

Oracle8i

Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド

リリース 8.1

2000 年 2 月

Oracle8i Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド, リリース 8.1

原本名: Pro*C/C++ Precompiler Programmer's Guide, Release 8.1.6

原著者: Jack Melnick, Tom Portfolio, Tim Smith

協力者: Ruth Baylis, Paul Lane, Valarie Moore, Bill Bailey, Julie Basu, Brian Becker, Beethoven Cheng, Michael Chiocca, Pierre Dufour, Nancy Ikeda, Alex Keh, Thomas Kurian, Shiao-Yen Lin, Vidya Nagaraj, Jacqui Pons, Ajay Popat, Ekkehard Rohwedder, Pamela Rothman, Alan Thiesen, Gael Stevens

Copyright © 1996, 1999, Oracle Corporation. All rights reserved.

制限付権利の説明

プログラム (ソフトウェアおよびドキュメントを含む) の使用、複製または開示は、オラクル社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権に関する法律により保護されています。

当プログラムのリバース・エンジニアリング等は禁止されています。

このドキュメントの情報は、予告なしに変更されることがあります。オラクル社は本ドキュメントの無謬性を保証しません。

* オラクル社とは、Oracle Corporation (米国オラクル) または日本オラクル株式会社 (日本オラクル) を指します。

危険な用途への使用について

オラクル社製品は、原子力、航空産業、大量輸送、医療あるいはその他の危険が伴うアプリケーションを用途として開発されておりません。オラクル社製品を上述のようなアプリケーションに使用することについての安全確保は、顧客各位の責任と費用により行ってください。万一かかる用途での使用によりクレームや損害が発生いたしましても、日本オラクル株式会社と開発元である Oracle Corporation (米国オラクル) およびその関連会社は一切責任を負いかねます。当プログラムを米国国防総省の米国政府機関に提供する際には、『Restricted Rights』と共に提供してください。この場合次の Notice が適用されます。

Restricted Rights Notice

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

このドキュメントに記載されているその他の会社名および製品名は、あくまでその製品および会社を識別する目的にのみ使用されており、それぞれの所有者の商標または登録商標です。

目次

はじめに	xxvii
このマニュアルの説明事項	xxviii
対象読者	xxviii
Pro*C/C++ マニュアルの構成	xxviii
表記規則	xxxix
BNF 表記法	xxxix
ANSI/ISO 準拠	xxxix
要件	xxxix
工業規格への Oracle Pro*C/C++ プリコンパイラの準拠性	xxxix
準拠性	xxxix
準拠の証示	xxxix
FIPS フラガー	xxxix
以前のリリースからのアプリケーションの移行	xxxix
 1 概要	
Oracle プリコンパイラ	1-2
Oracle Pro*C/C++ プリコンパイラを使用する理由	1-3
SQL を使用する理由	1-3
PL/SQL を使用する理由	1-4
Pro*C/C++ プリコンパイラの利点	1-5
よく聞かれる質問 (FAQ)	1-7
VARCHAR について説明してください。	1-7
Pro*C/C++ は、Oracle Call Interface のコールを生成しますか。	1-7
Pro*C/C++ を使用せず、単に SQLLIB のコールを使用してコーディングできますか。	1-7
PL/SQL のストアド・プロシージャを Pro*C/C++ プログラムからコールできますか。	1-8

C++ のコードを作成し、Pro*C/C++ を使用してプリコンパイルできますか。	1-8
SQL 文の任意の場所でバインド変数を使用できますか。	1-8
Pro*C/C++ の文字処理がよくわかりません。	1-8
文字ポインタについて説明してください。何か特別なことはありますか。	1-9
Pro*C/C++ で SPOOL が動作しない理由を説明してください。	1-9
サンプル・プログラムのオンライン版はどこにありますか。	1-9
アプリケーションをコンパイルしてリンクする方法を教えてください。	1-9
Pro*C/C++ では構造体をホスト変数として使用できますか。	1-10
Pro*C/C++ に再帰関数を入れることは、その関数内に埋込み SQL を使用した場合、 可能ですか。	1-10
Pro*C/C++ のすべてのリリースを、Oracle Server のすべてのバージョンで使用できますか。	1-10
アプリケーションを Oracle8i で実行すると、いつも ORA-01405 エラー (取り出した列の値が NULL です。) が発生します。	1-10
すべての SQLLIB 関数はプライベート関数ですか。	1-10
新しいオブジェクト型は、Oracle8i でどのようにサポートされていますか。	1-12

2 プリコンパイラ の概念

埋込み SQL プログラミングの主要概念	2-2
埋込み SQL 文	2-2
埋込み SQL の構文	2-4
静的 SQL 文と動的 SQL 文の対比	2-4
埋込み PL/SQL ブロック	2-5
ホスト変数および標識変数	2-5
Oracle のデータ型	2-6
配列	2-6
データ型の同値化	2-6
プライベート SQL 領域、カーソルおよびアクティブ・セット	2-6
トランザクション	2-7
エラーおよび警告	2-7
埋込み SQL アプリケーションの開発ステップ	2-8
プログラミングのガイドライン	2-9
コメント	2-9
定数	2-9
宣言文	2-9
デリミタ	2-10

ファイルの長さ	2-11
関数プロトタイプ	2-11
ホスト変数名	2-12
行の継続	2-12
行の長さ	2-12
MAXLITERAL デフォルト値	2-12
演算子	2-13
文終了記号	2-13
条件付きプリコンパイル	2-13
記号の定義	2-14
例	2-14
分割プリコンパイル	2-15
ガイドライン	2-15
コンパイルとリンク	2-16
サンプル表	2-16
サンプル・データ	2-17
サンプル・プログラム: 単純な問合せ	2-17

3 データベースの概念

データベースへの接続	3-2
ALTER AUTHORIZATION 句を使用したパスワードの変更	3-3
Net8 を利用した接続	3-4
自動接続	3-4
アドバンスト・コネクション・オプション	3-5
予備知識	3-5
同時ログイン	3-6
デフォルトのデータベースおよび接続	3-7
明示的接続	3-7
暗黙的接続	3-12
トランザクション用語の定義	3-14
トランザクションがデータベースを保護する方法	3-15
トランザクションの開始・終了方法	3-15
COMMIT 文の使用方法	3-16
SAVEPOINT 文の使用方法	3-17
ROLLBACK 文の使用方法	3-19
文レベルのロールバック	3-20

RELEASE オプションの使用方法	3-21
SET TRANSACTION 文の使用方法	3-21
デフォルト・ロックのオーバーライド	3-22
FOR UPDATE OF の使用方法	3-22
LOCK TABLE の使用方法	3-23
COMMIT をまたぐ FETCH	3-24
分散トランザクションの処理	3-24
ガイドライン	3-25
アプリケーションの設計	3-25
ロックの取得	3-26
PL/SQL の使用方法	3-26

4 データ型とホスト変数

Oracle のデータ型	4-2
内部データ型	4-2
外部データ型	4-4
ホスト変数	4-11
ホスト変数の宣言	4-11
ホスト変数の参照	4-14
標識変数	4-15
キーワード INDICATOR の使用	4-15
例	4-16
ガイドライン	4-17
Oracle 制限事項	4-17
VARCHAR 変数	4-17
VARCHAR 変数の宣言	4-17
VARCHAR 変数の参照	4-19
VARCHAR 変数に NULL を戻す	4-19
VARCHAR 変数を使用した NULL の挿入	4-19
VARCHAR 変数を関数に渡す	4-20
VARCHAR 配列コンポーネントの長さを調べる方法	4-20
サンプル・プログラム : sqlvcp() の使用	4-21
カーソル変数	4-24
カーソル変数の宣言	4-25
カーソル変数の割当て	4-25

カーソル変数のオープン	4-26
カーソル変数のクローズと解放	4-28
OCIでのカーソル変数の使用（リリース7のみ）	4-29
制限	4-30
サンプル・プログラム	4-31
CONTEXT 変数	4-34
汎用 ROWID	4-36
SQLRowidGet()	4-37
ホスト構造体	4-37
ホスト構造体と配列	4-39
PL/SQL レコード	4-39
ネストした構造体と共用体	4-39
ホスト標識構造体	4-40
サンプル・プログラム : カーソルとホスト構造体	4-41
ポインタ変数	4-43
ポインタ変数の宣言	4-44
ポインタ変数の参照	4-44
構造体ポインタ	4-44
各国語サポート	4-45
NCHAR 変数	4-47
CHARACTER SET [IS] NCHAR_CS	4-48
環境変数 NLS_NCHAR	4-48
VAR 内の CONVBUSZ 句	4-49
埋込み SQL 内の文字列	4-49
文字列の制限事項	4-49
標識変数	4-50

5 上級トピック

文字データの処理	5-2
プリコンパイラ・オプション CHAR_MAP	5-2
CHAR_MAP オプションのインラインでの使用方法	5-3
DBMS オプションおよび CHAR_MAP オプションの影響	5-3
VARCHAR 変数およびポインタ	5-7
Unicode 変数	5-9
データ型変換	5-11

データ型の同値化	5-11
ホスト変数の同値化	5-12
ユーザー定義型同値化	5-13
CHARF 外部データ型	5-14
EXEC SQL VAR と TYPE 宣言文の利用	5-14
Sample4.pc: データ型の同値化	5-15
C プリプロセッサ	5-27
Pro*C/C++ プリプロセッサの機能	5-27
プリプロセッサ宣言文	5-28
ORA_PROC マクロ	5-29
ヘッダー・ファイルの格納場所の指定	5-29
プリプロセッサの例	5-30
#include に使用することができない SQL 文	5-32
SQLCA、ORACA および SQLDA の組込み	5-32
EXEC SQL INCLUDE および #include の要約	5-33
定義済マクロ	5-34
インクルード・ファイル	5-34
プリコンパイル済のヘッダー・ファイル	5-34
プリコンパイル済のヘッダー・ファイルの作成	5-35
プリコンパイル済のヘッダー・ファイルの使用	5-35
例	5-36
オプションの効果	5-37
使用上の注意	5-40
Oracle プリプロセッサ	5-40
記号の定義	5-40
Oracle プリプロセッサの例	5-41
数値定数の評価	5-41
Pro*C/C++ での数値定数の使用	5-42
数値定数の規則および例	5-42
OCI リリース 8 の SQLLIB 拡張相互運用性	5-43
ランタイム・コンテキストと OCI リリース 8 環境の確立と終了	5-43
OCI リリース 8 環境ハンドルのパラメータ	5-44
OCI リリース 8 とのインタフェース	5-44
SQLEnvGet()	5-44
SQLSvcCtxGet()	5-45

OCI コールの埋込み	5-46
埋込み (OCI リリース 7) Oracle コール	5-47
LDA の設定	5-48
リモートの複数接続	5-48
SQLLIB パブリック関数の新しい名前	5-49
X/Open アプリケーションの開発	5-51
Oracle 固有の項目	5-53

6 埋込み SQL

ホスト変数の使用	6-2
出力ホスト変数および入力ホスト変数	6-2
標識変数の使用	6-3
NULL 値の挿入	6-4
戻された NULL 値の処理	6-5
NULL 値のフェッチ	6-5
NULL のテスト	6-5
切り捨てられた値のフェッチ	6-6
基本的な SQL 文	6-6
SELECT 文の使用法	6-7
INSERT 文の使用法	6-8
UPDATE 文の使用法	6-9
DELETE 文の使用法	6-10
WHERE 句の使用法	6-10
DML 戻り句	6-10
カーソルの使用法	6-11
DECLARE CURSOR 文の使用法	6-11
OPEN 文の使用法	6-12
FETCH 文の使用法	6-13
CLOSE 文の使用法	6-14
CLOSE_ON_COMMIT プリコンパイラ・オプション	6-14
PREFETCH プリコンパイラ・オプション	6-15
オブティマイザ・ヒント	6-15
ヒントの発行	6-16
CURRENT OF 句の使用法	6-16
制限	6-17

すべてのカーソル文の使用方法	6-17
完全な例	6-18
位置決定済み更新	6-19

7 埋込み PL/SQL

PL/SQL の利点	7-2
パフォーマンス向上	7-2
Oracle との統合	7-2
カーソル FOR ループ	7-2
プロシージャとファンクション	7-3
パッケージ	7-4
PL/SQL 表	7-4
ユーザー定義のレコード	7-5
埋込み PL/SQL ブロック	7-6
ホスト変数の使用	7-6
例	7-7
より複雑な例	7-8
VARCHAR 疑似型	7-10
制限	7-11
標識変数の使用	7-12
NULL の処理	7-12
切捨て値の処理	7-13
ホスト配列の使用	7-13
ARRAYLEN 文	7-16
オプション・キーワード EXECUTE	7-17
埋込み PL/SQL のカーソルの使用方法	7-18
ストアド PL/SQL および Java サブプログラム	7-19
ストアド・サブプログラムの生成	7-19
ストアド PL/SQL または Java サブプログラムのコール	7-21
ストアド・サブプログラムに関する情報を得る方法	7-27
外部プロシージャ	7-27
外部プロシージャの制限	7-28
外部プロシージャの作成	7-29
SQLExtProcError()	7-30
動的 SQL の使用	7-31

