error.
 */
EXEC ORACLE OPTION (RELEASE CURSOR=YES);
void dyn error();
main()
/* Declare the program host variables. */
    char *username = "SCOTT";
          *password = "TIGER";
    char
          *dynstmt1;
    char
            dynstmt2[10];
    char
    VARCHAR dynstmt3[80];
/* Call routine dyn error() if an ORACLE error occurs. */
    EXEC SQL WHENEVER SQLERROR DO dyn error("Oracle error:");
/* Save text of current SQL statement in the ORACA if an
 * error occurs.
    oraca.orastxtf = ORASTFERR;
/* Connect to Oracle. */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts("\nConnected to ORACLE.\n");
/* Execute a string literal to create the table. This
 * usage is actually not dynamic because the program does
 * not determine the SQL statement at run time.
    puts("CREATE TABLE dyn1 (col1 VARCHAR2(4))");
    EXEC SQL EXECUTE IMMEDIATE
         "CREATE TABLE dyn1 (col1 VARCHAR2(4))";
/* Execute a string to insert a row. The string must
 * be null-terminated. This usage is dynamic because the
 ^{\star} SQL statement is a string variable whose contents the
 * program can determine at run time.
    dynstmt1 = "INSERT INTO DYN1 values ('TEST')";
    puts(dynstmt1);
    EXEC SQL EXECUTE IMMEDIATE :dynstmt1;
```

```
/* Execute a SQL statement in a string to commit the insert.
 * Pad the unused trailing portion of the array with spaces.
 * Do NOT null-terminate it.
    strncpy(dynstmt2, "COMMIT
                                 ", 10);
    printf("%.10s\n", dynstmt2);
    EXEC SQL EXECUTE IMMEDIATE : dynstmt2;
/* Execute a VARCHAR to drop the table. Set the .len field
 * to the length of the .arr field.
    strcpy(dynstmt3.arr, "DROP TABLE DYN1");
    dynstmt3.len = strlen(dynstmt3.arr);
    puts((char *) dynstmt3.arr);
    EXEC SQL EXECUTE IMMEDIATE : dynstmt3;
/* Commit any outstanding changes and disconnect from Oracle. */
    EXEC SOL COMMIT RELEASE;
    puts("\nHave a good day!\n");
    return 0;
}
void
dyn error (msg)
char *msg;
/* This is the Oracle error handler.
* Print diagnostic text containing the error message,
 * current SQL statement, and location of error.
    printf("\n%.*s\n",
       sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf("in \"%.*s...\'\n",
       oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf("on line %d of %.*s.\n\n",
        oraca.oraslnr, oraca.orasfnm.orasfnml,
        oraca.orasfnm.orasfnmc);
/* Disable Oracle error checking to avoid an infinite loop
 * should another error occur within this routine as a
 * result of the rollback.
    EXEC SQL WHENEVER SQLERROR CONTINUE;
/* Roll back any pending changes and disconnect from Oracle. */
   EXEC SQL ROLLBACK RELEASE;
    exit(1);
```

## 13.8 Using Method 2

What Method 1 does in one step, Method 2 does in two. The dynamic SQL statement, which cannot be a query, is first PREPAREd (named and parsed), then EXECUTEd.

With Method 2, the SQL statement can contain placeholders for input host variables and indicator variables. You can PREPARE the SQL statement once, then EXECUTE it repeatedly using different values of the host variables. Furthermore, you need *not* rePREPARE the SQL statement after a COMMIT or ROLLBACK (unless you log off and reconnect).

You can use EXECUTE for non-queries with Method 4.

The syntax of the PREPARE statement follows:

```
EXEC SQL PREPARE statement_name
   FROM { :host string | string literal };
```

PREPARE parses the SQL statement and gives it a name.

The *statement\_name* is an identifier used by the precompiler, *not* a host or program variable, and should not be declared in the Declare Section. It simply designates the PREPAREd statement you want to EXECUTE.

The syntax of the EXECUTE statement is

```
EXEC SQL EXECUTE statement_name [USING host_variable_list];
```

where host\_variable\_list stands for the following syntax:

```
:host variable1[:indicator1] [, host variable2[:indicator2], ...]
```

EXECUTE executes the parsed SQL statement, using the values supplied for each input host variable.

In the following example, the input SQL statement contains the placeholder *n*:

```
int emp number
                INTEGER;
char delete stmt[120], search cond[40];;
strcpy(delete stmt, "DELETE FROM EMP WHERE EMPNO = :n AND ");
printf("Complete the following statement's search condition--\n");
printf("%s\n", delete stmt);
gets (search cond);
strcat(delete stmt, search cond);
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
for (;;)
    printf("Enter employee number: ");
   gets(temp);
    emp number = atoi(temp);
   if (emp number == 0)
       break;
   EXEC SQL EXECUTE sql_stmt USING :emp_number;
}
```

With Method 2, you must know the datatypes of input host variables at precompile time. In the last example, *emp\_number* was declared as an **int**. It could also have been declared as type **float**, or even a **char**, because Oracle supports all these datatype conversions to the internal Oracle NUMBER datatype.

### 13.8.1 The USING Clause

When the SQL statement is EXECUTEd, input host variables in the USING clause replace corresponding placeholders in the PREPAREd dynamic SQL statement.

Every placeholder in the PREPAREd dynamic SQL statement must correspond to a different host variable in the USING clause. So, if the same placeholder appears two or more times in the PREPAREd statement, each appearance must correspond to a host variable in the USING clause.

The names of the placeholders need not match the names of the host variables. However, the order of the placeholders in the PREPAREd dynamic SQL statement must match the order of corresponding host variables in the USING clause.

If one of the host variables in the USING clause is an array, all must be arrays.

To specify NULLs, you can associate indicator variables with host variables in the USING clause.

#### **Related Topics**

Indicator Variables

### 13.8.2 Example Program: Dynamic SQL Method 2

The following program uses dynamic SQL Method 2 to insert two rows into the EMP table, then delete them. This program is available on-line in your demo directory, in the file sample7.pc.

```
sample7.pc: Dynamic SQL Method 2
* This program uses dynamic SQL Method 2 to insert two rows into
   the EMP table, then delete them.
#include <stdio.h>
#include <string.h>
#define USERNAME "SCOTT"
#define PASSWORD "TIGER"
/* Include the SOL Communications Area, a structure through
* which ORACLE makes runtime status information such as error
* codes, warning flags, and diagnostic text available to the
* program.
*/
#include <sqlca.h>
/* Include the ORACLE Communications Area, a structure through
* which ORACLE makes additional runtime status information
* available to the program.
#include <oraca.h>
/* The ORACA=YES option must be specified to enable use of
* the ORACA.
EXEC ORACLE OPTION (ORACA=YES);
       *username = USERNAME;
char
```



```
*password = PASSWORD;
char
VARCHAR dynstmt[80];
         empno = 1234;
int
         deptno1 = 97;
int.
int
         deptno2 = 99;
/* Handle SQL runtime errors. */
void dyn error();
main()
/\star Call dyn_error() whenever an error occurs
 * processing an embedded SQL statement.
    EXEC SQL WHENEVER SQLERROR DO dyn error ("Oracle error");
/* Save text of current SQL statement in the ORACA if an
 * error occurs.
    oraca.orastxtf = ORASTFERR;
/* Connect to Oracle. */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts("\nConnected to Oracle.\n");
/* Assign a SQL statement to the VARCHAR dynstmt. Both
 * the array and the length parts must be set properly.
 * Note that the statement contains two host-variable
 * placeholders, v1 and v2, for which actual input
 * host variables must be supplied at EXECUTE time.
    strcpy(dynstmt.arr,
        "INSERT INTO EMP (EMPNO, DEPTNO) VALUES (:v1, :v2)");
    dynstmt.len = strlen(dynstmt.arr);
/\!\!\!\!\!^\star Display the SQL statement and its current input host
 * variables.
 */
    puts((char *) dynstmt.arr);
              v1 = %d, v2 = %d\n", empno, deptno1);
    printf("
/* The PREPARE statement associates a statement name with
 * a string containing a SQL statement. The statement name
 * is a SQL identifier, not a host variable, and therefore
 * does not appear in the Declare Section.
 ^{\star} A single statement name can be PREPAREd more than once,
 * optionally FROM a different string variable.
    EXEC SQL PREPARE S FROM : dynstmt;
/* The EXECUTE statement executes a PREPAREd SQL statement
 * USING the specified input host variables, which are
 * substituted positionally for placeholders in the
 * PREPAREd statement. For each occurrence of a
 ^{\star} placeholder in the statement there must be a variable
 ^{\star} in the USING clause. That is, if a placeholder occurs
 * multiple times in the statement, the corresponding
 * variable must appear multiple times in the USING clause.
```

```
* The USING clause can be omitted only if the statement
 * contains no placeholders.
 ^{\star} A single PREPAREd statement can be EXECUTEd more
 * than once, optionally USING different input host
 * variables.
    EXEC SQL EXECUTE S USING :empno, :deptno1;
/* Increment empno and display new input host variables. */
    empno++;
    printf("
             v1 = %d, v2 = %d\n", empno, deptno2);
/* ReEXECUTE S to insert the new value of empno and a
 * different input host variable, deptno2.
 * A rePREPARE is unnecessary.
    EXEC SQL EXECUTE S USING :empno, :deptno2;
/* Assign a new value to dynstmt. */
    strcpy(dynstmt.arr,
        "DELETE FROM EMP WHERE DEPTNO = :v1 OR DEPTNO = :v2");
    dynstmt.len = strlen(dynstmt.arr);
/* Display the new SQL statement and its current input host
 * variables.
    puts((char *) dynstmt.arr);
    printf("v1 = %d,
                        v2 = %d\n", deptno1, deptno2);
/* RePREPARE S FROM the new dynstmt. */
    EXEC SQL PREPARE S FROM : dynstmt;
/* EXECUTE the new S to delete the two rows previously
 * inserted.
 */
    EXEC SQL EXECUTE S USING :deptno1, :deptno2;
/* Commit any pending changes and disconnect from Oracle. */
    EXEC SQL COMMIT RELEASE;
   puts("\nHave a good day!\n");
    exit(0);
}
void
dyn error (msg)
char *msg;
/* This is the ORACLE error handler.
* Print diagnostic text containing error message,
 * current SQL statement, and location of error.
   printf("\n%s", msg);
    printf("\n%.*s\n",
        sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf("in \"%.*s...\"\n",
       oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
```

## 13.9 Using Method 3

Method 3 is similar to Method 2 but combines the PREPARE statement with the statements needed to define and manipulate a cursor. This allows your program to accept and process queries. In fact, if the dynamic SQL statement is a query, you *must* use Method 3 or 4.

For Method 3, the number of columns in the query select list and the number of placeholders for input host variables must be known at precompile time. However, the names of database objects such as tables and columns need not be specified until run time. Names of database objects cannot be host variables. Clauses that limit, group, and sort query results (such as WHERE, GROUP BY, and ORDER BY) can also be specified at run time.

With Method 3, you use the following sequence of embedded SQL statements:

```
PREPARE statement_name FROM { :host_string | string_literal };
DECLARE cursor_name CURSOR FOR statement_name;
OPEN cursor_name [USING host_variable_list];
FETCH cursor_name INTO host_variable_list;
CLOSE cursor_name;
```

Scrollable Cursors can also be used with Method 3. The following sequence of embedded SQL statements must be used for scrollable cursors.

Now we look at what each statement does.

## 13.9.1 PREPARE (Dynamic SQL)

PREPARE parses the dynamic SQL statement and gives it a name. In the following example, PREPARE parses the query stored in the character string *select\_stmt* and gives it the name *sql\_stmt*:

```
char select_stmt[132] =
    "SELECT MGR, JOB FROM EMP WHERE SAL < :salary";
EXEC SQL PREPARE sql stmt FROM :select stmt;</pre>
```

Commonly, the query WHERE clause is input from a terminal at run time or is generated by the application.

The identifier  $sql\_stmt$  is *not* a host or program variable, but must be unique. It designates a particular dynamic SQL statement.

The following statement is correct also:

```
EXEC SQL PREPARE sql stmt FROM SELECT MGR, JOB FROM EMP WHERE SAL < :salary;
```

The following prepare statement, which uses the '%' wildcard, is correct also:

```
EXEC SQL PREPARE S FROM select ename FROM test WHERE ename LIKE 'SMIT%';
```

### 13.9.2 DECLARE (Dynamic SQL)

DECLARE defines a cursor by giving it a name and associating it with a specific query. Continuing our example, DECLARE defines a cursor named *emp\_cursor* and associates it with *sql\_stmt*, as follows:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

The identifiers  $sql\_stmt$  and  $emp\_cursor$  are not host or program variables, but must be unique. If you declare two cursors using the same statement name, the precompiler considers the two cursor names synonymous.

We can define a scrollable cursor named emp\_cursor and associate it with sql\_stmt as follows:

```
EXEC SQL DECLARE emp cursor SCROLL CURSOR FOR sql stmt;
```

### For example, if you execute the statements

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
EXEC SQL DECLARE emp_cursor FOR sql_stmt;
EXEC SQL PREPARE sql_stmt FROM :delete_stmt;
EXEC SQL DECLARE dept cursor FOR sql stmt;
```

when you OPEN *emp\_cursor*, you will process the dynamic SQL statement stored in *delete stmt*, not the one stored in *select stmt*.

## 13.9.3 OPEN (Dynamic SQL)

OPEN allocates an Oracle cursor, binds input host variables, and executes the query, identifying its active set. OPEN also positions the cursor on the first row in the active set and zeroes the rows-processed count kept by the third element of *sqlerrd* in the SQLCA. Input host variables in the USING clause replace corresponding placeholders in the PREPAREd dynamic SQL statement.

In our example, OPEN allocates *emp\_cursor* and assigns the host variable *salary* to the WHERE clause, as follows:

```
EXEC SQL OPEN emp_cursor USING :salary;
```

### 13.9.4 FETCH (Dynamic SQL)

FETCH returns a row from the active set, assigns column values in the select list to corresponding host variables in the INTO clause, and advances the cursor to the next row. If there are no more rows, FETCH returns the "no data found" Oracle error code to sqlca.sqlcode.

In our example, FETCH returns a row from the active set and assigns the values of columns MGR and JOB to host variables  $mgr\_number$  and  $job\_title$ , as follows:

```
EXEC SQL FETCH emp cursor INTO :mgr number, :job title;
```

If the cursor is declared in SCROLL mode, you can then use the various FETCH orientation modes to randomly access the result set.

### 13.9.5 CLOSE (Dynamic SQL)

CLOSE disables the cursor. Once you CLOSE a cursor, you can no longer FETCH from it.

In our example, CLOSE disables *emp\_cursor*, as follows:

```
EXEC SQL CLOSE emp cursor;
```

### 13.9.6 Example Program: Dynamic SQL Method 3

The following program uses dynamic SQL Method 3 to retrieve the names of all employees in a given department from the EMP table. This program is available on-line in your demo directory, in the file sample8.pc

```
sample8.pc: Dynamic SQL Method 3
 * This program uses dynamic SQL Method 3 to retrieve the names
    of all employees in a given department from the EMP table.
#include <stdio.h>
#include <string.h>
#define USERNAME "SCOTT"
#define PASSWORD "TIGER"
/* Include the SQL Communications Area, a structure through
 * which ORACLE makes runtime status information such as error
 ^{\star} codes, warning flags, and diagnostic text available to the
 * program. Also include the ORACA.
#include <sqlca.h>
#include <oraca.h>
/* The ORACA=YES option must be specified to enable use of
 * the ORACA.
EXEC ORACLE OPTION (ORACA=YES);
char
       *username = USERNAME;
char *password = PASSWORD;
VARCHAR dynstmt[80];
VARCHAR ename[10];
       deptno = 10;
void dyn error();
main()
/* Call dyn error() function on any error in
 * an embedded SQL statement.
    EXEC SQL WHENEVER SQLERROR DO dyn error("Oracle error");
```



```
/* Save text of SQL current statement in the ORACA if an
* error occurs.
* /
   oraca.orastxtf = ORASTFERR;
/* Connect to Oracle. */
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    puts("\nConnected to Oracle.\n");
/* Assign a SQL query to the VARCHAR dynstmt. Both the
* array and the length parts must be set properly. Note
* that the query contains one host-variable placeholder,
* v1, for which an actual input host variable must be
* supplied at OPEN time.
    strcpy(dynstmt.arr,
       "SELECT ename FROM emp WHERE deptno = :v1");
    dynstmt.len = strlen(dynstmt.arr);
/* Display the SQL statement and its current input host
* variable.
*/
   puts((char *) dynstmt.arr);
   printf(" v1 = %d\n", deptno);
   printf("\nEmployee\n");
   printf("----\n");
/* The PREPARE statement associates a statement name with
* a string containing a SELECT statement. The statement
* name is a SQL identifier, not a host variable, and
* therefore does not appear in the Declare Section.
* A single statement name can be PREPAREd more than once,
^{\star} optionally FROM a different string variable.
   EXEC SQL PREPARE S FROM : dynstmt;
/* The DECLARE statement associates a cursor with a
* PREPAREd statement. The cursor name, like the statement
* name, does not appear in the Declare Section.
* A single cursor name cannot be DECLAREd more than once.
* /
   EXEC SQL DECLARE C CURSOR FOR S;
/* The OPEN statement evaluates the active set of the
* PREPAREd query USING the specified input host variables,
* which are substituted positionally for placeholders in
* the PREPAREd query. For each occurrence of a
* placeholder in the statement there must be a variable
* in the USING clause. That is, if a placeholder occurs
* multiple times in the statement, the corresponding
* variable must appear multiple times in the USING clause.
* The USING clause can be omitted only if the statement
^{\star} contains no placeholders. OPEN places the cursor at the
^{\star} first row of the active set in preparation for a FETCH.
* A single DECLAREd cursor can be OPENed more than once,
* optionally USING different input host variables.
```

```
EXEC SQL OPEN C USING :deptno;
/* Break the loop when all data have been retrieved. */
    EXEC SQL WHENEVER NOT FOUND DO break;
/* Loop until the NOT FOUND condition is detected. */
    for (;;)
/* The FETCH statement places the select list of the
* current row into the variables specified by the INTO
^{\star} clause, then advances the cursor to the next row. If
* there are more select-list fields than output host
 * variables, the extra fields will not be returned.
 * Specifying more output host variables than select-list
 * fields results in an ORACLE error.
        EXEC SQL FETCH C INTO :ename;
/* Null-terminate the array before output. */
        ename.arr[ename.len] = '\0';
        puts((char *) ename.arr);
        }
/* Print the cumulative number of rows processed by the
 * current SQL statement.
    printf("\nQuery returned %d row%s.\n\n", sqlca.sqlerrd[2],
        (sqlca.sqlerrd[2] == 1) ? "" : "s");
/* The CLOSE statement releases resources associated with
  the cursor.
    EXEC SQL CLOSE C;
/* Commit any pending changes and disconnect from Oracle. */
    EXEC SQL COMMIT RELEASE;
    puts("Sayonara.\n");
    exit(0);
}
void
dyn error (msg)
char *msg;
    printf("\n%s", msg);
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
    oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = '\0';
    oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = '\0';
    printf("\n%s\n", sqlca.sqlerrm.sqlerrmc);
    printf("in \"%s...\"\n", oraca.orastxt.orastxtc);
    printf("on line %d of %s.\n\n", oraca.oraslnr,
       oraca.orasfnm.orasfnmc);
/* Disable ORACLE error checking to avoid an infinite loop
 * should another error occur within this routine.
    EXEC SQL WHENEVER SQLERROR CONTINUE;
/* Release resources associated with the cursor. */
```

```
EXEC SQL CLOSE C;

/* Roll back any pending changes and disconnect from Oracle. */
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

## 13.10 Using Method 4

This section gives an overview of Oracle Dynamic SQL Method 4. Oracle Dynamic SQL Method 4 does not support object types, results sets, arrays of structs, or LOBs.

ANSI SQL does support all datatypes. Use ANSI SQL for all new applications.

There is a kind of dynamic SQL statement that your program cannot process using Method 3. When the number of select-list items or placeholders for input host variables is unknown until run time, your program must use a descriptor. A *descriptor* is an area of memory used by your program and Oracle to hold a complete description of the variables in a dynamic SQL statement.

Recall that for a multirow query, you FETCH selected column values INTO a list of declared output host variables. If the select list is unknown, the host-variable list cannot be established at precompile time by the INTO clause. For example, you know the following query returns two column values:

```
SELECT ename, empno FROM emp WHERE deptno = :dept number;
```

However, if you let the user define the select list, you might not know how many column values the query will return.

### **Related Topics**

- ANSI Dynamic SQL
- Oracle Dynamic SQL: Method 4

### 13.10.1 Need for the SQLDA

To process this kind of dynamic query, your program must issue the DESCRIBE SELECT LIST command and declare a data structure called the SQL Descriptor Area (SQLDA). Because it holds descriptions of columns in the query select list, this structure is also called a *select descriptor*.

Likewise, if a dynamic SQL statement contains an unknown number of placeholders for input host variables, the host-variable list cannot be established at precompile time by the USING clause.

To process the dynamic SQL statement, your program must issue the DESCRIBE BIND VARIABLES command and declare another kind of SQLDA called a *bind descriptor* to hold descriptions of the placeholders for input host variables. (Input host variables are also called *bind variables*.)

If your program has more than one active SQL statement (it might have OPENed two or more cursors, for example), each statement must have its own SQLDA(s). However, non-concurrent cursors can reuse SQLDAs. There is no set limit on the number of SQLDAs in a program.



### 13.10.2 The DESCRIBE Statement

DESCRIBE initializes a descriptor to hold descriptions of select-list items or input host variables.

If you supply a select descriptor, the DESCRIBE SELECT LIST statement examines each select-list item in a PREPAREd dynamic query to determine its name, datatype, constraints, length, scale, and precision. It then stores this information in the select descriptor.

If you supply a bind descriptor, the DESCRIBE BIND VARIABLES statement examines each placeholder in a PREPAREd dynamic SQL statement to determine its name, length, and the datatype of its associated input host variable. It then stores this information in the bind descriptor for your use. For example, you might use placeholder names to prompt the user for the values of input host variables.

### 13.10.3 What is a SQLDA?

A SQLDA is a host-program data structure that holds descriptions of select-list items or input host variables.

SQLDA variables are not defined in the Declare Section.

The select SQLDA contains the following information about a query select list:

- Maximum number of columns that can be DESCRIBEd
- Actual number of columns found by DESCRIBE
- Addresses of buffers to store column values
- Lengths of column values
- Datatypes of column values
- Addresses of indicator-variable values
- Addresses of buffers to store column names
- Sizes of buffers to store column names
- Current lengths of column names

The bind SQLDA contains the following information about the input host variables in a SQL statement:

- Maximum number of placeholders that can be DESCRIBEd
- Actual number of placeholders found by DESCRIBE
- Addresses of input host variables
- Lengths of input host variables
- Datatypes of input host variables
- Addresses of indicator variables
- Addresses of buffers to store placeholder names
- Sizes of buffers to store placeholder names
- Current lengths of placeholder names
- Addresses of buffers to store indicator-variable names
- · Sizes of buffers to store indicator-variable names



Current lengths of indicator-variable names



Oracle Dynamic SQL: Method 4 for information on the SQLDA structure and variable names

## 13.10.4 About Implementing Oracle Method 4

With Oracle Method 4, you generally use the following sequence of embedded SQL statements:

```
EXEC SQL PREPARE statement_name
    FROM { :host_string | string_literal };

EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;

EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
    INTO bind_descriptor_name;

EXEC SQL OPEN cursor_name
    [USING DESCRIPTOR bind_descriptor_name];

EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
    INTO select_descriptor_name;

EXEC SQL FETCH cursor_name
    USING DESCRIPTOR select_descriptor_name;

EXEC SQL CLOSE cursor_name;
```

However, select and bind descriptors need not work in tandem. So, if the number of columns in a query select list is known, but the number of placeholders for input host variables is unknown, you can use the Method 4 OPEN statement with the following Method 3 FETCH statement:

```
EXEC SQL FETCH emp_cursor INTO host_variable_list;
```

Conversely, if the number of placeholders for input host variables is known, but the number of columns in the select list is unknown, you can use the Method 3 OPEN statement

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

with the Method 4 FETCH statement.

EXECUTE can be used for nonqueries with Method 4.

### 13.10.5 Restriction

In Dynamic SQL Method 4, you cannot bind a host array to a PL/SQL procedure with a parameter of type "table."

# 13.11 About Using the DECLARE STATEMENT Statement

With Methods 2, 3, and 4, you might need to use the statement

```
EXEC SQL [AT db name] DECLARE statement name STATEMENT;
```

where *db\_name* and *statement\_name* are identifiers used by the precompiler, *not* host or program variables.

DECLARE STATEMENT declares the name of a dynamic SQL statement so that the statement can be referenced by PREPARE, EXECUTE, DECLARE CURSOR, and DESCRIBE. It is required if you want to execute the dynamic SQL statement at a nondefault database. An example using Method 2 follows:

```
EXEC SQL AT remote_db DECLARE sql_stmt STATEMENT;
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
EXEC SQL EXECUTE sql stmt;
```

In the example, remote\_db tells Oracle where to EXECUTE the SQL statement.

With Methods 3 and 4, DECLARE STATEMENT is also required if the DECLARE CURSOR statement precedes the PREPARE statement, as shown in the following example:

```
EXEC SQL DECLARE sql_stmt STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
EXEC SQL PREPARE sql stmt FROM :dyn string;
```

The usual sequence of statements is

```
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
EXEC SQL DECLARE emp cursor CURSOR FOR sql stmt;
```

### 13.11.1 About Using Host Arrays

The use of host arrays in static SQL and dynamic SQL is similar. For example, to use input host arrays with dynamic SQL Method 2, simply use the syntax

```
EXEC SQL EXECUTE statement_name USING host_array_list;
```

where host\_array\_list contains one or more host arrays.

Similarly, to use input host arrays with Method 3, use the following syntax:

```
OPEN cursor_name USING host_array_list;
```

To use output host arrays with Method 3, use the following syntax:

```
FETCH cursor name INTO host array list;
```

With Method 4, you must use the optional FOR clause to tell Oracle the size of your input or output host array.

### **Related Topics**

Oracle Dynamic SQL: Method 4

## 13.12 About Using PL/SQL

The Pro\*C/C++ Precompiler treats a PL/SQL block like a single SQL statement. So, like a SQL statement, a PL/SQL block can be stored in a string host variable or literal. When you store the PL/SQL block in the string, omit the keywords EXEC SQL EXECUTE, the keyword END-EXEC, and the ';' statement terminator.

However, there are two differences in the way the precompiler handles SQL and PL/SQL:

- The precompiler treats all PL/SQL host variables as input host variables whether they serve as input or output host variables (or both) inside the PL/SQL block.
- You cannot FETCH from a PL/SQL block because it might contain any number of SQL statements.

### 13.12.1 With Method 1

If the PL/SQL block contains no host variables, you can use Method 1 to EXECUTE the PL/SQL string in the usual way.

### 13.12.2 With Method 2

If the PL/SQL block contains a known number of input and output host variables, you can use Method 2 to PREPARE and EXECUTE the PL/SQL string in the usual way.

You must put all host variables in the USING clause. When the PL/SOL string is EXECUTEd. host variables in the USING clause replace corresponding placeholders in the PREPAREd string. Though the precompiler treats all PL/SQL host variables as input host variables, values are assigned correctly. Input (program) values are assigned to input host variables, and output (column) values are assigned to output host variables.

Every placeholder in the PREPAREd PL/SQL string must correspond to a host variable in the USING clause. So, if the same placeholder appears two or more times in the PREPAREd string, each appearance must correspond to a host variable in the USING clause.

### 13.12.3 With Method 3

Methods 2 and 3 are the same except that Method 3 allows FETCHing. Since you cannot FETCH from a PL/SQL block, just use Method 2 instead.

### 13.12.4 With Oracle Method 4

If the PL/SQL block contains an unknown number of input or output host variables, you must use Method 4.

To use Method 4, you set up one bind descriptor for all the input and output host variables. Executing DESCRIBE BIND VARIABLES stores information about input and output host variables in the bind descriptor. Because the precompiler treats all PL/SQL host variables as input host variables, executing DESCRIBE SELECT LIST has no effect.

#### WARNING:

In dynamic SQL Method 4, you cannot bind a host array to a PL/SQL procedure with a parameter of type "table."

### WARNING:

Do not use ANSI-style Comments (- -) in a PL/SQL block that will be processed dynamically because end-of-line characters are ignored. As a result, ANSI-style Comments extend to the end of the block, not just to the end of a line. Instead, use C-style Comments (/\* ... \*/).



### **Related Topics**

Oracle Dynamic SQL: Method 4

## 13.13 Dynamic SQL Statement Caching

Statement caching refers to the feature that provides and manages a cache of statements for each session. In the server, it means that cursors are ready to be used without the statement being parsed again. Statement caching can be enabled in the precompiler applications, which will help in the performance improvement of all applications that rely on the dynamic SQL statements. Performance improvement is achieved by removing the overhead of parsing the dynamic statements on reuse.

You can obtain this performance improvement by using a new command line option, stmt\_cache (for the statement cache size), which will enable the statement caching of the dynamic statements. By enabling the new option, the statement cache will be created at session creation time. The caching is only applicable for the dynamic statements and the cursor cache for the static statements co-exists with this feature.

The command line option <code>stmt\_cache</code> can be given any value in the range of 0 to 65535. Statement caching is disabled by default (value 0). The <code>stmt\_cache</code> option can be set to hold the anticipated number of distinct dynamic SQL statements in the application.

#### Example 13-1 Using the stmt\_cache Option

This example demonstrates the use of the stmt\_cache option. In this program, you insert rows into a table and select the inserted rows by using the cursor in the loop. When the stmt\_cache option is used to precompile this program, the performance increases compared to a normal precompilation.

```
stmtcache.pc
* NOTE:
^{\star} When this program is used to measure the performance with and without
* stmt cache option, do the following changes in the program,
* 1. Increase ROWSCNT to high value, say 10000.
* 2. Remove all the print statements, usually which comsumes significant
      portion of the total program execution time.
* HINT: In Linux, gettimeofday() can be used to measure time.
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>
#include <oraca.h>
#define ROWSCNT 10
char
       *username = "scott";
char *password = "tiger";
/* Function prototypes */
void sql error(char *msg);
void selectdata();
void insertdata();
int main()
```

```
EXEC SQL WHENEVER SQLERROR DO sql error("Oracle error");
  /* Connect using the default schema scott/tiger */
  EXEC SQL CONNECT :username IDENTIFIED BY :password;
  /* core functions to insert and select the data */
  insertdata();
  selectdata();
/* Rollback pll the changes and disconnect from Oracle. */
  EXEC SQL ROLLBACK WORK RELEASE;
  exit(0);
}
/*Insert the data for ROWSCNT items into tpc2sc01 */
void insertdata()
  varchar dynstmt[80];
  int i;
  varchar ename[10];
  float comm;
  char *str;
  /* Allocates temporary buffer */
  str = (char *)malloc (11 * sizeof(char));
  strcpy ((char *)dynstmt.arr,
          "INSERT INTO bonus (ename, comm) VALUES (:ename, :comm)");
  dynstmt.len = strlen(dynstmt.arr);
  EXEC SQL PREPARE S FROM : dynstmt;
  printf ("Inserts %d rows into bonus table using dynamic SQL statement\n",
          ROWSCNT);
  for (i=1; i \le ROWSCNT; i++)
    sprintf (str, "EMP %05d",i);
    strcpy (ename.arr, str);
    comm = i;
    ename.len = strlen (ename.arr);
    EXEC SQL EXECUTE S USING :ename, :comm;
  free(str);
}
/* Select the data using the cursor */
void selectdata()
  varchar dynstmt[80];
  varchar ename[10];
  float comm;
  int i;
  strcpy((char *)dynstmt.arr,
         "SELECT ename, comm FROM bonus WHERE comm = :v1");
  dynstmt.len = (unsigned short)strlen((char *)dynstmt.arr);
  printf ("Fetches the inserted rows using using dynamic SQL statement\n\n");
  printf (" ENAME
                       COMMISSION\n\n");
```

```
for (i=1; i \le ROWSCNT; i++)
    /* Do the prepare in the loop so that the advantage of stmt caching
      is visible*/
   EXEC SQL PREPARE S FROM : dynstmt;
    EXEC SQL DECLARE C CURSOR FOR S;
    EXEC SQL OPEN C USING :i;
    EXEC SQL WHENEVER NOT FOUND DO break;
    /* Loop until the NOT FOUND condition is detected. */
    for (;;)
     EXEC SQL FETCH C INTO :ename, :comm;
     ename.arr[ename.len] = '\0';
     printf ("%10s %7.2f\n", ename.arr, comm);
    /* Close the cursor so that the reparsing is not required for stmt cache */
    EXEC SQL CLOSE C;
 }
}
void sql error(char *msg)
   printf("\n%s", msg);
   sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
   oraca.orastxt.orastxtc[oraca.orastxt.orastxtl] = '\0';
   oraca.orasfnm.orasfnmc[oraca.orasfnm.orasfnml] = '\0';
    printf("\n%s\n", sqlca.sqlerrm.sqlerrmc);
    printf("in \"%s...\"\n", oraca.orastxt.orastxtc);
   printf("on line %d of %s.\n\n", oraca.oraslnr,
       oraca.orasfnm.orasfnmc);
   /* Disable ORACLE error checking to avoid an infinite loop
    ^{\star} should another error occur within this routine.
   EXEC SQL WHENEVER SQLERROR CONTINUE;
    /* Release resources associated with the cursor. */
    EXEC SOL CLOSE C;
    /* Roll back any pending changes and disconnect from Oracle. */
   EXEC SOL ROLLBACK WORK RELEASE;
    exit(1);
```

# 13.14 About Boolean Data Type Support

Using Dynamic SQL, applications can build SQL statements at runtime and take explicit control over data type comversions, With the support for boolean data type, applications can set, get bind, or define BOOLEAN column metadata using these statements:

- DESCRIBE BIND VARIABLES
- DESCRIBE SELECT LIST

#### DESCRIBE BIND VARIABLES

The DESCRIBE BIND VARIABLES statement puts descriptions of bind placeholders into a bind descriptor:

#### DESCRIBE SELECT LIST

If a dynamic SQL statement is a SELECT query, then a DESCRIBE SELECT LIST statement can be used. The DESCRIBE SELECT LIST statement sets the length and data type of each selectlist item:



# **ANSI Dynamic SQL**

This chapter describes Oracle's implementation of ANSI dynamic SQL (also known as SQL standard dynamic SQL) which should be used for new Method 4 applications. It has enhancements over the older Oracle dynamic SQL Method 4, described in the previous chapter.

The ANSI Method 4 supports all Oracle types, while the older Oracle Method 4 does *not* support object types, cursor variables, arrays of structs, DML returning clauses, Unicode variables, and LOBs.

In ANSI dynamic SQL, descriptors are internally maintained by Oracle, while in the older Oracle dynamic SQL Method 4, descriptors are defined in the user's Pro\*C/C++ program. In both cases, Method 4 means that your Pro\*C/C++ program accepts or builds SQL statements that contain a varying number of host variables.

This chapter contains the following topics:

- Basics of ANSI Dynamic SQL
- Overview of ANSI SQL Statements
- Oracle Extensions
- ANSI Dynamic SQL Precompiler Options
- Full Syntax of the Dynamic SQL Statements
- Example Programs

## 14.1 Basics of ANSI Dynamic SQL

Consider the SQL statement:

```
SELECT ename, empno FROM emp WHERE deptno = :deptno data
```

The steps you follow to use ANSI dynamic SQL are:

- Declare variables, including a string to hold the statement to be executed.
- Allocate descriptors for input and output variables.
- Prepare the statement.
- Describe input for the input descriptor.
- Set the input descriptor (in our example the one input host bind variable, deptno data).
- Declare and open a dynamic cursor.
- Set the output descriptors (in our example, the output host variables ename and empno).
- Repeatedly fetch data, using GET DESCRIPTOR to retrieve the ename and empno data fields from each row.
- Do something with the data retrieved (output it, for instance).
- Close the dynamic cursor and deallocate the input and output descriptors.

### 14.1.1 Precompiler Options

Set the micro precompiler option DYNAMIC to ANSI, or set the macro option MODE to ANSI, which causes the default value of DYNAMIC to be ANSI. The other setting of DYNAMIC is ORACLE.

In order to use ANSI type codes, set the precompiler micro option TYPE\_CODE to ANSI, or set the macro option MODE to ANSI which makes the default setting of TYPE\_CODE to ANSI. To set TYPE\_CODE to ANSI, DYNAMIC must also be ANSI.

Oracle's implementation of the ANSI SQL types in Overview of ANSI SQL Statements does not exactly match the ANSI standard. For example, a describe of a column declared as INTEGER will return the code for NUMERIC. As Oracle moves closer to the ANSI standard, small changes in behavior may be required. Use the ANSI types with precompiler option TYPE\_CODE set to ANSI if you want your application to be portable across database platforms and as ANSI compliant as possible. Do not use TYPE\_CODE set to ANSI if such changes are not acceptable.

## 14.2 Overview of ANSI SQL Statements

Allocate a descriptor area first before using it in a dynamic SQL statement.

The ALLOCATE DESCRIPTOR statement syntax is:

```
EXEC SQL ALLOCATE DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
[WITH MAX {:occurrences | numeric literal}];
```

A global descriptor can be used in any module in the program. A local descriptor can be accessed only in the file in which it is allocated. Local is the default.

The descriptor name, desc\_nam, can be a literal in single quotes or a character value stored in a host variable.

occurrences is the maximum number of bind variables or columns that the descriptor can hold. This must be a numeric literal. The default is 100.

When a descriptor is no longer needed, deallocate it to conserve memory. Otherwise, deallocation is done automatically when there are no more active database connections.

The deallocate statement is:

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL] {: desc nam | string literal};
```

Use the DESCRIBE statement to obtain information on a prepared SQL statement. DESCRIBE INPUT describes bind variables for the dynamic statement that has been prepared. DESCRIBE OUTPUT (the default) can give the number, type, and length of the output columns. The simplified syntax is:

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] sql_statement
    USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal};
```

If your SQL statement has input and output values, you must allocate two descriptors: one for input and one for output values. If there are no input values, for example:

```
SELECT ename, empno FROM emp;
```

then the input descriptor is not needed.



Use the SET DESCRIPTOR statement to specify input values for INSERTS, UPDATES, DELETES and the WHERE clauses of SELECT statements. Use SET DESCRIPTOR to set the number of input bind variables (stored in COUNT) when you have not done a DESCRIBE into your input descriptor:

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
COUNT = {:kount | numeric literal};
```

kount can be a host variable or a numeric literal, such as 5. Use a SET DESCRIPTOR statement for each host variable, giving at least the data source of the variable:

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
   VALUE item_number DATA = :hv3;
```

You can also set the type and length of the input host variable:



When TYPE\_CODE=ORACLE, if you do not set TYPE and LENGTH, either explicitly using the SET statement or implicitly by doing a DESCRIBE OUTPUT, the precompiler will use values for them derived from the host variable itself. When TYPE\_CODE=ANSI, you must set TYPE using the values in Table 14-1. You should also set LENGTH because the ANSI default lengths may not match those of your host variables.

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
VALUE item_number TYPE = :hv1, LENGTH = :hv2, DATA = :hv3;
```

We use the identifiers hv1, hv2, and hv3 to remind us that the values must be supplied by host variables. *item number* is the position of the input variable in the SQL statement.

TYPE is the Type Code selected from the following table, if TYPE CODE is set to ANSI:

Table 14-1 ANSI SQL Datatypes

Datatype	Type Code
CHARACTER	1
CHARACTER VARYING	12
DATE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2
REAL	7
SMALLINT	5

DATA is the value of the host variable that is to be input

You can also set other input values such as indicator, precision and scale.

Starting from release 12c, the numeric values in the SET DESCRIPTOR statement must be declared as either int or short int, except for indicator and returned length values which you must declare as int. For 11gR2 or earlier releases, the indicator and returned length values must be declared as short int.

For example, in the following example, when you want to retrieve an empno, set these values: VALUE = 2, because empno is the second output host variable in the dynamic SQL statement. The host variable empno\_typ is set to 3 (Oracle Type for integer). The length of a host integer, empno\_len, is set to 4, which is the size of the host variable. The DATA is equated to the host variable empno\_data which will receive the value from the database table. The code fragment is as follows:

```
char *dyn_statement = "SELECT ename, empno FROM emp
    WHERE deptno = :deptno_number";
int empno_data;
int empno_typ = 3;
int empno_len = 4;
...

EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE = :empno_typ, LENGTH = :empno_len,
    DATA = :empno data;
```

After setting the input values, execute or open your statement using the input descriptor. If there are output values in your statement, set them before doing a FETCH. If you have performed a DESCRIBE OUTPUT, you may have to test the actual type and length of your host variables. The DESCRIBE execution produces internal types and lengths that differ from your host variable external types and length.

After the FETCH of the output descriptor, use GET DESCRIPTOR to access the returned data. Again we show a simplified syntax with details later in this chapter:

```
EXEC SQL GET DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}
VALUE item number :hv1 = DATA, :hv2 = INDICATOR, :hv3 = RETURNED LENGTH;
```

desc\_nam and item\_number can be literals or host variables. A descriptor name can be a literal such as 'out'. An item number can be a numeric literal such as 2.

hv1, hv2, and hv3 are host variables. They must be host variables, not literals. Only three are shown in the example.

Use either **long**, **int** or **short** for all numeric values, except for indicator and returned length variables, which must be short.

### ✓ See Also:

- Table 15-2 for the Oracle type codes
- "SET DESCRIPTOR" for a complete discussion of all the possible descriptor item names
- Table 14-4 for a list of all possible items of returned data that you can get.

### 14.2.1 Example Code

The following example demonstrates the use of ANSI Dynamic SQL. It allocates an input descriptor ('in') and an output descriptor ('out') to execute a SELECT statement. Input values

are set using the SET DESCRIPTOR statement. The cursor is opened and fetched from and the resulting output values are retrieved using a GET DESCRIPTOR statement.

```
char* dyn statement = "SELECT ename, empno FROM emp WHERE deptno = :deptno data" ;
int deptno_type = 3, deptno_len = 2, deptno_data = 10 ;
int ename \overline{\text{type}} = 97, ename \overline{\text{len}} = 30;
char ename data[31] ;
int empno type = 3, empno len = 4;
int empno data ;
int SQLCODE = 0;
main ()
/* Place preliminary code, including connection, here. */
EXEC SQL ALLOCATE DESCRIPTOR 'in' ;
EXEC SQL ALLOCATE DESCRIPTOR 'out';
EXEC SQL PREPARE s FROM : dyn statement ;
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' ;
EXEC SQL SET DESCRIPTOR 'in' VALUE 1 TYPE = :deptno type,
  LENGTH = :deptno len, DATA = :deptno data ;
EXEC SQL DECLARE c CURSOR FOR s ;
EXEC SQL OPEN c USING DESCRIPTOR 'in' ;
EXEC SQL DESCRIBE OUTPUT s USING DESCRIPTOR 'out';
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE = :ename_type,
    LENGTH = :ename_len, DATA = :ename_data ;
EXEC SQL SET DESCRIPTOR 'out' VALUE 2 TYPE = :empno type,
    LENGTH = :empno_len, DATA = :empno_data ;
EXEC SQL WHENEVER NOT FOUND DO BREAK ;
while (SQLCODE == 0)
   EXEC SQL FETCH c INTO DESCRIPTOR 'out';
   EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :ename data = DATA ;
   EXEC SQL GET DESCRIPTOR 'out' VALUE 2 :empno_data = DATA ;
   printf("\nEname = %s Empno = %s", ename data, empno data) ;
EXEC SQL CLOSE c ;
EXEC SQL DEALLOCATE DESCRIPTOR 'in' ;
EXEC SQL DEALLOCATE DESCRIPTOR 'out';
}
```

Scrollable cursors can also be used with ANSI Dynamic SQL. In order to use ANSI dynamic SQL with scrollable cursors, we DECLARE the cursor in SCROLL mode. Use the various fetch orientation modes with the FETCH statement to access the result set.

### 14.3 Oracle Extensions

These extensions are described next:

- Reference semantics for data items in SET statements.
- Arrays for bulk operations.
- Support for object types, NCHAR columns, and LOBs.

### 14.3.1 Reference Semantics

The ANSI standard specifies *value* semantics. To improve performance, Oracle has extended this standard to include *reference* semantics.

Value semantics makes a copy of your host variables data. Reference semantics uses the addresses of your host variables, avoiding a copy. Thus, reference semantics can provide performance improvements for large amounts of data.

To help speed up fetches, use the REF keyword before the data clauses:

```
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE = :ename_type,
   LENGTH = :ename_len, REF DATA = :ename_data ;

EXEC SQL DESCRIPTOR 'out' VALUE 2 TYPE = :empno_type,
   LENGTH = :empno_len, REF DATA = :empno_data ;
```

Then the host variables receive the results of the retrieves. The GET statement is not needed. The retrieved data is written directly into <code>ename data</code> and <code>empno data</code> after each FETCH.

Use of the REF keyword is allowed *only* before DATA, INDICATOR and RETURNED\_LENGTH items (which can vary with each row fetched) as in this fragment of code:

```
int indi, returnLen ;
...
EXEC SQL SET DESCRIPTOR 'out' VALUE 1 TYPE = :ename_type,
    LENGTH = :ename_len, REF DATA = :ename_data,
    REF INDICATOR = :indi, REF RETURNED LENGTH = :returnLen ;
```

After each fetch, returnLen holds the actual retrieved length of the ename field, which is useful for CHAR or VARCHAR2 data.

ename\_len will not receive the returned length. It will not be changed by the FETCH statement. Use a DESCRIBE statement, followed by a GET statement to find out the maximum column width before fetching rows of data.

REF keyword is also used for other types of SQL statements than SELECT, to speed them up. With reference semantics, the host variable is used rather than a value copied into the descriptor area. The host variable data at the time of execution of the SQL statement is used, not its data at the time of the SET. Here is an example:

```
int x = 1 ; 

EXEC SQL SET DESCRIPTOR 'value' VALUE 1 DATA = :x ; 

EXEC SQL SET DESCRIPTOR 'reference' VALUE 1 REF DATA = :x ; 

x = 2 ; 

EXEC SQL EXECUTE s USING DESCRIPTOR 'value' ; /* Will use x = 1 */ 

EXEC SQL EXECUTE s USING DESCRIPTOR 'reference' ; /* Will use x = 2 */
```

### **Related Topics**

SET DESCRIPTOR

### 14.3.2 About Using Arrays for Bulk Operations

Oracle extends ANSI dynamic SQL by providing bulk operations. To use bulk operations, use the FOR clause with an array size to specify the amount of input data or the number of rows you want to process.

The FOR clause is used in the ALLOCATE statement to give the maximum amount of data or number of rows. For example, to use a maximum array size of 100:

```
EXEC SQL FOR 100 ALLOCATE DESCRIPTOR 'out';

Or:
int array_size = 100 ;
...
EXEC SQL FOR :array_size ALLOCATE DESCRIPTOR 'out';
```

The FOR clause is then used in subsequent statements that access the descriptor. In an output descriptor the FETCH statement must have an array size equal to or less than the array size already used in the ALLOCATE statement:

```
EXEC SQL FOR 20 FETCH c1 USING DESCRIPTOR 'out';
```

Subsequent GET statements for the same descriptor, that get DATA, INDICATOR, or RETURNED\_LENGTH values, must use the same array size as the FETCH statement.

```
int val_data[20];
short val_indi[20];
...
EXEC SQL FOR 20 GET DESCRIPTOR 'out' VALUE 1 :val_data = DATA,
    :val_indi = INDICATOR;
```

However, GET statements that reference other items which do not vary from row to row, such as LENGTH, TYPE and COUNT, must *not* use the FOR clause:

```
int cnt, len ;
...
EXEC SQL GET DESCRIPTOR 'out' :cnt = COUNT ;
EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :len = LENGTH ;
```

The same holds true for SET statements with reference semantics. SET statements which precede the FETCH and employ reference semantics for DATA, INDICATOR, or RETURNED LENGTH must have the same array size as the FETCH:

```
int ref_data[20] ;
short ref_indi[20] ;
...
EXEC SQL FOR 20 SET DESCRIPTOR 'out' VALUE 1 REF DATA = :ref_data,
    REF INDICATOR = :ref indi ;
```

Similarly, for a descriptor that is used for input, to insert a batch of rows, for instance, the EXECUTE or OPEN statement must use an array size equal to or less than the size used in the ALLOCATE statement. The SET statement, for both value and reference semantics, that accesses DATA, INDICATOR, or RETURNED\_LENGTH must use the same array size as in the EXECUTE statement.

The FOR clause is never used on the DEALLOCATE or PREPARE statements.

The following code example illustrates a bulk operation with no output descriptor (there is no output, only input to be inserted into the table <code>emp</code>). The value of <code>COUNT</code> is 2 (there are two host variables, <code>ename\_arr</code> and <code>empno\_arr</code>, in the INSERT statement). The data array <code>ename\_arr</code> holds three character strings: "Tom", "Dick" and "Harry", in that order. The indicator array <code>ename\_ind</code> has a value of -1 for the second element; so a NULL will be inserted instead of "Dick". The data array <code>empno\_arr</code> contains three employee numbers. A DML returning clause could be used to confirm the actual names inserted.

```
char* dyn_statement = "INSERT INTO emp (ename) VALUES (:ename_arr)";
char ename_arr[3][6] = {Tom","Dick","Harry"};
short ename ind[3] = {0,-1,0};
```



```
int ename_len = 6, ename_type = 97, cnt = 2;
int empno_arr[3] = {8001, 8002, 8003};
int empno_len = 4;
int empno_type = 3;
int array_size = 3;

EXEC SQL FOR :array_size ALLOCATE DESCRIPTOR 'in';

EXEC SQL SET DESCRIPTOR 'in' COUNT = :cnt;

EXEC SQL SET DESCRIPTOR 'in' VALUE 1 TYPE = :ename_type, LENGTH = :ename_len;

EXEC SQL SET DESCRIPTOR 'in' VALUE 2 TYPE = :empno_type, LENGTH = :empno_len;

EXEC SQL FOR :array_size SET DESCRIPTOR 'in' VALUE 1

DATA = :ename_arr, INDICATOR = :ename_ind;

EXEC SQL FOR :array_size SET DESCRIPTOR 'in' VALUE 2

DATA = :empno_arr;

EXEC SQL PREPARE s FROM :dyn_statement;

EXEC SQL FOR :array_size EXECUTE s USING DESCRIPTOR 'in';
...
```

### The preceding code will insert these values:

```
EMPNO ENAME
8001 Tom
8002
8003 Harry
```

### **Related Topics**

- The DML Returning Clause
- About Using the FOR Clause

### 14.3.3 Support for Arrays of Structs

You must set the HOST\_STRIDE\_LENGTH to the size of the struct, and the INDICATOR\_STRIDE\_LENGTH to the size of the indicator struct, and the RETURNED\_LENGTH\_STRIDE to the size of your returned length struct.

Arrays of structs are supported by ANSI dynamic SQL, but are *not* supported by the older Oracle dynamic SQL.

## 14.3.4 Support for Object Types

For the object types that you have defined yourself, use Oracle TYPE equal to 108. For an object type column, use a DESCRIBE statement to obtain USER\_DEFINED\_TYPE\_VERSION, USER\_DEFINED\_TYPE\_NAME, USER\_DEFINED\_TYPE\_NAME\_LENGTH, USER\_DEFINED\_TYPE\_SCHEMA, and USER\_DEFINED\_TYPE\_SCHEMA\_LENGTH.

If you do not employ the DESCRIBE statement to retrieve these values, you have to set them yourself through the SET DESCRIPTOR statement.

## 14.4 ANSI Dynamic SQL Precompiler Options

The macro option MODE sets ANSI compatibility characteristics and controls a number of functions. It can have the values ANSI or ORACLE. For individual functions there are micro options that override the MODE setting.

The precompiler micro option *DYNAMIC* specifies the descriptor behavior in dynamic SQL. The precompiler micro option *TYPE\_CODE* specifies whether ANSI or Oracle datatype codes are to be used.

When the macro option MODE is set to ANSI, the micro option DYNAMIC becomes ANSI automatically. When MODE is set to ORACLE, DYNAMIC becomes ORACLE.

DYNAMIC and TYPE\_CODE cannot be used inline.

This table describes functionality and how the DYNAMIC setting affects them.

Table 14-2 DYNAMIC Option Settings

Function	DYNAMIC = ANSI	DYNAMIC = ORACLE
Descriptor creation.	Must use ALLOCATE statement.	Must use function SQLSQLDAAlloc().
Descriptor destruction.	May use DEALLOCATE statement.	May use function SQLLDAFree().
Retrieving data.	May use both FETCH and GET statements.	Must use only FETCH statement.
Setting input data.	May use DESCRIBE INPUT statement. Must use SET statement.	Must set descriptor values in code. Must use DESCRIBE BIND VARIABLES statement.
Descriptor representation.	Single quoted literal or host identifier which contains the descriptor name.	Host variable, a pointer to SQLDA.
Data types available.	All ANSI types except BIT and all Oracle types.	Oracle types except objects, LOBs, arrays of structs and cursor variables.

The micro option TYPE\_CODE is set by the precompiler to the same setting as the macro option MODE. TYPE\_CODE can only equal ANSI if DYNAMIC equals ANSI.

Here is the functionality corresponding to the TYPE CODE settings:

Table 14-3 TYPE\_CODE Option Settings

Function	TYPE_CODE = ANSI	TYPE_CODE = ORACLE
Data type code numbers input and returned in dynamic SQL.	Use ANSI code numbers when ANSI type exists. Otherwise, use the negative of the Oracle code number. Only valid when DYNAMIC = ANSI.	Use Oracle code numbers.  May be used regardless of the setting of DYNAMIC.

### **Related Topics**

- MODE
- New Names for SQLLIB Public Functions

## 14.5 Full Syntax of the Dynamic SQL Statements

See Embedded SQL Statements and Directives for more details on all these statements.

### 14.5.1 ALLOCATE DESCRIPTOR

### **Purpose**

Use this statement to allocate a SQL descriptor area. Supply a descriptor and the maximum number of occurrences of host bind items, and an array size. This statement is only for the ANSI dynamic SQL.

### **Syntax**

```
EXEC SQL [FOR [:]array_size] ALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
{:desc_nam | string_literal} [WITH MAX occurrences];
```

#### **Variables**

array size

This is in an optional clause (it is an Oracle extension) that supports array processing. It tells the precompiler that the descriptor is usable for array processing.

```
GLOBAL | LOCAL
```

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

```
desc nam
```

Descriptor name. Local descriptors must be unique in the module. A runtime error is generated if the descriptor has been allocated, but not deallocated, previously. A global descriptor must be unique for the application, or a runtime error results.

#### occurrences

The maximum number of host variables possible in the descriptor. It must be an integer constant between 0 and 64K, or an error is returned. Default is 100. The clause is optional. A precompiler error results if it does not conform to these rules.

#### **Examples**

```
EXEC SQL ALLOCATE DESCRIPTOR 'SELDES' WITH MAX 50 ;

EXEC SQL FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25 ;
```

### 14.5.2 DEALLOCATE DESCRIPTOR

#### **Purpose**

Use this statement to deallocate a SQL descriptor area that has been previously allocated, to free memory. This statement is only used for the ANSI dynamic SQL.

#### **Syntax**

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL] {: desc nam | string literal} ;
```

### Variable

### GLOBAL | LOCAL

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

```
desc_nam
```

A runtime error results when a descriptor with the same name and scope has not been allocated, or has been allocated and deallocated already.



### **Examples**

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES';

EXEC SQL DEALLOCATE DESCRIPTOR :binddes;
```

### 14.5.3 GET DESCRIPTOR

#### **Purpose**

Use to obtain information from a SQL descriptor area.

#### **Syntax**

#### **Variables**

array size

The FOR array\_size is an optional Oracle extension. array\_size has to be equal to the field array size in the FETCH statement.

#### COUNT

The total number of bind variables.

desc\_nam

Descriptor name.

```
GLOBAL | LOCAL
```

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

```
VALUE item_number
```

The position of the item in the SQL statement. item\_number can be a variable or a constant. If item\_number is greater than COUNT, the "no data found" condition is returned. item\_number must be greater than 0.

```
hv1 .. hvN
```

These are host variables to which values are transferred.

```
item_name1 .. item_nameN
```

The descriptor item names corresponding to the host variables. The possible ANSI descriptor item names are:



Table 14-4 Definitions of Descriptor Item Names for GET DESCRIPTOR

Descriptor Item Name	Meaning
TYPE	Use the negative value of Oracle type code if the ANSI datatype is not in the table and TYPE_CODE=ANSI.
LENGTH	Length of data in the column: in characters for NCHAR; in bytes otherwise. Set by the DESCRIBE OUTPUT.
OCTET_LENGTH	Length of data in bytes.
RETURNED_LENGTH	The actual data length after a FETCH.
RETURNED_OCTET_LENGTH	Length of the returned data in bytes.
PRECISION	The number of digits.
SCALE	For exact numeric types, the number of digits to the right of the decimal point.
NULLABLE	If 1, the column can have NULL values. If 0,the column cannot have NULL values.
INDICATOR	The associated indicator value.
DATA	The data value.
NAME	Column name.
CHARACTER_SET_NAME	Column's character set.

The Oracle additional descriptor item names are:

Table 14-5 Oracle Extensions to Definitions of Descriptor Item Names for GET DESCRIPTOR

Descriptor Item Name	Meaning
NATIONAL_CHARACTER	If 2, NCHAR or NVARCHAR2. If 1, character. If 0, non-character.
INTERNAL_LENGTH	The internal length, in bytes.
HOST_STRIDE_LENGTH	The size of the host struct in bytes.
INDICATOR_STRIDE_LENGTH	The size of the indicator struct in bytes.
RETURNED_LENGTH_STRIDE	The size of the returned-length struct in bytes.
USER_DEFINED_TYPE_VERSION	Used for character representation of object type version.
USER_DEFINED_TYPE_NAME	Name of object type.
USER_DEFINED_TYPE_NAME_L ENGTH	Length of name of object type.
USER_DEFINED_TYPE_SCHEMA	Used for character representation of the object's schema.
USER_DEFINED_TYPE_SCHEMA _LENGTH	Length of USER_DEFINED_TYPE_SCHEMA.
NATIONAL_CHARACTER	If 2, NCHAR or NVARCHAR2. If 1, character. If 0, non-character.

### **Usage Notes**

Use the FOR clause in GET DESCRIPTOR statements which contain DATA, INDICATOR, and RETURNED\_LENGTH items only.

The internal type is provided by the DESCRIBE OUTPUT statement. For both input and output, you must set the type to be the external type of your host variable.

TYPE is the ANSI SQL Datatype code. Use the negative value of the Oracle type code if the ANSI type is not in the table.

LENGTH contains the column length in characters for fields that have fixed-width National Character Sets. It is in bytes for other character columns. It is set in DESCRIBE OUTPUT.

RETURNED\_LENGTH is the actual data length set by the FETCH statement. It is in bytes or characters as described for LENGTH. The fields OCTET\_LENGTH and RETURNED\_OCTET\_LENGTH are the lengths in bytes.

NULLABLE = 1 means that the column can have NULLS; NULLABLE = 0 means it cannot.

CHARACTER\_SET\_NAME only has meaning for character columns. For other types, it is undefined. The DESCRIBE OUTPUT statement obtains the value.

DATA and INDICATOR are the data value and the indicator status for that column. If data = NULL, but the indicator was not requested, an error is generated at runtime ("DATA EXCEPTION, NULL VALUE, NO INDICATOR PARAMETER").

#### **Oracle-Specific Descriptor Item Names**

NATIONAL\_CHARACTER = 2 if the column is an NCHAR or NVARCHAR2 column. If the column is a character (but not National Character) column, this item is set to 1. If a non-character column, this item becomes 0 after DESCRIBE OUTPUT is executed.

INTERNAL\_LENGTH is for compatibility with Oracle dynamic Method 4. It has the same value as the length member of the Oracle SQL descriptor area.

The following three items are not returned by a DESCRIBE OUTPUT statement.

- HOST STRIDE LENGTH is the size of the struct of host variables.
- INDICATOR STRIDE LENGTH is the size of the struct of indicator variables.
- RETURNED\_LENGTH\_STRIDE is the size of the struct of returned-length variables

The following items apply only to object types when the precompiler option OBJECTS has been set to YES.

- USER\_DEFINED\_TYPE\_VERSION contains the character representation of the type version.
- USER DEFINED TYPE NAME is the character representation of the name of the type.
- USER\_DEFINED\_TYPE\_NAME\_LENGTH is the length of the type name in bytes.
- USER\_DEFINED\_TYPE\_SCHEMA is the character representation of the schema name of the type.
- USER\_DEFINED\_TYPE\_SCHEMA\_LENGTH is the length in characters of the type's schema name.

#### **Examples**

```
EXEC SQL GET DESCRIPTOR :binddes :n = COUNT ;

EXEC SQL GET DESCRIPTOR 'SELDES' VALUE 1 :t = TYPE, :l = LENGTH ;

EXEC SQL FOR :batch GET DESCRIPTOR LOCAL 'SELDES'
   VALUE :sel item no :i = INDICATOR, :v = DATA ;
```



### See Also:

- Table 14-1 for the ANSI type codes
- "Table 15-2" for the Oracle type codes
- Oracle Dynamic SQL: Method 4

### 14.5.4 SET DESCRIPTOR

### **Purpose**

Use this statement to set information in the descriptor area from host variables. The SET DESCRIPTOR statement supports *only* host variables for the item names.

### **Syntax**

```
EXEC SQL [FOR array_size] SET DESCRIPTOR [GLOBAL | LOCAL]
  {:desc_nam | string_literal} {COUNT = :hv0 |
   VALUE item_number
  [REF] item_name1 = :hv1
  [{, [REF] item_nameN = :hvN}]};
```

#### **Variables**

array\_size

This optional Oracle clause permits using arrays when setting the descriptor items DATA, INDICATOR, and RETURNED\_LENGTH only. You cannot use other items in a SET DESCRIPTOR that contains the FOR clause. All host variable array sizes must match. Use the same array size for the SET statement that you use for the FETCH statement.

### GLOBAL | LOCAL

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

desc\_nam

The descriptor name. It follows the rules in ALLOCATE DESCRIPTOR.

#### COUNT

The number of bind (input) or define (output) variables.

VALUE item\_number

Position in the dynamic SQL statement of a host variable.

hv1 .. hvN

The host variables (not constants) that you set.

item\_nameI

In a similar way to the GET DESCRIPTOR syntax desc item name can take on these values.

Table 14-6 Descriptor Item Names for SET DESCRIPTOR

Descriptor Item Name	Meaning
TYPE	Use negative value of the Oracle type if there is no corresponding ANSI type.
LENGTH	Maximum length of data in the column.
INDICATOR	The associated indicator value. Set for reference semantics.
DATA	Value of the data to be set. Set for reference semantics.
CHARACTER_SET_NAME	Column's character set.
TYPE	Use negative value of the Oracle type if there is no corresponding ANSI type.

See Table 14-1 for the ANSI type codes and see "Table 15-2" for the Oracle type codes.

The Oracle extensions to the descriptor item names are:

Table 14-7 Oracle Extensions to Descriptor Item Names for SET DESCRIPTOR

Descriptor Item Name	Meaning
RETURNED_LENGTH	Length returned after a FETCH. Set if reference semantics is being used.
NATIONAL_CHARACTER	Set to 2 when the input host variable is an NCHAR or NVARCHAR2 type.
	Set to 0 when the National Character setting is clear.
HOST_STRIDE_LENGTH	Size of the host variable struct in bytes.
INDICATOR_STRIDE_LENGTH	Size of the indicator variable in bytes.
RETURNED_LENGTH_STRIDE	Size of the returned-length struct in bytes.
USER_DEFINED_TYPE_NAME	Name of object type.
USER_DEFINED_TYPE_NAME_L ENGTH	Length of name of object type.
USER_DEFINED_TYPE_SCHEMA	Used for character representation of the object's schema.
USER_DEFINED_TYPE_SCHEMA _LENGTH	Length of USER_DEFINED_TYPE_SCHEMA.

### **Usage Notes**

Reference semantics is another optional Oracle extension that speeds performance. Use the keyword REF before these descriptor items names only: DATA, INDICATOR, RETURNED\_LENGTH. When you use the REF keyword you do not need to use a GET statement. Complex data types (object and collection types, arrays of structs, and the DML returning clause) all require the REF form of SET DESCRIPTOR.

If the program reuses DESCRIPTOR for another SQL, the old values of DESCRIPTOR remain.

When REF is used the associated host variable itself is used in the SET. The GET is not needed in this case. The RETURNED\_LENGTH can only be set when you use the REF semantics, not the value semantics.

Use the same array size for the SET or GET statements that you use in the FETCH.

Set the NATIONAL CHAR field to 2 for NCHAR host input values.

Set the NATIONAL\_CHARACTER field to 0 when DESCRIPTOR is used for NCHAR host input values in the old SQL.

When setting an object type's characteristics, you must set USER\_DEFINED\_TYPE\_NAME and USER\_DEFINED\_TYPE\_NAME\_LENGTH.

If omitted, USER\_DEFINED\_TYPE\_SCHEMA and USER\_DEFINED\_TYPE\_SCHEMA\_LENGTH default to the current connection.

Set CHARACTER\_SET\_NAME to UTF16 for client-side Unicode support. The data will be in UCS2 encoding and the RETURNED\_LENGTH is in CHARS.

#### **Example**

```
int bindno = 2 ;
short indi = -1 ;
char data = "ignore" ;
int batch = 1 ;

EXEC SQL FOR :batch ALLOCATE DESCRIPTOR 'binddes' ;
EXEC SQL SET DESCRIPTOR GLOBAL :binddes COUNT = 3 ;
EXEC SQL FOR :batch SET DESCRIPTOR :bindes
    VALUE :bindno INDICATOR = :indi, DATA = :data ;
...
```

#### **Related Topics**

- GET DESCRIPTOR
- The DML Returning Clause
- About Using Arrays for Bulk Operations



### 14.5.5 Use of PREPARE

#### **Purpose**

The PREPARE statement used in this method is the same as the PREPARE statement used in the other dynamic SQL methods. An Oracle extension allows a quoted string for the SQL statement, as well as a variable.

### **Syntax**

```
EXEC SQL PREPARE statement id FROM :sql statement ;
```

#### **Variables**

statement id

This must not be declared; it is a undeclared SQL identifier.

sql statement

A character string (a constant or a variable) holding the embedded SQL statement.

### **Example**

```
char* statement = "SELECT ENAME FROM emp WHERE deptno = :d" ;
EXEC SQL PREPARE S1 FROM :statement ;
```

### 14.5.6 DESCRIBE INPUT

#### **Purpose**

This statement returns information about the bind variables.

### **Syntax**

```
EXEC SQL DESCRIBE INPUT statement_id USING [SQL] DESCRIPTOR
[GLOBAL | LOCAL] {:desc nam | string literal};
```

#### **Variables**

statement\_id

The same as used in PREPARE and DESCRIBE OUTPUT. This must not be declared; it is an undeclared SQL identifier.

```
GLOBAL | LOCAL
```

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

desc\_nam

The descriptor name.

#### **Usage Notes**

DESCRIBE INPUT only sets COUNT and NAME items.

### **Examples**

```
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR GLOBAL :binddes ; EXEC SQL DESCRIBE INPUT S2 USING DESCRIPTOR 'input' ;
```

### 14.5.7 DESCRIBE OUTPUT

#### **Purpose**

Use this statement to obtain information about the output columns in a PREPAREd statement. The ANSI syntax differs from the older Oracle syntax. The information which is stored in the SQL descriptor area is the number of values returned and associated information such as type, length, and name.

### **Syntax**

```
EXEC SQL DESCRIBE [OUTPUT] statement_id USING [SQL] DESCRIPTOR
[GLOBAL | LOCAL] {:desc_nam | string_literal};
```

#### **Variables**

statement id



The same as used in PREPARE. This must not be declared; it is an undeclared SQL identifier.

```
GLOBAL | LOCAL
```

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

desc nam

The descriptor name.

OUTPUT is the default and can be omitted.

#### **Examples**

```
char* desname = "SELDES";
EXEC SQL DESCRIBE S1 USING SQL DESCRIPTOR 'SELDES'; /* Or, */
EXEC SQL DESCRIBE OUTPUT S1 USING DESCRIPTOR :desname;
```

### **14.5.8 EXECUTE**

#### **Purpose**

EXECUTE matches input and output variables in a prepared SQL statement and then executes the statement. This ANSI version of EXECUTE differs from the older EXECUTE statement by allowing two descriptors in one statement to support DML returning clause.

#### **Syntax**

```
EXEC SQL [FOR :array_size] EXECUTE statement_id
[USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}]
[INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam | string_literal}];
```

#### **Variables**

array\_size

The number of rows the statement will process.

statement id

The same as used in PREPARE. This must not be declared; it is an undeclared SQL identifier. It can be a literal.

GLOBAL | LOCAL

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

desc\_nam

The descriptor name.

#### **Usage Notes**

The INTO clause implements the DML returning clause for INSERT, UPDATE and DELETE.

#### **Examples**

```
EXEC SQL EXECUTE S1 USING SQL DESCRIPTOR GLOBAL :binddes;

EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES';
```

#### **Related Topics**

The DML Returning Clause

### 14.5.9 Use of EXECUTE IMMEDIATE

#### **Purpose**

Executes a literal or host variable character string containing the SQL statement. The ANSI SQL form of this statement is the same as in the older Oracle dynamic SQL:

#### **Syntax**

```
EXEC SQL EXECUTE IMMEDIATE {:sql statement | string literal}
```

#### **Variable**

sql\_statement

The SQL statement or PL/SQL block in a character string.

#### **Example**

```
EXEC SQL EXECUTE IMMEDIATE :statement ;
```

### 14.5.10 Use of DYNAMIC DECLARE CURSOR

#### **Purpose**

Declares a cursor that is associated with a statement which is a query. This is a form of the generic Declare Cursor statement.

#### **Syntax**

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_id;
```

#### **Variables**

cursor\_name

A cursor variable (a SQL identifier, not a host variable).

statement\_id

An undeclared SQL identifier.

#### Example

```
EXEC SQL DECLARE C1 CURSOR FOR S1 ;
```



### 14.5.11 OPEN Cursor

#### **Purpose**

The OPEN statement associates input parameters with a cursor and then opens the cursor.

#### **Syntax**

```
EXEC SQL [FOR :array_size] OPEN dyn_cursor
[[USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam1 | string_literal}]
[INTO [SQL] DESCRIPTOR [GLOBAL | LOCAL] {:desc_nam2 | string_literal}]];
```

#### **Variables**

array\_size

This limit is less than or equal to number specified when the descriptor was allocated.

dyn\_cursor

The cursor variable.

GLOBAL | LOCAL

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

desc\_nam

The descriptor name.

### **Usage Notes**

If the prepared statement associated with the cursor contains colons or question marks, a USING clause must be specified, or an error results at runtime. The DML returning clause is supported.

#### **Examples**

```
EXEC SQL OPEN C1 USING SQL DESCRIPTOR :binddes ;

EXEC SQL FOR :limit OPEN C2 USING DESCRIPTOR :b1, :b2
   INTO SQL DESCRIPTOR :seldes ;
```

#### **Related Topics**

The DML Returning Clause

### 14.5.12 FETCH

#### **Purpose**

Fetches a row for a cursor declared with a dynamic DECLARE statement.

#### **Syntax**

```
EXEC SQL [FOR :array_size] FETCH cursor INTO [SQL] DESCRIPTOR
[GLOBAL | LOCAL] {:desc_nam | string_literal};
```



#### **Variables**

array size

The number of rows the statement will process.

cursor

The dynamic cursor that was previously declared.

```
GLOBAL | LOCAL
```

The optional scope clause defaults to LOCAL if not entered. A local descriptor can be accessed only in the file in which it is allocated. A global descriptor can be used in any module in the compilation unit.

desc\_nam

Descriptor name.

#### **Usage Notes**

The optional array\_size in the FOR clause must be less than or equal to the number specified in the ALLOCATE DESCRIPTOR statement.

#### **Examples**

```
EXEC SQL FETCH FROM C1 INTO DESCRIPTOR 'SELDES';

EXEC SQL FOR :arsz FETCH C2 INTO DESCRIPTOR :desc;
```

## 14.5.13 CLOSE a Dynamic Cursor

#### **Purpose**

Closes a dynamic cursor. Syntax has not changed from the older Oracle Method 4:

#### **Syntax**

```
EXEC SQL CLOSE cursor ;
```

#### Variable

cursor

The dynamic cursor that was previously declared.

#### **Example**

```
EXEC SQL CLOSE C1 ;
```

## 14.5.14 Differences From Oracle Dynamic Method 4

The ANSI dynamic SQL interface supports all the datatypes supported by the Oracle dynamic Method 4, with these additions:

 All datatypes, including object types, result sets, and LOB types are supported by ANSI Dynamic SQL.

- The ANSI mode uses an internal SQL descriptor area which is an expansion of the external SQLDA used in Oracle older dynamic Method 4 to store its input and output information.
- New embedded SQL statements are introduced: ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, DESCRIBE, GET DESCRIPTOR, and SET DESCRIPTOR.
- The DESCRIBE statement does not return the names of indicator variables in ANSI Dynamic SQL.
- ANSI Dynamic SQL does not allow you to specify the maximum size of the returned column name or expression. The default size is set at 128.
- The descriptor name must be either an identifier in single-quotes or a host variable preceded by a colon.
- For output, the optional SELECT LIST FOR clause in the DESCRIBE is replaced by the optional keyword OUTPUT. The INTO clause is replaced by the USING DESCRIPTOR clause, which can contain the optional keyword SQL.
- For input, the optional BIND VARIABLES FOR clause of the DESCRIBE can be replaced by the keyword INPUT. The INTO clause is replaced by the USING DESCRIPTOR clause, which can contain the optional keyword SOL.
- The optional keyword SQL can come before the keyword DESCRIPTOR in the USING clause of the EXECUTE, FETCH and OPEN statements.

## 14.5.15 Restrictions (ANSI Dynamic SQL)

Restrictions in effect on ANSI dynamic SQL are:

- You cannot mix ANSI and Oracle dynamic SQL methods in the same module.
- The precompiler option DYNAMIC must be set to ANSI. The precompiler option TYPE\_CODE can be set to ANSI only if DYNAMIC is set to ANSI.
- The SET statement supports only host variables as item names.

# 14.6 Example Programs

The following two programs are in the demo directory.

### 14.6.1 ansidyn1.pc

This program demonstrates using ANSI Dynamic SQL to process SQL statements which are not known until runtime. It is intended to demonstrate the simplest (though not the most efficient) approach to using ANSI Dynamic SQL. It uses ANSI compatible value semantics and ANSI type codes. ANSI SQLSTATE is used for error numbers. Descriptor names are literals. All input and output is through ANSI varying character type.

The program connects you to ORACLE using your username and password, then prompts you for a SQL statement. Enter legal SQL or PL/SQL statements using regular, not embedded, SQL syntax and terminate each statement with a semicolon. Your statement will be processed. If it is a query, the fetched rows are displayed.

You can enter multiline statements. The limit is 1023 characters. There is a limit on the size of the variables, MAX\_VAR\_LEN, defined as 255. This program processes up to 40 bind variables and 40 select-list items. DML returning clauses and user defined types are not supported with value semantics.



#### Precompile the program with mode = ansi, for example:

```
proc mode=ansi ansidyn1
```

```
Using mode=ansi will set dynamic and type code to ansi.
```

This program demonstates using ANSI Dynamic SQL to process SQL statements which are not known until runtime. It is intended to demonstrate the simplest (though not the most efficient) approach to using ANSI Dynamic SQL. It uses ANSI compatible value semantics and ANSI type codes. ANSI Sqlstate is used for error numbers. Descriptor names are literals. All input and output is through ANSI the varying character type.

The program connects you to ORACLE using your username and password, then prompts you for a SQL statement. Enter legal SQL or PL/SQL statements using regular, not embedded, SQL syntax and terminate each statement with a seimcolon. Your statement will be processed. If it is a query, the fetched rows are displayed.

You can enter multiline statements. The limit is 1023 characters. There is a limit on the size of the variables, MAX\_VAR\_LEN, defined as 255. This program processes up to 40 bind variables and 40 select-list items. DML returning statments and user defined types are not supported with value semantics.

Precompile the program with mode=ansi, for example:

```
proc mode=ansi ansidyn1
```

#include <stdio.h>

Using mode=ansi will set dynamic and type code to ansi.

```
#include <string.h>
#include <setjmp.h>
#include <stdlib.h>
#include <sqlcpr.h>
#define MAX OCCURENCES 40
#define MAX VAR LEN
#define MAX_NAME LEN 31
#ifndef NULL
#define NULL 0
#endif
/* Prototypes */
#if defined( STDC
 void sql error(void);
 int oracle connect(void);
 int get dyn statement(void);
 int process input (void);
 int process output (void);
  void help(void);
#else
```



```
void sql error(/* void */);
 int oracle connect(/* void */);
 int get_dyn_statement(/* void _*/);
 int process_input(/*_ void _*/);
int process_output(/*_ void _*/);
 void help(/*_ void _*/);
#endif
EXEC SQL INCLUDE sqlca;
char SQLSTATE[6];
/* global variables */
EXEC SQL BEGIN DECLARE SECTION;
char
       dyn statement[1024];
char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
/* Define a buffer to hold longjmp state info. */
jmp_buf jmp_continue;
/* A global flag for the error routine. */
int parse flag = 0;
/* A global flag to indicate statement is a select */
int select found;
void main()
    /* Connect to the database. */
    if (oracle_connect() != 0)
        exit(1);
    EXEC SQL WHENEVER SQLERROR DO sql error();
    /* Allocate the input and output descriptors. */
    EXEC SQL ALLOCATE DESCRIPTOR 'input descriptor';
    EXEC SQL ALLOCATE DESCRIPTOR 'output descriptor';
    /* Process SQL statements. */
    for (;;)
        (void) setjmp(jmp continue);
        /* Get the statement. Break on "exit". */
        if (get dyn statement() != 0)
            break;
        /\star Prepare the statement and declare a cursor. \star/
        parse flag = 1;  /* Set a flag for sql error(). */
        EXEC SQL PREPARE S FROM : dyn statement;
        parse flag = 0;
                           /* Unset the flag. */
        EXEC SQL DECLARE C CURSOR FOR S;
        /* Call the function that processes the input. */
        if (process_input())
            exit(1);
```

```
/\star Open the cursor and execute the statement. \star/
        EXEC SQL OPEN C USING DESCRIPTOR 'input descriptor';
        /* Call the function that processes the output. */
        if (process_output())
            exit(1);
        /* Close the cursor. */
        EXEC SQL CLOSE C;
    } /* end of for(;;) statement-processing loop */
    /* Deallocate the descriptors */
   EXEC SQL DEALLOCATE DESCRIPTOR 'input descriptor';
   EXEC SQL DEALLOCATE DESCRIPTOR 'output descriptor';
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL COMMIT WORK;
   puts("\nHave a good day!\n");
   EXEC SQL WHENEVER SQLERROR DO sql error();
   return;
}
int get dyn statement()
   char *cp, linebuf[256];
   int iter, plsql;
    for (plsql = 0, iter = 1; ;)
        if (iter == 1)
        {
            printf("\nSQL> ");
            dyn_statement[0] = '\0';
            select found = 0;
        fgets(linebuf, sizeof linebuf, stdin);
        cp = strrchr(linebuf, '\n');
        if (cp && cp != linebuf)
            *cp = ' ';
        else if (cp == linebuf)
           continue;
        if ((strncmp(linebuf, "SELECT", 6) == 0) ||
            (strncmp(linebuf, "select", 6) == 0))
        {
            select found=1;;
        }
        if ((strncmp(linebuf, "EXIT", 4) == 0) \mid \mid
            (strncmp(linebuf, "exit", 4) == 0))
        {
            return -1;
        }
        else if (linebuf[0] == '?' ||
```

```
(strncmp(linebuf, "HELP", 4) == 0) ||
            (strncmp(linebuf, "help", 4) == 0))
        {
           help();
           iter = 1;
            continue;
       if (strstr(linebuf, "BEGIN") ||
            (strstr(linebuf, "begin")))
           plsql = 1;
       strcat(dyn_statement, linebuf);
       if ((plsql && (cp = strrchr(dyn statement, '/'))) ||
            (!plsql && (cp = strrchr(dyn statement, ';'))))
           *cp = ' \ 0';
           break;
       }
       else
           iter++;
           printf("%3d ", iter);
    return 0;
int process input()
   int i;
   EXEC SQL BEGIN DECLARE SECTION;
     char name[31];
     int input_count, input_len, occurs, ANSI_varchar_type;
     char input buf[MAX VAR LEN];
   EXEC SQL END DECLARE SECTION;
    EXEC SQL DESCRIBE INPUT S USING DESCRIPTOR 'input descriptor';
   EXEC SQL GET DESCRIPTOR 'input descriptor' :input count = COUNT;
   ANSI_varchar_type=12;
    for (i=0; i < input count; i++)
       occurs = i +1;
                                             /* occurence is 1 based */
       EXEC SQL GET DESCRIPTOR 'input descriptor'
                VALUE :occurs :name = NAME;
       printf ("\nEnter value for input variable %*.*s: ", 10,31, name);
        fgets(input buf, sizeof(input buf), stdin);
       input_len = strlen(input_buf) - 1; /* get rid of new line */
       input buf[input len] = '\0';
                                        /* null terminate */
       EXEC SQL SET DESCRIPTOR 'input descriptor'
                 VALUE : occurs TYPE = : ANSI varchar type,
                              LENGTH = :input len,
                               DATA = :input_buf;
    return(sqlca.sqlcode);
```

```
int process output()
   int i, j;
   EXEC SQL BEGIN DECLARE SECTION;
     int output count, occurs, type, len, col len;
     short indi;
     char data[MAX VAR LEN], name[MAX NAME LEN];
   EXEC SQL END DECLARE SECTION;
   if (!select found)
       return(0);
   EXEC SQL DESCRIBE OUTPUT S USING DESCRIPTOR 'output descriptor';
   EXEC SQL GET DESCRIPTOR 'output descriptor' :output count = COUNT;
   printf ("\n");
   type = 12;
                         /* ANSI VARYING character type */
   len = MAX VAR LEN;
                       /* use the max allocated length */
   for (i = 0; i < output count; i++)</pre>
        occurs = i + 1;
        EXEC SQL GET DESCRIPTOR 'output descriptor' VALUE :occurs
                 :name = NAME;
        printf("%-*.*s ", 9,9, name);
        EXEC SQL SET DESCRIPTOR 'output descriptor' VALUE :occurs
                 TYPE = :type, LENGTH = :len;
    printf("\n");
    /* FETCH each row selected and print the column values. */
    EXEC SQL WHENEVER NOT FOUND GOTO end select loop;
    for (;;)
        EXEC SQL FETCH C INTO DESCRIPTOR 'output descriptor';
        for (i=0; i < output_count; i++)</pre>
          {
            occurs = i + 1;
            EXEC SQL GET DESCRIPTOR 'output descriptor' VALUE :occurs
                 :data = DATA, :indi = INDICATOR;
            if (indi == -1)
              printf("%-*.*s ", 9,9, "NULL");
            else
              printf("%-*.*s ", 9,9, data); /* simplified output formatting */
                          /* truncation will occur, but columns will line up */
         printf ("\n");
end select loop:
    return(0);
void help()
    \verb"puts("\n\nEnter a SQL statement or a PL/SQL block at the SQL> prompt.");
    puts ("Statements can be continued over several lines, except");
    puts("within string literals.");
    puts("Terminate a SQL statement with a semicolon.");
```

```
puts("Terminate a PL/SQL block (which can contain embedded semicolons)");
   puts("with a slash (/).");
   puts("Typing \"exit\" (no semicolon needed) exits the program.");
   puts("You typed \"?\" or \"help\" to get this message.\n\n");
void sql error()
    /* ORACLE error handler */
   printf("\n\nANSI sqlstate: %s: ", SQLSTATE);
   printf ("\n\n%.70s\n", sqlca.sqlerrm.sqlerrmc);
   if (parse_flag)
       printf
        ("Parse error at character offset %d in SQL statement.\n",
          sqlca.sqlerrd[4]);
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL ROLLBACK WORK;
   longjmp(jmp continue, 1);
int oracle connect()
    EXEC SQL BEGIN DECLARE SECTION;
       VARCHAR username[128];
       VARCHAR password[32];
    EXEC SQL END DECLARE SECTION;
    printf("\nusername: ");
    fgets((char *) username.arr, sizeof username.arr, stdin);
    username.arr[strlen((char *) username.arr)-1] = '\0';
    username.len = (unsigned short)strlen((char *) username.arr);
   printf("password: ");
    fgets((char *) password.arr, sizeof password.arr, stdin);
    password.arr[strlen((char *) password.arr) - 1] = '\0';
    password.len = (unsigned short)strlen((char *) password.arr);
    EXEC SQL WHENEVER SQLERROR GOTO connect error;
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user %s.\n", username.arr);
   return 0;
    fprintf(stderr, "Cannot connect to ORACLE as user %s\n", username.arr);
    return -1;
```

# 14.6.2 ansidyn2.pc

This program demonstrates using ANSI Dynamic SQL to process SQL statements which are not known until runtime. It uses the Oracle extensions for batch processing and reference semantics.

The program connects you to ORACLE using your username and password, then prompts you for a SQL statement. Enter legal SQL or PL/SQL statement using interactive, not embedded, SQL syntax, terminating the statement with a semicolon. Your statement will be processed. If it is a query, the fetched rows are displayed.

You can enter multiline statements. The limit is 1023 characters. There is a limit on the size of the variables, MAX\_VAR\_LEN, defined as 255. This program processes up to 40 bind variables and 40 select-list items.

Precompile the program with dynamic = ansi, for example:

```
proc dynamic=ansi ansidyn2
/********************
ANSI Dynamic Demo 2: ANSI Dynamic SQL with reference semantics,
                         batch processing and global descriptor
                         names in host variables
This program demonstates using ANSI Dynamic SQL to process SQL
statements which are not known until runtime. It uses the Oracle
extensions for batch processing and reference semantics.
The program connects you to ORACLE using your username and password,
then prompts you for a SQL statement. Enter legal SQL or PL/SQL
statement using interactive, not embedded, SQL syntax, terminating the
statement with a seimcolon. Your statement will be processed. If it
is a query, the fetched rows are displayed.
If your statement has input bind variables (other than in a where clause),
the program will ask for an input array size and then allow you to enter
that number of input values. If your statment has output, the program will
ask you for an output array size and will do array fetching using that value.
It will also output the rows fetched in one batch together, so using a small
value for the output array size will improve the look of the output.
For example, connected as scott/tiger, try select empno, ename from emp
with an output array size of 4;
You can enter multiline statements. The limit is 1023 characters.
There is a limit on the size of the variables, MAX VAR LEN, defined as 255.
This program processes up to 40 bind variables and 40 select-list items.
Precompile with program with dynamic=ansi, for example:
proc dynamic=ansi ansidyn2
************************
#include <stdio.h>
#include <string.h>
#include <setjmp.h>
#include <stdlib.h>
#include <sqlcpr.h>
#define MAX OCCURENCES 40
#define MAX ARRSZ 100
#define MAX VAR LEN 255
#define MAX NAME LEN 31
#ifndef NULL
#define NULL 0
#endif
```



```
/* Prototypes */
#if defined( STDC )
 void sql error(void);
  int oracle connect(void);
  int get dyn statement(void);
  int process input(void);
 int process output (void);
 void rows processed(void);
 void help(void);
#else
 void sql_error(/*_ void _*/);
 int oracle_connect(/*_ void _*/);
 int get dyn statement(/* void */);
 int process input(/* void */);
 int process output(/* void */);
 void rows processed(/* void */);
 void help(/* void */);
#endif
EXEC SQL INCLUDE sqlca;
/* global variables */
                                                                         * /
char dyn statement[1024];
                                                /* statement variable
EXEC SQL VAR dyn statement IS STRING(1024);
char indesc[]="input descriptor";
                                                 /* descriptor names
char outdesc[]="output descriptor";
char input[MAX OCCURENCES][MAX ARRSZ][MAX VAR LEN +1 ],
                                                          /* data areas */
      output[MAX OCCURENCES][MAX ARRSZ][MAX VAR LEN + 1];
short outindi[MAX OCCURENCES][MAX ARRSZ];
                                             /* output indicators
short *iptr;
                       /* size of input batch, that is, number of rows */
int in array size;
int out_array_size; /* size of input batch, that is, number of rows */
int max_array_size=MAX_ARRSZ; /* maximum arrays size used for allocates */
char *dml commands[] = {"SELECT", "select", "INSERT", "insert",
                       "UPDATE", "update", "DELETE", "delete"};
int select found, cursor open = 0;
/* Define a buffer to hold longjmp state info. */
jmp buf jmp continue;
/* A global flag for the error routine. */
int parse flag = 0;
void main()
    /* Connect to the database. */
   if (oracle connect() != 0)
       exit(1);
   EXEC SQL WHENEVER SQLERROR DO sql_error();
    /\star Allocate the input and output descriptors. \star/
    EXEC SQL FOR :max array size
            ALLOCATE DESCRIPTOR GLOBAL :indesc;
```

```
EXEC SQL FOR :max array size
            ALLOCATE DESCRIPTOR GLOBAL :outdesc;
    /* Process SQL statements. */
    for (;;)
        (void) setjmp(jmp continue);
        /* Get the statement. Break on "exit". */
       if (get dyn statement() != 0)
            break;
        /\star Prepare the statement and declare a cursor. \star/
       parse flag = 1;  /* Set a flag for sql error(). */
       EXEC SQL PREPARE S FROM :dyn_statement;
       parse flag = 0; /* Unset the flag. */
       EXEC SQL DECLARE C CURSOR FOR S;
       /* Call the function that processes the input. */
       if (process input())
            exit(1);
        /\star Open the cursor and execute the statement. \star/
       EXEC SQL FOR :in array size
            OPEN C USING DESCRIPTOR GLOBAL :indesc;
       cursor open = 1;
        /* Call the function that processes the output. */
       if (process output())
            exit(1);
        /* Tell user how many rows were processed. */
       rows processed();
    } /* end of for(;;) statement-processing loop */
    /* Close the cursor. */
   if (cursor open)
     EXEC SQL CLOSE C;
    /* Deallocate the descriptors */
    EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL :indesc;
   EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL :outdesc;
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL COMMIT WORK RELEASE;
    puts("\nHave a good day!\n");
   EXEC SQL WHENEVER SQLERROR DO sql error();
   return;
int get dyn statement()
   char *cp, linebuf[256];
   int iter, plsql;
   for (plsql = 0, iter = 1; ;)
```

```
if (iter == 1)
        {
            printf("\nSQL> ");
            dyn_statement[0] = ' \0';
            select found = 0;
        fgets(linebuf, sizeof linebuf, stdin);
        cp = strrchr(linebuf, '\n');
        if (cp && cp != linebuf)
            *cp = ' ';
        else if (cp == linebuf)
            continue;
        if ((strncmp(linebuf, "SELECT", 6) == 0) ||
            (strncmp(linebuf, "select", 6) == 0))
            select_found=1;;
        }
        if ((strncmp(linebuf, "EXIT", 4) == 0) ||
            (strncmp(linebuf, "exit", 4) == 0))
            return -1;
        else if (linebuf[0] == '?' ||
            (strncmp(linebuf, "HELP", 4) == 0) | |
            (strncmp(linebuf, "help", 4) == 0))
        {
            help();
            iter = 1;
            continue;
        }
        if (strstr(linebuf, "BEGIN") ||
            (strstr(linebuf, "begin")))
        {
            plsql = 1;
        strcat(dyn_statement, linebuf);
        if ((plsql && (cp = strrchr(dyn_statement, '/'))) ||
            (!plsql && (cp = strrchr(dyn_statement, ';'))))
            *cp = '\0';
            break;
        }
        else
            iter++;
            printf("%3d ", iter);
    return 0;
int process input()
```

}

```
int i, j;
   char name[31];
    int input count, input len= MAX VAR LEN;
    int occurs, string_type = 5;
    int string_len;
    char arr size[3];
    EXEC SQL DESCRIBE INPUT S USING DESCRIPTOR GLOBAL :indesc;
    EXEC SQL GET DESCRIPTOR GLOBAL :indesc :input count = COUNT;
    if (input count > 0 && !select found )
      { /* get input array size */
         printf ("\nEnter value for input array size (max is %d) : ",
                          max array size);
       fgets(arr_size, 4, stdin);
       in array size = atoi(arr size);
    else
        in_array_size = 1;
    for (i=0; i < input count; i++)
       occurs = i +1;
                                            /* occurence is 1 based */
       EXEC SQL GET DESCRIPTOR GLOBAL :indesc
                VALUE :occurs :name = NAME;
        for (j=0; j < in array size; j++)
         if (in array size == 1)
           printf ("\nEnter value for input variable %*.*s: ",10,31, name);
         else
           printf ("\nEnter %d%s value for input variable %*.*s: ",
              j +1, ((j==0) ? "st" : (j==1) ? "nd" : (j==2) ? "rd" : "th"),
                     10,31, name);
         fgets(input[i][j], sizeof(input[i][j]), stdin);
         string_len = strlen(input[i][j]);
         input[i][j][string len - 1 ] = '\0'; /* change \n to \0 */
       EXEC SQL SET DESCRIPTOR GLOBAL :indesc
                VALUE : occurs TYPE = :string type, LENGTH = :input len;
       EXEC SQL FOR :in array size
                SET DESCRIPTOR GLOBAL :indesc
                    VALUE :occurs REF DATA = :input[i];
    }
   return(sqlca.sqlcode);
int process output()
  int i, j;
  int output count, occurs;
  int type, output len= MAX VAR LEN;
  char name[MAX OCCURENCES][MAX NAME LEN];
  int rows_this_fetch=0, cumulative_rows=0;
  char arr_size[3];
  if (!select_found)
     return(0);
  EXEC SQL DESCRIBE OUTPUT S USING DESCRIPTOR GLOBAL :outdesc;
```

```
EXEC SQL GET DESCRIPTOR GLOBAL :outdesc :output count = COUNT;
if (output count > 0 )
     printf ("\nEnter value for output array size (max is %d) : ",
                   max array size);
     fgets(arr size, 4, stdin);
     out array size = atoi(arr size);
if (out_array_size < 1)</pre>
                           /* must have at least one */
    out array size = 1;
printf ("\n");
for (i = 0; i < output count; i++)
   occurs = i + 1;
   EXEC SQL GET DESCRIPTOR GLOBAL :outdesc VALUE :occurs
           :type = TYPE, :name[i] = NAME;
                                           /* occurence is one based */
   occurs = i + 1;
   type = 5; /* force all data to be null terminated character */
   EXEC SQL SET DESCRIPTOR GLOBAL :outdesc VALUE :occurs
            TYPE = :type, LENGTH = :output len;
   iptr = (short *)&outindi[i]; /* no mult-dimension non-char host vars */
   EXEC SQL FOR :out array size
            SET DESCRIPTOR GLOBAL :outdesc VALUE :occurs
            REF DATA = :output[i], REF INDICATOR = :iptr;
EXEC SQL WHENEVER NOT FOUND GOTO end select loop;
/* print the column headings */
for (j=0; j < out_array_size; j++)</pre>
   for (i=0; i < output_count; i++)</pre>
     printf("%-*.*s ", 9,9, name[i]);
printf("\n");
/* FETCH each row selected and print the column values. */
for (;;)
   EXEC SQL FOR :out array size
          FETCH C INTO DESCRIPTOR GLOBAL :outdesc;
   rows this fetch = sqlca.sqlerrd[2] - cumulative rows;
   cumulative rows = sqlca.sqlerrd[2];
   if (rows this fetch)
   for (j=0; j < out array size && j < rows this fetch; j++)
              /* output by columns using simplified formatting */
      for (i=0; i < output count; i++)
        {
             if (outindi[i][j] == -1)
                printf("%-*.*s ", 9, 9, "NULL");
            else
               printf("%-*.*s ", 9, 9, output[i][j]); /* simplified */
                           /* output formatting may cause truncation */
                           /* but columns will line up */
    printf ("\n");
```

```
end_select_loop:
   /* print any unprinted rows */
   rows_this_fetch = sqlca.sqlerrd[2] - cumulative_rows;
   cumulative rows = sqlca.sqlerrd[2];
   if (rows this fetch)
     for (j=0; j < out_array_size && j < rows_this fetch; j++)</pre>
                   /* output by columns using simplified formatting */
         for (i=0; i < output count; i++)
           {
              if (outindi[i][j] == -1)
                   printf("%-*.*s ",9, 9, "NULL");
                  printf("%-*.*s ", 9, 9, output[i][j]);
            }
   return(0);
}
void rows processed()
{
  int i;
   for (i = 0; i < 8; i++)
       if (strncmp(dyn statement, dml commands[i], 6) == 0)
            printf("\n\n%d row%c processed.\n", sqlca.sqlerrd[2],
                       sqlca.sqlerrd[2] == 1 ? ' ' : 's');
            break;
   return;
void help()
    puts("\n\nEnter a SQL statement or a PL/SQL block at the SQL> prompt.");
    puts ("Statements can be continued over several lines, except");
    puts("within string literals.");
    puts("Terminate a SQL statement with a semicolon.");
    puts("Terminate a PL/SQL block (which can contain embedded semicolons)");
    puts("with a slash (/).");
    puts("Typing \"exit\" (no semicolon needed) exits the program.");
    puts("You typed \"?\" or \"help\" to get this message.\n\n");
}
void sql error()
    /* ORACLE error handler */
    printf ("\n\n%.70s\n", sqlca.sqlerrm.sqlerrmc);
    if (parse flag)
        printf
        ("Parse error at character offset %d in SQL statement.\n",
           sqlca.sqlerrd[4]);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;
    longjmp(jmp continue, 1);
}
```

```
int oracle_connect()
   EXEC SQL BEGIN DECLARE SECTION;
       VARCHAR username[128];
       VARCHAR password[32];
   EXEC SQL END DECLARE SECTION;
   printf("\nusername: ");
   fgets((char *) username.arr, sizeof username.arr, stdin);
   username.arr[strlen((char *) username.arr)-1] = '\0';
   username.len = (unsigned short)strlen((char *) username.arr);
   printf("password: ");
   fgets((char *) password.arr, sizeof password.arr, stdin);
   password.arr[strlen((char *) password.arr) - 1] = '\0';
   password.len = (unsigned short)strlen((char *) password.arr);
   EXEC SQL WHENEVER SQLERROR GOTO connect error;
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
   printf("\nConnected to ORACLE as user %s.\n", username.arr);
   return 0;
connect error:
   fprintf(stderr, "Cannot connect to ORACLE as user %s\n", username.arr);
   return -1;
```



# Oracle Dynamic SQL: Method 4

This chapter shows you how to implement Oracle dynamic SQL Method 4, which lets your program accept or build dynamic SQL statements that contain a varying number of host variables. Use this to support existing applications. Use ANSI Dynamic SQL Method 4 for all new applications.

Oracle Dynamic SQL Method 4 does not support object types, cursor variables, arrays of structs, DML returning clauses, Unicode variables, and LOBs. Use ANSI Dynamic SQL Method 4 instead. This chapter contains the following topics:

- Meeting the Special Requirements of Method 4
- Understanding the SQLDA
- About Using the SQLDA Variables
- Some Preliminaries
- The Basic Steps
- A Closer Look at Each Step
- Example Program: Dynamic SQL Method 4
- Sample Program : Dynamic SQL Method 4 using Scrollable Cursors

```
See Also:
```

- Oracle Dynamic SQL for a discussion of dynamic SQL Methods 1, 2, and 3, and an overview of Method 4
  - ANSI Dynamic SQL

# 15.1 Meeting the Special Requirements of Method 4

Before looking into the requirements of Method 4, you should feel comfortable with the terms select-list item and placeholder. Select-list items are the columns or expressions following the keyword SELECT in a query. For example, the following dynamic query contains three select-list items:

```
SELECT ename, job, sal + comm FROM emp WHERE deptno = 20
```

Placeholders are dummy bind variables that hold places in a SQL statement for actual bind variables. You do not declare placeholders, and can name them anything you like.

Placeholders for bind variables are most often used in the SET, VALUES, and WHERE clauses. For example, the following dynamic SQL statements each contain two placeholders:

```
INSERT INTO emp (empno, deptno) VALUES (:e, :d)
DELETE FROM dept WHERE deptno = :num OR loc = :loc
```

### 15.1.1 What Makes Method 4 Special?

Unlike Methods 1, 2, and 3, dynamic SQL Method 4 lets your program

- Accept or build dynamic SQL statements that contain an unknown number of select-list items or placeholders, and
- Take explicit control over datatype conversion between Oracle and C types

To add this flexibility to your program, you must give the Oracle runtime library additional information.

### 15.1.2 What Information Does Oracle Need?

The Pro\*C/C++ Precompiler generates calls to Oracle for all executable dynamic SQL statements. If a dynamic SQL statement contains no select-list items or placeholders, Oracle needs no additional information to execute the statement. The following DELETE statement falls into this category:

```
DELETE FROM emp WHERE deptno = 30
```

However, most dynamic SQL statements contain select-list items or placeholders for bind variables, as does the following

#### **UPDATE** statement:

```
UPDATE emp SET comm = :c WHERE empno = :e
```

To execute a dynamic SQL statement that contains placeholders for bind variables or select-list items, Oracle needs information about the program variables that hold the input (bind) values, and that will hold the FETCHed values when a query is executed. The information needed by Oracle is:

- The number of bind variables and select-list items
- The length of each bind variable and select-list item
- The datatype of each bind variable and select-list item
- The address of each bind variable, and of the output variable that will receive each selectlist item

### 15.1.3 Where Is the Information Stored?

All the information Oracle needs about select-list items or placeholders for bind variables, except their values, is stored in a program data structure called the SQL Descriptor Area (SQLDA). The SQLDA struct is defined in the sqlda.h header file.

Descriptions of select-list items are stored in a *select descriptor*, and descriptions of placeholders for bind variables are stored in a *bind descriptor*.

The values of select-list items are stored in output variables; the values of bind variables are stored in input variables. You store the addresses of these variables in the select or bind SQLDA so that Oracle knows where to write output values and read input values.

How do values get stored in these data variables? Output values are FETCHed using a cursor, and input values are typically filled in by the program, usually from information entered interactively by the user.



## 15.1.4 How is the SQLDA Referenced?

The bind and select descriptors are usually referenced by pointer. A dynamic SQL program should declare a pointer to at least one bind descriptor, and a pointer to at least one select descriptor, in the following way:

```
#include <sqlda.h>
...
SQLDA *bind_dp;
SQLDA *select dp;
```

You can then use the SQLSQLDAAlloc() function to allocate the descriptor, as follows:

```
bind dp = SQLSQLDAAlloc(runtime context, size, name length, ind name length);
```

SQLSQLDAAlloc() was known as sqlaldt() before Oracle8.

The constant SQL\_SINGLE\_RCTX is defined as (dvoid\*) 0. Use it for *runtime\_context* when your application is single-threaded.

### See Also:

- Table 15-3, for information on this and other SQLLIB functions
- "Allocating a SQLDA" for detailed information about SQLSQLDAAlloc() and its parameters

### 15.1.5 How is the Information Obtained?

You use the DESCRIBE statement to help obtain the information Oracle needs.

The DESCRIBE SELECT LIST statement examines each select-list item to determine its name and name length. It then stores this information in the select SQLDA for your use. For example, you might use select-list names as column headings in a printout. The total number of select-list items is also stored in the SQLDA by DESCRIBE.

The DESCRIBE BIND VARIABLES statement examines each placeholder to determine its name and length, then stores this information in an input buffer and bind SQLDA for your use. For example, you might use placeholder names to prompt the user for the values of bind variables.

# 15.2 Understanding the SQLDA

This section describes the SQLDA data structure in detail. You learn how to declare it, what variables it contains, how to initialize them, and how to use them in your program.

### 15.2.1 Purpose of the SQLDA

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or placeholders for bind variables. To process this kind of dynamic SQL statement, your program must explicitly declare SQLDAs, also called *descriptors*. Each descriptor is a **struct** which you must copy or code into your program.

A *select descriptor* holds descriptions of select-list items, and the addresses of output buffers where the names and values of select-list items are stored.



The "name" of a select-list item can be a column name, a column alias, or the text of an expression such as sal + comm.

A *bind descriptor* holds descriptions of bind variables and indicator variables, and the addresses of input buffers where the names and values of bind variables and indicator variables are stored.

### 15.2.2 Multiple SQLDAs

If your program has more than one active dynamic SQL statement, each statement must have its own SQLDA(s). You can declare any number of SQLDAs with different names. For example, you might declare three select SQLDAs named  $sel\_desc1$ ,  $sel\_desc2$ , and  $sel\_desc3$ , so that you can FETCH from three concurrently OPEN cursors. However, non-concurrent cursors can reuse SQLDAs.

## 15.2.3 Declaring a SQLDA

To declare a SQLDA, include the sqlda.h header file. The contents of the SQLDA are:

```
struct SOLDA
                             /* Descriptor size in number of entries */
  int
                                Ptr to Arr of addresses of main variables */
  char
                       *L;
                                            /* Ptr to Arr of lengths of buffers */
  int
                      *T; /* Ptr to Arr of types of buffers */

**I; * Ptr to Arr of addresses of indicator vars */

F; /* Number of variables found by DESCRIBE */
  short
  short
  int
                      **S;
  char
                                        /* Ptr to Arr of variable name pointers */
                                   /* Ptr to Arr of max lengths of var. names */
                       *M;
  short
                              * Ptr to Arr of current lengths of var. names */
                       *C;
  short
  char
                      **X;
                                    /* Ptr to Arr of ind. var. name pointers */
  short
                       *Y; /* Ptr to Arr of max lengths of ind. var. names */
                       *Z; /* Ptr to Arr of cur lengths of ind. var. names */
  short.
                      *CP;
  long long
                                             /* Ptr to Arr of column properties */
                      **D;
                                          /* Ptr to Arr of domain name pointers */
  char
 unsigned short *DML; /* Ptr to Arr of max lengths of domain names */
unsigned short *DCL; /* Ptr to Arr of cur lengths of dom. names */
```

## 15.2.4 Allocating a SQLDA

After declaring a SQLDA, you allocate storage space for it with the SQLSQLDAAlloc() library function (known as sqlaldt() before Oracle8), using the syntax:

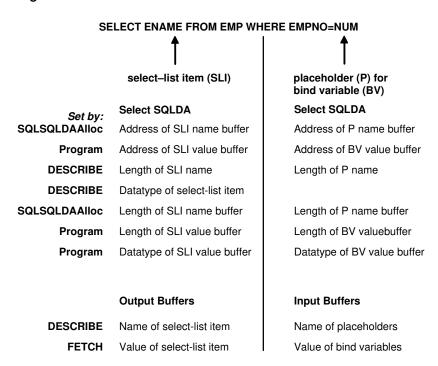
```
descriptor_name = SQLSQLDAAlloc (runtime_context, max_vars, max_name, max_ind_name);
where:
```

Syntax	Description
runtime_context	pointer to runtime context
max_vars	Is the maximum number of select-list items or placeholders that the descriptor can describe.
max_name	Is the maximum length of select-list or placeholder names.
max_ind_name	Is the maximum length of indicator variable names, which are optionally appended to placeholder names. This parameter applies to bind descriptors only, so set it to zero when allocating a select descriptor.

Besides the descriptor, SQLSQLDAAlloc() allocates data buffers to which descriptor variables point.

Figure 15-1 shows whether variables are set by SQLSQLDAAlloc() calls, DESCRIBE commands, FETCH commands, or program assignments.

Figure 15-1 How Variables Are Set



### **Related Topics**

- About Using the SQLDA Variables
- Allocate Storage Space for the Descriptors

# 15.3 About Using the SQLDA Variables

This section explains the purpose and use of each variable in the SQLDA.

### 15.3.1 The N Variable

**N** specifies the maximum number of select-list items or placeholders that can be DESCRIBEd. Thus, *N* determines the number of elements in the descriptor arrays.

Before issuing the optional DESCRIBE command, you must set N to the dimension of the descriptor arrays using the SQLSQLDAAlloc() library function. After the DESCRIBE, you must reset N to the actual number of variables DESCRIBEd, which is stored in the F variable.

### 15.3.2 The V Variable

V is a pointer to an array of addresses of data buffers that store select-list or bind-variable values.

When you allocate the descriptor, SQLSQLDAAlloc() zeros the elements V[0] through V[N-1] in the array of addresses.

For select descriptors, you must allocate data buffers and set this array before issuing the FETCH command. The statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

directs Oracle to store FETCHed select-list values in the data buffers to which V[0] through V[N-1] point. Oracle stores the *i*th select-list value in the data buffer to which V[i] points.

**For bind descriptors**, you must set this array before issuing the OPEN command. The statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

directs Oracle to execute the dynamic SQL statement using the bind-variable values to which V[0] through V[N-1] point. Oracle finds the ith bind-variable value in the data buffer to which V[i] points.

### 15.3.3 The L Variable

L is a pointer to an array of lengths of select-list or bind-variable values stored in data buffers.

For select descriptors, DESCRIBE SELECT LIST sets the array of lengths to the maximum expected for each select-list item. However, you might want to reset some lengths before issuing a FETCH command. FETCH returns at most n characters, where n is the value of  $\mathbb{L}[i]$  before the FETCH.

The format of the length differs among Oracle datatypes. For CHAR or VARCHAR2 select-list items, DESCRIBE SELECT LIST sets  $\mathbb{L}[i]$  to the maximum length of the select-list item. For NUMBER select-list items, scale and precision are returned respectively in the low and next-higher bytes of the variable. You can use the library function SQLNumberPrecV6() to extract precision and scale values from  $\mathbb{L}[i]$ .

You must reset L[i] to the required length of the data buffer before the FETCH. For example, when coercing a NUMBER to a C **char** string, set L[i] to the precision of the number plus two for the sign and decimal point. When coercing a NUMBER to a C **float**, set L[i] to the length of **floats** on your system.



**For bind descriptors**, you must set the array of lengths before issuing the OPEN command. For example, you can use strlen() to get the lengths of bind-variable character strings entered by the user, then set the appropriate array elements.

Because Oracle accesses a data buffer indirectly, using the address stored in V[i], it does not know the length of the value in that buffer. If you want to change the length Oracle uses for the *i*th select-list or bind-variable value, reset  $\mathbb{L}[i]$  to the length you need. Each input or output buffer can have a different length.

#### **Related Topics**

Extracting Precision and Scale

### 15.3.4 The *T* Variable

*T* is a pointer to an array of datatype codes of select-list or bind-variable values. These codes determine how Oracle data is converted when stored in the data buffers addressed by elements of the *V* array.

**For select descriptors**, DESCRIBE SELECT LIST sets the array of datatype codes to the *internal* datatype (CHAR, NUMBER, or DATE, for example) of the items in the select list.

Before FETCHing, you might want to reset some datatypes because the internal format of Oracle datatypes can be difficult to handle. For display purposes, it is usually a good idea to coerce the datatype of select-list values to VARCHAR2 or STRING. For calculations, you might want to coerce numbers from Oracle to C format.

The high bit of T[i] is set to indicate the NULL/not NULL status of the ith select-list item. You must always clear this bit before issuing an OPEN or FETCH command. You use the library function SQLColumnNullCheck() to retrieve the datatype code and clear the NULL/not NULL bit.

You should change the Oracle NUMBER internal datatype to an external datatype compatible with that of the C data buffer to which V[i] points.

**For bind descriptors**, DESCRIBE BIND VARIABLES sets the array of datatype codes to zeros. You must set the datatype code stored in each element before issuing the OPEN command. The code represents the external (C) datatype of the data buffer to which V[i] points. Often, bind-variable values are stored in character strings, so the datatype array elements are set to 1 (the VARCHAR2 datatype code). You can also use datatype code 5 (STRING).

To change the datatype of the *i*th select-list or bind-variable value, reset  $\mathbb{T}[i]$  to the datatype you want.

### **Related Topics**

- Converting Data
- Coercing Datatypes
- Handling NULL/Not NULL Datatypes

### 15.3.5 The I Variable

*I* is a pointer to an array of addresses of data buffers that store indicator-variable values.

You must set the elements I[0] through I[N-1] in the array of addresses.

**For select descriptors**, you must set the array of addresses before issuing the FETCH command. When Oracle executes the statement

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

if the *i*th returned select-list value is NULL, the indicator-variable value to which  $\mathbb{I}[i]$  points is set to -1. Otherwise, it is set to zero (the value is not NULL) or a positive integer (the value was truncated).

**For bind descriptors**, you must set the array of addresses and associated indicator variables before issuing the OPEN command. When Oracle executes the statement

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

the data buffer to which  $\mathbb{I}[i]$  points determines whether the *i*th bind variable has a NULL value. If the value of an indicator variable is -1, the value of its associated bind variable is NULL.

### 15.3.6 The F Variable

*F* is the actual number of select-list items or placeholders found by DESCRIBE.

F is set by DESCRIBE. If F is less than zero, DESCRIBE has found too many select-list items or placeholders for the allocated size of the descriptor. For example, if you set N to 10 but DESCRIBE finds 11 select-list items or placeholders, F is set to -11. This feature lets you dynamically reallocate a larger storage area for select-list items or placeholders if necessary.

### 15.3.7 The S Variable

S is a pointer to an array of addresses of data buffers that store select-list or placeholder names as they appear in dynamic SQL statements.

You use SQLSQLDAAlloc() to allocate the data buffers and store their addresses in the S array.

DESCRIBE directs Oracle to store the name of the *i*th select-list item or placeholder in the data buffer to which S[i] points.

## 15.3.8 The M Variable

*M* is a pointer to an array of maximum lengths of data buffers that store select-list or placeholder names. The buffers are addressed by elements of the *S* array.

When you allocate the descriptor, SQLSQLDAAlloc() sets the elements M[0] through M[N-1] in the array of maximum lengths. When stored in the data buffer to which S[i] points, the ith name is truncated to the length in M[i] if necessary.

### 15.3.9 The C Variable

C is a pointer to an array of current lengths of select-list or placeholder names.

DESCRIBE sets the elements  $\mathbb{C}[0]$  through  $\mathbb{C}[\mathbb{N}-1]$  in the array of current lengths. After a DESCRIBE, the array contains the number of characters in each select-list or placeholder name.



### 15.3.10 The *X* Variable

X is a pointer to an array of addresses of data buffers that store indicator-variable names. You can associate indicator-variable *values* with select-list items and bind variables. However, you can associate indicator-variable *names* only with bind variables. So, X applies only to bind descriptors.

Use SQLSQLDAAlloc() to allocate the data buffers and store their addresses in the X array.

DESCRIBE BIND VARIABLES directs Oracle to store the name of the *i*th indicator variable in the data buffer to which X[i] points.

### 15.3.11 The Y Variable

Y is a pointer to an array of maximum lengths of data buffers that store indicator-variable names. Like X, Y applies only to bind descriptors.

You use SQLSQLDAAlloc() to set the elements Y[0] through Y[N-1] in the array of maximum lengths. When stored in the data buffer to which X[i] points, the ith name is truncated to the length in Y[i] if necessary.

### 15.3.12 The *Z* Variable

Z is a pointer to an array of current lengths of indicator-variable names. Like X and Y, Z applies only to bind descriptors.

DESCRIBE BIND VARIABLES sets the elements  $\mathbb{Z}[0]$  through  $\mathbb{Z}[\mathbb{N}-1]$  in the array of current lengths. After a DESCRIBE, the array contains the number of characters in each indicator-variable name.

## 15.4 Some Preliminaries

You need a working knowledge of the following subjects to implement dynamic SQL Method 4:

- Converting Data
- Coercing Datatypes
- Handling NULL/Not NULL Datatypes

### 15.4.1 Converting Data

This section provides more detail about the T (datatype) descriptor array. In host programs that use neither datatype equivalencing nor dynamic SQL Method 4, the conversion between Oracle internal and external datatypes is determined at precompile time. By default, the precompiler assigns a specific external datatype to each host variable in the Declare Section. For example, the precompiler assigns the INTEGER external datatype to host variables of type int.

However, Method 4 lets you control data conversion and formatting. You specify conversions by setting datatype codes in the T descriptor array.



### 15.4.1.1 Internal Datatypes

Internal datatypes specify the formats used by Oracle to store column values in database tables, as well as the formats used to represent pseudocolumn values.

When you issue a DESCRIBE SELECT LIST command, Oracle returns the internal datatype code for each select-list item to the T descriptor array. For example, the datatype code for the ith select-list item is returned to T[i].

Table 15-1 shows the Oracle internal datatypes and their codes:

Table 15-1 Oracle Internal Datatypes

Oracle Internal Datatype	Code
VARCHAR2	1
NUMBER	2
LONG	8
BINARY_FLOAT	100
BINARY_DOUBLE	101
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHARACTER (or CHAR)	96
Universal ROWID	104

### 15.4.1.2 External Datatypes

External datatypes specify the formats used to store values in input and output host variables.

The DESCRIBE BIND VARIABLES command sets the T array of datatype codes to zeros. So, you must reset the codes *before* issuing the OPEN command. The codes tell Oracle which external datatypes to expect for the various bind variables. For the ith bind variable, reset T[i] to the external datatype you want.

Table 15-2 shows the Oracle external datatypes and their codes, as well as the C datatype normally used with each external datatype.

Table 15-2 Oracle External Datatypes and Datatype Codes

External Datatype	Code	C Datatype
VARCHAR2	1	char[n]
NUMBER	2	char[n] ( n <= 22)
INTEGER	3	int
FLOAT	4	float
STRING	5	char[n+1]
VARNUM	6	char[n] (n <= 22)
DECIMAL	7	float



Table 15-2 (Cont.) Oracle External Datatypes and Datatype Codes

External Datatype	Code	C Datatype
LONG	8	char[n]
SQLT_BFLOAT	21	float
SQLT_BDOUBLE	22	double
VARCHAR	9	char[n+2]
ROWID	11	char[n]
DATE	12	char[n]
VARRAW	15	char[n]
RAW	23	unsigned char[n]
LONG RAW	24	unsigned char[n]
UNSIGNED	68	unsigned int
DISPLAY	91	char[n]
LONG VARCHAR	94	char[n+4]
LONG VARRAW	95	unsigned char[n+4]
CHAR	96	char[n]
CHARF	96	char[n]
CHARZ	97	char[n+1]

#### **Related Topics**

- Oracle Datatypes
- Oracle Database SQL Language Reference

### 15.4.2 Coercing Datatypes

For a select descriptor, DESCRIBE SELECT LIST can return any of the Oracle internal datatypes. Often, as in the case of character data, the internal datatype corresponds exactly to the external datatype you want to use. However, a few internal datatypes map to external datatypes that can be difficult to handle. So, you might want to reset some elements in the T descriptor array. For example, you might want to reset NUMBER values to FLOAT values, which correspond to **float** values in C. Oracle does any necessary conversion between internal and external datatypes at FETCH time. So, be sure to reset the datatypes *after* the DESCRIBE SELECT LIST but *before* the FETCH.

For a bind descriptor, DESCRIBE BIND VARIABLES does *not* return the datatypes of bind variables, only their number and names. Therefore, you must explicitly set the T array of datatype codes to tell Oracle the external datatype of each bind variable. Oracle does any necessary conversion between external and internal datatypes at OPEN time.

When you reset datatype codes in the *T* descriptor array, you are "coercing datatypes." For example, to coerce the *i*th select-list value to STRING, you use the following statement:

```
/* Coerce select-list value to STRING. */
select_des->T[i] = 5;
```

When coercing a NUMBER select-list value to STRING for display purposes, you must also extract the precision and scale bytes of the value and use them to compute a maximum display

length. Then, before the FETCH, you must reset the appropriate element of the L (length) descriptor array to tell Oracle the buffer length to use.

For example, if DESCRIBE SELECT LIST finds that the ith select-list item is of type NUMBER, and you want to store the returned value in a C variable declared as **float**, simply set  $\mathbb{T}[i]$  to 4 and L[i] to the length of **float**s on your system.

#### WARNING:

In some cases, the internal datatypes that DESCRIBE SELECT LIST returns might not suit your purposes. Two examples of this are DATE and NUMBER. When you DESCRIBE a DATE select-list item, Oracle returns the datatype code 12 to the T descriptor array. Unless you reset the code before the FETCH, the date value is returned in its 7-byte internal format. To get the date in character format (DD-MON-YY), you can change the datatype code from 12 to 1 (VARCHAR2) or 5 (STRING), and increase the L value from 7 to 9 or 10.

Similarly, when you DESCRIBE a NUMBER select-list item, Oracle returns the datatype code 2 to the T array. Unless you reset the code before the FETCH, the numeric value is returned in its internal format, which is probably not what you want. So, change the code from 2 to 1 (VARCHAR2), 3 (INTEGER), 4 (FLOAT), 5 (STRING) or some other appropriate datatype.

#### **Related Topics**

**Extracting Precision and Scale** 

### 15.4.2.1 Extracting Precision and Scale

The library function SQLNumberPrecV6() (previously known as sqlprc()) extracts precision and scale. Normally, it is used after the DESCRIBE SELECT LIST, and its first argument is L[i]. You call SQLNumberPrecV6() using the following syntax:



#### Note:

See your platform-specific SQLNumberPrecV6 header file for the correct prototype for your platform.

SQLNumberPrecV6(dvoid \*runtime context, int \*length, int \*precision, int \*scale);

#### where:

Syntax	Description
runtime_context	Is the pointer to the runtime context
length	Is a pointer to a long integer variable that stores the length of an Oracle NUMBER value; the length is stored in L[i]. The scale and precision of the value are stored respectively in the low and next-higher bytes.



Syntax	Description
precision	Is a pointer to an integer variable that returns the precision of the NUMBER value. Precision is the number of significant digits. It is set to zero if the select-list item refers to a NUMBER of unspecified size. In this case, because the size is unspecified, you might want to assume the maximum precision (38).
scale	Is a pointer to an integer variable that returns the scale of the NUMBER value. Scale specifies where rounding will occur. For example, a scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46); a scale of -3 means the number is rounded to the nearest thousand (3456 becomes 3000).

When the scale is negative, add its absolute value to the length. For example, a precision of 3 and scale of -2 allow for numbers as large as 99900.

The following example shows how SQLNumberPrecV6() is used to compute maximum display lengths for NUMBER values that will be coerced to STRING:

```
/* Declare variables for the function call. */
sqlda *select des; /* pointer to select descriptor */
             prec; /* precision scal; /* scale
                                                            */
int
int
extern void SQLNumberPrecV6(); /* Declare library function. */
/* Extract precision and scale. */
SQLNumberPrecV6(SQL SINGLE RCTX, &(select des->L[i]), &prec, &scal);
/* Allow for maximum size of NUMBER. */
if (prec == 0)
    prec = 38;
/* Allow for possible decimal point and sign. */
select_des->L[i] = prec + 2;
/* Allow for negative scale. */
if (scal < 0)
    select_des->L[i] += -scal;
```

Notice that the first argument in this function call points to the *i*th element in the array of lengths, and that all three parameters are addresses.

The SQLNumberPrecV6() function returns zero as the precision and scale values for certain SQL datatypes. The SQLNumberPrecV7() function is similar, having the same argument list, and returning the same values, except in the cases of these SQL datatypes:

Table 15-3 Precision and Scale Values for SQL Datatypes

SQL Datatype	Binary Precision	Scale
FLOAT	126	-127
FLOAT(N)	N (range is 1 to 126)	-127
REAL	63	-127
DOUBLE PRECISION	126	-127

# 15.4.3 Handling NULL/Not NULL Datatypes

For every select-list column (not expression), DESCRIBE SELECT LIST returns a NULL/not NULL indication in the datatype array T of the select descriptor. If the ith select-list column is constrained to be not NULL, the high-order bit of  $\mathbb{T}[i]$  is clear; otherwise, it is set.

Before using the datatype in an OPEN or FETCH statement, if the NULL/not NULL bit is set, you must clear it. (Never set the bit.)

You can use the library function SQLColumnNullCheck() (previously was called sqlnul()) to find out if a column allows NULLs, and to clear the datatype's NULL/not NULL bit. You call SQLColumnNullCheck() using the syntax:

#### where:

Syntax	Description
context	Is a pointer to the runtime context
value_type	Is a pointer to an unsigned short integer variable that stores the datatype code of a select-list column; the datatype is stored in T[i].
type_code	Is a pointer to an unsigned short integer variable that returns the datatype code of the select-list column with the high-order bit cleared.
null_status	Is a pointer to an integer variable that returns the null status of the select- list column. 1 means the column allows nulls; 0 means it does not.

The following example shows how to use SQLColumnNullCheck():

Notice that the first and second arguments in the second call to the SQLColumnNullCheck() function point to the *i*th element in the array of datatypes, and that all three parameters are addresses.

# 15.5 The Basic Steps

Method 4 can be used to process *any* dynamic SQL statement. In the coming example, a query is processed so you can see how both input and output host variables are handled.

To process the dynamic query, our example program takes the following steps:

- Declare a host string in the Declare Section to hold the query text.
- 2. Declare select and bind SQLDAs.
- 3. Allocate storage space for the select and bind descriptors.
- 4. Set the maximum number of select-list items and placeholders that can be DESCRIBEd.
- **5.** Put the query text in the host string.



- PREPARE the query from the host string.
- DECLARE a cursor FOR the query.
- 8. DESCRIBE the bind variables INTO the bind descriptor.
- Reset the number of placeholders to the number actually found by DESCRIBE.
- 10. Get values and allocate storage for the bind variables found by DESCRIBE.
- 11. OPEN the cursor USING the bind descriptor.
- 12. DESCRIBE the select list INTO the select descriptor.
- 13. Reset the number of select-list items to the number actually found by DESCRIBE.
- Reset the length and datatype of each select-list item for display purposes.
- 15. FETCH a row from the database INTO the allocated data buffers pointed to by the select descriptor.
- 16. Process the select-list values returned by FETCH.
- 17. Deallocate storage space used for the select-list items, placeholders, indicator variables, and descriptors.
- 18. CLOSE the cursor.



Some of these steps are unnecessary if the dynamic SQL statement contains a known number of select-list items or placeholders.

# 15.6 A Closer Look at Each Step

This section discusses each step in detail. At the end of this chapter is a Commented, full-length program illustrating Method 4.

With Method 4, you use the following sequence of embedded SQL statements:

```
EXEC SQL PREPARE statement_name
    FROM { :host_string | string_literal };
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name
    INTO bind_descriptor_name;
EXEC SQL OPEN cursor_name
    [USING DESCRIPTOR bind_descriptor_name];
EXEC SQL DESCRIBE [SELECT LIST FOR] statement_name
    INTO select_descriptor_name;
EXEC SQL FETCH cursor_name
    USING DESCRIPTOR select_descriptor_name;
EXEC SQL CLOSE cursor_name;
```

Scrollable cursors can also be used with Method 4. The following sequence of embedded SQL statements must be used for scrollable cursors.



If the number of select-list items in a dynamic query is known, you can omit DESCRIBE SELECT LIST and use the following Method 3 FETCH statement:

```
EXEC SQL FETCH cursor_name INTO host_variable_list;
```

Or, if the number of placeholders for bind variables in a dynamic SQL statement is known, you can omit DESCRIBE BIND VARIABLES and use the following Method 3 OPEN statement:

```
EXEC SQL OPEN cursor name [USING host variable list];
```

Next, you see how these statements allow your host program to accept and process a dynamic SQL statement using descriptors.



Several figures accompany the following discussion. To avoid cluttering the figures, it was necessary to do the following:

- Confine descriptor arrays to 3 elements
- Limit the maximum length of names to 5 characters
- Limit the maximum length of values to 10 characters

# 15.6.1 Declare a Host String

Your program needs a host variable to store the text of the dynamic SQL statement. The host variable (*select\_stmt* in our example) must be declared as a character string.

```
int emp_number;
VARCHAR emp_name[10];
VARCHAR select_stmt[120];
float bonus;
```

## 15.6.2 Declare the SQLDAs

In our example, instead of hardcoding the SQLDA data structure, you use INCLUDE to copy it into your program, as follows:

```
#include <sqlda.h>
```

Then, because the query might contain an unknown number of select-list items or placeholders for bind variables, you declare pointers to select and bind descriptors, as follows:

```
sqlda *select_des;
sqlda *bind des;
```



### 15.6.3 Allocate Storage Space for the Descriptors

Recall that you allocate storage space for a descriptor with the SQLSQLDAAlloc() library function. The syntax, using ANSI C notation, is:

```
SQLDA *SQLDAAlloc(dvoid *context, unsigned int max_vars, unsigned int max name, unsigned int max ind name);
```

The SQLSQLDAAlloc() function allocates the descriptor structure and the arrays addressed by the pointer variables V, L, T, and I.

If  $max\_name$  is nonzero, arrays addressed by the pointer variables S, M, and C are allocated. If  $max\_ind\_name$  is nonzero, arrays addressed by the pointer variables X, Y, and Z are allocated. No space is allocated if  $max\_name$  and  $max\_ind\_name$  are zero.

If SQLSQLDAAlloc() succeeds, it returns a pointer to the structure. If SQLSQLDAAlloc() fails, it returns a zero.

In our example, you allocate select and bind descriptors, as follows:

```
select_des = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, (size_t) 5, (size_t) 0);
bind_des = SQLSQLDAAlloc(SQL_SINGLE_RCTX, 3, (size_t) 5, (size_t) 4);
```

For select descriptors, always set *max\_ind\_name* to zero so that no space is allocated for the array addressed by *X*.

### 15.6.4 Set the Maximum Number to DESCRIBE

Next, you set the maximum number of select-list items or placeholders that can be DESCRIBEd, as follows:

```
select_des->N = 3;
bind_des->N = 3;
```

Figure 15-2 and Figure 15-3 represent the resulting descriptors.



In the select descriptor (Figure 15-2), the section for indicator-variable names is crossed out to show that it is not used.

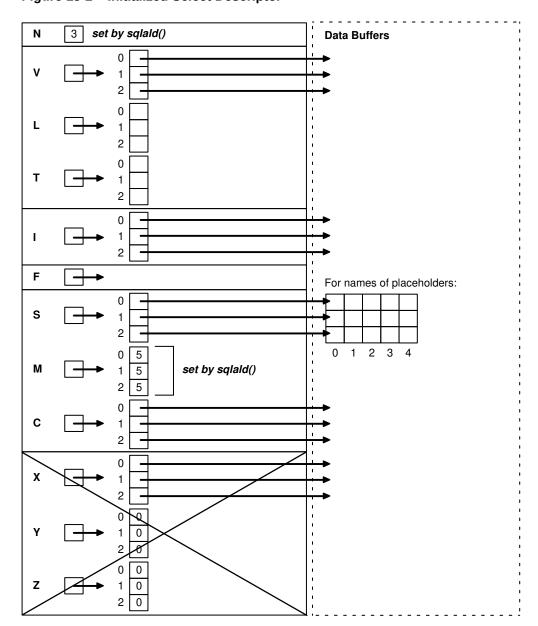


Figure 15-2 Initialized Select Descriptor

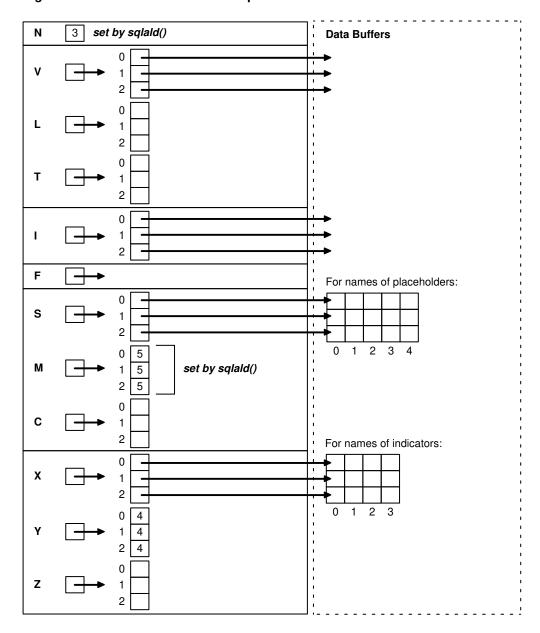


Figure 15-3 Initialized Bind Descriptor

# 15.6.5 Put the Query Text in the Host String

Continuing our example, you prompt the user for a SQL statement, then store the input string in *select\_stmt*, as follows:

```
printf("\n\nEnter SQL statement: ");
gets(select_stmt.arr);
select_stmt.len = strlen(select_stmt.arr);
```

We assume the user entered the following string:

"SELECT ename, empno, comm FROM emp WHERE comm < :bonus"

## 15.6.6 PREPARE the Query from the Host String

PREPARE parses the SQL statement and gives it a name. In our example, PREPARE parses the host string *select\_stmt* and gives it the name *sql\_stmt*, as follows:

EXEC SQL PREPARE sql stmt FROM :select stmt;

## 15.6.7 DECLARE a Cursor

DECLARE CURSOR defines a cursor by giving it a name and associating it with a specific SELECT statement.

To declare a cursor for static queries, you use the following syntax:

```
EXEC SQL DECLARE cursor name CURSOR FOR SELECT ...
```

To declare a cursor for *dynamic* queries, the statement name given to the dynamic query by PREPARE is substituted for the static query. In our example, DECLARE CURSOR defines a cursor named *emp\_cursor* and associates it with *sql\_stmt*, as follows:

EXEC SQL DECLARE emp cursor CURSOR FOR sql stmt;



You can declare a cursor for all dynamic SQL statements, not just queries. With non-queries, OPENing the cursor executes the dynamic SQL statement.

## 15.6.8 DESCRIBE the Bind Variables

DESCRIBE BIND VARIABLES puts descriptions of placeholders into a bind descriptor. In our example, DESCRIBE readies *bind\_des*, as follows:

```
EXEC SQL DESCRIBE BIND VARIABLES FOR sql stmt INTO bind des;
```

Note that *bind\_des* must *not* be prefixed with a colon.

The DESCRIBE BIND VARIABLES statement must follow the PREPARE statement but precede the OPEN statement.

Figure 15-4 shows the bind descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set *F* to the actual number of placeholders found in the processed SQL statement.

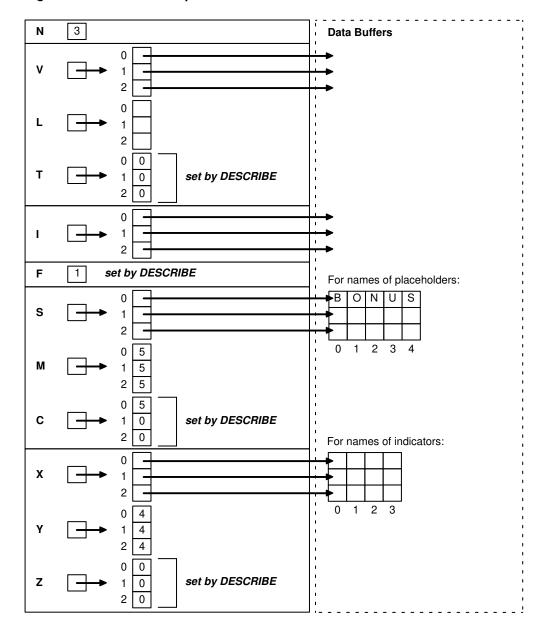


Figure 15-4 Bind Descriptor after the DESCRIBE

## 15.6.9 Reset Number of Placeholders

Next, you must reset the maximum number of placeholders to the number actually found by DESCRIBE, as follows:

bind\_des->N = bind\_des->F;

# 15.6.10 Get Values and Allocate Storage for Bind Variables

Your program must get values for the bind variables found in the SQL statement, and allocate memory for them. How the program gets the values is up to you. For example, they can be hardcoded, read from a file, or entered interactively.

In our example, a value must be assigned to the bind variable that replaces the placeholder *bonus* in the query WHERE clause. So, you choose to prompt the user for the value, then process it as follows:

```
for (i = 0; i < bind des -> F; i++)
    printf("\nEnter value of bind variable %.*s:\n? ",
        (int) bind_des->C[i], bind_des->S[i]);
    gets(hostval);
    /* Set length of value. */
    bind des->L[i] = strlen(hostval);
    /* Allocate storage for value and null terminator. */
    bind_des->V[i] = malloc(bind_des->L[i] + 1);
    /* Allocate storage for indicator value. */
    bind des->I[i] = (unsigned short *) malloc(sizeof(short));
    /* Store value in bind descriptor. */
    strcpy(bind des->V[i], hostval);
    /* Set value of indicator variable. */
    *(bind des->I[i]) = 0; /* or -1 if "null" is the value */
    /* Set datatype to STRING. */
    bind des->T[i] = 5;
```

Assuming that the user supplied a value of 625 for *bonus*, Figure 15-5 shows the resulting bind descriptor. Notice that the value is null-terminated.

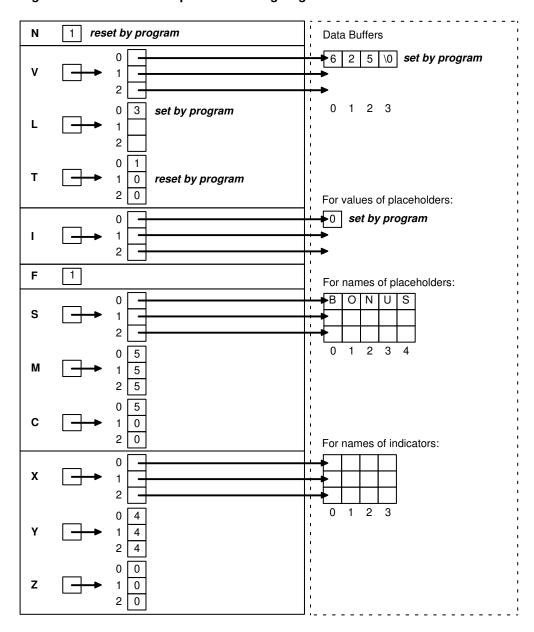


Figure 15-5 Bind Descriptor after Assigning Values

## 15.6.11 OPEN the Cursor

The OPEN statement used for dynamic queries is like that used for static queries except that the cursor is associated with a bind descriptor. Values determined at run time and stored in buffers addressed by elements of the bind descriptor arrays are used to evaluate the SQL statement. With queries, the values are also used to identify the active set.

In our example, OPEN associates *emp\_cursor* with *bind\_des*, as follows:

EXEC SQL OPEN emp cursor USING DESCRIPTOR bind des;

Remember, bind\_des must not be prefixed with a colon.

Then, OPEN executes the SQL statement. With queries, OPEN also identifies the active set and positions the cursor at the first row.

# 15.6.12 DESCRIBE the Select List

If the dynamic SQL statement is a query, the DESCRIBE SELECT LIST statement must follow the OPEN statement but precede the FETCH statement.

DESCRIBE SELECT LIST puts descriptions of select-list items in a select descriptor. In our example, DESCRIBE readies select des, as follows:

EXEC SQL DESCRIBE SELECT LIST FOR sql stmt INTO select des;

Accessing the Oracle data dictionary, DESCRIBE sets the length and datatype of each select-list value.

Figure 15-6 shows the select descriptor in our example after the DESCRIBE. Notice that DESCRIBE has set F to the actual number of items found in the query select list. If the SQL statement is not a query, F is set to zero.

Also notice that the NUMBER lengths are not usable yet. For columns defined as NUMBER, you must use the library function SQLNumberPrecV6() to extract precision and scale.



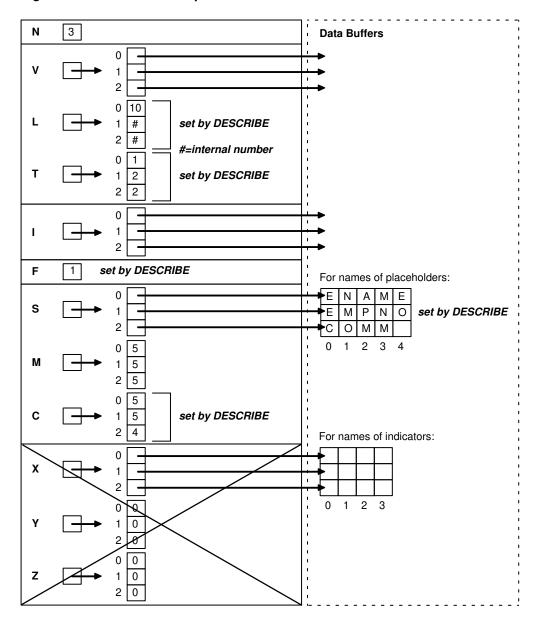


Figure 15-6 Select Descriptor after the DESCRIBE

#### **Related Topics**

Coercing Datatypes

## 15.6.13 Reset Number of Select-List Items

Next, you must reset the maximum number of select-list items to the number actually found by DESCRIBE, as follows:

select\_des->N = select\_des->F;

## 15.6.14 Reset Length/Datatype of Each Select-list Item

In our example, before FETCHing the select-list values, you allocate storage space for them using the library function malloc(). You also reset some elements in the length and datatype arrays for display purposes.

```
for (i=0; i < select des -> F; i++)
    /* Clear null bit. */
    SQLColumnNullCheck(SQL SINGLE RCTX, (unsigned short *)&(select des->T[i]),
         (unsigned short *)&(select des->T[i]), &nullok);
    /* Reset length if necessary. */
    switch(select des->T[i])
        case 1: break;
        case 2: SQLNumberPrecV6(SQL SINGLE RCTX, (unsigned long *)
                      &(select des->L[i]), &prec, &scal);
                 if (prec == 0) prec = 40;
                  select des->L[i] = prec + 2;
                 if (scal < 0) select des->L[i] += -scal;
        case 8: select des->L[i] = 240;
                 break:
        case 11: select des->L[i] = 18;
                 break;
        case 12: select des->L[i] = 9;
                 break;
        case 23: break;
        case 24: select des->L[i] = 240;
                 break;
    /* Allocate storage for select-list value. */
    select des->V[i] = malloc(select des->L[i]+1);
    /* Allocate storage for indicator value. */
    select des->I[i] = (short *)malloc(sizeof(short *));
    /* Coerce all datatypes except LONG RAW to STRING. */
    if (select des->T[i] != 24) select des->T[i] = 5;
}
```

Figure 15-7 shows the resulting select descriptor. Notice that the NUMBER lengths are now usable and that all the datatypes are STRING. The lengths in  $\mathbb{L}[1]$  and  $\mathbb{L}[2]$  are 6 and 9 because we increased the DESCRIBEd lengths of 4 and 7 by 2 to allow for a possible sign and decimal point.

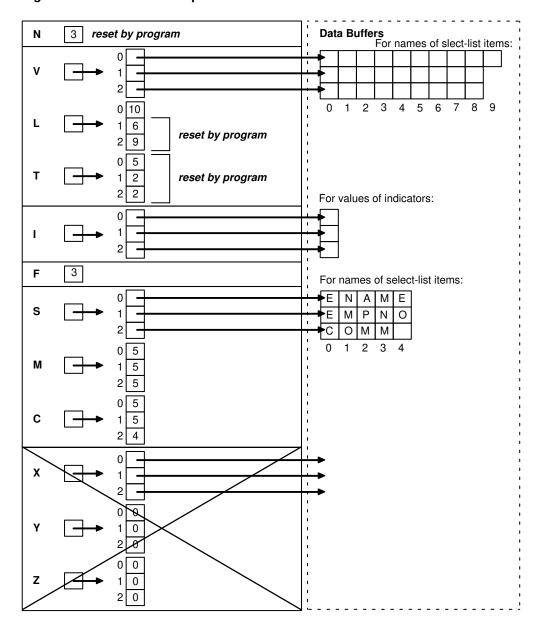


Figure 15-7 Select Descriptor before the FETCH

## 15.6.15 FETCH Rows from the Active Set

FETCH returns a row from the active set, stores select-list values in the data buffers, and advances the cursor to the next row in the active set. If there are no more rows, FETCH sets *sqlca.sqlcode* to the "no data found" Oracle error code. In our example, FETCH returns the values of columns ENAME, EMPNO, and COMM to *select\_des*, as follows:

EXEC SQL FETCH emp\_cursor USING DESCRIPTOR select\_des;

Figure 15-8 shows the select descriptor in our example after the FETCH. Notice that Oracle has stored the select-list and indicator values in the data buffers addressed by the elements of V and I.

For output buffers of datatype 1, Oracle, using the lengths stored in the *L* array, left-justifies CHAR or VARCHAR2 data and right-justifies NUMBER data. For output buffer of type 5 (STRING), Oracle left-justifies and null terminates CHAR, VARCHAR2, and NUMBER data.

The value 'MARTIN' was retrieved from a VARCHAR2(10) column in the EMP table. Using the length in  ${\tt L[0]}$ , Oracle left-justifies the value in a 10-byte field, filling the buffer.

The value 7654 was retrieved from a NUMBER(4) column and coerced to '7654'. However, the length in  ${\tt L[1]}$  was increased by 2 to allow for a possible sign and decimal point. So, Oracle left-justifies and null terminates the value in a 6-byte field.

The value 482.50 was retrieved from a NUMBER(7,2) column and coerced to '482.50'. Again, the length in  $\mathbb{L}[2]$  was increased by 2. So, Oracle left-justifies and null terminates the value in a 9-byte field.

## 15.6.16 Get and Process Select-List Values

After the FETCH, your program can process the returned values. In our example, values for columns ENAME, EMPNO, and COMM are processed.



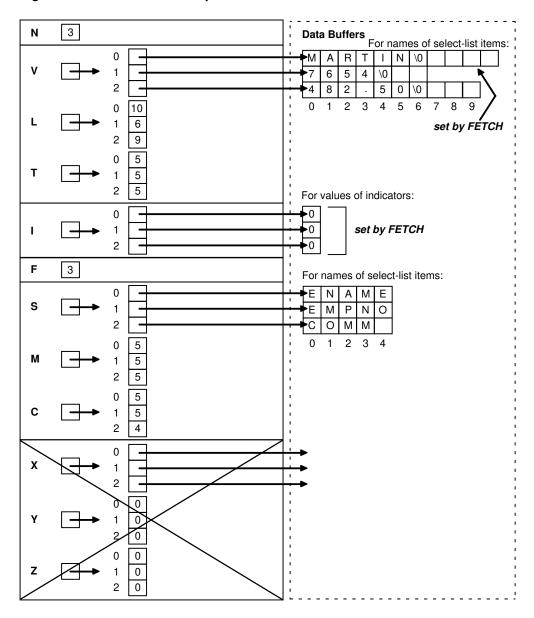


Figure 15-8 Selected Descriptor after the FETCH

## 15.6.17 Deallocate Storage

You use the free() library function to deallocate the storage space allocated by malloc(). The syntax is as follows:

```
free(char *pointer);
```

In our example, you deallocate storage space for the values of the select-list items, bind variables, and indicator variables, as follows:

```
{
    free(bind_des->V[i]);
    free(bind_des->I[i]);
}
```

You deallocate storage space for the descriptors themselves with the SQLSQLDAFree() library function, using the following syntax:

```
SQLSQLDAFree(context, descriptor_name);
```

The descriptor must have been allocated using SQLSQLDAAlloc(). Otherwise, the results are unpredictable.

In our example, you deallocate storage space for the select and bind descriptors as follows:

```
SQLSQLDAFree(SQL_SINGLE_RCTX, select_des);
SQLSQLDAFree(SQL SINGLE RCTX, bind des);
```

## 15.6.18 CLOSE the Cursor

CLOSE disables the cursor. In our example, CLOSE disables *emp\_cursor* as follows:

```
EXEC SQL CLOSE emp cursor;
```

## 15.6.19 About Using Host Arrays

To use input or output host arrays with Method 4, you must use the optional FOR clause to tell Oracle the size of your host array.

You must set descriptor entries for the ith select-list item or bind variable using the syntax

```
V[i] = array_address;
L[i] = element size;
```

where *array\_address* is the address of the host array, and *element\_size* is the size of one array element.

Then, you must use a FOR clause in the EXECUTE or FETCH statement (whichever is appropriate) to tell Oracle the number of array elements you want to process. This procedure is necessary because Oracle has no other way of knowing the size of your host array.

In the complete program example later, three input host arrays are used to INSERT rows into the EMP table. EXECUTE can be used for Data Manipulation Language statements other than queries with Method 4.



```
SQLDA
       *binda;
        names[ARRAY_SIZE][NAME_SIZE];
char
        numbers[ARRAY SIZE], depts[ARRAY SIZE];
int
/\star Declare and initialize indicator vars. for empno and deptno columns \star/
      ind_empno[ARRAY_SIZE] = {0,0,0,0,0};
      ind dept[ARRAY SIZE] = \{0,0,0,0,0,0\};
main()
    EXEC SQL WHENEVER SQLERROR GOTO sql error;
/* Connect */
    EXEC SQL CONNECT :username;
    printf("Connected.\n");
/* Allocate the descriptors and set the N component.
   This must be done before the DESCRIBE. */
   binda = SQLSQLDAAlloc(SQL SINGLE RCTX, 3, NAME SIZE, INAME SIZE);
   binda->N = 3;
/* Prepare and describe the SQL statement. */
    EXEC SQL PREPARE stmt FROM :sql stmt;
    EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO binda;
/* Initialize the descriptors. */
    binda->V[0] = (char *) numbers;
    binda->L[0] = (long) sizeof (int);
    binda \rightarrow T[0] = 3;
    binda->I[0] = ind_empno;
    binda->V[1] = (char *) names;
    binda->L[1] = (long) NAME SIZE;
    binda->T[1] = 1;
    binda \rightarrow I[1] = (short *)0;
    binda->V[2] = (char *) depts;
    binda->L[2] = (long) sizeof (int);
    binda \rightarrow T[2] = 3;
    binda->I[2] = ind dept;
/* Initialize the data buffers. */
    strcpy(&names[0] [0], "ALLISON");
    numbers[0] = 1014;
    depts[0] = 30;
    strcpy(&names[1] [0], "TRUSDALE");
    numbers[1] = 1015;
    depts[1] = 30;
    strcpy(&names[2] [0], "FRAZIER");
    numbers[2] = 1016;
    depts[2] = 30;
    strcpy(&names[3] [0], "CARUSO");
    numbers[3] = 1017;
                            /* set indicator to -1 to insert NULL */
    ind_dept[3] = -1;
                            /* value in depts[3] is ignored */
    depts[3] = 30;
    strcpy(&names[4] [0], "WESTON");
    numbers[4] = 1018;
```

```
depts[4] = 30;

/* Do the INSERT. */
    printf("Adding to the Sales force...\n");

    EXEC SQL FOR :array_size
    EXECUTE stmt USING DESCRIPTOR binda;

/* Print rows-processed count. */
    printf("%d rows inserted.\n\n", sqlca.sqlerrd[2]);
    EXEC SQL COMMIT RELEASE;
    exit(0);

sql_error:
/* Print Oracle error message. */
    printf("\n%.70s", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

#### **Related Topics**

Host Arrays

## 15.6.20 sample12.pc

A simple dynamic SQL example using array fetches is shown in file sample12.pc in the demo directory.

# 15.7 About Using Domain Metadata

An application can get define metadata for a dynamic SQL statement by using the DESCRIBE SELECT LIST command, which places metadata information into the SQL descriptor area.

```
EXEC SQL DESCRIBE SELECT LIST FOR statement_name INTO
    select_descriptor_name;
```

The SQLDA structure is extended to store the domain name and domain length.



Declaring a SQLDA

The CP element provides the following SQLColumnHasDomain() function to check if a column has a domain. This function sets the hasdomain argument to true if the column has domain constraint. Otherwise, it is set to false.

```
void SQLColumnHasDomain(
  void *uga,
  long long cp,
  boolean *hasdomain);
```

#### Parameters:

· uga (IN) - runtime context

- cp (IN) column properties
- hasdomain (OUT) True or False

#### The following is an example:

```
char *select_stmt = "SELECT empId, empEmail FROM empTable";
EXEC SQL DESCRIBE SELECT LIST FOR select_stmt INTO sel_des;
SQLColumnHasDomain(NULL, sel_des->CP[1], &hasdomain);
if(hasdomain)
  printf("Domain %.*s\n", sel_des->DCL[1], sel_des->D[1]);
```

# 15.8 Example Program: Dynamic SQL Method 4

This program shows the basic steps required to use dynamic SQL with Method 4. After connecting to Oracle, the program:

- Allocates memory for the descriptors using SQLSQLDAAlloc()
- Prompts the user for a SQL statement
- PREPAREs the statement
- DECLAREs a cursor
- Checks for any bind variables using DESCRIBE BIND
- OPENs the cursor
- DESCRIBEs any select-list items.

If the input SQL statement is a query, the program FETCHes each row of data, then CLOSEs the cursor. This program is available on-line in the demo directory, in the file sample10.pc.

```
Sample Program 10: Dynamic SQL Method 4
This program connects you to ORACLE using your username and
password, then prompts you for a SQL statement. You can enter
any legal SQL statement. Use regular SQL syntax, not embedded SQL.
Your statement will be processed. If it is a query, the rows
fetched are displayed.
You can enter multiline statements. The limit is 1023 characters.
This sample program only processes up to MAX ITEMS bind variables and
MAX ITEMS select-list items. MAX ITEMS is #defined to be 40.
#include <stdio.h>
#include <string.h>
#include <setjmp.h>
#include <sqlda.h>
#include <stdlib.h>
#include <sqlcpr.h>
/* Maximum number of select-list items or bind variables. */
#define MAX ITEMS
/* Maximum lengths of the names of the
   select-list items or indicator variables. */
#define MAX VNAME LEN
                       30
#define MAX INAME LEN
#ifndef NULL
```

```
#define NULL 0
#endif
/* Prototypes */
#if defined(__STDC_
  void sql error(void);
  int oracle connect(void);
  int alloc_descriptors(int, int, int);
  int get dyn statement(void);
  void set_bind_variables(void);
  void process_select_list(void);
  void help(void);
#else
  void sql_error(/*_ void _*/);
  int oracle_connect(/*_ void _*/);
  int alloc descriptors(/* int, int, int */);
  int get dyn statement(/* void */);
  void set bind variables(/* void -*/);
  void process select list(/* void */);
  void help(/*_ void _*/);
#endif
char *dml commands[] = {"SELECT", "select", "INSERT", "insert",
                        "UPDATE", "update", "DELETE", "delete"};
EXEC SQL INCLUDE sqlda;
EXEC SQL INCLUDE sqlca;
EXEC SQL BEGIN DECLARE SECTION;
          dyn statement[1024];
    EXEC SQL VAR dyn statement IS STRING(1024);
EXEC SQL END DECLARE SECTION;
SQLDA *bind dp;
SQLDA *select_dp;
/* Define a buffer to hold longjmp state info. */
jmp_buf jmp_continue;
/* A global flag for the error routine. */
int parse flag = 0;
void main()
{
    int i;
    /* Connect to the database. */
    if (oracle connect() != 0)
        exit(1);
    /* Allocate memory for the select and bind descriptors. */
    if (alloc descriptors (MAX ITEMS, MAX VNAME LEN, MAX INAME LEN) != 0)
        exit(1);
    /* Process SQL statements. */
    for (;;)
        (void) setjmp(jmp_continue);
        /* Get the statement. Break on "exit". */
        if (get dyn statement() != 0)
            break;
```

```
/\star Prepare the statement and declare a cursor. \star/
    EXEC SQL WHENEVER SQLERROR DO sql_error();
                       /* Set a flag for sql_error(). */
    parse flag = 1;
    EXEC SQL PREPARE S FROM : dyn statement;
                       /* Unset the flag. */
    parse flag = 0;
    EXEC SQL DECLARE C CURSOR FOR S;
    /* Set the bind variables for any placeholders in the
       SQL statement. */
    set_bind_variables();
    /* Open the cursor and execute the statement.
     ^{\star} If the statement is not a query (SELECT), the
     ^{\star} statement processing is completed after the
     * OPEN.
     * /
    EXEC SQL OPEN C USING DESCRIPTOR bind dp;
    /* Call the function that processes the select-list.
     * If the statement is not a query, this function
     * just returns, doing nothing.
    process select list();
    /* Tell user how many rows processed. */
    for (i = 0; i < 8; i++)
    {
       if (strncmp(dyn statement, dml commands[i], 6) == 0)
           printf("\n\n\%d row\%c processed.\n", sqlca.sqlerrd[2],
                   sqlca.sqlerrd[2] == 1 ? '\0' : 's');
           break:
       }
        /* end of for(;;) statement-processing loop */
/* When done, free the memory allocated for
  pointers in the bind and select descriptors. */
for (i = 0; i < MAX ITEMS; i++)
    if (bind dp->V[i] != (char *) 0)
       free(bind dp->V[i]);
    free(bind dp->I[i]); /* MAX ITEMS were allocated. */
    if (select dp->V[i] != (char *) 0)
        free(select dp->V[i]);
    free(select dp->I[i]); /* MAX ITEMS were allocated. */
/* Free space used by the descriptors themselves. */
SQLSQLDAFree ( SQL SINGLE RCTX, bind dp);
SQLSQLDAFree( SQL_SINGLE_RCTX, select_dp);
EXEC SQL WHENEVER SQLERROR CONTINUE;
/* Close the cursor. */
EXEC SQL CLOSE C;
EXEC SQL COMMIT WORK RELEASE;
puts("\nHave a good day!\n");
```

```
EXEC SQL WHENEVER SQLERROR DO sql error();
    return;
int oracle connect()
    EXEC SQL BEGIN DECLARE SECTION;
       VARCHAR username[128];
       VARCHAR password[32];
    EXEC SQL END DECLARE SECTION;
    printf("\nusername: ");
    fgets((char *) username.arr, sizeof username.arr, stdin);
    username.arr[strlen((char *) username.arr)-1] = '\0';
   username.len = (unsigned short)strlen((char *) username.arr);
   printf("password: ");
   fgets((char *) password.arr, sizeof password.arr, stdin);
   password.arr[strlen((char *) password.arr) - 1] = '\0';
   password.len = (unsigned short)strlen((char *) password.arr);
   EXEC SQL WHENEVER SQLERROR GOTO connect error;
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
    printf("\nConnected to ORACLE as user %s.\n", username.arr);
   return 0;
connect error:
   fprintf(stderr, "Cannot connect to ORACLE as user %s\n", username.arr);
   return -1;
}
* Allocate the BIND and SELECT descriptors using SQLSQLDAAlloc().
* Also allocate the pointers to indicator variables
* in each descriptor. The pointers to the actual bind
* variables and the select-list items are realloc'ed in
* the set bind variables() or process select list()
* routines. This routine allocates 1 byte for select dp->V[i]
* and bind_dp->V[i], so the realloc will work correctly.
alloc_descriptors(size, max_vname_len, max_iname_len)
int size;
int max vname len;
int max iname len;
   int i;
    * The first SQLSQLDAAlloc parameter is the runtime context.
    * The second parameter determines the maximum number of
    * array elements in each variable in the descriptor. In
     * other words, it determines the maximum number of bind
     * variables or select-list items in the SQL statement.
```

```
* The third parameter determines the maximum length of
     ^{\star} strings used to hold the names of select-list items
     * or placeholders. The maximum length of column
     * names in ORACLE is 30, but you can allocate more or less
     * as needed.
     ^{\star} The fourth parameter determines the maximum length of
     * strings used to hold the names of any indicator
     * variables. To follow ORACLE standards, the maximum
     * length of these should be 30. But, you can allocate
     * more or less as needed.
    if ((bind dp =
       SQLSQLDAAlloc(SQL SINGLE RCTX, size, max vname len, max iname len)) ==
         (SQLDA *) 0)
        fprintf(stderr,
            "Cannot allocate memory for bind descriptor.");
        return -1; /* Have to exit in this case. */
    if ((select dp =
        SQLSQLDAAlloc (SQL SINGLE RCTX, size, max vname len, max iname len)) ==
           (SQLDA *) 0)
        fprintf(stderr,
            "Cannot allocate memory for select descriptor.");
        return -1;
    select dp->N = MAX ITEMS;
    /* Allocate the pointers to the indicator variables, and the
      actual data. */
    for (i = 0; i < MAX ITEMS; i++) {
       bind_dp->I[i] = (short *) malloc(sizeof (short));
        select_dp->I[i] = (short *) malloc(sizeof(short));
        bind dp \rightarrow V[i] = (char *) malloc(1);
        select dp->V[i] = (char *) malloc(1);
    }
   return 0;
int get_dyn_statement()
   char *cp, linebuf[256];
   int iter, plsql;
    for (plsql = 0, iter = 1; ;)
        if (iter == 1)
            printf("\nSQL> ");
            dyn_statement[0] = '\0';
        fgets(linebuf, sizeof linebuf, stdin);
```

}

```
cp = strrchr(linebuf, '\n');
        if (cp && cp != linebuf)
            *cp = ' ';
        else if (cp == linebuf)
            continue;
        if ((strncmp(linebuf, "EXIT", 4) == 0) ||
            (strncmp(linebuf, "exit", 4) == 0))
        {
            return -1;
        else if (linebuf[0] == '?' ||
            (strncmp(linebuf, "HELP", 4) == 0) ||
            (strncmp(linebuf, "help", 4) == 0))
        {
            help();
            iter = 1;
            continue;
        if (strstr(linebuf, "BEGIN") ||
            (strstr(linebuf, "begin")))
        {
            plsql = 1;
        strcat(dyn statement, linebuf);
        if ((plsql && (cp = strrchr(dyn_statement, '/'))) ||
            (!plsql && (cp = strrchr(dyn statement, ';'))))
            *cp = '\0';
            break;
        }
        else
        {
            iter++;
            printf("%3d ", iter);
    }
    return 0;
void set bind variables()
    int i, n;
    char bind var[64];
    /* Describe any bind variables (input host variables) */
    EXEC SQL WHENEVER SQLERROR DO sql error();
    bind dp->N = MAX ITEMS; /* Initialize count of array elements. */
    EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind dp;
    /\star If F is negative, there were more bind variables
       than originally allocated by SQLSQLDAAlloc(). */
    if (bind_dp -> F < 0)
        printf ("\nToo many bind variables (%d), maximum is %d\n.",
```

```
-bind dp->F, MAX ITEMS);
   return:
/* Set the maximum number of array elements in the
  descriptor to the number found. */
bind dp \rightarrow N = bind dp \rightarrow F;
/* Get the value of each bind variable as a
 * character string.
 * C[i] contains the length of the bind variable
       name used in the SQL statement.
 * S[i] contains the actual name of the bind variable
       used in the SQL statement.
 * L[i] will contain the length of the data value
        entered.
 * V[i] will contain the address of the data value
        entered.
 * T[i] is always set to 1 because in this sample program
        data values for all bind variables are entered
        as character strings.
        ORACLE converts to the table value from CHAR.
 * I[i] will point to the indicator value, which is
        set to -1 when the bind variable value is "null".
 */
for (i = 0; i < bind dp->F; i++)
   printf ("\nEnter value for bind variable %.*s: ",
           (int)bind_dp->C[i], bind_dp->S[i]);
    fgets(bind_var, sizeof bind_var, stdin);
    /* Get length and remove the new line character. */
   n = strlen(bind var) - 1;
    /* Set it in the descriptor. */
   bind dp \rightarrow L[i] = n;
    /* (re-)allocate the buffer for the value.
       SQLSQLDAAlloc() reserves a pointer location for
       V[i] but does not allocate the full space for
       the pointer. */
    bind dp->V[i] = (char *) realloc(bind dp->V[i],
                     (bind dp->L[i] + 1);
    /* And copy it in. */
    strncpy(bind dp->V[i], bind var, n);
    /* Set the indicator variable's value. */
    if ((strncmp(bind dp->V[i], "NULL", 4) == 0) |
            (strncmp(bind dp->V[i], "null", 4) == 0))
        *bind dp->I[i] = -1;
   else
        *bind_dp->I[i] = 0;
    /* Set the bind datatype to 1 for CHAR. */
   bind dp->T[i] = 1;
```

```
return;
void process select list()
   int i, null ok, precision, scale;
    if ((strncmp(dyn statement, "SELECT", 6) != 0) &&
        (strncmp(dyn statement, "select", 6) != 0))
       select dp \rightarrow F = 0;
       return;
    /* If the SQL statement is a SELECT, describe the
       select-list items. The DESCRIBE function returns
       their names, datatypes, lengths (including precision
       and scale), and NULL/NOT NULL statuses. */
    select dp->N = MAX ITEMS;
    EXEC SQL DESCRIBE SELECT LIST FOR S INTO select dp;
    /\star If F is negative, there were more select-list
       items than originally allocated by SQLSQLDAAlloc(). */
    if (select dp->F < 0)
       printf ("\nToo many select-list items (%d), maximum is %d\n",
                -(select dp->F), MAX ITEMS);
       return;
    /* Set the maximum number of array elements in the
      descriptor to the number found. */
    select_dp->N = select_dp->F;
    /* Allocate storage for each select-list item.
       SQLNumberPrecV6() is used to extract precision and scale
       from the length (select_dp->L[i]).
       sqlcolumnNullCheck() is used to reset the high-order bit of
       the datatype and to check whether the column
       is NOT NULL.
       CHAR
               datatypes have length, but zero precision and
               scale. The length is defined at CREATE time.
       NUMBER datatypes have precision and scale only if
               defined at CREATE time. If the column
               definition was just NUMBER, the precision
               and scale are zero, and you must allocate
               the required maximum length.
       DATE
              datatypes return a length of 7 if the default
               format is used. This should be increased to
               9 to store the actual date character string.
               If you use the TO CHAR function, the maximum
               length could be 75, but will probably be less
```

```
(you can see the effects of this in SQL*Plus).
   ROWID
          datatype always returns a fixed length of 18 if
           coerced to CHAR.
   LONG and
   LONG RAW datatypes return a length of 0 (zero),
           so you need to set a maximum. In this example,
           it is 240 characters.
printf ("\n");
for (i = 0; i < select dp->F; i++)
   char title[MAX VNAME LEN];
    /* Turn off high-order bit of datatype (in this example,
      it does not matter if the column is NOT NULL). */
    SQLColumnNullCheck ((unsigned short *)&(select dp->T[i]),
         (unsigned short *)&(select dp->T[i]), &null ok);
   switch (select dp->T[i])
        case 1 : /* CHAR datatype: no change in length
                     needed, except possibly for TO CHAR
                     conversions (not handled here). */
            break;
        case 2 : /* NUMBER datatype: use SQLNumberPrecV6() to
                     extract precision and scale. */
            SQLNumberPrecV6( SQL SINGLE RCTX,
                   (unsigned long *)&(select dp->L[i]), &precision, &scale);
                  /* Allow for maximum size of NUMBER. */
            if (precision == 0) precision = 40;
                  /* Also allow for decimal point and
                     possible sign. */
            /* convert NUMBER datatype to FLOAT if scale > 0,
               INT otherwise. */
            if (scale > 0)
                select dp->L[i] = sizeof(float);
            else
                select dp->L[i] = sizeof(int);
            break;
        case 8 : /* LONG datatype */
            select dp->L[i] = 240;
            break;
        case 11 : /* ROWID datatype */
            select dp \rightarrow L[i] = 18;
            break;
        case 12 : /* DATE datatype */
            select dp \rightarrow L[i] = 9;
            break;
        case 23 : /* RAW datatype */
            break;
        case 24 : /* LONG RAW datatype */
            select dp \rightarrow L[i] = 240;
            break;
    }
```

```
/\star Allocate space for the select-list data values.
      SQLSQLDAAlloc() reserves a pointer location for
      V[i] but does not allocate the full space for
      the pointer. */
     if (select dp->T[i] != 2)
       select dp->V[i] = (char *) realloc(select dp->V[i],
                                select dp->L[i] + 1);
      select_dp->V[i] = (char *) realloc(select_dp->V[i],
                                select dp->L[i]);
    /* Print column headings, right-justifying number
       column headings. */
    /* Copy to temporary buffer in case name is null-terminated */
   memset(title, ' ', MAX VNAME LEN);
   strncpy(title, select dp->S[i], select dp->C[i]);
   if (select dp->T[i] == 2)
      if (scale > 0)
        printf ("%.*s ", select_dp->L[i]+3, title);
        printf ("%.*s ", select dp->L[i], title);
   else
      printf("%-.*s ", select dp->L[i], title);
   /* Coerce ALL datatypes except for LONG RAW and NUMBER to
       character. */
    if (select dp->T[i] != 24 && select dp->T[i] != 2)
        select dp \rightarrow T[i] = 1;
    /* Coerce the datatypes of NUMBERs to float or int depending on
      the scale. */
    if (select_dp->T[i] == 2)
      if (scale > 0)
         select dp->T[i] = 4; /* float */
         select dp->T[i] = 3; /* int */
printf ("\n\n");
/* FETCH each row selected and print the column values. */
EXEC SQL WHENEVER NOT FOUND GOTO end select loop;
for (;;)
   EXEC SQL FETCH C USING DESCRIPTOR select dp;
    /* Since each variable returned has been coerced to a
      character string, int, or float very little processing
      is required here. This routine just prints out the
      values on the terminal. */
   for (i = 0; i < select dp->F; i++)
       if (*select dp->I[i] < 0)
            if (select dp->T[i] == 4)
             printf ("%-*c ",(int)select dp->L[i]+3, ' ');
             printf ("%-*c ",(int)select_dp->L[i], ' ');
        else
                                         /* int datatype */
            if (select dp->T[i] == 3)
              printf ("%*d ", (int)select dp->L[i],
```

```
*(int *)select dp->V[i]);
                else if (select dp->T[i] == 4)
                                                /* float datatype */
                  printf ("%*.2f ", (int)select_dp->L[i],
                                 *(float *)select dp->V[i]);
                                              /* character string */
                  printf ("%-*.*s ", (int)select dp->L[i],
                (int)select dp->L[i], select dp->V[i]);
       printf ("\n");
end select loop:
   return;
void help()
   puts("\n\nEnter a SQL statement or a PL/SQL block at the SQL> prompt.");
   puts ("Statements can be continued over several lines, except");
   puts("within string literals.");
   puts("Terminate a SOL statement with a semicolon.");
   puts ("Terminate a PL/SQL block (which can contain embedded semicolons)");
   puts("with a slash (/).");
   puts("Typing \"exit\" (no semicolon needed) exits the program.");
   puts("You typed \"?\" or \"help\" to get this message.\n\n");
void sql_error()
    /* ORACLE error handler */
   printf ("\n\n%.70s\n", sqlca.sqlerrm.sqlerrmc);
   if (parse flag)
       printf
        ("Parse error at character offset %d in SQL statement.\n",
           sqlca.sqlerrd[4]);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK;
    longjmp(jmp continue, 1);
```

# 15.9 Sample Program : Dynamic SQL Method 4 using Scrollable Cursors

This demo program describes the scrollable cursor feature used with oracle dynamic method 4. This program is available on-line in the file scdemol.pc in your demo directory.