8 ホスト配列

配列を使用する理由	8-2
ホスト配列の宣言	8-2
制限	8-2
配列の最大サイズ	8-2
SQL 文での配列の使用	8-3
ホスト配列の参照	8-3
標識配列の使用方法	8-3
Oracle 制限事項	8-4
ANSI 制限事項および要件	8-4
配列への選択	8-4
カーソルのフェッチ	8-5
<code>sqlca.sqlerrd[2]</code> の使用方法	8-6
FETCH される行数	8-6
サンプル・プログラム 3: ホスト配列	8-7
制限	8-10
NULL 値のフェッチ	8-10
切り捨てられた値のフェッチ	8-10
配列での挿入	8-10
制限	8-11
配列での更新	8-11
制限	8-12
配列での削除	8-12
制限	8-13
FOR 句の使用方法	8-13
制限	8-14
WHERE 句の使用方法	8-15
構造体配列	8-16
構造体の配列の使用方法	8-16
構造体の配列の制限	8-17
構造体の配列の宣言	8-17
標識変数の使用方法	8-19
構造体の配列へのポインタの宣言	8-20
例	8-20
CURRENT OF の疑似実行	8-26

9 ランタイム・エラーの処理

エラー処理の必要性	9-2
エラー処理の代替手段	9-2
状態変数	9-2
SQL コミュニケーション領域	9-2
SQLSTATE 状態変数	9-3
SQLSTATE の宣言	9-4
SQLSTATE の値	9-4
SQLSTATE の使用	9-14
SQLCODE の宣言	9-15
SQLCA を使用したエラー報告の主要コンポーネント	9-15
ステータス・コード	9-15
警告フラグ	9-16
処理済の行数	9-16
解析エラー・オフセット	9-16
エラー・メッセージ・テキスト	9-17
SQL コミュニケーション領域 (SQLCA) の使用	9-17
SQLCA の宣言	9-17
SQLCA に含まれているもの	9-18
SQLCA の構造	9-20
PL/SQL の考慮事項	9-23
エラー・メッセージの全文の取得	9-23
WHENEVER 宣言文の使用	9-24
条件	9-25
アクション	9-25
例	9-26
DO BREAK と DO CONTINUE の利用	9-27
WHENEVER 文の適用範囲	9-29
WHENEVER のガイドライン	9-29
SQL 文のテキスト取得	9-32
制限	9-34
サンプル・プログラム	9-34
ORACLE コミュニケーション領域 (ORACA) の使用	9-35
ORACA の宣言	9-35
ORACA を使用可能にする	9-35

ORACA に含まれているもの	9-36
ランタイム・オプションの選択	9-38
ORACA の構造	9-38
ORACA の使用例	9-41

10 プリコンパイラのオプション

プリコンパイラのコマンド	10-2
大文字と小文字の区別	10-2
プリコンパイラのオプション	10-3
構成ファイル	10-3
オプション値の優先順位	10-4
マクロ・オプションおよびマイクロ・オプション	10-5
プリコンパイル中の状況	10-6
オプションの適用範囲	10-6
早見表	10-6
オプションの入力	10-9
コマンドライン	10-9
インライン	10-9
プリコンパイラ・オプションの使用	10-11
AUTO_CONNECT	10-11
CHAR_MAP	10-11
CLOSE_ON_COMMIT	10-12
CODE	10-13
COMP_CHARSET	10-14
CONFIG	10-14
CPP_SUFFIX	10-15
DBMS	10-15
DEF_SQLCODE	10-17
DEFINE	10-17
DURATION	10-19
DYNAMIC	10-19
ERRORS	10-20
ERRTYPE	10-20
FIPS	10-21
HEADER	10-22

HOLD_CURSOR	10-23
INAME	10-24
INCLUDE	10-24
INTYPE	10-25
LINES	10-26
LNAME	10-27
LTYPE	10-27
MAXLITERAL	10-28
MAXOPENCURSORS	10-28
MODE	10-30
NLS_CHAR	10-31
NLS_LOCAL	10-31
OBJECTS	10-32
ONAME	10-32
ORACA	10-33
PAGELEN	10-33
PARSE	10-34
PREFETCH	10-35
RELEASE_CURSOR	10-35
SELECT_ERROR	10-36
SQLCHECK	10-37
SYS_INCLUDE	10-37
THREADS	10-38
TYPE_CODE	10-39
UNSAFE_NULL	10-39
USERID	10-40
VARCHAR	10-40
VERSION	10-41

11 マルチスレッド・アプリケーション

スレッド	11-2
Pro*C/C++ のランタイム・コンテキスト	11-2
ランタイム・コンテキストの使用モデル	11-5
単一のランタイム・コンテキストを共有する複数のスレッド	11-5
複数のランタイム・コンテキストを共有する複数のスレッド	11-7

マルチスレッド・アプリケーションのユーザー・インタフェース	11-8
THREADS オプション	11-8
埋込み SQL 文と宣言文	11-8
CONTEXT USE の例	11-10
プログラミングの考慮事項	11-12
マルチスレッドの例	11-12

12 C++ アプリケーション

C++ サポートの理解	12-2
特殊なマクロ処理は不要	12-2
C++ のプリコンパイル	12-2
コードの生成	12-3
コードの解析	12-4
出力ファイル名の拡張子	12-5
システム・ヘッダー・ファイル	12-5
サンプル・プログラム	12-5
cppdemo1.pc	12-5
cppdemo2.pc	12-9
cppdemo3.pc	12-13

13 Oracle の動的 SQL

動的 SQL	13-2
動的 SQL の長所と短所	13-2
動的 SQL の使用	13-2
動的 SQL 文の要件	13-3
動的 SQL 文の処理方法	13-3
動的 SQL の使用方法	13-4
方法 1	13-4
方法 2	13-5
方法 3	13-5
方法 4	13-5
ガイドライン	13-6
方法 1 の使用方法	13-8
サンプル・プログラム : 動的 SQL 方法 1	13-9
方法 2 の使用方法	13-12

USING 句	13-13
サンプル・プログラム：動的 SQL 方法 2	13-14
方法 3 の使用方法	13-17
PREPARE	13-18
DECLARE	13-18
OPEN	13-19
FETCH	13-19
CLOSE	13-19
サンプル・プログラム：動的 SQL 方法 3	13-20
方法 4 の使用方法	13-23
SQLDA の必要性	13-24
DESCRIBE 文	13-24
SQLDA	13-25
Oracle 方法 4 の実行	13-26
制限	13-26
DECLARE STATEMENT 文の使用法	13-26
ホスト配列の使用	13-27
PL/SQL の使用方法	13-27
方法 1 の場合	13-28
方法 2 の場合	13-28
方法 3 の場合	13-28
Oracle 方法 4 の場合	13-28
注意	13-29

14 ANSI 動的 SQL

ANSI 動的 SQL の基本	14-2
プリコンパイラのオプション	14-2
ANSI SQL 文の概要	14-3
サンプル・コード	14-6
Oracle 拡張機能	14-7
参照セマンティクス	14-7
配列を使用したバルク操作	14-8
構造体配列のサポート	14-10
オブジェクト型のサポート	14-10
ANSI 動的 SQL プリコンパイラ・オプション	14-10

動的 SQL 文の完全な構文	14-12
ALLOCATE DESCRIPTOR	14-12
DEALLOCATE DESCRIPTOR	14-13
GET DESCRIPTOR	14-13
SET DESCRIPTOR	14-17
PREPARE の使用	14-20
DESCRIBE INPUT	14-21
DESCRIBE OUTPUT	14-21
EXECUTE	14-22
EXECUTE IMMEDIATE の使用	14-23
DYNAMIC DECLARE CURSOR の使用	14-24
OPEN カーソル	14-24
FETCH	14-25
動的カーソルの CLOSE	14-26
旧バージョンの Oracle 動的 method 4 との相違点	14-26
制限	14-27
サンプル・プログラム	14-27
ansidyn1.pc	14-27
ansidyn2.pc	14-36

15 Oracle の動的 SQL 方法 4

方法 4 の特殊要件	15-2
方法 4 が特別な理由	15-2
Oracle に必要な情報	15-2
情報の格納位置	15-3
SQLDA の参照方法	15-3
情報の取得方法	15-4
SQLDA の説明	15-4
SQLDA の用途	15-4
複数の SQLDA	15-4
SQLDA の宣言	15-5
SQLDA の割当て	15-5
SQLDA 変数の使用	15-6
N 変数	15-6
V 変数	15-7

L 変数	15-7
T 変数	15-8
I 変数	15-9
F 変数	15-9
S 変数	15-9
M 変数	15-10
C 変数	15-10
X 変数	15-10
Y 変数	15-10
Z 変数	15-10
予備知識	15-11
データの変換	15-11
データ型の強制変換	15-14
NULL/NOT NULL データ型の処理	15-16
基本ステップ	15-17
各ステップの詳細	15-18
ホスト文字列の宣言	15-19
SQLDA の宣言	15-19
記述子用の記憶領域の割当て	15-20
DESCRIBE への最大数の設定	15-20
ホスト文字列への問合せテキストの設定	15-23
ホスト文字列からの問合せの PREPARE	15-23
カーソルの宣言	15-23
バインド変数の DESCRIBE	15-23
プレースホルダの最大数のリセット	15-25
バインド変数の値の取得と記憶領域の割当て	15-25
カーソルの OPEN	15-27
選択リストの DESCRIBE	15-27
選択リスト項目の最大数のリセット	15-29
各選択リスト項目の長さでデータ型のリセット	15-29
アクティブ・セットからの行の FETCH	15-31
選択リストの値の取得と処理	15-31
記憶域の割当て解除	15-33
カーソルの CLOSE	15-33
ホスト配列の使用	15-33

sample12.pc	15-36
サンプル・プログラム: 動的 SQL 方法 4	15-36

16 ラージ・オブジェクト (LOB)

LOB	16-2
内部 LOB	16-2
外部 LOB	16-2
BFILE のセキュリティ	16-2
LOB 対 LONG および LONG RAW	16-3
LOB ロケータ	16-3
テンポラリ LOB	16-3
LOB バッファリング・サブシステム	16-4
プログラムでの LOB の使用方法	16-5
LOB にアクセスする 3 種類の方法	16-5
アプリケーションの LOB ロケータ	16-6
LOB の初期化	16-7
LOB 文のルール	16-9
すべての LOB 文に対するルール	16-9
LOB バッファリング・サブシステムに対するルール	16-9
ホスト変数に対するルール	16-11
LOB 文	16-11
APPEND	16-11
ASSIGN	16-12
CLOSE	16-13
COPY	16-13
CREATE TEMPORARY	16-14
DISABLE BUFFERING	16-15
ENABLE BUFFERING	16-15
ERASE	16-16
FILE CLOSE ALL	16-17
FILE SET	16-17
FLUSH BUFFER	16-18
FREE TEMPORARY	16-18
LOAD FROM FILE	16-19
OPEN	16-20

READ	16-21
TRIM	16-23
WRITE	16-23
DESCRIBE	16-25
LOB およびナビゲーション・インタフェース	16-28
一時オブジェクト	16-28
永続オブジェクト	16-28
ナビゲーション・インタフェースの例	16-28
LOB プログラムの例	16-30
BLOB の READ およびファイル書込みの例	16-30
ファイルの読み込みおよび BLOB の WRITE の例	16-31
lobdemo1.pc	16-34

17 オブジェクト

オブジェクトの概要	17-2
オブジェクト型	17-2
REF	17-2
Pro*C/C++ でのオブジェクト型の使用	17-3
NULL 標識	17-3
オブジェクト・キャッシュ	17-3
永続オブジェクト対一時コピー	17-4
結合インタフェース	17-4
結合インタフェースを使用する場合	17-4
ALLOCATE	17-5
FREE	17-5
CACHE FREE ALL	17-6
結合インタフェースによるオブジェクトへのアクセス	17-6
ナビゲーション・インタフェース	17-7
ナビゲーション・インタフェースを使用する場合	17-8
ナビゲーション文に使用されるルール	17-9
OBJECT CREATE	17-9
OBJECT Deref	17-10
OBJECT RELEASE	17-11
OBJECT DELETE	17-11
OBJECT UPDATE	17-11

OBJECT FLUSH	17-12
オブジェクトへのナビゲーションル・アクセス	17-12
オブジェクト属性と C 型の変換	17-14
OBJECT SET	17-14
OBJECT GET	17-16
オブジェクト・オプションの設定 / 取得	17-17
CONTEXT OBJECT OPTION SET	17-17
CONTEXT OBJECT OPTION GET	17-18
オブジェクトに対する新しいプリコンパイラ・オプション	17-19
VERSION	17-19
DURATION	17-19
OBJECTS	17-20
INTYPE	17-20
ERRTYPE	17-21
オブジェクトに対する SQLCHECK のサポート	17-21
実行時のタイプ・チェック	17-21
Pro*C/C++ のオブジェクト例	17-22
結合アクセス	17-22
ナビゲーションル・アクセス	17-23
ナビゲーションル・アクセスのサンプル・コード	17-24
C 構造体の使用	17-31
REF の使用	17-32
REF の C 構造体の生成	17-32
REF の宣言	17-32
埋込み SQL での REF の使用	17-33
OCIDate、OCIString、OCINumber および OCIRaw の使用	17-33
OCIDate、OCIString、OCINumber、OCIRaw の宣言	17-33
埋込み SQL での OCI 型の使用	17-33
OCI 型の操作	17-34
Pro*C/C++ の新しいデータベース型の概要	17-34
動的 SQL での Oracle8i データ型使用の制限	17-36