#### scdemo1.pc

```
/*

* This demo program exhibits the scrollable cursor feature

* used with oracle dynamic method 4. The scrollable cursor

* feature can also be used with ANSI dynamic method 4.

* This program takes as argument the username/passwd. Once

* logged in, it prompts for a select query. It then prompts

* for the orientation and prints the results of the query.
```



```
* Before executing this example, make sure that the HR
 * schema exists.
#include <oci.h>
#include <stdio.h>
#include <sqlca.h>
#include <sqlda.h>
#include <string.h>
#include <ctype.h>
#include <sqlcpr.h>
#include <stdlib.h>
#include <setjmp.h>
#define MAX SELECT ITEMS
                            200
#define MAX CHARS
/* Maximum size of a select-list item name */
#define MAX NAME SIZE
SQLDA *selda;
SQLDA *bind des;
jmp buf beginEnv;
jmp buf loopEnv;
/* Data buffer */
char c_data[MAX_SELECT_ITEMS][MAX_CHARS];
char username[60];
char stmt[500];
char stmt2[500];
/* Print the generic error message & exit */
void sql_error()
   char msgbuf[512];
   size_t msgbuf_len, msg_len;
    msqbuf len = sizeof(msqbuf);
    sqlglm(msgbuf, &msgbuf len, &msg len);
   printf ("\n\n%.*s\n", msg len, msgbuf);
   EXEC SQL WHENEVER SQLERROR CONTINUE;
   EXEC SQL ROLLBACK WORK RELEASE;
    exit(EXIT FAILURE);
/* Print the error message and continue to query
  the user */
void sql_loop_error()
   char msgbuf[512];
   size_t msgbuf_len, msg_len;
   int code = sqlca.sqlcode;
    msgbuf len = sizeof(msgbuf);
    sqlglm(msgbuf, &msgbuf len, &msg len);
```

```
printf ("\n%.*s\n", msg len, msgbuf);
    printf("The error code is %d\n", sqlca.sqlcode);
    if(code!=0)
      longjmp(beginEnv, 1);
    longjmp(loopEnv, 1);
/\star FETCH has returned the "no data found" error code.
   This means that either we have reached the end of
   the active set or the offset refers to a row beyond the
   active set */
void no data found()
  printf("\nNo Data available at the specified offset\n");
  longjmp(loopEnv, 1);
void main(int argc, char *argv[])
    int i, n;
   int sli;
                /* select-list item */
   int offset;
   int contFlag;
    char bindVar[20];
    char *u, temp[3];
    char choice;
    /* Error Handler */
    EXEC SQL WHENEVER SQLERROR DO sql error();
    if (argc == 1)
      printf("Logging in as default user hr\n");
      strcpy(username, "hr/hr");
    }
    else
      strcpy(username, argv[1]);
    /* Establish a connection to the data base */
    EXEC SQL CONNECT :username;
    u = username;
    while (*++u != '/');
    *u = ' \setminus 0';
    /* Error Handler */
    EXEC SQL WHENEVER SQLERROR DO sql loop error();
    for (;;)
      setjmp(beginEnv);
      printf("[%s] SQL > ", username);
      gets(stmt);
      if (!strlen(stmt))
        continue;
      for (i=0 ; i < strlen(stmt) ; i++)
           stmt2[i] = tolower(stmt[i]);
      stmt2[i]=0;
      if(!strcmp(stmt2, "exit"))
```

```
selda = SQLSQLDAAlloc(SQL_SINGLE_RCTX, MAX_SELECT_ITEMS, MAX_NAME_SIZE, 0);
bind des = SQLSQLDAAlloc(SQL SINGLE RCTX, MAX SELECT ITEMS,
                         MAX_NAME_SIZE, 30);
/* prepare a sql statement for the query*/
EXEC SQL PREPARE S FROM :stmt;
/* Declare a cursor as scrollable */
EXEC SOL DECLARE C SCROLL CURSOR FOR S;
for (i=0; i<MAX SELECT ITEMS; i++)
 bind des->I[i] = (short *) malloc(sizeof (short));
 bind des->V[i] = (char *) malloc(1);
bind des->N = MAX SELECT ITEMS;
EXEC SQL DESCRIBE BIND VARIABLES FOR S INTO bind des;
/* set up the bind variables */
if (bind des->F < 0)
 printf("Bind descriptor, value exceeds the limit\n");
  exit(-1);
bind des->N = bind des->F;
for (i=0; i< bind des->F; i++)
  printf("Enter the value for bind variable %.*s: ",
         (int)bind_des->C[i], bind_des->S[i]);
  fgets(bindVar, sizeof(bindVar), stdin);
  n = strlen(bindVar) - 1;
  bind des->L[i] = n;
  bind des->V[i] = (char *) realloc(bind des->V[i],
                   (bind des->L[i] +1));
  strncpy(bind des->V[i], bindVar, n);
  if ((strncmp(bind des->V[i], "NULL", 4) == 0) ||
        (strncmp(bind des->V[i], "null", 4) == 0))
    *bind des \rightarrowI[i] = -1;
    *bind des \rightarrowI[i] = 0;
 bind des->T[i] = 1;
/* open the cursor */
EXEC SQL OPEN C USING DESCRIPTOR bind des;
EXEC SQL DESCRIBE SELECT LIST FOR S INTO selda;
if (selda->F < 0)
 printf("Select descriptor, value exceeds the limit\n");
  exit(-1);
selda->N = selda->F;
```

break;

```
for (sli = 0; sli < selda->N; sli++)
          /* Set addresses of heads of the arrays
            in the V element. */
          selda->V[sli] = c data[sli];
          /* Convert everything to varchar on output. */
          selda->T[sli] = 1;
          /* Set the maximum lengths. */
          selda->L[sli] = MAX CHARS;
     contFlag = 'Y';
      setjmp(loopEnv);
     while(1)
       while(contFlag != 'Y' && contFlag != 'N')
       printf("\nContinue with the current fetch? [y/n] :");
       contFlag = toupper(getchar());
       /* To flush the input buffer */
       getchar();
       }
       if(contFlag != 'Y')
         break:
       contFlag = 'x';
       printf("\n\nEnter the row number to be fetched \n");
       printf("1.ABSOLUTE\n");
       printf("2.RELATIVE\n");
       printf("3.FIRST \n");
       printf("4.NEXT \n");
       printf("5.PREVIOUS \n");
       printf("6.LAST \n");
       printf("7.CURRENT \n");
       printf("Enter your choice --> ");
       scanf("%c",&choice);
       /* To flush the input buffer */
                                             getchar();
                                                                 EXEC SQL WHENEVER NOT
FOUND DO no data found();
       switch (choice)
        case '1': printf("\nEnter Offset :");
                   scanf("%d",&offset);
                   getchar();
                   EXEC SQL FETCH ABSOLUTE :offset C USING DESCRIPTOR selda;
                  break;
         case '2': printf("\nEnter Offset :");
                   scanf("%d", &offset);
                   getchar();
                   EXEC SQL FETCH RELATIVE :offset C USING DESCRIPTOR selda;
                  break;
         case '3': EXEC SQL FETCH FIRST C USING DESCRIPTOR selda;
                  break:
         case '4': EXEC SQL FETCH NEXT C USING DESCRIPTOR selda;
                  break;
         case '5': EXEC SQL FETCH PRIOR C USING DESCRIPTOR selda;
                  break;
         case '6': EXEC SQL FETCH LAST C USING DESCRIPTOR selda;
                  break;
```



16

## **LOBs**

This chapter describes the support provided by embedded SQL statements for the LOB (Large Object) datatypes.

The four types of LOBs are introduced and compared to the older LONG and LONG RAW datatypes.

The embedded SQL interface in Pro\*C/C++ is shown to provide similar functionality to that of the Oracle Call Interface API and the PL/SQL language.

The LOB statements and their options and host variables are presented.

Lastly, examples of Pro\*C/C++ programs using the LOB interface are presented as simple illustrations of usage.

This chapter contains the following topics:

- What are LOBs?
- How to Use LOBs in Your Program
- · Rules for LOB Statements
- LOB Statements
- LOBs and the Navigational Interface
- LOB Program Examples

## 16.1 What are LOBs?

Use LOB (large object) columns to store large amounts of data (maximum size is 4 Gigabytes) such as ASCII text, National Character text, files in various graphics formats, and sound wave forms.

## 16.1.1 Internal LOBs

Internal LOBs (BLOBs, CLOBs, NCLOBs) are stored in database table spaces and have transactional support (Commit, Rollback, and so on. work with them) of the database server.

BLOBs (Binary LOBs) store unstructured binary (also called "raw") data, such as video clips.

CLOBs (Character LOBs) store large blocks of character data from the database character set.

NCLOBs (National Character LOBs) store large blocks of character data from the National Character Set.

## 16.1.2 External LOBs

External LOBs are operating system files outside the database tablespaces, that have no transactional support from the database server.

BFILEs (Binary Files) store data in external binary files. A BFILE can be in GIF, JPEG, MPEG, MPEG2, text, or other formats.

# 16.1.3 Security for BFILEs

The DIRECTORY object is used to access and use BFILEs. The DIRECTORY is a logical alias name (stored in the server) for the actual physical directory in the server file system containing the file. Users are permitted to access the file only if granted access privilege on the DIRECTORY object.

- The DDL (data definition language) SQL statements CREATE, REPLACE, ALTER, and DROP are used with DIRECTORY database objects.
- The DML (Data Management Language) SQL statements are used to GRANT and REVOKE the READ system and object privileges on DIRECTORY objects.

A example CREATE DIRECTORY directive is:

```
EXEC SQL CREATE OR REPLACE DIRECTORY "Mydir" AS '/usr/home/mydir';
```

Other users or roles can read the directory only if you grant them permission with a DML (Data Manipulation Language) statement, such as GRANT. For example, to allow user scott to read BFILES in directory /usr/home/mydir:

```
EXEC SQL GRANT READ ON DIRECTORY "Mydir" TO scott;
```

Up to 10 BFILES can be opened simultaneously in one session. This default value can be changed by setting the SESSION MAX OPEN FILES parameter.

#### **Related Topics**

- Oracle Database Development Guide
- GRANT

## 16.1.4 LOBs versus LONG and LONG RAW

LOBs are different from the older LONG and LONG RAW datatypes in many ways.

- The maximum size of a LOB is 4 Gigabytes versus 2 Gigabytes for LONG and LONG RAW.
- You can use random as well as sequential access methods on LOBs; you can only use sequential access methods on LONG and LONG RAW.
- LOBs (except NCLOBs) can be attributes of an object type that you define.
- Tables can have multiple LOB columns, but can have only one LONG or LONG RAW column.

Migration of existing LONG and LONG Raw attributes to LOBs is recommended by Oracle. Oracle plans to end support of LONG and LONG RAW in future releases. See *Oracle® Database Migration Guide* for more information on migration.

## 16.1.5 LOB Locators

A LOB locator points to the actual LOB contents. The locator is returned when you retrieve the LOB, not the LOB's contents. LOB locators cannot be saved in one transaction or session and used again in a later transaction or session.



## 16.1.6 Temporary LOBs

You can create *temporary* LOBs, that are like local variables, to assist your use of database LOBs. Temporary LOBs are not associated with any table, are only accessible by their creator, have locators (which is how they are accessed), and are deleted when a session ends.

There is no support for temporary BFILES. Temporary LOBs are only permitted to be input variables (IN values) in the WHERE clauses of INSERT, UPDATE, or DELETE statements. They are also permitted as values inserted by an INSERT statement, or a value in the SET clause of an UPDATE statement. Temporary LOBs have no transactional support from the database server, which means that you cannot do COMMITS or ROLLBACKs on them.

Temporary LOB locators can span transactions. They also are deleted when the server abnormally terminates, and when an error is returned from a database SQL operation.

## 16.1.7 LOB Buffering Subsystem

The LBS (LOB Buffering Subsystem) is an area of user memory provided for use as a buffer for one or more LOBs in the client's address space.

Buffering has these advantages, especially for applications on a client that does many small reads and writes to specific regions of the LOB:

- The LBS reduces round trips to the server because you fill the buffer with multiple reads/ writes to the LOBs, then write to the server when a FLUSH directive is executed.
- Buffering also reduces the total number of LOB updates on the server. This creates better LOB performance and saves disk space.

Oracle provides a simple buffer subsystem; not a cache. Oracle does not guarantee that the contents of a buffer are always synchronized with the server LOB value. Use the FLUSH statement to actually write updates in the server LOB.

Buffered read/write of a LOB are performed through its locator. A locator enabled for buffering provides a consistent read version of the LOB until you perform a write through that locator.

After being used for a buffered WRITE, a locator becomes an updated locator and provides access to the latest LOB version as seen through the buffering subsystem. All further buffered WRITEs to the LOB can only be done through this updated locator. Transactions involving buffered LOB operations cannot migrate across user sessions.

The LBS is managed by the user, who is responsible for updating server LOB values by using FLUSH statements to update them. It is single-user and single threaded. Use ROLLBACK and SAVEPOINT actions to guarantee correctness in the server LOBs. Transactional support for buffered LOB operations is not guaranteed by Oracle. To ensure transactional semantics for buffered LOB updates, you must maintain logical savepoints to perform a rollback in the event of an error.

# 16.2 How to Use LOBs in Your Program

This section describes some of the important programming issues related to the use of LOBs in your Pro\*C/C++ application.

## 16.2.1 Three Ways to Access LOBs

The three methods available to access LOBs in Pro\*C/C++ are:

- The DBMS\_LOB package inside PL/SQL blocks.
- OCI (Oracle Call Interface) function calls.
- Embedded SQL statements.

The SQL statements are designed to give users a functional equivalent to the PL/SQL interface while avoiding the complexity of the OCI interface.

The following table compares LOB access by OCI function calls in Pro\*C/C++, PL/SQL, and embedded SQL statements in Pro\*C/C++. Empty boxes indicate missing functionality.

Table 16-1 LOB Access Methods

OCI <sup>1</sup>	PL/SQL <sup>2</sup>	Pro*C/C++ Embedded SQL
-	COMPARE()	-
-	INSTR()	-
-	SUBSTR()	-
OCILobAppend	APPEND()	APPEND
OCILobAssign	:=	ASSIGN
OCILobCharSetForm	-	-
OCICharSetId	-	-
OCILobClose	CLOSE()	CLOSE
OCILobCopy	COPY()	COPY
OCILobCreateTemporary	CREATETEMPORARY()	CREATE TEMPORARY
OCILobDisableBuffering	-	DISABLE BUFFERING
OCILobEnableBuffering	-	ENABLE BUFFERING
OCILobErase	ERASE()	ERASE
OCILobGetChunkSize	GETCHUNKSIZE()	DESCRIBE
OCILobisOpen	ISOPEN()	DESCRIBE
OCILobFileClose	FILECLOSE()	CLOSE
OCILobFileCloseAll	FILECLOSEALL()	FILE CLOSE ALL
OCILobFileExists	FILEEXISTS()	DESCRIBE
OCILobFileGetName	FILEGETNAME()	DESCRIBE
OCILobFileIsOpen	FILEISOPEN()	DESCRIBE
OCILobFileOpen	FILEOPEN()	OPEN
OCILobFileSetName	BFILENAME()	FILE SET <sup>3</sup>
OCILobFlushBuffer	-	FLUSH BUFFER
OCILobFreeTemporary	FREETEMPORARY()	FREE TEMPORARY
OCILobGetLength	GETLENGTH()	DESCRIBE
OCILobisEqual	=	-
OCILobIsTemporary	ISTEMPORARY()	DESCRIBE
OCILobLoadFromFile	LOADFROMFILE()	LOAD FROM FILE
OCILobLocatorIsInit	-	-
OCILobOpen	OPEN()	OPEN
OCILobRead	READ()	READ
OCILobTrim	TRIM()	TRIM

Table 16-1 (Cont.) LOB Access Methods

OCI 1	PL/SQL <sup>2</sup>	Pro*C/C++ Embedded SQL
OCILobWrite	WRITE()	WRITE
OCILobWriteAppend	WRITEAPPEND()	WRITE

- 1 For C/C++ users only. Prototypes for these functions are in ociap.h.
- <sup>2</sup> From dbmslob.sql. All routines are prefixed with 'DBMS\_LOB.' except BFILENAME.
- 3 The BFILENAME() built in SQL function may also be used.



You must explicitly lock the row before using any of the new statements that modify or change a LOB in any way. Operations that can modify a LOB value are APPEND, COPY, ERASE, LOAD FROM FILE, TRIM, and WRITE.

## 16.2.2 LOB Locators in Your Application

To use a LOB locator in your Pro\*C/C++ application, include the oci.h header file and declare a pointer to the type OCIBlobLocator for BLOBs, OCIClobLocator for CLOBs and NCLOBs, or OCIBFileLocator for BFILEs.

For an NCLOB, you must either

- Use the clause 'CHARACTER SET IS NCHAR CS' in the C/C++ declaration,
- Or, you must have already used an NLS\_CHAR precompiler option on the command line or in a configuration file to set the NLS\_NCHAR environment variable.

Here is how it is done:

```
/* In your precompiler program */
#include <oci.h>
...
OCIClobLocator CHARACTER SET IS NCHAR CS *a nclob ;
```

Or, if you have already set the precompiler option NLS\_CHAR this way when invoking Pro\*C/C++:

```
NLS_CHAR=(a_nclob)
```

you can omit the CHARACTER SET clause in your code:

```
#include <oci.h>
...
OCIClobLocator *a_nclob;
```

#### The other declarations are simple:

```
/* In your precompiler program */
#include <oci.h>
...
OCIBlobLocator *a_blob;
OCIClobLocator *a_clob;
OCIBFileLocator *a bfile;
```



## **Related Topics**

NLS CHAR

# 16.2.3 Initializing a LOB

There are different techniques which are used to intialize the different types of LOBs. Each is described in this section.

# 16.2.3.1 Internal LOBs

To initialize a BLOB to empty, use the EMPTY\_BLOB() function or, use the ALLOCATE SQL statement. For CLOBs and NCLOBs, use the EMPTY\_CLOB() function. See *Oracle Database SQL Language Reference* for more about EMPTY BLOB() and EMPTY CLOB().

These functions are permitted only in the VALUES clause of an INSERT statement or as the source of the SET clause in an UPDATE statement.

## For example:

```
EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
VALUES (EMPTY BLOB(), EMPTY CLOB());
```

The ALLOCATE statement allocates a LOB locator and initializes it to empty. So, the following code is equivalent to the previous example:

```
#include <oci.h>
...
OCIBlobLocator *blob;
OCIClobLocator *clob;
EXEC SQL ALLOCATE :blob;
EXEC SQL ALLOCATE :clob;
EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
    VALUES (:blob, :clob);
```

# 16.2.3.2 External LOBs

Use the LOB FILE SET statement to initialize the DIRECTORY alias of the BFILE and FILENAME this way:

```
#include <oci.h>
...
char *alias = "lob_dir";
char *filename = "image.gif";
OCIBFileLocator *bfile;
EXEC SQL ALLOCATE :bfile;
EXEC SQL LOB FILE SET :bfile
   DIRECTORY = :alias, FILENAME = :filename;
EXEC SQL INSERT INTO file table (a bfile) VALUES (:bfile);
```

Refer to *Oracle Database Advanced Application Developer's Guide* for a complete description of DIRECTORY object naming conventions and DIRECTORY object privileges.

Alternatively, you can use the BFILENAME('directory', 'filename') function in an INSERT or UPDATE statement to initialize a BFILE column or attribute for a particular row, and give the name of the actual physical directory and filename:

```
EXEC SQL INSERT INTO file_table (a_bfile)
   VALUES (BFILENAME('lob_dir', 'image.gif'))
   RETURNING a bfile INTO :bfile;
```





BFILENAME() does not check permissions on the directory or filename, or whether the physical directory actually exists. Subsequent file accesses that use the BFILE locator will do those checks and return an error if the file is inaccessible.

# 16.2.3.3 Temporary LOBs

A temporary LOB is initialized to empty when it is first created using the embedded SQL LOB CREATE TEMPORARY statement. The EMPTY\_BLOB() and EMPTY\_CLOB() functions cannot be used with temporary LOBs.

# 16.2.3.4 Freeing LOBs

The FREE statement is used to free the memory reserved by an ALLOCATE statement:

EXEC SQL FREE :a blob;

# 16.3 Rules for LOB Statements

Here are the rules for using LOB statements:

# 16.3.1 For All LOB Statements

These general restrictions and limitations apply when manipulating LOBs with the SQL LOB statements:

- The FOR clause is not allowed in EXEC SQL LOB statements since only one LOB locator can be used in those statements.
- Distributed LOBs are not supported. Although you may use the AT database clause in any
  of the new embedded SQL LOB statements, you cannot mix LOB locators that were
  created or ALLOCATEd using different database connections in the same SQL LOB
  statement.
- For the LOB READ and WRITE operations, OCI provides a callback mechanism whereby the client can specify a callback function that will be executed each time a piece of the LOB value is either read or written. The embedded SQL LOB approach does not support this capability.
- OCI provides a mechanism that allows users to create and specify their own durations that
  can be used when creating temporary LOBs. There is also a mechanism for specifying that
  the buffer cache be used for READ and WRITE operations on temporary LOBs. This
  interface does not support these capabilities.

# 16.3.2 For the LOB Buffering Subsystem

For the LBS, these rules must be followed:

- Errors in read or write accesses are reported at the next access to the server. Therefore, error recovery has to be coded by you, the user.
- When updating a LOB with buffered writes, do not update the same LOB with a method that bypasses the LOB Buffering Subsystem.

- An updated LOB locator enabled for buffering can be passed as an IN parameter to a PL/SQL procedure, but not as an IN OUT or OUT parameter. An error is returned, An error also is returned when there is an attempt to return an updated locator.
- An ASSIGN of an updated locator enabled for buffering to another locator is not allowed.
- You can append to the LOB value with buffered writes, but the starting offset must be one character after the end of the LOB. The LBS does not allow APPEND statements resulting in zero-byte fillers or spaces in LOBs in the database server.
- The character sets of the host locator bind variable and the database server CLOB must be the same.
- Only ASSIGN, READ and WRITE statements work with a locator enabled for buffering.
- The following statements result in errors when used with a locator enabled for buffering:
   APPEND, COPY, ERASE, DESCRIBE (LENGTH only), and TRIM. Errors are also returned
   when you use these statements with a locator that is not enabled for buffering, if the LOB
   pointed to by the locator is being accessed in buffered mode by another locator.



The FLUSH statement must be used on a LOB enabled by the LOB Buffering Subsystem before.

- Committing the transaction.
- Migrating from the current transaction to another.
- Disabling buffer operations on a LOB.
- Returning from an external procedure execution back to the PL/SQL routine.

# Note:

If an external callout is called from a PL/SQL block with a locator parameter, then all buffering, including the ENABLE statement should be done inside the external procedure.

## Follow this recipe:

- Call the external callout.
- ENABLE the locator for buffering.
- READ or WRITE using the locator.
- FLUSH the LOB (LOBs are never implicitly flushed).
- Disable the locator for buffering.
- Return to the function/procedure/method in PL/SQL.

You have to explicitly FLUSH a LOB.

# 16.3.3 For Host Variables

Use the following rules and notes for the LOB statements:



- src and dst can refer to either internal or external LOB locators, but file refers only to external locators.
- Numeric host values (amt, src\_offset, dst\_offset, and so on) are declared as a 4-byte unsigned integer variable. The values are restricted between 0 and 4 Gigabytes.
- The concept of NULL is part of a LOB locator. There is no need for indicator variables in the LOB statements. NULL cannot be used with numeric value variables such as amt, src offset, and so on and result in an error.
- The offset values src offset and dst offset have default values 1.

# 16.4 LOB Statements

The statements are presented alphabetically. In all the statements where it appears, *database* refers to a database connection

# 16.4.1 APPEND

# **Purpose**

This statement appends a LOB value at the end of another LOB.

#### **Syntax**

```
EXEC SQL [AT [:] database] LOB APPEND :src TO :dst ;
```

#### **Host Variables**

src (IN)

An internal LOB locator uniquely referencing the source LOB.

dst (IN OUT)

An internal LOB locator uniquely referencing the destination LOB.

# **Usage Notes**

The data is copied from the source LOB to the end of the destination LOB, extending the destination LOB up to a maximum of 4 Gigabytes. If the LOB is extended beyond 4 Gigabytes, an error will occur.

The source and destination LOBs must already exist and the destination LOB must be initialized.

Both the source and destination LOBs must be of the same internal LOB type. It is an error to have enabled LOB buffering for either type of locator.

# 16.4.2 ASSIGN

## **Purpose**

Assigns a LOB or BFILE locator to another.

# **Syntax**

```
EXEC SQL [AT [:]database] LOB ASSIGN :src to :dst;
```



#### **Host Variables**

src (IN)

LOB or BFILE locator source copied from.

dst (IN OUT)

LOB or BFILE locator copied to.

## **Usage Notes**

After the assignment, both locators refer to the same LOB value. The destination LOB locator must be a valid initialized (ALLOCATEd) locator.

For internal LOBs, the source locator's LOB value is copied to the destination locator's LOB value only when the destination locator is stored in the table. For Pro\*C/C++, issuing a FLUSH of an object containing the destination locator will copy the LOB value.

An error is returned when a BFILE locator is assigned to an internal LOB locator and viceversa. It is also an error if the *src* and *dst* LOBs are not of the same type.

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 value through the LOB Buffering Subsystem, and the buffers have not been FLUSHed since the WRITE, then the source locator cannot be assigned to the destination locator. This is because only one locator for each LOB can modify the LOB value through the LOB Buffering Subsystem.

# 16.4.3 CLOSE (for LOBs)

## **Purpose**

Close an open LOB or BFILE.

## **Syntax**

EXEC SQL [AT [:] database] LOB CLOSE :src ;

#### **Host Variables**

src (IN OUT)

The locator of the LOB or BFILE to be closed.

#### **Usage Notes**

It is an error to close the same LOB twice either with different locators or with the same locator. For external LOBs, no error is produced if the BFILE exists but has not been opened.

It is an error to COMMIT a transaction before closing all previously opened LOBs. At transaction ROLLBACK time, all LOBs that are still open will be discarded without first being closed.

# 16.4.4 COPY

## **Purpose**

Copy all or part of a LOB value into a second LOB.

#### **Syntax**

```
EXEC SQL [AT [:] database] LOB COPY : amt FROM : src [AT : src_offset] TO : dst [AT : dst_offset] ;
```

#### **Host Variables**

amt (IN)

The maximum number of bytes for BLOBs, or characters for CLOBs and NCLOBs, to copy.

src (IN)

The locator of the source LOB.

src offset (IN)

This is the number of characters for CLOB or NCLOB, and the number of bytes for a BLOB, starting from 1 at the beginning of the LOB.

dst (IN)

The locator of the destination LOB.

dst\_offset (IN)

The destination offset. Same rules as for src offset.

# **Usage Notes**

If the data already exists at the destination's offset and beyond, it is overwritten with the source data. If the destination's offset is beyond the end of the current data, zero-byte fillers (BLOBs) or spaces (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 a runtime error to extend this LOB beyond 4 Gigabytes.

It is also an error to try to copy from a LOB that is not initialized.

Both the source and destination LOBs must be of the same type. LOB buffering must not be enabled for either locator.

The amt variable indicates the maximum amount to copy. If the end of the source LOB is reached before the specified amount is copied, the operation terminates with ORA-22993 error.

To make a temporary LOB permanent, the COPY statement must be used to explicitly COPY the temporary LOB into a permanent one.

# 16.4.5 CREATE TEMPORARY

## **Purpose**

Creates a temporary LOB.

## **Syntax**

EXEC SQL [AT [:] database] LOB CREATE TEMPORARY :src ;



#### **Host Variables**

src (IN OUT)

Before execution, when IN, src is a LOB locator previously ALLOCATEd.

After execution, when OUT, src is a LOB locator that will point to a new empty temporary LOB.

# **Usage Notes**

After successful execution, the locator points to a newly created temporary LOB that resides on the database server independent of a table. The temporary LOB is empty and has zero length.

At the end of a session, all temporary LOBs are freed. READs and WRITEs to temporary LOBs never go through the buffer cache.

# 16.4.6 DISABLE BUFFERING

# **Purpose**

Disables LOB buffering for the LOB locator.

## **Syntax**

```
EXEC SQL [AT [:] database] LOB DISABLE BUFFERING :src ;
```

#### **Host Variable**

src (IN OUT)

An internal LOB locator.

# **Usage Notes**

This statement does not support BFILEs. Subsequent reads or writes will not be done through the LBS.



Use a FLUSH BUFFER command to make changes permanent, since this statement does not implicitly flush the changes made in the LOB Buffering Subsystem.

# 16.4.7 ENABLE BUFFERING

#### **Purpose**

Enables LOB buffering for the LOB locator.

## **Syntax**

EXEC SQL [AT [:] database] LOB ENABLE BUFFERING :src ;



#### **Host Variable**

src (IN OUT)

An internal LOB locator.

# **Usage Notes**

This statement does not support BFILEs. Subsequent reads and writes are done through the

# 16.4.8 ERASE

## **Purpose**

Erases a given amount of LOB data starting from a given offset.

## **Syntax**

```
EXEC SQL [AT [:]database] LOB ERASE :amt FROM :src [AT :src offset] ;
```

## **Host Variables**

amt (IN OUT)

The input is the number of bytes or characters to erase. The returned output is the actual number erased.

src (IN OUT)

An internal LOB locator.

src\_offset (IN)

The offset from the beginning of the LOB, starting from 1.

# **Usage Notes**

This statement does not support BFILEs.

After execution, amt returns the actual number of characters/bytes that were erased. The actual number and requested number will differ if the end of the LOB value is reached before erasing the requested number of characters/bytes. If the LOB is empty, amt will indicate that 0 characters/bytes were erased.

For BLOBs, erasing means zero-byte fillers overwrite the existing LOB value. For CLOBs, erasing means that spaces overwrite the existing LOB value.

# 16.4.9 FILE CLOSE ALL

## **Purpose**

Closes all BFILES opened in the current session.

## **Syntax**

```
EXEC SQL [AT [:] database] LOB FILE CLOSE ALL ;
```



## **Usage Notes**

If there are any open files in the session whose closure has not been handled properly, you can use the FILE CLOSE ALL statement to close all files opened in the session, and resume file operations from the beginning.

# 16.4.10 FILE SET

# **Purpose**

Sets DIRECTORY alias and FILENAME in a BFILE locator.

# **Syntax**

```
EXEC SQL [AT [:]database] LOB FILE SET :file
   DIRECTORY = :alias, FILENAME = :filename;
```

#### **Host Variables**

file (IN OUT)

BFILE locator where the DIRECTORY alias and FILENAME is set.

alias (IN)

DIRECTORY alias name to set.

filename (IN)

The FILENAME to set.

## **Usage Notes**

The given BFILE locator must be first ALLOCATEd prior to its use in this statement.

Both the DIRECTORY alias name and FILENAME must be provided.

The maximum length of the DIRECTORY alias is 128 bytes. The maximum length of the FILENAME is 255 bytes.

The only external datatypes supported for use with the DIRECTORY alias name and FILENAME attributes are CHARZ, STRING, VARCHAR, VARCHAR2 and CHARF.

It is an error to use this statement with anything but an external LOB locator.

# 16.4.11 FLUSH BUFFER

#### **Purpose**

Writes this LOB's buffers to the database server.

#### **Syntax**

```
EXEC SQL [AT [:] database] LOB FLUSH BUFFER :src [FREE] ;
```

# **Host Variables**

src (IN OUT)

Internal LOB locator.



## **Usage Notes**

Writes the buffer data to the database LOB in the server from the LOB referenced by the input locator.

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 include the optional FREE keyword to so indicate.

# 16.4.12 FREE TEMPORARY

## **Purpose**

Free the temporary space for the LOB locator.

## **Syntax**

```
EXEC SQL [AT [:] database] LOB FREE TEMPORARY :src ;
```

## **Host Variable**

src (IN OUT)

The LOB locator pointing to the temporary LOB.

## **Usage Notes**

The input locator must point to a temporary LOB. The output locator is marked not initialized and can be used in subsequent LOB statements.

# 16.4.13 LOAD FROM FILE

# **Purpose**

Copy all or a part of a BFILE into an internal LOB.

## **Syntax**

## **Host Variables**

amt (IN)

Maximum number of bytes to be loaded.

file (IN OUT)

The source BFILE locator.

src offset (IN)

The number of bytes offset from the beginning of the file, starting from 1.

dst (IN OUT)

The destination LOB locator which can be BLOB, CLOB, be NCLOB.

dst\_offset (IN)

The number of bytes (for BLOBs) or characters (CLOBs and NCLOBs) from the beginning of the destination LOB where writing will begin. It starts at 1.

## **Usage Notes**

The data is copied from the source BFILE to the destination internal LOB. No character set conversions are performed when copying the BFILE data to a CLOB or NCLOB. Therefore, the BFILE data must already be in the same character set as the CLOB or NCLOB in the database.

The source and destination 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 (BLOBs) or spaces (CLOBs and NCLOBs) are written into the destination LOB. The fillers are written to 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 this LOB beyond 4 Gigabytes.

It is also an error to copy from a BFILE that is not initialized.

The amount parameter indicates the maximum amount to load. If the end of the source BFILE is reached before the specified amount is loaded, the operation terminates with ORA-22993 error.

# 16.4.14 OPEN (for LOBs)

#### **Purpose**

Open a LOB or BFILE for read or read/write access.

## **Syntax**

```
EXEC SQL [AT [:]database] LOB OPEN :src [ READ ONLY | READ WRITE ] ;
```

## **Host Variables**

src (IN OUT)

LOB locator of the LOB or BFILE.

#### **Usage Notes**

The default mode in which a LOB or BFILE can be OPENed is for READ ONLY access.

For internal LOBs, being OPEN is associated with the LOB, not the locator. Assigning an already OPENed locator to another locator does not count as OPENing a new LOB. Instead, both locators refer to the same LOB. For BFILEs, being OPEN is associated with the locator.

Only 32 LOBs can be OPEN at any one time. An error will be returned when the 33rd LOB is OPENed.

There is no support for writable BFILEs. Therefore, when you OPEN a BFILE in READ WRITE mode, an error is returned.

It is also an error to open a LOB in READ ONLY mode and then attempt to WRITE to the LOB.



# 16.4.15 READ

## **Purpose**

Reads all or part of a LOB or BFILE into a buffer.

## **Syntax**

```
EXEC SQL [AT [:]database] LOB READ :amt FROM :src [AT :src_offset]
INTO :buffer [WITH LENGTH :buflen] ;
```

## **Host Variables**

amt (IN OUT)

The input is the number of characters or bytes to be read. The output is the actual number of characters or bytes that were 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 polling mode. On input if this value is 0, then the data will be read in a polling mode from the input offset until the end of the LOB.

The number of bytes or characters actually read is returned in amt. If the data is read in pieces, amt will always contain the length of the last piece read.

When the end of a LOB is reached an ORA-1403: no data found error will be issued.

When reading in a polling mode, the application must invoke the LOB READ repeatedly to read more pieces of the LOB until no more data is left. Control the use of the polling mode with the NOT FOUND condition in a WHENEVER directive to catch the ORA-1403 error.

src (IN)

The LOB or BFILE locator.

src offset (IN)

This is the absolute offset from the beginning of the LOB value from which to start reading. For character LOBs it is the number of characters from the beginning of the LOB. For binary LOBs or BFILEs it is the number of bytes. The first position is 1.

buffer (IN/OUT)

A buffer into which the LOB data will be read. The external datatype of the buffer is restricted to only a few types depending on the type of the source LOB. The maximum length of the buffer depends on the external datatype being used to store the LOB value. The following table summarizes the legal external datatypes and their corresponding maximum lengths categorized by source LOB type:



Table 16-2 Source LOB and Precompiler Datatypes

External LOB <sup>1</sup>	Internal LOB	Precompiler External Datatype	Precompiler Maximum Length <sup>2</sup>	PL/SQL Datatype	PL/SQL Maximum Length
DEIL E	BLOB	RAW	65535	RAW	32767
BFILE		VARRAW	65533		
		LONG RAW	2147483647		
		LONG VARRAW	2147483643		
BFILE	CLOB	VARCHAR2	65535	VARCHAR2	32767
		VARCHAR	65533		
		LONG VARCHAR	2147483643		
BFILE	NCLOB	NVARCHAR2	4000	NVARCHAR2	4000

<sup>1</sup> Any of the external datatypes shown can be used with BFILES.

buflen (IN)

Specifies the length of the given buffer when it cannot be determined otherwise.

## **Usage Notes**

A BFILE must already exist on the database server and it must have been opened using the input locator. The database must have permission to read the file and you, the user, must have read permission on the directory.

It is an error to try to read from an un-initialized LOB or BFILE.

The length of the buffer is determined this way:

- From buflen, when the WITH LENGTH clause is present.
- In the absence of the WITH LENGTH clause, the length is determined by treating the buffer host variable in OUT mode.

## **Related Topics**

- Globalization Support
- READ a BLOB, Write a File Example

# 16.4.16 TRIM

#### **Purpose**

Truncates the LOB value.

#### **Syntax**

EXEC SQL [AT [:] database] LOB TRIM :src TO :newlen ;

# **Host Variables**

src (IN OUT)

LOB locator for internal LOB.

<sup>&</sup>lt;sup>2</sup> Lengths are measured in bytes, not characters.

newlen (IN)

The new length of the LOB value.

## **Usage Notes**

This statement is not for BFILES. The new length cannot be greater than the current length, or an error is returned.

# 16.4.17 WRITE

#### **Purpose**

Writes the contents of a buffer to a LOB.

## **Syntax**

```
EXEC SQL [AT [:]database] LOB WRITE [APPEND] [ FIRST | NEXT | LAST | ONE ] :amt FROM :buffer [WITH LENGTH :buflen] INTO :dst [AT :dst offset] ;
```

#### **Host Variables**

## amt (IN OUT)

The input is the number of characters or bytes to be written.

The output is the actual number of characters or bytes that is written.

When writing using a polling method, amt will return the cumulative total length written for the execution of the WRITE statement after a WRITE LAST is executed. If the WRITE statement is interrupted, amt will be undefined.

#### buffer (IN)

A buffer from which the LOB data is written.

#### dst (IN OUT)

The LOB locator.

# dst\_offset (IN)

The offset from the beginning of the LOB (counting from 1), in characters for CLOBs and NCLOBs, in bytes for BLOBs.

## buflen (IN)

The buffer length when it cannot be calculated in any other way.

## **Usage Notes**

If LOB data already exists, it is overwritten with the data stored in the buffer. If the offset specified is beyond the end of the data currently in the LOB, zero-byte fillers or spaces are inserted into the LOB.

Specifying the keyword APPEND in the WRITE statement causes the data to automatically be written to the end of the LOB. When APPEND is specified, the destination offset is assumed to be the end of the LOB. It is an error to specify the destination offset when using the APPEND option in the WRITE statement.

The buffer can be written to the LOB in one piece (using the ONE orientation which is the default) or it can be provided piece-wise using a standard polling method.

Polling is begun by using FIRST, then NEXT to write subsequent pieces. The LAST keyword is used to write the final piece that terminates the write.



Using this piece-wise write mode, the buffer and the length can be different in each call if the pieces are of different sizes and from different locations.

If the total amount of data passed to Oracle is less than the amount specified by the amt parameter after doing all the writes, an error results.

The same rules apply for determining the buffer length as in the READ statement.

## **Related Topics**

- READ
- READ a BLOB, Write a File Example

# **16.4.18 DESCRIBE**

#### **Purpose**

This is a statement that is equivalent to several OCI and PL/SQL statements (which is why it is saved for last). Use the LOB DESCRIBE SQL statement to retrieve attributes from a LOB. This capability is similar to OCI and PL/SQL procedures. The LOB DESCRIBE statement has this format:

# **Syntax**

```
EXEC SQL [AT [:]database] LOB DESCRIBE :src GET attribute1 [{, attributeN}]
INTO :hv1 [[INDICATOR] :hv ind1] [{, :hvN [[INDICATOR] :hv indN] }];
```

where an attribute can be any of these choices:

```
CHUNKSIZE | DIRECTORY | FILEEXISTS | FILENAME | ISOPEN | ISTEMPORARY | LENGTH
```

#### **Host variables**

src (IN)

The LOB locator of an internal or external LOB.

```
hv1 ... hvN ... (OUT)
```

The host variables that receive the attribute values, in the order specified in the attribute name list.

```
hv_ind1 ... hv_indN ... (OUT)
```

Optional host variables that receive the indicator NULL status in the order of the attribute name list.

This table describes the attributes, which LOB it is associated with, and the C types into which they should be read:



Table 16-3 LOB Attributes

LOB Attribute	Attribute Description	Restrictions	С Туре
CHUNKSIZE	The amount (in bytes for BLOBs and characters for CLOBs/NCLOBs) of space used in the LOB chunk to store the LOB value. You speed up performance if you issue READ/WRITE requests using a multiple of this chunk size. If all WRITEs are done on a chunk basis, no extra/excess versioning is done nor duplicated. Users could batch up the WRITE until they have enough for a chunk instead of issuing several WRITE calls for the same CHUNK.	BLOBs, CLOBs, and NCLOBs only	unsigned int
DIRECTORY	The name of the DIRECTORY alias for the BFILE. The maximum length is 128 bytes.	FILE LOBs only	char * 1
FILEEXISTS	Determines whether or not the BFILE exists on the server's operating system's file system. FILEEXISTS is true when it is nonzero; false when it equals 0.	FILE LOBs only	signed int
FILENAME	The name of the BFILE. The maximum length is 255 bytes.	FILE LOBs only	char *
ISOPEN	For BFILEs, if the input BFILE locator was never used in an OPEN statement, the BFILE is considered not to be OPENed by this locator. However, a different BFILE locator may have OPENed the BFILE. More than one OPEN can be performed on the same BFILE using different locators. For LOBs, if a different locator OPENed the LOB, the LOB is still considered to be OPEN by the input locator. ISOPEN is true when it is nonzero; false when it equals 0.	-	signed int
ISTEMPORARY	Determines whether or not the input LOB locator refers to a temporary LOB or not. ISTEMPORARY is true when it is nonzero; false when it equals 0.	BLOBs, CLOBs, and NCLOBs only	signed int
LENGTH	Length of BLOBs and BFILEs in bytes, CLOBs and NCLOBs in characters. For BFILEs, the length includes EOF if it exists. Empty internal LOBs have zero length. LOBs/BFILEs that are not initialized have undefined length.	-	unsigned int

<sup>&</sup>lt;sup>1</sup> For DIRECTORY and FILENAME attributes, the only external datatypes that will be supported are CHARZ, STRING, VARCHAR, VARCHAR2 and CHARF.

# **Usage Notes**

Indicator variables should be declared short. After execution has completed, sqlca.sqlerrd[2] contains the number of attributes retrieved without error. If there was an execution error, the attribute at which it occurred is one more than the contents of sqlca.sqlerrd[2].

# **DESCRIBE Example**

Here is a simple Pro\*C/C++ example that extracts the DIRECTORY and FILENAME attributes of a given BFILE:



The oci.h header file is needed for the proper type resolution and compilation of the following OCIBFileLocator declaration:

```
#include <oci.h>
...
OCIBFileLocator *bfile;
char directory[31], filename[256];
short d ind, f ind;
```

Finally, select a BFILE locator from some LOB table and perform the DESCRIBE:

```
EXEC SQL ALLOCATE :bfile ;
EXEC SQL SELECT a_bfile INTO :bfile FROM lob_table WHERE ...;
EXEC SQL LOB DESCRIBE :bfile
   GET DIRECTORY, FILENAME INTO :directory:d_ind, :filename:f_ind;
```

Indicator variables are only valid for use with the DIRECTORY and FILENAME attributes. These attributes are character strings whose values may be truncated if the host variable buffers used to hold their values aren't large enough. When truncation occurs, the value of the indicator will be set to the original length of the attribute.

# 16.5 LOBs and the Navigational Interface

The navigational interface can also be used to work with object types that contain LOBs as attributes.

## **Related Topics**

Navigational Access to Objects

# 16.5.1 Transient Objects

Use the OBJECT CREATE statement to create transient and persistent objects with LOB attributes. You can ASSIGN a temporary LOB to the LOB attribute of a transient object, then copy the value to a permanent LOB or a LOB attribute of a persistent object to save the data. Or, ASSIGN the temporary LOB to the LOB attribute and use FLUSH to write the value to the database.

You can create a transient object with a BFILE attribute and read data from the BFILE on disk. Remember, temporary BFILEs are not supported.

# 16.5.2 Persistent Objects

When you create a persistent object in the object cache that contains an internal LOB attribute, the LOB attribute is implicitly set to empty. You must first flush this object using the OBJECT FLUSH statement, thereby inserting a row into the table and creating an empty LOB. Once the object is refreshed in the object cache (using the VERSION=LATEST option), the real locator is read into the attribute.

When creating an object with a BFILE attribute, the BFILE is set to NULL. It must be updated with a valid directory alias and filename before the BFILE can be read.

A temporary LOB may be ASSIGNed to a LOB attribute of a persistent object. The actual LOB value will be copied when the object is flushed. A user may also explicitly copy a temporary LOB's value to a LOB attribute of a persistent object using the temporary LOB locator and a locator for the LOB attribute in a COPY statement.

# 16.5.3 Navigational Interface Example

Use the OBJECT GET and SET statements to handle LOBs through the navigational interface.

You can retrieve a LOB locator that is an attribute of an object type and use it in any of the new embedded SQL LOB statements. Place LOB locators back into object types as attributes using the OBJECT SET statement.

Doing so is the same as a direct LOB ASSIGN operation. The same rules apply to an OBJECT GET or SET of a LOB attribute from or to an object type that would apply if a LOB ASSIGN had been performed instead, including type enforcement.

For example, suppose we had this simple type definition

```
CREATE TYPE lob_type AS OBJECT (a_blob BLOB) ;
```

This example assumes that the type is a column in the database with a valid (and initialized) BLOB attribute.

The OTT-generated C structure usable by Pro\*C/C++ looks like this:

```
struct lob_type
{
   OCIBlobLocator *a_blob ;
} ;
typedef struct lob_type lob_type ;
```

You can write a Pro\*C/C++ program to extract the BLOB attribute and retrieve the BLOB's current length in a DESCRIBE statement. You can then TRIM the BLOB to half its size, setting the attribute back with a SET OBJECT and then make the change permanent with an OBJECT FLUSH.

First include oci.h and make some local variable declarations:

```
#include <oci.h>
lob_type *lob_type_p;
OCIBlobLocator *blob = (OCIBlobLocator *)0;
unsigned int length;
```

Select the BLOB attribute from the object, do an OBJECT GET, and do a DESCRIBE to get the current length of the BLOB:

```
EXEC SQL ALLOCATE :blob ;
EXEC SQL SELECT a_column
   INTO :lob_type_p FROM a_table WHERE ... FOR UPDATE ;
EXEC SQL OBJECT GET a_blob FROM :lob_type_p INTO :blob ;
EXEC SQL LOB DESCRIBE :blob GET LENGTH INTO :length ;
```

Cut the length in half and TRIM the BLOB to that new length:

```
length = (unsigned int)(length / 2);
EXEC SQL LOB TRIM :blob TO :length;
```

Once the BLOB has been changed, set the BLOB attribute back into the object, FLUSH the change back to the server, and commit:

```
EXEC SQL OBJECT SET a_blob OF :lob_type_p TO :blob ;
EXEC SQL OBJECT FLUSH :lob_type_p ;
EXEC SQL FREE :blob ;
EXEC SQL COMMIT WORK ;
```





The Object Type Translator for information on creating an INTYPE file for OTT and running OTT.

# 16.6 LOB Program Examples

Here are two examples which show how to read and write BFILEs and BLOBs.

# 16.6.1 READ a BLOB, Write a File Example

In this example we will be reading data from a BLOB with an unknown arbitrary length into a buffer and then writing the data from the buffer into an external file. Our buffer is small, so depending on the size of the BLOB we are reading, we may be able to read the BLOB value into the buffer in a single READ statement or we may be required to utilize a standard polling method instead.

First we start off with oci.h and some simple local variable declarations

```
#include <oci.h>
OCIBlobLocator *blob;
FILE *fp;
unsigned int amt, offset = 1;
```

Now we need a buffer to store the BLOB value and then write to the file from:

```
#define MAXBUFLEN 5000
unsigned char buffer[MAXBUFLEN] ;
EXEC SQL VAR buffer IS RAW(MAXBUFLEN) ;
```

Allocate the BLOB host variable and select a BLOB which we will READ:

```
EXEC SQL ALLOCATE :blob ;
EXEC SQL SELECT a_blob INTO :blob FROM lob_table WHERE ... ;
```

We can then open the external file to which we will write the BLOB value:

```
fp = fopen((const char *)"image.gif", (const char *)"w") ;
```

If the buffer can hold the entire LOB value in a single READ we need to catch the NOT FOUND condition to signal LOB READ termination:

```
EXEC SQL WHENEVER NOT FOUND GOTO end of lob;
```

Now do our first READ. We set the amount to the maximum value of 4 Gigabytes. It is larger than our buffer so if the LOB doesn't fit we will READ using a polling mode:

```
amt = 4294967295 ;
EXEC SQL LOB READ :amt FROM :blob AT :offset INTO :buffer ;
```

If we get here then it means that the buffer was not large enough to hold the entire LOB value, so we must write what we have using binary I/O and continue reading:

```
(void) fwrite((void *)buffer, (size_t)MAXBUFLEN, (size_t)1, fp) ;
```

We use a standard polling method to continue reading with the LOB READ inside of an infinite loop. We can set up the NOT FOUND condition to terminate the loop:

```
EXEC SQL WHENEVER NOT FOUND DO break ;
while (TRUE)
{
```

During polling, the offset is not used so we can omit it in subsequent LOB READs. We need the amount, however, because it will tell us how much was READ in the last READ invocation

```
EXEC SQL LOB READ :amt FROM :blob INTO :buffer ;
  (void) fwrite((void *)buffer, (size_t)MAXBUFLEN, (size_t)1, fp) ;
}
```

Here, we have reached the end of the LOB value. The amount holds the amount of the last piece that was READ. During polling, the amount for each interim piece was set to MAXBUFLEN, or the maximum size of our buffer:

```
end_of_lob:
(void) fwrite((void *)buffer, (size t)amt, (size t)1, fp);
```

The basic structure of this code should allow internal LOBs of arbitrary length to be READ into local buffers and then written to external files. It has been modeled after OCI and PL/SQL.

# 16.6.2 Read a File, WRITE a BLOB Example

In this example we will be reading data from a file with a known arbitrary length into a buffer and then writing the data from the buffer into an internal BLOB. Our buffer is small, so depending on the size of the file we are reading, we may be able to write the file data into the LOB in a single WRITE or we may be required to utilize a standard polling method instead.

First we start off with oci.h and some simple local variable declarations

```
#include <oci.h>
OCIBlobLocator *blob;
FILE *fp;
unsigned int amt, offset = 1;
unsigned filelen, remainder, nbytes;
boolean last;
```

We need a buffer to store the file data and then write to the LOB:

```
#define MAXBUFLEN 5000
unsigned char buffer[MAXBUFLEN] ;
EXEC SQL VAR buffer IS RAW(MAXBUFLEN) ;
```

We initialize an empty BLOB in an empty table and retrieve that BLOB into an ALLOCATEd locator and then fill it with the data from the file:

```
EXEC SQL ALLOCATE :blob ;
EXEC SQL INSERT INTO lob_table (a_blob) VALUES (EMPTY_BLOB())
RETURNING a blob INTO :blob ;
```

Open the binary file and determine its length. The total amount we write to our BLOB is the actual length of the binary file:

```
fp = fopen((const char *)"image.gif", (const char *)"r");
(void) fseek(fp, OL, SEEK_END);
filelen = (unsigned int)ftell(fp);
amt = filelen;
```

Set up our initial read of the file, determining the number of bytes to read based on our buffer size:

```
if (filelen > MAXBUFLEN)
   nbytes = MAXBUFLEN ;
else
   nbytes = filelen ;
```

Issue a file I/O operation to read n bytes of data from our file, fp, into our buffer and determine how much is left to read. Start reading from the beginning of the file:

```
(void) fseek(fp, OL, SEEK_SET) ;
(void) fread((void *)buffer, (size_t)nbytes, (size_t)1, fp) ;
remainder = filelen - nbytes ;
```

Based on what is left, either write the buffer in a single piece or initiate polling to write the data from the file in several smaller pieces:

```
if (remainder == 0)
{
```

In this case we can write the data in a single piece:

```
EXEC SQL LOB WRITE ONE :amt
    FROM :buffer INTO :blob AT :offset ;
}
else
{
```

Initiate the polling method for writing the data piece-wise into the LOB. First, to initiate the polling method, we use the FIRST orientation on the initial WRITE:

```
EXEC SQL LOB WRITE FIRST :amt
FROM :buffer INTO :blob AT :offset ;
```

Set up a simple loop to implement the polling method:

```
last = FALSE ;
EXEC SQL WHENEVER SQLERROR DO break ;
do
  {
```

Calculate the number of bytes to read from the file and subsequently to WRITE into the destination LOB. Also determine if this will be our LAST piece:

```
if (remainder > MAXBUFLEN)
    nbytes = MAXBUFLEN;
else
    {
        nbytes = remainder;
        last = TRUE;
}
```

Again read the next nbytes from the file on the file system into our buffer. If any error occurs during a file read, we automatically set the next WRITE to be the LAST one:

```
if fread((void *)buffer, (size_t)nbytes, (size_t)1, fp) != 1)
  last = TRUE;
```

At this point, either WRITE the LAST piece or an interim NEXT piece which would indicate that there is still data left to be processed from the file:

```
if (last)
{
    EXEC SQL LOB WRITE LAST :amt
    FROM :buffer INTO :blob ;
```

```
else
{
    EXEC SQL LOB WRITE NEXT :amt
    FROM :buffer INTO :blob ;
}
remainder -= nbytes ;
}
while (!last && !feof(fp)) ;
```

This code example allows files of arbitrary length to be read into a local buffer and then written to a LOB. It has been modeled after OCI examples.

# 16.6.3 lobdemo1.pc

This program, <code>lobdemol.pc</code>, illustrates several LOB embedded SQL statements. The source code is in the <code>demo</code> directory. The application uses a table named <code>license\_table</code> whose columns are social security number, name, and a CLOB containing text summarizing driving offenses. Several simplified SQL operations of a typical motor vehicle department are modeled.

The possible actions are:

Add new records.

To run the demo:

- List records by social security number.
- List information in a record, given a social security number.
- Append a new traffic violation to an existing CLOB's contents.

```
1. Execute the script, lobdemolc.sql in SQL*Plus
  2. Precompile using Pro*C/C++
       proc lobdemo1 user=scott/tiger sqlcheck=full
   3. Compile/Link (This step is platform specific)
*************************
/*** The following will be added to the creation script for this example ***
 *** This code can be found in lobdemolc.sql
connect scott/tiger;
set serveroutput on;
Rem Make sure database has no license table floating around
drop table license table;
Rem ABSTRACTION:
Rem A license table reduces the notion of a driver's license into three
Rem distinct components - a unique social security number (sss),
Rem a name (name), and a text summary of miscellaneous information.
Rem IMPLEMENTATION:
Rem Our implementation follows this abstraction
create table license table (
 sss char(9),
 name varchar2(50),
 txt_summary clob);
insert into license table
   values('971517006', 'Dennis Kernighan',
    'Wearing a Bright Orange Shirt - 31 Oct 1996');
insert into license table
   values('555001212', 'Eight H. Number',
   'Driving Under the Influence - 1 Jan 1997');
insert into license table
    values('010101010', 'P. Doughboy',
    'Impersonating An Oracle Employee - 10 Jan 1997');
insert into license table
    values('555377012', 'Calvin N. Hobbes',
    'Driving Under the Influence - 30 Nov 1996');
select count(*) from license_table;
Rem Commit to save
commit;
*****************************
/********
 * Begin lobdemol.pc code *
 *********
#define EX SUCCESS
#define EX FAILURE
#ifndef STDIO
```

```
# include <stdio.h>
#endif /* STDIO */
#ifndef SQLCA ORACLE
# include <sqlca.h>
#endif /* SQLCA ORACLE */
#ifndef OCI ORACLE
# include <oci.h>
#endif /* OCI ORACLE */
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#ifndef LOBDEMO1 ORACLE
# include "lobdemo1.h"
#endif /* LOBDEMO1 ORACLE */
/******
* Defines *
*********
#define SSS LENGTH 12
\#define NAME LENGTH 50 /* corresponds with max length of name in table */
#define BUFLEN 1024
#define MAXCRIME 5
#define DATELENGTH 12
/******
* Globals *
char *CrimeList[MAXCRIME]={ "Driving Under the Influence",
               "Grand Theft Auto",
               "Driving Without a License",
               "Impersonating an Oracle Employee",
                "Wearing a Bright Orange Shirt" };
char curdate[DATELENGTH];
/*******
* Function prototypes *
#if defined( STDC )
 void GetDate( void );
 void PrintSQLError( void );
 void Driver( void );
 void ListRecords( void );
 void PrintCrime( OCIClobLocator *a clob );
 void GetRecord( void );
 void NewRecord( void );
 char *NewCrime( void );
 void GetName( char *name holder );
 void AppendToClob( OCIClobLocator *a clob, char *charbuf );
 void AddCrime( void );
 void ReadClob( OCIClobLocator *a_clob );
 boolean GetSSS( char *suggested sss );
#else
 void GetDate();
 void PrintSQLError();
```

```
void Driver();
  void ListRecords();
  void PrintCrime(/* OCIClobLocator *a_clob */);
  void GetRecord();
  void NewRecord();
  char *NewCrime();
  void GetName(/* char *name holder */);
  void AppendToClob(/* OCIClobLocator *a clob, char *charbuf */);
  void AddCrime();
  boolean GetSSS(/* char *suggested_sss */);
#endif
* NAME
 * GetDate
* DESCRIPTION
 * Get date from user
 * LOB FEATURES
 * none
 * /
void GetDate()
{
  time_t now;
 now = time(NULL);
  strftime(curdate, 100, " - %d %b %Y", localtime(&now));
main()
{
  char * uid = "scott/tiger";
  EXEC SQL WHENEVER SQLERROR DO PrintSQLError();
  printf("Connecting to license database account: %s \n", uid);
  EXEC SQL CONNECT : uid;
  GetDate();
  printf("\t*********************************");
  printf("\t* Welcome to the DMV Database *\n");
  printf("\t**********************\n\n");
  printf("Today's Date is%s\n", curdate);
  Driver();
  EXEC SQL COMMIT RELEASE;
  return (EX SUCCESS);
* NAME
 * Driver
 * DESCRIPTION
 * Command Dispatch Routine
 * LOB FEATURES
 * none
 */
void Driver()
```

```
char choice[20];
 boolean done = FALSE;
 while (!done)
   printf("\nLicense Options:\n");
   printf("\t(L)ist available records by SSS number\n");
   printf("\t(G)et information on a particular record\n");
   printf("\t(A)dd crime to a record\n");
   printf("\t(I)nsert new record to database\n");
   printf("\t(Q)uit\n");
   printf("Enter your choice: ");
   fgets(choice, 20, stdin);
   switch(toupper(choice[0]))
   case 'L':
     ListRecords();
     break;
   case 'G':
     GetRecord();
     break;
   case 'A':
     AddCrime();
     break;
   case 'I':
     NewRecord();
     break;
   case 'Q':
     done = TRUE;
     break;
   default:
     break;
 }
}
/*
* NAME
* ListRecords
* DESCRIPTION
* List available records by sss number
* LOB FEATURES
   none
void ListRecords()
 char *select sss = "SELECT SSS FROM LICENSE TABLE";
 char sss[10];
 EXEC SQL PREPARE sss exec FROM :select sss;
 EXEC SQL DECLARE sss cursor CURSOR FOR sss exec;
 EXEC SQL OPEN sss_cursor;
 printf("Available records:\n");
 EXEC SQL WHENEVER NOT FOUND DO break;
 while (TRUE)
     EXEC SQL FETCH sss cursor INTO :sss;
```

```
printf("\t%s\n", sss);
   }
 EXEC SQL WHENEVER NOT FOUND CONTINUE;
 EXEC SQL CLOSE sss_cursor;
/*
* NAME
* PrintCrime
* DESCRIPTION
* Tests correctness of clob
* LOB FEATURES
* OCIlobRead and OCILobGetLength
void PrintCrime(a clob)
 OCIClobLocator *a clob;
{
 ub4 lenp;
 printf("\n");
 printf("=======n");
 printf(" CRIME SHEET SUMMARY \n");
 printf("========\n\n");
 EXEC SQL LOB DESCRIBE :a clob GET LENGTH INTO :lenp;
 if(lenp == 0) /* No crime on file */
   {
     printf("Record is clean\n");
 else
     ub4 amt = lenp;
     varchar *the_string = (varchar *)malloc(2 + lenp);
     the string->len = (ub2)lenp;
     EXEC SQL WHENEVER NOT FOUND CONTINUE;
     EXEC SQL LOB READ :amt
       FROM :a_clob INTO :the_string WITH LENGTH :lenp;
     printf("%.*s\n", the_string->len, the_string->arr);
     free(the_string);
}
* NAME
* GetRecord
* DESCRIPTION
* Get license of single individual
* LOB FEATURES
   allocate and select of blob and clob
void GetRecord()
 char sss[SSS LENGTH];
```

```
if(GetSSS(sss) == TRUE)
   {
    OCIClobLocator *license_txt;
     char name [NAME LENGTH] = \{ ' \setminus 0' \};
    EXEC SQL ALLOCATE : license txt;
    EXEC SQL SELECT name, txt summary INTO :name, :license txt
   FROM license table WHERE sss = :sss;
    printf("======\n\n");
    printf("NAME: %s\tSSS: %s\n", name, sss);
    PrintCrime(license_txt);
    printf("\n\n======\n");
    EXEC SQL FREE :license_txt;
   }
 else
    printf("SSS Number Not Found\n");
}
* NAME
* NewRecord
* DESCRIPTION
* Create new record in database
* LOB FEATURES
* EMPTY CLOB() and OCILobWrite
void NewRecord()
 char sss[SSS_LENGTH], name[NAME_LENGTH] = {'\0'};
 if(GetSSS(sss) == TRUE)
    printf("Record with that sss number already exists.\n");
   }
 else
    OCIClobLocator *license txt;
    EXEC SQL ALLOCATE : license txt;
    GetName(name);
    EXEC SQL INSERT INTO license table
   VALUES (:sss, :name, empty clob());
    EXEC SQL SELECT TXT SUMMARY INTO :license txt FROM LICENSE TABLE
   WHERE SSS = :sss;
    printf("======\n\n");
    printf("NAME: %s\tSSS: %s\n", name, sss);
    PrintCrime(license_txt);
    printf("\n\n======\n");
    EXEC SQL FREE :license txt;
   }
```

```
* NAME
 * NewCrime
* DESCRIPTION
 * Query user for new crime
 * LOB FEATURES
   None
 */
char *NewCrime()
 int SuggestedCrimeNo;
 int i;
 char crime[10];
 printf("Select from the following:\n");
  for (i = 1; i \le MAXCRIME; i++)
   printf("(%d) %s\n", i, CrimeList[i-1]);
 printf("Crime (1-5): ");
  fgets(crime, 10, stdin);
  SuggestedCrimeNo = atoi(crime);
  while((SuggestedCrimeNo < 1) || (SuggestedCrimeNo > MAXCRIME))
   {
     printf("Invalid selection\n");
     printf("Crime (1-5): ");
     fgets(crime, 10, stdin);
     SuggestedCrimeNo = atoi(crime);
 return CrimeList[SuggestedCrimeNo-1];
/*
* NAME
 * AppendToClob
 * DESCRIPTION
   Append String charbuf to a Clob in the following way:
   if the contents of the clob a clob were <foo> and the
   contents of charbuf were <bar>, after the append a clob
 * will contain: <foo>\n<bar> - <curdate>
   where <curdate> is today's date as obtained by the
   GetDate procedure.
 * LOB FEATURES
  OCILobWrite
 * NOTE
   Potentially, charbuf can be a very large string buffer.
    Furthermore, it should be noted that lobs and lob
    performance were designed for large data. Therefore,
    users are encouraged to read and write large chunks of
    data to lobs.
void AppendToClob(a clob, charbuf)
 OCIClobLocator *a_clob;
 char *charbuf;
 ub4 ClobLen, WriteAmt, Offset;
 int CharLen = strlen(charbuf);
```

```
int NewCharbufLen = CharLen + DATELENGTH + 4;
 varchar *NewCharbuf;
 NewCharbuf = (varchar *) malloc(2 + NewCharbufLen);
 NewCharbuf->arr[0] = '\n';
 NewCharbuf->arr[1] = ' \setminus 0';
 strcat((char *)NewCharbuf->arr, charbuf);
 NewCharbuf->arr[CharLen + 1] = '\0';
 strcat((char *)NewCharbuf->arr, curdate);
 NewCharbuf->len = NewCharbufLen;
 EXEC SQL LOB DESCRIBE :a clob GET LENGTH INTO :ClobLen;
 WriteAmt = NewCharbufLen;
 Offset = ClobLen + 1;
 EXEC SQL LOB WRITE ONE :WriteAmt FROM :NewCharbuf
   WITH LENGTH : NewCharbufLen INTO :a clob AT :Offset;
 free (NewCharbuf);
}
* NAME
* AddCrime
* DESCRIPTION
* Add a crime to a citizen's crime file
* LOB FEATURES
* OCILobWrite
void AddCrime()
 char sss[SSS LENGTH];
 if (GetSSS(sss) == TRUE)
     OCIClobLocator *license txt;
     char *crimebuf;
     char name[NAME LENGTH] = {'\0'};
     EXEC SQL ALLOCATE :license_txt;
     EXEC SQL SELECT txt summary INTO :license txt FROM license table
   WHERE sss = :sss FOR UPDATE;
     crimebuf = NewCrime();
     printf("Added %s to CrimeList\n", crimebuf);
     AppendToClob(license txt, crimebuf);
     EXEC SQL SELECT name INTO :name FROM license table WHERE sss = :sss;
     printf("NAME: %s SSS: %s\n", name, sss);
     PrintCrime(license txt);
     EXEC SQL COMMIT;
     EXEC SQL FREE :license_txt;
 else
```

```
printf("SSS Number Not Found\n");
/*
* NAME
 * GetSSS
 * DESCRIPTION
    Fills the passed buffer with a client-supplied social security number
   Returns FALSE if sss does not correspond to any entry in the database,
    else returns TRUE
 * LOB FEATURES
   none
 */
boolean GetSSS(suggested sss)
 char *suggested sss;
 int count = 0;
 int i;
 printf("Social Security Number: ");
 fgets(suggested sss, SSS LENGTH, stdin);
  for(i = 0; ((suggested sss[i] != '\0') && (i < SSS LENGTH)); i++)
     if(suggested_sss[i] == '\n')
    suggested sss[i]='\0';
  EXEC SQL SELECT COUNT(*) INTO :count FROM license table
   WHERE sss = :suggested_sss;
 return (count != 0);
}
/*
* NAME
* GetName
* DESCRIPTION
 * Get name from user.
 * LOB FEATURES
   none
void GetName(name holder)
 char *name holder;
 int count=0;
 int i;
 printf("Enter Name: ");
  fgets(name holder, NAME LENGTH + 1, stdin);
  for(i = 0; name holder[i] != '\0'; i++)
     if(name_holder[i] == '\n')
    name_holder[i]='\0';
```

```
return;
}
/*
* NAME
* PrintSQLError
* DESCRIPTION
 * Prints an error message using info in sqlca and calls exit.
 * COLLECTION FEATURES
 * none
void PrintSQLError()
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 printf("SQL error occurred...\n");
 printf("%.*s\n", (int)sqlca.sqlerrm.sqlerrml,
        (CONST char *)sqlca.sqlerrm.sqlerrmc);
 EXEC SQL ROLLBACK RELEASE;
  exit(EX_FAILURE);
}
```

17

# Objects

This chapter describes the support in Pro\*C/C++ for user-defined objects. This chapter contains the following topics:

- · Introduction to Objects
- About Using Object Types in Pro\*C/C++
- The Object Cache
- Associative Interface
- Navigational Interface
- Converting Object Attributes and C Types
- Object Options Set/Get
- New Precompiler Options for Objects
- An Object Example in Pro\*C/C++
- Example Code for Type Inheritance
- Example Code for Navigational Access
- About Using C Structures
- About Using REFs
- About Using OCIDate OCIString OCINumber and OCIRaw
- Summarizing the New Database Types in Pro\*C/C++
- Restrictions on Using Oracle Datatypes in Dynamic SQL

# 17.1 Introduction to Objects

In addition to the Oracle relational datatypes supported since Oracle8, Pro\*C/C++ supports user-defined datatypes, which are:

- Object types
- REFs to object types
- Collection object types
- Type inheritance

## **Related Topics**

Collections

# 17.1.1 Object Types

An object type is a user-defined datatype that has *attributes*, the variables that form the datatype defined by a CREATE TYPE SQL statement, and *methods*, functions and procedures that are the set of allowed behaviors of the object type. We consider object types with only attributes in this guide.

## For example:

LONG, LONG RAW, NCLOB, NCHAR and NCHAR Varying are *not* allowed as datatypes in attributes of objects.

# 17.1.2 REFs to Object Types

REF (short for "reference") is a reference to an object stored in a database table, instead of the object itself. REF types can occur in relational columns and also as datatypes of an object type. For example, a table <code>employee\_tab</code> can have a column that is a REF to an object type <code>employee\_t</code> itself:

# 17.1.3 Type Inheritance

Oracle supports type inheritance of objects. This enables sharing of attributes and methods between similar object types, as well as extending the characteristics of an object type.

Pro\*C/C++ supports type inheritance of object types with the following SQL operators:

- IS OF type
- TREAT

The IS OF *type* operator is used to test an object instance for specific *type* information.

The following code example returns the references to all p objects where p is of type <code>Employee t and Student t.</code>

```
SELECT REF(p)
  FROM person_tab p
  WHERE VALUE(p) IS OF (Employee t, Student t);
```

The following code example returns all rows where p is of type Student t only.

```
SELECT VALUE(p)
  FROM person_tab p
  WHERE VALUE(p) IS OF (ONLY Student t);
```

The TREAT operator is used to modify the declared type of an expression.

The following code example returns all rows where p is of type  $Student_t$ . For all instances of p that are not of type  $Student_t$ , null is returned.

```
SELECT TREAT(VALUE(p) AS Student_t)
FROM person tab p;
```

The following code example returns the references to all p objects of type Student\_t and any objects of subtype PartTimeStudent t.

```
SELECT TREAT(REP(p) AS REF Student_t)
FROM person_tab p
WHERE VALUE(p) IS OF (Student t);
```

# 17.2 About Using Object Types in Pro\*C/C++

Declare pointers to C structures generated by the OTT (Object Type Translator) as host and indicator variables in your Pro\*C/C++ application. Use of an indicator variable is optional for an object type, but Oracle recommends it.

Represent object types in a Pro\*C/C++ program as C structures generated from the database objects using OTT. You must

- Include in the Pro\*C/C++ program the OTT-generated header file with structure definitions and the associated NULL indicator structures, and the C type for a REF to the object type.
- Enter the typefile generated by OTT as an INTYPE Pro\*C/C++ command-line option. This
  typefile encodes the correspondence between the OTT-generated C structures and the
  associated object types in the database, as well as schema and type version information.

# **Related Topics**

The Object Type Translator

# 17.2.1 NULL Indicators

C structures representing the NULL status of object types are generated by the Object Type Translator. You must use these generated structure types in declaring indicator variables for object types.

Other Oracle types do not require special treatment for NULL indicators.

Because object types have internal structure, NULL indicators for object types also have internal structure. A NULL indicator structure for a non-collection object type provides atomic (single) NULL status for the object type as a whole, as well as the NULL status of every attribute. OTT generates a C structure to represent the NULL indicator structure for the object type. The name of the NULL indicator structure is <code>Object\_typename\_ind</code> where <code>Object\_typename</code> is the name of the C structure for the user-defined type in the database.

## **Related Topics**

Datatypes and Host Variables

# 17.3 The Object Cache

The object cache is an area of memory on the client that is allocated for your program's use in interfacing with database objects. There are two interfaces to working with objects. The associative interface manipulates "transient" copies of the objects and the navigational interface manipulates "persistent" objects.

# 17.3.1 Persistent Versus Transient Copies of Objects

Objects that you allocated in the cache with EXEC SQL ALLOCATE statements in Pro\*C/C++ are *transient copies* of persistent objects in the Oracle database. As such, you can update these copies in the cache after they are fetched in, but in order to make these changes persistent in the database, you must use explicit SQL commands. This "transient copy" or "value-based" object caching model is an extension of the relational model, in which scalar columns of relational tables can be fetched into host variables, updated in place, and the updates communicated to the server.

# 17.4 Associative Interface

The associative interface manipulates transient copies of objects. Memory is allocated in the object cache with the EXEC SQL ALLOCATE statement.

One object cache is created for each SQLLIB runtime context.

Objects are retrieved by the EXEC SQL SELECT or EXEC SQL FETCH statements. These statements set values for the attributes of the host variable. If a NULL indicator is provided, it is also set.

Objects are inserted, updated, or deleted using EXEC SQL INSERT, EXEC SQL UPDATE, and EXEC SQL DELETE statements. The attributes of the object host variable must be set before the statement is executed.

Transactional statements EXEC SQL COMMIT and EXEC SQL ROLLBACK are used to write the changes permanently on the server or to cancel the changes.

You explicitly free memory in the cache for the objects by use of the EXEC SQL FREE statement. When a connection is terminated, Oracle implicitly frees its allocated memory.

# 17.4.1 When to Use the Associative Interface

Use in these cases:

- To access large collections of objects where explicit joins between tables are not expensive.
- To access objects that are not referenceable; they do not have object identity. For example, an object type in a relational column.
- When an operation such as UPDATE or INSERT is applied to a set of objects. For example, add a bonus of \$1000 to all employees in a department.

# 17.4.2 ALLOCATE

You allocate space in the object cache with this statement. The syntax is:

```
EXEC SQL [AT [:] database] ALLOCATE :host ptr [[INDICATOR]:ind ptr] ;
```

Variables entered are:

database (IN)

a zero-terminated string containing the name of the database connection, as established previously through the statement:

```
EXEC SQL CONNECT :user [AT [:]database];
```



If the AT clause AT is omitted, or if *database* is an empty string, the default database connection is assumed.

host\_ptr (IN)

a pointer to a host structure generated by OTT for object types, collection object types, or REFs, or a pointer to one of the new C datatypes: OCIDate, OCINumber, OCIRaw, or OCIString.

ind\_ptr (IN)

The indicator variable, *ind\_ptr*, is optional, as is the keyword INDICATOR. Only pointers to struct-typed indicators can be used in the ALLOCATE and FREE statements.

host\_ptr and ind\_ptr can be host arrays.

The duration of allocation is the session. Any instances will be freed when the session (connection) is terminated, even if not explicitly freed by a FREE statement.

#### **Related Topics**

- ALLOCATE (Executable Embedded SQL Extension)
- FREE (Executable Embedded SQL Extension)

## 17.4.3 FREE

```
EXEC SQL [AT[:]database] [OBJECT] FREE :host ptr [[INDICATOR]:ind ptr];
```

You de-allocate the space for an object that is placed in the object cache using the FREE statement. Variables used are the same as in the ALLOCATE statement.



Pointers to host and indicator variables are *not* set to null.

## 17.4.4 CACHE FREE ALL

```
EXEC SQL [AT [:] database] [OBJECT] CACHE FREE ALL;
```

Use the earlier statement to free all object cache memory for the specified database connection.

#### **Related Topics**

CACHE FREE ALL (Executable Embedded SQL Extension)

## 17.4.5 Accessing Objects Using the Associative Interface

When accessing objects using SQL, Pro\*C/C++ applications manipulate transient copies of the persistent objects. This is a direct extension of the relational access interface, which uses SELECT, UPDATE and DELETE statements.

In Figure 17-1, you allocate memory in the cache for a transient copy of the persistent object with the ALLOCATE statement. The allocated object does not contain data, but it has the form of the struct generated by the OTT.

```
person *per_p;
...
EXEC SQL ALLOCATE :per_p;
```

You can execute a SELECT statement to populate the cache. Or, use a FETCH statement or a C assignment to populate the cache with data.

```
EXEC SQL SELECT ... INTO :per p FROM person tab WHERE ...
```

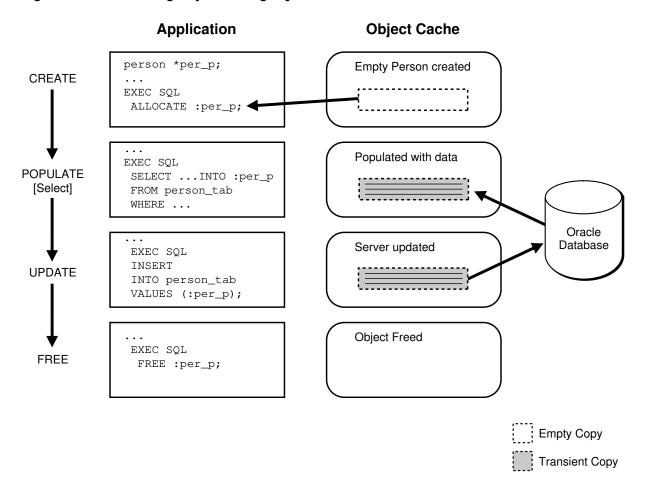
Make changes to the server objects with INSERT, UPDATE or DELETE statements, as shown in the illustration. You can insert the data is into the table by the INSERT statement:

```
EXEC SQL INSERT INTO person_tab VALUES(:per_p);
```

Finally, free memory associated with the copy of the object with the FREE statement:

```
EXEC SQL FREE :per p;
```

Figure 17-1 Accessing Objects Using SQL



# 17.5 Navigational Interface

Use the navigational interface to access the same schema as the associative interface. The navigational interface accesses objects, both persistent and transient) by dereferencing REFs to objects and traversing ("navigating") from one object to another. Some definitions follow.

Pinning an object is the term used to mean dereferencing the object, allowing the program to access it.

Unpinning means indicating to the cache that the object is no longer needed.

Dereferencing can be defined as the server using the REF to create a version of the object in the client. While the cache maintains the association between objects in the cache and the corresponding server objects, it does not provide automatic coherency. You have the responsibility to ensure correctness and consistency of the contents of the objects in the cache.

*Releasing* an object copy indicates to the cache that the object is not currently being used. To free memory, release objects when they are no longer needed to make them eligible for implicit freeing.

Freeing an object copy removes it from the cache and releases its memory area.

*Marking* an object tells the cache that the object copy has been updated in the cache and the corresponding server object must be updated when the object copy is flushed.

Un-marking an object removes the indication that the object has been updated.

Flushing an object writes local changes made to marked copies in the cache to the corresponding objects in the server. The object copies in the cache are also unmarked at this time.

Refreshing an object copy in the cache replaces it with the latest value of the corresponding object in the server.

The navigational and associative interfaces can be used together.

Use the EXEC SQL OBJECT statements, the navigational interface, to update, delete, and flush cache copies (write changes in the cache to the server).



"Example Code for Navigational Access" for an illustration of using the navigational and associative interfaces together

## 17.5.1 When to Use the Navigational Interface

Use the navigational interface:

- To access a single or small set of objects where explicit joins between tables are expensive. When you use dereferencing to navigate between objects, you perform implicit joins which are less expensive than an explicit join across two entire tables.
- To make many small changes to many different objects. It is more convenient to fetch all
  objects to the client, make changes, mark them as updated, and flush all the changes back
  to the server.

## 17.5.2 Rules Used in the Navigational Statements

Embedded SQL OBJECT statements are described later with these assumptions:

- If an AT clause is absent, the default (unnamed) connection is assumed.
- Host variables can be arrays, except where specifically noted.



- Use the FOR clause to explicitly specify the array dimension. If absent, the minimum dimension of the pertinent host variables is used.
- After execution of the statement, if the SQLCA is provided as a status variable, the number
  of elements processed is returned in sqlca.sqlerrd[2].
- Parameters have IN or OUT (or both) specified to signify input or output.

#### **Related Topics**

Embedded SQL Statements and Directives

## 17.5.3 OBJECT CREATE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT CREATE :obj [INDICATOR]: obj ind [TABLE tab] [RETURNING REF INTO :ref];
```

where tab is:

{:hv | [schema.]table}

Use this statement to create a referenceable object in the object cache. The type of the object corresponds to the host variable *obj*. When optional type host variables (:obj ind,:ref,:ref ind) are supplied, they must all correspond to the same type.

The referenceable object can be either persistent (TABLE clause is supplied) or transient (TABLE clause is absent). Persistent objects are implicitly pinned and marked as updated. Transient objects are implicitly pinned.

The host variables are:

obj (OUT)

The object instance host variable, *obj*, must be a pointer to a structure generated by OTT. This variable is used to determine the referenceable object that is created in the object cache. After a successful execution, *obj* will point to the newly created object.

```
obj ind (OUT)
```

This variable points to an OTT-generated indicator structure. Its type must match that of the object instance host variable. After a successful execution, *obj\_ind* will be a pointer to the parallel indicator structure for the referenceable object.

tab (IN)

Use the table clause to create persistent objects. The table name can be specified as a host variable, *hv*, or as an undeclared SQL identifier. It can be qualified with a schema name. Do not use trailing spaces in host variables containing the table name.

hv (IN)

A host variable specifying a table. If a host variable is used, *it must not* be an array. It must not be blank-padded. It is case-sensitive. When an array of persistent objects is created, they are all associated with the same table.

table (IN)

An undeclared SQL identifier which is case-sensitive.

ref (OUT)



The reference host variable must be a pointer to the OTT-generated reference type. The type of *ref* must match that of the object instance host variable. After execution, *ref* contains a pointer to the ref for the newly created object.

Attributes are initially set to null. Creating new objects for object views is not currently supported.

Creating new objects for object views is *not* currently supported.

#### 17.5.4 OBJECT DEREF

```
EXEC SQL [AT [:] database] [FOR [:] count] OBJECT DEREF :ref INTO :obj [[INDICATOR]:obj_ind ] [FOR UPDATE [NOWAIT]];
```

Given an object reference, *ref*, the OBJECT DEREF statement pins the corresponding object or array of objects in the object cache. Pointers to these objects are returned in the variables *obj* and *obj ind*.

The host variables are:

ref (IN)

This is the object reference variable, which must be a pointer to the OTT-generated reference type. This variable (or array of variables) is dereferenced, returning a pointer to the corresponding object in the cache.

obj (OUT)

The object instance host variable, *obj*, must be a pointer to an OTT-generated structure. Its type must match that of the object reference host variable. After successful execution, *obj* contains a pointer to the pinned object in the object cache.

```
obj_ind (OUT)
```

The object instance indicator variable, *obj\_ind*, must be a pointer to an OTT-generated indicator structure. Its type must match that of the object reference indicator variable. After successful execution, *obj\_ind* contains a pointer to the parallel indicator structure for the referenceable object.

#### FOR UPDATE

If this clause is present, an exclusive lock is obtained for the corresponding object in the server.

**NOWAIT** 

If this optional keyword is present, an error is immediately returned if another user has already locked the object.

## 17.5.5 OBJECT RELEASE

```
EXEC SQL [AT [:] database] [FOR [:] count] OBJECT RELEASE : obj ;
```

This statement unpins the object in the object cache. When an object is not pinned and not updated, it is eligible for implicit freeing.

If an object has been dereferenced n times, it must be released n times to be eligible for implicit freeing from the object cache. Oracle advises releasing all objects that are no longer needed.

## 17.5.6 OBJECT DELETE

```
EXEC SQL [AT [:] database] [FOR [:] count] OBJECT DELETE : obj ;
```

For persistent objects, this statement marks an object or array of objects as deleted in the object cache. The object is deleted in the server when the object is flushed or when the cache is flushed. The memory reserved in the object cache is not freed.

For transient objects, the object is marked as deleted. The memory for the object is not freed.

## 17.5.7 OBJECT UPDATE

```
EXEC SQL [AT [:] database] [FOR [:] count] OBJECT UPDATE : obj ;
```

For persistent objects, this statement marks them as updated in the object cache. The changes are written to the server when the object is flushed or when the cache is flushed.

For transient objects, this statement is a no-op.

## 17.5.8 OBJECT FLUSH

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT FLUSH :obj ;
```

This statement flushes persistent objects that have been marked as updated, deleted, or created, to the server.



An exclusive lock is implicitly obtained when the object is flushed. After the statement successfully completes, the objects are unmarked. If the object version is LATEST (see next section), then the object will be implicitly refreshed.

# 17.5.9 Navigational Access to Objects

See Figure 17-2 for an illustration of the navigational interface.

Use the ALLOCATE statement to allocate memory in the object cache for a copy of the REF to the *person* object. The allocated REF does not contain data.

```
person *per_p;
person_ref *per_ref_p;
...
EXEC SQL ALLOCATE :per p;
```

Populate the allocated memory by using a SELECT statement to retrieve the REF of the *person* object (exact format depends on the application):

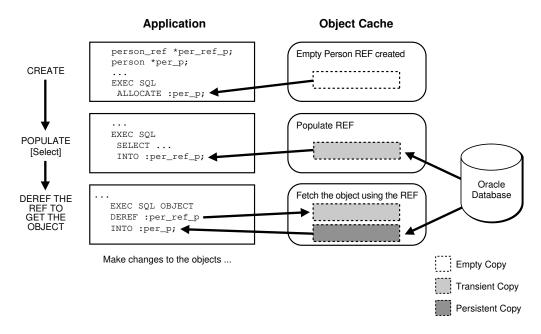
```
EXEC SQL SELECT ... INTO :per ref p;
```

The DEREF statement is then used to pin the object in the cache, so that changes can be made in the object. The DEREF statement takes the pointer  $per\_ref\_p$  and creates an instance of the person object in the client-side cache. The pointer  $per\_p$  to the person object is returned.

```
EXEC SQL OBJECT DEREF :per_ref_p INTO :per_p;
```



Figure 17-2 Navigational Access



Make changes to the object in the cache by using C assignment statements, or by using data conversions with the OBJECT SET statement.

Then you must mark the object as updated. See Figure 17-3. To mark the object in the cache as updated, and eligible to be flushed to the server:

```
EXEC SQL OBJECT UPDATE :per p;
```

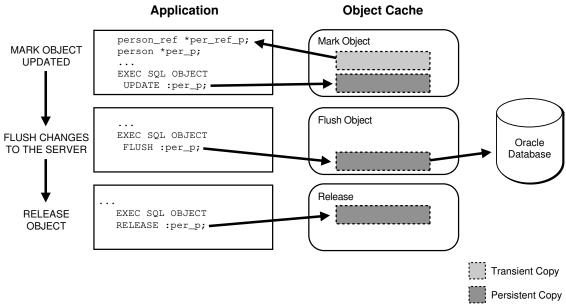
You send changes to the server by the FLUSH statement:

```
EXEC SQL OBJECT FLUSH :per_p;
```

#### You release the object:

EXEC SQL OBJECT RELEASE :per p;

Figure 17-3 Navigational Access (continued)



The statements in the next section are used to make the conversions between object attributes and C types.

# 17.6 Converting Object Attributes and C Types

This section covers issues relating to attribute and type conversion.

## 17.6.1 OBJECT SET

```
EXEC SQL [AT [:]database]
  OBJECT SET [ {'*' | {attr[, attr]} } OF]
  :obj [[INDICATOR]:obj_ind]
    TO {:hv [[INDICATOR]:hv_ind]
        [, :hv [INDICATOR]:hv ind]]};
```

Use this statement with objects created by both the associative and the navigational interfaces. This statement updates the attributes of the object. For persistent objects, the changes will be written to the server when the object is updated and flushed. Flushing the cache writes all changes made to updated objects to the server.

The OF clause is optional. If absent, all the attributes of *obj* are set. The same result is achieved by writing:

```
... OBJECT SET * OF ...
```

The host variable list can include structures that are exploded to provide values for the attributes. However, the number of attributes in *obj* must match the number of elements in the exploded variable list.

Host variables and attributes are:

attr

The attributes are not host variables, but rather simple identifiers that specify which attributes of the object will be updated. The first attribute in the list is paired with the first expression in the list, and so on. The attribute must be one of either OCIString, OCINumber, OCIDate, or OCIRef.

#### obj (IN/OUT)

*obj* specifies the object to be updated. The bind variable *obj* must not be an array. It must be a pointer to an OTT-generated structure.

```
obj_ind (IN/OUT)
```

The parallel indicator structure that will be updated. It must be a pointer to an OTT-generated indicator structure.

```
hv (IN)
```

This is the bind variable used as input to the OBJECT SET statement. *hv* must be an int, float, OCIRef \*, a one-dimensional char array, or a structure of these types.

```
hv_ind (IN)
```

This is the associated indicator that is used as input to the OBJECT SET statement. *hv\_ind* must be a 2-byte integer scalar or a structure of 2-byte integer scalars.

Using Indicator Variables:

If a host variable indicator is present, then an object indicator must also be present.

If hv\_ind is set to -1, the associated field in the obj\_ind is set to -1.

The following implicit conversions are permitted:

- [OCIString | STRING | VARCHAR | CHARZ] to OCIString
- OCIRef to OCIRef
- [OCINumber | int | float | double] to OCINumber
- [OCIDate | STRING | VARCHAR | CHARZ ] to OCIDate

#### Note:

- Nested structures are not allowed.
- This statement cannot be used to set a referenceable object to be atomically NULL. Set the appropriate field of the NULL indicator instead.
- Conversions between the OCIDateTime or OCIInterval datatypes and OCIString are not supported.

## 17.6.2 OBJECT GET

This statement converts the attributes of an object into native C types.

The FROM clause is optional. If absent, all the attributes of *obj* are converted. The same result is achieved by writing:

```
... OBJECT GET * FROM ...
```

The host variable list may include structures that are exploded to receive the values of the attributes. However, the number of attributes in *obj* must match the number of elements in the exploded host variable list.

Host variables and attributes:

attr

The attributes are not host variables, but simple identifiers that specify which attributes of the object will be retrieved. The first attribute in the list is paired with the first host variable in the list, and so on. The attribute must represent a base type. It must be OCIString, OCINumber, OCIRef, or OCIDate.

obj (IN)

This specifies the object that serves as the source for the attribute retrieval. The bind variable *obj* must not be an array.

hv (OUT)

This is the bind variable used to hold output from the OBJECT GET statement. It can be an int, float, double, a one-dimensional char array, or a structure containing those types. The statement returns the converted attribute value in this host variable.

hv ind (OUT)

This is the associated indicator variable for the attribute value. It is a 2-byte integer scalar or a structure of 2-byte integer scalars.

Using Indicator Variables:

If no object indicator is specified, it is assumed that the attribute is valid. *It is a program error* to convert object attributes to C types if the object is atomically NULL or if the requested attribute is NULL and no object indicator variable is supplied. *It may not be possible to raise an Oracle error in this situation.* 

If the object variable is atomically NULL or the requested attribute is NULL, and a host variable indicator (*hv\_ind*) is supplied, then it is set to -1.

If the object is atomically NULL or the requested attribute is NULL, and no host variable indicator is supplied, then an error is raised.

The following implicit conversions are permitted:

- OCIString to [STRING | VARCHAR | CHARZ | OCIString]
- OCINumber to [int | float | double | OCINumber]
- OCIRef to OCIRef
- OCIDate to [STRING | VARCHAR | CHARZ | OCIDate]





- Nested structures are not allowed
- Conversions between the OCIDateTime or OCIInterval datatypes and OCIString are not supported

# 17.7 Object Options Set/Get

The runtime context has options which are set to default values when the runtime context is created and allocated. You can then set these options with this embedded SQL directive:

## 17.7.1 CONTEXT OBJECT OPTION SET

```
EXEC SQL CONTEXT OBJECT OPTION SET \{option[, option]\}\ TO \{:hv[, :hv]\}\ ; where the variables are: :hv(IN) ...
The input bind variables hv ..., are of type STRING, VARCHAR, or CHARZ. option ...
```

Simple identifiers that specify which option of the runtime context to update. The first option is paired with the first input bind variable, and so on. Here are the values supported at this time:

Table 17-1 Valid Choices for CONTEXT OBJECT OPTION Values

Option Value	Specifies		
DATEFORMAT	Format for Date attributes and collection elements.		
DATELANG	Globalization Support language for all Date and Datetime types.		

#### An example is:

```
char *new_format = "DD-MM-YYYY";
char *new_lang = "French";
char *new_date = "14-07-1789";
/* One of the attributes of the license type is dateofbirth */
license *aLicense;
...
/* Declaration and allocation of context ... */
EXEC SQL CONTEXT OBJECT OPTION SET DATEFORMAT, DATELANG TO :new_format, :new_lang;
/* Navigational object obtained */
...
EXEC SQL OBJECT SET dateofbirth OF :aLicense TO :new_date;
```

#### **Related Topics**

- Oracle Database SQL Language Reference
- CONTEXT OBJECT OPTION SET (Executable Embedded SQL Ext)

### 17.7.2 CONTEXT OBJECT OPTION GET

The context affected is understood to be the context in use at the time. To determine the values of these options, use this directive:

```
EXEC SQL CONTEXT OBJECT OPTION GET {option[, option]} INTO {:hv[, :hv]};
```

Where the values of option are found in Table 17-1.

The bind variables, *hv* ... are used as output, and are of type STRING, VARCHAR, or CHARZ. The context affected is understood to be the context in use at the time.

#### **Related Topics**

- Oracle Database SQL Language Reference
- CONTEXT OBJECT OPTION GET (Executable Embedded SQL Extension)

# 17.8 New Precompiler Options for Objects

To support objects, use these precompiler options:

#### 17.8.1 VERSION

This option determines which version of the object is returned by the EXEC SQL OBJECT DEREF statement. This gives you varying levels of consistency between cache objects and server objects.

Use the EXEC ORACLE OPTION statement to set it inline. Permitted values are:

#### **RECENT (default)**

If the object has been selected into the object cache in the current transaction, then return that object. If the object has not been selected, it is retrieved from the server. For transactions that are running in serializable mode, this option has the same behavior as VERSION=LATEST without incurring as many network round trips. This value can be safely used with most Pro\*C/C++ applications.

#### **LATEST**

If the object does not reside in the object cache, it is retrieved from the database. If it does reside in the object cache, it is refreshed from the server. Use this value with caution because it will incur the greatest number of network round trips. Use it only when it is imperative that the object cache be kept as coherent as possible with the server-side buffer.

#### **ANY**

If the object already resides in the object cache, then return that object. If the object does not reside in the object cache, retrieve it from the server. This value will incur the fewest number of network round trips. Use in applications that access read-only objects or when a user will have exclusive access to the objects.

## 17.8.2 DURATION

Use this precompiler option to set the pin duration used by subsequent EXEC SQL OBJECT CREATE and EXEC SQL OBJECT DEREF statements. Objects in the cache are implicitly unpinned at the end of the duration.

Use with navigational interface only.



You can set this option in the EXEC ORACLE OPTION statement. Permitted values are:

#### TRANSACTION (default)

Objects are implicitly unpinned when the transaction completes.

#### **SESSION**

Objects are implicitly unpinned when the connection is terminated.

## **17.8.3 OBJECTS**

This precompiler option provides the ability to use the object cache.

The OBJECTS default value, for DBMS=NATIVE | V8, is YES. The default size of the object cache is the same as the OCI default cache size, 8 Mbytes.

#### **Related Topics**

OBJECTS

#### 17.8.4 INTYPE

If your program uses any object types, collection object types, or REFs, you must give the INTYPE files in this command-line option.

Specify the INTYPE option using the syntax:

```
INTYPE=filename1 INTYPE=filename2 ...
```

where *filename1*, and so on., is the name of the typefiles generated by OTT. These files are meant to be a read-only input to Pro\*C/C++. The information in it, though in plain-text form, might be encoded, and might not necessarily be interpretable by you, the user.

You can provide more than one INTYPE file as input to a single Pro\*C/C++ precompilation unit.

This option cannot be used inline in EXEC ORACLE statements.

OTT generates C structure declarations for object types created in the database, and writes type names and version information to a file called the *typefile*.

An object type may not necessarily have the same name as the C structure type or C++ class type that represents it. This could arise for the following reasons:

- The name of the object type specified in the server includes characters not legal in a C or C++ identifier
- The user asked OTT to use a different name for the structure or class.
- The user asked OTT to change the case of names

Under these circumstances, it is impossible to infer from the structure or class declaration which object type it matches. This information, which is required by Pro\*C/C++, is generated by OTT in the type file.

## **17.8.5 ERRTYPE**

ERRTYPE=filename

Writes errors to the file specified, as well as to the screen. If omitted, errors are directed to the screen only. Only one ERRTYPE is allowed. As is usual with other single-valued command-line

options, if you enter multiple values for ERRTYPE on the command line, the last one supersedes the earlier values.

This option cannot be used inline in EXEC ORACLE statements.

## 17.8.6 SQLCHECK Support for Objects

Object types and their attributes are represented in a C program according to the C binding of Oracle types. If the precompiler command-line option SQLCHECK is set to SEMANTICS or FULL, Pro\*C/C++ verifies during precompilation that host variable types conform to the mandated C bindings for the types in the database schema. In addition, runtime checks are always performed to verify that Oracle types are mapped correctly during program execution.

Relational datatypes are checked in the usual manner.

A relational SQL datatype is compatible with a host variable type if the two types are the same, or if a conversion is permitted between the two. Object types, on the other hand, are compatible only if they are the same type. They must

- Have the same name
- Be in the same schema (if a schema is explicitly specified)

When you specify the option SQLCHECK=SEMANTICS or FULL, during precompilation Pro\*C/C++ logs onto the database using the specified userid and password, and verifies that the object type from which a structure declaration was generated is identical to the object type used in the embedded SQL statement.

## 17.8.7 Type Checking at Runtime

Pro\*C/C++ gathers the type name, version, and possibly schema information for Object, collection Object, and REF host variables, for a type from the input INTYPE file, and stores this information in the code that it generates. This enables access to the type information for Object and REF bind variables at runtime. Appropriate errors are returned for type mismatches.

# 17.9 An Object Example in Pro\*C/C++

Let us examine a simple object example. You create a type *person* and a table *person\_tab*, which has a column that is also an object type, *address*:

Insert data in the table, and proceed.

## 17.9.1 Associative Access

Consider the case of how to change a *lastname* value from "Smith" to "Smythe", using Pro\*C/C++.

Run the OTT to generate C structures which map to *person*. In your Pro\*C/C++ program you must include the header file generated by OTT.

In your application, declare a pointer, *person\_p*, to the persistent memory in the client-side cache. Then allocate memory and use the returned pointer:

```
char *new_name = "Smythe";
person *person_p;
...
EXEC SQL ALLOCATE :person_p;
```

Memory is now allocated for a copy of the persistent object. The allocated object does not yet contain data.

Populate data in the cache either by C assignment statements or by using SELECT or FETCH to retrieve an existing object:

```
EXEC SQL SELECT VALUE(p) INTO :person p FROM person tab p WHERE lastname = 'Smith';
```

Changes made to the copy in the cache are transmitted to the server database by use of INSERT, UPDATE, and DELETE statements:

```
EXEC SQL OBJECT SET lastname OF :person_p TO :new_name;
EXEC SQL INSERT INTO person_tab VALUES(:person_p);
```

Free cache memory in this way:

```
EXEC SQL FREE :person p;
```

## 17.9.2 Navigational Access

Allocate memory in the object cache for a copy of the REF to the object *person*. The ALLOCATE statement returns a pointer to the REF:

```
person *person_p;
person_ref *per_ref_p;
...
EXEC SQL ALLOCATE :per ref p;
```

The allocated REF contains no data. To populate it with data, retrieve the REF of the object:

```
EXEC SQL SELECT ... INTO :per_ref_p;
```

Then dereference the REF to put an instance of object in the client-side cache. The dereference command takes the *per\_ref\_p* and creates an instance of the corresponding object in the cache:

```
EXEC SQL OBJECT DEREF :per ref p INTO :person p;
```

Make changes to data in the cache by using C assignments, or by using OBJECT GET statements:

```
/* lname is a C variable to hold the result */
EXEC SQL OBJECT GET lastname FROM :person_p INTO :lname;
...
EXEC SQL OBJECT SET lastname OF :person_p TO :new_name;
/* Mark the changed object as changed with OBJECT UPDATE command */;
EXEC SQL OBJECT UPDATE :person_p;
EXEC SQL FREE :per ref p;
```

To make the changes permanent in the database, use FLUSH:

```
EXEC SQL OBJECT FLUSH :person_p;
```



Changes have been made to the server; the object can now be released. Objects that are released are not necessarily freed from the object cache memory immediately. They are placed on a least-recently used stack. When the cache is full, the objects are swapped out of memory.

Only the object is released; the REF to the object remains in the cache. To release the REF, use the RELEASE statement. for the REF. To release the object pointed to by person\_p:

```
EXEC SQL OBJECT RELEASE :person p;
```

Or, issue a transaction commit and all objects in the cache are released, provided the pin duration has been set appropriately.

# 17.10 Example Code for Type Inheritance

The following code example creates four object types:

- Person t
- Employee t as a subtype of Person t
- Student t as a subtype of Person t
- PartTimeStudent\_t as a subtype of Student\_t

#### and one table:

person\_tab to hold Person\_t and its subtype objects

The SQL file, inhdemol.sql, that creates the object types and table, and then inserts values into the table, is:

```
connect scott/tiger;
rem ** Always drop your objects in reverse dependency order
drop table person tab;
drop type PartTimeStudent t;
drop type Student t;
drop type Employee t;
drop type Person t;
rem ** Create the TYPES, TYPED TABLES and TABLES we need
rem ** Create a Person t ADT
CREATE TYPE Person t AS OBJECT
( ssn NUMBER,
 name VARCHAR2(30),
 address VARCHAR2(100)) NOT FINAL;
rem ** Create a Person t subtype Employee t
CREATE TYPE Employee t UNDER Person t
( empid NUMBER,
 mgr VARCHAR2(30));
rem ** Create a Person_t subtype Student_t
CREATE TYPE Student t UNDER Person t
( deptid NUMBER,
  major VARCHAR2(30)) NOT FINAL;
```

```
rem ** Create a Student_t subtype PartTimeStudent_t
CREATE TYPE PartTimeStudent t UNDER Student t
( numhours NUMBER);
rem ** Create a typed table for person t objects
CREATE table person tab of person t;
rem ** Insert 2 Employee t objects into the person t typed table
insert into person tab values
  (Employee t(123456, 'Alison Laurence', '100 Geary Street, San Francisco, CA 94013',
1001, 'CEO'));
insert into person_tab values
  (Employee t(234567, 'William Bates', '123 Main Street, Anytown, WA 97818',
1002, 'CFO'));
rem ** Insert 2 Student t objects into the person t typed table
insert into person tab values
  (Student t(20001, 'Van Gates', '1825 Aikido Way, Los Angeles, CA, 45300', 20,
'English'));
insert into person tab values
  (Student t(20002, 'Bill Wallace', '12 Shugyo Blvd, Los Angeles, CA, 95100', 30,
'Computer Science'));
rem ** Insert 1 PartTimeStudent t object into the person t typed table
insert into person tab values
  (PartTimeStudent t(20003, 'Jean Claude', '874 Richmond Street, New York, NY 45100',
40, 'Music',20));
commit;
```

#### Here is the listing of the intype file for our example, inhdemo1.typ:

```
case=same
type person_t
type employee_t
type student_t
type parttimestudent t
```

#### Here is the listing of the precompiler file, inhdemol.pc:

```
4. Compile/Link (This step is platform specific)
    *************************
/* Include files */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlda.h>
#include <sqlca.h>
                                                 /* SOL Communications Area */
#include <sql2oci.h>
                          /* SQLLIB interoperability routines for OCI8 */
#include "inhdemo1.h"
                           /* OTT-generated header with C typedefs for the */
                                   /* database types "person" and "address" */
/* Macros */
#define ARRAY SIZE 10
#define NAME LENGTH 31
#define ADDR LENGTH 101
/* Global variables */
 char *uid="scott/tiger";
 int i;
 int count;
 VARCHAR dynstmt[100];
main()
{
  printf("\n*** STARTING OBJECT TYPE INHERITANCE DEMO ***\n");
 EXEC SQL WHENEVER SQLERROR DO sql error("ORACLE error--\n");
  EXEC SQL connect :uid;
 printf("Connected successfully.\n");
  exec sql select count(*) into :count from person tab;
  printf("\nThere are %d entries in table person tab.\n", count);
  do fetch all();
                                                 /* Fetch person t objects */
  do fetch employee();
                                               /* Fetch employee t objects */
                                           /* Fetch only student t objects */
  do fetch student();
                                         /* Fetch parttimestuden t objects */
  do fetch parttimestudent();
  do_fetch_student_employee(); /* Fetch student_t and employee_t objects */
 printf("\nFetching only student t objects with dynamic sql:\n");
  strcpy((char *)dynstmt.arr,
  "SELECT value(p) from person tab p where value(p) is of (only student t)");
  do dynamic fetch();
                               /* Fetch student t object with dynamic sql */
  printf("\nFetching student t and its subtype objects with dynamic sql:\n");
  strcpy((char *)dynstmt.arr,
  "SELECT treat(value(p) as student t) from person tab p where value(p) is
of(student t)");
 do dynamic fetch();
                                 /* Fetch student t object with dynamic sql */
 printf("\n*** END OF OBJECT TYPE INHERITANCE DEMO ***\n");
  exit(EXIT SUCCESS);
}
```

```
void printPerson(person)
 person t *person;
  int writtenSSN=-1;
  text writtenName[NAME LENGTH];
 text writtenAddr[ADDR LENGTH];
 EXEC SQL OBJECT GET SSN, NAME, ADDRESS FROM :person INTO
     :writtenSSN, :writtenName, :writtenAddr;
 printf("\nSSN=%10d\nNAME=%s\nAddr=%s\n", writtenSSN, writtenName,
          writtenAddr);
void printEmployee(employee)
  employee_t *employee;
 int writtenID=-1;
 text writtenMgr[NAME LENGTH];
 printPerson(employee);
 EXEC SQL OBJECT GET EMPID, MGR FROM :employee INTO :writtenID, :writtenMgr;
 printf("EMPID=%10d\nMGR=%s\n", writtenID, writtenMgr);
}
void printStudent(student)
  student t *student;
  int writtendeptid=-1;
  text writtenMajor[NAME LENGTH];
 printPerson(student);
 EXEC SQL OBJECT GET DEPTID, MAJOR FROM :student INTO :writtendeptid, :writtenMajor;
 printf("DEPTID=%10d\nMAJOR=%s\n", writtendeptid, writtenMajor);
void printPartTimeStudent(parttimes)
 parttimestudent_t *parttimes;
  int written numhours=-1;
 printStudent(parttimes);
 EXEC SQL OBJECT GET NUMHOURS FROM :parttimes INTO :written numhours;
 printf("NUMHOURS=%10d\n", written numhours);
}
/\star Declare error handling function. \star/
sql error(msg)
    char *msg;
    char err msg[128];
    size t buf len, msg len;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    printf("\n%s\n", msg);
    buf len = sizeof (err msg);
    sqlglm(err msg, &buf len, &msg len);
    printf("%.*s\n", msg_len, err_msg);
    EXEC SQL ROLLBACK RELEASE;
    exit(EXIT FAILURE);
```

```
/*****************************
 * The following function shows how to select person_t objects
do fetch all()
person t *personArray[ARRAY SIZE];
person t ind *personArray ind[ARRAY SIZE];
 printf("\nFetching person t objects:\n");
 exec sql declare c1 cursor for
   select value(p) from person tab p;
 exec sql allocate :personArray:personArray ind;
 exec sql open c1;
 exec sql whenever not found goto :done;
 while(sqlca.sqlcode==0)
     exec sql fetch cl into :personArray:personArray ind;
     if (sqlca.sqlcode == 1403) goto done;
     for (i=0; i < ARRAY SIZE; i++ )
       printPerson(personArray[i]);
   }
 done:
  for (i=0; i < sqlca.sqlerrd[2] % ARRAY SIZE; i++)</pre>
   printPerson(personArray[i]);
 printf("Total number of person t objects fetched: %d.\n",
         sqlca.sqlerrd[2]);
 exec sql close c1;
 exec sql free :personArray:personArray ind;
/**********************
 ^{\star} The following function shows how to select person t subtype employee t
           do_fetch_employee()
employee t *empArray[ARRAY SIZE];
employee_t_ind *empArray_ind[ARRAY_SIZE];
 printf("\nFetching employee t objects:\n");
 exec sql allocate :empArray:empArray ind;
 exec sql declare c2 cursor for
   select value(p) from person tab p
     where value(p) is of (employee t);
 exec sql open c2;
  exec sql whenever not found goto :done_emp;
 while(sqlca.sqlcode==0)
   {
```

```
exec sql fetch c2 into :empArray:empArray ind;
    for (i=0; i < ARRAY SIZE; i++ )
      printEmployee(empArray[i]);
done emp:
  for (i=0; i < sqlca.sqlerrd[2] % ARRAY SIZE; i++)</pre>
    printEmployee(empArray[i]);
  printf("Total number of employee t objects fetched: %d.\n",
        sqlca.sqlerrd[2]);
 exec sql close c2;
 exec sql free :empArray:empArray ind;
}
* The following function shows how to select person t subtype student t
do fetch student()
student t *studentArray[ARRAY SIZE];
student t ind *studentArray ind[ARRAY SIZE];
 printf("\nFetching student t objects:\n");
 exec sql declare c3 cursor for
   select value(p) from person tab p
    where value(p) is of (student t);
 exec sql allocate :studentArray:studentArray ind;
 exec sql open c3;
 exec sql whenever not found goto :done student;
 for (;;)
    exec sql fetch c3 into :studentArray:studentArray ind;
    for (i=0; i < ARRAY SIZE; i++ )
      printStudent(studentArray[i]);
done student:
 for (i=0; i < sqlca.sqlerrd[2] % ARRAY SIZE; i++)</pre>
   printStudent(studentArray[i]);
 printf("Total number of student t objects fetched: %d.\n",
        sqlca.sqlerrd[2]);
 exec sql close c3;
 exec sql free :studentArray:studentArray ind;
/*****************************
* The following function shows how to select student_t subtype
* parttimestudent objects
do fetch parttimestudent()
```

```
parttimestudent_t *parttimestudentArrayArray[ARRAY_SIZE];
parttimestudent t ind *parttimestudentArrayArray ind[ARRAY SIZE];
 printf("\nFetching parttimestudent t objects:\n");
  exec sql declare c4 cursor for
   select value(p) from person tab p
     where value(p) is of (parttimestudent t);
 exec sql allocate :parttimestudentArrayArray:parttimestudentArrayArray ind;
 exec sql open c4;
 exec sql whenever not found goto :done parttimestudent;
 while(sqlca.sqlcode==0)
     exec sql fetch c4 into :parttimestudentArrayArray:parttimestudentArrayArray ind;
     for (i=0; i < ARRAY SIZE; i++ )
       printPartTimeStudent(parttimestudentArrayArray[i]);
 done parttimestudent:
  for (i=0; i < sqlca.sqlerrd[2] % ARRAY SIZE; i++)</pre>
   printPartTimeStudent(parttimestudentArrayArray[i]);
 printf("Total number of parttimestudent t objects fetched: %d.\n",
         sqlca.sqlerrd[2]);
 exec sql close c4;
  exec sql free :parttimestudentArrayArray:parttimestudentArrayArray ind;
/******************************
 * The following function shows how to select person_t subtypes student_t
 * and employee_t objects
 do_fetch_student_employee()
person t *personArray[ARRAY SIZE];
person t ind *personArray ind[ARRAY SIZE];
 printf("\nFetching only student t and employee t objects:\n");
  exec sql declare c5 cursor for
   select value(p) from person tab p
     where value(p) is of (only student t, employee t);
  exec sql allocate :personArray:personArray ind;
  exec sql open c5;
  exec sql whenever not found goto :done student employee;
 while(sqlca.sqlcode==0)
     exec sql fetch c5 into :personArray:personArray ind;
     for (i=0; i < ARRAY SIZE; i++ )
       printPerson(personArray[i]);
 done student employee:
  for (i=0; i < sqlca.sqlerrd[2] % ARRAY SIZE; i++)</pre>
```

```
printPerson(personArray[i]);
 printf("Total number of stuent_t and employee_t objects fetched: %d.\n",
sqlca.sqlerrd[2]);
 exec sql close c5;
 exec sql free :personArray:personArray ind;
/************************
^{\star} The following function shows how to select person t subtype student t
* objects using dynamic sql.
do dynamic fetch()
student_t *student;
student t ind *student ind;
 exec sql allocate :student:student ind;
 dynstmt.len = (unsigned short)strlen((char *)dynstmt.arr);
 EXEC SQL PREPARE S FROM : dynstmt;
 EXEC SQL DECLARE C CURSOR FOR S;
 EXEC SQL OPEN C;
 exec sql whenever not found do break;
 for (;;)
    EXEC SQL FETCH C INTO :student:student ind;
    printStudent(student);
 printf("\nQuery returned %d row%s.\n", sqlca.sqlerrd[2],
        (sqlca.sqlerrd[2] == 1) ? "" : "s");
 EXEC SQL CLOSE C;
 exec sql free :student:student_ind;
```

# 17.11 Example Code for Navigational Access

The example object code creates three object types; budoka is a martial arts expert:

- Customer
- Budoka
- Location

and two tables:

- person\_tab
- customer tab

The SQL file, navdemo1.sql, which creates the types and tables, and then inserts values into the tables, is:

```
connect scott/tiger
drop table customer tab;
```

```
drop type customer;
drop table person tab;
drop type budoka;
drop type location;
create type location as object (
       num
             number,
        street varchar2(60),
       city
               varchar2(30),
        state char(2),
        zip
               char (10)
);
create type budoka as object (
       lastname varchar2(20),
       firstname varchar(20), birthdate date,
                       int,
        addr
                       location
);
create table person tab of budoka;
create type customer as object (
        account number varchar(20),
        aperson ref budoka
);
create table customer tab of customer;
insert into person_tab values (
        budoka('Seagal', 'Steven', '14-FEB-1963', 34,
                location(1825, 'Aikido Way', 'Los Angeles', 'CA', 45300)));
insert into person tab values (
        budoka('Norris', 'Chuck', '25-DEC-1952', 45,
                location(291, 'Grant Avenue', 'Hollywood', 'CA', 21003)));
insert into person tab values (
        budoka('Wallace', 'Bill', '29-FEB-1944', 53,
                location(874, 'Richmond Street', 'New York', 'NY', 45100)));
insert into person tab values (
        budoka ('Van Damme', 'Jean Claude', '12-DEC-1964', 32,
                location(12, 'Shugyo Blvd', 'Los Angeles', 'CA', 95100)));
insert into customer tab
        select 'AB123', ref(p)
         from person tab p where p.lastname = 'Seagal';
insert into customer tab
        select 'DD492', ref(p)
         from person tab p where p.lastname = 'Norris';
insert into customer tab
        select 'SM493', ref(p)
         from person tab p where p.lastname = 'Wallace';
insert into customer tab
        select 'AC493', ref(p)
          from person_tab p where p.lastname = 'Van Damme';
commit work;
```

Here is a listing of the intype file for our example, navdemo1.typ:

```
case=lower
type location
type budoka
type customer
```

The header file produced by the OTT, navdemo1.h, is included in the precompiler code with the #include preprocessor directive.

Read the comments throughout the precompiler code. The program adds one new budoka object (for Jackie Chan), then prints out all the customers in the *customer\_tab* table.

Here is a listing of the precompiler file, navdemo1.pc:

```
/***************************
 ^{\star} This is a simple Pro^{\star}\text{C/C++} program designed to illustrate the
 * Navigational access to objects in the object cache.
 * To build the executable:
   1. Execute the SOL script, navdemol.sql in SOL*Plus
    2. Run OTT: (The following command should appear on one line)
        ott intype=navdemo1.typ hfile=navdemo1.h outtype=navdemo1 o.typ
           code=c user=scott/tiger
   3. Precompile using Pro*C/C++:
        proc navdemol intype=navdemol o.typ
   4. Compile/Link (This step is platform specific)
 **********************
#include "navdemo1.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlca.h>
void whoops(errcode, errtext, errtextlen)
 int errcode;
 char *errtext;
 int errtextlen;
 printf("ERROR! sqlcode=%d: text = %.*s", errcode, errtextlen, errtext);
 EXEC SOL WHENEVER SOLERROR CONTINUE;
 EXEC SOL ROLLBACK WORK RELEASE;
 exit(EXIT FAILURE);
void main()
 char *uid = "scott/tiger";
      /* The following types are generated by OTT and defined in navdemo1.h */
 customer *cust p;
                                        /* Pointer to customer object */
 budoka *budo p;
                                           /* Pointer to budoka object */
                                 /* Pointer to budoka object reference */
 budoka_ref *budo_ref;
                              /* Pointer to indicator struct for budoka */
 budoka ind *budo ind;
   /* These are data declarations to be used to insert/retrieve object data */
 VARCHAR acct[21];
```

```
struct { char lname[21], fname[21]; int age; } pers;
struct { int num; char street[61], city[31], state[3], zip[11]; } addr;
EXEC SQL WHENEVER SQLERROR DO whoops(
  sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc, sqlca.sqlerrm.sqlerrml);
EXEC SQL CONNECT : uid;
EXEC SQL ALLOCATE :budo ref;
/* Create a new budoka object with an associated indicator
 * variable returning a REF to that budoka as well.
EXEC SQL OBJECT CREATE :budo p:budo ind TABLE PERSON TAB
            RETURNING REF INTO :budo ref;
/* Create a new customer object with an associated indicator */
EXEC SQL OBJECT CREATE : cust p:cust ind TABLE CUSTOMER TAB;
/* Set all budoka indicators to NOT NULL. We
* will be setting all attributes of the budoka.
budo ind-> atomic = budo ind->lastname = budo ind->firstname =
 budo ind->age = OCI IND NOTNULL;
/* We will also set all address attributes of the budoka */
budo ind->addr. atomic = budo ind->addr.num = budo ind->addr.street =
  budo ind->addr.city = budo ind->addr.state = budo ind->addr.zip =
    OCI IND NOTNULL;
/* All customer attributes will likewise be set */
cust ind-> atomic = cust ind->account number = cust ind->aperson =
  OCI_IND_NOTNULL;
/* Set the default CHAR semantics to type 5 (STRING) */
EXEC ORACLE OPTION (char map=string);
strcpy((char *)pers.lname, (char *)"Chan");
strcpy((char *)pers.fname, (char *)"Jackie");
pers.age = 38;
/* Convert native C types to OTS types */
EXEC SQL OBJECT SET lastname, firstname, age OF :budo p TO :pers;
addr.num = 1893;
strcpy((char *)addr.street, (char *)"Rumble Street");
strcpy((char *)addr.city, (char *)"Bronx");
strcpy((char *)addr.state, (char *)"NY");
strcpy((char *)addr.zip, (char *)"92510");
/* Convert native C types to OTS types */
EXEC SQL OBJECT SET :budo p->addr TO :addr;
acct.len = strlen(strcpy((char *)acct.arr, (char *)"FS926"));
^{\prime \star} Convert native C types to OTS types - Note also the REF type ^{\star \prime}
EXEC SQL OBJECT SET account number, aperson OF :cust p TO :acct, :budo ref;
/* Mark as updated both the new customer and the budoka */
EXEC SQL OBJECT UPDATE : cust p;
EXEC SQL OBJECT UPDATE :budo p;
```

```
/* Now flush the changes to the server, effectively
 * inserting the data into the respective tables.
EXEC SQL OBJECT FLUSH :budo p;
EXEC SQL OBJECT FLUSH : cust p;
/* Associative access to the REFs from CUSTOMER TAB */
EXEC SQL DECLARE ref cur CURSOR FOR
  SELECT REF(c) FROM customer tab c;
EXEC SQL OPEN ref cur;
printf("\n");
/* Allocate a REF to a customer for use in the following */
EXEC SQL ALLOCATE :cust ref;
EXEC SQL WHENEVER NOT FOUND DO break;
while (1)
 EXEC SQL FETCH ref cur INTO :cust ref;
  /* Pin the customer REF, returning a pointer to a customer object */
  EXEC SQL OBJECT DEREF : cust ref INTO : cust p:cust ind;
  /* Convert the OTS types to native C types */
  EXEC SQL OBJECT GET account number FROM :cust p INTO :acct;
  printf("Customer Account is %.*s\n", acct.len, (char *)acct.arr);
  /* Pin the budoka REF, returning a pointer to a budoka object */
  EXEC SQL OBJECT DEREF : cust p->aperson INTO :budo p:budo ind;
  /* Convert the OTS types to native C types */
  EXEC SQL OBJECT GET lastname, firstname, age FROM :budo p INTO :pers;
  printf("Last Name: %s\nFirst Name: %s\nAge: %d\n",
     pers.lname, pers.fname, pers.age);
  /* Do the same for the address attributes as well */
  EXEC SQL OBJECT GET :budo p->addr INTO :addr;
  printf("Address:\n");
  printf(" Street: %d %s\n City: %s\n State: %s\n Zip: %s\n\n",
     addr.num, addr.street, addr.city, addr.state, addr.zip);
  /* Unpin the customer object and budoka objects */
 EXEC SQL OBJECT RELEASE : cust p;
  EXEC SQL OBJECT RELEASE :budo p;
EXEC SQL CLOSE ref cur;
EXEC SQL WHENEVER NOT FOUND DO whoops (
  sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc, sqlca.sqlerrm.sqlerrml);
/* Associatively select the newly created customer object */
EXEC SQL SELECT VALUE(c) INTO :cust p FROM customer tab c
          WHERE c.account number = 'FS926';
/* Mark as deleted the new customer object */
EXEC SQL OBJECT DELETE : cust p;
/* Flush the changes, effectively deleting the customer object */
EXEC SQL OBJECT FLUSH : cust p;
```

#### When the program is executed, the result is:

```
Customer Account is AB123
Last Name: Seagal
First Name: Steven
Birthdate: 02-14-1963
Age: 34
Address:
  Street: 1825 Aikido Way
 City: Los Angeles
  State: CA
  Zip: 45300
Customer Account is DD492
Last Name: Norris
First Name: Chuck
Birthdate: 12-25-1952
Age: 45
Address:
  Street: 291 Grant Avenue
 City: Hollywood
  State: CA
  Zip: 21003
Customer Account is SM493
Last Name: Wallace
First Name: Bill
Birthdate: 02-29-1944
Age: 53
Address:
  Street: 874 Richmond Street
  City: New York
  State: NY
  Zip: 45100
Customer Account is AC493
Last Name: Van Damme
First Name: Jean Claude
Birthdate: 12-12-1965
```



```
Age: 32
Address:
 Street: 12 Shugyo Blvd
 City: Los Angeles
 State: CA
 Zip: 95100
Customer Account is FS926
Last Name: Chan
First Name: Jackie
Birthdate: 10-10-1959
Age: 38
Address:
 Street: 1893 Rumble Street
 City: Bronx
 State: NY
  Zip: 92510
```



"The OTT Command Line" for a description of the format for the intype file

# 17.12 About Using C Structures

Before Oracle8, Pro\*C/C++ allowed you to specify a C structure as a single host variable in a SQL SELECT statement. In such cases, each member of the structure is taken to correspond to a single database column in a relational table; that is, each member represents a single item in the select list returned by the query.

In Oracle8*i* and later versions, an object type in the database is a single entity and can be selected as a single item. This introduces an ambiguity with the Oracle7 notation: is the structure for a group of scalar variables, or for an object?

Pro\*C/C++ uses the following rule to resolve the ambiguity:

A host variable that is a C structure is considered to represent an object type only if its C declaration was generated using OTT, and therefore its type description appears in a typefile specified in an INTYPE option to Pro\*C/C++. All other host structures are assumed to be uses of the Oracle7 syntax, even if a datatype of the same name resides in the database.

Thus, if you use new object types that have the same names as existing structure host variable types, be aware that Pro\*C/C++ uses the object type definitions in the INTYPE file. This can lead to compilation errors. To correct this, you might rename the existing host variable types, or use OTT to choose a new name for the object type.

The preceding rule extends *transitively* to user-defined datatypes that are aliased to OTT-generated datatypes. To illustrate, let *emptype* be a structure generated by OTT in a header file dbtypes.h and you have the following statements in your Pro\*C/C++ program:

```
#include <dbtypes.h>
typedef emptype myemp;
myemp *employee;
```

The typename *myemp* for the variable *employee* is aliased to the OTT-generated typename *emptype* for some object type defined in the database. Therefore, Pro\*C/C++ considers the variable *employee* to represent an object type.

The preceding rules do not imply that a C structure having or aliased to an OTT-generated type cannot be used for fetches of non-object type data. The only implication is that Pro\*C/C++ will not automatically expand such a structure -- the user is free to employ the "longhand syntax" and use individual fields of the structure for selecting or updating single database columns.

# 17.13 About Using REFs

The REF type denotes a reference to an object, instead of the object itself. REF types may occur in relational columns and also in attributes of an object type.

## 17.13.1 Generating a C Structure for a REF

The C representation for a REF to an object type is generated by OTT during type translation. For example, a reference to a user-defined PERSON type in the database may be represented in C as the type "Person\_ref". The exact type name is determined by the OTT options in effect during type translation. The OTT-generated typefile must be specified in the INTYPE option to Pro\*C/C++ and the OTT-generated header #included in the Pro\*C/C++ program. This scheme ensures that the proper type-checking for the REF can be performed by Pro\*C/C++ during precompilation.

A REF type does not require a special indicator structure to be generated by OTT; a scalar signed 2-byte indicator is used instead.

## 17.13.2 Declaring REFs

A host variable representing a REF in Pro\*C/C++ must be declared as a pointer to the appropriate OTT-generated type.

Unlike object types, the indicator variable for a REF is declared as the signed 2-byte scalar type <code>OCIInd</code>. As always, the indicator variable is optional, but it is a good programming practice to use one for each host variable declared.

## 17.13.3 Using REFs in Embedded SQL

REFs reside in the object cache. However, indicators for REFs are scalars and cannot be allocated in the cache. They generally reside in the user stack.

Prior to using the host structure for a REF in embedded SQL, allocate space for it in the object cache by using the EXEC SQL ALLOCATE command. After use, free using the EXEC SQL FREE or EXEC SQL CACHE FREE ALL commands.

Memory for scalar indicator variables is not allocated in the object cache, and hence indicators are not permitted to appear in the ALLOCATE and FREE commands for REF types. Scalar indicators declared as <code>OCIInd</code> reside on the program stack. At runtime, the ALLOCATE statement causes space to be allocated in the object cache for the specified host variable. For the navigational interface, use EXEC SQL GET and EXEC SQL SET, not C assignments.

Pro\*C/C++ supports REF host variables in associative SQL statements and in embedded PL/SQL blocks.

#### **Related Topics**

Navigational Interface



# 17.14 About Using OCIDate, OCIString, OCINumber, and OCIRaw

These OCI types are new C representations for a date, a varying-length zero-terminated string, an Oracle number, and varying-length binary data respectively. In certain cases, these types provide more functionality than earlier C representations of these quantities. For example, the OCIDate type provides client-side routines to perform DATE arithmetic, which in earlier releases required SQL statements at the server.

## 17.14.1 Declaring OCIDate, OCIString, OCINumber, OCIRaw

The OCI\* types appear as object type attributes in OTT-generated structures, and you use them as part of object types in Pro\*C/C++ programs. Other than their use in object types, Oracle recommends that the beginner-level C and Pro\*C/C++ user avoid declaring individual host variables of these types. An experienced Pro\*C/C++ user may wish to declare C host variables of these types to take advantage of the advanced functionality these types provide. The host variables must be declared as pointers to these types, for example, OCIString \*s. The associated (optional) indicators are scalar signed 2-byte quantities, declared, for example, OCIInd s\_ind.

## 17.14.2 Use of the OCI Types in Embedded SQL

Space for host variables of these types may be allocated in the object cache using EXEC SQL ALLOCATE. Scalar indicator variables are not permitted to appear in the ALLOCATE and FREE commands for these types. You allocate such indicators statically on the stack, or dynamically on the heap. De-allocation of space can be done using the statement EXEC SQL FREE, EXEC SQL CACHE FREE ALL, or automatically at the end of the session.

#### **Related Topics**

Navigational Interface

## 17.14.3 Manipulating the OCI Types

Except for OCIDate, which is a structure type with individual fields for various date components: year, month, day, hour and so on., the other OCI types are encapsulated, and are meant to be opaque to an external user. In contrast to the way existing C types like VARCHAR are currently handled in Pro\*C/C++, you include the OCI header file oci.h and employ its functions to perform DATE arithmetic, and to convert these types to and from native C types such as *int*, *char*, and so on.

# 17.15 Summarizing the New Database Types in Pro\*C/C++

Table 17-2 lists the new database types for Object support:



Table 17-2 Using New Database Types in Pro\*C/C++

Operations Database Type	DECLARE	ALLOCATE	FREE	MANIPULATE
Object type	Host: Pointer to OTT- generated C struct Indicator: Pointer to OTT- generated indicator struct	Associative interface: EXEC SQL ALLOCATE Navigational interface: EXEC SQL OBJECT CREATE EXEC SQL OBJECT DEREF allocates memory for host var and indicator in object cache	Freed by EXEC SQL FREE, or EXEC SQL CACHE FREE ALL, or automatically at end of session.	Dereference the C pointer to get each attribute. Manipulation method depends on type of attribute (see later).
COLLECTION Object type (NESTED TABLE AND VARYING ARRAY)	Host: Pointer to OTT- generated C struct Indicator: OCIInd	EXEC SQL ALLOCATE allocates memory for host var in object cache.	Freed by EXEC SQL FREE, or EXEC SQL CACHE FREE ALL, or automatically at end of session.	Use OCIColl* functions (defined in oci.h) to get/set elements. See also Collections.
REF	Host: Pointer to OTT- generated C struct Indicator: OCIInd	EXEC SQL ALLOCATE allocates memory for host var in object cache.	Freed by EXEC SQL FREE, or EXEC SQL CACHE FREE ALL, or automatically at end of session.	Use EXEC SQL OBJECT DEREF Use EXEC SQL OBJECT SET/GET for navigational interface.
LOB	Host: OCIBlobLocat or *, OCIClobLoca tor *, or OCIBfileLocat or *. Indicator: OCIInd	EXEC SQL ALLOCATE allocates memory for the host var in user heap using malloc().	Freed by EXEC SQL FREE, or automatically when all Pro*C/C++ connections are closed. EXEC SQL CACHE FREE ALL frees only LOB attributes of objects.	Or use embedded PL/SQL stored procedures in the dbms_lob package, or Use OCILob* functions defined in oci.h. See also "LOBs".
Note: Host arrays of these types may be declared and used in bulk fetch/insert SQL operations in Pro*C/C++.	-	-	-	-

Table 17-3 shows how to use the new C datatypes in Pro\*C/C++:

Table 17-3 Using New C Datatypes in Pro\*C/C++

Operations C Type	DECLARE	ALLOCATE	FREE	MANIPULATE
OCIDate	Host: OCIDate * Indicator: OCIInd	EXEC SQL ALLOCATE allocates memory for host var in object cache	Freed by EXEC SQL FREE, or EXEC SQL CACHE FREE ALL, or automatically at end of session.	(1) Use OCIDate* functions defined in oci.h. (2) Use EXEC SQL OBJECT GET/SET, or
				(3) Use OCINumber* functions defined in oci.h.
OCINumber	Host: OCINumber *	EXEC SQL ALLOCATE	Freed by EXEC SQL FREE, or EXEC SQL CACHE FREE ALL, or automatically at end of session.	(1) Use EXEC SQL OBJECT GET/SET,
	Indicator: OCIInd	allocates memory for host var in object cache		or (2) Use OCINumber* functions defined in oci.h.
OCIRaw	Host: OCIRaw * Indicator: OCIInd	EXEC SQL ALLOCATE allocates memory for host var in object cache	Freed by EXEC SQL FREE, or EXEC SQL CACHE FREE ALL, or automatically at end of session.	Use OCIRaw* functions defined in oci.h.
OCIString	Host: OCIString * Indicator: OCIInd	EXEC SQL ALLOCATE allocates memory for host var in object cache	EXEC SQL FREE, or EXEC SQL CACHE FREE ALL, or automatically at end of session.	(1) Use EXEC SQL OBJECT GET/SET, or (2) use OCIString* functions defined in
Note:		object cache		oci.h.
Host arrays of these types may not be used in bulk fetch/insert SQL operations in Pro*C/C++.	-			-

The new datatypes for Oracle8 were Ref, BLOB, NCLOB, CLOB, and BFILE. These types may be used in objects or in relational columns. In either case, they are mapped to host variables according to the C bindings.



"Summarizing the New Database Types in Pro\*C/C++" for a description of the C bindings

# 17.16 Restrictions on Using Oracle Datatypes in Dynamic SQL

Pro\*C/C++ currently supports these different types of dynamic SQL methods: methods 1, 2, 3, and 4 (ANSI and Oracle).

The dynamic methods 1, 2, and 3 will handle all Pro\*C/C++ extensions mentioned earlier, including the new object types, REF, Nested Table, Varying Array, NCHAR, NCHAR Varying and LOB types.

The older Dynamic SQL method 4 is generally restricted to the Oracle types supported by Pro\*C/C++ prior to release 8.0. It does allow host variables of the NCHAR, NCHAR Varying and LOB datatypes. Dynamic method 4 is not available for object types, Nested Table, Varying Array, and REF types.

Instead, use ANSI Dynamic SQL Method 4 for all new applications, because it supports all datatypes introduced in Oracle8*i*.

#### **Related Topics**

- Oracle Dynamic SQL
- ANSI Dynamic SQL
- Oracle Dynamic SQL: Method 4



# Collections

This chapter describes other kinds of object types, known as *collections*, and the ways of using them in Pro\*C/C++. We present methods to access collections and their elements. This chapter contains the following topics:

- Collections
- Descriptors for Collections
- OBJECT GET and SET
- Collection Statements
- Collection Example Code

## 18.1 Collections

There are two kinds of collection object types: nested tables and varrays.

Collections may occur both in relational columns and also as attributes within an object type. All collections must be *named* object types in the database. In the case of varrays, you must first create a named type in the database, specifying the desired array element type and maximum array dimension.

With Oracle, you are no longer limited to a single level of nesting. Oracle supports multilevel collections of objects enabling multiple levels of nested tables and varrays.

## 18.1.1 Nested Tables

A nested table is a collection of rows, called elements, within a column. For each row of the database table there are many such elements. A simple example is the list of tasks that each employee is working on. Thus many-to-one relationships can be stored in one table, without needing to join employee and task tables.

Nested tables differ from C and C++ arrays in these ways:

- Arrays have a fixed upper bound; nested tables are unbounded (have no maximum index).
- Arrays have consecutive subscripts (are *dense*); nested tables can be either dense or sparse. When a nested table is retrieved into an array in a program, the gaps are skipped, resulting in a filled array that has no gaps.

Use the CREATE TYPE statement to define a table type that can be nested within other object types in one or more columns of a relational table.

For example:, to store several projects in each department of an organization:

```
CREATE TABLE depts (
dno CHAR(5),
dname CHAR(20),
budgets_limit NUMBER(15,2),
projects project_table)
NESTED TABLE projects STORE AS depts projects;
```

## **18.1.2 Varrays**

Unlike nested tables, you must specify the maximum number of elements when you create a VARRAY type. Varrays are only dense, unlike nested tables, which can be dense or sparse. The elements of a varray and a nested table are both numbered from 0.

You create a varray with CREATE TYPE statements such as:

```
CREATE TYPE employee AS OBJECT (
   name    VARCHAR2(10),
   salary NUMBER(8,2)
);
CREATE TYPE employees AS VARRAY(15) OF employee;
CREATE TYPE department AS OBJECT (
   name VARCHAR2(15),
   team employees
);
```

Use VARRAY as a datatype of a column of a relational table, or as the attribute of an object type. This saves storage space compared to a relational table that has up to 15 records for each team, each containing the team's name.

## 18.1.3 C and Collections

In a C or C++ program a nested table is read in starting from index value 0 of the collection. When you write the nested table into an array, the elements of the nested table are stored starting at array index 0. When a nested table which is sparse (has gaps in its index) is stored into an array, the gaps are skipped. When reading the array back into the nested table, the gaps are re-created.

In a C or C++ program varrays are written into an array, starting at index 0. When read back into the varray, the elements are restored starting at index 0 in the same order. Thus, arrays are easily used to access collections.

# 18.2 Descriptors for Collections

The C type for a nested table is a pointer to OCITable. For varrays, it is a pointer to OCIArray. (Both are subtypes of a pointer to OCIColl). Use the OTT (Object Type Translator) utility to generate typedefs in header files that you then include in your application code.

The host structure for a collection is a *descriptor* through which the elements in the collection may be accessed. These descriptors do not hold the actual elements of the collection, but instead contain the pointers to them. Memory for both the descriptor and its associated elements come from the object cache.

Following the usual procedure for object types, the OTT-generated typefile must be specified in the INTYPE precompiler option to Pro\*C/C++ and the OTT-generated headers included in the Pro\*C/C++ program using the #include preprocessor directive. This ensures that the proper type-checking for the collection object type can be performed during precompilation.

Unlike other object types, however, a collection object type does not require a special indicator structure to be generated by OTT; a scalar indicator is used instead. This is because an atomic NULL indicator is sufficient to denote whether a collection as a whole is NULL. The NULL status of each individual element in a collection may (optionally) be represented in separate indicators associated to each element.

### **Related Topics**

The Object Type Translator

### 18.2.1 Declarations for Host and Indicator Variables

As for the other object types, a host variable representing a collection object type must be declared as a pointer to the appropriate OTT-generated typedef.

Unlike other object types, however, the indicator variable for a collection object type as a whole is declared as a scalar signed 2-byte type, <code>OCIInd</code>. The indicator variable is optional, but it is a good programming practice to use one for each host variable declared in Pro\*C/C++.

# 18.2.2 About Manipulating Collections

There are two ways to manipulate a collection: the collection is treated as an autonomous entity without access to its elements, or its elements are accessed, appended to, truncated, and so on.

### 18.2.2.1 Autonomous Collection Access

Using a C collection descriptor (OCITable or OCIArray) allows only assignment of the collection as a whole. The OBJECT GET embedded SQL statement binds the collection to a C host variable descriptor. The opposite occurs in an OBJECT SET statement which binds the C host descriptor to the collection.

It is possible to bind more than one collection to a compatible C descriptor in the same statement, or include bindings of other scalars in the same statement where binding of a collection to a C descriptor occurs.

#### **Related Topics**

Converting Object Attributes and C Types

### 18.2.2.2 Collection Element Access

The C collection descriptor is used to access elements of a collection. The descriptor contains the internal attributes of the collection such as its start and end points, and other information.

A *slice* of elements are bound to a host array that has a compatible data type. A slice of a collection is defined as the contents between a starting index and an ending index. The slice maps to an array, which can have a dimension larger than the number of slice elements.

Binding a scalar is the same as having a host array of dimension one, or having an optional FOR clause evaluated to one.

## 18.2.3 Rules for Access

Access rules are different for autonomous and element access.



### 18.2.3.1 Autonomous Access

- The FOR clause is not allowed since the collection is treated as a whole.
- Because nested tables and varrays are defined differently, assignments between them are not possible.
- Multiple assignments of several collections to C descriptors are allowed in the same statement. You can assign a collection to a C descriptor and also bind other scalar datatypes in the same statement.

### 18.2.3.2 Element Access

- FOR clauses are permitted. If omitted, the smallest array dimension gives the number of iterations to perform.
- Only one collection can be accessed at a time.



The FOR clause variable specifies the number of array elements to be processed. Make sure the number does not exceed the smallest array dimension. Internally, the value is treated as an unsigned quantity. An attempt to pass a negative value through the use of a signed host variable will result in unpredictable behavior.

## 18.2.4 Indicator Variables

Each access method has its own way of using indicator variables.

# 18.2.4.1 Autonomous Bindings

The single indicator variable holds the NULL status of the collection as a single entity. This says nothing about the NULL status in the elements.

# 18.2.4.2 Element Bindings

The indicator variables show whether an element is NULL or not. If a slice of collection data is bound to a host array which has its own indicator array, that indicator array will contain the NULL status of each element in the slice.

When the collection element type is a user-defined object type, the indicator variable associated with the host variable contains the NULL status of the object and of its attributes.

# 18.3 OBJECT GET and SET

The navigational statements OBJECT SET and OBJECT GET permit you to retrieve and update collection attributes as well as object types defined by you.

For elements that are object types, the OBJECT GET statement retrieves all attributes of the object into the host variables when either the

```
'*' | {attr [, attr]} FROM
```

#### clause is omitted, or 'OBJECT GET \* FROM ' is used:

The OBJECT SET statement causes all attributes of the object to be updated using the host variables when either the

```
'*' | {attr, [, attr]} OF
```

clause is omitted or 'OBJECT SET \* OF' is used:

This table shows how object and collection types are mapped by these two statements:

Table 18-1 Object and Collection Attributes

Attribute Type	Representation	Host Data Type
Object	OTT-Generated Structure	Pointer to OTT Structure
Collection	OCIArray, OCITable (OCIColl)	OCIArray *, OCITable * (OCIColl *)

The object or collection must be type compatible with the attribute to which it is bound. Collection attributes are type compatible if and only if they are both either varray or nested table, and their element types are compatible.

This next table shows how type compatibility for the elements of two collection types is obtained.

Table 18-2 Collection and Host Array Allowable Type Conversions

Collection Element Type	Representation	Host Data Type
CHAR, VARCHAR, VARCHAR2	OCIString	string, VARCHAR, CHARZ, OCIString
REF	OCIRef	OCIRef
INTEGER, SMALLINT, INT	OCINumber	int, short, OCINumber
NUMBER, NUMERIC, REAL, FLOAT, DOUBLE PRECISION	OCINumber	int, float, double, OCINumber
DATE	OCIDate	string, VARCHAR, CHARZ, OCIDate

The object to which the REF refers must be type compatible with the REF to which it is bound.

In both tables OBJECT GET converts database types from the type on the left, using the format specified in the Representation column to an internal datatype using the format in the Host Data Type column. OBJECT SET converts in the opposite direction.

#### No Explicit Type Checking

The precompiler does not support explicit type checking on the bindings between collection element datatypes and host variable datatypes. Type-checking is done at runtime.

### **Related Topics**

Converting Object Attributes and C Types

# 18.4 Collection Statements

This section includes descriptions of the collection statements.

### 18.4.1 COLLECTION GET

#### **Purpose**

The COLLECTION GET statement is analogous to the OBJECT GET statement, but is designed for collections. It retrieves the elements of a collection, sets the current slice, and converts elements to C types where appropriate.

#### **Syntax**

#### **Variables**

num (IN)

The number of elements requested. If this clause is omitted, the array size of the host variable (scalar has 1) determines the number of elements retrieved from the collection.

collect (IN)

A host variable C collection descriptor.

collect\_ind (IN)

An optional indicator variable returning the NULL status of the collection.

hv (OUT)

The host variable that receives the collection element values.

hv ind (OUT)

An optional indicator variable returning the NULL status of hv if it is a scalar, or an array holding the status of each element in the slice.

### **Usage Notes**

The number of elements actually returned in the last COLLECTION GET is set in sqlca.sqlerrd[2] (not the cumulative total of all GETs).

The number returned can be less than the number requested when one or both of the slice endpoints exceed the collection bounds. This can occur when:

- The collection descriptor has not been initialized in a syntactically correct ALLOCATE statement, is NULL, or invalid for any other reason.
- The collection is NULL. Its associated indicator is -1.
- The collection is empty (has no elements).
- More elements were requested than are remaining in the collection.



 A COLLECTION TRIM statement has been executed, resulting in the endpoint index being below the starting index of the current slice.

An improperly initialized C collection descriptor results in an error. All other situations in the list will raise an ORA-01403: no data found error condition. In this case, the total number of elements successfully retrieved before the error occurs is still stored in sqlca.sqlerrd[2].

The initial GET or the first GET after a RESET affects the slice as follows:

The ending index of the slice will be the index where the final element was found; it
depends on how many elements are requested. If there were not enough elements in the
remaining portion of the collection to satisfy the request, then the ending index will be the
last index in the collection.

Subsequent GETs affect the slice indexes as follows:

- The index of the start-point is the index where the first element was found after the endpoint of the previous slice. If no more elements are left after the end-point of the previous slice, the starting index is that of the last element in the collection.
- The ending index of the next slice will be the index where the final element was found; it
  depends on how many elements are requested. If there were not enough elements in the
  remaining portion of the collection to satisfy the request given the position of the previous
  slice, then the ending index will be the last index in the collection.

### **Related Topics**

sqlerrd

### 18.4.2 COLLECTION SET

### **Purpose**

The COLLECTION SET statement is analogous to the OBJECT SET statement; it is used for updating element values of collections. Elements in the current slice are converted from the native C types to Oracle datatypes.

### **Syntax**

```
EXEC SQL [AT [:]database] [FOR :num]
COLLECTION SET :collect [[INDICATOR] :collect_ind]
TO :hv [[INDICATOR] :hv ind];
```

#### **Variables**

num (IN)

This optional scalar value is the maximum number of elements to be updated in the slice. If this clause is omitted, the array size of the host variable (scalar has 1) determines the number of elements updated from the collection.

collect (OUT)

A host variable C collection descriptor.

collect\_ind (OUT)

An optional indicator variable which determines the NULL status of the collection.

hv (IN)

A host variable that contains the values to be updated in the collection.

hv\_ind (IN)

An associated indicator variable representing the NULL status of the host variable.

### **Usage Notes**

The following restrictions apply:

- A COLLECTION GET must be executed before a COLLECTION SET.
- The starting and ending indexes of the slice always remain unchanged. This is true even if less elements are placed in the collection than can be stored in the current slice. A SET statement never changes the endpoints of the slice. It only changes elements in the current slice.
- The COLLECTION SET is intended only to update those elements in the current slice. You
  cannot append new elements to a collection using the COLLECTION SET statement.
- If an attempt is made to SET more elements than the current slice contains, only those elements that fit in the existing slice are updated. Remaining elements beyond the end of the slice are unaffected and any extra values supplied by the host variable are unused.

The dimension of the host variable or the value <code>num</code> specified in an optional FOR clause gives the maximum number of elements requested to be updated in the collection.

The variable sqlca.sqlerrd[2] returns the number of elements successfully updated by the previous SET statement (not a cumulative total), which can be less than the number requested to be set (as for the GET), for these cases:

- The C collection descriptor has not been correctly initialized in a syntactically correct ALLOCATE statement, or is NULL, or otherwise invalid.
- The collection is empty.
- There were fewer elements in the remaining portion of the collection than were requested to be set given the position of the current slice in the collection.
- The end of the current slice was breached. This can happen only when an attempt is made to set more elements than are in the existing slice.
- A TRIM has been performed on the collection, bringing the maximum endpoint index value
  of the collection below the starting index of the current slice.

Issuing a COLLECTION SET immediately after a COLLECTION GET or SET will only update the values of the elements in the *existing* slice. A COLLECTION GET that immediately follows a COLLECTION SET will advance the slice as already described.

## 18.4.3 COLLECTION RESET

#### **Purpose**

Reset the collection slice endpoints back to the beginning of the collection.

#### **Syntax**

```
EXEC SQL [AT [:]database]
  COLLECTION RESET :collect
       [ [INDICATOR] :collect ind] ;
```

#### **Variables**

collect (IN/OUT)



The collection whose endpoints are to be reset.

collect\_ind

Optional indicator variable determining NULL status of the collection.

### **Usage Notes**

An error occurs if the given collection is NULL, or otherwise invalid.

COLLETION RESET does not affect the size or contents of the collection.

### 18.4.4 COLLECTION APPEND

#### **Purpose**

This statement appends a set of elements (one or more) to the end of a collection, increasing the size of the collection.

### **Syntax**

```
EXEC SQL [AT [:]database] [FOR :num]
COLLECTION APPEND :src [[INDICATOR] :src_ind]
TO :collect [[INDICATOR] :collect ind];
```

#### **Variables**

num (IN)

A scalar that contains the number of elements to be appended. If absent, the array size of src is the number of elements to be appended.

src (IN)

Scalar or array of elements to be appended to the collection.

```
src ind (IN)
```

An optional indicator variable (scalar or array) determining the NULL status of the elements appended.

```
collect (IN OUT)
```

The collection to which elements are appended.

```
collect ind (IN)
```

An optional indicator variable determining the NULL status of the collection.

#### **Usage Notes**

Elements are appended one at a time (the collection is increased in size by one, the data is copied to that element, and so on).

The variable sqlca.sqlerrd[2] returns the number of elements successfully appended by the latest APPEND (not a cumulative total). An error results if there is an attempt to add elements beyond the upper bound of the collection, or to append to a NULL collection. Only the elements that fit will be appended.



### 18.4.5 COLLECTION TRIM

### **Purpose**

This statement removes elements from the end of a collection.

#### **Syntax**

```
EXEC SQL [AT [:]database]
  COLLECTION TRIM :num
   FROM :collect [[INDICATOR] :collect ind] ;
```

#### **Variables**

num (IN)

A host scalar variable which is how many elements are to be removed. The maximum permitted value is two Gigabytes.

```
collect (IN OUT)
```

The collection to be trimmed.

```
collect ind (IN)
```

An optional indicator variable determining the NULL status of the collection.

### **Usage Notes**

Restrictions applied:

- The FOR clause is not allowed.
- The maximum value of num is 2 Gigabytes (the largest number in a 4-byte signed binary variable.
- No indicator is allowed with num.

If num is greater than the size of the collection, an error is returned. A warning is returned if a TRIM removes any elements from the current slice.

# 18.4.6 COLLECTION DESCRIBE

#### **Purpose**

This statement returns information about a collection.

### **Syntax**



#### **Variables**

collect (IN)

The host variable C collection descriptor.

collect\_ind (IN)

An optional indicator variable containing the NULL status of the collection.

hv1 .. hvN (OUT)

Output host variables where the information is to be stored.

hv\_ind1 .. hv\_indN (OUT)

Indicator variables for the output host variables.

### **Usage Notes**

These restrictions are in effect:

- The collection cannot be NULL.
- Host variable types should be compatible with the types of the returned attributes.
- Indicator variables for the attributes are only required for TYPE\_NAME and TYPE SCHEMA attribute values where text truncation can occur.
- A FOR clause is not allowed.
- The variable sqlca.sqlerrd[2] returns the number of successful attributes retrieved with no
  errors. If the DESCRIBE statement incurs an error, then sqlca.sqlqerrd[2] contains the
  number of attributes returned before that error, and is one less than the attribute where the
  error occurred.

The following table gives attributes, descriptions, and C types for the attributes retrieved:

Table 18-3 Attributes of a COLLECTION DESCRIBE

Attribute	Description	С Туре	Notes
DATA_SIZE	The maximum size of the type attribute. The returned length is in bytes for strings. It is 22 for NUMBERs.	unsigned short	Not valid for object or object REF elements.
TYPECODE	OCI type code.	OCITypeCode	-
DATA_TYPE	The internal numeric type code of the collection items.	unsigned short	-
NUM_ELEMENTS	The maximum number of elements in the varray.	unsigned int	Only valid for type VARRAY.
PRECISION	The precision of numeric type attributes. When the returned value is 0 the item being described is not initialized and is NULL in the data dictionary.	unsigned char	Only valid for elements of type NUMBER.
SCALE	The scale of numeric type attributes. When the returned value is -127 the item being described is not initialized and is NULL in the data dictionary.	signed char	Only valid for elements of type NUMBER.



Table 18-3 (Cont.) Attributes of a COLLECTION DESCRIBE

Attribute	Description	С Туре	Notes
TYPE_NAME	A string containing the name of the type. For an object type, its name is returned. For a REF the name of the data type pointed to by the REF is returned. External data types allowed are CHARZ, STRING, and VARCHAR.	char *	Only valid for object and object REF elements.
TYPE_SCHEMA	The schema name where the type was created. External data types allowed are CHARZ, STRING, and VARCHAR.	char *	Only valid for object and object REF elements.
SIZE	The number of elements actually stored in the collection. For nested tables, SIZE includes the empty elements. A TRIM statement decrements the SIZE of the collection by the number of elements trimmed.	signed int	-
TABLE_SIZE	The number of elements in the nested table. It does not include the gaps.	signed int	Only valid for nested tables.

### 18.4.6.1 Notes on the Table

Pro\*C/C++ only supports the external datatypes CHARZ, STRING, and VARCHAR for the attributes TYPE NAME and TYPE SCHEMA.

All the DESCRIBE attributes, except for SIZE and TABLE\_SIZE, depend on the element type of the collection and are independent of any one particular instance of the collection. The SIZE and TABLE\_SIZE attributes, on the other hand, are attributes whose values strictly depend on a specific instance of the collection. The SIZE or TABLE\_SIZE values will change from one collection instance to another in cases where an allocated collection descriptor is being reused to refer to different instances of the same collection. NUM\_ELEMENTS is an attribute of the collection type (a VARRAY in this case), not the collection element type, and is independent of any one particular instance of the collection.

# 18.4.7 Rules for the Use of Collections

- A host variable collection descriptor must always be explicitly allocated.
- The metadata (internal Oracle data from the database about a collection and its element type) is collected during an ALLOCATE. That metadata becomes invalid when the connection in which the ALLOCATE was made is closed or if the type changes after the ALLOCATE.
- Use the ALLOCATE and FREE statements to begin and end use of each C collection descriptor.

# 18.5 Collection Example Code

Here are examples of SQL and Pro\*C/C++ code that illustrates the use of the COLLECTION SQL statements:

# 18.5.1 Type and Table Creation

Assuming a connection as scott/tiger, we create the following types using SQL:

```
CREATE TYPE employee AS OBJECT (
   name    VARCHAR2(10),
   salary NUMBER(8,2)
);
CREATE TYPE employees AS VARRAY(15) OF employee;
CREATE TYPE department AS OBJECT (
   name VARCHAR2(15),
   team employees
);
```

Now for the header file generated by the Object Type Translator. The following intype file (called in.typ) will be used as input to OTT:

```
case=lower
type employee
type employees
type department
```

The following command will generate the header file:

```
ott intype=in.typ outtype=out.typ hfile=example.h user=scott/tiger code=c
```

This header file, example.h, is produced by OTT:

```
#ifndef EXAMPLE ORACLE
# define EXAMPLE ORACLE
#ifndef OCI ORACLE
# include <oci.h>
#endif
typedef OCIRef employee ref;
typedef OCIArray employees;
typedef OCIRef department ref;
struct employee
  OCIString * name ;
  OCINumber salary ;
typedef struct employee employee;
struct employee_ind
  OCIInd _atomic ;
  OCIInd name ;
  OCIInd salary ;
typedef struct employee ind employee ind;
struct department ind
{
  OCIInd atomic;
  OCIInd name ;
  OCIInd team ;
} ;
typedef struct department_ind department_ind ;
#endif
```





The <code>oci.h</code> file includes orl.h which has a typedef that defines OCIArray. That typedef looks like the following 'typedef OCIColl OCIArray;' where OCIColl is an opaque structure representing a generic collection.

Now create a simple table that has one column as follows:

```
CREATE TABLE division ( subdivision department ) ;
```

Now we will insert a few rows into that table:

We can now use these type definitions and table information in the examples to follow.

# 18.5.2 GET and SET Example

Suppose we want to retrieve the values from the collection attribute of an example object, modify them in some simple way and place them back into the collection.