18 コレクション

コレクション	18-2
NESTED TABLE	18-2

VARRAY (可変長配列)	18-3
C およびコレクション	18-3
コレクションの記述子	18-3
ホスト変数と標識変数の宣言	18-4
コレクションの操作	18-4
アクセスのルール	18-5
標識変数	18-5
OBJECT GET および SET	18-6
COLLECTION 文	18-7
COLLECTION GET	18-7
COLLECTION SET	18-9
COLLECTION RESET	18-11
COLLECTION APPEND	18-11
COLLECTION TRIM	18-12
COLLECTION DESCRIBE	18-13
コレクションを使用する場合の規則	18-15
コレクション・サンプル・コード	18-16
型および表の作成	18-16
GET および SET の例	18-18
DESCRIBE の例	18-19
RESET の例	18-20
サンプル・プログラム :coldemo1.pc	18-22

19 オブジェクト型トランスレータ

OTT 概要	19-2
オブジェクト型トランスレータ	19-2
データベースにおける型の作成	19-4
OTT の起動	19-5
OTT コマンドライン	19-6
Intype ファイル	19-8
OTT データ型のマップ	19-9
NULL 標識構造体	19-14
Outtype ファイル	19-15
OCI アプリケーションでの OTT の使用	19-17
OCI によるオブジェクトのアクセスと操作	19-18

初期化関数のコール	19-19
初期化関数のタスク	19-21
Pro*C/C++ アプリケーションでの OTT の使用	19-21
OTT 参照	19-23
OTT コマンドライン構文	19-24
OTT パラメータ	19-25
OTT パラメータの使用場所	19-28
Intype ファイルの構造	19-29
ネストした #include ファイル生成	19-31
SCHEMA_NAMES の使用方法	19-33
デフォルト名のマッピング	19-35
制限	19-36

20 ユーザー・イグジット

ユーザー・イグジット	20-2
ユーザー・イグジットを作成する理由	20-3
ユーザー・イグジットの開発	20-3
ユーザー・イグジットの作成	20-4
変数の要件	20-4
IAF GET 文	20-4
IAF PUT 文	20-5
ユーザー・イグジットのコール	20-6
ユーザー・イグジットへのパラメータの引渡し	20-7
フォームへの値のリターン	20-8
IAP 定数	20-8
SQLIEM 関数の使用方法	20-8
WHENEVER の使用方法	20-9
例	20-9
ユーザー・イグジットのプリコンパイルおよびコンパイル	20-10
サンプル・プログラム: ユーザー・イグジット	20-10
GENXTB ユーティリティの使用方法	20-12
ユーザー・イグジットの SQL*Forms へのリンク	20-13
ガイドライン	20-13
イグジットの命名	20-13
Oracle への接続	20-13
I/O コールの発行	20-14

ホスト変数の使用方法	20-14
表の更新	20-14
コマンドの発行	20-14
EXEC TOOLS 文	20-14
Toolset ユーザー・イグジットの作成	20-14
EXEC TOOLS SET	20-15
EXEC TOOLS GET	20-15
EXEC TOOLS SET CONTEXT	20-16
EXEC TOOLS GET CONTEXT	20-16
EXEC TOOLS MESSAGE	20-17

A 新機能

構造体の配列	A-2
プリコンパイル済ヘッダー・ファイル	A-2
CALL 文	A-2
実行時のパスワードの変更	A-2
各国語キャラクタ・セットのサポート	A-2
CHAR_MAP プリコンパイラ・オプション	A-3
SQLLIB 関数の新規名	A-3
WHENEVER 文の新規アクション	A-3
オブジェクト型のサポート	A-3
オブジェクト型トランスレータ	A-3
LOB サポート	A-4
ANSI 動的 SQL	A-4
コレクション	A-4
その他のトピック	A-4
Unicode サポート	A-4
PREFETCH オプション	A-4
外部プロシージャ	A-4
PL/SQL からの Java のコール	A-5
DML 戻り句	A-5
汎用 ROWID	A-5
CONNECT 文の SYSDBA/SYSOPER 権限	A-5
CLOSE_ON_COMMIT プリコンパイラ・オプション	A-5
文字列	A-5
エラー・メッセージ・コード	A-6

LINES オプション	A-6
以前のリリースからの移行	A-6
B 予約語、キーワードおよび名前領域	
予約語およびキーワード	B-2
Oracle の予約名前領域	B-4
C パフォーマンスの最適化	
パフォーマンスを低下させる原因	C-2
パフォーマンスの改善方法	C-2
ホスト配列の使用	C-3
埋込み PL/SQL の利用	C-3
SQL 文の最適化	C-4
オブティマイザ・ヒント	C-5
トレース機能	C-5
索引の使用	C-6
行レベル・ロックの利用	C-6
不要な解析の排除	C-6
明示的なカーソルの操作	C-7
カーソル管理オプションの使用	C-8
D 構文検査と意味検査	
構文検査と意味検査	D-2
検査の種類および範囲の制御	D-2
SQLCHECK=SEMANTICS の指定	D-3
意味検査の使用許可	D-3
SQLCHECK=SYNTAX の指定	D-5
SQLCHECK オプションの入力	D-5
E システム固有の参照	
システム固有の情報	E-2
標準ヘッダー・ファイルの位置	E-2
C コンパイラ用組込みファイルの位置指定	E-2
ANSI C サポート	E-2
構造体コンポーネントの位置合せ	E-2

整数と ROWID のサイズ	E-2
バイトの並び	E-2
Oracle8i への接続	E-3
XA ライブラリでのリンク	E-3
Pro*C/C++ 実行モジュールの位置	E-3
システム構成ファイル	E-3
INCLUDE オプションの構文	E-3
コンパイルとリンク	E-3
ユーザー・イグジット	E-3

F 埋込み SQL 文および宣言文

プリコンパイラの宣言文と埋込み SQL 文の概要	F-5
文の説明	F-9
構文図の読み方	F-9
必須のキーワードとパラメータ	F-10
オプションのキーワードとパラメータ	F-11
構文ループ	F-11
複数パーツの図	F-11
Oracle の名前	F-12
文終了記号	F-12
ALLOCATE (実行可能埋込み SQL 拡張要素)	F-12
ALLOCATE DESCRIPTOR (実行可能埋込み SQL)	F-14
CACHE FREE ALL (実行可能埋込み SQL 拡張要素)	F-16
CALL (実行可能埋込み SQL)	F-17
CLOSE (実行可能埋込み SQL)	F-18
COLLECTION APPEND (実行可能埋込み SQL 拡張要素)	F-19
COLLECTION DESCRIBE (実行可能埋込み SQL 拡張要素)	F-20
COLLECTION GET (実行可能埋込み SQL 拡張要素)	F-22
COLLECTION RESET (実行可能埋込み SQL 拡張要素)	F-22
COLLECTION SET (実行可能埋込み SQL 拡張要素)	F-23
COLLECTION TRIM (実行可能埋込み SQL 拡張要素)	F-24
COMMIT (実行可能埋込み SQL)	F-24
CONNECT (実行可能埋込み SQL 拡張要素)	F-26
CONTEXT ALLOCATE (実行可能埋込み SQL 拡張要素)	F-28
CONTEXT FREE (実行可能埋込み SQL 拡張要素)	F-29
CONTEXT OBJECT OPTION GET (実行可能埋込み SQL 拡張要素)	F-30

CONTEXT OBJECT OPTION SET (実行可能埋込み SQL 拡張要素)	F-31
CONTEXT USE (Oracle 埋込み SQL 宣言文)	F-32
DEALLOCATE DESCRIPTOR (埋込み SQL 文)	F-34
DECLARE CURSOR (埋込み SQL 宣言文)	F-35
DECLARE DATABASE (Oracle 埋込み SQL 宣言文)	F-37
DECLARE STATEMENT (埋込み SQL 宣言文)	F-38
DECLARE TABLE (Oracle 埋込み SQL 宣言文)	F-39
DECLARE TYPE (Oracle 埋込み SQL 宣言文)	F-41
DELETE (実行可能埋込み SQL)	F-42
DESCRIBE (実行可能埋込み SQL 拡張要素)	F-46
DESCRIBE DESCRIPTOR (実行可能埋込み SQL)	F-47
ENABLE THREADS (実行可能埋込み SQL 拡張要素)	F-49
EXECUTE... END-EXEC (実行可能埋込み SQL 拡張要素)	F-50
EXECUTE (実行可能埋込み SQL)	F-51
EXECUTE DESCRIPTOR (実行可能埋込み SQL)	F-53
EXECUTE IMMEDIATE (実行可能埋込み SQL)	F-55
FETCH (実行可能埋込み SQL)	F-57
FETCH DESCRIPTOR (実行可能埋込み SQL)	F-59
FREE (実行可能埋込み SQL 拡張要素)	F-62
GET DESCRIPTOR (実行可能埋込み SQL)	F-63
INSERT (実行可能埋込み SQL)	F-65
LOB APPEND (実行可能埋込み SQL 拡張要素)	F-68
LOB ASSIGN (実行可能埋込み SQL 拡張要素)	F-69
LOB CLOSE (実行可能埋込み SQL 拡張要素)	F-70
LOB COPY (実行可能埋込み SQL 拡張要素)	F-70
LOB CREATE TEMPORARY (実行可能埋込み SQL 拡張要素)	F-71
LOB DESCRIBE (実行可能埋込み SQL 拡張要素)	F-71
LOB DISABLE BUFFERING (実行可能埋込み SQL 拡張要素)	F-73
LOB ENABLE BUFFERING (実行可能埋込み SQL 拡張要素)	F-73
LOB ERASE (実行可能埋込み SQL 拡張要素)	F-74
LOB FILE CLOSE ALL (実行可能埋込み SQL 拡張要素)	F-74
LOB FILE SET (実行可能埋込み SQL 拡張要素)	F-75
LOB FLUSH BUFFER (実行可能埋込み SQL 拡張要素)	F-75
LOB FREE TEMPORARY (実行可能埋込み SQL 拡張要素)	F-76
LOB LOAD (実行可能埋込み SQL 拡張要素)	F-77
LOB OPEN (実行可能埋込み SQL 拡張要素)	F-77
LOB READ (実行可能埋込み SQL 拡張要素)	F-78
LOB TRIM (実行可能埋込み SQL 拡張要素)	F-78

LOB WRITE (実行可能埋込み SQL 拡張要素)	F-79
OBJECT CREATE (実行可能埋込み SQL 拡張要素)	F-80
OBJECT DELETE (実行可能埋込み SQL 拡張要素)	F-81
OBJECT Deref (実行可能埋込み SQL 拡張要素)	F-82
OBJECT FLUSH (実行可能埋込み SQL 拡張要素)	F-83
OBJECT GET (実行可能埋込み SQL 拡張要素)	F-84
OBJECT RELEASE (実行可能埋込み SQL 拡張要素)	F-86
OBJECT SET (実行可能埋込み SQL 拡張要素)	F-87
OBJECT UPDATE (実行可能埋込み SQL 拡張要素)	F-88
OPEN (実行可能埋込み SQL)	F-89
OPEN DESCRIPTOR (実行可能埋込み SQL)	F-91
PREPARE (実行可能埋込み SQL)	F-93
REGISTER CONNECT (実行可能埋込み SQL 拡張要素)	F-95
ROLLBACK (実行可能埋込み SQL)	F-96
SAVEPOINT (実行可能埋込み SQL)	F-99
SELECT (実行可能埋込み SQL)	F-100
SET DESCRIPTOR (実行可能埋込み SQL)	F-103
TYPE (Oracle 埋込み SQL 宣言文)	F-106
UPDATE (実行可能埋込み SQL)	F-107
VAR (Oracle 埋込み SQL 宣言文)	F-111
WHENEVER (埋込み SQL 宣言文)	F-114

索引

はじめに

このマニュアルは総合的なユーザーズ・ガイドで、Oracle Pro*C/C++ プリコンパイラのリファレンスとして使用できます。Pro*C/C++ とともにデータベース言語 SQL および Oracle のプロシージャ型拡張要素 PL/SQL を使用して、Oracle8i データベースでデータを操作する方法を説明します。基礎を形成する概念から高度なプログラミング技法までを解説し、またコード例を使用しています。

この章のトピックは、次のとおりです。

- [このマニュアルの説明事項](#)
- [対象読者](#)
- [Pro*C/C++ マニュアルの構成](#)
- [表記規則](#)
- [ANSI/ISO 準拠](#)

このマニュアルの説明事項

このマニュアルでは、Oracle Pro*C/C++ プリコンパイラおよび埋込み SQL を、アプリケーション開発のプロセス全般で有効に利用する方法を説明します。Oracle の機能を活用するアプリケーションの設計および開発方法を説明します。また、短時間で埋込み SQL プログラムの作成方法を習得するのに役立ちます。

このマニュアルの特徴は、Pro*C/C++ および埋込み SQL を最大限に活用することに重点を置いていることです。これらのツールの習得に役立つように、このマニュアルではプログラムのパフォーマンスを改善する方法などのテクニックを掲載しています。また、埋込み SQL および埋込み PL/SQL についての理解を深め、その有用性を確認できるように、多数のプログラム例を掲載しています。

注意：このマニュアルには、インストレーションの指示および他のシステム固有の情報はありません。各システム固有の Oracle ドキュメントを参照してください。

対象読者

このマニュアルは、Oracle 環境で動作する新規のアプリケーションを開発する場合や、既存のアプリケーションを Oracle 環境用に変換する場合に役立ちます。また、このマニュアルは特にプログラムを対象として書かれていますが、Oracle Pro*C/C++ プリコンパイラについて総合的に解説していますので、システム・アナリストやプロジェクト・マネージャ、埋込み SQL アプリケーションに関心のあるその他のユーザーにも役立つ内容となっています。このマニュアルを有効に使用するには、C または C++ のアプリケーション・プログラミングの実用知識が必要です。この本では、埋込み SQL プログラミングの複雑な部分をほとんど説明していますが、SQL データベース言語に精通していると理解しやすくなります。

Pro*C/C++ マニュアルの構成

第 1 章～第 10 章では、Pro*C/C++ プログラミングの基本について説明します。Pro*C/C++ 開発者の多くは、これらの章を読むと、便利かつ強力な Pro*C/C++ アプリケーションを記述できます。

第 11 章～付録 F では、Pro*C/C++ についてより詳しく説明します。動的 SQL、マルチスレッド、ユーザー定義オブジェクト、コレクションおよびラージ・オブジェクトを使用する Pro*C/C++ プログラムなど、この製品で使用可能な、より複雑な機能について説明します。

第 1 章「概要」

この章では、Pro*C/C++ について説明します。Oracle データを操作するアプリケーション・プログラムを開発する上での Pro*C/C++ の役割について説明します。この章には、よく聞かれる重要な質問（FAQ）も含まれています。

第2章「プリコンパイラ概念」

この章では、埋込み SQL プログラムの動作について説明します。その後でプログラミングのガイドラインについて説明します。また、Pro*C/C++ アプリケーションのサンプル問合せとともに、使用する表のサンプルが紹介されます。

第3章「データベース概念」

この章では、トランザクションの処理について説明します。データベースの整合性を維持するための基本的な技術およびデータベース・サーバーへの接続方法について説明します。

第4章「データ型とホスト変数」

Oracle データ型、ホスト変数、標識変数、データ変換および Unicode 文字列について説明します。

第5章「上級トピック」

この章では、データ型同値化の利用方法、C プリプロセッサのサポート、SQLLIB 関数の新しい名前および OCI へのインタフェースなど高度なトピックについて説明します。

第6章「埋込み SQL」

埋込み SQL プログラミングの基本事項について説明します。ホスト変数、標識変数、カーソルおよびカーソル変数の使用方法と、Oracle データの挿入、更新、選択および削除を行う基本的な SQL コマンドの使用方法を説明します。

第7章「埋込み PL/SQL」

この章では、PL/SQL トランザクション処理ブロックをプログラム内に埋め込むことによりパフォーマンスを改善する方法について説明します。ホスト変数、標識変数、カーソル、ストアド・プロシージャ、ホスト配列および動的 SQL を使用した PL/SQL の使用方法について学びます。

第8章「ホスト配列」

この章では、配列を使用してプログラムのパフォーマンスを改善する方法について説明します。配列を使用して Oracle データを操作する方法、単一の SQL 文を使用して配列内のすべての要素を処理する方法および処理対象となる配列の要素数を制限する方法を説明します。

第9章「ランタイム・エラーの処理」

この章では、エラーの報告およびリカバリについて説明します。状態変数 SQLSTATE および SQLCODE を WHENEVER 文とともに使用してエラーおよび状態変化を検出する方法を示します。また、SQLCA および ORACA を使用して、エラー条件を検出し問題を発見する方法も示します。

第10章「プリコンパイラのオプション」

この章では、Oracle Pro*C/C++ プリコンパイラを実行するための必要条件を詳しく説明します。プリコンパイルの過程、プリコンパイラ・コマンドの発行方法および各種プリコンパイラ・オプションの指定方法を学習します。

第11章「マルチスレッド・アプリケーション」

この章では、マルチスレッド・アプリケーションの作成について説明します。マルチスレッド・アプリケーションを作成するには、マルチスレッドをサポートするコンパイラを使用する必要があります。

第12章「C++ アプリケーション」

この章では、C++ アプリケーションをプリコンパイルする方法を説明し、サンプル C++ プログラムを3つ紹介します。

第13章「Oracle の動的 SQL」

この章では、動的 SQL の利点を活用する方法について説明します。ユーザーが、たとえば実行時に SQL 文を対話形式で構築できるように、柔軟なプログラムを作成する3つの方法を説明します。

第14章「ANSI 動的 SQL」

新しい ANSI 動的 SQL では、新しい方法4 アプリケーションすべてに使用します。この方法は、いくつかの変数を含む SQL 文を受取り、作成できます。ANSI 動的 SQL は、オブジェクト型、コレクション、カーソル変数、構造体の配列および LOB などの複雑な型を含むアプリケーションに使用する必要があります。

第15章「Oracle の動的 SQL 方法4」

この章では、動的 SQL 方法4（記述子を使用した動的 SQL）を詳細に説明します。Oracle リリース 8.1 以前のバージョンで開発された既存のアプリケーションの変更方法について説明します。

第16章「ラージ・オブジェクト (LOB)」

この章では、ラージ・オブジェクト・データ型 (BLOB、CLOB、NCLOB および BFILE) について説明します。OCI および PL/SQL と同等な機能を提供する埋込み SQL コマンドについて説明し、そのコマンドを使用したサンプル・コードを紹介します。

第17章「オブジェクト」

この章では、次のオブジェクト・サポート機能について説明します。アソシエイティブとナビゲーション・インタフェース（埋込み SQL コマンド）、オブジェクトのプリコンパイラ・オプションおよび Oracle 動的 SQL におけるデータ型の使用制限について説明します。

第18章「コレクション」

この章では、コレクション型 (VARRAY および NESTED TABLE) について説明します。コレクション型を使用する埋込み SQL 文について、その使用例を説明します。

第19章「オブジェクト型トランスレータ」

この章では、Pro*C/C++ アプリケーションで使用される C 構造体にオブジェクト型をマップするオブジェクト型トランスレータ (OTT) について説明します。また OTT オプション、OTT の使用方法およびその結果についても説明します。

第 20 章「ユーザー・イグジット」

この章では、Oracle Tools アプリケーション用のユーザー・イグジットの作成方法を説明します。フォーム・アプリケーションと Pro*C/C++ ユーザー・イグジットをインタフェースするために使用されるコマンドおよびフォーム・ユーザー・イグジットを作成およびリンクする方法について学びます。

付録 A「新機能」

この付録では、Pro*C/C++ のリリース 8.0 および 8.1 に導入された機能の改善点と新機能について説明し、このマニュアルでの掲載場所を示します。

付録 B「予約語、キーワードおよび名前領域」

この付録では、Oracle で特別の意味を持つ予約語とキーワードおよび Oracle ライブラリ用に確保されている名前領域について説明します。

付録 C「パフォーマンスの最適化」

この付録では、アプリケーションのパフォーマンスを改善させる手軽な方法をいくつか紹介します。

付録 D「構文検査と意味検査」

この付録では、埋込み SQL 文および PL/SQL ブロックに対して行う構文および意味の検査の種類と範囲を、SQLCHECK オプションを使用して制御する方法を説明します。

付録 E「システム固有の参照」

この付録では、Pro*C/C++ のシステム固有の側面について説明します。

付録 F「埋込み SQL 文および宣言文」

この付録では、プリコンパイラ宣言文の説明（構文図、キーワードとパラメータの定義など）、埋込み SQL 文および Oracle 埋込み SQL の拡張要素について説明します。

表記規則

本文中の英大文字はデータベース・オブジェクトおよび SQL キーワードを示し、イタリックの英小文字は、C 変数および SQL パラメータの名前を示します。C キーワードは**太字**になっています。

BNF 表記法

このマニュアルでは、BNF 構文に対して次のような表記上の規則を使用しています。

- [] 大カッコは構文記述のオプション項目を示します。
- <> 山カッコは構文記述の構文要素の名前を示します。山カッコのかわりにイタリック体
が使用される場合もあります。
- { } 中カッコは、その中の 1 つの項目のみが必須であることを示します。
- | 垂直バーは、大カッコまたは中カッコ内の項目を区切るために使用します。
- .. 2 つのドットは、ある範囲内の最低値と最高値を区切ります。
- ... 省略記号は、その前のパラメータが繰返し可能であることを示します。または関連の
ない文または句がコード例から省略されていることを示します。

ANSI/ISO 準拠

Pro*C/C++ プリコンパイラは ANSI および ISO SQL 規格に完全に準拠しています。これらの規格に準拠することは、National Institute of Standards and Technology (NIST) により公認されています。ANSI/ISO SQL への拡張要素をフラグするために、FIPS フラガーが提供されています。

要件

ANSI 規格 X3.135-1992 (俗称 SQL92) には、次の 3 つのレベルの準拠があります。

- Full SQL
- Intermediate SQL (Full SQL のサブセット)
- Entry SQL (Intermediate SQL のサブセット)

ANSI 規格 X3.168-1992 は、C などの標準プログラミング言語で作成されたアプリケーション・プログラムに SQL 文を埋め込むための構文および方法を指定しています。

SQL 準拠の処理系では、少なくとも Entry SQL がサポートされている必要があります。Oracle Pro*C/C++ プリコンパイラは Entry SQL92 に準拠しています。

連邦政府における使用目的で取得された RDBMS ソフトウェアに適用される NIST 規格 FIPS PUB 127-1 では、ANSI 規格にも順守することになっています。また、この規格では、データベース構文のための最小限のサイズ・パラメータを指定しています。さらに、"FIPS フラガー" により ANSI 拡張要素を識別することを定めています。

ANSI 規格書を入手するには、次の宛先に書面で申請してください。

米国規格協会 (ANSI)
1430 Broadway
New York, NY 10018
USA

工業規格への Oracle Pro*C/C++ プリコンパイラの準拠性

SQL はリレーショナル・データベース管理システムの標準言語になりました。この項では、次の団体によって確立された SQL 規格への Pro*C/C++ プリコンパイラの準拠性について説明します。

- 米国規格協会 (ANSI)
- 国際標準化機構 (ISO)
- 米国連邦情報・技術局 (NIST)

これらの組織は SQL を次の文書で定義されたものとして承認しました。

- ANSI 規格 X3.135-1992、『Database Language SQL (データベース言語 SQL)』
- ISO/IEC 規格 9075:1992、『Database Language SQL (データベース言語 SQL)』
- ANSI 規格 X3.135-1989、『Database Language SQL with Integrity Enhancement (整合性拡張機能付きデータベース言語 SQL)』
- ANSI 規格 X3.168-1989、『Database Language Embedded SQL (データベース言語埋込み SQL)』
- ISO 規格 9075-1989、『Database Language SQL with Integrity Enhancement (整合性拡張機能付きデータベース言語 SQL)』
- NIST 規格 FIPS PUB 127-1、『Database Language SQL (データベース言語 SQL)』(FIPS は Federal Information Processing Standards (連邦情報処理標準) の頭文字です。)

ISO 規格書を入手するには、ISO 加盟団体の国内事務所に書面で問い合わせてください。ISO 規格書を入手するには、ISO 加盟団体の国内事務所に書面で問い合わせてください。NIST 規格書を入手するには、次の宛先に書面で問い合わせてください。

National Technical Information Service
U.S. Department of Commerce
Springfield, VA 22161
USA

準拠性

Pro*C/C++ プリコンパイラは、現在の ANSI/ISO 規格に 100% 準拠しています。

Pro*C/C++ プリコンパイラは NIST 規格にも 100% 準拠しています。FIPS フラガーと FIPS という名前のオプションを用意してありますが、これによって FIPS フラガーが有効になります。詳細は xxxiv ページの「[FIPS フラガー](#)」を参照してください。

準拠の証示

NIST は SQL Test Suite を使用して、ANSI SQL92 準拠性について Pro*C/C++ プリコンパイラをテストしました。SQL Test Suite は約 300 のテスト・プログラムで構成されています。特に、これらのテスト・プログラムを使用して C の埋込み SQL 規格に準拠しているかどうかテストされました。その結果、Oracle Pro*C/C++ プリコンパイラは Entry SQL92 として 100% ANSI 準拠と証明されました。

テストの詳細は、次の宛先に書面で問い合わせてください。

National Computer Systems Laboratory
Attn: Software Standards Testing Program
National Institute of Standards and Technology
Gaithersburg, MD 20899
USA