First, we need to include example.h and declare a variable for the object type:

```
#include <example.h>
department *dept p ;
```

Now we will select the 'Development' department from the division table:

```
EXEC SQL ALLOCATE :dept_p ;
EXEC SQL SELECT subdivision INTO :dept_p
   FROM division WHERE name = 'Development' ;
```

We also need a variable for the team varray of employee object types and one to represent a single employee object. We will be giving all team members of the 'Development' department a raise so we need a variable for that also:

```
employees *emp_array ;
employee *emp_p ;
double salary ;
```

Now we must ALLOCATE our varray C Collection and employee object descriptors. We will retrieve the actual collection from the object using the navigational interface:

```
EXEC SQL ALLOCATE :emp_array ;
EXEC SQL ALLOCATE :emp_p ;
EXEC SQL OBJECT GET team FROM :dept_p INTO :emp_array ;
```

We will use a loop to iterate through the varray elements, controlling loop termination by using the WHENEVER directive:

```
EXEC SQL WHENEVER NOT FOUND DO break ;
while (TRUE)
{
```

First, retrieve the element from the collection so that we can alter it. The actual element type is an employee object:

```
EXEC SQL COLLECTION GET :emp array INTO :emp p ;
```

Now that we have the actual object element, we can then proceed to change an attribute's value using the existing navigational interface. In this case, we give a 10% salary increase to everyone:

```
EXEC SQL OBJECT GET salary FROM :emp_p INTO :salary ;
salary += (salary * .10) ;
EXEC SQL OBJECT SET salary OF :emp p TO :salary ;
```

Once we are done making our changes, we can update the value of the attribute of the object element we are currently at in the collection:

```
EXEC SQL COLLECTION SET :emp_array TO :emp_p ;
}
```

Once we are done iterating through all of the collection elements, we must then update the column of the table that stores the object that contains the collection we just finished modifying:

```
EXEC SQL UPDATE division SET subdivision = :dept_p ;
```

We can then FREE all of our resources and COMMIT our work thus terminating this sequence of operations:

```
EXEC SQL FREE :emp_array ;
EXEC SQL FREE :emp_p ;
EXEC SQL FREE :dept_p ;
EXEC SQL COMMIT WORK ;
```

Although this is a fairly small and simple example, it is quite comprehensive. It clearly demonstrates how a collection attribute can be retrieved from an object into a C Collection Descriptor using the semantic extensions to the navigational OBJECT GET statement. Using that C Descriptor we then saw how to use the new COLLECTION GET and SET statements to retrieve and update the actual elements of that collection. We used the navigational interface to modify the attribute values of the collection object element type.

# 18.5.3 DESCRIBE Example

This example illustrates a simple use of the DESCRIBE SQL statement. We want to find out some basic information about a given collection.

First we need our example header file, an object pointer and a SQL Collection Descriptor:

```
#include <example.h>
department *dept p ;
```

Now we ALLOCATE the object pointer and retrieve our object from the table as before:



```
EXEC SQL ALLOCATE :dept_p ;
EXEC SQL SELECT subdivision INTO :dept_p
   FROM division WHERE name = 'Research' ;
```

Declare Pro\*C/C++ variables to store the collection attribute information we want:

```
int size ;
char type_name[9] ;
employees *emp array ;
```

Allocate the collection descriptor, then use the navigational interface to get the collection attribute from the object:

```
EXEC SQL ALLOCATE :emp_array ;
EXEC SQL OBJECT GET team FROM :dept p INTO :emp array ;
```

Finally, we can use the new COLLECTION DESCRIBE statement to extract the desired collection attribute information:

```
EXEC SQL COLLECTION DESCRIBE :emp_array
   GET SIZE, TYPE_NAME INTO :size, :type_name ;
```



You are permitted to use host variable names that are the same as the desired collection attribute name, as in this example.

Because the type employees is a VARRAY of object employee, we can extract the type name.

After successful completion of the DESCRIBE, the value of size should be 2 (there are 2 elements, Albert and Alison, in this collection instance, Research). The type\_name variable should read "EMPLOYEE\0" (it is a CHARZ by default).

Once we are finished with the SQL Descriptor and the object pointer we can FREE their resources:

```
EXEC SQL FREE :emp_array ;
EXEC SQL FREE :dept p ;
```

We have just illustrated that the DESCRIBE mechanism is used to extract useful information from a C Collection Descriptor about the underlying collection to which the descriptor refers.

# 18.5.4 RESET Example

Now suppose that instead of giving just the employees in Development a raise, as in the GET and SET example, we give raises to everybody in the entire division.

We start off as before, including our example header file generated by the Object Type Translator. This time, however, we will be using a cursor to iterate through all departments in the division, one at a time:

```
#include <example.h>
EXEC SQL DECLARE c CURSOR FOR SELECT subdivision FROM division ;
```

We will need some local variables to manipulate our data:

```
department *dept_p ;
employees *emp_array ;
```



```
employee *emp_p ;
double salary ;
int size ;
```

Before we can use the object and collection variables, we must first initialize them by using this ALLOCATE statement:

```
EXEC SQL ALLOCATE :emp_array ;
EXEC SQL ALLOCATE :emp p ;
```

Now we are ready to use our cursor to iterate through all of the departments in the division:

```
EXEC SQL OPEN c ;
EXEC SQL WHENEVER NOT FOUND DO break ;
while (TRUE)
{
          EXEC SQL FETCH c INTO :dept_p ;
```

At this point, we have a department object. We need to use the Navigational Interface to extract the 'team' VARRAY attribute from the department:

```
EXEC SQL OBJECT GET team FROM :dept p INTO :emp array ;
```

Before we can start referring to the collection, we need to guarantee that the slice endpoints are set to the beginning of the *current* collection instance (not the end of a previous instance) by use of the RESET statement:

```
EXEC SQL COLLECTION RESET :emp array ;
```

Now we will iterate through all elements of the varray and update the salaries as we did before. The existing WHENEVER directive remains in effect for this loop as well:

```
while (TRUE)
{
   EXEC SQL COLLECTION GET :emp_array INTO :emp_p;
   EXEC SQL OBJECT GET salary FROM :emp_p INTO :salary;
   salary += (salary * .05);
   EXEC SQL OBJECT SET salary OF :emp p TO :salary;
```

When we are finished, we'll update the collection attribute:

```
EXEC SQL COLLECTION SET :emp_array TO :emp_p ;
}
```

As before, we need to update the column of the table that stores the object that contains the collection that we just finished modifying:

```
EXEC SQL UPDATE division SET subdivision = :dept_p ;
}
```

Loop termination signifies the end of processing. We can now FREE all of our resources and COMMIT our work:

```
EXEC SQL CLOSE c;

EXEC SQL FREE :emp_p;

EXEC SQL FREE :emp_array;

EXEC SQL FREE :dept_p;

EXEC SQL COMMIT WORK;
```

This example demonstrates how it is possible to reuse an ALLOCATEd Collection Descriptor for different instances of the same collection type. The COLLECTION RESET statement ensures the slice endpoints are set back to the beginning of the *current* collection instance.

They do not remain in their existing positions after being moved during the referencing of a previous collection instance.

By using the COLLECTION RESET statement in this fashion, application developers need not explicitly FREE and reALLOCATE a Collection Descriptor with each new instance of the same collection type.

# 18.5.5 Example Program:coldemo1.pc

The following program, coldemo1.pc, is in the demo directory.

This example demonstrates three ways for the Pro\*C client to navigate through collectiontyped database columns. Although the examples presented use nested tables, they also apply to varrays.

Here is the SQL\*Plus file, coldemo1.sql, that sets up the table using the inserts and data contained in calidata.sql:

```
REM ** This is a SOL*Plus script to demonstrate collection manipulation
REM ** in Pro*C/C++.
REM ** Run this script before executing OTT for the coldemol.pc program
connect scott/tiger;
set serveroutput on;
REM Make sure database has no old version of the table and types
DROP TABLE county tbl;
DROP TYPE citytbl t;
DROP TYPE city t;
REM ABSTRACTION:
REM The counties table contains census information about each of the
REM counties in a particular U.S. state (California is used here).
REM Each county has a name, and a collection of cities.
REM Each city has a name and a population.
REM IMPLEMENTATION:
REM Our implementation follows this abstraction
REM Each city is implemented as a "city" OBJECT, and the
REM collection of cities in the county is implemented using
REM a NESTED TABLE of "city" OBJECTS.
CREATE TYPE city t AS OBJECT (name CHAR(30), population NUMBER);
CREATE TYPE citytbl t AS TABLE OF city t;
CREATE TABLE county tbl (name CHAR(30), cities citytbl t)
 NESTED TABLE cities STORE AS citytbl t tbl;
REM Load the counties table with data. This example uses estimates of
REM California demographics from Januarry 1, 1996.
@calidata.sql;
```

```
REM Commit to save
COMMIT;
```

See the comments at the beginning of the following program for explanations of how to set up the table, and then the functionality demonstrated by this program.

```
/* ********** */
             Demo program for Collections in Pro*C */
          /* ********* */
/********************************
  In SQL*Plus, run the SQL script coldemo1.sql to create:
    - 2 types: city t (OBJECT) and citytbl t (NESTED TABLE)
    - 1 relational table county tbl which contains a citytbl t nested table
  Next, run the Object Type Translator (OTT) to generate typedefs of C structs
  corresponding to the city t and citytbl t types in the databases:
    ott int=coldemo1.typ outt=out.typ hfile=coldemo1.h code=c user=scott/tiger
  Then, run the Pro*C/C++ Precompiler as follows:
    proc coldemol intype=out.typ
  Finally, link the generated code using the Pro*C Makefile:
    (Compiling and Linking applications is a platform dependent step).
 *******************
    We consider the example of a database used to store census
    information for the state of California. The database has a table
    representing the counties of California. Each county has a name
    and a collection of cities. Each city has a name and a population.
  Application Overview:
    This example demonstrates three ways for the Pro*C client to
    navigate through collection-typed database columns. Although the
    examples presented use nested tables, they also apply to varrays.
    Collections-specific functionality is demonstrated in three
    different functions, as described in the following section.
    PrintCounties shows examples of
    * Declaring collection-typed host variables and arrays
    * Allocating and freeing collection-typed host variables
    * Using SQL to load a collection-typed host variable
    * Using indicators for collection-typed host variables
    * Using OCI to examine a collection-typed host variables
    PrintCounty shows examples of
    * Binding a ref cursor host variable to a nested table column
    * Allocating and freeing a ref cursor
    * Using the SQL "CURSOR" clause
    CountyPopulation shows examples of
    * Binding a "DECLARED" cursor to a nested table column
    * Using the SQL "THE" clause
******************************
/* Include files */
```



```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
                                                  /* SQL Communications Area */
#include <sqlca.h>
#include <coldemo1.h>
                            /* OTT-generated header with C typedefs for the */
                          /* database types city t and citytbl t */
#ifndef EXIT SUCCESS
# define EXIT SUCCESS
#endif
#ifndef EXIT FAILURE
# define EXIT FAILURE
#endif
#define CITY NAME LEN
                         30
#define COUNTY NAME LEN 30
#define MAX COUNTIES
/* Function prototypes */
#if defined( STDC )
void OptionLoop( void );
boolean GetCountyName( char *countyName );
void PrintCounties( void );
long CountyPopulation( CONST char *countyName );
void PrintCounty( CONST char *countyName );
void PrintSQLError( void );
void PrintCountyHeader( CONST char *county );
void PrintCity( city t *city );
#else
void OptionLoop();
boolean GetCountyName(/* char *countyName */);
void PrintCounties();
long CountyPopulation(/*_ CONST char *countyName _*/);
void PrintCounty(/*_ CONST char *countyName _*/);
void PrintSQLError(/*_ void _*/);
void PrintCountyHeader(/*_ CONST char *county _*/);
void PrintCity(/*_ city_t *city _*/);
#endif
* NAME
* main
* COLLECTION FEATURES
   none
int main()
 char * uid = "scott/tiger";
 EXEC SQL WHENEVER SQLERROR DO PrintSQLError();
 printf("\nPro*Census: Release California - Jan 1 1996.\n");
 EXEC SQL CONNECT : uid;
 OptionLoop();
 printf("\nGoodbye\n\n");
 EXEC SQL ROLLBACK RELEASE;
 return(EXIT SUCCESS);
/*
```

```
* NAME
 * OptionLoop
 * DESCRIPTION
  A command dispatch routine.
 * COLLECTION FEATURES
   none
 * /
void OptionLoop()
  char choice[30];
  boolean done = FALSE;
  char countyName[COUNTY_NAME_LEN + 1];
  while (!done)
    printf("\nPro*Census options:\n");
    printf("\tlist information for (A)ll counties\n");
    printf("\tlist information for one (C)ounty\n");
    printf("\tlist (P)opulation total for one county\n");
    printf("\t(Q)uit\n");
    printf("Choice? ");
    fgets(choice, 30, stdin);
    switch(toupper(choice[0]))
    {
    case 'A':
     PrintCounties();
     break;
    case 'C':
      if (GetCountyName(countyName))
    PrintCounty(countyName);
    case 'P':
      if (GetCountyName(countyName))
    printf("\nPopulation for %s county: ld\n",
           countyName, CountyPopulation(countyName));
     break;
    case 'Q':
      done = TRUE;
      break;
    default:
      break;
  }
* NAME
 * GetCountyName
 * DESCRIPTION
 * Fills the passed buffer with a client-supplied county name.
    Returns TRUE if the county is in the database, and FALSE otherwise.
 * COLLECTION FEATURES
    none
boolean GetCountyName(countyName)
  char *countyName;
  int count;
  int
      i;
```

```
printf("County name? ");
 fgets(countyName, COUNTY NAME LEN + 1, stdin);
  /* Convert the name to uppercase and remove the trailing '\n' */
 for (i = 0; countyName[i] != ' \setminus 0'; i++)
     countyName[i] = (char)toupper(countyName[i]);
      if (countyName[i] == '\n') countyName[i] = '\0';
 EXEC SQL SELECT COUNT(*) INTO :count
   FROM county tbl WHERE name = :countyName;
 if (count != 1)
     printf("\nUnable to find %s county.\n", countyName);
     return FALSE;
   }
 else
   return TRUE;
}
* NAME
* PrintCounties
* DESCRIPTION
  Prints the population and name of each city of every county
   in the database.
* COLLECTION FEATURES
    The following features correspond to the inline commented numbers
    1) Host variables for collection-typed objects are declared using
       OTT-generated types. Both array and scalar declarations are allowed.
       Scalar declarations must be of type pointer-to-collection-type, and
       array declarations must of type array-of-pointer-to-collection-type.
    2) SQL ALLOCATE should be used to allocate space for the collection.
       SQL FREE should be used to free the memory once the collection is
       no longer needed. The host variable being allocated or free'd
       can be either array or scalar.
    3) SQL is used to load into or store from collection-typed host variables
       and arrays. No special syntax is needed.
    4) The type of an indicator variable for a collection is OCIInd.
       An indicators for a collections is declared and used just like
       an indicator for an int or string.
    5) The COLLECTION GET Interface is used to access and manipulate the
       contents of collection-typed host variables. Each member of the
       collection used here has type city t, as generated by OTT.
void PrintCounties()
                                                                      /* 1 */
 citytbl t *cityTable[MAX COUNTIES];
                                                                      /* 4 */
 OCIInd cityInd[MAX_COUNTIES];
            county[MAX COUNTIES][COUNTY NAME LEN + 1];
 char
            i, numCounties;
  int.
 city t
           *city;
                                                                      /* 2 */
 EXEC SQL ALLOCATE :cityTable;
 EXEC SQL ALLOCATE :city;
 EXEC SQL SELECT name, cities
                                                                  /* 3, 4 */
   INTO :county, :cityTable:cityInd FROM county tbl;
```

```
numCounties = sqlca.sqlerrd[2];
  for (i = 0; i < numCounties; i++)</pre>
                                                                       /* 4 */
   if (cityInd[i] == OCI_IND_NULL)
      printf("Unexpected NULL city table for %s county\n", county[i]);
   else
                                                                       /* 5 */
       PrintCountyHeader(county[i]);
       EXEC SQL WHENEVER NOT FOUND DO break;
       while (TRUE)
         {
           EXEC SQL COLLECTION GET :cityTable[i] INTO :city;
           PrintCity(city);
       EXEC SQL WHENEVER NOT FOUND CONTINUE;
      }
 }
 EXEC SQL FREE :city;
 EXEC SQL FREE : cityTable;
                                                                       /* 2 */
}
/*
* NAME
* PrintCountyHeader
* COLLECTION FEATURES
* none
void PrintCountyHeader(county)
 CONST char *county;
{
 printf("\nCOUNTY: %s\n", county);
/*
* NAME
* PrintCity
* COLLECTION FEATURES
* none
* /
void PrintCity(city)
 city t *city;
{
 varchar newCITY[CITY NAME LEN];
 int newPOP;
 EXEC SQL OBJECT GET NAME, POPULATION from :city INTO :newCITY, :newPOP;
 printf("CITY: %.*s POP: %d\n", CITY NAME LEN, newCITY.arr, newPOP);
/*
* NAME
* PrintCounty
* DESCRIPTION
* Prints the population and name of each city in a particular county.
* COLLECTION FEATURES
* The following features correspond to the inline commented numbers
^{\star} 1) A ref cursor host variable may be used to scroll through the
```

```
rows of a collection.
   2) Use SQL ALLOCATE/FREE to create and destroy the ref cursor.
    3) The "CURSOR" clause in SQL can be used to load a ref cursor
       host variable. In such a case, the SELECT ... INTO does an
        implicit "OPEN" of the ref cursor.
 * IMPLEMENTATION NOTES
    In the case of SQL SELECT statements which contain an embedded
    CURSOR(...) clause, the Pro*C "select error" flag must be "no"
    to prevent cancellation of the parent cursor.
 * /
void PrintCounty(countyName)
 CONST char *countyName;
 sql cursor cityCursor;
                                                                      /* 1 */
 city t *city;
                                                                      /* 2 */
 EXEC SQL ALLOCATE :cityCursor;
 EXEC SQL ALLOCATE :city;
 EXEC ORACLE OPTION(select error=no);
 EXEC SOL SELECT
   CURSOR (SELECT VALUE (c) FROM TABLE (county tbl.cities) c)
     INTO :cityCursor
     FROM county tbl
                                                                      /* 3 */
     WHERE county tbl.name = :countyName;
  EXEC ORACLE OPTION(select error=yes);
  PrintCountyHeader(countyName);
 EXEC SQL WHENEVER NOT FOUND DO break;
  while (TRUE)
     EXEC SQL FETCH : cityCursor INTO : city;
     PrintCity(city);
 EXEC SQL WHENEVER NOT FOUND CONTINUE;
 EXEC SQL CLOSE :cityCursor;
                                                                     /* 2 */
 EXEC SQL FREE :cityCursor;
 EXEC SQL FREE :city;
}
 * NAME
 * CountyPopulation
 * DESCRIPTION
  Returns the number of people living in a particular county.
 * COLLECTION FEATURES
   The following features correspond to the inline commented numbers
    1) A "DECLARED" cursor may be used to scroll through the
       rows of a collection.
    2) The "THE" clause in SQL is used to convert a single nested-table
       column into a table.
long CountyPopulation(countyName)
 CONST char *countyName;
 long population;
  long populationTotal = 0;
```

```
EXEC SQL DECLARE cityCursor CURSOR FOR
   SELECT c.population
   FROM THE (SELECT cities FROM county_tbl
                                                             /* 1, 2 */
         WHERE name = :countyName) AS c;
 EXEC SQL OPEN cityCursor;
 EXEC SQL WHENEVER NOT FOUND DO break;
 while (TRUE)
     EXEC SQL FETCH cityCursor INTO :population;
     populationTotal += population;
 EXEC SQL WHENEVER NOT FOUND CONTINUE;
 EXEC SQL CLOSE cityCursor;
 return populationTotal;
* NAME
* PrintSQLError
* DESCRIPTION
* Prints an error message using info in sqlca and calls exit.
* COLLECTION FEATURES
* none
*/
void PrintSQLError()
 EXEC SQL WHENEVER SQLERROR CONTINUE;
 printf("SQL error occurred...\n");
 printf("\$.*s\n", (int) sqlca.sqlerrm.sqlerrml,
         (CONST char *)sqlca.sqlerrm.sqlerrmc);
 EXEC SQL ROLLBACK RELEASE;
 exit(EXIT_FAILURE);
```



# The Object Type Translator

This chapter discusses the OTT (Object Type Translator), which maps database object types, LOB types, and collection types to C structs for use in Pro\*C/C++ applications.

The chapter includes the following sections:

- OTT Overview
- What is the Object Type Translator
- AUsing OTT with OCI Applications
- About Using OTT with Pro\*C/C++ Applications
- OTT Reference

# 19.1 OTT Overview

OTT (The Object Type Translator) assists in the development of applications that make use of user-defined types in an Oracle 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, Pro\*C/C++, or Java developer can access objects stored in the tables.

An application that accesses object data must be able to represent the data in a host language format. This is accomplished by representing object types as C structs. It would be possible for a developer 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. 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 OTT. In OCI, the application also needs to call an initialization function generated by OTT.

In addition to creating structs that represent stored datatypes, OTT also generates parallel indicator structs which indicate whether an object type or its fields are NULL.

### **Related Topics**

- LOBs
- Objects
- Collections

# 19.2 What is 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.

OCI and Pro\*C/C++ developers must explicitly invoke OTT to translate database types to C representations. OCI developers 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 OTT. In Pro\*C/C++, the type version information is recorded in the OUTTYPE file which is passed as a parameter to Pro\*C/C++.

On most operating systems, 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 (for OCI developers). The following is an example of a command that invokes OTT:

```
ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=c hfile=demo.h
```

This command causes 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, <code>demoin.typ</code>. The resulting structs are output to the header file, <code>demo.h</code>, for the host language (C) specified by the <code>CODE</code> parameter. The <code>OUTTYPE</code> file, <code>demoout.typ</code>, receives information about the translation.

Each of these parameters is described in more detail in later sections of this chapter.

Sample demoin.typ file:

```
CASE=LOWER
TYPE employee

Sample demoout.typ file:

CASE = LOWER
TYPE SCOTT.EMPLOYEE AS employee
    VERSION = "$8.0"
    HFILE = demo.h
```

In this example, the <code>demoin.typ</code> file contains the type to be translated, preceded by TYPE (for example, TYPE employee). The structure of the <code>OUTTYPE</code> file is similar to the <code>INTYPE</code> file, with the addition of information obtained by OTT.

Once 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 OTT (demo.h) includes, among other items, the following declarations:

```
struct employee
{
    OCIString * name;
    OCINumber empno;
    OCINumber deptno;
    OCIDate hiredate;
    OCINumber salary;
};
typedef struct emp_type emp_type;
struct employee ind
```



```
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd deptno;
    OCIInd hiredate;
    OCIInd salary;
};
typedef struct employee_ind employee_ind;
```

The datatypes that appear in the struct declarations (for example, **OCIString** and **OCIInd**) are special datatypes which were new in Oracle 8.

The following sections describe these aspects of using OTT:

- About Creating Types in the Database
- About Invoking OTT
- The OTT Command Line
- The INTYPE File
- OTT Datatype Mappings
- NULL Indicator Structs
- OTT Support for Type Inheritance

The remaining sections of the chapter discuss the use of OTT with OCI and Pro\*C/C++, followed by a reference section that describes command line syntax, parameters, INTYPE file structure, nested #include file generation, schema names usage, default name mapping, and restrictions.

#### **Related Topics**

OTT Datatype Mappings

# 19.2.1 About Creating Types in the Database

The first step in using 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.

#### **Related Topics**

Objects

# 19.2.2 About Invoking OTT

The next step is to invoke OTT.

You can specify OTT parameters on the command line, or in a file called a configuration file. Certain parameters can also be specified in the INTYPE file.

If you specify a parameter 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.

For global options -- that is, options on the command line or options at the beginning of the INTYPE file before any TYPE statements -- the value on the command line overrides the value in the INTYPE file. (The options that can be specified globally in the INTYPE file are CASE, CODE, INITFILE, OUTDIR, and INITFUNC, but not HFILE.) Anything in the INTYPE file in a TYPE

specification applies to a particular type only, and overrides anything on the command line that would otherwise apply to the type. So if you enter TYPE person HFILE=p.h, it applies to person only and overrides the HFILE on the command line. The statement is not considered a command-line parameter.

### 19.2.2.1 Command Line

Parameters (also called options) set on the command line override any set elsewhere.

### **Related Topics**

Objects

## 19.2.2.2 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 can be named on the command line. In addition, a default configuration file is always read. This default configuration file must always exist, but can be empty. The name of the default configuration file is ottofg.cfg, and the location of the file is system-specific. See your platform-specific documentation for further information.

### 19.2.2.3 INTYPE File

The INTYPE file gives a list of types for OTT to translate.

The parameters CASE, HFILE, INITFUNC, and INITFILE can appear in the INTYPE file.

#### **Related Topics**

The INTYPE File

# 19.2.3 The OTT Command Line

On most platforms, 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 platformspecific documentation to see how to invoke OTT on your platform.

The following is an example (example 1) of an OTT invocation from the command line:

ott userid=scott/tiger intype=demoin.typ outtype=demoout.typ code=c hfile=demo.h



No spaces are permitted around the equals sign (=).

The following sections describe the elements of the command line used in this example.

### **Related Topics**

OTT Reference

### 19.2.3.1 OTT

Causes OTT to be invoked. It must be the first item on the command line.

### 19.2.3.2 Userid

Specifies the database connection information which OTT will use. In example one, OTT will attempt to connect with username *scott* and password *tiger*.

### 19.2.3.3 INTYPE

Specifies the name of the INTYPE file which will be used. In example 1, the name of the INTYPE file is specified as <code>demoin.typ</code>.

### 19.2.3.4 OUTTYPE

Specifies the name of the OUTTYPE file. When 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 that is translated, including its version string, and the header file to which its C representation was written.

In example one, the name of the OUTTYPE file is specified as demoout.typ.

### Note:

If the file specified by OUTTYPE already exists, it will be overwritten when OTT runs, with one exception: if the contents of the file as generated by OTT are identical to the previous contents of the file, 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.

### 19.2.3.5 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 value, 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.

### 19.2.3.6 HFILE

Specifies the name of the C header file to which the generated structs should be written. In example 1, the generated structs will be stored in a file called demo.h.





If the file specified by HFILE already exists, it will be overwritten when OTT runs, with one exception: if the contents of the file as generated by OTT are identical to the previous contents of the file, 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.

### 19.2.3.7 INITFILE

Specifies the use of the C source file into which the type initialization function is to be written.

The initialization function is only needed in OCI programs. In Pro\*C/C++ programs, the Pro\*C/C++ runtime library initializes types for the user.



If the file specified by INITFILE already exists, it will be overwritten when OTT runs, with one exception: if the contents of the file as generated by OTT are identical to the previous contents of the file, 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.

### 19.2.3.8 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 INITELLE.

This function is only needed in OCI programs.

# 19.2.4 The INTYPE File

When you run OTT, the INTYPE file tells OTT which database types should be translated. It can also control the naming of the generated structs. You can create the INTYPE file, or use the OUTTYPE file of a previous invocation of OTT. If the INTYPE parameter is not used, all types in the schema to which 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.

The lines that begin with the TYPE keyword specify which types in the database should be translated. In this case, the *EMPLOYEE*, *ADDRESS*, *ITEM*, *PERSON*, and *PURCHASE\_ORDER* types.

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 you do not use AS 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 can happen, for example, if the program uses two types with the same name from different schemas.
- the developer prefers a different name

OTT may need to translate additional types that are not listed in the INTYPE file. This is because 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*, OTT would still translate *ADDRESS* because it is required to define the *Person* type.



As of Release 1 (9.0.1), users may specify TRANSITIVE = FALSE to *not* generate types that are not specified in the INTYPE file. TRANSITIVE is set to TRUE by default.

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, for example, CREATE TYPE "Person". A SQL identifier is case-sensitive if it was quoted when it was declared.

Quotation marks can also be used to refer to a SQL identifier that is also an OTT-reserved word, for example, TYPE "CASE". In this case, the quoted name must be in upper case if the SQL identifier was created in a case-insensitive manner, for example, CREATE TYPE Case. If an OTT-reserved word is used to refer to the name of a SQL identifier but is not quoted, OTT will report a syntax error in the INTYPE file.

### **Related Topics**

CASE



Structure of the INTYPE File

# 19.2.5 OTT Datatype Mappings

When 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 that are used in Oracle object data types. The datatypes found in Oracle include a set of predefined, primitive types, and provide for the creation of user-defined types, like object types and collections.

The set of predefined types includes standard types that are familiar to most developers, including number and character types. It also includes datatypes that were introduced with Oracle8 (for example, BLOB or CLOB).

Oracle also includes a set of predefined types that 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
  OCIString * 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 "NULL Indicator Structs".

The datatypes in the struct declarations—<code>OCIString</code>, <code>OCINumber</code>, <code>OCIDate</code>, <code>OCIInd</code>—are C mappings of object types introduced in Oracle8. They are used here to map the datatypes of the object type attributes. The <code>NUMBER</code> datatype of the <code>empno</code> attribute, maps to the new <code>OCINumber</code> datatype, for example. These new datatypes can also be used as the types of bind and define variables.

### **Related Topics**

OCI Data Structures

## 19.2.5.1 Mapping Object Datatypes to C

This section describes the mappings of object attribute types to C types generated by OTT. Table 19-1 lists the mappings from types that can be used as attributes of object datatypes that are generated by OTT.

Table 19-1 Object Datatype Mappings for Object Type Attributes

Object Attribute Types	C Mapping
VARCHAR2(N)	OCIString *
VARCHAR(N)	OCIString *
CHAR(N), CHARACTER(N)	OCIString *
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 OCIRef *.
RAW(N)	OCIRaw *

Table 19-2 shows the mappings of named collection types to object datatypes generated by OTT:

Table 19-2 Object Datatype Mappings for Collection Types

Named Collection Type	C Mapping
VARRAY	Declared using typedef; equivalent to OCIArray *.
NESTED TABLE	Declared using typedef; equivalent to OCITable *.



For REF, VARRAY, and NESTED TABLE types, OTT generates a typedef. The type declared in the typedef is then used as the type of the data member in the struct declaration.

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, OTT first generates the nested type (if TRANSITIVE = TRUE.) It then maps the object type attribute to a nested struct of the type of the nested object type.

The C datatypes to which OTT maps non-object database attribute types are structures, which, except for OCIDate, are opaque.

### **Related Topics**

OTT Type Mapping Example

# 19.2.5.2 OTT Type Mapping Example

The following example demonstrates the various type mappings created by OTT.

Given the following database types:

#### and an INTYPE file that includes:

```
CASE = LOWER
TYPE many_types
```

### OTT would generate the following C structs:



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 OCIRef many types ref;
typedef OCIRef object_type_ref;
typedef OCIArray my_varray;
                                        /* part of many types */
typedef OCITable my_table;
                                          /* part of many types*/
typedef OCIRef other type ref;
                                          /* part of many_types */
struct object type
   OCIString * object_name;
};
typedef struct object_type object_type;
struct object_type_ind
                                           /*indicator struct for*/
                                          /*object types*/
  OCIInd atomic;
  OCIInd object name;
typedef struct object type ind object type ind;
struct many types
{
  OCIString * the_varchar;
OCIString * the_char;
  OCIBlobLocator * the blob;
  OCIClobLocator * the_clob;
  struct object type the object;
  other_type_ref * another_ref;
many_types_ref * the_ref;
  my_varray * the_varray;
my_table * the_date;
OCIDate the_num;
OCIRaw * the_raw;
typedef struct many_types many_types;
                                           /*indicator struct for*/
struct many_types_ind
                                          /*many_types*/
   OCIInd atomic;
   OCIInd the varchar;
   OCIInd the char;
   OCIInd the blob;
   OCIInd the clob;
   struct object_type_ind the_object; /*nested*/
   OCIInd another ref;
  OCIInd the ref;
   OCIInd the varray;
   OCIInd the table;
  OCIInd the date;
  OCIInd the num;
   OCIInd the raw;
typedef struct many types ind many types ind;
#endif
```

Notice that even though only one item was listed for translation in the INTYPE file, two object types and two named collection types were translated. This is because the OTT parameter "TRANSITIVE", has the default value of TRUE. As described in that section, when TRANSITIVE=TRUE, 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 that 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 OCIRef many_types_ref;
typedef OCIRef object_type_ref;
typedef OCIArray my_varray;
typedef OCITable my_table;
typedef OCIRef 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;
    ...
}
```

### 19.2.6 NULL Indicator Structs

Each time 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 can 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
{
    OCIInd _atomic;
    OCIInd the_varchar;
    OCIInd the_char;
    OCIInd the_blob;
    OCIInd the_clob;
    struct object_type_ind the_object;
    OCIInd another_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 earlier 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 <code>OCIInd</code>.

For more information about atomic NULLness, refer to the discussion of object types in Data Types.

# 19.2.7 OTT Support for Type Inheritance

To support type inheritance of objects, OTT generates a C struct to represent an object subtype by declaring the inherited attributes in an encapsulated struct with the special name '\_super', before declaring the new attributes. Thus, for an object subtype that inherits from a supertype, the first element in the struct is named '\_super', followed by elements corresponding to each attribute of the subtype. The type of the element named '\_super' is the name of the supertype.

For example, given the type Person\_t, with subtype Student\_t and subtype Employee\_t, which are created as follows:

and, given an INTYPE file which includes:

```
CASE=SAME
TYPE EMPLOYEE_T
TYPE STUDENT_T
TYPE PERSON T
```

OTT generates the following C structs for Person\_t, Student\_t, and Employee\_t and their null indicator structs:

```
#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif

typedef OCIRef EMPLOYEE_T_ref;
typedef OCIRef STUDENT_T_ref;
typedef OCIRef PERSON_T_ref;

struct PERSON_T
{
   OCINumber SSN;
   OCIString * NAME;
   OCIString * ADDRESS;
```

```
typedef struct PERSON T PERSON T;
struct PERSON T ind
  OCIInd atomic;
  OCIInd SSN;
  OCIInd NAME;
  OCIInd ADDRESS;
};
typedef struct PERSON T ind PERSON T ind;
struct EMPLOYEE T
  PERSON_T _super;
  OCINumber EMPID;
  OCIString * MGR;
typedef struct EMPLOYEE T EMPLOYEE T;
struct EMPLOYEE T ind
  PERSON T super;
  OCIInd EMPID;
  OCIInd MGR;
typedef struct EMPLOYEE T ind EMPLOYEE T ind;
struct STUDENT T
  PERSON T super;
  OCINumber DEPTID;
  OCIString * MAJOR;
typedef struct STUDENT_T STUDENT_T;
struct STUDENT T ind
  PERSON T super;
  OCIInd DEPTID;
  OCIInd MAJOR;
typedef struct STUDENT T ind STUDENT T ind;
#endif
```

The earlier C mapping convention allows simple up-casting from an instance of a subtype to an instance of a supertype in C to work properly. For example:

The null indicator structs are generated similarly. Note that for the supertype  $Person_t$  null indicator struct, the first element is '\_atomic', and that for the subtypes  $Employee_t$  and  $Student_t$  null indicator structs, the first element is '\_super' (no atomic element is generated for subtypes).

### 19.2.7.1 Substitutable Object Attributes

For attributes of NOT FINAL types (and therefore potentially substitutable), the embedded attribute is represented as a pointer.

#### Consider a type Book t created as:

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

The corresponding C struct generated by OTT contains a pointer to Person t:

```
struct Book_t
{
   OCIString *title;
   Person_t *author; /* pointer to Person_t struct */
}
```

The null indicator struct corresponding to the earlier type is:

```
struct Book_t_ind
{
   OCIInd _atomic;
   OCIInd title;
   OCIInd author;
}
```

Note that the null indicator struct corresponding to the author attribute can be obtained from the author object itself.

If a type is defined to be FINAL, it cannot have any subtypes. An attribute of a FINAL type is therefore not substitutable. In such cases, the mapping is as before: the attribute struct is inline. Now, if the type is altered and defined to be NOT FINAL, the mapping will have to change. The new mapping is generated by running OTT again.

#### **Related Topics**

OBJECT GET

### 19.2.8 The OUTTYPE File

The OUTTYPE file is named on the OTT command line. When 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 that 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 that should be translated. In two of these types, naming conventions are specified.

The following example shows what the OUTTYPE file looks like after running 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 that 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.

When the parameter TRANSITIVE has been set to TRUE (it is the default), OTT analyzes the types in the INTYPE file for type dependencies before performing the translation, and translates other types as necessary.

## 19.3 Using OTT with OCI Applications

C header and implementation files that have been generated by OTT can be used by an OCI application that accesses objects in a database server. Incorporate the header file 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.

Figure 19-1 shows the steps involved in using OTT with OCI.

- 1. SQL is used to create type definitions in the database.
- 2. 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.
- The OCI application, including the implementation file generated by OTT, is compiled and linked with the OCI libraries.
- 6. The OCI executable is run against the Oracle Server.

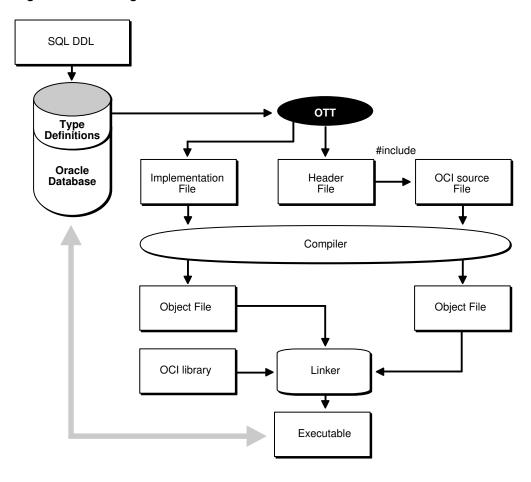


Figure 19-1 Using OTT with OCI

### 19.3.1 About 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 that 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 specifically designed to work on attributes of object types and named collection types.

Some of the available functions follow:

- 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 the following chapters of *Oracle Call Interface Developer's Guide*:

- Chapter 2, which covers OCI concepts, including binding and defining
- · Chapter 6, which covers object access and navigation
- Chapter 7, which covers datatype mapping and manipulation
- Chapter 12, which lists datatype mapping and manipulation functions

### 19.3.2 About Calling the Initialization Function

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 can specify a name for the initialization function when invoking OTT with the INITFUNC option, or may allow 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 OTT separately for each piece requiring, for each piece, one initialization file, containing an initialization function.

After you create an environment handle by an explicit OCI object call, for example, by calling <code>OCIEnvInit()</code>, you must also call the initialization functions explicitly for each environment handle. This gives each handle access to all the datatypes used in the entire program.

If an environment handle is implicitly created using 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 relevant for Pro\*C/C++ applications, or 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 SCOTT.PERSON
TYPE SCOTT.ADDRESS
```

#### and the command line

ott userid=scott/tiger intype=ex2c outtype=ex2co hfile=ex2ch.h initfile=ex2cv.c

#### OTT generates the following to the file ex2cv.c:

```
"ADDRESS", 7,
"$8.0", 4);
return status;
}
```

The function *ex2cv* creates the type version table and inserts the types SCOTT.PERSON and SCOTT.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 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 OTT must also be compiled and linked into program P. Doing this correctly is your responsibility.

### 19.3.3 Tasks of the Initialization Function

The C initialization function supplies version information about the types processed by 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 the Open Type Manager (OTM) to determine which version of a type a particular program uses. Different initialization functions generated by OTT at different times may add some of the same types to the type version table. When a type is added more than once, OTM ensures that the same version of the type is registered each time.

It is the OCI developer's responsibility to declare a function prototype for the initialization function, and to call the function.



In the current release of Oracle, each type has only one version. Initialization of the type version table is required only for compatibility with future releases of Oracle.

## 19.4 About Using OTT with Pro\*C/C++ Applications

When building Pro\*C/C++ applications, the type-translation process can be simpler than when building OCI-based applications. This is because precompiler-generated code will automatically initialize the type version table.

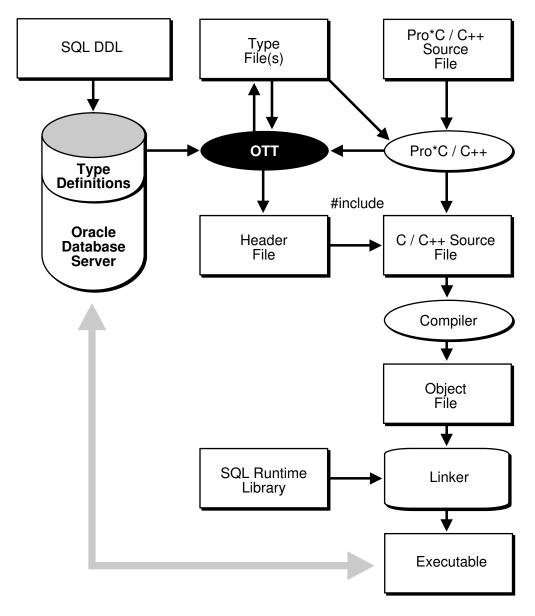
A C header file generated by OTT can be used by a Pro\*C/C++ application to access objects in a database server. The header file is incorporated into the code with an #include statement. Once the header file has been included, the Pro\*C/C++ application can access and manipulate object data in the host language format.

Figure 19-2 shows the steps involved in using OTT with Pro\*C/C++.

SQL is used to create type definitions in the database.

- OTT generates a header file containing C representations of object types, REF types, and named collection types. It also generates an OUTTYPE file that is passed as the INTYPE parameter to Pro\*C/C++.
- 3. The header file is included in a Pro\*C/C++ source code file.
- 4. The Pro\*C/C++ application is compiled and linked with the Pro\*C/C++ run-time library SQLLIB.
- 5. The Pro\*C/C++ executable is run against the Oracle Server.

Figure 19-2 Building an Object-oriented Pro\*C/C++ Application



As noted in step 2, earlier, the OUTTYPE file generated by OTT serves a special purpose for Pro\*C/C++ developers. Pass the OUTTYPE file to the new INTYPE command line parameter when invoking Pro\*C/C++. The contents of this file are used by the precompiler to determine which database types correspond to which OTT-generated structs. OCI developers must make this association explicitly through the use of special bind, define, and type information access functions.

Also, the precompiler generates code to initialize the type version table with the types named in the OTT OUTTYPE (Pro\*C/C++ INTYPE) file.



Oracle recommends that the OUTTYPE file from OTT always serve as the INTYPE file to Pro\*C/C++. It would be possible for you to write an INTYPE file for Pro\*C/C++, but this is not recommended, due to the possibility of errors being introduced.

One way to manipulate the attributes of objects retrieved from the server is to call the OCI datatype mapping and manipulation functions. Before doing this, the application must first call SQLEnvGet() to obtain an OCI environment handle to pass to the OCI functions, and SQLSvcCtxGet() to obtain an OCI service context to pass to the OCI functions. There are also Pro\*C facilities that can be used to manipulate object attributes.

#### **Related Topics**

- Objects
- About Accessing and Manipulating Objects with OCI
- olink:LNOCI030

## 19.5 OTT Reference

Behavior of 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
- Restriction

The following conventions are used in this chapter to describe OTT syntax:

- Italic strings are to be supplied by the user.
- Strings in UPPERCASE are entered as shown, except that case is not significant.
- 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 earlier are entered as shown. These include '.', '@', and so on.



## 19.5.1 OTT Command Line Syntax

The OTT command-line interface is used when explicitly invoking OTT to translate database types into C structs. This is always required when developing OCI applications or Pro\*C/C++ applications that use objects.

An OTT command-line statement consists of the keyword OTT, followed by a list of OTT parameters.

The parameters that 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]
[INITFILE=filename]
[INITFUNC=filename]
[CASE={SAME|LOWER|UPPER|OPPOSITE}]
[SCHEMA_NAMES={ALWAYS|IF_NEEDED|FROM_INTYPE}]
[TRANSITIVE=TRUE|FALSE]
```

#### 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 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 (enter it as one line):

```
OTT userid=scott/tiger intype=in.typ outtype=out.typ code=c hfile=demo.h errtype=demo.tls case=lower
```

Each of the OTT command line parameters is described in the following sections.

### 19.5.2 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.

#### 19.5.2.1 USERID

The USERID parameter specifies the Oracle username, password, and optional database name (Oracle Net database specification string). If the database name is omitted, the default database is assumed. 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 you omit it, OTT automatically attempts to connect to the default database as user CLUSTER\$username, where username is the user's operating system user name.

#### 19.5.2.2 INTYPE

The INTYPE parameter specifies the name of the file from which to read the list of object type specifications. 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.

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.

#### **Related Topics**

Structure of the INTYPE File

#### 19.5.2.3 OUTTYPE

The name of a file into which 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 (if TRANSITIVE=TRUE) 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 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.

#### 19.5.2.4 CODE

CODE= C|KR C|ANSI C

This is the desired host language for OTT output, that may be specified as <code>CODE=KR\_C</code>, <code>CODE=KR\_C</code>, or <code>CODE=ANSI C</code>. "CODE=C" is equivalent to "CODE=ANSI C".

There is no default value for this parameter; it must be supplied.

#### 19.5.2.5 INITFILE

The INITFILE parameter specifies the name of the file where the OTT-generated initialization file is to be written. OTT does not generate the initialization function if you omit this parameter.

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

#### 19.5.2.6 INITFUNC

The INITFUNC parameter is used only in OCI programs. It specifies the name of the initialization function generated by OTT. If this parameter is omitted, the name of the initialization function is derived from the name of the INITFILE.

INITFUNC=filename

### 19.5.2.7 HFILE

The name of the include (.h) file to be generated by 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 (if TRANSITIVE=TRUE), 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



#### 19.5.2.8 CONFIG

The CONFIG parameter specifies the name of the OTT configuration file, that 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.

#### 19.5.2.9 ERRTYPE

If you supply this parameter, 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 that 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

#### 19.5.2.10 CASE

This parameter affects the case of certain C identifiers generated by 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 parameter affects only those identifiers (attributes or types not explicitly listed) not mentioned in the INTYPE file. Case conversion takes place after a legal identifier has been generated.



The case of the C 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 OTT will generate

struct Worker {...};

On the other hand, if the INTYPE file were written as



TYPE wOrKeR

#### OTT would generate

```
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.

### 19.5.2.11 SCHEMA NAMES

This parameter offers control in qualifying the database name of a type from the default schema with a schema name in the <code>OUTTYPE</code> file. The <code>OUTTYPE</code> file generated by OTT contains information about the types processed by OTT, including the type names.

#### **Related Topics**

SCHEMA\_NAMES Usage

#### 19.5.2.12 TRANSITIVE

Takes the values TRUE (the default) or FALSE. Indicates whether type dependencies not explicitly listed in the INTYPE file are to be translated, or not. If TRANSITIVE=FALSE is specified, then types *not* mentioned in the INTYPE file are not generated. This is the case even if they were used as attribute types of other generated types.

### 19.5.3 Where OTT Parameters Can Appear

Supply OTT parameters on the command line, in a CONFIG file named on the command line, or both. Some parameters are also allowed in the INTYPE file.

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 named *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. Each line in the configuration may contain one parameter, with no whitespace on the line.

If OTT is executed without any arguments, an on-line parameter reference is displayed.

The types for 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 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.



### 19.5.4 Structure of the INTYPE File

The INTYPE and OUTTYPE files list the types translated by 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 OTT generates from it, see "OTT Support for Type Inheritance".

### 19.5.4.1 INTYPE File Type Specifications

A type specification in the INTYPE names an object datatype that is to be translated. The following is an example of a user-created INTYPE file:

```
TYPE employee
TRANSLATE SALARY$ AS salary
DEPTNO AS department
TYPE ADDRESS
TYPE PURCHASE_ORDER AS p_o
```

The structure of a type specification is as follows:

```
TYPE type_name [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 that owns the given object datatype, and *type\_name* is the name of the type. The default schema is that of the user running OTT. The default database is the local database.

The components of a type specification are:

- type name is the name of an 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 that was used when the code was generated
  by a previous invocation of OTT. The version string is generated by OTT and written to the
  OUTTYPE file, that may later be used as the INTYPE file when 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.



- 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.
- *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. You can
  specify identifiers in this way for any number of attributes. The default name mapping
  algorithm will be used for the attributes that are not mentioned.

An object datatype may need to be translated for one of two reasons:

- It appears in the INTYPE file.
- It is required to declare another type that must be translated, and TRANSITIVE = TRUE.

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.

#### **Related Topics**

Default Name Mapping

### 19.5.5 Nested #include File Generation

Every HFILE generated by OTT #includes other necessary files, and #defines a symbol constructed from the name of the file, that 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 pxl
hfile tott95a.h
type px3
hfile tott95b.h
```

#### If we invoke 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 TOTT95B_ORACLE
#define TOTT95B_ORACLE
#ifndef OCI_ORACLE
#include <oci.h>
#endif
#ifndef TOTT95A_ORACLE
#include "tott95a.h"
#endif
typedef OCIRef px3 ref;
```



```
struct px3
{
    struct px1 col1;
};
typedef struct px3 px3;
struct px3_ind
{
    OCIInd _atomic;
    struct px1_ind col1
};
typedef struct px3_ind px3_ind;
#endif
```

#### File tott95a.h is:

```
#ifndef TOTT95A ORACLE
#define TOTT95A ORACLE
#ifndef OCI ORACLE
#include <oci.h>
#endif
typedef OCIRef px1_ref;
struct px1
  OCINumber col1;
  OCINumber col2;
typedef struct px1 px1;
struct px1 ind
  OCIInd atomic;
  OCIInd col1;
  OCIInd col2;
typedef struct px1 ind px1 ind;
#endif
```

In this file, the symbol TOTT95B\_ORACLE is defined first so that the developer may conditionally include <code>tott95b.h</code> without having to worry whether <code>tott95b.h</code> depends on the include file using the following construct:

```
#ifndef TOTT95B_ORACLE
#include "tott95b.h"
#endif
```

Using this technique, you can 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 OTT includes oci.h, that contains type and function declarations that the Pro\*C/C++ or OCI developer can find useful. This is the only case in which OTT uses angle brackets in an #include.

Next, the file tott95a.h is included because it contains the declaration of "struct px1", that tott95b.h requires. When the INTYPE file requests that type declarations be written to more than one file, OTT will determine which other files each HFILE must include, and will generate the necessary #includes.

Note that OTT uses quotes in this #include. When a program including tott95b.h is compiled, the search for tott95a.h begins where the source program was found, and will thereafter follow an implementation-defined search rule. If tott95a.h cannot be found in this way, a

complete file name (for example, a UNIX absolute path name beginning with /) should be used in the INTYPE file to specify the location of tott95a.h.

### 19.5.6 SCHEMA\_NAMES Usage

This parameter affects whether the name of a type from the default schema to which OTT is connected is qualified with a schema name in the OUTTYPE file.

The name of a type from a schema other that the default schema is always qualified with a schema name in the OUTTYPE file.

The schema name, or its absence, determines in which schema the type is found during program execution.

There are three 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 is written with a schema name only if the first type encountered by OTT that depends on it was written with a schema name. However, a type that is not in the default schema to which OTT is connected is always written with an explicit schema name.

The OUTTYPE file generated by OTT 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 OTT used.

#### An Example

If  ${\tt SCHEMA\_NAMES}$  is set to  ${\tt FROM}$  INTYPE, and the  ${\tt INTYPE}$  file reads:

```
TYPE Person
TYPE joe.Dept
TYPE sam.Company
```

then the Pro\*C/C++ application that uses the OTT-generated structs will use the types sam.Company, joe.Dept, and Person. Person without a schema name refers to the Person type in the schema to which the application is connected.

If OTT and the application both connect to schema joe, the application will use the same type (joe.Person) that OTT used. If OTT connected to schema joe but the application connects to schema mary, the application will use the type mary.Person. This behavior is appropriate only if the same "CREATE TYPE Person" statement has been executed in schema joe and schema mary.

On the other hand, the application will use type joe. 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, 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 OTT connects to schema joe,  $SCHEMA\_NAMES=FROM\_INTYPE$  is specified, and the user's INTYPE files include either

```
TYPE Person or TYPE joe.Person
```

but do not mention the type joe.Address, which is used as a nested object type in type *joe.Person*. If "TYPE joe.Person" appeared in the INTYPE file, "TYPE joe.Person" and "TYPE joe.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 *joe.Address* type is embedded in several types translated by OTT, but is not explicitly mentioned in the INTYPE file, the decision of whether to use a schema name is made the first time OTT encounters the embedded *joe.Address* type. If, for some reason, the user wants type *joe.Address* to have a schema name but does not want type *Person* to have one, you must explicitly request

```
TYPE joe.Address
```

in the INTYPE FILE.

In the usual case in which each type is declared in a single schema, it is safest for you to qualify all type names with schema names in the INTYPE file.

### 19.5.7 Default Name Mapping

When 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 OTT. Next, if a translation of the resulting name is supplied in the INTYPE file, that translation is used. Otherwise, OTT translates the name character-by-character to the compiler character set, applying the CASE option. The following describes this in more detail.

When OTT reads the name of a database entity, the name is automatically translated from the database character set to the character set used by OTT. In order for 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 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 OTT uses by setting the NLS\_LANG environment variable, or by some other platform-specific mechanism.

Once OTT has read the name of a database entity, it translates the name from the character set used by OTT to the compiler's character set. If a translation of the name appears in the INTYPE file, OTT uses that translation.

Otherwise, OTT attempts to translate the name as follows:

- 1. First, if the OTT character set is a multibyte character set, all multibyte characters in the name that have singlebyte equivalents are converted to those singlebyte equivalents.
- 2. Next, the name is converted from the OTT character set to the compiler character set. The compiler character set is a singlebyte 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, OTT gives a warning message. If all the characters in a name are replaced by underscores, OTT gives an error message.

Character-by-character name translation does not alter underscores, digits, or singlebyte letters that appear in the compiler character set, so legal C identifiers are not altered.

Name translation may, for example, translate accented singlebyte characters such as "o" with an umlaut or "a" with an accent grave to "o" or "a", and may translate a multibyte letter to its singlebyte equivalent. Name translation will typically fail if the name contains multibyte characters that lack singlebyte equivalents. In this case, the user must specify name translations in the INTYPE file.

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.

### 19.5.8 Restriction

The following restriction affects the use of OTT.

### 19.5.8.1 File Name Comparison

Currently, 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 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/smith/foo1.h, OTT should generate one #include if the two files are the same, and two #includes if the files are different. In practice, though, it concludes that the two files are different, and generates two #includes, as follows:

#ifndef FOO1\_ORACLE
#include "foo1.h"
#endif
#ifndef FOO1\_ORACLE
#include "/private/smith/foo1.h"
#endif



If foo1.h and /private/smith/foo1.h are different files, only the first one will be included. If foo1.h and /private/smith/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.



20

## **User Exits**

This chapter focuses on writing user exits for your Oracle Tools applications. You learn how C subroutines can do certain jobs more quickly and easily than SQL\*Forms and Oracle Forms. This chapter contains the following topics:

- What Is a User Exit?
- Why Write a User Exit?
- About Developing a User Exit
- About Writing a User Exit
- EXEC TOOLS Statements
- About Calling a User Exit
- About Passing Parameters to a User Exit
- About Returning Values to a Form
- An Example of Using User Exits
- About Precompiling and Compiling a User Exit
- Example Program: A User Exit
- About Using the GENXTB Utility
- About Linking a User Exit into SQL\*Forms
- Guidelines

This chapter is supplemental. For more information about user exits, refer to the *SQL\*Forms Designer's Reference*, the *Oracle Forms Reference Manual, Vol. 2*, and your system-specific Oracle documentation.

### 20.1 What Is a User Exit?

A *user exit* is a C subroutine written by you and called by Oracle Forms to do special-purpose processing. You can embed SQL statements and PL/SQL blocks in your user exit, then precompile it as you would a host program.

When called by an Oracle Forms V3 trigger, the user exit runs, then returns a status code to Oracle Forms. Your exit can display messages on the Oracle Forms status line, get and set field values, do high-speed computations and table lookups, and manipulate Oracle data.

## 20.2 Why Write a User Exit?

SQL\*Forms provides the ability to use PL/SQL blocks in triggers. So, in most cases, instead of calling a user exit, you can use the procedural power of PL/SQL. If the need arises, you can call user exits from a PL/SQL block with the USER\_EXIT function. User exits are harder to write and implement than SQL, PL/SQL, or SQL\*Forms commands. So, you will probably use them only to do processing that is beyond the scope of SQL, PL/SQL, and SQL\*Forms. Some common uses follow:

- Operations more quickly or easily done in a third generation languages like C (numerical integration, for instance)
- Controlling real time devices or processes (issuing a sequence of instructions to a printer or graphics device, for example)
- Data manipulations that need extended procedural capabilities (recursive sorting, for example)
- Special file I/O operations

## 20.3 About Developing a User Exit

This section outlines the way to develop a SQL\*Forms user exit; later sections go into more detail.

To incorporate a user exit into a form, you take the following steps:

- Write the user exit in Pro\*C.
- Precompile the source code.
- Compile the. c file from step 2.
- Use the GENXTB utility to create a database table, IAPXTB.
- Use the GENXTB form in SQL\*Forms to insert your user exit information into the table.
- Use the GENXTB utility to read the information from the table and create an IAPXIT source code module. Then compile the source code module.
- Create a new SQL\*Forms executable by linking the standard SQL\*Forms modules, your user exit object, and the IAPXIT object created in step 6.
- In the form, define a trigger to call the user exit.
- Instruct operators to use the new IAP when running the form. This is unnecessary if the new IAP replaces the standard one. For details, see the Oracle installation or user's guide for your system.

#### **Related Topics**

EXEC TOOLS Statements

## 20.4 About Writing a User Exit

You can use the following kinds of statements to write your SQL\*Forms user exit:

- C code
- EXEC SQL
- EXEC ORACLE
- EXEC TOOLS

This section focuses on the EXEC TOOLS statements, which let you pass values between SQL\*Forms and a user exit.

### 20.4.1 Requirements for Variables

The variables used in EXEC TOOLS statements must correspond to field names used in the form definition.



## 20.5 EXEC TOOLS Statements

EXEC TOOLS statements support the basic Oracle Toolset by providing a generic way to handle get, set, and exception callbacks from user exits. The following discussion focuses on Oracle Forms but the same concepts apply to Oracle Report.

### 20.5.1 About Writing a Toolset User Exit

Besides EXEC SQL, EXEC ORACLE, and host language statements, you can use the following EXEC TOOLS statements to write an Oracle Forms user exit:

- SET
- GET
- SET CONTEXT
- GET CONTEXT
- MESSAGE

The EXEC TOOLS GET and SET statements replace the EXEC IAF GET and PUT statements used with earlier versions of Oracle Forms. Unlike IAF GET and PUT, however, TOOLS GET and SET accept indicator variables. The EXEC TOOLS MESSAGE statement replaces the message-handling function *sqliem*. Now, let us take a brief look at all the EXEC TOOLS statements. For more information, see the *Oracle Forms Reference Manual*, *Vol 2*.

### 20.5.2 EXEC TOOLS SET

The EXEC TOOLS SET statement passes values from a user exit to Oracle Forms. Specifically, it assigns the values of host variables and constants to Oracle Forms variables and items. Values passed to form items display after the user exit returns control to the form. To code the EXEC TOOLS SET statement, you use the syntax

```
EXEC TOOLS SET form_variable[, ...]
    VALUES ({:host_variable :indicator | constant}[, ...]);
```

where *form\_variable* is an Oracle Forms field, block.field, system variable, or global variable, or a host variable (prefixed with a colon) containing the value of one of the foregoing items. In the following example, a user exit passes an employee name to Oracle Forms:

```
char ename[20];
short ename_ind;
...
    strcpy(ename, "MILLER");
    ename_ind = 0;
    EXEC TOOLS SET emp.ename VALUES (:ename :ename ind);
```

In this example, *emp.ename* is an Oracle Forms block.field.

### 20.5.3 EXEC TOOLS GET

The EXEC TOOLS GET statement passes values from Oracle Forms to a user exit. Specifically, it assigns the values of Oracle Forms variables and items to host variables. As soon as the values are passed, the user exit can use them for any purpose. To code the EXEC TOOLS GET statement, you use the syntax

```
EXEC TOOLS GET form_variable[, ...]
    INTO :host variable:indicator[, ...];
```

where *form\_variable* is an Oracle Forms field, block.field, system variable, or global variable, or a host variable (prefixed with a colon) containing the value of one of the foregoing items. In the following example, Oracle Forms passes an item name from a block to your user exit:

```
char name_buff[20];
VARCHAR name_fld[20];

strcpy(name_fld.arr, "EMP.NAME");
name_fld.len = strlen(name_fld.arr);
EXEC TOOLS GET :name fld INTO :name buff;
```

### 20.5.4 EXEC TOOLS SET CONTEXT

The EXEC TOOLS SET CONTEXT statement saves context information from a user exit for later use in another user exit. A pointer variable points to a block of memory in which the context information is stored. With SET CONTEXT, you need not declare a global variable to hold the information. To code the EXEC TOOLS SET CONTEXT statement, you use the syntax

```
EXEC TOOLS SET CONTEXT :host_pointer_variable
   IDENTIFIED BY context name;
```

where *context\_name* is an undeclared identifier or a character host variable (prefixed with a colon) that names the context area.

```
char *context_ptr;
char context[20];
strcpy(context, "context1")
EXEC TOOLS SET CONTEXT :context IDENTIFIED BY application1;
```

### 20.5.5 EXEC TOOLS GET CONTEXT

The EXEC TOOLS GET CONTEXT statement retrieves context information (saved earlier by SET CONTEXT) into a user exit. A host-language pointer variable points to a block of memory in which the context information is stored. To code the EXEC TOOLS GET CONTEXT statement, you use the syntax

where *context\_name* is an undeclared identifier or a character host variable (prefixed with a colon) that names the context area. In the following example, your user exit retrieves context information 1saved earlier:

```
char *context_ptr;

EXEC TOOLS GET CONTEXT application1 INTO :context ptr;
```

### 20.5.6 EXEC TOOLS MESSAGE

The EXEC TOOLS MESSAGE statement passes a message from a user exit to Oracle Forms. The message is displayed on the Oracle Forms message line after the user exit returns control to the form. To code the EXEC TOOLS MESSAGE statement, you use the syntax

```
EXEC TOOLS MESSAGE message text [severity code];
```

where *message\_text* is a quoted string or a character host variable (prefixed with a colon), and the optional *severity\_code* is an integer constant or an integer host variable (prefixed with a colon). The MESSAGE statement does *not* accept indicator variables. In the following example, your user exit passes an error message to Oracle Forms:

```
EXEC TOOLS MESSAGE 'Bad field name! Please reenter.';
```

## 20.6 About Calling a User Exit

You call a user exit from a SQL\*Forms trigger using a packaged procedure named USER\_EXIT (supplied with SQL\*Forms). The syntax you use is

```
USER_EXIT(user_exit_string [, error_string]);
```

where *user\_exit\_string* contains the name of the user exit plus optional parameters and *error\_string* contains an error message issued by SQL\*Forms if the user exit fails. For example, the following trigger command calls a user exit named LOOKUP:

```
USER EXIT('LOOKUP');
```

Notice that the user exit string is enclosed by single (not double) quotes.

## 20.7 About Passing Parameters to a User Exit

When you call a user exit, SQL\*Forms passes it the following parameters automatically:

Parameters	Description
Command Line	Is the user exit string.
Command Line Length	Is the length (in characters) of the user exit string.
Error Message	Is the error string (failure message) if one is defined.
Error Message Length	Is the length of the error string.
In-Query	Is a Boolean value indicating whether the exit was called in normal or query mode.

However, the user exit string provides the ability to pass additional parameters to the user exit. For example, the following trigger command passes two parameters and an error message to the user exit LOOKUP:

Notice that the user exit string is enclosed by single (not double) quotes.

```
USER EXIT('LOOKUP 2025 A', 'Lookup failed');
```

You can use this feature to pass field names to the user exit, as the following example shows:

```
USER EXIT('CONCAT firstname, lastname, address');
```

However, it is up to the user exit, not SQL\*Forms, to parse the user exit string.

## 20.8 About Returning Values to a Form

When a user exit returns control to SQL\*Forms, it must also return a code indicating whether it succeeded, failed, or suffered a fatal error. The return code is an integer constant defined by SQL\*Forms (see the next section). The three results have the following meanings:

Results	Meaning
success	The user exit encountered no errors. SQL*Forms proceeds to the success label or the next step, unless the Reverse Return Code switch is set by the calling trigger step.
failure	The user exit detected an error, such as an invalid value in a field. An optional message passed by the exit appears on the message line at the bottom of the SQL*Forms screen and on the Display Error screen. SQL*Forms responds as it does to a SQL statement that affects no rows.
fatal error	The user exit detected a condition that makes further processing impossible, such as an execution error in a SQL statement. An optional error message passed by the exit appears on the SQL*Forms Display Error screen. SQL*Forms responds as it does to a fatal SQL error. If a user exit changes the value of a field, then returns a failure or fatal error code, SQL*Forms does not discard the change. Nor does SQL*Forms discard changes when the Reverse Return Code switch is set and a success code is returned.

### 20.8.1 The IAP Constants

SQL\*Forms defines three symbolic constants for use as return codes. Depending on the host language, they are prefixed with IAP or SQL. For example, they might be IAPSUCC, IAPFAIL, and IAPFTL.

## 20.8.2 About Using WHENEVER

You can use the WHENEVER statement in an exit to detect invalid datatype conversions (SQLERROR), truncated values PUT into form fields (SQLWARNING), and queries that return no rows (NOT FOUND).

## 20.9 An Example of Using User Exits

The following example shows how a user exit that uses the EXEC TOOLS GET and PUT routines, as well as the EXEC TOOLS MESSAGE function, is coded.

```
int
myexit()
{
    char field1[20], field2[20], value1[20], value2[20];
    char result_value[20];
    char errmsg[80];
    int errlen;

#include sqlca.h
    EXEC SQL WHENEVER SQLERROR GOTO sql_error;
    /* get field values into form */
    EXEC TOOLS GET :field1, :field2 INTO :value1, :value2;
    /* manipulate the values to obtain result_val */
    ...
```

```
/* put result_val into form field result */
EXEC TOOLS PUT result VALUES (:result_val);
return IAPSUCC; /* trigger step succeeded */

sql_error:
    strcpy(errmsg, CONCAT("MYEXIT", sqlca.sqlerrm.sqlerrmc);
    errlen = strlen(errmsg);
    EXEC TOOLS MESSAGE :errmsg ; /* send error msg to Forms */
    return IAPFAIL;
```

## 20.10 About Precompiling and Compiling a User Exit

User exits are precompiled like standalone host programs. For instructions on compiling a user exit, see the Oracle installation or user's guide for your system.

#### **Related Topics**

Precompiler Options

## 20.11 Example Program: A User Exit

The following example shows a user exit.

```
Sample Program 5: SQL*Forms User Exit
This user exit concatenates form fields. To call the user
exit from a SQL*Forms trigger, use the syntax
  user exit('CONCAT field1, field2, ..., result field');
where user exit is a packaged procedure supplied with SQL*Forms
and CONCAT is the name of the user exit. A sample form named
CONCAT invokes the user exit.
\#define min(a, b) ((a < b) ? a : b)
#include <stdio.h>
#include <string.h>
/* Include the SQL Communications Area, a structure through which
* Oracle makes runtime status information such as error
* codes, warning flags, and diagnostic text available to the
* program.
#include <sqlca.h>
/* All host variables used in embedded SQL in this example
* appear in the Declare Section.
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR field[81];
   VARCHAR value[81];
   VARCHAR result[241];
EXEC SQL END DECLARE SECTION;
/* Define the user exit, called "concat". */
int concat(cmd, cmdlen, msg, msglen, query)
```

```
char *cmd;
              /* command line in trigger step ("CONCAT...") */
int *cmdlen; /* length of command line */
char *msg;
               /* trigger step failure message from form */
int *msglen; /* length of failure message */
               /* TRUE if invoked by post-query trigger,
int *query;
                  FALSE otherwise */
                           /* pointer to field list in
    char *cp = cmd + 7;
                              cmd string; 7 characters
                              are needed for "CONCAT " */
    char *fp = (char*)&field.arr[0]; /* pointer to a field name in
                                         cmd string */
    char errmsg[81];
                           /* message returned to SQL*Forms
                              on error */
    int
          errlen:
                            /* length of message returned
                              to SQL*Forms */
/* Branch to label sqlerror if an ORACLE error occurs. */
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    result.arr[0] = ' \setminus 0';
/* Parse field names from cmd string. */
    for (; *cp != '\0'; cp++)
       if (*cp != ',' && *cp != ' ')
           /* Copy a field name into field.arr from cmd. */
           *fp = *cp;
           fp++;
       else
           if (*cp == ' ')
              /* Have whole field name now. */
               *fp = ' \setminus 0';
               field.len = strlen((char *) field.arr);
               /* Get field value from form. */
               EXEC TOOLS GET : field INTO : value;
               value.arr[value.len] = '\0';
               strcat((char *) result.arr, (char *) value.arr);
               fp = (char *)&field.arr[0]; /* Reset field pointer. */
    }
/* Have last field name now. */
    *fp = ' \setminus 0';
    field.len = strlen((char *) field.arr);
    result.len = strlen((char *) result.arr);
/* Put result into form. */
    EXEC TOOLS PUT : field VALUES (:result);
/* Trigger step succeeded. */
    return (IAPSUCC);
sqlerror:
    strcpy(errmsg, "CONCAT: ");
    strncat(errmsg, sqlca.sqlerrm.sqlerrmc, min(72,
        sqlca.sqlerrm.sqlerrml));
    errlen = strlen(errmsg);
/* Pass error message to SQL*Forms status line. */
     EXEC TOOLS MESSAGE :errmsg ;
```

```
return(IAPFAIL); /* Trigger step failed. */
```

## 20.12 About Using the GENXTB Utility

The IAP program table IAPXTB in module IAPXIT contains an entry for each user exit linked into IAP. IAPXTB tells IAP the name, location, and host language of each user exit. When you add a new user exit to IAP, you must add a corresponding entry to IAPXTB. IAPXTB is derived from a database table, also named IAPXTB. You can modify the database table by running the GENXTB form on the operating system command line, as follows:

RUNFORM GENXTB username/password

A form is displayed for you to enter the following information for each user exit you define:

- Exit name
- C-language code
- Date created
- Date last modified
- Comments

After modifying the IAPXTB database table, use the GENXTB utility to read the table and create an Assembler or C source program that defines the module IAPXIT and the IAPXTB program table it contains. The source language used depends on your operating system. The syntax you use to run the GENXTB utility is

```
GENXTB username/password outfile
```

where outfile is the name you give the Assembler or C source program that GENXTB creates.

#### **Related Topics**

Guidelines

## 20.13 About Linking a User Exit into SQL\*Forms

Before running a form that calls a user exit, you must link the user exit into IAP, the SQL\*Forms component that runs a form. The user exit can be linked into your standard version of IAP or into a special version for those forms that call the exit.

To produce a new executable copy of IAP, link your user exit object module, the standard IAP modules, the IAPXIT module, and any modules needed from the Oracle and C link libraries.

The details of linking are system-dependent. Check the Oracle installation or user's guide for your system.

### 20.14 Guidelines

The guidelines in this section will help you avoid some common problems.

### 20.14.1 About Naming the Exit

The name of your user exit cannot be an Oracle reserved word. Also avoid using names that conflict with the names of SQL\*Forms commands, function codes, and externally defined names used by SQL\*Forms. The name of the user exit entry point in the source code becomes

the name of the user exit itself. The exit name must be a valid C function name, and a valid filename for your operating system.

SQL\*Forms converts the name of a user exit to upper case before searching for the exit. Therefore, the exit name must be in upper case in your source code.

## 20.14.2 About Connecting to Oracle

User exits communicate with Oracle using the connection made by SQL\*Forms. However, a user exit can establish additional connections to any database using Oracle Net Services.

#### **Related Topics**

Advanced Connection Options

### 20.14.3 About Issuing I/O Calls

File I/O is supported but screen I/O is not.

### 20.14.4 About Using Host Variables

Restrictions on the use of host variables in a standalone program also apply to user exits. Host variables must be prefixed with a colon in EXEC SQL and EXEC TOOLS statements. The use of host arrays is not allowed in EXEC TOOLS statements.

### 20.14.5 About Updating Tables

Generally, a user exit should not UPDATE database tables associated with a form. For example, suppose an operator updates a record in the SQL\*Forms work space, then a user exit UPDATEs the corresponding row in the associated database table. When the transaction is COMMITted, the record in the SQL\*Forms work space is applied to the table, overwriting the user exit UPDATE.

## 20.14.6 About Issuing Commands

Avoid issuing a COMMIT or ROLLBACK command from your user exit because Oracle will commit or roll back work begun by the SQL\*Forms operator, not just work done by the user exit. Instead, issue the COMMIT or ROLLBACK from the SQL\*Forms trigger. This also applies to data definition commands (such as ALTER, CREATE, and GRANT) because they issue an implicit COMMIT before and after executing.



# Part III

# **Appendixes**

This section contains the following appendixes:

- Reserved Words\_ Keywords\_ and Namespaces
- Performance Tuning
- · Syntactic and Semantic Checking
- System-Specific References
- Embedded SQL Statements and Directives
- Sample Programs
- Integrating Pro\*C/C++ into Microsoft Visual Studio .NET



## Reserved Words, Keywords, and Namespaces

This appendix contains the following topics:

- Reserved Words and Keywords
- Oracle Reserved Namespaces

## A.1 Reserved Words and Keywords

Some words are reserved by Oracle. That is, they have a special meaning to Oracle and cannot be redefined. For this reason, you cannot use them to name database objects such as columns, tables, or indexes. To view the lists of the Oracle reserved words for SQL and PL/SQL, see the *Oracle Database SQL Language Reference* and the *Oracle Database PL/SQL Language Reference*.

Pro\*C/C++ keywords, like C or C++ keywords, should not be used as variables in your program. Otherwise, an error will be generated. An error may result if they are used as the name of a database object such as a column. Here are the keywords used in Pro\*C/C++:

Keywords	Keywords	Keywords
all	allocate	alter
analyze	and	any
append	arraylen	as
asc	assign	at
audit	authorization	avg
begin	between	bind
both	break	buffer
buffering	by	call
cast	char	character
character_set_name	charf	charz
check	chunksize	close
collection	comment	commit
connect	constraint	constraints
context	continue	convbufsz
сору	count	create
current	currval	cursor
data	database	date
dateformat	datelang	datetime_interval_code
datetime_interval_precision	day	deallocate
dec	decimal	declare
default	define	delete
deref	desc	describe

Keywords	Keywords	Keywords
descriptor	directory	disable
display	distinct	do
double	drop	eject
else	enable	end
endif	end-exec	erase
escape	exec	execute
exists	explain	extract
fetch	file	fileexists
filename	first	float
flush	for	force
found	free	from
function	get	global
go	goto	grant
group	having	hold
host_stride_length	hour	iaf
identified	ifdef	ifndef
mmediate	in	include
ndicator	indicator_stride_length	input
nsert	integer	internal_length
ntersect	interval	into
s	isopen	istemporary
ast	leading	length
evel	like	list
oad	lob	local
lock	long	max
message	min	minus
minute	mode	month
multiset	name	national_character
nchar	nchar_cs	next
nextval	noaudit	not
notfound	nowait	null
nullable	number	numeric
nvarchar	nvarchar2	object
ocibfilelocator	ocibloblocator	ocicloblocator
ocidate	ociextproccontext	ocinumber
ociraw	ocirowid	ocistring
octet_length	of	one
only	open	option
or,	oracle	order
output	overlaps	overpunch



Keywords	Keywords	Keywords
package	partition	perform
precision	prepare	prior
procedure	put	raw
read	real	ref
reference	register	release
rename	replace	return
returned_length	returned_octet_length	returning
revoke	role	rollback
owid	rownum	savepoint
scale	second	section
select	set	skip1
skip2	skip3	smallint
some	sql	sql-context
sql-cursor	sqlerror	sqlwarning
start	statement	stddev
stop	string	sum
sysbackup	sysdba	sysoper
sysdate		
able	temporary	threads
ime	timestamp	timezone_hour
imezone_minute	to	tools
itle	trailing	transaction
rigger	trim	truncate
ype	uid	ulong_varchar
ınion	unique	unsigned
user_defined_type_name	user_defined_type_name_length	user_defined_type_schema
user_defined_type_schema_leng th	user_defined_type_version	update
use	user	using
validate	value	values
/ar	varchar	varchar2
variables	variance	varnum
varraw varraw	view	whenever
where	with	work
write	year	zone

## A.2 Oracle Reserved Namespaces

The following table 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, do not use function names that begin with these

characters. For example, the Oracle Net Transparent Network Service functions all begin with the characters  ${\tt NS}$ , so you need to avoid naming functions that begin with  ${\tt NS}$ .

**Table A-1 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	Oracle Net Native services product
NC	Oracle Net RPC project
ND	Oracle Net Directory
NL	Oracle Net Network Library layer
NM	Oracle Net Net Management Project
NR	Oracle Net Interchange
NS	Oracle Net Transparent Network Service
NT	Oracle Net Drivers
NZ	Oracle Net Security Service
OSN	Oracle Net V1
TTC	Oracle Net Two task
GEN, L, ORA	Core library functions
LI, LM, LX	function names from the Oracle Globalization Support layer
S	function names from system-dependent libraries

The list in the table 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.



B

## Performance Tuning

This appendix shows you some simple, easy-to-apply methods for improving the performance of your applications. Using these methods, you can often reduce processing time by 25% or more. This appendix contains the following topics:

- What Causes Poor Performance?
- How Can Performance Be Improved?
- About Using Host Arrays
- About Using Embedded PL/SQL
- About Optimizing SQL Statements
- About Statement Caching
- About Using Indexes
- About Taking Advantage of Row-Level Locking
- · About Eliminating Unnecessary Parsing
- About Using Connection Pooling
- About Using Oracle Connection Manager in Traffic Director Mode

### **B.1 What Causes Poor Performance?**

One cause of poor performance is high communications overhead. The server must process SQL statements one at a time. Thus, each statement results in another call to single and higher overhead. In a networked environment, SQL statements must be sent over the network, adding to network traffic. Heavy network traffic can slow down your application significantly.

Another cause of poor performance is inefficient SQL statements. Because SQL is so flexible, you can get the same result with two different statements, but one statement might be less efficient. For example, the following two SELECT statements return the same rows (the name and number of every department having at least one employee):

```
EXEC SQL SELECT dname, deptno
    FROM dept
    WHERE deptno IN (SELECT deptno FROM emp);

EXEC SQL SELECT dname, deptno
    FROM dept
    WHERE EXISTS
    (SELECT deptno FROM emp WHERE dept.deptno = emp.deptno);
```

However, the first statement is slower because it does a time-consuming full scan of the EMP table for every department number in the DEPT table. Even if the DEPTNO column in EMP is indexed, the index is not used because the subquery lacks a WHERE clause naming DEPTNO.

A third cause of poor performance is unnecessary parsing and binding. Recall that before executing a SQL statement, the server must parse and bind it. Parsing means examining the SQL statement to make sure it follows syntax rules and refers to valid database objects.

Binding means associating host variables in the SQL statement with their addresses so that the server can read or write their values.

Many applications manage cursors poorly. This results in unnecessary parsing and binding, which adds noticeably to processing overhead.

## **B.2 How Can Performance Be Improved?**

If you are unhappy with the performance of your precompiled programs, there are several ways you can reduce overhead.

You can greatly reduce communications overhead, especially in networked environments, by

- Using host arrays
- Using embedded PL/SQL

You can reduce processing overhead—sometimes dramatically—by

- Optimizing SQL statements
- Using indexes
- Taking advantage of row-level locking
- Eliminating unnecessary parsing
- Avoiding unnecessary reparsing

The following sections look at each of these ways to cut overhead.

# **B.3 About Using Host Arrays**

Host arrays can increase performance because they let you manipulate an entire collection of data with a single SQL statement. For example, suppose you want to INSERT salaries for 300 employees into the EMP table. Without arrays your program must do 300 individual INSERTs—one for each employee. With arrays, only one INSERT is necessary. Consider the following statement:

```
EXEC SQL INSERT INTO emp (sal) VALUES (:salary);
```

If salary is a simple host variable, the server executes the INSERT statement once, inserting a single row into the EMP table. In that row, the SAL column has the value of salary. To insert 300 rows this way, you must execute the INSERT statement 300 times.

However, if *salary* is a host array of size 300, all 300 rows are inserted into the EMP table at once. In each row, the SAL column has the value of an element in the *salary* array.

#### **Related Topics**

Host Arrays

## B.4 About Using Embedded PL/SQL

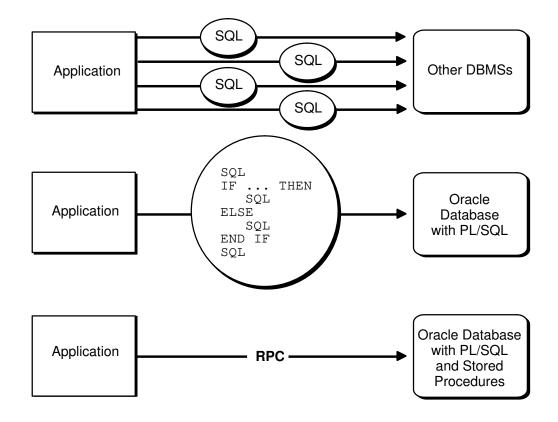
As Figure B-1 shows, if your application is database-intensive, you can use control structures to group SQL statements in a PL/SQL block, then send the entire block to the database server. This can drastically reduce communication between your application and the database server.

Also, you can use PL/SQL subprograms to reduce calls from your application to the server. For example, to execute ten individual SQL statements, ten calls are required, but to execute a subprogram containing ten SQL statements, only one call is required.

PL/SOL Ingrange Portermones

Figure B-1 PL/SQL Boosts Performance

#### PL/SQL Increases Performance Especially in Networked Environments



PL/SQL can also cooperate with Oracle application development tools such as Oracle Forms. By adding procedural processing power to these tools, PL/SQL boosts performance. Using PL/SQL, a tool can do any computation quickly and efficiently without calling on the database server. This saves time and reduces network traffic.

#### **Related Topics**

- Embedded PL/SQL
- Oracle Database PL/SQL Language Reference

# **B.5 About Optimizing SQL Statements**

For every SQL statement, the Oracle optimizer generates an *execution plan*, which is a series of steps that the server takes to execute the statement. These steps are determined by rules given in Oracle Application Developer's Guide - Fundamentals. Following these rules will help you write optimal SQL statements.