FIPS フラガー

FIPS PUB 127-1 によると、「この規格で指定されていない付加的な機能を用意するインプリメンテーションは、適合しない SQL 言語または適合する SQL 言語にフラグをセットするオプションを用意すべきです。これは規格に適合しない方法で処理してもかまいません」とあります。この要件を満たすために、Pro*C/C++ プリコンパイラには FIPS フラガーが用意してあります。これにより ANSI 拡張機能にフラグをセットします。拡張要素とは、ANSI の形式または構文規則（権限付与規則は除く）に違反する SQL 要素を指します。標準 SQL の Oracle 拡張機能のリストは『Oracle8i SQL リファレンス』を参照してください。

FIPS フラガーを使用すると、次の SQL 要素を識別できます。

- ANSI 準拠の環境にアプリケーションを移した場合に修正の必要が生じることになる、非準拠 SQL 要素
- 別の処理環境では異なる動作が予想される、準拠 SQL 要素

つまり、FIPS フラガーは移植性のあるアプリケーションを開発するのに役立ちます。

FIPS オプション

FIPS プリコンパイラ・オプションによって、FIPS フラガーを制御します。FIPS フラガーを使用可能にするには、FIPS=YES をインラインまたはコマンドラインで指定します。FIPS オプ

ションの詳細は、10-11 ページの「[プリコンパイラ・オプションの使用](#)」を参照してください。

以前のリリースからのアプリケーションの移行

Oracle7 と Oracle8 ではデータベース操作の意味が一部変更されています。この変更が Pro*C/C++ アプリケーションに与える影響は『Oracle8i 移行ガイド』を参照してください。

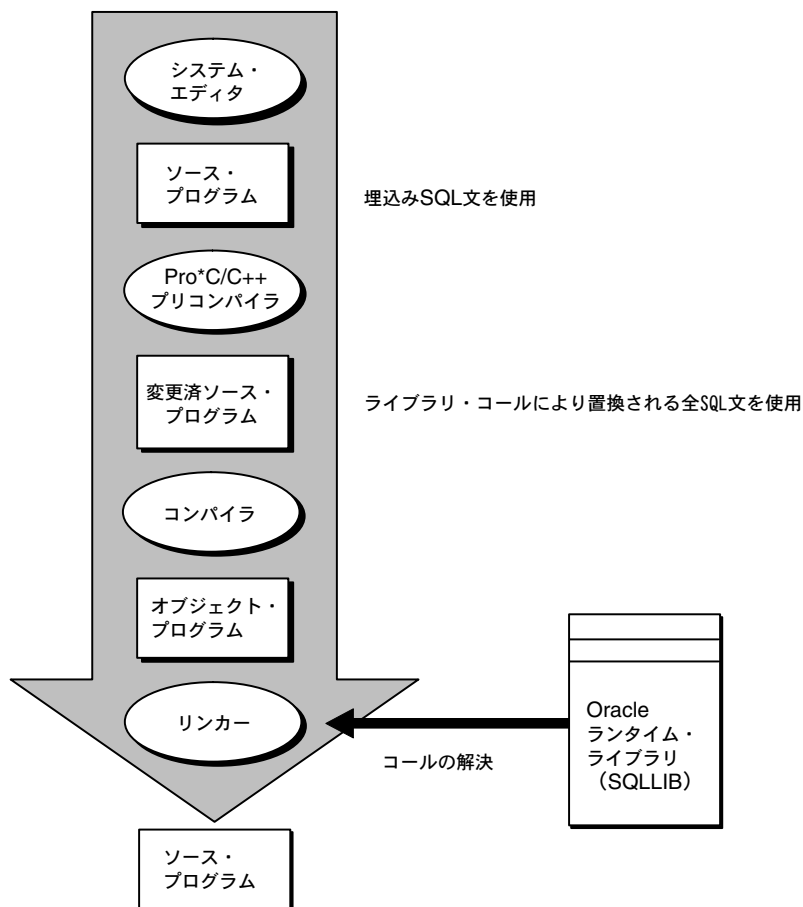
この章では、Pro*C/C++ プリコンパイラについて説明します。Oracle データを操作するアプリケーション・プログラムを開発する上での Pro*C/C++ プリコンパイラの役割と、Pro*C/C++ プリコンパイラによってアプリケーションが実行できる処理について説明します。この章で説明する内容は、次のとおりです。

- Oracle プリコンパイラ
- Oracle Pro*C/C++ プリコンパイラを使用する理由
- SQL を使用する理由
- PL/SQL を使用する理由
- Pro*C/C++ プリコンパイラの利点
- よく聞かれる質問 (FAQ)

Oracle プリコンパイラ

Oracle プリコンパイラとは、高級言語のソース・プログラムへの SQL 文の埋込みを可能にするプログラミング・ツールです。図 1-1 に示したように、プリコンパイラはソース・プログラムを入力として受け入れ、埋込み SQL 文を標準 Oracle ランタイム・ライブラリ・コールに変換して、通常の方法でコンパイル、リンクおよび実行できる変更済ソース・ファイルを生成します。

図 1-1 埋込み SQL プログラム開発



Oracle Pro*C/C++ プリコンパイラを使用する理由

Oracle Pro*C/C++ プリコンパイラを使用すると、アプリケーション・プログラムに強力な柔軟な SQL を含めることができます。便利で使用しやすいインタフェースによって、アプリケーションから直接 Oracle にアクセスできます。

多くのアプリケーション開発ツールとは異なり、Pro*C/C++ では、アプリケーションを高度にカスタマイズできます。たとえば、最新のウィンドウ機能およびマウス技術を埋め込んだユーザー・インタフェースを作成できます。ユーザーとの対話を介さずに、バックグラウンドで実行するアプリケーションも作成できます。

さらに、Pro*C/C++ プリコンパイラはアプリケーションの微調整に役立ちます。Pro*C/C++ プリコンパイラでは、リソースの使用状況、SQL 文の実行状況および各種のランタイム標識を綿密に監視できます。この情報に基づいて、パフォーマンスを最大化するようにプログラム・パラメータを操作できます。

プリコンパイルを行うとアプリケーションの開発プロセスの工程が増えますが、自動的にプリコンパイラが各埋込み SQL 文を Oracle ランタイム・ライブラリ (SQLLIB) の複数のコールに変換するので、時間の節約になります。

SQL を使用する理由

Oracle データにアクセスし、それを操作するには、SQL が必要です。SQL*Plus によって SQL を対話形式で使用するか、アプリケーション内に埋め込むかは、行う作業によって決まります。C または C++ のプロシージャ型処理がジョブに必要な場合やジョブを定期的に行う場合は、埋込み SQL を使用してください。

SQL は、その柔軟かつ強力な特性と習得しやすさによって、最もすぐれたデータベース言語となりました。SQL は非プロシージャ型言語であるため、目的とする処理を指定するとき、その方法を指定する必要がありません。英文に似た少数の文によって、Oracle データを一度に 1 行または複数行ずつ容易に操作できます。

任意の (SQL*Plus 以外の) SQL 文をアプリケーション・プログラムから実行できます。たとえば、次のような SQL 文です。

- データベースの表の動的な CREATE、ALTER、DROP
- データの行の SELECT、INSERT、UPDATE、DELETE
- トランザクションの COMMIT および ROLLBACK

SQL 文をアプリケーション・プログラムに埋め込む前に、SQL*Plus を使用して SQL 文を対話形式でテストできます。通常、対話型 SQL から埋込み SQL に切り替えるためにはわずかな変更で済みます。

PL/SQL を使用する理由

SQL を拡張した PL/SQL は、プロシージャ構造、変数宣言および強力なエラー処理をサポートするトランザクション処理言語です。同一 PL/SQL ブロック内で、SQL および PL/SQL の拡張機能のすべてを使用できます。

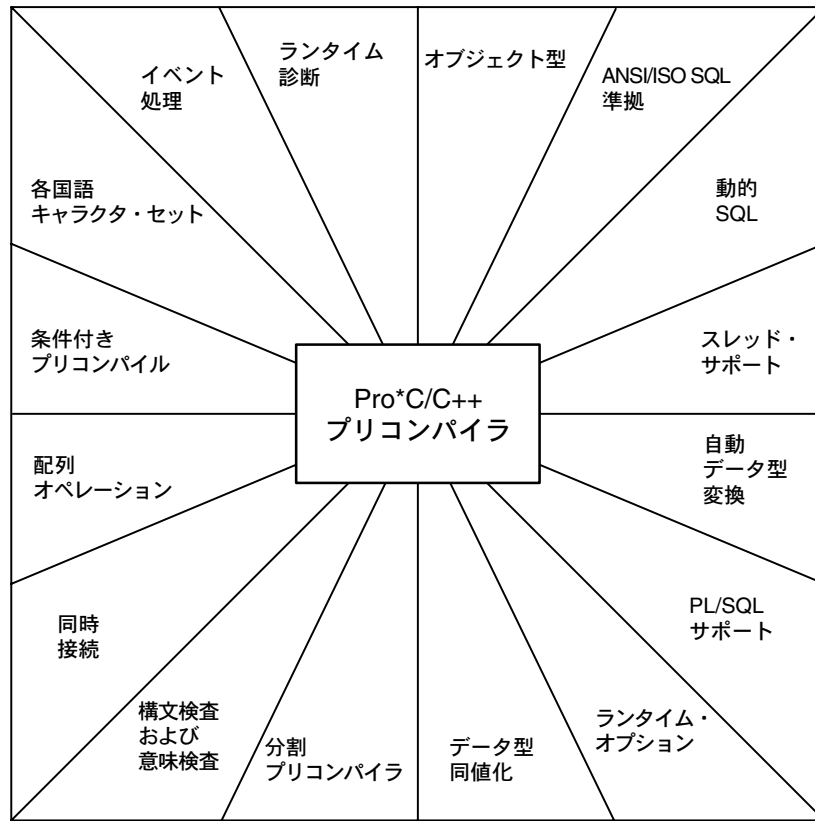
埋込み PL/SQL の主な利点はパフォーマンスの向上です。SQL と異なり、PL/SQL では、SQL 文を論理的にグループ化し、1 文ずつではなくブロック単位で Oracle に送ることができます。この結果、ネットワークの通信量および処理のオーバーヘッドが減少します。

アプリケーション・プログラムへの埋込み方法などの PL/SQL についての情報は、[第 7 章「埋込み PL/SQL」](#)を参照してください。

Pro*C/C++ プリコンパイラの利点

図 1-2 に示すように、Pro*C/C++ は多くの機能と利点を提供します。これは効果的で信頼性の高いアプリケーションの開発に役立ちます。

図 1-2 機能と利点



たとえば、Pro*C/C++ を使用すると次のことが可能です。

- C または C++ 言語でアプリケーションを作成します。
- 高級言語に SQL 文を埋め込むとき ANSI/ISO 規格に従います。
- 高度なプログラム技法である動的 SQL を使用することにより、プログラム実行時に適切な SQL 文を組込みまたは作成します。

- 高度にカスタマイズしたアプリケーションを設計および開発します。
- マルチスレッド・アプリケーションを開発します。
- Oracle の内部データ型と高級言語のデータ型の間で自動的な変換を実行します。
- アプリケーション・プログラム内に PL/SQL トランザクション処理ブロックを埋め込むことによって、パフォーマンス（性能）を向上させます。
- 有用なプリコンパイラ・オプションをインラインまたはコマンドラインで指定し、プリコンパイル中にそれらの値を変更します。
- データ型の同値化を使用して、Oracle での入力データの解釈方法および出力データの形式を制御します。
- 複数のプログラム・モジュールを別々にプリコンパイルし、それらをリンクして 1 つの実行プログラムにします。
- 埋め込まれた SQLDML 操作文および PL/SQL ブロックの構文と意味を完全にチェックします。
- Net8 を使用して、複数のノード上の Oracle データベースに同時にアクセスします。
- 配列を入力プログラム変数および出力プログラム変数として使用します。
- ホスト・プログラム内のコード・セクションを条件付きでプリコンパイルし、異なる複数の環境で実行可能にします。
- 高級言語で作成されたユーザー・イグジットによって、SQL*Forms と直接インタフェースします。
- SQL 通信領域（SQLCA）および WHENEVER 文または DO 文によって、エラーおよび警告を処理します。
- Oracle 通信領域（ORACA）によって提供される強力な診断機能を利用します。
- データベース内でユーザー定義のオブジェクト型を処理します。
- データベースでコレクション（VARRAY および NESTED TABLE）を使用します。
- データベースで LOB（ラージ・オブジェクト）を使用します。
- データベースに格納された各国語キャラクタ・セットを使用します。
- プログラム内で OCI（Oracle Call Interface）関数を使用します。

このように、Pro*C/C++ は、充実した埋込み SQL プログラミング技法をサポートする多機能ツールです。

よく聞かれる質問 (FAQ)

この項では、Pro*C/C++ および Pro*C/C++ と関連する Oracle8i についての一般的な質問をいくつか示します。質問に対する回答は、このマニュアルの他の部分と比較して正式なものではありませんが、参照資料を探すためのリファレンスになります。

VARCHAR について説明してください。

次は VARCHAR の簡単な説明です。

VARCHAR2	データベースの列の一種で、可変長文字データが入っています。列型になり得るため、Oracle ではこれを「内部データ型」とよびます。4-5 ページの「 VARCHAR2 」を参照してください。
VARCHAR	Oracle の「外部データ型」(データ型コード 9) のひとつです。これを使用するのは動的 SQL 方法 4、もしくはデータ型同値化を使用するときのみです。データ型同値化については、4-7 ページの、「 VARCHAR 」、第 14 章「 ANSI 動的 SQL 」および第 15 章「 Oracle の動的 SQL 方法 4 」を参照してください。
VARCHAR[n] varchar[n]	これは Pro*C/C++ プログラムでホスト変数として宣言できる Pro*C/C++ の「疑似型」です。実際には Pro*C/C++ は、これを 2 バイト長の要素と [n] バイト長の文字配列からなる 構造体 として生成します。詳細は、4-17 ページの「 VARCHAR 変数の宣言 」を参照してください。

Pro*C/C++ は、Oracle Call Interface のコールを生成しますか。

いいえ。Pro*C/C++ はデータ構造体を生成して、ランタイム・ライブラリ:SQLLIB (UNIX では *libsql.a*) をコールします。

Pro*C/C++ を使用せず、単に SQLLIB のコールを使用してコーディングできますか。

SQLLIB は外部文書化もサポートもされておらず、リリースごとに異なる可能性があります。一方、Pro*C/C++ は ANSI/ISO に準拠した製品であり、埋込み SQL の標準要件に従っています。

ローレベルのコーディングが必要なときは、Oracle Call Interface を使用してください。Oracle では、Oracle Call Interface のサポートに力を入れています。

OCI と Pro*C/C++ を組み合わせて使用することもできます。5-43 ページの「[OCI リリース 8 の SQLLIB 拡張相互運用性](#)」を参照してください。

PL/SQL のストアド・プロシージャを Pro*C/C++ プログラムからコールできますか。

もちろんできます。第 7 章「埋込み PL/SQL」を参照してください。7-21 ページの「ストアド PL/SQL または Java サブプログラムのコール」にデモ・プログラムがあります。

C++ のコードを作成し、Pro*C/C++ を使用してプリコンパイルできますか。

はい。第 12 章「C++ アプリケーション」を参照してください。

SQL 文の任意の場所でバインド変数を使用できますか。

たとえば、実行時に、SQL 文の表名を入力可能にする場合に、ホスト変数を使用すると、プリコンパイラのエラーが発生します。

一般的に、式を入力できる位置であれば、SQL 文または PL/SQL 文のどの位置にもホスト変数を入力できます。4-14 ページの「ホスト変数の参照」を参照してください。

ただし、次の SQL 文は無効です（この場合、`table_name` はホスト変数です）。

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

問題を解決するには、動的 SQL を使用する必要があります。第 13 章「Oracle の動的 SQL」を参照してください。13-9 ページの「サンプル・プログラム：動的 SQL 方法 1」にデモ・プログラムがあります。

Pro*C/C++ の文字処理がよくわかりません。

多数のオプションがありますが、簡単に説明します。第 1 に、従来のプリコンパイラおよび Oracle7 との互換性が必要な場合には、VARCHAR[n] ホスト変数を使用するのが最も安全な方法です。4-17 ページの「VARCHAR 変数の宣言」を参照してください。

Pro*C/C++ では他のすべての文字変数のデフォルト・データ型は CHARZ です。4-10 ページの「CHARZ」を参照してください。簡単に言うと、入力時には文字列に NULL 終了記号を付ける必要があります。出力時には空白が埋め込まれ、かつ、NULL 終了記号が付けられた文字列が戻されます。

リリース 8.0 では、文字変数のデフォルト・マッピングを指定できるように、CHAR_MAP プリコンパイラ・オプションが追加されています。5-2 ページの「プリコンパイラ・オプション CHAR_MAP」を参照してください。

アプリケーションで VARCHAR も CHARZ も不適当であり、完全に C と同様の動作 (NULL 終了記号は付けるが、空白の埋込みは一切行わない) が必要な場合には、TYPE コマンドと C の `typedef` 文を使用し、データ型の同値化を使用して文字ホスト変数を文字列に変換してください。5-13 ページの「ユーザー定義型同値化」を参照してください。TYPE コマンドの使用方法を示すサンプル・プログラムは、4-41 ページの「サンプル・プログラム：カーソルとホスト構造体」を参照してください。

文字ポインタについて説明してください。何か特別なことはありますか。

はい。Pro*C/C++ は入力ホスト変数または出力ホスト変数をバインドするとき、その長さを知る必要があります。VARCHAR[n] を使用するか、char[n] 型のホスト変数を宣言すると、Pro*C/C++ はその宣言から長さの情報を得ます。しかし、プログラム内で、文字ポインタをホスト変数として使用し、malloc() を使用してバッファを定義すると、Pro*C/C++ は長さの情報を得られません。

出力時には、バッファを割り当てるのみでなく、NULL でない文字を埋め込んでから NULL 終了記号を付ける必要があります。入力時または出力時に、Pro*C/C++ は長さを得るためにバッファに対する strlen() コールを実行します。4-43 ページの「[ポインタ変数](#)」を参照してください。

Pro*C/C++ で SPOOL が動作しない理由を説明してください。

SPOOL は SQL*Plus で使用される特殊なコマンドです。埋込み SQL コマンドではありません。2-2 ページの「[埋込み SQL プログラミングの主要概念](#)」を参照してください。

サンプル・プログラムのオンライン版はどこにありますか。

各 Oracle 導入システムには、それぞれ demo ディレクトリがあります。demo ディレクトリがないか、あってもサンプル・プログラムが入っていない場合には、システム管理者またはデータベース管理者に確認してください。

アプリケーションをコンパイルしてリンクする方法を教えてください。

コンパイルとリンクの方法はプラットフォームごとにより異なります。使用しているシステムの Oracle マニュアルに、Pro*C/C++ アプリケーションのリンク方法に関する説明が載っています。UNIX システムでは、demo ディレクトリに proc.mk という Make ファイルがあります。たとえば、デモ・プログラム sample1.pc をリンクするには、コマンドラインに次のように入力します。

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

特別なプリコンパイラ・オプションを使用する必要がある場合は、Pro*C/C++ を別に実行してから make を実行します。または、独自のカスタム Make ファイルを作成することもできます。たとえば、プログラムに埋込み PL/SQL コードが入っている場合は、次のように入力します。

```
proc cv_demo userid=scott/tiger sqlcheck=semantics
make -f proc.mk cv_demo
```

VMS システムでは、Pro*C/C++ アプリケーションをリンクするための LNPROC というスクリプトがあります。

Pro*C/C++ では構造体をホスト変数として使用できますか。

配列インタフェースではどのようなになるのか教えてください。

1つの構造体の内部で複数の配列が使用できます。また、構造体の配列が配列インタフェースとともに使用できます。4-37 ページの「[ホスト構造体](#)」および 4-43 ページの「[ポインタ変数](#)」を参照してください。

Pro*C/C++ に再帰関数を入れることは、その関数内に埋込み SQL を使用した場合、可能ですか。

はい。Pro*C/C++ では、再帰関数内でカーソル変数を使用することもできます。

Pro*C/C++ のすべてのリリースを、Oracle Server のすべてのバージョンで使用できますか。

いいえ。古いバージョンの Pro*C または Pro*C/C++ は新しいバージョンのサーバーで使用できますが、新しいバージョンの Pro*C/C++ は古いバージョンのサーバーでは使用できません。

たとえば、Pro*C/C++ のリリース 2.2 は、Oracle8i で使用できますが、Pro*C/C++ のリリース 8 は、Oracle7 Server では使用できません。

アプリケーションを Oracle8i で実行すると、いつも ORA-01405 エラー（取り出した列の値が NULL です。）が発生します。

標識変数が結合されていないホスト変数に NULL を代入しています。これは ANSI/ISO 規格に準拠していないため、Oracle7 からは仕様が変更されました。

可能ならば、標識変数を使用してプログラムを書き換え、今後の開発には標識変数を使用します。標識変数の詳細は、4-15 ページの「[標識変数](#)」を参照してください。

代わりに MODE=ORACLE と DBMS=V7 または V8 でプリコンパイルしている場合は、ORA-01405 メッセージを無効にするためにコマンドラインで UNSAFE_NULL=YES を指定（詳細は「[UNSAFE_NULL](#)」を参照）してください。

すべての QLLIB 関数はプライベート関数ですか。

いいえ。自分のプログラムまたはそのデータに関する情報を得るためにコールできる QLLIB 関数がいくつかあります。パブリック QLLIB 関数を次に示します。

<code>SQLSQLDAAlloc()</code>	SQL ディスクリプタ配列 (SQLDA) を動的 SQL 方法 4 で割り当てるのに使用します。15-3 ページの「 SQLDA の参照方法 」を参照してください。
<code>SQLCDAFromResultSetCursor()</code>	1 つの Pro*C/C++ カーソル変数を 1 つの OCI カーソル・データ領域に変換するために使用します。5-49 ページの「 SQLLIB パブリック関数の新しい名前 」を参照してください。
<code>SQLSQLDAFree()</code>	<code>SQLSQLDAAlloc()</code> を使用して割り当てた SQLDA を解放するために使用します。詳細は、5-49 ページの「 SQLLIB パブリック関数の新しい名前 」を参照してください。
<code>SQLCDataToResultSetCursor()</code>	1 つの OCI カーソル・データ領域を 1 つの Pro*C/C++ カーソル変数に変換するために使用します。5-49 ページの「 SQLLIB パブリック関数の新しい名前 」を参照してください。
<code>SQLErrorGetText()</code>	長いエラー・メッセージを戻すために使用します。9-21 ページの「 sqlerrm 」を参照してください。
<code>SQLStmntGetText()</code>	最後に実行された SQL 文のテキストを戻すために使用します。9-32 ページの「 SQL 文のテキスト取得 」を参照してください。
<code>SQLLDAGetNamed()</code>	OCI コールを Pro*C/C++ プログラム内で使用する とき、指定された接続に有効なログイン・データ領域を取得するのに使用します。5-49 ページの「 SQLLIB パブリック関数の新しい名前 」を参照してください。
<code>SQLLDAGetCurrent()</code>	OCI コールを Pro*C/C++ プログラム内で使用する とき、最後の接続に有効なログイン・データ領域を取得するのに使用します。5-49 ページの「 SQLLIB パブリック関数の新しい名前 」を参照してください。
<code>SQLColumnNullCheck()</code>	動的 SQL 方法 4 の NULL 状態の表示を戻します。15-16 ページの「 NULL/NOT NULL データ型の処理 」を参照してください。
<code>SQLNumberPrecV6()</code>	数値の精度およびスケールを戻します。15-15 ページの「 精度とスケールの抽出 」を参照してください。
<code>SQLNumberPrecV7()</code>	<code>SQLNumberPrecV6()</code> の変形。15-15 ページの「 精度とスケールの抽出 」を参照してください。
<code>SQLVarcharGetLength()</code>	<code>VARCHAR[n]</code> の埋め込んだサイズを得るのに使います。4-20 ページの「 VARCHAR 配列コンポーネントの長さを調べる方法 」を参照してください。

<code>SQLEnvGet()</code>	所定 SQLLIB ランタイム・コンテキストの OCI 環境 ハンドルを戻します。5-44 ページの「 SQLEnvGet() 」 を参照してください。
<code>SQLSvcCtxGet()</code>	データベース接続の OCI サービス・コンテキストを 戻します。5-45 ページの「 SQLSvcCtxGet() 」を参照 してください。
<code>SQLRowidGet()</code>	挿入された最後の行の汎用 ROWID を戻します。 4-37 ページの「 SQLRowidGet() 」を参照してくださ い。
<code>SQLExtProcError()</code>	外部 C プロシージャでエラーが発生した場合に PL/SQL に制御を戻すことができます。7-30 ページ の「 SQLExtProcError() 」を参照してください。

このリストの関数はスレッドに対して安全なパブリック SQLLIB 関数です。すべての新しいアプリケーションでこれらの関数を使用します。関数名はリリース 8.0 用に変更されていますが、Pro*C/C++ では以前の名前もサポートされています。これらのスレッドに対して安全なパブリック関数についての情報は（従来の名称も含めて）5-50 ページの「[SQLLIB パブリック関数 – 新しい名前](#)」の表を参照してください。

新しいオブジェクト型は、Oracle8i でどのようにサポートされていますか。

Pro*C/C++ アプリケーションでのオブジェクト型の使用方法は第 17 章「オブジェクト」および第 19 章「オブジェクト型トランスレータ」を参照してください。

プリコンパイラ の 概念

この章では埋込み SQL プログラムの動作について説明します。埋込み SQL プログラムが動作する環境と、その環境がアプリケーションの設計にどう影響するかを検討します。

埋込み SQL プログラミングの主要概念およびアプリケーション開発の手順について説明した後、簡単なプログラムを使用して要点を具体的に説明します。

この章では、次の事項について説明します。

- 埋込み SQL プログラミングの主要概念
- 埋込み SQL アプリケーションの開発ステップ
- プログラミングのガイドライン
- サンプル表
- サンプル・プログラム : 単純な問合せ