## **B.5.1 Optimizer Hints**

In some cases, you can suggest to the server the way to optimize a SQL statement. These suggestions, called *hints*, let you influence decisions made by the optimizer.

Hints are not directives; they merely help the optimizer do its job. Some hints limit the scope of information used to optimize a SQL statement, while others suggest overall strategies.

You can use hints to specify the

- Optimization approach for a SQL statement
- Access path for each referenced table
- Join order for a join
- Method used to join tables

Hence, hints fall into the following four categories:

- Optimization Approach
- Access Path
- Join Order
- Join Operation

For example, the two optimization approach hints, COST and NOCOST, invoke the cost-based optimizer and the rule-based optimizer, respectively.

You give hints to the optimizer by placing them in a C-style comment immediately after the verb in a SELECT, UPDATE, INSERT, or DELETE statement. For instance, the optimizer uses the cost-based approach for the following statement:

```
SELECT /*+ COST */ ename, sal INTO ...
```

For C++ code, optimizer hints in the form //+ are also recognized.

#### **Related Topics**

Using Edition-Based Redefinition

## **B.5.2 Trace Facility**

You can use the SQL trace facility and the EXPLAIN PLAN statement to identify SQL statements that might be slowing down your application.

The SQL trace facility generates statistics for every SQL statement executed. From these statistics, you can determine which statements take the most time to process. Then, you can concentrate your tuning efforts on those statements.

The EXPLAIN PLAN statement shows the execution plan for each SQL statement in your application. An *execution plan* describes the database operations that must be carried out to execute a SQL statement. You can use the execution plan to identify inefficient SQL statements.

#### **Related Topics**

Tools for Performance

# **B.6 About Statement Caching**

This is a feature that will help in the performance improvement of all the precompiler applications that rely on dynamic SQL statements. The new implementation will remove the overhead of parsing the dynamic statements on reuse. The precompiler application user can obtain this performance improvement using a new command line option (for the statement cache size), which will enable the statement caching of the dynamic statements. By enabling



the new option, the statement cache will be created at session creation time. The caching is only applicable for the dynamic statements and the cursor cache for the static statements coexists with this feature.

## **B.7 About Using Indexes**

Using ROWIDs, an *index* associates each distinct value in a table column with the rows containing that value. An index is created with the CREATE INDEX statement. For details, see CREATE INDEX.

You can use indexes to boost the performance of queries that return less than 15% of the rows in a table. A query that returns 15% or more of the rows in a table is executed faster by a *full scan*, that is, by reading all rows sequentially.

Any query that names an indexed column in its WHERE clause can use the index. For guidelines that help you choose which columns to index, see *Oracle Database Advanced Application Developer's Guide*.

# **B.8 About Taking Advantage of Row-Level Locking**

By default, data is locked at the row level rather than at the table level. Row-level locking allows multiple users to access different rows in the same table concurrently. The resulting performance gain is significant.

You can specify table-level locking, but it lessens the effectiveness of the transaction processing option.

Applications that do online transaction processing benefit most from row-level locking. If your application relies on table-level locking, modify it to take advantage of row-level locking. In general, avoid explicit table-level locking.

#### **Related Topics**

Using LOCK TABLE

# **B.9 About Eliminating Unnecessary Parsing**

Eliminating unnecessary parsing requires correct handling of cursors and selective use of the following cursor management options:

- MAXOPENCURSORS
- HOLD\_CURSOR
- RELEASE CURSOR

These options affect implicit and explicit cursors, the cursor cache, and private SQL areas.

## **B.9.1 About Handling Explicit Cursors**

Recall that there are two types of cursors: implicit and explicit. A cursor is implicitly declared for all data definition and data manipulation statements. However, for queries that return more than one row, you must explicitly declare a cursor (or use host arrays). You use the DECLARE CURSOR statement to declare an explicit cursor. The way you handle the opening and closing of explicit cursors affects performance.

If you need to reevaluate the active set, simply reOPEN the cursor. OPEN will use any new host-variable values. You can save processing time if you do not CLOSE the cursor first.





To make performance tuning easier, you can reOPEN an already open cursor. However, this is an ANSI extension. So, when MODE=ANSI, you must CLOSE a cursor before reOPENing it.

Only CLOSE a cursor when you want to free the resources (memory and locks) acquired by OPENing the cursor. For example, your program should CLOSE all cursors before exiting.

#### B.9.1.1 Cursor Control

In general, there are three ways to control an explicitly declared cursor:

- Use DECLARE, OPEN, and CLOSE.
- Use PREPARE, DECLARE, OPEN, and CLOSE.
- COMMIT closes the cursor when MODE=ANSI.

With the first way, beware of unnecessary parsing. OPEN does the parsing, but only if the parsed statement is unavailable because the cursor was CLOSEd or never OPENed. Your program should DECLARE the cursor, reOPEN it every time the value of a host variable changes, and CLOSE it only when the SQL statement is no longer needed.

With the second way (for dynamic SQL Methods 3 and 4), PREPARE does the parsing, and the parsed statement is available until a CLOSE is executed. Your program should do the following:

- PREPARE the SQL statement
- DECLARE the cursor
- Again OPEN the cursor every time the value of a host variable changes
- Again PREPARE the SQL statement
- Again OPEN the cursor if the SQL statement changes
- CLOSE the cursor only when the SQL statement is no longer needed.

When possible, avoid placing OPEN and CLOSE statements in a loop; this is a potential cause of unnecessary reparsing of the SQL statement. In the next example, both the OPEN and CLOSE statements are inside the outer *while* loop. When MODE=ANSI, the CLOSE statement must be positioned as shown, because ANSI requires a cursor to be CLOSEd before being reOPENed.

With MODE=ORACLE, however, a CLOSE statement can execute without the cursor being OPENed. By placing the CLOSE statement outside the outer *while* loop, you can avoid possible reparsing at each iteration of the OPEN statement.

```
while (salary < 5000)
{
    EXEC SQL OPEN emp_cursor;
    while (sqlca.sqlcode==0)
    {
        EXEC SQL FETCH emp_cursor INTO ....
    }
    salary += 1000;
}
EXEC SQL CLOSE emp cursor;</pre>
```

## B.9.2 About Using the Cursor Management Options

A SQL statement need be parsed only once unless you change its makeup. For example, you change the makeup of a query by adding a column to its select list or WHERE clause. The HOLD\_CURSOR, RELEASE\_CURSOR, and MAXOPENCURSORS options give you some control over how the server manages the parsing and reparsing of SQL statements. Declaring an explicit cursor gives you maximum control over parsing.

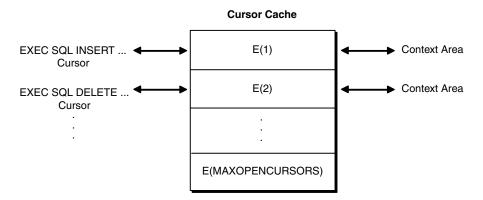
## B.9.2.1 SQL Areas and Cursor Cache

When a data manipulation statement is executed, its associated cursor is linked to an entry in the Pro\*C/C++ cursor cache. The cursor cache is a continuously updated area of memory used for cursor management. The cursor cache entry is in turn linked to a private SQL area.

The private SQL area, a work area created dynamically at run time, contains the addresses of host variables, and other information needed to process the statement. An explicit cursor lets you name a SQL statement, access the information in its private SQL area, and, to some extent, control its processing.

Figure B-2 represents the cursor cache after your program has done an INSERT and a DELETE.

Figure B-2 Cursors Linked using the Cursor Cache





#### B.9.2.2 Resource Use

The maximum number of open cursors for each user session is set by the initialization parameter OPEN\_CURSORS.

MAXOPENCURSORS specifies the *initial* size of the cursor cache. If a new cursor is needed and there are no free cache entries, the server tries to reuse an entry. Its success depends on the values of HOLD\_CURSOR and RELEASE\_CURSOR and, for explicit cursors, on the status of the cursor itself.

If the value of MAXOPENCURSORS is less than the number of cache entries actually needed, the server uses the first cache entry marked as reusable. For example, suppose an INSERT statement's cache entry E(1) is marked as reusable, and the number of cache entries already equals MAXOPENCURSORS. If the program executes a new statement, cache entry E(1) and its private SQL area might be reassigned to the new statement. To reexecute the INSERT statement, the server would have to reparse it and reassign another cache entry.

The server allocates an additional cache entry if it cannot find one to reuse. For example, if MAXOPENCURSORS=8 and all eight entries are active, a ninth is created. If necessary, the server keeps allocating additional cache entries until it runs out of memory or reaches the limit set by OPEN\_CURSORS. This dynamic allocation adds to processing overhead.

Thus, specifying a low value for MAXOPENCURSORS saves memory but causes potentially expensive dynamic allocations and deallocations of new cache entries. Specifying a high value for MAXOPENCURSORS assures speedy execution but uses more memory.

## **B.9.2.3 Infrequent Execution**

Sometimes, the link between an *infrequently* executed SQL statement and its private SQL area should be temporary.

When HOLD\_CURSOR=NO (the default), after the server executes the SQL statement and the cursor is closed, the precompiler marks the link between the cursor and cursor cache as reusable. The link is reused as soon as the cursor cache entry to which it points is needed for another SQL statement. This frees memory allocated to the private SQL area and releases parse locks. However, because a PREPAREd cursor must remain active, its link is maintained even when HOLD CURSOR=NO.

When RELEASE\_CURSOR=YES, after the server executes the SQL statement and the cursor is closed, the private SQL area is automatically freed and the parsed statement lost. This might be necessary if, for example, MAXOPENCURSORS is set low at your site to conserve memory.

If a data manipulation statement precedes a data definition statement and they reference the same tables, specify RELEASE\_CURSOR=YES for the data manipulation statement. This avoids a conflict between the parse lock obtained by the data manipulation statement and the exclusive lock required by the data definition statement.

When RELEASE\_CURSOR=YES, the link between the private SQL area and the cache entry is immediately removed and the private SQL area freed. Even if you specify HOLD\_CURSOR=YES, the server must still reallocate memory for a private SQL area and reparse the SQL statement before executing it because RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES.

However, when RELEASE\_CURSOR=YES, the reparse might still require no extra processing because the server caches the parsed representations of SQL statements and PL/SQL blocks in its *Shared SQL Cache*. Even if its cursor is closed, the parsed representation remains available until it is aged out of the cache.



### **B.9.2.4 Frequent Execution**

The links between a *frequently* executed SQL statement and its private SQL area should be maintained because the private SQL area contains all the information needed to execute the statement. Maintaining access to this information makes subsequent execution of the statement much faster.

When HOLD\_CURSOR=YES, the link between the cursor and cursor cache is maintained after the server executes the SQL statement. Thus, the parsed statement and allocated memory remain available. This is useful for SQL statements that you want to keep active because it avoids unnecessary reparsing.

When RELEASE\_CURSOR=NO (the default), the link between the cache entry and the private SQL area is maintained after the server executes the SQL statement and is not reused unless the number of open cursors exceeds the value of MAXOPENCURSORS. This is useful for SQL statements that are executed often because the parsed statement and allocated memory remain available.



With versions of Oracle prior to Oracle8*i*, when RELEASE\_CURSOR=NO and HOLD\_CURSOR=YES, after the server executes a SQL statement, its parsed representation remains available. But with later versions of Oracle, when RELEASE\_CURSOR=NO and HOLD\_CURSOR=YES, the parsed representation remains available only until it is aged out of the Shared SQL Cache. Normally, this is not a problem, but you might get unexpected results if the definition of a referenced object changes before the SQL statement is reparsed.

## B.9.2.5 Embedded PL/SQL Considerations

For the purposes of cursor management, an embedded PL/SQL block is treated just like a SQL statement. When an embedded PL/SQL block is executed, a parent cursor is associated with the entire block and a link is created between the cache entry and the private SQL area in the PGA for the embedded PL/SQL block. Be aware that each SQL statement inside the embedded block also requires a private SQL area in the PGA. These SQL statements use child cursors that PL/SQL manages itself. The disposition of the child cursors is determined through its associated parent cursor. That is, the private SQL areas used by the child cursors are freed after the private SQL area for its parent cursor is freed.

#### **B.9.2.6 Parameter Interactions**

Figure B-1 shows how HOLD\_CURSOR and RELEASE\_CURSOR interact. Notice that HOLD\_CURSOR=NO overrides RELEASE\_CURSOR=NO and that RELEASE\_CURSOR=YES overrides HOLD\_CURSOR=YES.

Table B-1 HOLD CURSOR and RELEASE CURSOR Interactions

HOLD_CURSOR	RELEASE_CURSOR	Links are
NO	NO	marked as reusable
YES	NO	maintained



Table B-1 (Cont.) HOLD\_CURSOR and RELEASE \_CURSOR Interactions

HOLD_CURSOR	RELEASE_CURSOR	Links are
NO	YES	removed immediately
YES	YES	removed immediately

# **B.10 About Avoiding Unnecessary Reparsing**

Avoiding unnecessary reparsing involves eliminating errors encountered during the execute phase of an SQL statement in a loop. When an embedded SQL statement is executed in a loop, the SQL statement is parsed only once. However, if the execution of the SQL statement results in an error, the SQL statement is usually reparsed. In this case, reparsing will occur for all errors encountered, except the following:

- ORA-1403 (not found)
- ORA-1405 (truncation)
- ORA-1406 (null value)

By eliminating all other errors, you can avoid unnecessary reparsing.

# **B.11 About Using Connection Pooling**

This section describes performance tuning using connection pooling. If an application is multithreaded and is performing concurrent operations on the same database, users can use the connection pooling feature to achieve better performance. Users can tune an application's performance by choosing the appropriate values for parameters used with connection pooling, and can achieve up to three times performance increase, when compared to existing application performance.

#### **Related Topics**

- Multithreaded Applications
- Connection Pooling
- Oracle Call Interface Programmer's Guide

# B.12 About Using Oracle Connection Manager in Traffic Director Mode

Oracle Connection Manager in Traffic Director Mode is a proxy that is placed between supported database clients and database instances.

#### **Note:**

A multitenant container database is the only supported architecture in Oracle Database 21c. While the documentation is being revised, legacy terminology may persist. In most cases, "database" and "non-CDB" refer to a CDB or PDB, depending on context. In some contexts, such as upgrades, "non-CDB" refers to a non-CDB from a previous release.

Supported clients from Oracle Database 11g Release 2 (11.2) and later can connect to Oracle Connection Manager in Traffic Director Mode. Oracle Connection Manager in Traffic Director Mode provides improved high availability (HA) for planned and unplanned database server outages, connection multiplexing support, and load balancing. Support for Oracle Connection Manager in Traffic Director Mode is described in more detail in the following sections

- Modes of Operation
- Key Features

#### **Modes of Operation**

Oracle Connection Manager in Traffic Director Mode supports the following modes of operation:

- In pooled connection mode, Oracle Connection Manager in Traffic Director Mode supports any application using the following database client releases:
  - OCI, OCCI, and Open Source Drivers (Oracle Database 11g release 2 (11.2.0.4) and later))
  - JDBC (Oracle Database 12c release 1 (12.1) and later)
  - ODP.NET (Oracle Database 12c release 2 (12.2) and later)

In addition, applications must use DRCP. That is, the application must enable DRCP in the connect string (or in the tnsnames.ora alias).

 In non-pooled connection (or dedicated) mode, Oracle Connection Manager in Traffic Director Mode supports any application using database client releases Oracle Database 11g release 2 (11.2.0.4) and later. In this mode, some capabilities, such as connection multiplexing are not available.

#### **Key Features**

Oracle Connection Manager in Traffic Director Mode furnishes support for the following:

- Transparent performance enhancements and connection multiplexing, which includes:
  - Statement caching, rows prefetching, and result set caching are auto-enabled for all modes of operation.
  - Database session multiplexing (pooled mode only) using the proxy resident connection pool (PRCP), where PRCP is a proxy mode of Database Resident Connection Pooling (DRCP). Applications get transparent connect-time load balancing and run-time load balancing between Oracle Connection Manager in Traffic Director Mode and the database.
  - For multiple Oracle Connection Manager in Traffic Director Mode instances, applications get increased scalability through client-side connect time load balancing or with a load balancer (BIG-IP, NGINX, and others)
- Zero application downtime



- Planned database maintenance or pluggable database (PDB) relocation
  - \* Pooled mode

Oracle Connection Manager in Traffic Director Mode responds to Oracle Notification Service (ONS) events for planned outages and redirects work. Connections are drained from the pool on Oracle Connection Manager in Traffic Director Mode when the request completes. Service relocation is supported for Oracle Database 11*g* release 2 (11.2.0.4) and later.

For PDB relocation, Oracle Connection Manager in Traffic Director Mode responds to in-band notifications when a PDB is relocated, that is even when ONS is not configured (for Oracle Database release 18c, version 18.1 and later server only)

Non-pooled or dedicated mode

When there is no request boundary information from the client, Oracle Connection Manager in Traffic Director Mode supports planned outage for many applications (as long as only simple session state and cursor state need to be preserved across the request/transaction boundaries). This support includes:

- \* Stop service/PDB at the transaction boundary or it leverages Oracle Database release 18c continuous application availability to stop the service at the request boundary
- \* Oracle Connection Manager in Traffic Director Mode leverages Transparent Application Failover (TAF) failover restore to reconnect and restore simple states.
- Unplanned database outages for read-mostly workloads
- High Availability of Oracle Connection Manager in Traffic Director Mode to avoid a single point of failure. This is supported by:
  - Multiple instances of Oracle Connection Manager in Traffic Director Mode using a load balancer or client side load balancing/failover in the connect string
  - Rolling upgrade of Oracle Connection Manager in Traffic Director Mode instances
  - Graceful close of existing connections from client to Oracle Connection Manager in Traffic Director Mode for planned outages
  - In-band notifications to Oracle Database release 18c and later clients
  - For older clients, notifications are sent with the response of the current request
- For security and isolation, Oracle Connection Manager in Traffic Director Mode furnishes:
  - Database Proxy supporting transmission control protocol/transmission control protocol secure (TCP/TCPS) and protocol conversion
  - Firewall based on the IP address, service name, and secure socket layer/transport layer security (SSL/TLS) wallets
  - Tenant isolation in a multi-tenant environment
  - Protection against denial-of-service and fuzzing attacks
  - Secure tunneling of database traffic across Oracle Database on-premises and Oracle Cloud



#### ✓ See Also:

- Oracle Database Net Services Administrator's Guide for information about configuring cman.ora configuration file to set up Oracle Connection Manager in Traffic Director Mode
- Oracle Database Net Services Administrator's Guide for information about configuring databases for Oracle Connection Manager in Traffic Director Mode proxy authentication
- Oracle Database Net Services Administrator's Guide for information about configuring Oracle Connection Manager in Traffic Director Mode for unplanned down events
- Oracle Database Net Services Administrator's Guide for information about configuring Oracle Connection Manager in Traffic Director Mode for planned down events
- Oracle Database Net Services Administrator's Guide for information about configuring proxy resident connection pools for use by Oracle Connection Manager in Traffic Director Mode
- Oracle Database Net Services Administrator's Guide for information about functionality not supported for all drivers with Oracle Connection Manager in Traffic Director Mode
- Oracle Database Net Services Reference for an overview of Oracle CMAN configuration file



C

# Syntactic and Semantic Checking

By checking the syntax and semantics of embedded SQL statements and PL/SQL blocks, the Pro\*C/C++ Precompiler helps you quickly find and fix coding mistakes. This appendix shows you how to use the SQLCHECK option to control the type and extent of checking. This appendix contains the following topics:

- What Is Syntactic and Semantic Checking?
   About Controlling the Type and Extent of Checking
- About Specifying SQLCHECK=SEMANTICS
- About Specifying SQLCHECK=SYNTAX
- About Entering the SQLCHECK Option

# C.1 What Is Syntactic and Semantic Checking?

Rules of syntax specify how language elements are sequenced to form valid statements. Thus, syntactic checking verifies that keywords, object names, operators, delimiters, and so on are placed correctly in your SQL statement. For example, the following embedded SQL statements contain syntax errors:

```
EXEC SQL DELETE FROM EMP WHER DEPTNO = 20;
-- misspelled keyword WHERE
EXEC SQL INSERT INTO EMP COMM, SAL VALUES (NULL, 1500);
-- missing parentheses around column names COMM and SAL
```

Rules of semantics specify how valid external references are made. Thus, *semantic checking* verifies that references to database objects and host variables are valid and that host variable datatypes are correct. For example, the following embedded SQL statements contain semantic errors:

```
EXEC SQL DELETE FROM empp WHERE deptno = 20;
    -- nonexistent table, EMPP
EXEC SQL SELECT * FROM emp WHERE ename = :emp_name;
    -- undeclared host variable, emp name
```

The rules of SQL syntax and semantics are defined in *Oracle Database SQL Language Reference*.

# C.2 About Controlling the Type and Extent of Checking

You control the type and extent of checking by specifying the SQLCHECK option on the command line. With SQLCHECK, the *type* of checking can be syntactic, semantic, or both. The *extent* of checking can include the following:

- Data definition statements (such as CREATE and GRANT)
- Data manipulation statements (such as SELECT and INSERT)
- PL/SQL blocks

However, SQLCHECK cannot check dynamic SQL statements because they are not fully defined until run time.

You can specify the following values for SQLCHECK:

- SEMANTICS or FULL
- SYNTAX

The default value is SYNTAX.

The use of SQLCHECK does not affect the normal syntax checking done on data control, cursor control, and dynamic SQL statements.

## C.3 About Specifying SQLCHECK=SEMANTICS

When SQLCHECK=SEMANTICS, the precompiler checks the syntax and semantics of

- Data manipulation statements (INSERT, UPDATE, and so on)
- PL/SQL blocks
- Host variable datatypes

as well as the syntax of

Data definition statements (CREATE, ALTER, and so on)

However, only syntactic checking is done on data manipulation statements that use the AT *db\_name* clause.

When SQLCHECK=SEMANTICS, the precompiler gets information needed for a semantic check by using embedded DECLARE TABLE statements or if you specify the USERID option on the command line, by connecting to the Oracle server and accessing the data dictionary. You need not connect to the Oracle server if every table referenced in a data manipulation statement or PL/SQL block is defined in a DECLARE TABLE statement.

If you connect to the Oracle server, but some needed information cannot be found in the data dictionary, you must use DECLARE TABLE statements to supply the missing information. A DECLARE TABLE definition overrides a data dictionary definition if they conflict.

If you embed PL/SQL blocks in a host program, you must specify SQLCHECK=SEMANTICS.

When checking data manipulation statements, the precompiler uses the syntax rules found in the *Oracle Database SQL Language Reference*, but uses a stricter set of semantic rules. In particular, stricter datatype checking is done. As a result, existing applications written for earlier versions of Oracle might not precompile successfully when SQLCHECK=SEMANTICS.

Specify SQLCHECK=SEMANTICS when you precompile new programs or want stricter datatype checking.

## C.3.1 About Enabling a Semantic Check

When SQLCHECK=SEMANTICS, the precompiler can get information needed for a semantic check in either of the following ways:

- Connect to the Oracle server and access the data dictionary.
- Use embedded DECLARE TABLE and DECLARE TYPE statements.



## C.3.1.1 About Connecting to the Oracle server

To do a semantic check, the precompiler can connect to an Oracle database that maintains definitions of tables, types, and views referenced in your host program.

After connecting to the Oracle server, the precompiler accesses the data dictionary for needed information. The *data dictionary* stores table and column names, table and column constraints, column lengths, column datatypes, and so on.

If some of the needed information cannot be found in the data dictionary (because your program refers to a table not yet created, for example), you must supply the missing information using the DECLARE TABLE statement (discussed later in this appendix).

To connect to the Oracle server, specify the USERID option on the command line, using the syntax:

```
USERID=username/password
```

where *username* and *password* comprise a valid Oracle userid. If you omit the password, you are prompted for it.

If, instead of a username and password, you specify

```
USERID=/
```

the precompiler attempts to automatically connect to the Oracle server. The attempt succeeds only if an existing Oracle username matches your operating system ID prefixed with "CLUSTER\$", or whatever value the parameter OS\_AUTHENT\_PREFIX is set to in the INIT.ORA file. For example, if your operating system ID is MBLAKE, an automatic connect only succeeds if CLUSTER\$MBLAKE is a valid Oracle username.

If you omit the USERID option, the precompiler must get needed information from embedded DECLARE TABLE statements.

If you try connecting to the Oracle server but cannot (because the database is unavailable, for example), an error message is issued and your program is not precompiled.

## C.3.1.2 About Using DECLARE TABLE

The precompiler can do a semantic check without connecting to the Oracle server. To do the check, the precompiler must get information about tables and views from embedded DECLARE TABLE statements. Thus, every table referenced in a data manipulation statement or PL/SQL block must be defined in a DECLARE TABLE statement.

The syntax of the DECLARE TABLE statement is:

```
EXEC SQL DECLARE table_name TABLE
    (col name col datatype [DEFAULT expr] [NULL|NOT NULL], ...);
```

where *expr* is an integer that can be used as a default column value in the CREATE TABLE statement.

For user-defined object datatypes, the size is optional because it is not used.

If you use DECLARE TABLE to define a database table that already exists, the precompiler uses your definition, ignoring the one in the data dictionary.



## C.3.1.3 About Using DECLARE TYPE

Similarly, for TYPE, there is a DECLARE TYPE statement whose syntax is:

```
EXEC SQL DECLARE type TYPE
[AS OBJECT (col_name col_datatype, ...)] |
[AS VARRAY(size) OF element_type ] |
[AS TABLE OF object type ];
```

This allows for better type-checking for user-defined types when SQLCHECK=SEMANTICS at precompile-time. When SQLCHECK=SYNTAX, the DECLARE TYPE statements serve as documentation only and are commented out and ignored.

# C.4 About Specifying SQLCHECK=SYNTAX

When SQLCHECK=SYNTAX, the precompiler checks the syntax of SQL statements documented in *Oracle Database SQL Language Reference*:

- Data manipulation statements
- Host-variable expressions

No semantic check is done, and the following restrictions apply:

- No connection to the Oracle server is attempted and USERID becomes an invalid option. If you specify USERID, a warning message is issued.
- DECLARE TABLE and DECLARE TYPE statements are ignored; they serve only as documentation.
- PL/SQL blocks are not allowed. If the precompiler finds a PL/SQL block, an error message is issued.

When checking data manipulation statements, the precompiler uses Oracle syntax rules. These rules are downwardly compatible, so specify SQLCHECK=SYNTAX when migrating your precompiled programs.

## C.5 About Entering the SQLCHECK Option

You can enter the SQLCHECK option inline or on the command line. However, the level of checking you specify inline cannot be higher than the level you specify (or accept by default) on the command line. For example, if you specify SQLCHECK=SYNTAX on the command line, you cannot specify SQLCHECK=SEMANTICS inline.



D

# System-Specific References

This appendix groups together in one place all references in this guide to system-specific information.

This appendix contains this section:

System-Specific Information

# D.1 System-Specific Information

System-specific information is described in the appropriate Oracle system-specific documentation for your platform.

### D.1.1 Location of Standard Header Files

The location of the standard Pro\*C/C++ header files—sqlca.h, oraca.h, and sqlda.h—is system specific. For other your system, see your Oracle system-specific documentation.

## D.1.2 About Specifying Location of Included Files for the C Compiler

When you use the Pro\*C/C++ command-line option INCLUDE= to specify the location of a non-standard file to be included, you should also specify the same location for the C compiler. The way you do this is system specific. See "Include Files".

## D.1.3 ANSI C Support

Use the CODE= option to make the C code that Pro\*C/C++ generates compatible with your system's C compiler. See "Function Prototyping".

## D.1.4 Struct Component Alignment

C compilers vary in the way they align struct components, usually depending on the system hardware. Use the sqlvcp() function to determine the padding added to the .arr component of a VARCHAR struct. See the section "Find the Length of the VARCHAR Array Component".

## D.1.5 Size of an Integer and ROWID

The size in bytes of integer datatypes and the binary external size of ROWID datatypes are system dependent. See "INTEGER" and "ROWID".

## D.1.6 Byte Ordering

The order of bytes in a word is platform dependent. See the section "UNSIGNED".

## D.1.7 About Connecting to the Oracle Server

Connecting to the Oracle server using the Oracle Net drivers involves system-specific network protocols. See the section "Interface to OCI Release 8" for more details.

## D.1.8 About Linking in an XA Library

You link in your XA library in a system-dependent way. See the section "Linking", and your Oracle installation or user's guides, for more information.

### D.1.9 Location of the Pro\*C/C++ Executable

The location of the Pro\*C/C++ Precompiler is system specific. See the section "The Precompiler Command", and your installation or user's guides, for more information.

## D.1.10 System Configuration File

Each precompiler installation has a system configuration file. This file is not shipped with the precompiler; it must be created by the system administrator. The location (directory path) which Pro\*C/C++ searches for the system configuration file is system dependent. See the section "What Occurs During Precompilation?" for more information.

## D.1.11 INCLUDE Option Syntax

The syntax for the value of the INCLUDE command-line option is system specific. See "INCLUDE".

## D.1.12 About Compiling and Linking

Compiling and linking your Pro\*C/C++ output to get an executable application is always system dependent. See the section "Compile and Link", and the following sections, for additional information.

## D.1.13 User Exits

Compiling and linking Oracle Forms user exits is system specific. See User Exits.



F

# **Embedded SQL Statements and Directives**

This appendix contains descriptions of both the SQL standard embedded statements and directives, as well as the Oracle embedded SQL extensions.



Only statements which differ in syntax from non-embedded SQL are described in this appendix. For details of the non-embedded SQL statements, see Types of SQL Statements

#### This appendix contains the following topics:

- Summary of Precompiler Directives and Embedded SQL Statements
- About The Statement Descriptions
- How to Read Syntax Diagrams
- ALLOCATE (Executable Embedded SQL Extension)
- ALLOCATE DESCRIPTOR (Executable Embedded SQL)
- CACHE FREE ALL (Executable Embedded SQL Extension)
- CALL (Executable Embedded SQL)
- CLOSE (Executable Embedded SQL)
- COLLECTION APPEND (Executable Embedded SQL Extension)
- COLLECTION DESCRIBE (Executable Embedded SQL Extension)
- COLLECTION GET (Executable Embedded SQL Extension)
- COLLECTION RESET (Executable Embedded SQL Extension)
- COLLECTION SET (Executable Embedded SQL Extension)
- COLLECTION TRIM (Executable Embedded SQL Extension)
- COMMIT (Executable Embedded SQL)
- CONNECT (Executable Embedded SQL Extension)
- CONTEXT ALLOCATE (Executable Embedded SQL Extension)
- CONTEXT FREE (Executable Embedded SQL Extension)
- CONTEXT OBJECT OPTION GET (Executable Embedded SQL Extension)
- CONTEXT OBJECT OPTION SET (Executable Embedded SQL Ext)
- CONTEXT USE (Oracle Embedded SQL Directive)
- DEALLOCATE DESCRIPTOR (Embedded SQL Statement)
- DECLARE CURSOR (Embedded SQL Directive)
- DECLARE DATABASE (Oracle Embedded SQL Directive)



- DECLARE STATEMENT (Embedded SQL Directive)
- DECLARE TABLE (Oracle Embedded SQL Directive)
- DECLARE TYPE (Oracle Embedded SQL Directive)
- DELETE (Executable Embedded SQL)
- DESCRIBE (Executable Embedded SQL Extension)
- DESCRIBE DESCRIPTOR (Executable Embedded SQL)
- ENABLE THREADS (Executable Embedded SQL Extension)
- EXECUTE ... END-EXEC (Executable Embedded SQL Extension)
- EXECUTE (Executable Embedded SQL)
- EXECUTE DESCRIPTOR (Executable Embedded SQL)
- EXECUTE IMMEDIATE (Executable Embedded SQL)
- FETCH (Executable Embedded SQL)
- FETCH DESCRIPTOR (Executable Embedded SQL)
- FREE (Executable Embedded SQL Extension)
- GET DESCRIPTOR (Executable Embedded SQL)
- INSERT (Executable Embedded SQL)
- LOB APPEND (Executable Embedded SQL Extension)
- LOB ASSIGN (Executable Embedded SQL Extension)
- LOB CLOSE (Executable Embedded SQL Extension)
- LOB COPY (Executable Embedded SQL Extension)
- LOB CREATE TEMPORARY (Executable Embedded SQL Extension)
- LOB DESCRIBE (Executable Embedded SQL Extension)
- LOB DISABLE BUFFERING (Executable Embedded SQL Extension)
- LOB ENABLE BUFFERING (Executable Embedded SQL Extension)
- LOB ERASE (Executable Embedded SQL Extension)
- LOB FILE CLOSE ALL (Executable Embedded SQL Extension)
- LOB FILE SET (Executable Embedded SQL Extension)
- LOB FLUSH BUFFER (Executable Embedded SQL Extension)
- LOB FREE TEMPORARY (Executable Embedded SQL Extension)
- LOB LOAD (Executable Embedded SQL Extension)
- LOB OPEN (Executable Embedded SQL Extension)
- LOB READ (Executable Embedded SQL Extension)
- LOB TRIM (Executable Embedded SQL Extension)
- LOB WRITE (Executable Embedded SQL Extension)
- OBJECT CREATE (Executable Embedded SQL Extension)
- OBJECT DELETE (Executable Embedded SQL Extension)
- OBJECT DEREF (Executable Embedded SQL Extension)
- OBJECT FLUSH (Executable Embedded SQL Extension)



- OBJECT GET (Executable Embedded SQL Extension)
- OBJECT RELEASE (Executable Embedded SQL Extension)
- OBJECT SET (Executable Embedded SQL Extension)
- OBJECT UPDATE (Executable Embedded SQL Extension)
- OPEN (Executable Embedded SQL)
- OPEN DESCRIPTOR (Executable Embedded SQL)
- PREPARE (Executable Embedded SQL)
- REGISTER CONNECT (Executable Embedded SQL Extension)
- ROLLBACK (Executable Embedded SQL)
- SAVEPOINT (Executable Embedded SQL)
- SELECT (Executable Embedded SQL)
- SET DESCRIPTOR (Executable Embedded SQL)
- TYPE (Oracle Embedded SQL Directive)
- UPDATE (Executable Embedded SQL)
- VAR (Oracle Embedded SQL Directive)
- WHENEVER (Embedded SQL Directive)

# E.1 Summary of Precompiler Directives and Embedded SQL Statements

Embedded SQL statements place DDL, DML, and Transaction Control statements within a Pro\*C/C++ program. Table E-1 provides a functional summary of the embedded SQL statements and directives.

The Source/Type column in Table E-2 is displayed in the format:

Table E-1 Functional Summary of the Embedded SQL Statements and Directives

Source/Type	Description
Source	Is either standard SQL (S) or an Oracle extension (O).
Туре	Is either an executable (E) statement or a directive (D).

Table E-2 Precompiler Directives and Embedded SQL Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
ALLOCATE	O/E	To allocate memory for a cursor variable or an Object type.
ALLOCATE DESCRIPTOR	S/E	Allocate a descriptor for ANSI dynamic SQL.
CACHE FREE ALL	O/E	Frees all allocated object cache memory.
CALL	S/E	Call a stored procedure.
CLOSE	S/E	To disable a cursor, releasing the resources it holds.
COLLECTION APPEND	O/E	To append elements of one collection to the end of another collection.



Table E-2 (Cont.) Precompiler Directives and Embedded SQL Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
COLLECTION DESCRIBE	O/E	To obtain information about a collection.
COLLECTION GET	O/E	To retrieve the elements of a collection.
COLLECTION RESET	O/E	To reset the collection slice endpoints back to the beginning of the collection.
COLLECTION SET	O/E	To update values of a collection.
COLLECTION TRIM	O/E	To remove elements from the end of a collection.
COMMIT	S/E	To end the current transaction, making all database change permanent (optionally frees resources and disconnects from the database)
CONNECT	O/E	To log on to an instance.
CONTEXT ALLOCATE	O/E	To allocate memory for a SQLLIB runtime context.
CONTEXT FREE	O/E	To free memory for a SQLLIB runtime context.
CONTEXT OBJECT OPTION GET	O/E	To determine how options are set.
CONTEXT OBJECT OPTION SET	O/E	To set options.
CONTEXT USE	O/D	To specify which SQLLIB runtime context to use for subsequent executable SQL statements.
DEALLOCATE DESCRIPTOR	S/E	To deallocate a descriptor area to free memory.
DECLARE CURSOR	S/D	To declare a cursor, associating it with a query.
DECLARE DATABASE	O/D	To declare an identifier for a nondefault database to be accessed in subsequent embedded SQL statements.
DECLARE STATEMENT	S/D	To assign a SQL variable name to a SQL statement.
DECLARE TABLE	O/D	To declare the table structure for semantic checking of embedded SQL statements by Pro*C/C++.
DECLARE TYPE	O/D	To declare the type structure for semantic checking of embedded SQL statements by Pro*C/C++.
DELETE	S/E	To remove rows from a table or from a view's base table.
DESCRIBE	S/E	To initialize a descriptor, a structure holding host variable descriptions.
DESCRIBE DESCRIPTOR	S/E	Obtain information about the variables in an ANSI SQL statement.
ENABLE THREADS	O/E	To initialize a process that supports multiple threads.
EXECUTEEND-EXEC	O/E	To execute an anonymous PL/SQL block.
EXECUTE	S/E	To execute a prepared dynamic SQL statement.
EXECUTE DESCRIPTOR	S/E	To execute an ANSI Method 4 dynamic SQL statement.
EXECUTE IMMEDIATE	S/E	To prepare and execute a SQL statement with no host variables.
FETCH	S/E	To retrieve rows selected by a query.
FETCH DESCRIPTOR	S/E	To retrieve rows selected using ANSI Method 4 Dynamic SQL.
FREE	O/E	To free memory allocated in the object cache, or cursor.
GET DESCRIPTOR	S/E	To move information from an ANSI SQL descriptor area into host variables.
INSERT	S/E	To add rows to a table or to a view's base table.
LOB APPEND	O/E	To append a LOB to the end of another IOB.



Table E-2 (Cont.) Precompiler Directives and Embedded SQL Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
LOB ASSIGN	O/E	To assign a LOB or BFILE locator to another locator.
LOB CLOSE	O/E	To close an open LOB or BFILE.
LOB COPY	O/E	To copy all or part of a LOB value into another LOB.
LOB CREATE TEMPORARY	O/E	To create a temporary LOB.
LOB DESCRIBE	O/E	To retrieve attributes from a LOB.
LOB DISABLE BUFFERING	O/E	To disable LOB buffering.
LOB ENABLE BUFFERING	O/E	To enable LOB buffering.
LOB ERASE	O/E	To erase a given amount of LOB data starting from a given offset
LOB FILE CLOSE ALL	O/E	To close all open BFILEs.
LOB FILE SET	O/E	To set DIRECTORY and FILENAME in a BFILE locator.
LOB FLUSH BUFFER	O/E	To write the LOB buffers to the database server.
LOB FREE TEMPORARY	O/E	To free temporary space for the LOB locator.
LOB LOAD	O/E	To copy all or part of a BFILE into an internal LOB.
LOB OPEN	O/E	To open a LOB or BFILE to read or read/write access.
LOB READ	O/E	To read all or part of a LOB or BFILE into a buffer.
LOB TRIM	O/E	To truncate a lob value.
LOB WRITE	O/E	To write the contents of a buffer to a LOB.
OBJECT CREATE	O/E	To create a referenceable object in the cache.
OBJECT DELETE	O/E	To mark an object as deleted.
OBJECT DEREF	O/E	To dereference an object.
OBJECT FLUSH	O/E	To transmit persistent objects to server.
OBJECT GET	O/E	To convert an object attribute to a C type.
OBJECT RELEASE	O/E	To "unpin" an object in the cache.
OBJECT SET	O/E	To update object attributes in the cache.
OBJECT UPDATE	O/E	To mark an object in the cache as updated.
OPEN	S/E	To execute the query associated with a cursor.
OPEN DESCRIPTOR	S/E	To execute the query associated with a cursor (ANSI Dynamic SQL Method 4).
PREPARE	S/E	To parse a dynamic SQL statement.
REGISTER CONNECT	O/E	To enable a call to an external procedure.
ROLLBACK	S/E	To end the current transaction, discard all changes in the current transaction, and release all locks (optionally release resources and disconnect from the database).
SAVEPOINT	S/E	To identify a point in a transaction to which you can later roll back
SELECT	S/E	To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.
SET DESCRIPTOR	S/E	To set information in the descriptor area from host variables.
TYPE	O/D	To assign an external datatype to a whole class of host variables by equivalencing the external datatype to a user-defined datatype.
UPDATE	S/E	To change existing values in a table or in a view's base table.



Table E-2 (Cont.) Precompiler Directives and Embedded SQL Statements and Clauses

EXEC SQL Statement	Source/Type	Purpose
VAR	O/D	To override the default datatype and assign a specific external datatype to a host variable.
WHENEVER	S/D	To specify handling for error and warning conditions.

## **E.2 About The Statement Descriptions**

The directives and statements appear alphabetically. The description of each contains the following sections:

Directives and Statements	Description	
- Directives and Statements		
Purpose	Describes the basic uses of the statement.	
Prerequisites	Lists privileges you must have and steps that you must take before using the statement. Unless otherwise noted, most statements also require that the database be open by your instance.	
Syntax	Shows the keywords and parameters of the statement.	
Keywords and Parameters	Describes the purpose of each keyword and parameter.	
Usage Notes	Discusses how and when to use the statement.	
Examples	Shows example statements of the statement.	
Related Topics	Lists related statements, clauses, and sections of this manual.	

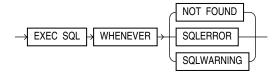
# E.3 How to Read Syntax Diagrams

*Syntax diagrams* are used to illustrate embedded SQL syntax. They are drawings that depict valid syntax paths.

Trace each diagram from left to right, in the direction shown by the arrows.

Statements and other keywords appear in UPPER CASE inside rectangles. Type them exactly as shown in the rectangles. Parameters appear in lower case inside ovals. Substitute variables for the parameters in statements you write. Operators, delimiters, and terminators appear in circles. Following the conventions defined in the Preface, a semicolon terminates statements.

If the syntax diagram has more than one path, you can choose any path to travel. If you have the choice of more than one keyword, operator, or parameter, your options appear in a vertical list. In the following example, you can travel down the vertical line as far as you like, then continue along any horizontal line:



According to the diagram, all of the following statements are valid:

```
EXEC SQL WHENEVER NOT FOUND ...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```

## E.3.1 Required Keywords and Parameters

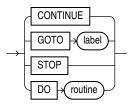
Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the *main path*, that is, on the horizontal line you are currently traveling. In the following example, *cursor* is a required parameter:



If there is a cursor named *emp\_cursor*, then, according to the diagram, the following statement is valid:

```
EXEC SQL CLOSE emp_cursor;
```

If any of the keywords or parameters in a vertical list appears on the main path, one of them is required. That is, you must choose one of the keywords or parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the four actions:



## E.3.2 Optional Keywords and Parameters

If keywords and parameters appear in a vertical list in the main path, they are optional. In the following example, AT :db\_name and WORK are optional:



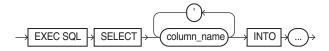
If there is a database named *oracle2*, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL ROLLBACK;
EXEC SQL ROLLBACK WORK;
EXEC SOL AT oracle2 ROLLBACK;
```



## E.3.3 Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, *column\_name* is inside a loop. So, after choosing one column name, you can go back repeatedly to choose another, separating the column names by a comma.

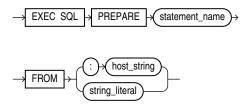


If DEBIT, CREDIT, and BALANCE are column names, then, according to the diagram, all of the following statements are valid:

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

# E.3.4 Multipart Diagrams

Read a multipart diagram as if all the main paths were joined end-to-end. The following example is a two-part diagram:



According to the diagram, the following statement is valid:

```
EXEC SQL PREPARE statement name FROM string literal;
```

## E.3.5 Oracle Names

The names of Oracle database objects, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs (\$), pound signs (#), and underscores (\_).

However, if a name is enclosed by quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle names are not case-sensitive except when enclosed by quotation marks.

## E.3.6 Statement Terminator

In all embedded SQL diagrams, each statement is understood to end with the statement terminator ";".

# E.4 ALLOCATE (Executable Embedded SQL Extension)

#### **Purpose**

To allocate a cursor variable to be referenced in a PL/SQL block, or to allocate space in the object cache.

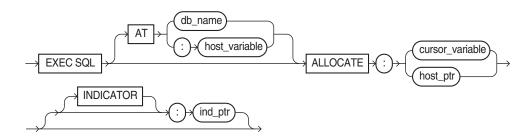
#### **Prerequisites**

A cursor variable of type sql\_cursor must be declared before allocating memory for the cursor variable.

Pointers to a host struct and, optionally, an indicator struct must be declared before allocating memory in the object cache.

An active connection to a database is required.

#### **Syntax**



#### **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
db_name	A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If it is omitted, or if it is an empty string, the default database connection is assumed.
host_variable	A host variable containing the name of the database connection.
cursor_variable	A cursor variable to be allocated.
host_ptr	A pointer to a host struct generated by OTT for object types, a context variable of type sql_context, a ROWID variable of type pointer to OCIRowid, or a LOB locator variable corresponding to the type of LOB.
ind_ptr	An optional pointer to an indicator struct.

#### **Usage Notes**

While a cursor is static, a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query.

#### Example

This partial example illustrates the use of the ALLOCATE statement in a Pro\*C/C++ program:

```
EXEC SQL BEGIN DECLARE SECTION;
SQL_CURSOR emp_cv;
struct{ ... } emp_rec;
```



```
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE :emp_cv;
EXEC SQL EXECUTE

BEGIN
     OPEN :emp_cv FOR SELECT * FROM emp;
END;
END-EXEC;
for (;;)
     {
     EXEC SQL FETCH :emp_cv INTO :emp_rec;
     ...
}
```

#### **Related Topics**

CACHE FREE ALL (Executable Embedded SQL Extension).

CLOSE (Executable Embedded SQL).

EXECUTE (Executable Embedded SQL) .

FETCH (Executable Embedded SQL).

FETCH DESCRIPTOR (Executable Embedded SQL).

FREE (Executable Embedded SQL Extension).

# E.5 ALLOCATE DESCRIPTOR (Executable Embedded SQL)

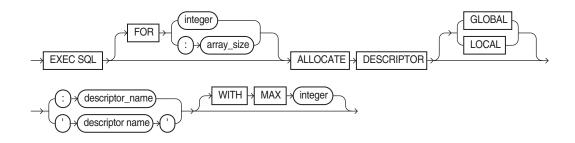
#### **Purpose**

An ANSI dynamic SQL statement that allocates a descriptor.

#### **Prerequisites**

None.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description	
array_size	Host variable containing number of rows to be processed.	
integer	Number of rows to be processed	

Keywords and Parameters	Description
descriptor_name	Host variable containing the name of the ANSI descriptor.
descriptor name	Name of the ANSI descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
WITH MAX integer	Maximum number of host variables. The default is 100.

#### **Usage Notes**

Use DYNAMIC=ANSI precompiler option. For information on using this statement, see "ALLOCATE DESCRIPTOR".

#### **Example**

EXEC SQL FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25 ;

#### **Related Topics**

DESCRIBE DESCRIPTOR (Executable Embedded SQL).

**DEALLOCATE DESCRIPTOR (Embedded SQL Statement)** 

GET DESCRIPTOR (Executable Embedded SQL).

SET DESCRIPTOR (Executable Embedded SQL).

# E.6 CACHE FREE ALL (Executable Embedded SQL Extension)

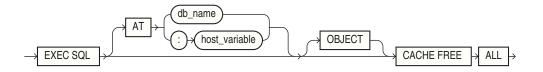
#### **Purpose**

To free all memory in the object cache.

#### **Prerequisites**

An active database connection must exist.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
db_name	A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If it is omitted, or if it is an empty string, the default database connection is assumed.
host_variable	A host variable containing the name of the database connection.



#### **Usage Notes**

When the connection count drops to zero, SQLLIB automatically frees all object cache memory. For more information, see "CACHE FREE ALL".

#### **Example**

EXEC SQL AT mydb CACHE FREE ALL ;

#### **Related Topics**

ALLOCATE (Executable Embedded SQL Extension).

FREE (Executable Embedded SQL Extension).

# E.7 CALL (Executable Embedded SQL)

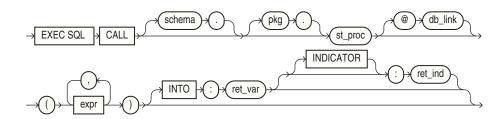
#### **Purpose**

To call a stored procedure.

#### **Prerequisites**

An active database connection must exist.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
schema	Is the schema containing the procedure. If you omit schema, Oracle assumes the procedure is in your own schema.
pkg	The package where the procedure is stored.
st_proc	The stored procedure to be called.
db_link	The complete or partial name of a database link to a remote database where the procedure is located. For information on referring to database links, see the <i>Oracle Database SQL Language Reference</i> .
expr	The list of expressions that are the parameters of the procedure.
ret_var	The host variable that receives the returned value of a function.
ret_ind	The indicator variable for ret_var.

#### **Usage Notes**

For more about this statement, see "About Calling a Stored PL/SQL or Java Subprogram".

#### **Example**

```
int emp_no;
char emp_name[10];
float salary;
char dept_name[20];
...
emp_no = 1325;
EXEC SQL CALL get_sal(:emp_no, :emp_name, :salary) INTO :dept_name;
/* Print emp_name, salary, dept_name */
```

#### **Related Topics**

None

# E.8 CLOSE (Executable Embedded SQL)

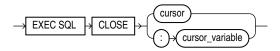
#### **Purpose**

To disable a cursor, freeing the resources acquired by opening the cursor, and releasing parse locks.

#### **Prerequisites**

The cursor or cursor variable be open if MODE=ANSI.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
cursor	A cursor to be closed.
cursor_variable	A cursor variable to be closed.

#### **Usage Notes**

Rows cannot be fetched from a closed cursor. A cursor need not be closed to be reopened. The HOLD\_CURSOR and RELEASE\_CURSOR precompiler options alter the effect of the CLOSE statement. For information on these options, see Precompiler Options.

#### Example

This example illustrates the use of the CLOSE statement:

```
EXEC SQL CLOSE emp_cursor;
```

#### **Related Topics**

PREPARE (Executable Embedded SQL).

DECLARE CURSOR (Embedded SQL Directive).

OPEN (Executable Embedded SQL).

# E.9 COLLECTION APPEND (Executable Embedded SQL Extension)

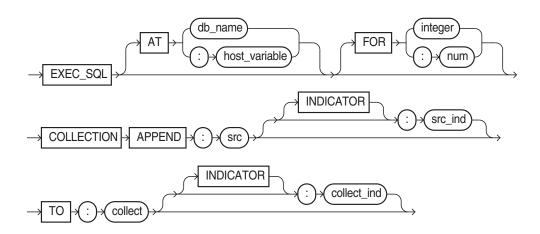
#### **Purpose**

To append elements of one collection to the end of another collection.

#### **Prerequisites**

You cannot append to a NULL collection, or append beyond the upper bound of a collection.

#### **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "COLLECTION APPEND".

#### **Related Topics**

See the other COLLECTION statements.

# E.10 COLLECTION DESCRIBE (Executable Embedded SQL Extension)

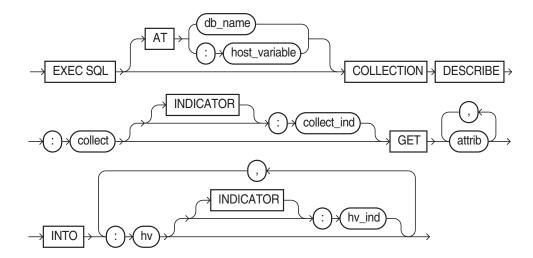
#### **Purpose**

To obtain information about a collection.

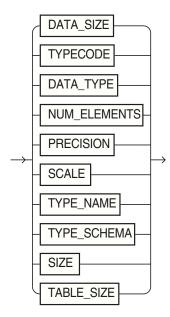
#### **Prerequisites**

Use the ALLOCATE and OBJECT GET statements to allocate a descriptor and to store collection attributes in the descriptor.

#### **Syntax**



where attrib is:



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "COLLECTION DESCRIBE".

#### **Related Topics**

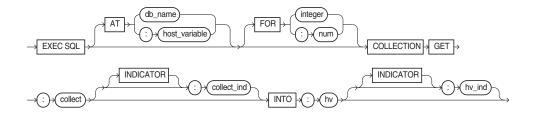
See the other COLLECTION statements.

# E.11 COLLECTION GET (Executable Embedded SQL Extension)

#### **Purpose**

To retrieve the elements of a collection.

#### **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "Collection Statements".

#### **Related Topics**

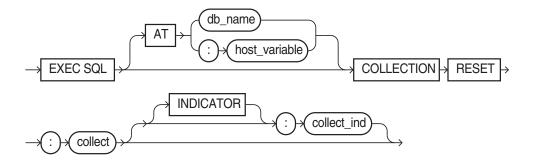
See the other COLLECTION statements.

# E.12 COLLECTION RESET (Executable Embedded SQL Extension)

#### **Purpose**

To reset the collection slice endpoints back to the beginning of the collection.

#### **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "COLLECTION RESET".

#### **Related Topics**

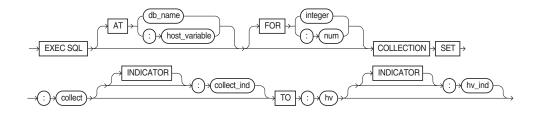
See the other COLLECTION statements.

# E.13 COLLECTION SET (Executable Embedded SQL Extension)

#### **Purpose**

To update element values in the current slice of a collection.

#### **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "COLLECTION SET".

#### **Related Topics**

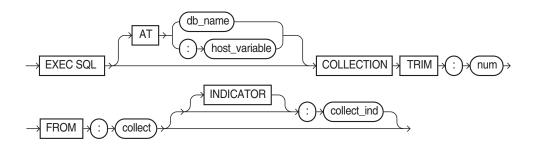
See the other COLLECTION statements.

# E.14 COLLECTION TRIM (Executable Embedded SQL Extension)

#### **Purpose**

To remove elements from the end of collection.

#### **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "COLLECTION TRIM".

#### **Related Topics**

See the other COLLECTION statements.

# E.15 COMMIT (Executable Embedded SQL)

#### **Purpose**

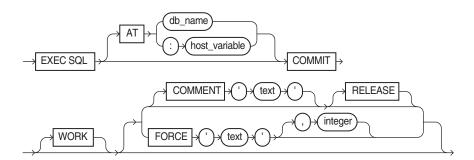
To end your current transaction, making permanent all its changes to the database and optionally freeing all resources and disconnecting.

#### **Prerequisites**

To commit your current transaction, no privileges are necessary.

To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

#### **Syntax**



#### **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
AT	Identifies the database to which the COMMIT statement is issued. The database can be identified by either:
	db_name: is a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: is a host variable whose value is a db_name. If you omit this clause, Oracle issues the statement to your default database.
WORK	Is supported only for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.
COMMENT	Specifies a Comment to be associated with the current transaction. The 'text' is a quoted literal of up to 50 characters that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt.
RELEASE	Frees all resources and disconnects the application from the server.



Keywords and Parameters	Description
FORCE	Manually commits an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can also use the optional integer to explicitly assign the transaction a system change number (SCN). If you omit the integer, the transaction is committed using the current SCN.

Always explicitly commit or rollback the last transaction in your program by using the COMMIT or ROLLBACK statement and the RELEASE option. Oracle automatically rolls back changes if the program terminates abnormally.

The COMMIT statement has no effect on host variables or on the flow of control in the program. For more information on this statement, see <u>Database Concepts</u>.

# **Example**

This example illustrates the use of the embedded SQL COMMIT statement:

```
EXEC SQL AT sales db COMMIT RELEASE;
```

#### **Related Topics**

ROLLBACK (Executable Embedded SQL) .

SAVEPOINT (Executable Embedded SQL).

# E.16 CONNECT (Executable Embedded SQL Extension)

# **Purpose**

To log on to a database.

# **Prerequisites**

You must have CREATE SESSION system privilege in the specified database.

# **Syntax**

```
EXEC SQL
CONNECT { :user IDENTIFIED BY :oldpswd | :usr_psw }
[[AT { dbname | :host_variable }] USING :connect_string ]
[ {ALTER AUTHORIZATION :newpswd | IN {SYSDBA | SYSOPER | SYSBACKUP} MODE} ]
END-EXEC.
```

Keywords and Parameters	Description
user password	Specifies your username and password separately.



Keywords and Parameters	Description
user_password	A single host variable containing the username and password separated by a slash (/).
	To allow Oracle to verify your connection through your operating system, specify "/" as the :user_password value.
AT	Identifies the database to which the connection is made. The database can be identified by either:
	db_name: is a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: is a host variable whose value is a db_name. If you omit this clause, Oracle issues the statement to your default database.
USING	Uses the Oracle Net database specification string used to connect to a nondefault database. If you omit this clause, you are connected to your default database.
ALTER AUTHORIZATION	Change password to the following string.
new_password	The new password.
IN SYSDBA MODE IN SYSOPER MODE IN SYSBACKUP MODE	Connect with SYSDBA, SYSOPER, or SYSBACKUP system privileges. Not allowed when ALTER AUTHORIZATION is used, or precompiler option AUTO_CONNECT is set to YES.

A program can have multiple connections, but can only connect once to your default database. For more information on this statement, see "Embedded OCI Release 7 Calls".

# **Example**

The following example illustrate the use of CONNECT:

```
EXEC SQL CONNECT :username
    IDENTIFIED BY :password ;
```

You can also use this statement in which the value of *userid* is the value of *username* and the value of *password* separated by a "/", such as 'SCOTT/TIGER':

```
EXEC SQL CONNECT :userid ;
```

# **Related Topics**

COMMIT (Executable Embedded SQL).

DECLARE DATABASE (Oracle Embedded SQL Directive).

ROLLBACK (Executable Embedded SQL).

# E.17 CONTEXT ALLOCATE (Executable Embedded SQL Extension)

# **Purpose**

To initialize a SQLLIB runtime context that is referenced in an EXEC SQL CONTEXT USE statement.

#### **Prerequisites**

The runtime context must be declared of type sql context.

# **Syntax**



#### **Keyword and Parameters**

<b>Keywords and Parameters</b>	Description
context	Is the SQLLIB runtime context for which memory is to be allocated.

# **Usage Notes**

In a multithreaded application, execute this function once for each runtime context.

For more information on this statement, see "SQLLIB Extensions for OCI Release 8 Interoperability".

#### **Example**

This example illustrates the use of a CONTEXT ALLOCATE statement in a Pro\*C/C++ program:

EXEC SQL CONTEXT ALLOCATE :ctx1;

# **Related Topics**

CONTEXT FREE (Executable Embedded SQL Extension).

CONTEXT USE (Oracle Embedded SQL Directive).

ENABLE THREADS (Executable Embedded SQL Extension).

# E.18 CONTEXT FREE (Executable Embedded SQL Extension)

# **Purpose**

To free all memory associated with a runtime context and place a null pointer in the host program variable.

# **Prerequisites**

The CONTEXT ALLOCATE statement must be used to allocate memory for the specified runtime context before the CONTEXT FREE statement can free the memory allocated for it.

# **Syntax**





### **Keyword and Parameter**

Keyword and Parameter	Description
:context	The allocated runtime context for which the memory is to be deallocated.

# **Usage Notes**

For more information on this statement, see "SQLLIB Extensions for OCI Release 8 Interoperability".

# **Example**

This example illustrates the use of a CONTEXT FREE statement in a Pro\*C/C++ program:

EXEC SQL CONTEXT FREE :ctx1;

#### **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension).

CONTEXT USE (Oracle Embedded SQL Directive).

ENABLE THREADS (Executable Embedded SQL Extension).

# E.19 CONTEXT OBJECT OPTION GET (Executable Embedded SQL Extension)

# **Purpose**

To determine the values of options set by CONTEXT OBJECT OPTION SET for the context in use.

# **Prerequisites**

Precompiler option OBJECTS must be set to YES.

# **Syntax**



Keywords and Parameters	Description
option	Option values are described in CONTEXT OBJECT OPTION SET.
host_variable	Output variable of type STRING, VARCHAR, or CHARZ, in the same order as the <i>option</i> list.



See "CONTEXT OBJECT OPTION SET".

#### **Example**

```
char EuroFormat[50];
...
EXEC SQL CONTEXT OBJECT OPTION GET DATEFORMAT INTO :EuroFormat ;
printf("Date format is %s\n", EuroFormat);
```

# **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension).

CONTEXT FREE (Executable Embedded SQL Extension).

CONTEXT OBJECT OPTION SET (Executable Embedded SQL Ext) .

CONTEXT USE (Oracle Embedded SQL Directive).

# E.20 CONTEXT OBJECT OPTION SET (Executable Embedded SQL Ext)

#### **Purpose**

To set options to specified values of Date attributes: DATEFORMAT, DATELANG for the context in use.

#### **Prerequisites**

Precompiler option OBJECTS must be set to YES.

#### **Syntax**



# **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
option	Option values are described in CONTEXT OBJECT OPTION SET.
host_variable	Input variable of type STRING, VARCHAR, or CHARZ. In the same order as the <i>option</i> list.

# **Usage Notes**

See "CONTEXT OBJECT OPTION GET".

#### **Example**

```
char *new_format = "DD-MM-YYY";
char *new_lang = "French";
```



• • •

EXEC SQL CONTEXT OBJECT OPTION SET DATEFORMAT, DATELANG TO :new format, :new lang;

# **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension).

CONTEXT FREE (Executable Embedded SQL Extension).

CONTEXT USE (Oracle Embedded SQL Directive).

CONTEXT OBJECT OPTION SET (Executable Embedded SQL Ext) .

# E.21 CONTEXT USE (Oracle Embedded SQL Directive)

# **Purpose**

To instruct the precompiler to use the specified SQLLIB runtime context on subsequent executable SQL statements.

# **Prerequisites**

The runtime context specified by the CONTEXT USE directive must be previously declared.

# **Syntax**



# **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
context	The allocated runtime context to use for subsequent executable SQL statements that follow it. For example, after specifying in your source code which context to use (multiple contexts can be allocated), you can connect to the Oracle Server and perform database operations within the scope of that context. DEFAULT indicates that the global context that you worked with is to be used.
DEFAULT	Indicates that the global context is to be used.

#### **Usage Notes**

This statement has no effect on declarative statements such as EXEC SQL INCLUDE or EXEC ORACLE OPTION. It works similarly to the EXEC SQL WHENEVER directive in that it affects all executable SQL statements which positionally follow it in a given source file without regard to standard C scope rules.

For more information on this statement, see "SQLLIB Extensions for OCI Release 8 Interoperability".

#### **Example**

This example illustrates the use of a CONTEXT USE directive in a Pro\*C/C++ embedded SQL program:



EXEC SQL CONTEXT USE :ctx1;

#### **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension).

CONTEXT FREE (Executable Embedded SQL Extension).

ENABLE THREADS (Executable Embedded SQL Extension).

# E.22 DEALLOCATE DESCRIPTOR (Embedded SQL Statement)

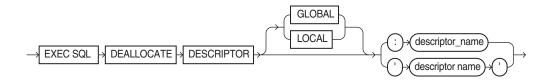
#### **Purpose**

An ANSI dynamic SQL statement that deallocates a descriptor area to free memory.

#### **Prerequisites**

The descriptor specified by the DEALLOCATE DESCRIPTOR statement must be previously allocated using the ALLOCATE DESCRIPTOR statement.

#### **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
descriptor_name 'descriptor name'	Host variable containing the name of the allocated ANSI descriptor.  Name of the allocated ANSI descriptor.

## **Usage Notes**

Use DYNAMIC=ANSI precompiler option.

For more information on this statement, see "DEALLOCATE DESCRIPTOR".

#### **Example**

EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES';

# **Related Topics**

ALLOCATE DESCRIPTOR (Executable Embedded SQL).

DESCRIBE (Executable Embedded SQL Extension).

GET DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# SET DESCRIPTOR (Executable Embedded SQL).

# E.23 DECLARE CURSOR (Embedded SQL Directive)

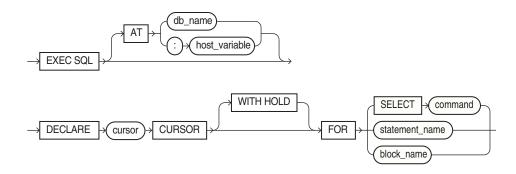
# **Purpose**

To declare a cursor, giving it a name and associating it with a SQL statement or a PL/SQL block.

# **Prerequisites**

If you associate the cursor with an identifier for a SQL statement or PL/SQL block, you must have declared this identifier in a previous DECLARE STATEMENT statement.

#### **Syntax**



#### **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
AT	Identifies the database on which the cursor is declared. The database can be identified by either:
db_name	Database identifier declared in a previous DECLARE DATABASE statement.
host_variable	Host variable whose value is a previously declared db_name.
	If you omit this clause, Oracle declares the cursor on your default database.
cursor	Name of the cursor to be declared.
WITH HOLD	Cursor remains open after a COMMIT. The cursor must not be declared for UPDATE.
SELECT statement	Is a SELECT statement to be associated with the cursor. The following statement cannot contain an INTO clause.
statement_name	Identifies a SQL statement or PL/SQL block to be associated with the cursor. The <i>statement_name</i> or <i>block_name</i> must be previously declared in a DECLARE STATEMENT statement.

# **Usage Notes**

You must declare a cursor before referencing it in other embedded SQL statements. The scope of a cursor declaration is global within its precompilation unit and the name of each cursor must be unique in its scope. You cannot declare two cursors with the same name in a single precompilation unit.

You can reference the cursor in the WHERE clause of an UPDATE or DELETE statement using the CURRENT OF syntax if the cursor has been opened with an OPEN statement and positioned on a row with a FETCH statement. For more information on this statement, see "Cursor Usage in Embedded PL/SQL".

# **Example**

This example illustrates the use of a DECLARE CURSOR statement:

```
EXEC SQL DECLARE emp_cursor CURSOR

FOR SELECT ename, empno, job, sal

FROM emp

WHERE deptno = :deptno

FOR UPDATE OF sal;
```

# **Related Topics**

```
CLOSE (Executable Embedded SQL).
```

DECLARE DATABASE (Oracle Embedded SQL Directive).

DECLARE STATEMENT (Embedded SQL Directive).

DELETE (Executable Embedded SQL).

FETCH (Executable Embedded SQL).

OPEN DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

SELECT (Executable Embedded SQL).

UPDATE (Executable Embedded SQL).

# E.24 DECLARE DATABASE (Oracle Embedded SQL Directive)

# **Purpose**

To declare an identifier for a nondefault database to be accessed in subsequent embedded SQL statements.

#### **Prerequisites**

You must have access to a username on the nondefaultnondefault database.

#### **Syntax**



Keyword and Parameter	Description
db_name	Is the identifier established for the nondefault database.



You declare a *db\_name* for a nondefault database so that other embedded SQL statements can refer to that database using the AT clause. Before issuing a CONNECT statement with an AT clause, you must declare a *db\_name* for the nondefault database with a DECLARE DATABASE statement.

For more information on this statement, see "Single Explicit Connection".

#### **Example**

This example illustrates the use of a DECLARE DATABASE directive:

```
EXEC SQL DECLARE oracle3 DATABASE;
```

# **Related Topics**

COMMIT (Executable Embedded SQL).

CONNECT (Executable Embedded SQL Extension).

DECLARE CURSOR (Embedded SQL Directive).

DECLARE STATEMENT (Embedded SQL Directive) .

DELETE (Executable Embedded SQL).

EXECUTE ... END-EXEC (Executable Embedded SQL Extension) .

EXECUTE IMMEDIATE (Executable Embedded SQL).

INSERT (Executable Embedded SQL).

SELECT (Executable Embedded SQL).

UPDATE (Executable Embedded SQL).

# E.25 DECLARE STATEMENT (Embedded SQL Directive)

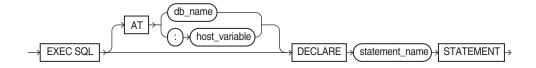
#### **Purpose**

To declare an identifier for a SQL statement or PL/SQL block to be used in other embedded SQL statements.

# **Prerequisites**

None.

## **Syntax**





### **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
AT	Identifies the database on which the SQL statement or PL/SQL block is declared. The database can be identified by either:
	db_name: Is a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: Is a host variable whose value is a db_name. If you omit this clause, Oracle declares the SQL statement or PL/SQL block to your default database.
statement_name	Is the declared identifier for the statement.

#### **Usage Notes**

You must declare an identifier for a SQL statement or PL/SQL block with a DECLARE STATEMENT statement only if a DECLARE CURSOR statement referencing the identifier appears physically (not logically) in the embedded SQL program before the PREPARE statement that parses the statement or block and associates it with its identifier.

The scope of a statement declaration is global within its precompilation unit, like a cursor declaration. For more information on this statement, see Datatypes and Host Variables and Oracle Dynamic SQL.

#### Example I

This example illustrates the use of the DECLARE STATEMENT statement:

```
EXEC SQL AT remote_db DECLARE my_statement STATEMENT;
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL EXECUTE my statement;
```

#### **Example II**

In this example from a Pro\*C/C++ embedded SQL program, the DECLARE STATEMENT statement is required because the DECLARE CURSOR statement precedes the PREPARE statement:

```
EXEC SQL DECLARE my_statement STATEMENT;
EXEC SQL DECLARE emp_cursor CURSOR FOR my_statement;
EXEC SQL PREPARE my_statement FROM :my_string;
...
```

# **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE DATABASE (Oracle Embedded SQL Directive).

FETCH (Executable Embedded SQL).

OPEN DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).



# E.26 DECLARE TABLE (Oracle Embedded SQL Directive)

# **Purpose**

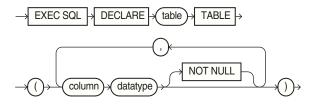
To define the structure of a table or view, including each column's datatype, default value, and NULL or NOT NULL specification for semantic checking by the Oracle Precompilers.

#### **Prerequisites**

None.

### **Syntax**

For relational tables, the syntax is:



For object tables, the syntax is:.



# **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
table	Is the name of the declared table.
column	Is a column of the table.
datatype	Is the datatype of a column. For information on datatypes, see "Oracle Datatypes".
	If the datatype is a user-defined object, a <i>size</i> may be entered in parentheses. The size cannot be a macro or a complex C expression. The size can be omitted. See examples.
NOT NULL	Specifies that a column cannot contain NULLs.
obj_type	Is an object type.

# **Usage Notes**

For information on using this statement, see "About Using DECLARE TABLE".

#### **Examples**

The following statement declares the PARTS table with the PARTNO, BIN, and QTY columns:



```
EXEC SQL DECLARE parts TABLE
(partno NUMBER NOT NULL,
bin NUMBER,
qty NUMBER);
```

#### Use of an object type:

```
EXEC SQL DECLARE person TYPE AS OBJECT (name VARCHAR2(20), age INT); EXEC SQL DECLARE odjtab1 TABLE OF person;
```

# **Related Topics**

DECLARE TYPE (Oracle Embedded SQL Directive).

# E.27 DECLARE TYPE (Oracle Embedded SQL Directive)

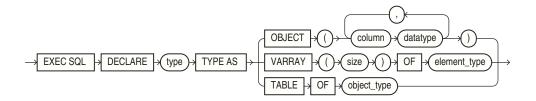
# **Purpose**

To define the attributes of a type for a semantics check by the precompiler.

# **Prerequisites**

None.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
column	Name of column.
datatype	The datatype of the column
size	Number of elements in the VARRAY.
element_type	Type of element in the VARRAY. It can be an object.
object_type	A previously declared object type.

#### **Usage Notes**

For information on using this statement, see "About Using DECLARE TYPE".

#### **Example**

```
EXEC SQL DECLARE project_type TYPE AS OBJECT(

pno CHAR(5),

pname CHAR(20),

budget NUMBER);

EXEC SQL DECLARE project_array TYPE as VARRAY(20) OF project_type;

EXEC SQL DECLARE employees TYPE AS TABLE OF emp_objects;
```

# **Related Topics**

DECLARE TABLE (Oracle Embedded SQL Directive) .

# E.28 DELETE (Executable Embedded SQL)

# **Purpose**

To remove rows from a table or from a view's base table.

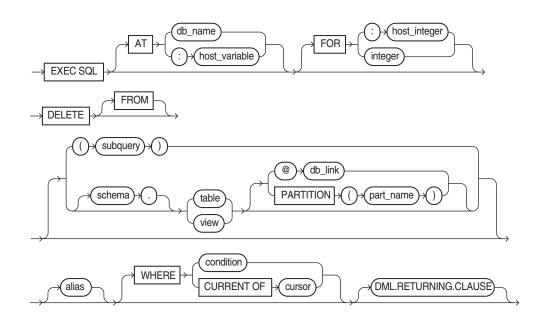
# **Prerequisites**

For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.

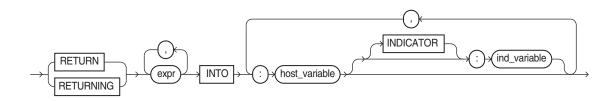
For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

The DELETE ANY TABLE system privilege also provides the ability to delete rows from any table or any view's base table.

# **Syntax**



Where the DML returning clause is:





#### **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database to which the DELETE statement is issued. The database can be identified by either:
	db_name: a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: a host variable whose value is a previously declared db_name.
	If you omit this clause, the DELETE statement is issued to your default database.
FOR :host_integer	Limits the number of times the statement is executed if the WHERE clause contains array host variables. If you omit this clause, the database executes the statement once for each component of the smallest array.
subquery	Is a subquery that returns new values that are assigned to the corresponding columns. For the syntax of a subquery, see SELECT.
schema	Is the schema containing the table or view. If you omit schema, the database assumes the table or view is in your own schema.
table	The name of a table from which the rows are to be deleted.
view	The name of a view. the database deletes rows from the view's base table.
FOR :host_integer	Limits the number of times the statement is executed if the WHERE clause contains array host variables. If you omit this clause, the database executes the statement once for each component of the smallest array.
subquery	Is a subquery that returns new values that are assigned to the corresponding columns. For the syntax of a subquery, see SELECT.
schema	Is the schema containing the table or view. If you omit schema, the database assumes the table or view is in your own schema.
table	The name of a table from which the rows are to be deleted.
view	The name of a view. The database deletes rows from the view's base table.

#### **Usage Notes**

The host variables in the WHERE clause must be either all scalars or all arrays. If they are scalars, the database executes the DELETE statement only once. If they are arrays, the database executes the statement once for each set of array components. Each execution may delete zero, one, or multiple rows.

Array host variables in the WHERE clause can have different sizes. In this case, the number of times the database executes the statement is determined by the smaller of the following values:

- The size of the smallest array
- The value of the :host\_integer in the optional FOR clause

If no rows satisfy the condition, no rows are deleted and the SQLCODE returns a NOT\_FOUND condition.

The cumulative number of rows deleted is returned through the SQLCA. If the WHERE clause contains array host variables, this value reflects the total number of rows deleted for all components of the array processed by the DELETE statement.



If no rows satisfy the condition, the database returns an error through the SQLCODE of the SQLCA. If you omit the WHERE clause, the database raises a warning flag in the fifth component of SQLWARN in the SQLCA. For more information on this statement and the SQLCA, see Handling Runtime Errors.

You can use Comments in a DELETE statement to pass instructions, or *hints*, to the optimizer. The optimizer uses hints to choose an execution plan for the statement.

# **Example**

This example illustrates the use of the DELETE statement within a Pro\*C/C++ embedded SQL program:

```
EXEC SQL DELETE FROM emp

WHERE deptno = :deptno

AND job = :job;

EXEC SQL DECLARE emp_cursor CURSOR

FOR SELECT empno, comm

FROM emp;

EXEC SQL OPEN emp_cursor;

EXEC SQL FETCH c1

INTO :emp_number, :commission;

EXEC SQL DELETE FROM emp

WHERE CURRENT OF emp_cursor;
```

# **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive) .

DECLARE STATEMENT (Embedded SQL Directive).

# E.29 DESCRIBE (Executable Embedded SQL Extension)

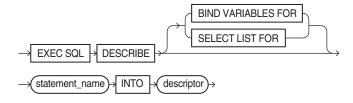
## **Purpose**

To populate an Oracle descriptor with information about a dynamic SQL statement or PL/SQL block.

#### **Prerequisites**

You must have prepared the SQL statement or PL/SQL block in a previous embedded SQL PREPARE statement.

#### **Syntax**





#### **Keywords and Parameters**

Keywords and Parameters	Description
BIND VARIABLES FOR	Initializes the descriptor to hold information about the input variables for the SQL statement or PL/SQL block.
SELECT LIST FOR	Initializes the descriptor to hold information about the select list of a SELECT statement.  The default is SELECT LIST FOR.
statement_name	Identifies a SQL statement or PL/SQL block previously prepared with a PREPARE statement.
descriptor	Is the name of the descriptor to be populated.

# **Usage Notes**

You must issue a DESCRIBE statement before manipulating the bind or select descriptor within an embedded SQL program.

You cannot describe both input variables and output variables into the same descriptor.

The number of variables found by a DESCRIBE statement is the total number of place-holders in the prepare SQL statement or PL/SQL block, rather than the total number of uniquely named place-holders. For more information on this statement, see Oracle Dynamic SQL.

# Example

This example illustrates the use of the DESCRIBE statement in a Pro\*C/C++ embedded SQL program:

```
EXEC SQL PREPARE my_statement FROM :my_string;

EXEC SQL DECLARE emp_cursor

FOR SELECT empno, ename, sal, comm

FROM emp

WHERE deptno = :dept_number;

EXEC SQL DESCRIBE BIND VARIABLES FOR my_statement

INTO bind_descriptor;

EXEC SQL OPEN emp_cursor

USING bind_descriptor;

EXEC SQL DESCRIBE SELECT LIST FOR my_statement

INTO select_descriptor;

EXEC SQL FETCH emp_cursor

INTO select descriptor;
```

#### **Related Topics**

PREPARE (Executable Embedded SQL).

# E.30 DESCRIBE DESCRIPTOR (Executable Embedded SQL)

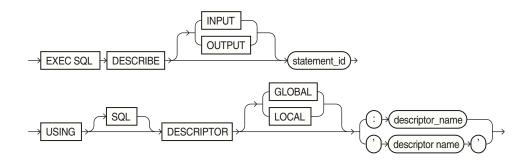
#### **Purpose**

An ANSI dynamic SQL statement used to obtain information about a SQL statement, and to store it in a descriptor.

#### **Prerequisites**

You must have prepared the SQL statement in a previous embedded SQL PREPARE statement.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
statement_id	The name of the previously prepared SQL statement or PL/SQL block. OUTPUT is the default.
desc_name	Host variable containing the name of the descriptor that will hold information about the SQL statement.
'descriptor name'	The name of the descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.

#### **Usage Notes**

Use DYNAMIC=ANSI precompiler option.

Only COUNT and NAME are implemented for the INPUT descriptor.

The number of variables found by a DESCRIBE statement is the total number of place-holders in the prepare SQL statement or PL/SQL block, rather than the total number of uniquely named place-holders. For more information on this statement, see "DESCRIBE INPUT", and "DESCRIBE OUTPUT".

# **Example**

```
EXEC SQL PREPARE s FROM :my_stament;
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in';
```

#### **Related Topics**

ALLOCATE DESCRIPTOR (Executable Embedded SQL).

DEALLOCATE DESCRIPTOR (Embedded SQL Statement).

GET DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

SET DESCRIPTOR (Executable Embedded SQL).

# E.31 ENABLE THREADS (Executable Embedded SQL Extension)

# **Purpose**

To initialize a process that supports multiple threads.

# **Prerequisites**

You must be developing a precompiler application for and compiling it on a platform that supports multithreaded applications, and THREADS=YES must be specified on the command line.



When using XA with the Pro\*C/C++ Precompiler, you must use multithreaded processing provided by XA. Use of multithreaded processing provided by Pro\*C/C++ using the statement EXEC SQL ENABLE THREADS will result in an error.

#### **Syntax**



# **Keywords and Parameters**

None.

#### **Usage Notes**

The ENABLE THREADS statement must be executed before any other executable SQL statement and before spawning any thread. This statement does not require a host-variable specification.

For more information on this statement, see "SQLLIB Extensions for OCI Release 8 Interoperability".

#### **Example**

This example illustrates the use of the ENABLE THREADS statement in a Pro\*C/C++ program:

EXEC SQL ENABLE THREADS;

# **Related Topics**

CONTEXT ALLOCATE (Executable Embedded SQL Extension).

CONTEXT FREE (Executable Embedded SQL Extension).

CONTEXT USE (Oracle Embedded SQL Directive).



# E.32 EXECUTE ... END-EXEC (Executable Embedded SQL Extension)

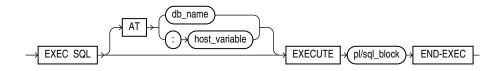
# **Purpose**

To embed an anonymous PL/SQL block into a Pro\*C/C++ program.

#### **Prerequisites**

None.

#### **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database on which the PL/SQL block is executed. The database can be identified by either:
	db_name: Is a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: Is a host variable whose value is a previously declared db_name. If you omit this clause, the PL/SQL block is executed on your default database
pl/sql_block	For information on PL/SQL, including how to write PL/SQL blocks, see the <i>Oracle Database PL/SQL Language Reference</i> .
END-EXEC	This keyword must appear after the embedded PL/SQL block, regardless of which programming language your Oracle Precompiler program uses. The keyword END-EXEC must be followed by the C/C++ statement terminator, ";".

# **Usage Notes**

Since the Pro\*C/C++ treats an embedded PL/SQL block like a single embedded SQL statement, you can embed a PL/SQL block anywhere in a program that you can embed a SQL statement. For more information on embedding PL/SQL blocks in Oracle Precompiler programs, see Embedded PL/SQL.