埋込み SQL プログラミングの主要概念

この項では、後の各章で説明する内容の基本概念について学習します。次の事項について説明します。

- 埋込み SQL 文
- 埋込み SQL の構文
- 静的 SQL 文と動的 SQL 文の対比
- 埋込み PL/SQL ブロック
- ホスト変数および標識変数
- Oracle のデータ型
- 配列
- データ型の同値化
- プライベート SQL 領域、カーソルおよびアクティブ・セット
- トランザクション
- エラーおよび警告

埋込み SQL 文

埋込み SQL とは、アプリケーション・プログラム内に記述されている SQL 文のことです。SQL 文を含むアプリケーション・プログラムは、ホスト・プログラムと呼ばれ、その記述言語はホスト言語と呼ばれます。たとえば、Pro*C/C++ では、特定の SQL 文を C または C++ ホスト・プログラムに埋め込むことができます。

Oracle データの操作および問合せを行うには、INSERT、UPDATE、DELETE および SELECT 文を使用します。INSERT はデータベースの表にデータの行を追加し、UPDATE は行を変更し、DELETE は不要な行を削除し、SELECT は検索条件を満たす行を取り出します。

強力な SET ROLE 文を使用すると、データベース権限を動的に管理できます。「ロール」とは、ユーザーまたは他のロールに付与された、関連するシステム権限やオブジェクト権限の名前付きグループです。ロール定義は Oracle データ・ディクショナリに格納されます。アプリケーションでは、必要に応じてロールを有効または無効にするために SET ROLE 文を使用できます。

アプリケーション・プログラムでは、SQL 文のみ（SQL*Plus 文は含まれません）が有効です。（SQL*Plus にはレポートの書式化、SQL 文の編集、環境パラメータの設定のため文が追加されています。）

実行文と宣言文

埋込み SQL 文には、すべての対話型 SQL 文に加えて、Oracle とホスト・プログラム間でデータを転送できる他の文があります。埋込み SQL 文には、「実行文」と「宣言文」の 2 種類があります。実行文では、ランタイム・ライブラリ SQLLIB のコールが発生します。実行文は Oracle への接続、Oracle データの定義、問合せ、操作、Oracle データへのアクセス制御、そしてトランザクションの処理に使用します。実行文は、C または C++ 言語の実行文を配置できる位置であれば、どこにでも記述できます。

一方、宣言文では SQLLIB のコールは発生せず、Oracle データの操作も行われません。宣言文は、Oracle オブジェクト、通信領域および SQL 変換を宣言するために使用します。宣言文は、C または C++ の変数宣言を配置できる位置であれば、どこにでも記述できます。ただし、ALLOCATE 文は、宣言文ではなく実行文として扱われます。

表 2-1 では、各種の埋込み SQL 文の一部をグループ分けしています。

表 2-1 埋込み SQL 文

宣言文	用途
ARRAYLEN*	PL/SQL でのホスト配列の使用
BEGIN DECLARE SECTION*	ホスト変数の宣言（オプション）
END DECLARE SECTION*	
DECLARE*	Oracle スキーマ・オブジェクトの命名
INCLUDE*	ファイルへの複写
TYPE*	データ型の同値化
VAR*	変数の同値化
WHENEVER*	ランタイム・エラーの処理
実行文	
ALLOCATE*	Oracle データの定義および制御
ALTER	DML
ANALYZE	
DELETE	
INSERT	
SELECT	
UPDATE	
COMMIT	トランザクションの処理
ROLLBACK	

表 2-1 埋込み SQL 文

実行文	用途
SAVEPOINT	
SET TRANSACTION	
DESCRIBE*	動的 SQL の使用
EXECUTE*	
PREPARE*	
ALTER SESSION	セッションの制御
SET ROLE	
* には、対話形式のものはありません。	

埋込み SQL の構文

作成したアプリケーション・プログラムでは、自由に完全な SQL 文と完全な C 文を混在させ、SQL 文の C 変数または構造体を使用できます。SQL 文をホスト・プログラム内に作成するための特別な必要条件是、キーワード EXEC SQL で SQL 文を開始し、セミコロンで終了するのみです。Pro*C/C++ はすべての EXEC SQL 文をランタイム・ライブラリ QLLIB のコールに変換します。

多くの埋込み SQL 文では、それに対応する対話型の文との違いは、新しい句が 1 つ追加されているか、プログラム変数が使用されていることのみです。次の例で、対話型 ROLLBACK 文と埋込み ROLLBACK 文を比較します。

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

これらの文の効果は同じですが、対話型 SQL 環境（SQL*Plus を実行している場合など）では前者を、Pro*C/C++ プログラムでは後者を使用します。

静的 SQL 文と動的 SQL 文の対比

大部分のアプリケーション・プログラムは、静的 SQL 文および固定的なトランザクションを処理するように設計されています。この場合、実行前にそれぞれの SQL 文およびトランザクションの構成を理解しています。つまり、どの SQL コマンドが発行され、どのデータベースの表が変更され、どの列が更新されるかといったことが理解されています。

しかし、アプリケーションによっては、任意の有効な SQL 文を実行時に受け入れて処理することを要求される場合もあります。したがって、関係する SQL コマンド、データベースの表および列が実行時までわからないことがあります。

動的 SQL は、プログラムの実行時に SQL 文を受け入れさせるか作成させて、データ型の変換を明示的に管理する高度なプログラミング技術です。

埋込み PL/SQL ブロック

Pro*C/C++ は、PL/SQL ブロックを 1 つの埋込み SQL 文と同様に取り扱います。したがって、PL/SQL ブロックは、アプリケーション・プログラム内の SQL 文を記述できる位置であれば、どこにでも記述できます。PL/SQL をホスト・プログラム内に埋め込むには、PL/SQL と共有する変数を宣言し、キーワード EXEC SQL EXECUTE および END-EXEC を使用して PL/SQL ブロックを囲むのみで済みます。

PL/SQL はすべての SQL データ操作コマンドおよびトランザクション処理コマンドをサポートしているため、埋込み PL/SQL ブロックから Oracle データを柔軟かつ安全に操作できます。PL/SQL の詳細は、[第 7 章「埋込み PL/SQL」](#)を参照してください。

ホスト変数および標識変数

ホスト変数は Oracle とプログラムとの間の通信を仲介します。「ホスト変数」とは、C 言語で宣言され、Oracle で共有される（つまり、プログラムと Oracle の両方がその値を参照できる）スカラー変数または集合変数です。

プログラムは「入力」ホスト変数を介して Oracle にデータを引き渡します。Oracle は「出力」ホスト変数を介してプログラムにデータおよびステータス情報を引き渡します。プログラムは入力ホスト変数に値を割り当て、Oracle は出力ホスト変数に値を割り当てます。

ホスト変数は、SQL 式を使用できる位置であれば、どこにでも使用できます。SQL 文内で、ホスト変数と Oracle オブジェクトを区別するには、ホスト変数の前にコロン (:) が必要です。

C 構造体を使用して、これに複数のホスト変数を含めることもできます。コロンを前に付けて埋込み SQL 文の構造体を命名すると、構造体の各コンポーネントがホスト変数として Oracle によって使用されます。

オプションのホスト変数にオプションの標識変数を関連付けることができます。「標識変数」とは、ホスト変数の値または状態を示す短整数型変数のことです。標識変数は、入力ホスト変数への NULL の割当てと、出力ホスト変数に入っている NULL 値または切捨て値の検出に使用されます。「NULL 値」とは欠落している値、未知の値、または適用不能な値のことです。

SQL 文の場合、標識変数は、コロンを前に付けて、対応するホスト変数の直後に記述する必要があります。さらに明確にするため、ホスト変数とその標識変数との間にキーワード INDICATOR を記述できます。

ホスト変数が構造体にパッケージされている場合に、標識変数を使用するときは、ホスト構造体の各ホスト変数に対する標識変数を入れた構造体を作成し、SQL 文で標識の構造体の名前の前にコロンを付け、ホスト変数の構造体の直後に指定するのみで済みます。また、INDICATOR キーワードを使用して、ホスト構造体とそれに対応付けられた標識構造体を分離することもできます。

Oracle のデータ型

通常、ホスト・プログラムはデータを Oracle に入力し、Oracle はデータをプログラムに出力します。Oracle はデータベース表に入力データを格納し、出力データをプログラム・ホスト変数に格納します。データ項目を格納するために、Oracle はそのデータ型を認識する必要があります。データ型によって、記憶形式および値の有効範囲が指定されます。

Oracle では、「内部データ型」と「外部データ型」の 2 種類のデータ型が識別されます。内部データ型は Oracle がデータベース列にデータを格納する方法を指定します。さらに Oracle は、データベース疑似列を表現するために内部データ型を使用します。データベース疑似列は特定のデータ項目を戻しますが、表に実際の列はありません。

外部データ型は、データがホスト変数に格納される方法を指定します。ホスト・プログラムが Oracle にデータを入力するとき、Oracle は必要に応じて、入力ホスト変数の外部データ型と格納先データベース列の内部データ型との間で変換を行います。Oracle はホスト・プログラムにデータを出力するとき、必要に応じて、ソース・データベース列の内部データ型と出力ホスト変数の外部データ型との間で変換を行います。

配列

Pro*C/C++ では、配列ホスト変数（「ホスト配列」と呼ばれる）および構造体の配列を定義でき、次にそれらを単一の SQL 文で操作することができます。配列に対する SELECT、FETCH、DELETE、INSERT、UPDATE 文を使用すると、大量のデータを簡単に問い合わせたり操作したりできます。ホスト変数**構造体**の中でホスト配列を使用することもできます。

データ型の同値化

Pro*C/C++ プリコンパイラではデータ型を「同値化」できるため、アプリケーションの柔軟性が向上します。つまり、Oracle が入力データを解釈し、出力データをフォーマットする方法をカスタマイズできます。

個々の変数ごとに、サポートされている C のデータ型を、Oracle の外部データ型と同値化できます。また、ユーザー定義のデータ型を Oracle の外部データ型と同値化することもできます。

プライベート SQL 領域、カーソルおよびアクティブ・セット

Oracle では、SQL 文を処理するために「プライベート SQL 領域」と呼ばれる作業領域がオープンされます。このプライベート SQL 領域には SQL 文の実行に必要な情報が保存されます。「カーソル」と呼ばれる識別子を使用すると、SQL 文に名前を付け、そのプライベート SQL 領域に保存されている情報にアクセスし、その処理をある程度まで制御できます。

静的 SQL 文には、「暗黙のカーソル」と「明示的なカーソル」の 2 種類のカーソルがあります。Oracle では、1 行のみを戻す SELECT 文（問合せ）を含めて、すべてのデータ定義文および DML 文のためにカーソルが 1 つ暗黙的に宣言されます。ただし、複数行を戻す問合せで 2 行目以降を処理する場合は、明示的にカーソルを宣言するか、ホスト配列を使用する必要があります。

戻された一連の行を「アクティブ・セット」と呼びます。そのサイズは問合せの検索条件を満たす行が何行あるかによって変わります。現在処理している行（「カレント行」と呼びます）を識別するには、明示的なカーソルを使用します。

端末の画面に戻された一連の行を想像してみましょう。画面上のカーソルは最初に処理する行に、その後は次の行、その後はさらに次の行というように次々移動します。同様にして、明示的なカーソルはアクティブ・セット内のカレント行を「指し」ます。これによって、プログラムは一度に 1 行ずつ処理できます。

トランザクション

「トランザクション」とは、論理的に関連のある一連の SQL 文です（たとえば、ある銀行勘定の貸方に記入し、別の銀行勘定の借方に記入する 2 つの UPDATE 文など）。Oracle ではトランザクションは 1 つの単位として扱われるため、トランザクションを構成する文による変更の確定または取消しはすべて同時に行われます。

データ定義、COMMIT 文または ROLLBACK 文が最後に実行された時点以降に実行する DML 文すべてが、カレント・トランザクションを構成します。

データベースの整合性を維持するために、Pro*C/C++ では COMMIT 文、ROLLBACK 文および SAVEPOINT 文を使用してトランザクションを定義できます。

COMMIT はカレント・トランザクションに対する変更をすべて確定します。ROLLBACK はカレント・トランザクションを終了し、そのトランザクションが開始された後に行われた変更をすべて取り消します。SAVEPOINT はトランザクションの処理におけるカレント・ポイントに印を付けます。SAVEPOINT を ROLLBACK とともに使用することによって、トランザクションを部分的に取り消すことができます。

エラーおよび警告

埋込み SQL 文を実行すると、処理が成功もしくは失敗した場合にエラーまたは警告が発生します。その際、これらの結果を処理する方法が必要になります。Pro*C/C++ には、SQL 通信領域（SQLCA）および WHENEVER 文の、2 種類のエラー処理機構が用意されています。