# Example

Placing this EXECUTE statement in a Pro\*C/C++ program embeds a PL/SQL block in the program:

```
EXEC SQL EXECUTE
    BEGIN
    SELECT ename, job, sal
    INTO :emp_name:ind_name, :job_title, :salary
```



```
FROM emp
     WHERE empno = :emp_number;
IF :emp_name:ind_name IS NULL
     THEN RAISE name_missing;
END IF;
END;
END-EXEC;
```

# **Related Topics**

EXECUTE IMMEDIATE (Executable Embedded SQL).

# E.33 EXECUTE (Executable Embedded SQL)

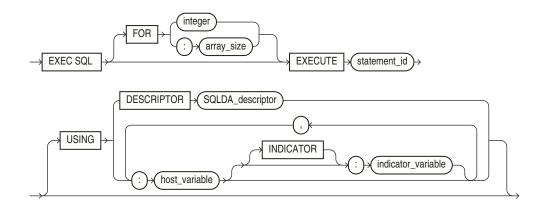
# **Purpose**

In Oracle dynamic SQL, to execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block that has been previously prepared with an embedded SQL PREPARE statement. For ANSI Dynamic SQL Method 4 see "EXECUTE DESCRIPTOR (Executable Embedded SQL)".

#### **Prerequisites**

You must first prepare the SQL statement or PL/SQL block with an embedded SQL PREPARE statement.

# **Syntax**



Keywords and Parameters	Description
FOR :array_size	Host variable containing the number of rows to be processed.  Number of rows to be processed.
FOR integer	Limits the number of times the statement is executed when the USING clause contains array host variables If you omit this clause, Oracle executes the statement once for each component of the smallest array.
statement_id	A precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE statement to associate the precompiler identifier with the statement or PL/SQL block.

Keywords and Parameters	Description
USING DESCRIPTOR SQLDA_descriptor	Uses an Oracle descriptor. CANNOT be used together with an ANSI descriptor (INTO clause).
USING	Specifies a list of host variables with optional indicator variables that Oracle substitutes as input variables into the statement to be executed. The host and indicator variables must be either all scalars or all arrays.
host_variable	Host variables.
indicator_variable	Indicator variables.

For more information on this statement, see Oracle Dynamic SQL for the Oracle version.

# **Example**

This example illustrates the use of the EXECUTE statement in a Pro\*C/C++ program:

```
EXEC SQL PREPARE my_statement
   FROM :my_string;
EXEC SQL EXECUTE my_statement
   USING :my_var;
```

# **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive).

PREPARE (Executable Embedded SQL).

# E.34 EXECUTE DESCRIPTOR (Executable Embedded SQL)

#### **Purpose**

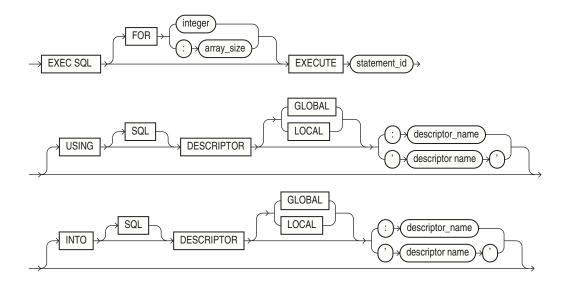
In ANSI SQL Method 4, to execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block that has been previously prepared with an embedded SQL PREPARE statement.

#### **Prerequisites**

You must first prepare the SQL statement or PL/SQL block with an embedded SQL PREPARE statement.



# **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
FOR :array_size	Host variable containing the number of rows to be processed.  Number of rows to be processed.
FOR integer	Limits the number of times the statement is executed. Oracle executes the statement once for each component of the smallest array.
statement_id	A precompiler identifier associated with the SQL statement or PL/SQL block to be executed. Use the embedded SQL PREPARE statement to associate the precompiler identifier with the statement or PL/SQL block.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
USING	An ANSI descriptor.
descriptor_name	Host variable containing name of the input descriptor.
'descriptor name'	Name of the input descriptor.
INTO	An ANSI descriptor.
descriptor_name	Host variable containing name of the output descriptor.
'descriptor name'	Name of the output descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.

# **Usage Notes**

For more information on this statement, see ANSI Dynamic SQL.

# **Examples**

The ANSI dynamic SQL Method 4 allows DML Returning clauses in a SELECT to be supported by the INTO clause in EXECUTE:

EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES';

#### **Related Topics**

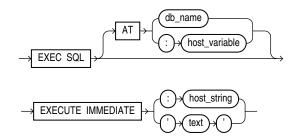
DECLARE DATABASE (Oracle Embedded SQL Directive).

PREPARE (Executable Embedded SQL).

# E.35 EXECUTE IMMEDIATE (Executable Embedded SQL)

# **Purpose**

To prepare and execute a DELETE, INSERT, or UPDATE statement or a PL/SQL block containing no host variables.



# **Prerequisites**

None.

# **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
AT	Identifies the database on which the SQL statement or PL/SQL block is executed. The database can be identified by either:
	db_name: Is a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: Is a host variable whose value is a previously declared db_name. If you omit this clause, the statement or block is executed on your default database.
text	Is a quoted text literal (or a text literal without quotes) containing the SQL statement or PL/SQL block to be executed.
	The SQL statement can only be a DELETE, INSERT, or UPDATE statement.
host_string	A host variable containing a SQL statement.

## **Usage Notes**

When you issue an EXECUTE IMMEDIATE statement, Oracle parses the specified SQL statement or PL/SQL block, checking for errors, and executes it. If any errors are encountered, they are returned in the SQLCODE component of the SQLCA.

For more information on this statement, see Oracle Dynamic SQL and ANSI Dynamic SQL.

# **Example**

This example illustrates the use of the EXECUTE IMMEDIATE statement:

EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM emp WHERE empno = 9460';

#### **Related Topics**

EXECUTE (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.36 FETCH (Executable Embedded SQL)

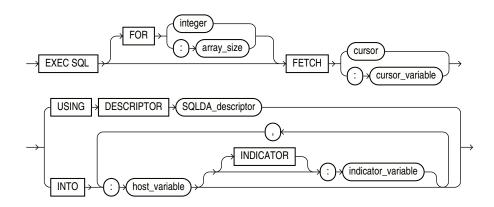
# **Purpose**

In Oracle dynamic SQL, to retrieve one or more rows returned by a query, assigning the select list values to host variables. For ANSI Dynamic SQL Method 4 see "FETCH DESCRIPTOR (Executable Embedded SQL)".

# **Prerequisites**

You must first open the cursor with an OPEN statement.

# **Syntax**



<b>Keywords and Parameters</b>	Description
FOR :array_size	Host variable containing the number of rows to be processed.  Number of rows to be processed.
FOR integer	Limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle fetches enough rows to fill the smallest array.
cursor	A cursor that is declared by a DECLARE CURSOR statement. The FETCH statement returns one of the rows selected by the query associated with the cursor.



Keywords and Parameters	Description
cursor_variable	A cursor variable is allocated an ALLOCATE statement. The FETCH statement returns one of the rows selected by the query associated with the cursor variable.
INTO	Specifies a list of host variables and optional indicator variables into which data is fetched. These host variables and indicator variables must be declared within the program.
host_variable	Host variables to receive data,
indicator_variables	Host indicator variables.
USING SQLDA_variable	Specifies the Oracle descriptor referenced in a previous DESCRIBE statement. Only use this clause with dynamic embedded SQL, method 4. The USING clause does not apply when a cursor variable is used.

The FETCH statement reads the rows of the active set and names the output variables which contain the results. Indicator values are set to -1 if their associated host variable is NULL. The first FETCH statement for a cursor also sorts the rows of the active set, if necessary.

The number of rows retrieved is specified by the size of the output host variables and the value specified in the FOR clause. The host variables to receive the data must be either all scalars or all arrays. If they are scalars, Oracle fetches only one row. If they are arrays, Oracle fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle fetches is determined by the smaller of the following values:

- The size of the smallest array
- The value of the :array\_size in the optional FOR clause

Of course, the number of rows fetched can be further limited by the number of rows that actually satisfy the query.

If a FETCH statement does not retrieve all rows returned by the query, the cursor is positioned on the next returned row. When the last row returned by the query has been retrieved, the next FETCH statement results in an error code returned in the SQLCODE element of the SQLCA.

The FETCH statement does not contain an AT clause. You must specify the database accessed by the cursor in the DECLARE CURSOR statement.

You can only move forward through the active set with FETCH statements. If you want to revisit any of the previously fetched rows, you must reopen the cursor and fetch each row in turn. If you want to change the active set, you must assign new values to the input host variables in the cursor's query and reopen the cursor.

For more information, see "The FETCH Statement" for the Oracle descriptor.

#### **Example**

This example illustrates the FETCH statement:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT job, sal FROM emp WHERE deptno = 30;
EXEC SQL OPEN emp_cursor;
...
EXEC SQL WHENEVER NOT FOUND GOTO ...
for(;;)
```



```
{
EXEC SQL FETCH emp_cursor INTO :job_title1, :salary1;
...
}
```

# **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE CURSOR (Embedded SQL Directive).

OPEN (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.37 FETCH DESCRIPTOR (Executable Embedded SQL)

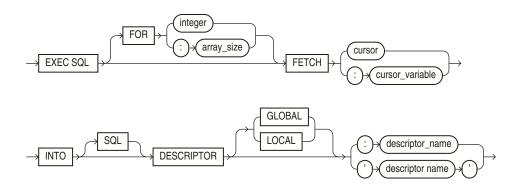
# **Purpose**

To retrieve one or more rows returned by a query, assigning the select list values to host variables. Used in ANSI Dynamic SQL Method 4.

# **Prerequisites**

You must first open the cursor with an the OPEN statement.

# **Syntax**



Keywords and Parameters	Description
array_size	Host variable containing the number of rows to be processed.  Number of rows to be processed.
integer	Limits the number of rows fetched if you are using array host variables. If you omit this clause, Oracle fetches enough rows to fill the smallest array.
cursor	A cursor that has been declared by a DECLARE CURSOR statement. The FETCH statement returns one of the rows selected by the query associated with the cursor.
cursor_variable	A cursor variable is allocated an ALLOCATE statement. The FETCH statement returns one of the rows selected by the query associated with the cursor variable.

Keywords and Parameters	Description
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
INTO	Specifies a list of host variables and optional indicator variables into which data is fetched. These host variables and indicator variables must be declared within the program.
descriptor name'	Name of the output ANSI descriptor.
:descriptor_name	Host variable containing the name of the output descriptor.

The number of rows retrieved is specified by the size of the output host variables and the value specified in the FOR clause. The host variables to receive the data must be either all scalars or all arrays. If they are scalars, Oracle fetches only one row. If they are arrays, Oracle fetches enough rows to fill the arrays.

Array host variables can have different sizes. In this case, the number of rows Oracle fetches is determined by the smaller of the following values:

- The size of the smallest array.
- The value of the :array\_size in the optional FOR clause.

Of course, the number of rows fetched can be further limited by the number of rows that actually satisfy the query.

If a FETCH statement does not retrieve all rows returned by the query, the cursor is positioned on the next returned row. When the last row returned by the query has been retrieved, the next FETCH statement results in an error code returned in the SQLCODE element of the SQLCA.

The FETCH statement does not contain an AT clause. You must specify the database accessed by the cursor in the DECLARE CURSOR statement.

You can only move forward through the active set with FETCH statements. If you want to revisit any of the previously fetched rows, you must reopen the cursor and fetch each row in turn. If you want to change the active set, you must assign new values to the input host variables in the cursor's query and reopen the cursor.

Use DYNAMIC=ANSI precompiler option for the ANSI SQL Method 4 application. For more information, see "FETCH" for the ANSI SQL Method 4 application.

#### **Example**

```
EXEC SQL ALLOCATE DESCRIPTOR 'output_descriptor';
...

EXEC SQL PREPARE S FROM :dyn_statement;

EXEC SQL DECLARE mycursor CURSOR FOR S;
...

EXEC SQL FETCH mycursor INTO DESCRIPTOR 'output_descriptor';
...
```

# **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE CURSOR (Embedded SQL Directive).

OPEN DESCRIPTOR (Executable Embedded SQL).



# PREPARE (Executable Embedded SQL).

# E.38 FREE (Executable Embedded SQL Extension)

# **Purpose**

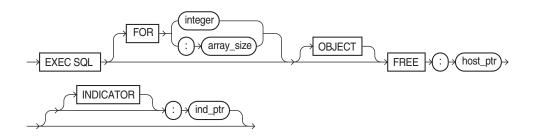
To free memory in the object cache.

# **Prerequisites**

The memory has to have been already allocated.

An active database connection must exist.

# **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
dbname	A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If omitted, or if an empty string, the default database connection is assumed.
host_ptr	A host-variable pointer that was previously ALLOCATED.
ind_ptr	An indicator pointer.

# **Usage Notes**

Any memory in the object cache will be freed automatically when the connection is terminated. See "FREE" for more information.

#### **Example**

EXEC SQL FREE :ptr ;

# **Related Topics**

ALLOCATE (Executable Embedded SQL Extension).

CACHE FREE ALL (Executable Embedded SQL Extension).



# E.39 GET DESCRIPTOR (Executable Embedded SQL)

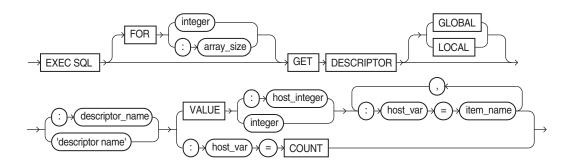
# **Purpose**

To obtain information about host variables from a SQL descriptor area.

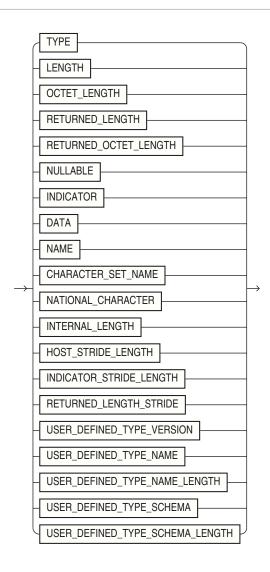
# **Prerequisites**

Use only with value semantics.

# **Syntax**



where item\_name can be one of these choices:



Keywords and Parameters	Description
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed.
descriptor_n ame 'descriptor name'	Host variable containing the name of the allocated ANSI descriptor.  Name of the allocated ANSI descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
host_var = COUNT	Host variable containing the total number of input or output variables.
integer	Total number of input or output variables.
VALUE :host _integer	Host variable containing the position of the referenced input or output variable.

Keywords and Parameters	Description
VALUE integer	The position of the referenced input or output variable.
host_var	Host variable which will receive the item's value.
item_name	The <i>item_name</i> is found in Table 10-4, and Table 10-5, under the "Descriptor Item Name" column heading.

Use DYNAMIC=ANSI precompiler option. The array size clause can be used with DATA, RETURNED\_LENGTH, and INDICATOR item names. See GET DESCRIPTOR.

#### **Example**

EXEC SQL GET DESCRIPTOR GLOBAL 'mydesc' :mydesc\_num\_vars = COUNT ;

# **Related Topics**

ALLOCATE DESCRIPTOR (Executable Embedded SQL).

DEALLOCATE DESCRIPTOR (Embedded SQL Statement).

SET DESCRIPTOR (Executable Embedded SQL).

# E.40 INSERT (Executable Embedded SQL)

# **Purpose**

To add rows to a table or to a view's base table.

# **Prerequisites**

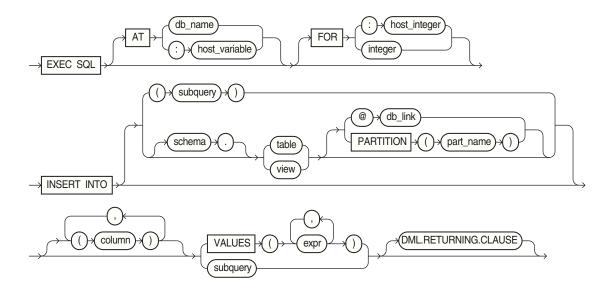
For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.

For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

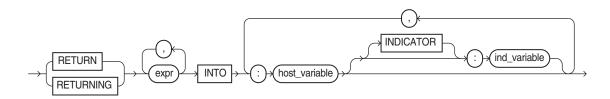
The INSERT ANY TABLE system privilege also provides the ability to insert rows into any table or any view's base table.



# **Syntax**



# Where the DML returning clause is:



Keywords and Parameters	Description
AT	Identifies the database on which the INSERT statement is executed. The database can be identified by either:
	db_name: Is a database identifier declared in a previous DECLARE DATABASE statement.
	<b>host_variable:</b> Is a host variable whose value is a db_name. If you omit this clause, the INSERT statement is executed on your default database.
FOR :host_integer	Limits the number of times the statement is executed if the VALUES clause contains array host variables.
integer	If you omit this clause, Oracle executes the statement once for each component in the smallest array.
schema	The schema containing the table or view. If you omit schema, Oracle assumes the table or view is in your own schema.
table	The name of the table into which rows are to be inserted.
view	If you specify view, Oracle inserts rows into the view's base table.

Keywords and Parameters	Description
db_link	A complete or partial name of a database link to a remote database where the table or view is located.
	You can only insert rows into a remote table or view if you are using Oracle with the distributed option.
	If you omit dblink, Oracle assumes that the table or view is on the local database.
part_name	Is the name of a partition of the table.
column	If you omit one of the table's columns from this list, the column's value for the inserted row is the column's default value as specified when the table was created. If you omit the column list altogether, the VALUES clause or query must specify values for all columns in the table.
VALUES	Specifies a row of values to be inserted into the table or view. See the syntax description in <i>Oracle Database SQL Language Reference</i> . The expressions can be host variables with optional indicator variables. You must specify an expression in the VALUES clause for each column in the column list.
subquery	Is a subquery that returns rows that are inserted into the table. The select list of this subquery must have the same number of columns as the column list of the INSERT statement. For the syntax description of a subquery, see SELECT.
DML returning clause	See "The DML Returning Clause".

Any host variables that appear in the WHERE clause must be either all scalars or all arrays. If they are scalars, Oracle executes the INSERT statement once. If they are arrays, Oracle executes the INSERT statement once for each set of array components, inserting one row each time.

Array host variables in the WHERE clause can have different sizes. In this case, the number of times Oracle executes the statement is determined by the smaller of the following values:

- Size of the smallest array
- The value of the :host\_integer in the optional FOR clause.

For more information on this statement, see "The INSERT Statement".

# **Example I**

This example illustrates the use of the embedded SQL INSERT statement:

```
EXEC SQL
    INSERT INTO emp (ename, empno, sal)
    VALUES (:ename, :empno, :sal);
```

# **Example II**

This example shows an embedded SQL INSERT statement with a subquery:

```
EXEC SQL
  INSERT INTO new_emp (ename, empno, sal)
  SELECT ename, empno, sal FROM emp
  WHERE deptno = :deptno ;
```

# **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive).

# E.41 LOB APPEND (Executable Embedded SQL Extension)

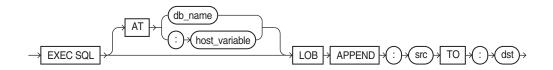
# **Purpose**

To append a LOB to the end of another LOB.

# **Prerequisites**

LOB buffering must not be enabled. The destination LOB must have been initialized.

# **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "APPEND".

# **Related Topics**

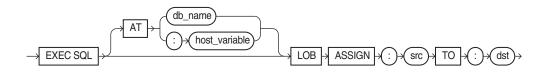
See the other LOB statements.

# E.42 LOB ASSIGN (Executable Embedded SQL Extension)

# **Purpose**

To assign a LOB or BFILE locator to another locator.

#### **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "ASSIGN".

#### **Related Topics**

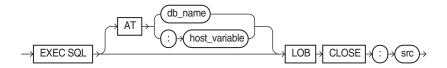
See the other LOB statements.

# E.43 LOB CLOSE (Executable Embedded SQL Extension)

# **Purpose**

To close an open LOB or BFILE.

#### **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "CLOSE (for LOBs)".

# **Related Topics**

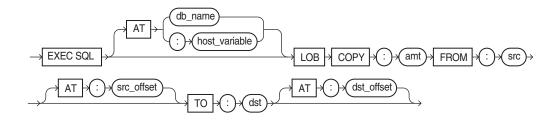
See the other LOB statements.

# E.44 LOB COPY (Executable Embedded SQL Extension)

# **Purpose**

To copy all or part of a LOB value into another LOB.

# **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "COPY".

# **Related Topics**

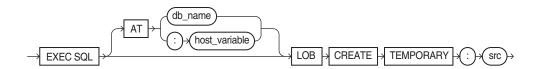
See the other LOB statements.

# E.45 LOB CREATE TEMPORARY (Executable Embedded SQL Extension)

#### **Purpose**

To create a temporary LOB.

#### **Syntax**



### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "CREATE TEMPORARY".

# **Related Topics**

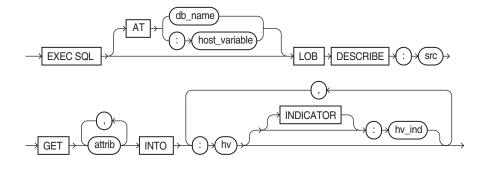
See the other LOB statements.

# E.46 LOB DESCRIBE (Executable Embedded SQL Extension)

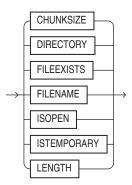
### **Purpose**

To retrieve attributes from a LOB.

# **Syntax**



where attrib is:



## **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "DESCRIBE".

#### **Related Topics**

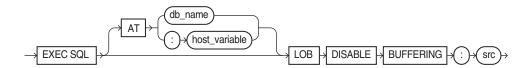
See the other LOB statements.

# E.47 LOB DISABLE BUFFERING (Executable Embedded SQL Extension)

#### **Purpose**

To disable LOB buffering.

## **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "DISABLE BUFFERING".

#### **Related Topics**

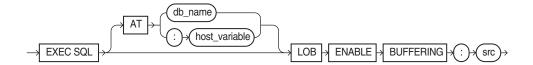
See the other LOB statements.

# E.48 LOB ENABLE BUFFERING (Executable Embedded SQL Extension)

## **Purpose**

To enable LOB buffering.





### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "ENABLE BUFFERING"

### **Related Topics**

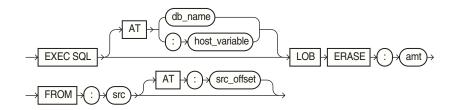
See the other LOB statements.

# E.49 LOB ERASE (Executable Embedded SQL Extension)

### **Purpose**

To erase a given amount of LOB data starting from a given offset.

## **Syntax**



### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "ERASE".

#### **Related Topics**

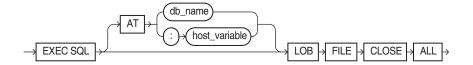
See the other LOB statements.

# E.50 LOB FILE CLOSE ALL (Executable Embedded SQL Extension)

#### **Purpose**

To close all open BFILEs in the current session.





### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "FILE CLOSE ALL".

#### **Related Topics**

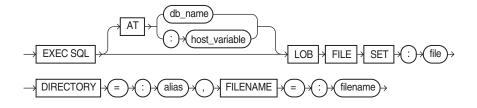
See the other LOB statements.

# E.51 LOB FILE SET (Executable Embedded SQL Extension)

### **Purpose**

To set DIRECTORY and FILENAME in a BFILE locator.

## **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "FILE SET".

### **Related Topics**

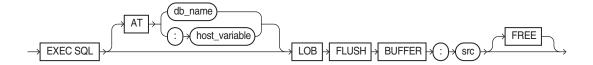
See the other LOB statements.

# E.52 LOB FLUSH BUFFER (Executable Embedded SQL Extension)

### **Purpose**

To write the LOB buffers to the database server.





# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "FLUSH BUFFER".

#### **Related Topics**

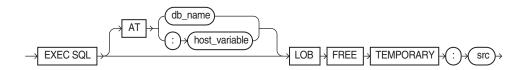
See the other LOB statements.

# E.53 LOB FREE TEMPORARY (Executable Embedded SQL Extension)

### **Purpose**

To free temporary space for the LOB locator.

### **Syntax**



# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "FREE TEMPORARY".

### **Related Topics**

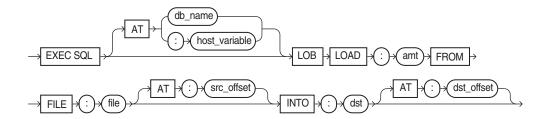
See the other LOB statements.

# E.54 LOB LOAD (Executable Embedded SQL Extension)

#### **Purpose**

To copy all or part of a BFILE into an internal LOB.





#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "LOAD FROM FILE".

### **Related Topics**

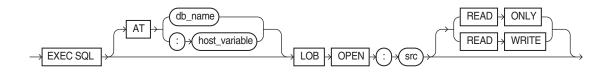
See the other LOB statements.

# E.55 LOB OPEN (Executable Embedded SQL Extension)

# **Purpose**

To open a LOB or BFILE for read or read/write access.

### **Syntax**



### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "OPEN (for LOBs)".

## **Related Topics**

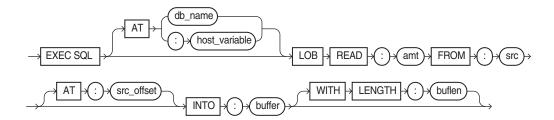
See the other LOB statements.

# E.56 LOB READ (Executable Embedded SQL Extension)

## **Purpose**

To read all or part of a LOB or BFILE into a buffer.





# **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "READ".

# **Related Topics**

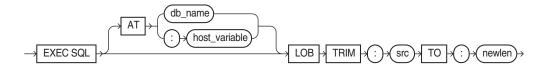
See the other LOB statements.

# E.57 LOB TRIM (Executable Embedded SQL Extension)

### **Purpose**

To truncate a LOB value.

## **Syntax**



#### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "TRIM".

## **Related Topics**

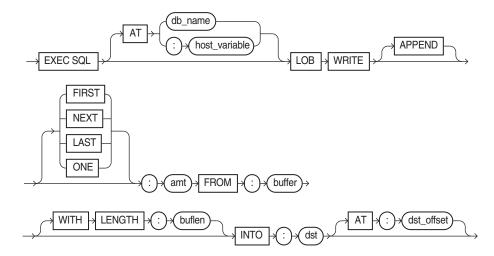
See the other LOB statements.

# E.58 LOB WRITE (Executable Embedded SQL Extension)

#### **Purpose**

To write the contents of a buffer to a LOB.





### **Usage Notes**

For usage notes as well as keywords, parameters, and examples, see "WRITE".

### **Related Topics**

See the other LOB statements.

# E.59 OBJECT CREATE (Executable Embedded SQL Extension)

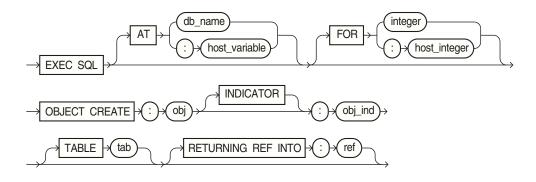
### **Purpose**

To create a referenceable object in the object cache.

#### **Prerequisites**

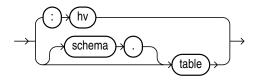
Precompiler option OBJECTS must be set to YES. The INTYPE option must specify the OTT-generated type files. Include OTT-generated header files in your program.

## **Syntax**





#### where tab is:



#### **Usage Notes**

For usage notes as well as keywords and parameters, see "OBJECT CREATE".

#### **Example**

#### **Related Topics**

See all other OBJECT statements in this appendix.

# E.60 OBJECT DELETE (Executable Embedded SQL Extension)

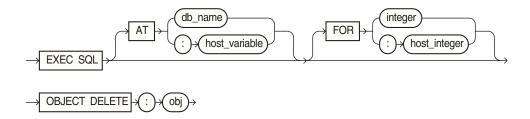
### **Purpose**

To mark a persistent object or array of objects as deleted in the object cache.

#### **Prerequisites**

Precompiler option OBJECTS must be set to YES. The INTYPE option must specify the OTT-generated type files. Include OTT-generated header files in your program.

#### **Syntax**



### **Usage Notes**

For usage notes as well as keywords and parameters, see "OBJECT DELETE".

#### **Example**

```
customer *cust_p;
...
EXEC SQL OBJECT DELETE :cust p;
```

# **Related Topics**

See all other OBJECT statements in this Appendix. For persistent objects, this statement marks an object or array of objects as deleted in the object cache.

# E.61 OBJECT DEREF (Executable Embedded SQL Extension)

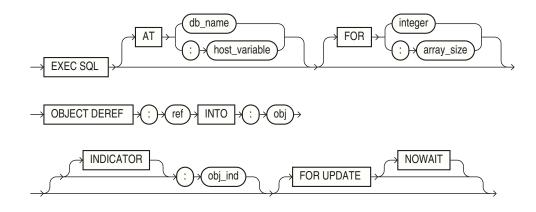
### **Purpose**

To pin an object or array of objects in the object cache.

### **Prerequisites**

Precompiler option OBJECTS must be set to YES. The INTYPE option must specify the OTT-generated type files. Include OTT-generated header files in your program.

# **Syntax**



#### **Usage Notes**

For usage notes as well as keywords and parameters, see "OBJECT DEREF".

#### **Example**

```
person *pers_p;
person_ref *pers_ref;
...
/* Pin the person REF, returning a pointer to the person object */
EXEC SQL OBJECT DEREF :pers ref INTO :pers p;
```

# **Related Topics**

See all other OBJECT statements in this Appendix. See "ALLOCATE (Executable Embedded SQL Extension)".

# E.62 OBJECT FLUSH (Executable Embedded SQL Extension)

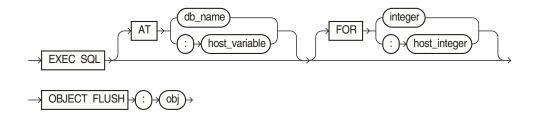
#### **Purpose**

To flush persistent objects that have been marked as updated, deleted, or created, to the server

#### **Prerequisites**

Precompiler option OBJECTS must be set to YES. The INTYPE option must specify the OTT-generated type files. Include OTT-generated header files in your program.

## **Syntax**



# **Usage Notes**

For usage notes as well as keywords and parameters, see "OBJECT FLUSH".

#### **Example**

```
person *pers_p;
...

EXEC SQL OBJECT DELETE :pers_p;
/* Flush the changes, effectively deleting the person object */
EXEC SQL OBJECT FLUSH :pers_p;
/* Finally, free all object cache memory and logoff */
EXEC SQL OBJECT CACHE FREE ALL;
EXEC SQL COMMIT WORK RELEASE;
```

## **Related Topics**

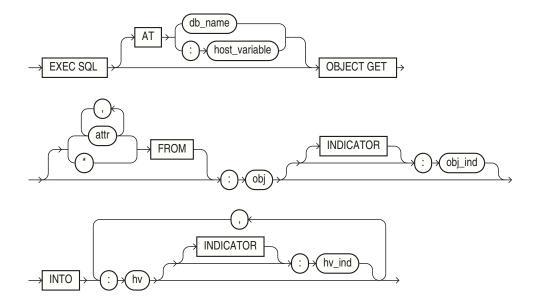
See all other OBJECT statements in this Appendix.

# E.63 OBJECT GET (Executable Embedded SQL Extension)

## **Purpose**

To convert attributes of an object type to native C types

#### **Prerequisites**



#### **Usage Notes**

For usage notes as well as keywords and parameters, see "OBJECT GET".

#### **Example**

### **Related Topics**

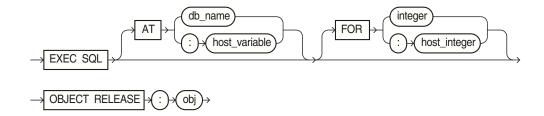
See all other OBJECT statements in this Appendix.

# E.64 OBJECT RELEASE (Executable Embedded SQL Extension)

### **Purpose**

To unpin an object in the object cache. When an object is not pinned and not updated, it is eligible for implicit freeing.

### **Prerequisites**



# **Usage Notes**

For usage notes as well as keywords and parameters, see "OBJECT RELEASE".

#### **Example**

```
person *pers_p;
...
EXEC SQL OBJECT RELEASE :pers p;
```

## **Related Topics**

See all other OBJECT statements in this Appendix.

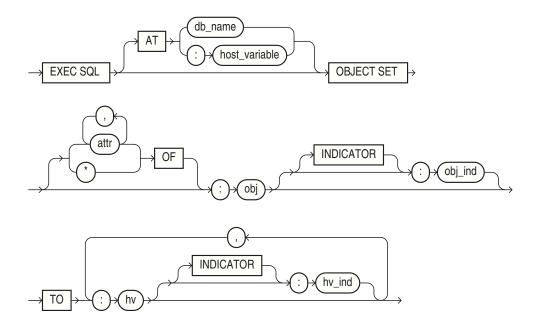
# E.65 OBJECT SET (Executable Embedded SQL Extension)

### **Purpose**

To update attributes of persistent objects, marking them eligible for writing to the server when the object is flushed or the cache is flushed.

To update the attributes of a transient object.

# **Prerequisites**



#### **Usage Notes**

For usage notes as well as keywords and parameters, see "OBJECT FLUSH".

### **Example**

```
person *pers_p;
struct {int num; char street[61], city[31], state[3], zip[11];} addr1;
...
addr1.num = 500;
strcpy((char *)addr1.street , (char *)"Oracle Parkway");
strcpy((char *)addr1.city, (char *)"Redwood Shores");
strcpy((char *)addr1.state, (char *)"CA");
strcpy((char *)addr1.zip, (char *)"94065");

/* Convert native C types to object types */
EXEC SQL OBJECT SET :pers p->addr TO :addr1;
```

#### **Related Topics**

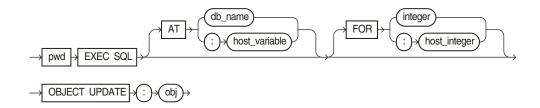
See all other OBJECT statements in this Appendix.

# E.66 OBJECT UPDATE (Executable Embedded SQL Extension)

#### **Purpose**

To mark a persistent object or array of objects as updated in the object cache.

# **Prerequisites**



### **Usage Notes**

For usage notes as well as keywords and parameters, see "OBJECT UPDATE".

### **Example**

```
person *pers_p;
...
/* Mark as updated */
EXEC SQL OBJECT UPDATE :pers p;
```

## **Related Topics**

See all other OBJECT statements in this Appendix.

# E.67 OPEN (Executable Embedded SQL)

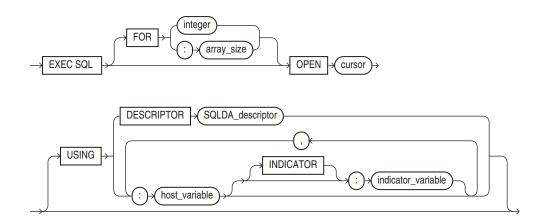
### **Purpose**

To open a cursor, evaluating the associated query and substituting the host variable names supplied by the USING clause into the WHERE clause of the query. For the ANSI Dynamic SQL Method 4 version, see "OPEN DESCRIPTOR (Executable Embedded SQL)".

### **Prerequisites**

You must declare the cursor with a DECLARE CURSOR embedded SQL statement before opening it.

### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed.
cursor	The (previously declared) cursor to be opened.
host_variable	Specifies a host variable with an optional indicator variable to be substituted into the statement associated with the cursor.  CANNOT be used together with an ANSI descriptor (INTO clause).
DESCRIPTOR SQLDA_descriptor	Specifies an Oracle descriptor that describes the host variables to be substituted into the WHERE clause of the associated query. The descriptor must be initialized in a previous DESCRIBE statement. The substitution is based on position. The host variable names specified in this statement can be different from the variable names in the associated query.  CANNOT be used together with an ANSI descriptor (INTO clause).

### **Usage Notes**

The OPEN statement defines the active set of rows and initializes the cursor just before the first row of the active set. The values of the host variables at the time of the OPEN are substituted in the statement. This statement does not actually retrieve rows; rows are retrieved by the FETCH statement.

Once you have opened a cursor, its input host variables are not reexamined until you reopen the cursor. To change any input host variables and therefore the active set, you must reopen the cursor.

All cursors in a program are in a closed state when the program is initiated or when they have been explicitly closed using the CLOSE statement.

You can reopen a cursor without first closing it. For more information on this statement, see "The INSERT Statement".

### **Example**

This example illustrates the use of the OPEN statement in a Pro\*C/C++ program:

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, empno, job, sal
    FROM emp
    WHERE deptno = :deptno;
EXEC SQL OPEN emp_cursor;
```

#### **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE STATEMENT (Embedded SQL Directive).

FETCH (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).



# E.68 OPEN DESCRIPTOR (Executable Embedded SQL)

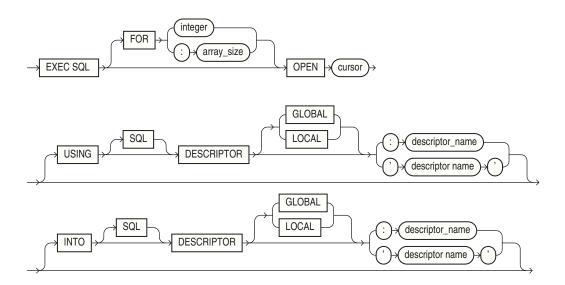
## **Purpose**

To open a cursor (for ANSI Dynamic SQL Method 4), evaluating the associated query and substituting the input host variable names supplied by the USING clause into the WHERE clause of the query. The INTO clause denotes the output descriptor.

## **Prerequisites**

You must declare the cursor with a DECLARE CURSOR embedded SQL statement before opening it.

#### **Syntax**



# **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed.
cursor	The (previously declared) cursor to be opened.
USING DESCRIPTOR	Specifies an ANSI input descriptor.
descriptor_name	The host variable containing the name of the ANSI descriptor,
'descriptor name'	The name of the ANSI descriptor.
INTO DESCRIPTOR	Specifies an ANSI output descriptor.
descriptor_name	The host variable containing the name of the ANSI descriptor,
'descriptor name'	The name of the ANSI descriptor.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.



#### **Usage Notes**

Set the precompiler option DYNAMIC to ANSI.

The OPEN statement defines the active set of rows and initializes the cursor just before the first row of the active set. The values of the host variables at the time of the OPEN are substituted in the SQL statement. This statement does not actually retrieve rows; rows are retrieved by the FETCH statement.

Once you have opened a cursor, its input host variables are not reexamined until you reopen the cursor. To change any input host variables and therefore the active set, you must reopen the cursor.

All cursors in a program are in a closed state when the program is initiated or when they have been explicitly closed using the CLOSE statement.

You can reopen a cursor without first closing it. For more information on this statement, see "The INSERT Statement".

#### **Example**

```
char dyn_statement[1024] ;
...
EXEC SQL ALLOCATE DESCRIPTOR 'input_descriptor' ;
EXEC SQL ALLOCATE DESCRIPTOR 'output descriptor'
...
EXEC SQL PREPARE S FROM :dyn_statement ;
EXEC SQL DECLARE C CURSOR FOR S ;
...
EXEC SQL OPEN C USING DESCRIPTOR 'input_descriptor' ;
...
```

#### **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE CURSOR (Embedded SQL Directive).

FETCH DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.69 PREPARE (Executable Embedded SQL)

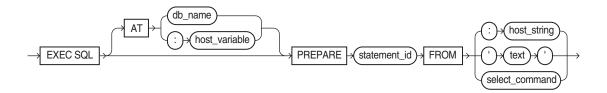
## **Purpose**

To parse a SQL statement or PL/SQL block specified by a host variable and associate it with an identifier.

#### **Prerequisites**

None.





#### **Keywords and Parameters**

Keywords and Parameters	Description
statement_id	Is the identifier to be associated with the prepared SQL statement or PL/SQL block. If this identifier was previously assigned to another statement or block, the prior assignment is superseded.
db_name	A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If it is omitted, or if it is an empty string, the default database connection is assumed.
host_variable	A host variable containing the name of the database connection.
text	Is a string literal containing a SQL statement or PL/SQL block to be prepared.
select_command	Is a select statement

### **Usage Notes**

Any variables that appear in the *host\_string* or *text* are placeholders. The actual host variable names are assigned in the USING clause of the OPEN statement (input host variables) or in the INTO clause of the FETCH statement (output host variables).

A SQL statement is prepared only once, but can be executed any number of times. For more information, see "PREPARE (Dynamic SQL)".

#### **Example**

This example illustrates the use of a PREPARE statement in a Pro\*C/C++ embedded SQL program:

```
EXEC SQL PREPARE my_statement FROM :my_string; EXEC SQL EXECUTE my_statement;
```

# **Related Topics**

CLOSE (Executable Embedded SQL).

DECLARE CURSOR (Embedded SQL Directive).

FETCH DESCRIPTOR (Executable Embedded SQL).

OPEN (Executable Embedded SQL).

# E.70 REGISTER CONNECT (Executable Embedded SQL Extension)

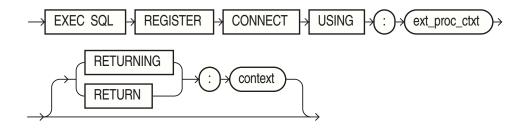
## **Purpose**

To allow an external C procedure to be called from a Pro\*C/C++ application.

### **Prerequisites**

None.

#### **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
ext_proc_ctxt	The external procedure context passed to the procedure by PL/SQL. It is of type pointer to OCIExtProcContext.
context	The runtime context returned. It is of type sql_context. Currently, it is the default (global) context

### **Usage Notes**

For a complete discussion of how to write an external procedure, and the restrictions that are in effect, see External Procedures.

#### **Example**

```
void myfunction(epctx)
OCIExtProcContext *epctx;
sql_context context;
...
{
EXEC SQL REGISTER CONNECT USING :epctx;
EXEC SQL USE :context;
```

### **Related Topics**

None



# E.71 ROLLBACK (Executable Embedded SQL)

## **Purpose**

To undo work done in the current transaction.

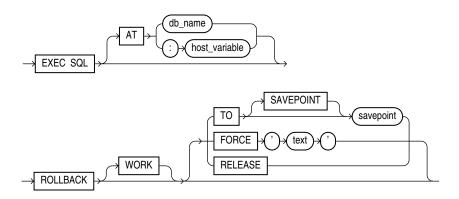
You can also use this statement to manually undo the work done by an in-doubt distributed transaction.

### **Prerequisites**

To roll back your current transaction, no privileges are necessary.

To manually roll back an in-doubt distributed transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

### **Syntax**



### **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database on which the savepoint is created. The database can be identified by either:
	db_name: A null-terminated string containing the database connection name, as established previously in a CONNECT statement. If it is omitted, or if it is an empty string, the default database connection is assumed.
	host_variable: A host variable containing the name of the database connection. If you omit this clause, the savepoint is created on your default database.
WORK	Is optional and is provided for ANSI compatibility.
ТО	Rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction.



Keywords and Parameters	Description
FORCE	Manually rolls back an in-doubt distributed transaction. The transaction is identified by the text containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING.
	ROLLBACK statements with the FORCE clause are not supported in PL/SQL.
RELEASE	Frees all resources and disconnects the application from the database. The RELEASE clause is not allowed with SAVEPOINT and FORCE clauses.
savepoint	savepoint to which you roll back

#### **Usage Notes**

A transaction (or a logical unit of work) is a sequence of SQL statements that Oracle treats as a single unit. A transaction begins with the first executable SQL statement after a COMMIT, ROLLBACK or connection to the database. A transaction ends with a COMMIT statement, a ROLLBACK statement, or disconnection (intentional or unintentional) from the database. Oracle issues an implicit COMMIT statement before and after processing any data definition language statement.

Using the ROLLBACK statement without the TO SAVEPOINT clause performs the following operations:

- Ends the transaction
- Undoes all changes in the current transaction
- Erases all savepoints in the transaction
- Releases the transaction's locks

Using the ROLLBACK statement with the TO SAVEPOINT clause performs the following operations:

- Rolls back just the portion of the transaction after the savepoint.
- Loses all savepoints created after that savepoint. The named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- Releases all table and row locks acquired since the savepoint. Other transactions that
  have requested access to rows locked after the savepoint must continue to wait until the
  transaction is committed or rolled back. Other transactions that have not already requested
  the rows can request and access the rows immediately.

It is recommended that you explicitly end transactions in application programs using either a COMMIT or ROLLBACK statement. If you do not explicitly commit the transaction and the program terminates abnormally, Oracle rolls back the last uncommitted transaction.

#### Example 1

The following statement rolls back your entire current transaction:

EXEC SQL ROLLBACK;

## Example 2

The following statement rolls back your current transaction to savepoint SP5:

EXEC SQL ROLLBACK TO SAVEPOINT sp5;



#### **Distributed Transactions**

Oracle with the distributed option provides the ability to perform distributed transactions, or transactions that modify data on multiple databases. To commit or roll back a distributed transaction, you need only issue a COMMIT or ROLLBACK statement as you would any other transaction.

If there is a network failure during the commit process for a distributed transaction, the state of the transaction may be unknown, or in-doubt. After consultation with the administrators of the other databases involved in the transaction, you may decide to manually commit or roll back the transaction on your local database. You can manually roll back the transaction on your local database by issuing a ROLLBACK statement with the FORCE clause.

You cannot manually roll back an in-doubt transaction to a savepoint.

A ROLLBACK statement with a FORCE clause only rolls back the specified transaction. Such a statement does not affect your current transaction.

#### **Example III**

The following statement manually rolls back an in-doubt distributed transaction:

```
EXEC SQL

ROLLBACK WORK

FORCE '25.32.87';
```

## **Related Topics**

COMMIT (Executable Embedded SQL).

SAVEPOINT (Executable Embedded SQL).

# E.72 SAVEPOINT (Executable Embedded SQL)

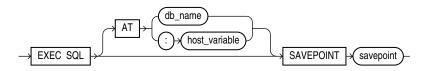
#### **Purpose**

To identify a point in a transaction to which you can later roll back.

#### **Prerequisites**

None.

#### **Syntax**





#### **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
AT	Identifies the database on which the savepoint is created. The database can be identified by either:
	db_name: Is a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: Is a host variable whose value is a previously declared db_name.
	If you omit this clause, the savepoint is created on your default database.
savepoint	Is the name of the savepoint to be created.

### **Usage Notes**

For more information on this statement, see "Using the SAVEPOINT Statement".

#### **Example**

This example illustrates the use of the embedded SQL SAVEPOINT statement:

EXEC SOL SAVEPOINT save3;

#### **Related Topics**

COMMIT (Executable Embedded SQL).

ROLLBACK (Executable Embedded SQL) .

# E.73 SELECT (Executable Embedded SQL)

# **Purpose**

To retrieve data from one or more tables, views, or snapshots, assigning the selected values to host variables.

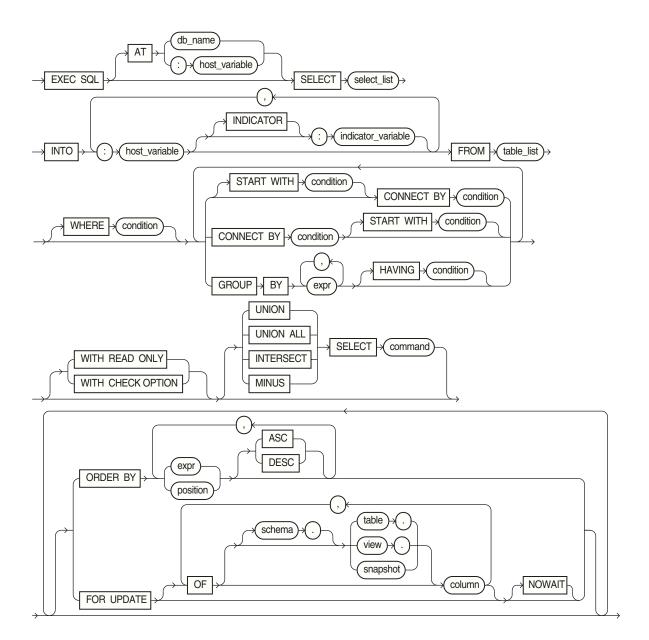
### **Prerequisites**

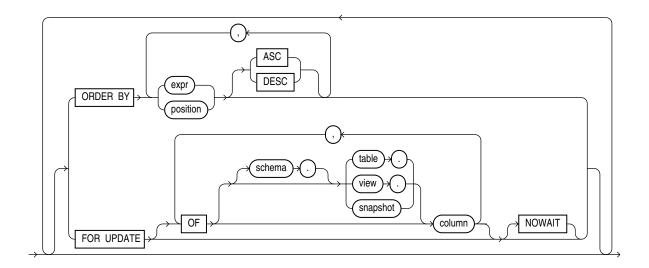
For you to select data from a table or snapshot, the table or snapshot must be in your own schema or you must have SELECT privilege on the table or snapshot.

For you to select rows from the base tables of a view, the owner of the schema containing the view must have SELECT privilege on the base tables. Also, if the view is in a schema other than your own, you must have SELECT privilege on the view.

The SELECT ANY TABLE system privilege also provides the ability to select data from any table or any snapshot or any view's base table.







### **Keywords and Parameters**

<b>Keywords and Parameters</b>	Description
AT	Identifies the database to which the SELECT statement is issued. The database can be identified by either:
	db_name: A database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: Host variable whose value is a previously declared db_name. If you omit this clause, the SELECT statement is issued to your default database.
select_list	Identical to the non-embedded SELECT statement except that a host variables can be used in place of literals.
INTO	Specifies output host variables and optional indicator variables to receive the data returned by the SELECT statement. These variables must be either all scalars or all arrays, but arrays need not have the same size.
WHERE	Restricts the rows returned to those for which the condition is TRUE. See the syntax description of condition in Conditions. The condition can contain host variables, but cannot contain indicator variables. These host variables should be scalars and cannot be arrays.
	All other keywords and parameters are identical to the non-embedded SQL SELECT statement. ASC, ascending, is the default for the ORDER BY clause.

# **Usage Notes**

If no rows meet the WHERE clause condition, no rows are retrieved and Oracle returns an error code through the SQLCODE component of the SQLCA.

You can use Comments in a SELECT statement to pass instructions, or *hints*, to the optimizer.

# Example

This example illustrates the use of the embedded SQL SELECT statement:

```
EXEC SQL SELECT ename, sal + 100, job
    INTO :ename, :sal, :job
```

```
FROM emp
WHERE empno = :empno;
```

## **Related Topics**

DECLARE CURSOR (Embedded SQL Directive).

DECLARE DATABASE (Oracle Embedded SQL Directive).

EXECUTE (Executable Embedded SQL).

FETCH (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.74 SET DESCRIPTOR (Executable Embedded SQL)

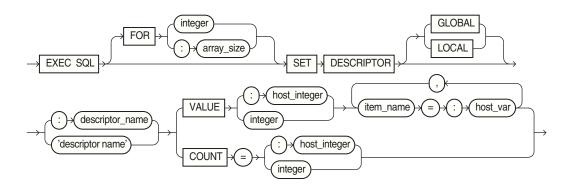
### **Purpose**

Use this ANSI dynamic SQL statement to set information in the descriptor area from host variables.

## **Prerequisites**

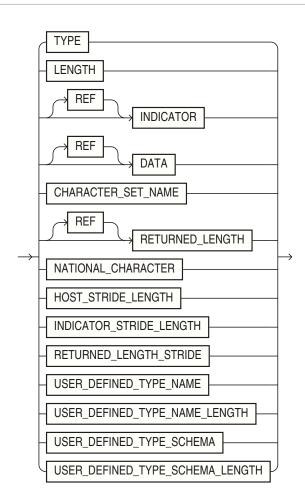
Use after a DESCRIBE DESCRIPTOR statement.

### **Syntax**



where item name can be one of these choices:

where item name can be one of these choices:



# **Keywords and Parameters**

Kewords and Parameters	Description
array_size	Host variable containing the number of rows to be processed.
integer	Number of rows to be processed. The array size clause can only be used with DATA, RETURNED_LENGTH and INDICATOR item names.
GLOBAL   LOCAL	LOCAL (the default) means file scope, as opposed to GLOBAL, which means application scope.
descriptor_name	Host variable containing the name of the allocated ANSI descriptor.
'descriptor name'	Name of the allocated ANSI descriptor.
COUNT	The number of input or output variables.
VALUE	The position of the referenced host variable.
item_name	See Table 10-6 and Table 10-7 for lists of the <i>item_names</i> , and their descriptions.
host_integer	The host variables used to set the item or the COUNT or VALUE.
integer	An integer used to set the COUNT or VALUE.
host_var	The host variables used to set the descriptor item.
REF	Reference semantics are to be used. Can be used only with RETURNED_LENGTH, DATA, and INDICATOR item names.
	Must be used to set RETURNED_LENGTH.

#### **Usage Notes**

Use DYNAMIC=ANSI precompiler option.

Set CHARACTER\_SET\_NAME to UTF16 for client-side Unicode support.

See "SET DESCRIPTOR" for complete details, including tables of descriptor item names.

#### **Example**

EXEC SQL SET DESCRIPTOR GLOBAL :mydescr COUNT = 3 ;

#### **Related Topics**

ALLOCATE DESCRIPTOR (Executable Embedded SQL).

DEALLOCATE DESCRIPTOR (Embedded SQL Statement).

DESCRIBE (Executable Embedded SQL Extension).

GET DESCRIPTOR (Executable Embedded SQL).

PREPARE (Executable Embedded SQL).

# E.75 TYPE (Oracle Embedded SQL Directive)

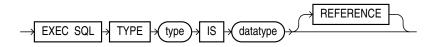
#### **Purpose**

To perform *user-defined type equivalencing*, or to assign an external datatype to a whole class of host variables by equivalencing the external datatype to a user-defined datatype.

#### **Prerequisites**

The user-defined datatype must be previously declared in an embedded SQL program.

#### **Syntax**



#### **Keywords and Parameters**

Keywords and Parameters	Description
type	Is the user-defined datatype to be equivalenced with an external datatype.
datatype	Is an external datatype recognized by the precompiler (not an internal datatype). The datatype may include a length, precision, or scale. This external datatype is equivalenced to the user-defined type and assigned to all host variables assigned the type. For a list of external datatypes, see "Oracle Datatypes".
REFERENCE	Makes the equivalenced type a pointer type.



#### **Usage Notes**

User-defined type equivalencing is one kind of datatype equivalencing. You can only perform user-defined type equivalencing with the embedded SQL TYPE statement in a Pro\*C/C++ program. You may want to use datatype equivalencing for one of the following purposes:

- To automatically null-terminate a character host variable
- To store program data as binary data in the database
- To override default datatype conversion

Pro\*C/C++ expects VARCHAR and VARRAW arrays to be word-aligned. When you equivalence an array type to the VARCHAR or VARRAW datatype, make sure that length+2 is divisible by 4.

Pro\*C/C++ also supports the embedded SQL VAR statement for host variable equivalencing. For more information, see "User-Defined Type Equivalencing".

#### Example I

This example shows an embedded SQL TYPE statement in a Pro\*C/C++ Precompiler program:

```
struct screen {
    short len;
    char buff[4002];
};

typedef struct screen graphics;

EXEC SQL TYPE graphics IS VARRAW(4002);
    graphics crt; -- host variable of type graphics
...
```

#### **Related Topics**

VAR (Oracle Embedded SQL Directive) .

# E.76 UPDATE (Executable Embedded SQL)

#### **Purpose**

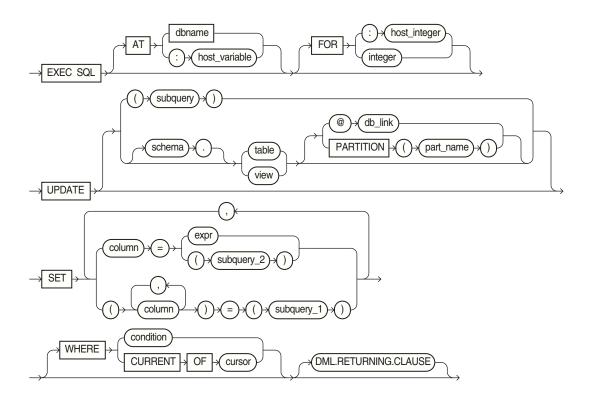
To change existing values in a table or in a view's base table.

#### **Prerequisites**

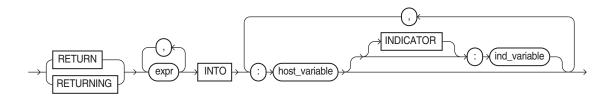
For you to update values in a table or snapshot, the table must be in your own schema or you must have UPDATE privilege on the table.

For you to update values in the base table of a view, the owner of the schema containing the view must have UPDATE privilege on the base table. Also, if the view is in a schema other than your own, you must have UPDATE privilege on the view.

The UPDATE ANY TABLE system privilege also provides the ability to update values in any table or any view's base table.



# where the DML returning clause is:



# **Keywords and Parameters**

Keywords and Parameters	Description
AT	Identifies the database to which the UPDATE statement is issued. The database can be identified by either:
	dbname: Is a database identifier declared in a previous DECLARE DATABASE statement.
	host_variable: Is a host variable whose value is a previously declared db_name.
	If you omit this clause, the UPDATE statement is issued to your default database.

<b>Keywords and Parameters</b>	Description
FOR :host_integer integer	Limits the number of times the UPDATE statement is executed if the SET and WHERE clauses contain array host variables. If you omit this clause, Oracle executes the statement once for each component of the smallest array.
schema	Is the schema containing the table or view. If you omit schema, Oracle assumes the table or view is in your own schema.
table, view	Is the name of the table to be updated. If you specify view, Oracle updates the view's base table.
dblink	Is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see the <i>Oracle Database SQL Language Reference</i> You can only use a database link to update a remote table or view if you are using Oracle with the distributed option.
part_name	Name of a partition of the table.
column	Is the name of a column of the table or view that is to be updated. If you omit a column of the table from the SET clause, that column's value remains unchanged.
expr	Is the new value assigned to the corresponding column. This expression can contain host variables and optional indicator variables.
subquery_1	Is a subquery that returns new values that are assigned to the corresponding columns. For the syntax of a subquery, see SELECT.
subquery_2	Is a subquery that return a new value that is assigned to the corresponding column. For the syntax of a subquery, see SELECT.
WHERE	Specifies which rows of the table or view are updated:
condition	Updates only rows for which this condition is true. This condition can contain host variables and optional indicator variables. See the syntax of condition in Conditions.
CURRENT OF	Updates only the row most recently fetched by the cursor. The cursor cannot be associated with a SELECT statement that performs a join unless its FOR UPDATE clause explicitly locks only one table.
	If you omit this clause entirely, all rows of the table or view are updated.
DML returning clause	See "The DML Returning Clause".

#### **Usage Notes**

Host variables in the SET and WHERE clauses must be either all scalars or all arrays. If they are scalars, Oracle executes the UPDATE statement only once. If they are arrays, Oracle executes the statement once for each set of array components. Each execution may update zero, one, or multiple rows.

Array host variables can have different sizes. In this case, the number of times Oracle executes the statement is determined by the smaller of the following values:

- The size of the smallest array
- The value of the :host\_integer in the optional FOR clause

The cumulative number of rows updated is returned through the third element of the SQLERRD component of the SQLCA. When arrays are used as input host variables, this count reflects the total number of updates for all components of the array processed in the UPDATE statement. If no rows satisfy the condition, no rows are updated and Oracle returns an error message through the SQLCODE element of the SQLCA. If you omit the WHERE clause, all

rows are updated and Oracle raises a warning flag in the fifth component of the SQLWARN element of the SQLCA.

You can use Comments in an UPDATE statement to pass instructions, or *hints*, to the optimizer. The optimizer uses hints to choose an execution plan for the statement.

For more information on this statement, see Embedded SQL, and Database Concepts.

#### **Examples**

The following examples illustrate the use of the embedded SQL UPDATE statement:

```
EXEC SQL UPDATE emp
   SET sal = :sal, comm = :comm INDICATOR :comm_ind
   WHERE ename = :ename;

EXEC SQL UPDATE emp
   SET (sal, comm) =
        (SELECT AVG(sal)*1.1, AVG(comm)*1.1
        FROM emp)
   WHERE ename = 'JONES';
```

#### **Related Topics**

DECLARE DATABASE (Oracle Embedded SQL Directive).

# E.77 VAR (Oracle Embedded SQL Directive)

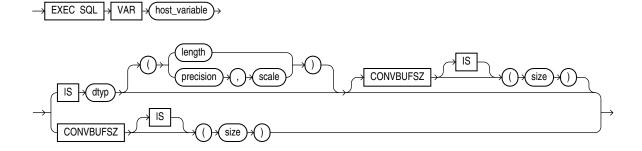
# **Purpose**

To perform *host variable equivalencing*, or to assign a specific external datatype to an individual host variable, overriding the default datatype assignment. Also has an optional CONVBUFSZ clause that specifies the size of a buffer for character set conversion.

#### **Prerequisites**

The host variable must be previously declared in the Pro\*C/C++ program.

#### **Syntax**



# **Keywords and Parameters**

Keywords and Parameters	Description
host_variable	Is an input or output host variable (or host table) declared earlier.  The VARCHAR and VARRAW external datatypes have a 2-byte length field followed by an <i>n</i> -byte data field, where <i>n</i> lies in the range 1 65533. So, if <i>type_name</i> is VARCHAR or VARRAW, <i>host_variable</i> must be at least 3 bytes long.
	The LONG VARCHAR and LONG VARRAW external datatypes have a 4-byte length field followed by an <i>n</i> -byte data field, where <i>n</i> lies in the range 1 2147483643. So, if <i>type_name</i> is LONG VARCHAR or LONG VARRAW, <i>host_variable</i> must be at least 5 bytes long.
dtyp	Is an external datatype recognized by Pro*C/C++ (not an internal datatype). The datatype may include a length, precision, or scale. This external datatype is assigned to the host_variable. For a list of external datatypes, see "External Datatypes".
length	Length of the datatype. It is a constant expression or a constant integer specifying a valid length in bytes. The value of length must be large enough to accommodate the external datatype.
	When <i>type_name</i> is ROWID or DATE, you cannot specify <i>length</i> because it is predefined. For other external datatypes, <i>length</i> is optional. It defaults to the length of <i>host_variable</i> .
	When specifying <i>length</i> , if <i>type_name</i> is VARCHAR, VARRAW, LONG VARCHAR, or LONG VARRAW, use the maximum length of the data field. Pro*C/C++ accounts for the length field. If <i>type_name</i> is LONG VARCHAR or LONG VARRAW and the data field exceeds 65533 bytes, put "-1" in the <i>length</i> field.
precision and scale	Are constant expressions or constants that represent, respectively, the number of significant digits and the point at which rounding will occur. For example, a scale of 2 means the value is rounded to the nearest hundredth (3.456 becomes 3.46); a scale of -3 means the number is rounded to the nearest thousand (3456 becomes 3000).
	You can specify a <i>precision</i> of 1 99 and a <i>scale</i> of -84 99. However, the maximum precision and scale of a database column are 38 and 127, respectively. So, if <i>precision</i> exceeds 38, you cannot insert the value of <i>host_variable</i> into a database column. On the other hand, if the scale of a column value exceeds 99, you cannot select or fetch the value into <i>host_variable</i> .
size	The size, in bytes, of a buffer used to perform conversion of the specified host_variable to another character set. A constant or constant expression.
	Is the size in bytes of a buffer in the runtime library used to perform conversion between character sets of the host_variable

# **Usage Notes**

length, precision, scale and size can be constant expressions.

Host variable equivalencing is one kind of datatype equivalencing. Datatype equivalencing is useful for any of the following purposes:

- To automatically null-terminate a character host variable
- To store program data as binary data in the database
- To override default datatype conversion



Note that *size*, *length*, *precision* and *scale* can be any arbitrarily complex C constant expression whose value is known when the precompiler is run.

#### For example:

```
#define LENGTH 10
...
char character set is nchar_cs ename[LENGTH+1];
exec sql var ename is string(LENGTH+1) convbufsz is (LENGTH*2);
```

Note also that macros are permitted in this statement.

When you have not used the CONVBUFSZ clause, the Oracle runtime automatically determines a buffer size based on the ratio of the host variable character size (determined by NLS\_LANG) and the character size of the database character set. This can sometimes result in the creation of a buffer of LONG size. Database tables are allowed to have only one LONG column. An error is raised if there is more than one LONG value.

To avoid such errors, you use a length shorter than the size of a LONG. If a character set conversion results in a value longer than the length specified by CONVBUFSZ, then an error is returned at runtime. The Pro\*C/C++ Precompiler also supports the precompiler TYPE directive for user-defined type equivalencing. See also "Host Variable Equivalencing".

#### **Example**

This example equivalences the host variable DEPT\_NAME to the datatype STRING and the host variable BUFFER to the datatype RAW(200):

```
EXEC SQL BEGIN DECLARE SECTION;

...

char dept_name[15]; -- default datatype is CHAR

EXEC SQL VAR dept_name IS STRING; -- reset to STRING

...

char buffer[200]; -- default datatype is CHAR

EXEC SQL VAR buffer IS RAW(200); -- refer to RAW

...

EXEC SQL END DECLARE SECTION;
```

### **Related Topics**

TYPE (Oracle Embedded SQL Directive) .

# E.78 WHENEVER (Embedded SQL Directive)

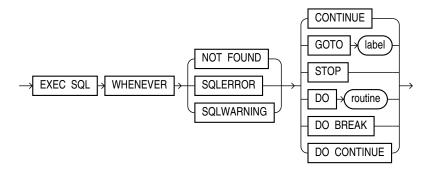
## **Purpose**

To specify the action to be taken when an error or warning results from executing an embedded SQL statement.

## **Prerequisites**

None.





## **Keywords and Parameters**

Keywords and Parameters	Description
NOT FOUND	Identifies any exception condition that returns an error code of +1403 to SQLCODE (or a +100 code when MODE=ANSI).
SQLERROR	Identifies a condition that results in a negative return code.
SQLWARNING	Identifies a non-fatal warning condition.
CONTINUE	Indicates that the program should progress to the next statement.
GOTO label	Indicates that the program should branch to the statement named by <i>label</i> .
STOP	Stops program execution.
DO routine	Indicates that the program should call a function that is named routine
DO BREAK	Performs a break statement from a loop when the condition is met.
DO CONTINUE	Performs a continue statement from a loop when the condition is met.

# **Usage Notes**

The WHENEVER directive allows your program to transfer control to an error handling routine in the event an embedded SQ

L statement results in an error or warning.

The scope of a WHENEVER directive is positional, rather than logical. A WHENEVER statement applies to all embedded SQL statements that textually follow it in the source file, not in the flow of the program logic. A WHENEVER directive remains in effect until it is superseded by another WHENEVER directive checking for the same condition.

For more information on this directive, see "About Using the WHENEVER Directive".

Do not confuse the WHENEVER embedded SQL directive with the WHENEVER SQL\*Plus command.

### **Examples**

The following two examples illustrates the uses of the WHENEVER directive in an embedded SQL program:

Example 1:



```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
...

EXEC SQL WHENEVER SQLERROR GOTO sql_error;
...

sql_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
...

EXAMPLE 2:

EXEC SQL WHENEVER SQLERROR GOTO connect_error;
...

connect_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    printf("\nInvalid username/password\n");
    exit(1);
```

#### **Related Topics**

None

F

# Sample Programs

This chapter describes how to build Oracle database applications with Pro\*C/C++ using the sample programs that are included with this release.

This chapter contains the following topics:

- Sample Program Descriptions
- Building the Demonstration Tables
- About Building the Sample Programs

# F.1 Sample Program Descriptions

When you install Pro\*C/C++, Oracle Universal Installer copies a set of Pro\*C/C++ sample programs to the  $ORACLE\_BASE \setminus ORACLE\_HOME \setminus precomp \setminus demo \setminus proc directory$ . These sample programs are listed in Table F-1 and described in the subsequent section.

When built, the sample programs that Oracle provides produce .exe executables.

For some sample programs, as indicated in the Notes column of the table, you must run the SQL scripts in the sample directory before you precompile and run the sample program. The SQL scripts set up the correct tables and data so that the sample programs run correctly.

These SQL scripts are located in the ORACLE BASE\ORACLE HOME\precomp\demo\sql directory.

Oracle Corporation recommends that you build and run these sample programs to verify that Pro\*C/C++ has been installed successfully and operates correctly. You can delete the programs after you use them.

You can build the sample program by using a batch file called pcmake.bat or by using Microsoft Visual Studio.

See Also:

"About Building the Sample Programs"

**Table F-1** Sample Programs

Sample Program	Source Files	Pro*C/C++ GUI Project File	MSVC Compiler Project File	Notes
ANSIDYN1	ansidyn1.pc	ansidyn1.pre	ansidyn1.dsp	-
ANSIDYN2	ansidyn2.pc	ansidyn2.pre	ansidyn2.dsp	-
COLDEMO1	<pre>coldemo1.h coldemo1.pc coldemo1.sql coldemo1.typ</pre>	coldemo1.pre	coldemo1.dsp	Run coldemol.sql and the Object Type Translator before building coldemol.
CPDEMO1	cpdemo1.pc	cpdemo1.pre	cpdemo1.dsp	-



Table F-1 (Cont.) Sample Programs

Sample Program	Source Files	Pro*C/C++ GUI Project File	MSVC Compiler Project File	Notes
CPDEMO2	cpdemo2.pc	cpdemo2.pre	cpdemo2.dsp	-
CPPDEMO1	cppdemo1.pc	cppdemo1.pre	cppdemo1.dsp	-
CPPDEMO2	<pre>cppdemo2.pc empclass.pc cppdemo2.sql empclass.h</pre>	cppdemo2.pre	cppdemo2.dsp	Run cppdemo2.sql before building cppdemo2.
CPPDEMO3	cppdemo3.pc	cppdemo3.pre	cppdemo3.dsp	-
CVDEMO	<pre>cv_demo.pc cv_demo.sql</pre>	cv_demo.pre	cv_demo.dsp	Run cv_demo.sql before building cv_demo.
EMPCLASS	<pre>cppdemo2.pc empclass.pc cppdemo2.sql empclass.h</pre>	empclass.pre	empclass.dsp	Run cppdemo2.sql before building empclass.
LOBDEMO1	lobdemo1.h lobdemo1.pc lobdemo1.sql	lobdemo1.pre	lobdemo1.dsp	Run lobdemo1.sql before building lobdemo1.
MLTTHRD1	mltthrd1.pc mltthrd1.sql	mltthrd1.pre	mltthrd1.dsp	Run mltthrdl.sql before building mltthrdl.
NAVDEMO1	navdemol.h navdemol.pc navdemol.sql navdemol.typ	navdemo1.pre	navdemo1.dsp	Run navdemo1.sql and the Object Type Translator before building navdemo1.
OBJDEMO1	objdemo1.h objdemo1.pc objdemo1.sql objdemo1.typ	objdemo1.pre	objdemo1.dsp	Run objdemo1.sql and the Object Type Translator before building objdemo1.
ORACA	oraca.pc oracatst.sql	oraca.pre	oraca.dsp	Run oracatst.sql before building oraca
PLSSAM	plssam.pc	plssam.pre	plssam.dsp	-
SAMPLE	sample.pc	sample.pre	sample.dsp	-
SAMPLE1	sample1.pc	sample1.pre	sample1.dsp	-
SAMPLE2	sample2.pc	sample2.pre	sample2.dsp	-
SAMPLE3	sample3.pc	sample3.pre	sample3.dsp	-
SAMPLE4	sample4.pc	sample4.pre	sample4.dsp	-
SAMPLE5	<pre>sample5.pc exampbld.sql examplod.sql</pre>	sample5.pre	sample5.dsp	Run exampbld.sql, then run examplod.sql, befo building sample5.
SAMPLE6	sample6.pc	sample6.pre	sample6.dsp	-
SAMPLE7	sample7.pc	sample7.pre	sample7.dsp	-
SAMPLE8	sample8.pc	sample8.pre	sample8.dsp	-



Table F-1 (Cont.) Sample Programs

Sample Program	Source Files	Pro*C/C++ GUI Project File	MSVC Compiler Project File	Notes
SAMPLE9	sample9.pc calldemo.sql	sample9.pre	sample9.dsp	Run calldemo.sql before building sample9.
SAMPLE10	sample10.pc	sample10.pre	sample10.dsp	-
SAMPLE11	<pre>sample11.pc sample11.sql</pre>	sample11.pre	sample11.dsp	Run sample11.sql before building sample11.
SAMPLE12	sample12.pc	sample12.pre	sample12.dsp	-
SCDEM01	scdemo1.pc	scdemo1.pre	scdemo1.dsp	-
SCDEMO2	scdemo2.pc	scdemo2.pre	scdemo2.dsp	-
SQLVCP	sqlvcp.pc	sqlvcp.pre	sqlvcp.dsp	-
WINSAM	resource.h winsam.h winsam.ico winsam.pc winsam.rc	winsam.pre	winsam.dsp	-

The following subsections describe the functionality of the sample programs.

#### ANSIDYN1

Demonstrates using ANSI dynamic SQL to process SQL statements that are not known until runtime. This program is intended to demonstrate the simplest (though not the most efficient) approach to using ANSI dynamic SQL.

#### **ANSIDYN2**

Demonstrates using ANSI dynamic SQL to process SQL statements that are not known until runtime. This program uses the Oracle extensions for batch processing and reference semantics.

#### COLDEMO1

Fetches census information for California counties. This program demonstrates various ways to navigate through collection-typed database columns.

#### CPDEMO1

Demonstrates how the connection pool feature can be used. It also shows how different connection pool options can be used to optimize performance.

#### CPDEMO2

Demonstrates connection pool feature with relatively complex set of SQL statements and shows how performance gain depends on the kind of SQL statements used by the program.

#### CPPDEMO1

Prompts the user for an employee number, then queries the emp table for the employee's name, salary, and commission. This program uses indicator variables (in an indicator struct) to determine whether the commission is NULL.



#### CPPDEMO2

Retrieves the names of all employees in a given department from the emp table (dynamic SQL Method 3).

#### CPPDEMO3

Finds all salespeople and prints their names and total earnings (including commissions). This program is an example of C++ inheritance.

#### **CVDEMO**

Declares and opens a ref cursor.

#### **EMPCLASS**

The EMPCLASS and CPPDEMO2 files were written to provide an example of how to write Pro\*C/C+ + programs within a C++ framework. EMPCLASS encapsulates a specific query on the emp table and is implemented using a cursor variable. EMPCLASS instantiates an instance of that query and provides cursor variable functionality (that is: open, fetch, close) through C++ member functions that belong to the emp class. The empclass.pc file is *not* a standalone demo program. It was written to be used by the cppdemo2 demo program. To use the emp class, you have to write a driver (cppdemo2.pc) which declares an instance of the emp class and issues calls to the member functions of that class.

#### LOBDEMO1

Fetches and adds crime records to the database based on the person's Social Security Number. This program demonstrates the mechanisms for accessing and storing large objects (LOBs) to tables and manipulating LOBs through the stored procedures available through the DBMS LOB package.

#### MLTTHRD1

Shows how to use threading in conjunction with precompilers. The program creates as many sessions as there are threads.



" Multithreaded Applications"

#### **NAVDEMO1**

Demonstrates navigational access to objects in the object cache.

#### **OBJDEMO1**

Demonstrates the use of objects. This program manipulates the object types *person* and *address*.

#### **ORACA**

Demonstrates how to use ORACA to determine various performance parameters at runtime.

#### **PLSSAM**

Demonstrates the use of embedded PL/SQL blocks. This program prompts you for an employee name that already resides in a database. It then executes a PL/SQL block, which returns the results of four SELECT statements.

#### **SAMPLE**

Adds new employee records to the personnel database and checks database integrity. The employee numbers in the database are automatically selected using the current maximum employee number +10.



#### **SAMPLE1**

Logs on to an Oracle database, prompts the user for an employee number, queries the database for the employee's name, salary, and commission, and displays the result. The program continues until the user enters 0 as the employee number.

#### SAMPLE2

Logs on to an Oracle database, declares and opens a cursor, fetches the names, salaries, and commissions of all salespeople, displays the results, and closes the cursor.

#### SAMPLE3

Logs on to an Oracle database, declares and opens a cursor, fetches in batches using arrays, and prints the results using the print rows () function.

#### SAMPLE4

Demonstrates the use of type equivalencies using the LONG VARRAW external datatype.

#### SAMPLE5

Prompts the user for an account number and a debit amount. The program verifies that the account number is valid and that there are sufficient funds to cover the withdrawal before it debits the account. This program shows the use of embedded SQL.

#### SAMPLE6

Creates a table, inserts a row, commits insert, and drops the table (dynamic SQL Method 1).

#### SAMPLE7

Inserts two rows into the emp table and deletes them (dynamic SQL Method 2).

#### SAMPLE8

Retrieves the names of all employees in a given department from the emp table (dynamic SQL Method 3).

#### SAMPLE9

Connects to an Oracle database using the <code>scott/tiger</code> account. The program declares several host arrays and calls a PL/SQL stored procedure (<code>GET\_EMPLOYEES</code> in the <code>CALLDEMO</code> package). The PL/SQL procedure returns up to <code>ASIZE</code> values. The program keeps calling <code>GET\_EMPLOYEES</code>, getting <code>ASIZE</code> arrays each time, and printing the values, until all rows have been retrieved.

#### SAMPLE10

Connects to an Oracle database using your username and password and prompts for a SQL statement. You can enter any legal SQL statement, but you must use regular SQL syntax, not embedded SQL. Your statement is processed. If the statement is a query, the rows fetched are displayed (dynamic SQL Method 4).

#### SAMPLE11

Fetches from the emp table, using a cursor variable. The cursor is opened in the stored PL/SQL procedure open cur, in the EMP DEMO PKG package.

#### SAMPLE12

Demonstrates how to do array fetches using dynamic SQL Method 4.

#### SCDEM01

Demonstrates how the scrollable cursor can be used with Oracle dynamic SQL Method 4. Scrollable cursor can also be used with ANSI dynamic SQL Method 4.

#### SCDEMO2

Demonstrates the use of scrollable cursor with host arrays.



#### **SQLVCP**

Demonstrates how you can use the sqlvcp() function to determine the actual size of a VARCHAR struct. The size is then used as an offset to increment a pointer that steps through an array of VARCHARS.

This program also demonstrates how to use the SQLStmtGetText() function to retrieve the text of the last SQL statement that was executed.

#### **WINSAM**

Adds new employee records to the personnel database and checks database integrity. You can enter as many employee names as you want and perform the SQL commands by selecting the appropriate buttons in the *Employee Record* dialog box. This is a GUI version of the sample program.

## F.2 Building the Demonstration Tables

To run the sample programs, you must have a database account with the username  $\mathtt{scott}$  and the password  $\mathtt{tiger}$ . Also, you must have a database with the sample tables  $\mathtt{emp}$  and  $\mathtt{dept}$ . This account is included in the starter database for your Oracle Database  $\mathtt{10}g$  server. If the account does not exist on your database, create the account before running the sample programs. If your database does not contain  $\mathtt{emp}$  and  $\mathtt{dept}$  tables, you can use the  $\mathtt{demobld}.\mathtt{sql}$  script to create them.

To build the sample tables:

- Start SQL\*Plus
- 2. Connect as username scott with the password tiger.
- 3. Run the demobld.sql script:

```
SQL> @ORACLE BASE\ORACLE HOME\sqlplus\demo\demobld.sql;
```

#### **Related Topics**

Oracle Database Administrator's Reference for Microsoft Windows

## F.3 About Building the Sample Programs

You can build the sample programs in two ways:

- Using the pcmake.bat file provided
- Using Microsoft Visual Studio

### F.3.1 Using pcmake.bat

The pcmake.bat file for compiling Pro\*C/C++ demos is found in the following location:

```
ORACLE_BASE\ORACLE_HOME\precomp\demo\proc
```

This batch file is designed to illustrate how Pro\*C/C++ applications can be built at the command prompt.

In order to use this batch file, Microsoft Visual Studio must be installed. The environment variable MSVCDir must be set. Pro\*C/C++ command line options and linker options vary depending on your application.

You can use this file to build a demo, to build sample1 for example:



Navigate to the location of the demo file and enter the following at the command prompt:

C:\> CD ORACLE\_BASE\ORACLE\_HOME\precomp\demo\proc\sample1

Enter the following:

% pcmake sample1

## F.4 Using Microsoft Visual Studio

Microsoft Visual Studio project files have an extension of .dsp. The .dsp files in the <code>ORACLE\_BASE\ORACLE\_HOME\precomp\demo\proc</code> directory guide and control the steps necessary to precompile, compile, and link the sample programs.

Pro\*C/C++, SQL\*Plus, and the Object Type Translator have been integrated into the Microsoft Visual Studio sample project files. You do not have to run Pro\*C/C++, SQL\*Plus, and the Object Type Translator separately before compilation.

To build a sample program:

- 1. Open a Microsoft Visual Studio project file, such as sample1.dsp.
- Check the paths in the project file to ensure that they correspond to the configuration of your system. If they do not, change the paths accordingly. Your system may produce error messages if the paths to all components are not correct.



All of the sample programs were created with C:\oracle\ora92 as the default drive.

Select Build > Rebuild All. Microsoft Visual Studio creates the executable.

#### **Related Topics**

- Setting the Path for the Sample .pre Files
- The Object Type Translator

## F.5 Setting the Path for the Sample .pre Files

By default the sample <code>.pre</code> files search for their corresponding <code>.pc</code> files in the <code>C:\oracle\ora92</code> directory where <code>C:\</code> is the drive that you are using, and <code>oracle\ora92</code> represents the location of the Oracle home. If the Oracle base and Oracle home directories are different on your computer, you must change the directory path to the correct path.

To change the directory path for a sample .pre file:

- 1. In Pro\*C/C++, open the .pre file.
- 2. Double-click the filename in the Input File area to display the Input File dialog box.
- Change the directory path to the correct path.
- Click Open.



G

# Integrating Pro\*C/C++ into Microsoft Visual Studio .NET

This appendix describes how to integrate Pro\*C/C++ into the Microsoft Visual Studio .NET 2002/2003 (or later) integrated development environment.

This appendix contains the following topics:

- Integrating Pro\*C/C++ within Microsoft Visual Studio .NET Projects
- Adding Pro\*C/C++ to the Tools Menu

# G.1 Integrating Pro\*C/C++ within Microsoft Visual Studio .NET Projects

This section describes how to fully integrate Pro\*C/C++ within Microsoft Visual Studio .NET projects.

All the precompiler errors and warnings are displayed in the output box where Microsoft Visual Studio .NET 2002/2003 displays compiler and linker messages. You do not have to precompile a file separately from the Microsoft Visual Studio .NET 2002/2003 build environment. More importantly, Microsoft Visual Studio .NET 2002/2003 maintains the dependencies between .c and .pc files. Microsoft Visual Studio .NET 2002/2003 maintains the dependency and precompile files, if needed.

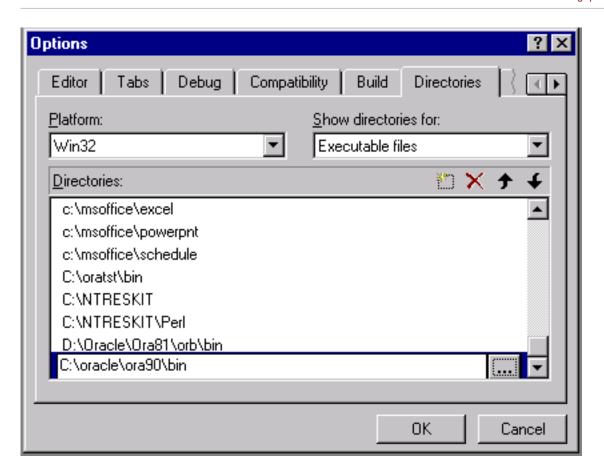
All of the procedures in this section are performed within Microsoft Visual Studio .NET.

### G.1.1 Specifying the Location of the Pro\*C/C++ Executable

For Microsoft Visual Studio .NET to run Pro\*C/C++, it must know the location of the Pro\*C/C++ executable. If Microsoft Visual Studio .NET was installed before any Oracle release products were installed, then you must add the directory path.

To specify the location of the Pro\*C/C++ executable:

- 1. Select **Options** from the Tools menu.
  - The Options dialog appears.
- 2. Click the **Directories** tab.
- 3. Select Executable files from Show directories For.
- 4. Scroll to the bottom of the Directories box and click the dotted rectangle.
- **5. Enter the** *ORACLE BASE\ORACLE HOME\*bin **directory. For example**:
  - C:\oracle\ora121\bin
- 6. Click OK.



## G.1.2 Specifying the Location of the Pro\*C/C++ Header Files

To specify the location of the Pro\*C/C++ header files:

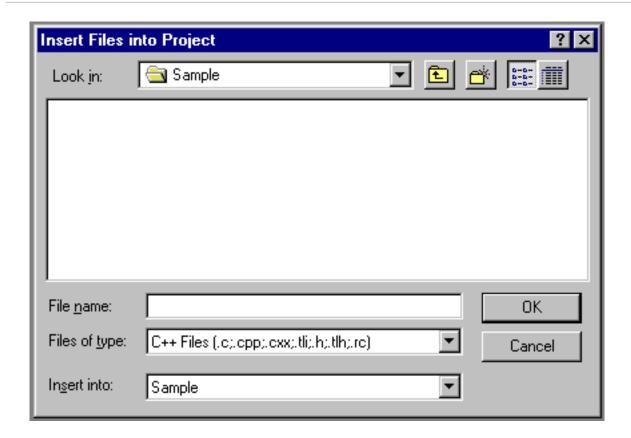
- 1. Select **Options** from the Tools menu. The Options dialog appears.
- 2. Click the **Directories** tab.
- 3. Select Include Files from the Show Directories For list.
- 4. Scroll to the bottom of the Directories box and click the **dotted rectangle**.
- 5. Enter the ORACLE\_BASE\ORACLE\_HOME\precomp\public directory. For example:
  C:\oracle\ora121\precomp\public
- 6. Click OK.

# G.2 Adding .pc Files to a Project

After you create a project, you need to add the .pc files.

To add a .pc file to a project:

 Select Add To Project from the Project menu and then select Files. The Insert Files into Project dialog appears.



- Select All Files from the Files list.
- 3. Select the .pc file.
- 4. Click OK.

## G.2.1 Adding References to .c Files to a Project

For each .pc file, you need to add a reference to the .c file that will result from precompiling.

To add a reference to a .c file to a project:

- Select Add To Project from the Project menu, and then select Files. The Insert Files into Project dialog appears.
- 2. Type the name of the .c file in the File Name box.
- 3. Click **OK**. Because the .c file has not been created yet, Microsoft Visual Studio .NET displays the following message: "The specified file does not exist. Do you want to add a reference to the project anyway?"
- Click Yes.

## G.2.2 Adding the Pro\*C/C++ Library to a Project

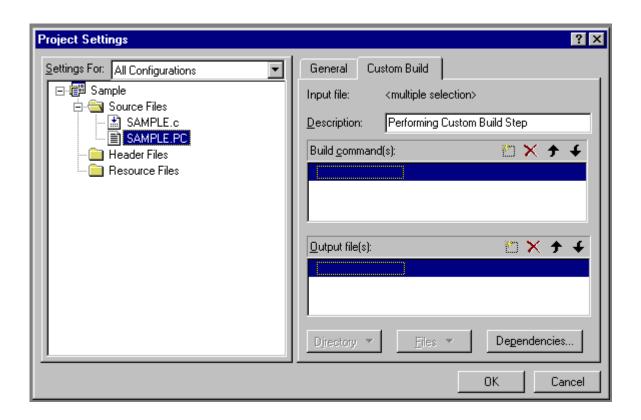
Pro\*C/C++ applications must link with the library file orasq112.lib.

To add the Pro\*C/C++ library to a project:

 Select Add To Project from the Project menu, and then select Files. The Insert Files into Project dialog appears.

- Select All Files from the Files list.
- 3. Select orasql12.lib from the ORACLE BASE\ORACLE HOME\precomp\lib\msvc directory.
- 4. Click OK.

## G.2.3 Specifying Custom Build Options



To specify Custom Build options:

- 1. In FileView, right-click a .pc file and select Settings. The Project Settings dialog appears with the Custom Build tab displayed.
- 2. In the Build commands box, on one line, set the build to use the same hardcoded path as that of the \$ORACLE HOME setting.
- 3. In the Output files box, enter one of the following:
  - If you are generating .c files, then enter \$(ProjDir)\\$(InputName).c.
  - If you are generating .cpp files, then enter \$(ProjDir) \\$(InputName).cpp.

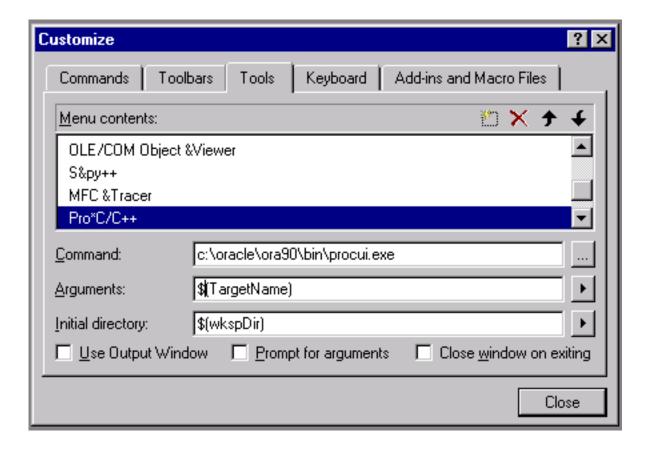
\$(ProjDir) and \$MSDEVDIR are macros for custom build commands in Microsoft Visual Studio .NET. When the project is built, Microsoft Visual Studio .NET checks the date of the output files to determine whether they need to be rebuilt for any new modifications made to the source code.

4. Click OK.

See Also:

Microsoft Visual Studio .NET documentation

## G.3 Adding Pro\*C/C++ to the Tools Menu



You can include Pro\*C/C++ as a choice in the Tools menu of Microsoft Visual Studio .NET.

To add Pro\*C/C++ to the Tools menu:

- **1.** From within Microsoft Visual Studio .NET, select **Customize** from the Tools menu. The Customize dialog appears.
- Click the Tools tab.
- 3. Scroll to the bottom of the Menu contents box and click the dotted rectangle.
- 4. Enter the following text:

Pro\*C/C++

5. In the Command box, type the path and filename of the graphical Pro\*C/C++ executable, or use the Browse button to the right of the box to select the file name. For example:

C:\oracle\ora121\bin\procui.exe

6. In the Arguments box, enter the following text:

\$ (TargetName)

When you select Pro\*C/C++ from the Tools menu, Microsoft Visual Studio .NET uses the \$ (TargetName) argument to pass the name of the current development project to Pro\*C/C++ then opens a precompile project with the same name as the opened project, but with a .pre extension in the project directory.

7. In the Initial directory box, enter the following text:

\$(WkspDir)

The Customize dialog should now look like the following graphic (although the Oracle home directory may be different on your computer).

8. Click Close. Microsoft Visual Studio .NET adds Pro\*C/C++ to the Tools menu.



# Index

Symbols	arrays (continued)		
	chapter discusses how to use, 8-1		
.dsp files, F-7	definition of, 4-37		
.pre files	host arrays, 2-5		
checking the paths, <i>F-7</i>	operations, 2-5		
#include	varying length, 18-2		
file inclusion, Pro*C compared to C, 5-22	associative interface, 17-4		
	when to use, 17-4		
Numerics	AT clause		
	in CONNECT statement, 3-6 in DECLARE CURSOR statement, 3-7		
sqlerrd[2] component	in DECLARE CORSOR statement, 3-7 in DECLARE STATEMENT statement, 3-8		
component, 9-12, 9-16	in EXECUTE IMMEDIATE statement, 3-8		
returns N or rows fetched, 8-6	of COMMIT statement, <i>E-18</i>		
use with data manipulation statements, 8-5	of DECLARE CURSOR directive, <i>E-26</i>		
	of EXECUTE IMMEDIATE statement, <i>E-42</i>		
A	of EXECUTE statement, E-38		
1	of INSERT statement, <i>E-51</i>		
abnormal termination	of SAVEPOINT statement, <i>E-78</i>		
automatic rollback, <i>E-19</i>	of SELECT statement, <i>E-80</i>		
active set	of SROLLBACK statement, <i>E-75</i>		
changing, 6-12, 6-13	of UPDATE statement, <i>E-85</i>		
cursor movement through, 6-12	restriction on, 3-8		
definition of, 2-5	use of, 3-7		
how identified, 6-10	attributes of a collection		
if empty, 6-13	descriptions, 18-11		
when fetched from, 6-12 when no longer defined, 6-10	AUTO_CONNECT, 10-14		
ALLOCATE	precompiler option, 3-3		
allocating a cursor variable, 4-26	AUTO_CONNECT precompiler option, 10-14		
ALLOCATE DESCRIPTOR statement, 14-9, E-10	automatic connections, 3-3, 3-6		
ALLOCATE SQL statement, 17-4			
ALLOCATE SQL statements, <i>E-9</i>	В		
allocating			
cursor variables, 4-26	batch fetch		
cursors, E-9	advantage of, 8-4		
thread context, 11-6, E-20	example of, 8-4		
ANSI C Support, D-1	number of rows returned by, 8-6		
ANSI dynamic SQL, F-1	BFILES		
reference semantics, 14-6	security, 16-2		
See also dynamic SQL (ANSI), 14-1	bind descriptor, 13-19, 15-4		
application development process, 2-6	definition of, 13-19		
array of structs, 8-16	information in, 13-20		
ARRAYLEN statement, 7-12	bind SQLDA		
arrays	purpose of, 15-2		
batch fetch, 8-4	bind variables		
bulk operations (ANSI dynamic SQL), 14-6	input host variables, 13-19		



binding	COLLECT GET statement
definition of, 13-3	example, <i>18-14</i>
BREAK action	COLLECTION APPEND, E-14
of WHENEVER, <i>E-89</i>	COLLECTION APPEND statement, 18-9
Byte Ordering, <i>D-1</i>	SQL statements
	COLLECTION APPEND, <i>E-14</i>
<b>C</b>	COLLECTION DESCRIBE
C	example, <i>18-15</i>
C preprocessor	COLLECTION DESCRIBE statement, 18-10
directives supported by Pro*C, 5-21	SQL statements
how used in Pro*C, 5-21	COLLECTION DESCRIBE, E-14
C structs	COLLECTION GET statement, 18-6
generating for a REF, 17-34	SQL statements
using, 17-33	COLLECTION GET, E-16
C types of collection attributes, 18-11	collection object types
C variable in SQLDA	handling, 18-3
how value is set, 15-8	COLLECTION RESET statement, 18-8
purpose of, 15-8	example, 18-16
C++, 1-7	SQL statements
C++ applications, <i>12-1</i>	COLLECTION RESET, E-16
cache, 17-3	COLLECTION SET statement, 18-7
CACHE FREE ALL SQL statement, 17-5	example, 18-14
CACHE FREE ALL statement, <i>E-11</i>	SQL statements
CALL SQL statement, <i>E-12</i>	COLLECTION SET, E-17
CALL SQL statement, 7-20	COLLECTION TRIM statement, 18-10
example, 7-21	SQL statements
CASE OTT parameter, 19-25	COLLECTION TRIM, <i>E-17</i>
case sensitivity	collection types
in precompiler options, 10-2	structs for, 18-2
CHAR datatype, 4-8	collections
· ·	and C, 18-2
CHAR_MAP precompiler option, 5-1, 10-14 character data, 5-1	autonomous access, 18-3
	descriptors, 18-2
character strings multibyte, 4-45	element access, 18-3
	manipulating, 18-3
CHARF datatype, 4-9, 5-11	nested tables, 18-1
CHARZ datatype, 4-9	OBJECT GET statement, 18-4
CLOSE CURSOR statement, 14-21	OBJECT SET statement, 18-4
CLOSE SQL statements, <i>E-13</i>	VARRAYS, 18-2
CLOSE statement	column list
dependence on precompiler options, 6-13	in INSERT statements, 6-7
example, 6-13, E-13	when permissible to omit, 6-7
purpose of, 6-10, 6-13	COMMENT clause
use in dynamic SQL method 4, 15-30	of COMMIT statement, <i>E-18</i>
CLOSE_ON_COMMIT	
precompiler option, <i>3-14</i> , <i>6-11</i> , <i>10-15</i>	comments
CLOSE_ON_COMMIT precompiler option, 10-15	ANSI, 2-7
closing	which allowed, 2-7
cursors, <i>E-13</i>	Comments
CODE	restrictions in PL/SQL block, 13-23
precompiler option, 12-2	commit
CODE option, 10-7	automatic, 3-12
CODE OTT parameter, 19-24	explicit versus implicit, 3-12
code page, 4-43	function of, 3-12
CODE precompiler option, 10-17	COMMIT SQL statement, <i>E-18</i>
coding conventions, 2-7	COMMIT statement, 3-13
	effect of. 3-13

COMMIT statement (continued)	context block
ending a transaction, <i>E-76</i>	definition of, 20-2
example of, 3-13	CONTEXT FREE statement, 11-7, E-21
examples, <i>E-19</i>	CONTEXT OBJECT OPTION GET SQL
purpose of, 3-13	statement, 17-16
RELEASE option in, 3-13	CONTEXT OBJECT OPTION SET SQL
using in a PL/SQL block, 3-21	statement, <i>17-15</i>
where to place, 3-13	CONTEXT USE directive, 11-6
committing	CONTEXT USE SQL directive, E-24
transactions, <i>E-18</i>	CONTEXT USE SQL statement, 11-6
communicating over a network, 3-4	CONTINUE action
COMP_CHARSET precompiler option, 10-18	in the WHENEVER statement, 9-20
compiling, 2-14	of WHENEVER directive, E-89, E-90
specifying include file locations, 5-26	result of, 9-20
concurrency	CONVBUFSZ clause, 4-45
definition of, 3-11	CPP_SUFFIX
concurrent connections, 3-4	precompiler option, 12-3
conditional precompilation, 2-12	CPP_SUFFIX precompiler option, 10-19
defining symbols, 5-33	CPP_SUFFIX precompiler options, 10-19
example, 2-12, 5-33	CREATE PROCEDURE statement
CONFIG OTT parameter, 19-25	embedded, 7-15
CONFIG precompiler option, 10-15–10-17, 10-19,	creating
10-20	savepoints, <i>E-77</i>
configuration files, 10-3, 10-7	creating temporary LOB, 16-11
and the Object Type Translator, 19-3	CURRENT OF clause, 8-3
location, 10-3, 10-7	example of, 6-21
system, 10-4	mimicking with ROWID, 3-20, 8-24
user, 10-4	purpose of, 6-21
CONNECT statement, <i>E-19</i>	restrictions on, 6-22
AT clause in, 3-6	current row
connecting to Oracle with, 3-1	definition of, 2-5
examples, <i>E-20</i>	using FETCH to retrieve, 6-10
requirements for, 3-2	cursor cache
USING clause in, 3-6	definition of, 9-29
using to enable a semantic check, C-3	purpose of, <i>B-7</i>
connecting to Oracle, 3-1	cursor control statements
automatic connections, 3-3	example of typical sequence, 6-22
concurrently, 3-4	cursor operations
example of, 3-2	overview of, 6-10
using Oracle Net, 3-4	cursor variables, 4-25, E-9
connection pooling, 11-14	allocating, 4-26
demo program1, <i>11-18</i>	declaring, 4-26
demo program2, <i>11-22</i>	restrictions on, 4-30
example, <i>11-24</i>	cursors, 2-13, 4-25
using, 11-16	allocating, <i>E-9</i>
connections	allocating a cursor variable, 4-26
concurrent, 3-9	analogy for, 2-5
default versus nondefault, 3-5	association with queries, 6-10
Explicit connections, 3-5	closing, <i>E-13</i>
implicit, 3-10	declaring, 6-11
naming of, 3-6	definition of, 2-5
const	explicit versus implicit, 2-5
declaring constants, 5-35	fetching rows from, E-43, E-45
CONTEXT ALLOCATE SQL statement, E-20	for multirow queries, 6-10
CONTEXT ALLOCATE statement, 11-6	how handling affects performance, B-5
	movement through active set, 6-12

cursors (continued) opening, E-69, E-71	DBMS interaction with MODE, 10-21 DBMS option, 5-11, 10-7
purpose of, 6-10	DBMS precompiler option, 10-20, 10-21
reopening, 6-12, 6-13	deadlock
restrictions on declaring, 6-11	definition of, 3-11
rules for naming, 6-11	effect on transactions, 3-17
scope of, 6-11	how broken, 3-17
scrollable cursors, 6-14	DEALLOCATE DESCRIPTOR statement, 14-10,
statements for manipulating, 6-10	E-25
types of, 2-5	declaration
using more than one, 6-11	of cursors, 6-11
using more than one, 0-11	of host arrays, 8-1
D	of pointer variables, 4-41
	of SQLCA, 9-13
data definition language	declarative SQL statements
in transactions, 3-12	in transactions, 3-12
data integrity, 3-10	DECLARE CURSOR directives
definition of, 3-11	examples, <i>E-27</i>
data locks, 3-11	DECLARE CURSOR statement, 14-19
database link	AT clause in, 3-7
creating synonym for, 3-10	use in dynamic SQL method 4, 15-20
defining, 3-10	DECLARE DATABASE SQL directives, <i>E-27</i>
example using, 3-10	Declare Section
using in INSERT statement, <i>E-52</i>	allowable statements, 2-9
where stored, 3-10	form, 2-8
database types	purpose, 2-8
new, 17-35	required when MODE=ANSI, 10-34
databases	requirements, 2-8
naming, 3-5	rules for defining, 2-9
datatype equivalencing, 2-5	when MODE=ANSI, 5-11
datatypes	when required, 2-8, 4-14
ANSI DATE, 4-10	DECLARE statement
codes used in descriptors, 15-12	example of, 6-11
conversions, 5-9	purpose of, 6-10
dealing with ORACLE internal, 15-11	required placement of, 6-11
equivalencing, 5-9	scope of, <i>E-29</i>
equivalencing, purpose of, 2-5	use in dynamic SQL method 3, 13-15
internal, 4-1	DECLARE STATEMENT directive, <i>E-28</i>
internal versus external, 2-4	DECLARE STATEMENT statement
INTERVAL DAY TO SECOND, 4-11	AT clause in, 3-8
INTERVAL YEAR TO MONTH, 4-11	example of using, 13-22
list of internal, 15-10	using with dynamic SQL, 13-21
need to coerce, 15-11	when required, 13-22
Oracle, 2-4	DECLARE statements
OTT mappings, 19-8	examples, <i>E-29</i>
restrictions on using, 17-38	DECLARE TABLE directive
TIMESTAMP, 4-10	examples, <i>E-30</i>
•	using with the SQLCHECK option, C-3
TIMESTAMP WITH LOCAL TIME ZONE, 4-11	DECLARE TABLE SQL directive, <i>E-30</i>
TIMESTAMP WITH TIME ZONE, 4-10	DECLARE TABLE statement
user-defined type equivalencing, <i>E-83</i>	need for with AT clause, 3-7
when to reset, 15-11	DECLARE TYPE directive, <i>E-31</i>
DATE AND	DEF_SQLCODE precompiler option, 10-22
DATE, ANSI	default connections, 3-5
datatype, 4-10	
datetime	default database, 3-5
avoiding unexpected results 4-11	default file name extensions, 19-31

DEFINE precompiler option, 10-22	dynamic SQL (continued)
used in application migration, 5-27	boolean support, 13-26
defining symbols, 2-12	cannot use cursor variables with, 4-30
DELETE SQL statement, <i>E-32</i>	choosing the right method, 13-5
DELETE statement	definition of, 2-3
embedded SQL examples, <i>E-34</i>	guidelines for, 13-5
example of, 6-8	method 1, F-5
purpose of, 6-8	method 2, F-5
using host arrays in, 8-13	method 3, <i>F-4</i> , <i>F-5</i>
WHERE clause in, 6-8	method 4, F-5
delimiters	overview of, 13-1
C versus SQL, 2-9	restriction on, 6-22
DEPT table, <i>2-14</i>	restrictions on use of datatypes, 17-38
DESCRIBE BIND VARIABLES statement	restrictions on using datatypes in, 17-38
use in dynamic SQL method 4, 15-20	uses for, 13-2
DESCRIBE command	using the AT clause in, 3-8
use with PREPARE command, <i>E-34</i>	when to use, 13-2
DESCRIBE DESCRIPTOR statement, <i>E-35</i>	dynamic SQL (ANSI)
DESCRIBE INPUT statement, 14-17	basics, 14-1
DESCRIBE OUTPUT statement, 14-17	bulk operations, 14-6
DESCRIBE SELECT LIST statement	differences from Oracle dynamic, 14-21
use in dynamic SQL method 4, 15-24	Oracle extensions, 14-5
DESCRIBE SQL statement, E-34	Overview, 14-2
DESCRIBE statement	precompiler options, 14-8
examples, <i>E-35</i>	Precompiler Options, 14-2
use in dynamic SQL Method 4, 13-19	reference semantics, 14-6
descriptions of collection attributes, 18-11	sample programs, 14-28
descriptors, 15-4	sample programs, 14-22
bind descriptor, 13-19	dynamic SQL method 1
definition of, 13-19	commands used with, 13-4
need for, 15-4	description of, 13-7
select descriptor, 13-19	example of, 13-7
using the sqlald() function to allocate, 15-4	how to use, 13-6
using the sqlclu() function to deallocate, 15-30	requirements for, 13-4
determining current value of precompiler options,	use of EXECUTE IMMEDIATE with, 13-7
10-4	use of PL/SQL with, 13-23
directory structures, 1-5	dynamic SQL method 2
distributed processing	commands used with, 13-4
support for, 3-5	description of, 13-10
using Oracle Net for, 3-5	example of, 13-11
distributed transactions, <i>E-77</i>	requirements for, 13-4
DML returning clauses, 6-9	use of DECLARE STATEMENT with, 13-21
DO action	use of EXECUTE with, 13-10
in the WHENEVER statement, 9-20	use of PL/SQL with, 13-23
of WHENEVER directive, <i>E-90</i>	use of PREPARE with, 13-10
result of, 9-20	dynamic SQL method 3
DTP model, 5-43	commands used with, 13-4
dummy host variables	compared to method 2, 13-14
placeholders, 13-3	example program, 13-16
DURATION precompiler option, 10-24, 17-16	requirements for, 13-4
Dynamic Link Libraries (DLLs), 1-6	sequence of statements used with, 13-14
dynamic PL/SQL	use of DECLARE STATEMENT with, 13-21
rules for, 13-22	use of DECLARE with, 13-15
versus dynamic SQL, 13-22	use of FETCH with, 13-15
dynamic SQL	use of OPEN with, 13-15
advantages and disadvantages of, 13-2	use of PL/SQL with, 13-23

dynamic SQL method 3 (continued)	embedded SQL (continued)
use of PREPARE with, 13-14	definition of, 2-1
dynamic SQL method 4	difference from interactive SQL, 2-3
need for descriptors with, 15-4	ENABLE THREADS statement, 11-6
overview of, 13-19	EXEC SQL CACHE FREE ALL, 17-5
prerequisites for using, 15-9	EXECUTE statement, <i>E-38</i>
requirements for, 13-4	key concepts of, 2-1
requirements of, 15-2	mixing with host-language statements, 2-3
sample program, 15-33	OPEN statement, <i>E-69</i>
sequence of statements used with, 13-21,	overview of, 2-1
15-15	PREPARE statement, <i>E-72</i>
steps for, 15-14	
•	requirements for, 2-3
use of CLOSE statement in, 15-30	SAVEPOINT statement, <i>E-77</i>
use of DECLARE CURSOR statement in,	SELECT statement, <i>E-78</i>
15-20	syntax for, 2-3
use of DECLARE STATEMENT with, 13-21	testing with SQL*Plus, 1-3
use of DESCR, 13-19	TYPE directive, <i>E-83</i>
use of DESCRIBE statement in, 15-20, 15-24	UPDATE statement, <i>E-84</i>
use of descriptors with, 13-19	using OCI types in, 17-35
use of FETCH statement in, 15-27	using REFs in, 17-34
use of OPEN statement in, 15-23	VAR directive, <i>E-87</i>
use of PL/SQL with, 13-23	when to use, 1-3
use of PREPARE statement in, 15-20	WHENEVER directive, <i>E</i> -89
use of the SQLDA in, 13-19, 15-3	embedded SQL statements
when needed, 13-19	labels for, 9-20
dynamic SQL methods	referencing host arrays in, 8-2
overview of, 13-3	referencing host variables in, 4-17
dynamic SQL statements	suffixes and prefixes not allowed, 2-8
binding of host variables in, 13-3	terminator for, 2-12
definition of, 13-1	use of apostrophes in, 2-9
requirements for, 13-2	use of quotation marks in, 2-9
use of placeholders in, 13-3	embedding
using host arrays in, 13-22	PL/SQL blocks in precompiler programs, <i>E-38</i>
versus static SQL statements, 13-2	EMP table, 2-14
versus static SQL statements, 13-2	ENABLE THREADS SQL statement, <i>E-37</i>
E	ENABLE THREADS statement, 11-6
	enabling
embedded PL/SQL	threads, 11-6
advantages of, <b>7-1</b>	encoding scheme (character set or code page),
cursor FOR loop, 7-2	4-43
example of, 7-5, 7-6	enqueues
overview of, 2-3	locking, 3-11
packages, 7-3	entering options, 10-10
PL/SQL tables, 7-3	environment variables, 10-2
procedures and functions, 7-2	equivalencing
requirements for, 7-5	host variable equivalencing, <i>E-87</i>
user-defined records, 7-4	user-defined type equivalencing, <i>E-83</i>
using %TYPE, 7-2	equivalencing of datatypes
using the VARCHAR pseudotype with, 7-8	datatype equivalencing, 2-5
using to improve performance, <i>B-2</i>	error detection
where allowed, 7-4	error reporting, <i>E-90</i>
embedded SQL, <i>F-4</i>	error handling, 2-6
ALLOCATE statement, <i>E-9</i>	alternatives, 9-1
CLOSE statement, <i>E-13</i>	need for, 9-1
	overview of, 2-6
CONTEXT FREE statement, 11-6, E-20	SQLCA versus WHENEVER statement, 9-2
CONTEXT FREE statement, 11-7	SQLOTT VOISUS VVITEIVE VEIT STATETHER, 3-2

error handling (continued)	EXECUTE IMMEDIATE statement (continued)
use of ROLLBACK statement in, 3-16	use in dynamic SQL method 1, 13-7
error messages	EXECUTE optional keyword of ARRAYLEN
maximum length of, 9-18	statement, 7-13
using the sqlglm() function to get, 9-17	EXECUTE SQL statement, <i>E-39</i>
error reporting	EXECUTE statement, 14-18
key components of, 9-11	examples, <i>E-38</i> , <i>E-40</i>
use of parse error offset in, 9-12	use in dynamic SQL method 2, 13-10
use of rows-processed count in, 9-11	execution of statements, 13-3
use of warning flags in, 9-11	execution plan, B-3, B-4
WHENEVER directive, E-90	EXPLAIN PLAN statement
ERRORS precompiler option, 10-24	function of, B-4
ERRTYPE	using to improve performance, <b>B-4</b>
precompiler option, 10-24	explicit connections, 3-6
ERRTYPE OTT parameter, 19-25	description of, 3-6
ERRTYPE precompiler option, 17-17	multiple, 3-9
exception, PL/SQL	single, 3-6
definition of, 7-10	extensions
EXEC ORACLE DEFINE statement, 5-33	default file name, 19-31
EXEC ORACLE ELSE statement, 2-12, 5-33	external datatypes
EXEC ORACLE ENDIF statement, 2-12, 5-33	definition of, 2-4
EXEC ORACLE IFDEF statement, 2-12, 5-33	FLOAT, 4-5
EXEC ORACLE IFNDEF statement, 2-12, 5-33	INTEGER, 4-5
EXEC ORACLE OPTION statement	STRING, 4-5
set option values inline, 10-10	external procedures
EXEC ORACLE statement	callbacks, 7-22
scope of, 10-11	calling from PL/SQL, 7-22
syntax for, 10-10	creating, 7-23
uses for, 10-11	error-handling, 7-24
EXEC ORACLE statements, 2-12	restrictions, 7-22
EXEC SQL CACHE FREE statement, 17-5	1030100013, 7 22
EXEC SQL clause	_
using to embed SQL, 2-3	F
EXEC SQL INCLUDE	F variable in SQLDA
contrasted with #include, 5-26	
EXEC SQL VAR statement	how value is set, 15-8
CONVBUFSZ clause, 4-45	purpose of, 15-8
EXEC TOOLS	FAQs, 1-6
GET CONTEXT statement, 20-4	FETCH DESCRIPTOR SQL statement, <i>E-45</i>
GET statement, 20-3	FETCH SQL statement, <i>E-43</i>
MESSAGE statement, 20-4	FETCH statement, 14-20
SET CONTEXT statement, 20-4	example of, 6-12
SET statement, 20-3	examples, <i>E-44</i>
EXEC TOOLS statements, 20-3	INTO clause in, 6-12
executable SQL statements	purpose of, 6-10, 6-12
purpose of, 6-5	results of, 6-12
uses for, 2-2	use in dynamic SQL method 3, 13-15
,	use in dynamic SQL method 4, 15-27
where allowed, 2-2	used after OPEN command, <i>E-72</i>
EXECUTE END-EXEC SQL statement, <i>E-38</i>	used after OPEN statement, <i>E-70</i>
EXECUTE DESCRIPTOR statement	fetching
SQL statements	rows from cursors, <i>E-43</i> , <i>E-45</i>
EXECUTE DESCRIPTOR, E-40	fetching in batches
EXECUTE IMMEDIATE statement, <i>E-42</i>	batch fetch, 8-4
EXECUTE IMMEDIATE statement, 14-19	I II II : Alexandra
	FIPS flagger
AT clause in, 3-8 examples, <i>E-43</i>	warns of array usage, 8-3 warns of missing Declare Section, 4-14

FIPS flagger (continued)	guidelines
warns of use of pointers as host variables, 5-5	for dynamic SQL, 13-5
FIPS precompiler option, 10-25	for separate precompilations, 2-13
flags	for the WHENEVER statement, 9-22
warning flags, 9-11	for transactions, 3-21
FLOAT datatype, 4-5	
FOR clause	Н
example of using, 8-14	
of embedded SQL EXECUTE DESCRIPTOR	header files
statement, <i>E-41</i>	location of, 5-30
of embedded SQL EXECUTE statement, <i>E-39</i>	HEADER precompiler option, 5-27, 10-26
purpose of, 8-14	heap
requirements for, 8-14	definition of, 9-29
restrictions on, 8-15	HFILE OTT parameter, 19-24
using with host arrays, 8-14	hints
when variable negative or zero, 8-14	COST, B-4
FOR UPDATE OF clause	for the ORACLE SQL statement optimizer,
locking rows with, 3-18	6-16
purpose of, 3-19	in DELETE statements, <i>E-34</i>
when to use, 3-18	in SELECT statements, <i>E-80</i>
FORCE clause	in UPDATE statements, E-87
of COMMIT statement, <i>E-19</i>	HOLD_CURSOR
of ROLLBACK statement, E-76	precompiler option
forward references	used to improved performance, B-9
why not allowed, 6-11	what it affects, B-5
FREE SQL statement, 17-5, E-47	HOLD_CURSOR option
free() function, 15-29	of ORACLE Precompilers, <i>E-13</i>
example of using, 15-29	HOLD_CURSOR precompiler option, 10-26
freeing	host arrays
thread context, 11-7, E-21	advantages of, 8-1
Frequently Asked Questions, 1-6	declaring, 8-1
full scan	dimensioning, 8-1
description of, B-5	in the DELETE statement, 8-13
function prototype	in the INSERT statement, 8-11
definition of, 10-17	in the SELECT statement, 8-4
functions	in the UPDATE statement, 8-12
cannot serve as host variables, 4-17	in the WHERE clause, 8-15
	matching sizes of, 8-2
C	referencing, 8-2
G	restrictions on, 8-3, 8-11–8-14
generic documentation references	used as input host variables, 8-2
default values for options, 10-7	used as output host variables, 8-2
header files, location of, 5-30	using in dynamic SQL statements, 13-22
linking, 1-6	using the FOR clause with, 8-14
GENXTB form	using to improve performance, <i>B-2</i>
how to run, 20-9	when not allowed, 8-2
use with user exits, 20-9	host language
GENXTB utility	definition of, 2-1, 2-2
how to run, 20-9	host program
use with user exits, 20-9	definition of, 2-1
GET DESCRIPTOR statement, 14-11	host structures
Globalization Support, 4-42	arrays in, 4-37
GOTO action	declaring, 4-36
in the WHENEVER statement, 9-20	host variables, 6-1
	assigning values to, 2-4
of WHENEVER directive, <i>E-90</i>	
result of, 9-20	declarations, 18-3

host variables (continued)	indicator variables (continued)	
declaring, 2-8, 18-3	using in PL/SQL, 7-9	
definition of, 2-4	using to detect NULLs, 6-3	
dummy, 13-3	using to detect truncated values, 6-3	
host variable equivalencing, <i>E-87</i>	using to insert NULLs, 6-3	
in EXECUTE statement, <i>E-40</i>	using to return NULLs, 6-4	
in OPEN statement, <i>E-70</i>	using to test for NULLs, 6-4	
in user exits, 20-2	with structures, 4-38	
input versus output, 6-1	INITFILE OTT parameter, 19-24	
must resolve to an address, 4-17	INITFUNC OTT parameter, 19-24	
overview of, 2-4	initialization function	
purpose of, 6-1	calling, 19-18	
rules for naming, 2-10	tasks of, 19-19	
using in PL/SQL, 7-5	input host variables	
doing in 1 2,0022, 7 0	assigning values to, 6-2	
	definition of, 6-1	
	restrictions on, 6-2	
Lyariable in COLDA	uses for, 6-1	
I variable in SQLDA	where allowed, 6-1	
how value is set, 15-7	INSERT SQL statement, <i>E-50</i>	
purpose of, 15-7	examples, <i>E-52</i>	
IAP in SQL*Forms	INSERT statement	
purpose of, 20-9	column list in, 6-7	
identifiers, ORACLE	example of, 6-7	
how to form, E-8	INTO clause in, 6-7	
implicit connections, 3-10	purpose of, 6-7	
multiple, 3-11	·	
single, 3-10	requirements for, 6-7	
IN OUT parameter mode, 7-3	using host arrays in, 8-11	
IN parameter mode, 7-3	VALUES clause in, 6-7	
in-doubt transaction, 3-20	inserting	
INAME precompiler option, 10-28	rows into tables and views, <i>E-50</i>	
INCLUDE	INTEGER datatype, 4-5	
using to include the SQLCA, 9-13	interface	
INCLUDE option, 10-7	native, 5-43	
INCLUDE precompiler option, <i>D-2</i>	XA, 5-43	
INCLUDE, SYS_INCLUDE Precompiler Options,	internal datatypes	
10-2	definition of, 2-4	
indexes	INTERVAL DAY TO SECOND datatype, 4-11	
using to improve performance, B-5	INTERVAL YEAR TO MONTH datatype, 4-11	
indicator arrays, 8-3	INTO clause	
example of using, 8-3	for output host variables, 6-1	
uses for, 8-3	in FETCH statements, 6-12	
INDICATOR keyword, 4-18	in INSERT statements, 6-7	
indicator variables	in SELECT statements, 6-6	
assigning values to, 6-2	of FETCH DESCRIPTOR statement, <i>E-46</i>	
association with host variables, 6-2	of FETCH statement, <i>E-44</i>	
declarations, 18-3	of SELECT statement, <i>E-80</i>	
declaring, 4-18, 18-3	used with FETCH instead of SELECT, 6-11	
definition of, 2-4	intype file, 19-27	
function of, 6-2	providing when running OTT, 19-6	
guidelines, 4-19	structure of, 19-27	
interpreting values of, 6-2	INTYPE OTT parameter, 19-23	
naming of, 4-39	INTYPE precompiler option, 10-30	
referencing, 4-18	invalid use	
requirements for, 6-3	of precompiler preprocessor, 5-24	
used with multibyte character strings, 4-45		

J	LOB FREE TEMPORARY statement, 16-15
joins	LOB LOAD SOL statement, 16-15
restriction on, 6-22	LOB LOAD SQL statement, <i>E-59</i> LOB OPEN SQL statement, <i>E-60</i>
resultation on, v ZZ	LOB OPEN statement, 16-16
	LOB READ SQL statement, E-60
L	LOB READ statement, 16-17
L variable in SOLDA	LOB TRIM SQL statement, <i>E-61</i>
L variable in SQLDA how value is set, 15-6	LOB WRITE SQL statement, <i>E-61</i>
purpose of, 15-6	LOBs, <i>F-4</i>
label name	access methods, 16-3
maximum length of, 9-20	buffering system, 16-7
large objects, <i>F-4</i>	external, 16-1
LDA, 5-39	initializing, 16-6
remote and multiple connections, 5-39	internal, 16-1
setup for OCI version 8, 5-39	locators, 16-2
lines	locators in C, 16-5
continuation, 2-11	temporary, 16-3
maximum length, 2-11	Location of Included Files, D-1
LINES precompiler option, 10-30	location of the Pro*C/C++ Executable, D-2
link	location transparency
database link, <i>3-10</i>	how provided, 3-11
linking, 1-6, 2-14	lock
on UNIX, 1-9	released by ROLLBACK statement, <i>E-76</i>
on VMS, 1-9	LOCK TABLE statement
two-task, 2-14	closes mall cursors, 3-20
Linking in an XA Library, D-2	example of, 3-19
LNAME precompiler option, 10-31	locking tables with, 3-19
LNPROC	NOWAIT parameter in, 3-19
VMS link script, 1-9	purpose of, 3-19
LOB APPEND SQL statement, <i>E-53</i>	locking, 3-18
LOB APPEND statement, 16-9	definition of, 3-11
LOB ASSIGN SQL statement, <i>E-53</i>	explicit versus implicit, 3-18
LOB ASSIGN statement, 16-9	modes of, 3-11
LOB CLOSE SQL statement, <i>E-54</i>	overriding default, 3-18
LOB CLOSE statement, 16-10	privileges needed to obtain, 3-21
LOB COPY SQL statement, <i>E-54</i>	table versus row, 3-18
LOB COPY statement, 16-10	uses for, 3-18
LOB CREATE TEMPORARY SQL statement,	with FOR UPDATE OF, 3-18
E-55	with the LOCK TABLE statement, 3-19
LOB CREATE temporary statement, 16-11	logon, 3-1
LOB DESCRIBE SQL statement, <i>E-55</i>	Logon Data Area, 5-39
LOB DISABLE BUFFERING SQL statement, <i>E-56</i>	LONG BAW determs 4.8
LOB DISABLE BUFFERING statement, 16-12	LONG RAW datatype, 4-8
LOB ENABLE BUFFERING SQL statement, <i>E-56</i>	LONG VARCHAR
LOB ENABLE BUFFERING statement, 16-12	datatype, 4-8 LONG VARRAW datatype, 4-8
LOB ERASE SQL statement, E-57	LTYPE precompiler option, 10-31
LOB ERASE statement, 16-13	LITPE precompiler option, 10-31
LOB FILE CLOSE ALL SQL statement, <i>E-57</i>	
LOB file close all statement, 16-13	M
LOB FILE CLOSE ALL statement, 16-13	M. variable in COLDA
LOB FILE SET SQL statement, E-58	M variable in SQLDA
LOB FILE SET statement, 16-14	how value is set, 15-8
LOB FLUSH BUFFER SQL statement, E-58	purpose of, 15-8
LOB FLUSH BUFFER statement, 16-14	macro precompiler options, 10-6

malloc()	NLS parameters (continued)
example of using, 15-26	NLS_LANGUAGE, 4-42
purpose of, 15-26	NLS_NUMERIC_CHARACTERS, 4-42
MAXLITERAL	NLS TERRITORY, 4-42
default value for, 2-11	NLS_CHAR precompiler option, 10-34, 10-35
MAXLITERAL precompiler option, 10-32	NLS_LOCAL precompiler option, 10-35
MAXOPENCURSORS	node
precompiler option	current, 3-5
· · · · · · · · · · · · · · · · · · ·	
effect on performance, <i>B-8</i>	definition of, 3-4
specifying for separate precompilation,	NOT FOUND condition
2-13	in the WHENEVER statement, 9-19
what it affects, B-5	meaning of, 9-19
MAXOPENCURSORS precompiler option, 10-33	of WHENEVER directive, <i>E-90</i>
MEMFORPREFETCH precompiler option, 10-41	NOWAIT parameter
metadata, <i>18-12</i>	effect of, 3-19
micro precompiler options, 10-6	in LOCK TABLE statements, 3-19
Microsoft Visual Studio	omitting, 3-19
integrating Pro*C/C++ into, G-1	null-terminated strings, 4-6
migration	NULLs
include files, 5-27	definition of, 2-4
MODE interaction with DBMS, 10-21	detecting, 6-3
MODE precompiler option, 10-34	handling in dynamic SQL method 4, 15-13
modes, parameter, 7-3	hardcoding, 6-3
msvcrt.lib runtime library, 1-6	inserting, 6-3
multithreaded applications, <i>F-1</i>	restrictions on, 6-4
···	
sample program, 11-9	returning, 6-4
user-interface features	testing for, 6-4
embedded SQL statements and	using the sqlnul() function to test for, 15-14
<del>-</del>	
directives, 11-6	NUMBER datatype, 4-5
<del>-</del>	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12
directives, 11-6	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions
<del>-</del>	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12
directives, 11-6	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions
N variable in SQLDA	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17
N variable in SQLDA how value is set, 15-6	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions
N variable in SQLDA how value is set, 15-6 purpose of, 15-6	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, <i>E-8</i>	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statements, 17-10
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statements, 17-10 OBJECT GET SQL statement, 17-13, E-65
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1 network	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1 network communicating over, 3-4	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22 creating types in the database, 19-3
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1 network communicating over, 3-4 protocols, 3-4 reducing traffic, B-3	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statements, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22 creating types in the database, 19-3 default name mapping, 19-31 outtype file, 19-15
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1 network communicating over, 3-4 protocols, 3-4 reducing traffic, B-3 NLS parameters, 4-42	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22 creating types in the database, 19-3 default name mapping, 19-31 outtype file, 19-15 parameters, 19-22
N  N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10  NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1 network communicating over, 3-4 protocols, 3-4 reducing traffic, B-3  NLS parameters, 4-42 NLS_CURRENCY, 4-42	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22 creating types in the database, 19-3 default name mapping, 19-31 outtype file, 19-15 parameters, 19-22 providing an intype file, 19-6
N  N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10  NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1 network communicating over, 3-4 protocols, 3-4 reducing traffic, B-3  NLS parameters, 4-42 NLS_CURRENCY, 4-42 NLS_DATE_FORMAT, 4-42	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22 creating types in the database, 19-3 default name mapping, 19-31 outtype file, 19-15 parameters, 19-22 providing an intype file, 19-6 reference, 19-21
N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10 NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1 network communicating over, 3-4 protocols, 3-4 reducing traffic, B-3 NLS parameters, 4-42 NLS_CURRENCY, 4-42 NLS_DATE_FORMAT, 4-42 NLS_DATE_LANGUAGE, 4-42	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22 creating types in the database, 19-3 default name mapping, 19-31 outtype file, 19-15 parameters, 19-22 providing an intype file, 19-6 reference, 19-21 restriction, 19-32
N  N variable in SQLDA how value is set, 15-6 purpose of, 15-6 naming of cursors, 6-11 of database objects, E-8 of select-list items, 15-3 of SQL*Forms user exits, 20-10  NATIVE value of DBMS option, 10-20, 10-21 native interface, 5-43 navigational access sample program, 17-27 nested tables, 18-1 creation, 18-1 network communicating over, 3-4 protocols, 3-4 reducing traffic, B-3  NLS parameters, 4-42 NLS_CURRENCY, 4-42 NLS_DATE_FORMAT, 4-42	NUMBER datatype, 4-5 using the sqlprc() function with, 15-12 numeric expressions cannot serve as host variables, 4-17  O  object cache, 17-3 OBJECT CREATE SQL statement, 17-8, E-62 OBJECT DELETE SQL statement, 17-10, E-63 OBJECT DEREF SQL statement, 17-9, E-64 OBJECT FLUSH SQL statement, E-65 OBJECT FLUSH SQL statement, 17-10 OBJECT GET SQL statement, 17-13, E-65 OBJECT RELEASE SQL statement, E-66 OBJECT SET SQL statement, 17-12, E-67 Object Type Translator (OTT), F-7 command line, 19-4 command line syntax, 19-22 creating types in the database, 19-3 default name mapping, 19-31 outtype file, 19-15 parameters, 19-22 providing an intype file, 19-6 reference, 19-21

OBJECT UPDATE SQL statement, 17-10, E-68	OPEN statement (continued)
objects	use in dynamic SQL method 4, 15-23
accessing with OCI, 19-17	opening
demonstration program, <i>F-4</i>	a cursor variable, 4-27
introduction to, 17-1	cursors, <i>E-69</i> , <i>E-71</i>
manipulating with OCI, 19-17	operators
persistent, 17-4	C versus SQL, 2-11
persistent versus transient copies of, 17-4	restrictions on, 2-11
references to, 17-2	optimization approach, <i>B-4</i>
support, 17-1	optimizer hints, <i>B-3</i>
• • • •	in C, 6-16
transient, 17-4	
types, 17-1	in C++, <i>6-17</i> , <i>12-2</i> ORACA
using object types in Pro*C/C++, 17-3	
OBJECTS precompiler option, 10-25, 10-36,	example of using, 9-32
17-17	using to gather cursor cache statistics, 9-31
OCI applications	ORACA precompiler option, 10-27, 10-32, 10-37,
using the OTT with, 19-16	10-43
OCI calls	ORACAID component, 9-29
embedding, 5-39	Oracle
OCI onblon() call	datatypes, <del>2-4</del>
not used to connect, 5-39	Forms Version 4, 20-3
OCI orlon() call	Open Gateway
not used to connect, 5-39	using the ROWID datatype with, 4-7
OCI Release 8, 5-35	Toolset, 20-3
accessing and manipulating objects, 19-17	Oracle Call Interface version 7, 5-39
embedding in Pro*C/C++, 5-38	Oracle Communications Area, 9-27
interfacing to, 5-36	Oracle Connection Manager in Traffic Director
parameters in the environment handle, 5-36	Mode, <i>B-10</i>
SQLLIB extensions, 5-35	Oracle names
OCI types	how to form, E-8
declaring, 17-35	Oracle Net
manipulating, 17-35	connecting to Oracle through, 3-5
OCIDate, 17-35	connection syntax, 3-4
OCINumber, 17-35	for concurrent connections, 3-4
OCIRaw, 17-35	function of, 3-4
OCIString, 17-35	Oracle Net Services
using in embedded SQL, 17-35	connecting, 3-3
OCIDate, 17-35	orasql9.lib, <i>G-4</i>
	•
declaring, 17-35	orasql9.lib library file, 1-6
ocidfn.h, 5-39	OTT (Object Type Translator), F-7
OCINumber, 17-35	OTT parameter TRANSITIVE, 19-26
declaring, 17-35	OTT parameters
OCIRaw, 17-35	CASE, 19-25
declaring, 17-35	CODE, 19-24
OCIString, 17-35	CONFIG, 19-25
declaring, 17-35	ERRTYPE, 19-25
ONAME precompiler option, 10-36	HFILE, 19-24
OPEN CURSOR statement, 14-20	INITFILE, 19-24
OPEN DESCRIPTOR SQL statement, <i>E-71</i>	INITFUNC, <i>19-24</i>
OPEN SQL statement, <i>E</i> -69	INTYPE, 19-23
OPEN statement	OUTTYPE, 19-23
effect of, 6-12	SCHEMA_NAMES, 19-26
example of, 6-12	USERID, 19-23
examples, E-70	where they appear, 19-26
purpose of, 6-10, 6-12	OUT parameter mode, 7-3
use in dynamic SQL method 3, 13-15	

output host variables	PL/SQL (continued)		
assigning values to, 6-1 definition of, 6-1 outtype file, 19-27 when running OTT, 19-15 OUTTYPE OTT parameter, 19-23 overhead	user-defined records, 7-4		
	PL/SQL blocks embedded in precompiler programs, <i>E-38</i>		
			placeholders
	duplicate, <i>13-11</i> , <i>13-23</i>		
	naming, <u>13-11</u>		
	reducing, B-2	proper order of, 13-11	
-	use in dynamic SQL statements, 13-3		
P	PLAN_BASELINE		
<u></u>	precompiler option, 10-39		
PAGELEN	PLAN_ENABLED		
precompiler option, 10-38	precompiler option, 10-41		
parameter modes, 7-3	PLAN_FIXED		
PARSE	precompiler option, 10-40		
precompiler option, 10-38	PLAN_PREFIX		
parse error offset	precompiler option, 10-40		
how to interpret, 9-12	PLAN_RUN		
use in error reporting, 9-12	precompiler option, 10-40		
PARSE option, 10-7	pointer		
parsing dynamic statements	definition of, 4-41		
PREPARE statement, <i>E-72</i>	to cursor variables		
password	restrictions on, 4-26		
defining, 3-2	pointer variables		
paths	declaring, 4-41		
checking, <i>F-7</i>	determining size of referenced value, 4-41		
checking the .pre files, F-7	referencing, 4-41		
pcscfg.cfg configuration file, 10-7	referencing struct members with, 4-41		
performance	precedence of precompiler options, 10-4		
eliminating extra parsing to improve, B-5	precision		
optimizing SQL statements to improve, B-3	definition of, 15-13		
reasons for poor, B-1	using sqlprc() to extract, 15-13		
using embedded PL/SQL to improve, B-2	when not specified, 15-13		
using HOLD_CURSOR to improve, B-9	precompilation		
using host arrays to improve, B-2	conditional, 2-12		
using indexes to improve, B-5	separate, 2-13		
using RELEASE_CURSOR to improve, B-9	precompilation unit, 3-2, 10-7		
using row-level locking to improve, B-5	precompiled header files, 5-27		
persistent copies of objects, 17-4	C++ restrictions, 5-31		
persistent objects, 17-4	CODE option, 5-31 PARSE option, 5-31		
PL/SQL, 1-3	precompiler options		
anonymous block	alphabetized list, 10-7, 10-13		
used to open a cursor variable, 4-27	AUTO_CONNECT, 10-14		
cursor FOR loop, 7-2	case sensitivity, 10-2		
description of, 1-3	CHAR MAP, 5-1, 10-14		
difference from SQL, 1-3	CLOSE ON COMMIT, 6-11, 6-15, 10-15		
executing a block using the AT clause, 3-7	CODE, 10-17		
integration with database server, 7-2	COMP CHARSET, 10-18		
main advantage of, 1-3	CONFIG, 10-15–10-17, 10-19, 10-20		
packages, 7-3	configuration files, <i>10-3</i>		
PL/SQL tables, 7-3	CPP_SUFFIX, 10-19		
procedures and functions, 7-2 RECORD type	DBMS, 10-20, 10-21		
cannot be bound to a C struct, 4-37	DEF_SQLCODE, 10-22		
relationship with SQL, 1-3	DEFINE, 10-22		
setting SQLCA, 9-17	determining current value, 10-4		
	5		

precompiler options (continued)	PREPARE statement, 14-16
DURATION, 10-24	effect on data definition statements, 13-4
DYNAMIC, 14-8	examples, <i>E-73</i>
entering, <i>10-10</i>	use in dynamic SQL, 13-10, 13-14
entering on the command line, 10-10	use in dynamic SQL method 4, 15-20
ERRORS, 10-24	preprocessor
ERRTYPE, 10-24	example, 5-33
FIPS, 10-25	EXEC ORACLE directives, 5-33
HEADER, 10-26	preprocessor directives
HOLD_CURSOR, 10-26, 10-27	directives not supported by Pro*C, 5-22
INAME, 10-28	preprocessor, support of, 4-1
INCLUDE, 10-29	private SQL area
INTYPE, 10-30	association with cursors, 2-5
LINES, 10-30	definition of, 2-5
list of, 10-13	opening of, 2-5
LNAME, <i>10-31</i>	purpose of, B-7
LTYPE, 10-31	Pro*C/C++
MAXLITERAL, 2-11, 10-32	configuration files, 10-7
MAXOPENCURSORS, 10-33	integrating into Microsoft Visual Studio, G-1
MEMFORPREFETCH, 10-41	library file, G-3
micro and macro, 10-6	linking, 1-6
MODE, 10-34, 14-8	Pro*C/C++ Precompiler
NLS_CHAR, 10-34, 10-35	common uses for, 1-2
NLS_LOCAL, 10-35	Globalization Support, 4-44
OBJECTS, 10-25, 10-36	new database types, 17-35
ONAME, 10-36	object support in, 17-1
	runtime context, 5-35
ORACA, 10-27, 10-32, 10-37, 10-43	
PAGELEN, 10-38	use of PL/SQL with, 7-4
PARSE, 10-38	using OTT with, 19-19
PLAN_BASELINE, 10-39	procedural database extension, 7-3
PLAN_ENABLED, 10-41	program termination
PLAN_FIXED, 10-40	normal versus abnormal, 3-17
PLAN_PREFIX, 10-40	programming guidelines, 2-7
PLAN_RUN, <i>10-40</i>	project files, F-7
precedence, 10-4	
PREFETCH, <i>10-42</i>	Q
RELEASE_CURSOR, 10-42	<u> </u>
scope, 10-7	queries
scope of, 10-7	association with cursors, 6-10
SELECT_ERROR, 10-43, 10-44	forwarding, 3-10
specifying, 10-10	incorrectly coded, 6-6
SQLCHECK, 17-18	kinds of, 6-5
syntax for, 10-10	requirements for, 6-5
SYS_INCLUDE, 10-44	returning more than one row, 6-5
THREADS, 10-46, 11-5	returning more than one row, 0 3
TYPE_CODE, 10-46, 14-8	_
UNSAFE NULL, 10-46	R
USERID, 10-47	
	RAW datatype, 4-7
using, 10-13	READ ONLY parameter
VARCHAR, 10-48	in SET TRANSACTION statement, 3-18
VERSION, 10-48	read-only transactions
Precompiler Options, SYS_INCLUDE, INCLUDE,	description of, 3-18
10-2	example of, 3-18
predefined symbols, 2-12	how ended, 3-18
PREFETCH precompiler option, 6-15, 10-42	record, 7-4
PREPARE SOL statement F-72	

REF	roll back
structure for, 17-34	to a savepoint, <i>E-77</i>
REF (reference to object), 17-2	to the same savepoint multiple times, <i>E-76</i>
REFERENCE clause	ROLLBACK SQL statement, <i>E-75</i>
in TYPE statement, 5-11	ROLLBACK statement
reference semantics (ANSI dynamic SQL), 14-6	effect of, 3-16
references to objects (REFs)	ending a transaction, <i>E-76</i>
declaring, 17-34	example of, 3-16
using, 17-34	examples, <i>E-76</i>
using in embedded SQL, 17-34	in error handling routines, 3-16
referencing	purpose of, 3-16
of host arrays, 8-2	RELEASE option in, 3-16
REFs	TO SAVEPOINT clause in, 3-16
declaring, <i>17-34</i>	using in a PL/SQL block, 3-21
using, 17-34	where to place, 3-16
using in embedded SQL, 17-34	rollbacks
REGISTER CONNECT SQL statement, E-74	function of, 3-12
RELEASE option, 3-17	statement-level, 3-17
if omitted, 3-17	row locks
in COMMIT statement, 3-13	acquiring with FOR UPDATE OF, 3-19
in ROLLBACK statement, 3-16	advantage of, B-5
purpose of, 3-13	using to improve performance, B-5
restriction on, 3-16	when acquired, 3-19
RELEASE_CURSOR	when released, 3-19
precompiler option	ROWID
what it affects, B-5	logical, 4-6
RELEASE_CURSOR option	pseudocolumn, 3-20
of ORACLE Precompilers, <i>E-13</i>	using to mimic CURRENT OF, 3-20, 8-24
using to improve performance, B-9	universal, 4-6
RELEASE_CURSOR precompiler option, 10-42	ROWID datatype, 4-6
remote database	rows
declaration of, <i>E-27</i>	fetching from cursors, <i>E-43</i> , <i>E-45</i>
reserved namespaces, A-3	inserting into tables and views, <i>E-50</i>
reserved words and keywords, A-1	updating, <i>E-84</i>
resource manager, 5-42	rows-processed count
restrictions	use in error reporting, 9-11
on AT clause, 3-8	runtime context
on Comments, 13-23	establishing, 5-35
on CURRENT OF clause, 6-22	terminating, 5-35
on declaring cursors, 6-11	runtime type checking, 17-18
on FOR clause, 8-15	
on host arrays, 8-3, 8-11–8-14	S
on input host variables, 6-2	
on NULLs, 6-4	S variable in SQLDA
on separate precompilation, 2-13	how value is set, 15-8
on SET TRANSACTION statement, 3-18	purpose of, 15-8
use of CURRENT OF clause, 8-3	sample database tables
retrieving rows from a table	DEPT table, 2-14
embedded SQL, <i>E-78</i>	EMP table, 2-14
return codes	sample object type code, 17-27
user exits, 20-6	sample programs
returning clause	ANSIDYN1, F-1
DELETE, 6-8	ansidyn1.pc, <i>14-22</i>
in INSERT, 6-9	ANSIDYN2, F-1
in UPDATE, 6-8	ansidyn2.pc, <i>14-28</i>
returning clauses, 6-9	building, <i>F-6</i>

sample programs (continued)	sample programs (continued)
calldemo.sql, with sample9.pc, 7-17	SQLVCP, F-1
COLDEMO1, F-1	sqlvcp.pc, <i>4-23</i>
coldemo1.pc, 18-18	WINSAM, F-1
CPPDEMO1, F-1	sample tables
cppdemo1.pc, <u>12-4</u>	building, <i>F-6</i>
CPPDEMO2, F-1	SAVEPOINT SQL statement, <i>E-77</i>
cppdemo2.pc, <u>12-7</u>	SAVEPOINT statement, <i>E-77</i>
CPPDEMO3, F-1	example of, 3-14
cppdemo3.pc, 12-10	examples, E-78
cursor variable demos, 4-31	purpose of, 3-14
CV DEMO, F-1	savepoints
cv_demo.pc, 4-31	creating, E-77
cv_demo.sql, 4-31	definition of, 3-14
default drive, <i>F-7</i>	uses for, 3-14
described, F-3	when erased, 3-15
EMPCLASS, F-1	scale
extp1.pc, 7-23	definition of, 15-13, E-88
how to precompile, 2-15, 2-17	using sqlprc() to extract, 15-13
LOBDEMO1, <i>F-1</i>	·
	when negative, 15-13, E-88
lobdemo1.pc, 16-27	Scale
location of, 1-5, F-1	using SQLPRC to extract, E-88
MLTTHRD1, F-1	SCHEMA_NAMES OTT parameter, 19-26
NAVDEMO1, F-1	usage, <i>19-30</i>
navdemo1.pc, 17-27	scope
OBJDEMO1, F-1	of a cursor variable, 4-26
ORACA, <i>F-1</i>	of DECLARE STATEMENT directive, <i>E-29</i>
oraca.pc, 9-32	of precompiler options, 10-7
PLSSAM, F-1	of the EXEC ORACLE statement, 10-11
SAMPLE, F-1	of WHENEVER statement, 9-22
SAMPLE1, F-1	search condition
sample1.pc, 2-15	definition of, 6-8
SAMPLE10, <i>F-1</i>	in the WHERE clause, 6-8
sample10.pc, 15-33	select descriptor, 15-4
SAMPLE11, F-1	information in, 13-20
sample11.pc, 4-31	select list
SAMPLE12, <i>F-1</i>	definition of, 6-6
sample12.pc, 15-32	number of items in, 6-6
SAMPLE2, F-1	using the free() function for, 15-29
sample2.pc, 4-39	using the malloc() function for, 15-26
SAMPLE3, F-1	SELECT SQL statement, <i>E-78</i>
sample3.pc, 8-6	select SQLDA
SAMPLE4, F-1	purpose of, 15-2
sample4.pc, 5-12	SELECT statement, 6-6
SAMPLE5, F-1	clauses available for, 6-6
sample5.pc, <i>20-7</i>	embedded SQL examples, <i>E-80</i>
SAMPLE6, <i>F-1</i>	example of, 6-6
sample6.pc, <i>13-7</i>	INTO clause in, 6-6
SAMPLE7, <i>F-1</i>	purpose of, 6-6
sample7.pc, <i>13-11</i>	testing, 6-7
SAMPLE8, F-1	using host arrays in, 8-4
sample8.pc, <i>13-16</i>	WHERE clause in, 6-6
SAMPLE9, F-1	SELECT_ERROR
sample9.pc, 7-17	precompiler option, 6-6, 10-43, 10-44
setting the path, F-7	semantic checking
setting the path for the .pre files, F-7	controlling with the SQLCHECK option, C-1

semantic checking (continued)	SQL statements (continued)
definition of, <i>C-1</i>	CONTEXT FREE, <i>E-21</i>
enabling, <i>C-2</i>	CONTEXT OBJECT OPTION GET, E-22
with the SQLCHECK option, C-1	CONTEXT OBJECT OPTION SET, E-23
separate precompilation	DEALLOCATE DESCRIPTOR, E-25
guidelines for, 2-13	DELETE, E-32
referencing cursors for, 2-13	DESCRIBE, E-34
restrictions on, 2-13	DESCRIBE DESCRIPTOR, E-35
specifying MAXOPENCURSORS for, 2-13	ENABLE THREADS, E-37
using a single SQLCA with, 2-13	executable versus directive, 2-2
server	EXECUTE, <i>E-39</i>
integration with PL/SQL, 7-2	EXECUTE END-EXEC, E-38
session	EXECUTE IMMEDIATE, E-42
definition of, 3-11	FETCH, <i>E-43</i>
sessions	FETCH DESCRIPTOR, E-45
beginning, <i>E-19</i>	for defining and controlling transactions, 3-12
SET clause	for manipulating a cursor, 6-6, 6-10
in UPDATE statements, 6-8	FREE, <i>E-47</i>
purpose of, 6-8	INSERT, <i>E-50</i>
use of subqueries in, 6-8	LOB APPEND, E-53
SET DESCRIPTOR statement, 14-14, E-81	LOB ASSIGN, E-53
SET TRANSACTION statement	LOB CLOSE, E-54
example of, 3-18	LOB COPY, E-54
purpose of, 3-18	LOB CREATE, E-55
READ ONLY parameter in, 3-18	LOB DESCRIBE, E-55
requirements for, 3-18	LOB DISABLE BUFFERING, E-56
restrictions on, 3-18	LOB ENABLE BUFFERING, E-56
Size of an Integer and ROWID, <i>D-1</i>	LOB ERASE, <i>E-57</i>
SQL	LOB FILE CLOSE, <i>E-57</i>
benefits of, 1-3	LOB FILE SET, E-58
Embedded SQL, 1-3	LOB FLUSH BUFFER, <i>E</i> -58
nature of, <i>1-3</i>	LOB FREE TEMPORARY, <i>E-59</i>
need for, 1-3	LOB LOAD, <i>E-59</i>
SQL Communications Area, 9-2	LOB OPEN, <i>E-60</i>
SQLCA, 9-12	LOB READ, <i>E-60</i>
SQL Descriptor Area	LOB TRIM, <i>E-61</i>
SQLDA, 13-19, 15-3	LOB WRITE, <i>E-61</i>
SQL directives	OBJECT CREATE, E-62
CONTEXT USE, <i>11-6</i> , <i>E-24</i>	OBJECT DELETE, E-63
DECLARE DATABASE, E-27	OBJECT DEREF, E-64
DECLARE STATEMENT, E-28	OBJECT FLUSH, E-65
DECLARE TABLE, E-30	OBJECT GET, <i>E-65</i>
DECLARE TYPE, E-31	OBJECT RELEASE, E-66
TYPE, <i>E-83</i>	OBJECT SET, E-67
VAR, <i>E-87</i>	OBJECT UPDATE, <i>E-68</i>
WHENEVER, E-89	OPEN, <i>E-69</i>
SQL statements	OPEN DESCRIPTOR, E-71
ALLOCATE, <i>E-9</i>	optimizing to improve performance, <i>B-3</i>
ALLOCATE DESCRIPTOR TYPE, <i>E-10</i>	PREPARE, <i>E-72</i>
CACHE FREE ALL, <i>E-11</i>	REGISTER CONNECT, E-74
CALL, 7-20, E-12	ROLLBACK, E-75
CLOSE, <i>E-13</i> COMMIT, <i>E-18</i>	rules for executing, <i>B-3</i> SAVEPOINT, <i>E-77</i>
concerns when executing, 6-5	SELECT, E-78
CONNECT, E-19	SET DESCRIPTOR, E-81
CONTEXT ALLOCATE, <i>E-20</i>	summary of, <i>E-3</i>

SQL statements (continued)	sqlclut() function
types of, 2-2	see SQLSQLDAFree(), 5-41
UPDATE, <i>E-84</i>	sqlcode
SQL_CURSOR, E-9	component in SQLCA, 9-11
SQL_SINGLE_RCTX	SQLCODE
defined constant, 5-40	with MODE=ANSI, 10-34
definition of, 5-36	SQLCODE status variable
SQL, dynamic, 2-3	declaring, 9-10
SQL*Forms	when declared with the SQLCA, 9-11
display error screen in, 20-6	when used, <i>9-11</i>
IAP constants in, 20-6	sqlcpr.h, 9-18
returning values to, 20-6	sqlcurt() function
reverse return code switch in, 20-6	see SQLDAToResultSetCursor(), 5-41
SQL*Plus, 1-3	SQLDA
using to test SELECT statements, 6-7	bind versus select, 13-20
versus embedded SQL, 1-3	C variable in, 15-8
sqlald() function	definition of, 13-20
example of using, 15-17	F variable in, 15-8
purpose of, 15-4	I variable in, 15-7
syntax for, 15-4	information stored in, 13-20
sqlaldt() function	L variable in, 15-6
see SQLSQLDAAlloc, 5-41	M variable in, 15-8
SQLCA, 9-2, 9-11	N variable in, 15-6
=	purpose of, 13-19
sqlerrd[2] component in, 9-16	• •
components in, <i>9-15</i>	S variable in, 15-8
components set for a PL/SQL block, 9-17	struct, contents of, 15-4
declaring, 9-13	structure of, 15-5
description of, 9-12	use in dynamic SQL method 4, 15-3
explicit versus implicit checking of, 9-2	V variable in, 15-6
including multiple times, 5-26	X variable in, 15-9
overview of, 2-6	Y variable in, 15-9
SQLCABC component in, 9-15	Z variable in, 15-9
SQLCAID component in, 9-15	sqlda.h, <i>15-2</i>
sqlerrmc component in, 9-16	SQLDAToResultSetCursor(), 5-41
sqlerrml component in, 9-16	SQLEnvGet function in SQLLIB, 5-37
sqlwarn, 9-17	SQLEnvGet(), 5-42
use in separate precompilations, 2-13	sqlerrd[2] component, 9-11, 9-16
using more than one, 9-12	sqlerrmc component, 9-16
sqlca.h	sqlerrml component, 9-16
listing of, 9-14	SQLERROR
use of SQLCA_STORAGE_CLASS with, 2-13	WHENEVER directive condition, <i>E-90</i>
SQLCAID component, 9-15	SQLERROR condition
SQLCDAFromResultSetCursor(), 5-41	in the WHENEVER statement, 9-19
SQLCDAGetCurrent, 5-41	meaning of, 9-19
sqlcdat()	SQLErrorGetText(), 5-41
see SQLCDAFromResultSetCursor(), 5-41	SQLExtProcError(), 5-42, 7-24
SQLCHECK option	sqlglm(), 9-18
restrictions on, C-2	sqlglm() function, 9-17
what it affects, C-2	example of using, 9-18
SQLCHECK precompiler option, 17-18, C-3	parameters of, 9-17
SQLCHECK support for objects, 17-18	sqlglmt()
sqlclu() function	see SQLErrorGetText, 5-41
example of using, 15-30	sqlgls() function, 9-25
purpose of, 15-30	example of use, 4-23
syntax for, <i>15-30</i>	sample program for, 9-27

sqlgls() function (continued)	SQLSTATE (continued)
see SQLLIB	using, <u>9-10</u>
function SQLStmGetText, 4-23	values, 9-3
sqlglst() function	with MODE=ANSI, 10-34
see SQLStmtGetText, 5-41	SQLStmtGetText, 5-41
sqlld2t() function	SQLStmtGetText() function, F-6
see SQLLDAGetName, 5-41	SQLSvcCtxGet function in SQLLIB, 5-37
SQLLDAGetName, 5-41	SQLSvcCtxGet(), 5-42
sqlldat() function	SQLVarcharGetLength, 5-42
see SQLCDAGetCurrent, 5-41	sqlvcp() function, F-6
SQLLIB	sqlvcp() function, see SQLLIB
and embedded SQL, 2-3	function SQLVarcharGetLength, 4-22
extensions for OCI interoperability, 5-35	sqlvcpt() function
function	see SQLVarcharGetLength, 5-42
SQLCDAFromResultSetCursor, 5-41	sqlwarn
function SQLCDAGetCurrent, 5-41	flag, <i>9-17</i>
function SQLColumnNullCheck, 5-41	SQLWARNING
function SQLDAFree, 5-41	WHENEVER directive condition, <i>E-90</i>
function SQLDATTEE, 5-41 function SQLDATOResultSetCursor, 5-41	SQLWARNING condition
function SQLEnvGet, 5-37, 5-42	
function SQLErivGet, 5-37, 5-42	in the WHENEVER statement, 9-19
function SQLExtProcError, 5-42, 7-24	meaning of, 9-19 Standard Header Files, <i>D-1</i>
	,
function SQLLDAGetName, 5-41	statement-level rollback
function SQLNumberPrecV6, 5-42	description of, 3-17
function SQLNumberPrecV7, 5-42	to break deadlocks, 3-17
function SQLRowidGet, 5-42	status codes
function SQLStmtGetText(), 5-41	meaning of, <i>9-11</i>
function SQLSvcCtxGet, 5-37, 5-42	STOP action
function SQLVarcharGetLength, 4-22	in the WHENEVER statement, 9-20
new names for public functions, 5-40	of WHENEVER directive, <i>E-90</i>
SQLLIB function	result of, 9-20
SQLSQLDAAlloc, 5-41	stored procedures
SQLVarcharGetLength, 5-42	program example, 7-17
sqlnul() function	stored subprograms
example of using, 15-14	calling, 7-17
purpose of, 15-14	creating, 7-15
syntax for, 15-14	packaged versus standalone, 7-15
sqlnult() function	stored versus inline, 7-15
see SQLColumnNullCheck(), 5-41	STRING datatype, 4-5
SQLNumberPrecV6, 5-42	string host variables
SQLNumberPrecV7, 5-42	declaring, 5-6
sqlpr2() function, 15-13	Struct Component Alignment, D-1
sqlpr2t() function	structs
see SQLNumberPrecV7, 5-42	array of, 8-16
sqlprc() function, 15-12, 15-13	as host variables, 4-36
sqlprct() function	C, using, 17-33
see SQLNumberPrecV6, 5-42	for collection object types, 18-2
SQLRowidGet(), 5-42	generating C structs for a REF, 17-34
SQLSQLDAAlloc, 5-41	pointers as host variables, 4-42
SQLSQLDAFree(), 5-41	structs (structures)
SQLSTATE	cannot be nested, 4-37
class codes, 9-3	structures (structs)
declaring, 9-3	nesting not permitted for host, 4-37
mapping to Oracle errors, 9-6	subqueries
status codes, 9-6	definition of, 6-7
status variable, 9-3	example of, 6-7, 6-8

subqueries (continued)	TIMESTAMP WITH LOCAL TIME ZONE datatype
uses for, 6-7	4-11
using in the SET clause, 6-8	TIMESTAMP WITH TIME ZONE datatype, 4-10
using in the VALUES clause, 6-7	TO clause
symbols	of ROLLBACK statement, E-75
defining, 2-12	TO SAVEPOINT clause
syntax checking	in ROLLBACK statement, 3-16
controlling with the SQLCHECK option, <i>C-1</i>	purpose of, 3-16
definition of, <i>C-1</i>	restriction on, 3-16
syntax diagram	Toolset
description of, <i>E-6</i>	Oracle, 20-3
how to read, <i>E-6</i>	trace facility
how to use, <i>E-6</i>	function of, <i>B-4</i>
symbols used in, <i>E-6</i>	using to improve performance, <i>B-4</i>
syntax, embedded SQL, 2-3	transaction processing
SYS_INCLUDE	overview of, 2-5
system header files in C++, 12-4	statements used for, 2-5
SYS_INCLUDE precompiler option, 10-44	transaction processing monitor, 5-42
SYS_INCLUDE, INCLUDE Precompiler Options,	transactions
10-2	committing, <i>E-18</i>
system configuration file, 10-4	contents of, 2-5, 3-12
System Configuration File, <i>D-2</i>	definition of, 2-5
system failure	description of, 3-12
effect on transactions, 3-12	distributed, E-77
System Global Area (SGA), 7-15	failure during, 3-13
system header files	guarding databases with, 3-12
specifying the location of, 12-4	guidelines for, 3-21
system-specific Oracle documentation, 1-9, 2-14,	how to begin, 3-12
3-5, 5-44, 20-1	how to end, 3-12
system-specific reference, 4-5, 10-2, 10-3, 10-29,	making permanent, 3-13
10-45	read-only, 3-18
10-45	
	rolling back, E-75
T	subdividing with savepoints, 3-14
	terminating, 3-13
table locks	undoing, 3-16
acquiring with LOCK TABLE, 3-19	undoing parts of, 3-14
effect of, 3-20	when rolled back automatically, 3-13
row share, 3-19	transient copies of objects, 17-4
when released, 3-20	transient objects, 17-4
tables	TRANSITIVE OTT parameter, 19-26
inserting rows into, <i>E-50</i>	truncated values
nested, 18-1	detecting, 6-3, 7-10
updating rows in, <i>E-84</i>	truncation error
terminal	when generated, 6-5
encoding scheme, 4-43	tuning, performance, B-1
termination, program	two-task
normal versus abnormal, 3-17	linking, <i>2-14</i>
threads, <i>E-20</i>	type checking at runtime, 17-18
	TYPE directive
allocating context, 11-6, E-20	examples, <i>E-84</i>
enabling, 11-6, E-37	· · · · · · · · · · · · · · · · · · ·
freeing context, 11-7, E-21	type inheritance, 17-2
use context, 11-6	example, 17-20
THREADS	IS OF type operator, 17-2
precompiler option, 10-46, 11-5	example, 17-2
TIMESTAMP datatype, 4-10	TREAT operator, 17-2
	example, <del>17-3</del>

TYPE CODE	usernames
TYPE_CODE precompiler option, 10-46	defining, 3-2 using C structures, 17-33
precompiler option, 10-40	USING clause
	in CONNECT statement, 3-6
U	in the EXECUTE statement, 13-11
undo a transaction E 75	of FETCH statement, <i>E-44</i>
undo a transaction, <i>E-75</i> Unicode character set, <i>5-7</i>	of OPEN statement, E-70
unions	purpose of, 13-11
cannot be nested in host structures, 4-37	using collection types, 17-34
not permitted as host structures, 4-37	using dbstring
universal ROWID, 4-6	Oracle Net database specification string, <i>E-20</i>
UNIX	Using REFs in Embedded SQL, 17-34
linking a Pro*C application under, 1-9	
UNSAFE_NULL precompiler option, 10-46	V
UNSIGNED datatype, 4-8	<u> </u>
UPDATE SQL statement, <i>E-84</i>	V variable in SQLDA
UPDATE statement	how value is set, 15-6
embedded SQL examples, <i>E-87</i>	purpose of, 15-6
example of, 6-7	V7
purpose of, 6-7	value of DBMS option, 10-20, 10-21
SET clause in, 6-8	VALUES clause
using host arrays in, 8-12	in INS, 6-7
WHERE clause in, 6-8	of embedded SQL INSERT statement, <i>E-52</i>
updating	of INSERT statement, E-52
rows in tables and views, <i>E-84</i>	use of subqueries in, 6-7
USE	VAR directive
thread context, 11-6, E-24	examples, <i>E-89</i>
user configuration file	VAR SQL directive, <i>E-87</i> VAR statement
to set precompiler options, 10-4 user exits	
calling from a SQL*Forms trigger, 20-5	syntax for, <i>5-10</i> VARCHAR
common uses for, 20-1	arrays of, 8-2
example of, 20-6	VARCHAR datatype, 4-6
kinds of statements allowed in, <i>20-2</i>	VARCHAR precompiler option, 10-48
linking into IAP, 20-9	VARCHAR pseudotype
meaning of codes returned by, 20-6	requirements for using with PL/SQL, 7-8
naming, <i>20-10</i>	VARCHAR variables
passing parameters to, 20-5	advantages of, 4-20
requirements for variables in, 20-2	declaring, 4-20
running the GENXTB form, 20-9	length member in, 4-20
running the GENXTB utility for, 20-9	must be passed to a function by reference,
use of WHENEVER statement in, 20-6	4-22
User Exits, D-2	specifying the length of, 4-20
user session	structure of, 4-20
definition of, 3-11	using macros to define length of, 5-21
user-defined record, 7-4	versus character arrays, 5-7
user-defined stored function	VARCHAR2 datatype, 4-4, 5-11
used in WHERE clause, 6-8	variables, 2-4
user-defined type equivalencing, <i>E-83</i>	cursor, 4-25
USERID option	host, 18-3
when required, 10-47	indicator, 18-3
USERID OTT parameter, 19-23	VARNUM datatype, 4-6
USERID precompiler option, 10-47 using with the SQLCHECK option, C-3	VARRAW datatype, 4-8
asing with the Square rect option, C-3	varrays creation, 18-2
	· · · · · · · · · · · · · · · · · · ·

varying length arrays, 18-2	WHERE clause (continued)
VERSION precompiler option, 10-48, 17-16	if omitted, 6-8
views	in DELETE statements, 6-8
inserting rows into, <i>E-50</i>	in SELECT statements, 6-6
updating rows in, <i>E-84</i>	in UPDATE statements, 6-8
VMS	purpose of, 6-8
linking a precompiler application, 1-9	search condition in, 6-8
3 1 1 11 /	WHERE CURRENT OF clause
14/	CURRENT OF clause, 6-21
W	WITH HOLD
warning flags	clause of DECLARE CURSOR statement,
use in error reporting, 9-11	6-11
WHENEVER directive	WORK option
examples, <i>E-90</i>	of COMMIT statement, <i>E-18</i>
WHENEVER SQL directive, <i>E-89</i>	of ROLLBACK statement, <i>E-75</i>
WHENEVER SQL directive, 2-09 WHENEVER statement	
automatic checking of SQLCA with, <i>9-19</i>	V
CONTINUE action in, 9-20	X
DO action in, 9-20	X variable in SQLDA
DO BREAK action in, 9-20	how value is set, 15-9
DO CONTINUE action in, 9-20	purpose of, 15-9
examples of, 9-20	X/Open, 5-43
GOTO action in, 9-20	application development, 5-42
	• • • • • • • • • • • • • • • • • • • •
guidelines for, 9-22	XA interface, 5-43
maintaining addressability for, 9-24	
NOT FOUND condition in, 9-19	Υ
overview of, 2-6	
scope of, 9-22	Y variable in SQLDA
SQLERROR condition in, 9-19	how value is set, 15-9
SQLWARNING condition in, 9-19	purpose of, 15-9
STOP action in, 9-20	
use in user exits, 20-6	Z
using to avoid infinite loops, 9-23	<u>-</u>
using to handle end-of-data conditions, 9-23	Z variable in SQLDA
where to place, 9-23	how value is set, 15-9
WHERE clause	purpose of, 15-9
host arrays in, 8-15	