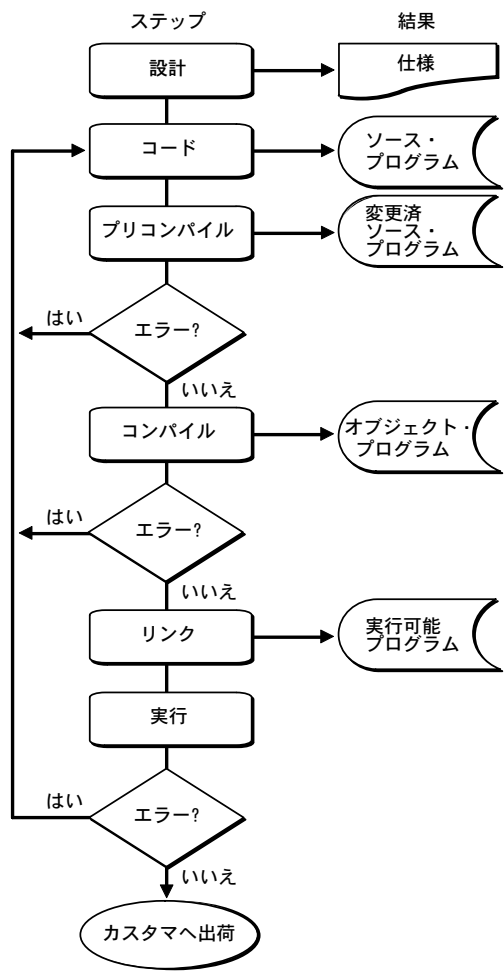
SQLCA は、ホスト・プログラム内に組み入れる（つまりハードコードする）データ構造体です。SQLCA は Oracle によって使用されるプログラム変数を定義して、ランタイム・ステータス情報をプログラムに引き渡します。SQLCA を使用すると、直前に実行を試みた作業に関する Oracle からのフィードバックに基づいて、別の処理を行うことができます。たとえば、PL/SQL を使用しないと、Oracle は一度に SQL 文を 1 つずつ処理する必要があります。

WHENEVER 文を使用すると、Oracle がエラーまたは警告状態を検出した際に自動的に行われるアクションを指定できます。これらのアクションは、次の文の継続実行、関数のコール、ラベル付き文への分岐、および停止です。

埋込み SQL アプリケーションの開発ステップ

図 2-1 では、埋込み SQL アプリケーション開発の過程を示しています。

図 2-1 埋込み SQL アプリケーションの開発過程



この図に示されるように、プリコンパイルの結果、通常のコンパイルが可能な変更済ソース・プログラムが得られます。従来の開発プロセスにプリコンパイルの処理が追加されますが、このステップによってきわめて柔軟なアプリケーションを作成できます。

プログラミングのガイドライン

この項では、埋込み SQL 構文、コーディング規則、および C 固有の機能および制限を扱います。トピックは見やすいようにアルファベット順に配列しています。

コメント

SQL 文内には、空白を入力できる位置（ただしキーワード EXEC SQL の間以外）であればどこにでも、C 形式のコメント（/*...*/）を記述できます。SQL 文内には、次の例のように、ANSI 形式のコメント（--...）を入力できます。

```
EXEC SQL SELECT ENAME, SAL
        INTO :emp_name, :salary -- output host variables
        FROM EMP
        WHERE DEPTNO = :dept_number;
```

CODE=CPP プリコンパイラ・オプションを使用してプリコンパイルする場合は、C++ 形式のコメント（//）を Pro*C/C++ ソース内で使用できます。

定数

L または *l* を接尾辞として付けると、**long** 型定数の指定になります。*U* もしくは *u* を接尾辞として付けると符号なし **unsigned** 型定数の指定になります。*0X* もしくは *0x* を接頭辞として付けると、16 進整数定数の指定になります。*F* もしくは *f* を接頭辞に付けると、**浮動小数点定数**の指定になります。これらの形式は SQL 文では使用できません。

宣言文

ホスト変数宣言およびそのフォームを含む「宣言文」

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

次の文で始まり、

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

次の文で終わる宣言文

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

これらの2つの文の間で使用できるものは、次のとおりです。

- ホスト変数および標識変数宣言
- 非ホスト C/C++ 変数
- EXEC SQL DECLARE 文
- EXEC SQL INCLUDE 文
- EXEC SQL VAR 文
- EXEC SQL TYPE 文
- EXEC ORACLE 文
- C/C++ コメント

MODE=ANSI、CODE=CPP（C++ アプリケーション内）、PARSE=NONE または PARTIAL の場合には、宣言文が必要です。PARSE オプションの詳細は、12-4 ページの「[コードの解析](#)」を参照してください。

複数の宣言文を、異なるコード・モジュールで使用できます。

デリミタ

C では、引用符を次のように単一文字を区切るときに使用します。

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

SQL では、引用符を次のように文字列を区切るときに使用します。

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

C では、二重引用符を次のように文字列を区切るときに使用します。

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

SQL では、二重引用符を特殊文字もしくは小文字を含んだ識別子を区切るときに使用します。

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

ファイルの長さ

Pro*C/C++ が処理できるソース・ファイルの長さには制限があります。許可される行数には制限があります。ソース・ファイルの次のような事項が、ファイル・サイズを制約する要因となります。

- 埋込み SQL 文の複雑さ（たとえば、バインド変数と定義変数の数）
- データベース名の使用の有無（たとえば、AT 句を使用してデータベース名に接続する場合）
- 埋込み SQL 文の数

この制約に関連した問題を防ぐには、複数のプログラム単位を使用してソース・ファイルのサイズを小さくしてください。

関数プロトタイプ

ANSI C 標準 (X3.159-1989) は、関数プロトタイプを提供しています。「関数プロトタイプ」では、関数およびその引数のデータ型を宣言するため、C コンパイラは欠落している引数または一致しない引数を検出できます。

CODE オプションによって、プリコンパイラで C または C++ コードがどのように生成されるかが決まります。このオプションは、コマンドラインまたは構成ファイルに入力できます。

ANSI_C

CODE=ANSI_C を指定してプログラムをプリコンパイルすると、プリコンパイラは完全にプロトタイプ化された関数宣言を生成します。たとえば、次のとおりです。

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

KR_C

CODE=KR_C (KR は「Kernighan」と「Ritchie」の頭文字です) を指定してプログラムをプリコンパイルすると、関数パラメータのリストがコメント・アウトされていることを除き、ANSI_C を指定してプログラムをプリコンパイルする場合と同じように、プリコンパイラは関数プロトタイプを生成します。たとえば、次のとおりです。

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

したがって、ANSI C がサポートされていない C コンパイラを使用する場合は、必ずプリコンパイラのオプション CODE を KR_C に設定してください。CODE オプションが ANSI_C

に設定されると、プリコンパイラは他の ANSI 特有の構文（たとえば **const** 型の修飾子など）も生成できます。

CPP

CODE=CPP でコンパイルすると、C++ と互換性のある関数プロトタイプが生成されます。このオプションは、C++ コンパイラで使用してください。C++ の使用方法の詳細は、[第 12 章「C++ アプリケーション」](#)を参照してください。

ホスト変数名

ホスト変数名は、英文字および数字、アンダースコアから構成されますが、最初の文字は英文字にする必要があります。長さは任意ですが、Pro*C/C++ にとって意味があるのは先頭の 31 文字までです。C コンパイラもしくはリンカーによっては最大長がもっと短いことがあります。使用する C コンパイラのユーザーズ・ガイドを調べてください。

SQL92 規格合致性のために、ホスト変数名の長さは 18 文字以下に制限されます。

アプリケーション内での使用方法に制限がある用語の一覧については、[付録 B「予約語、キーワードおよび名前領域」](#)を参照してください。

行の継続

SQL 文は、ある行から次の行に続けることができます。文字列リテラルをある行から次の行に続けるには、次の例に示されているように、バックスラッシュ（\）の使用が必要です。

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

このコンテキストでは、プリコンパイラはバックスラッシュを継続文字として扱います。

行の長さ

ASCII 文字のみで構成される行の最大長は、1299 です。各国語サポート文字の場合は、324 です。

MAXLITERAL デフォルト値

プリコンパイラ・オプション MAXLITERAL によって、プリコンパイラが生成する文字列リテラルの最大長を指定できます。MAXLITERAL のデフォルト値は 1024 です。必要に応じてより小さな値を指定してください。たとえば使用している C コンパイラが 512 文字より長い文字列リテラルを処理できない場合は、MAXLITERAL=512 を指定します。使用している C コンパイラのユーザーズ・ガイドを参照してください。

演算子

論理演算子と関係演算子「equal to」は C と SQL では下に示すように異なります。これらの C 演算子は SQL 文では、使用できません。

SQL 演算子	C 演算子
NOT	!
AND	&&
OR	
=	==

同様に使用できない演算子は次のとおりです。

タイプ	C 演算子
アドレス	&
ビット単位	&, , ^, ~
コンパウンド代入	+=, -=, *= など
条件付き	?:
デクリメント	--
インクリメント	++
間接参照	*
モジュラス	%
シフト	>>, <<

文終了記号

次の例に示すように、埋込み SQL 文の終わりには必ずセミコロン (;) を付けます。

```
EXEC SQL DELETE FROM emp WHERE deptno = :dept_number;
```

条件付きプリコンパイル

条件付きプリコンパイルでは、条件に従ってコードのある部分をホスト・プログラムに組み入れたり、除外したりします。たとえば、UNIX でプリコンパイルする場合にコードの特定のセクションを含め、VMS でプリコンパイルする場合に別のセクションを含めることがあ

ります。条件付きプリコンパイルを使用すれば、異なる複数の環境下で実行可能なプログラムを記述できます。

環境および処理を定義する文によってコードの条件文が区切られます。これらの条件文には、C または C++ の文と EXEC SQL 文をあわせて記述できます。次の文でプリコンパイルの条件を制御します。

```
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
```

EXEC ORACLE 文の終わりには、必ずセミコロンを付けてください。

記号の定義

記号を定義するには 2 通りの方法があります。次の文を含みます。

```
EXEC ORACLE DEFINE symbol;
```

または、次の構文を使用してコマンドラインで記号を定義します。

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

このとき symbol の部分は大文字小文字の区別がありません。

警告 : #define プリプロセッサ宣言文は EXEC ORACLE DEFINE 文と同一のものではありません。

Pro*C/C++ をシステムにインストールするときに、ポート固有の記号がいくつか事前定義されます。たとえば、オペレーティング・システムの事前定義済記号には、CMS、MVS、MS-DOS、UNIX および VMS があります。

例

次の例では、記号 site2 が定義されているときのみ SELECT 文がプリコンパイルされます。

```
EXEC ORACLE IFDEF site2;
EXEC SQL SELECT DNAME
        INTO :dept_name
        FROM DEPT
        WHERE DEPTNO= :dept_number;
EXEC ORACLE ENDIF;
```

C、C++ または埋込み SQL コードは IFDEF と ENDIF の間に置いて、そのシンボルを定義しないことで「コメント・アウト」できます。

分割プリコンパイル

複数の C または C++ プログラム・モジュールを別々にプリコンパイルした後、それらをリンクして 1 つの実行可能プログラムにすることができます。これは、別のプログラマによってプログラムの機能コンポーネントが作成およびデバッグされた場合に必要で、構造化プログラミングもサポートしています。個々のプログラム・モジュールは、同一の言語で記述する必要はありません。

ガイドライン

次のガイドラインに従うと、いくつかの問題を回避できます。

カーソルの参照

カーソル名は SQL 識別子であり、その適用範囲はプリコンパイル・ユニットです。このためカーソル操作が複数のプリコンパイル・ユニット（ファイル）に及ぶことはありません。つまり、あるファイル内で宣言したカーソルを別のファイルからオープンしたり、フェッチしたりすることはできません。したがって、プリコンパイルを個別に実行するときは、指定のカーソルに対する定義および参照がすべて 1 つのファイル内に記述されていることを確認してください。

MAXOPENCURSORS の指定

Oracle に CONNECT するプログラム・モジュールをプリコンパイルするときは、すべてのプログラム・モジュールに十分対応できるよう MAXOPENCURSORS の値を指定してください。CONNECT しない別のプログラム・モジュールに MAXOPENCURSORS を使用しても、MAXOPENCURSORS の値は無視されます。実行時は CONNECT に対する有効値のみが使用されます。

単一の SQLCA の使用

SQLCA を 1 つのみ使用するときは、1 つのプログラム・モジュール内で global として宣言し、その他のモジュールでは extern として宣言する必要があります。extern 記憶域クラスを使用し、コードに次の文を追加します。

```
#define SQLCA_STORAGE_CLASS extern
```

これはプリコンパイラに他のプログラム・モジュールの SQLCA を探すよう指示します。SQLCA を外部参照用に宣言しないかぎり、それぞれのプログラム・モジュールは局所的な SQLCA を使用します。

注意：アプリケーションのソース・ファイルはすべて、名前が一意である必要があります。一意でない場合はエラーが発生します。

コンパイルとリンク

実行可能プログラムを作成するには、プリコンパイラによって作成された出力 .c ソース・ファイルをコンパイルし、その結果生成されるオブジェクト・モジュールを SQLLIB およびシステム固有の Oracle ライブラリの必要なモジュールにリンクする必要があります。プリコンパイラ・コードと OCI コールを併用しているときは、OCI ランタイム・ライブラリ (UNIX システムでは *liboci.a*) にもリンクしてください。

リンカーはオブジェクト・モジュール内のシンボリック参照を解決します。これらの参照で競合が発生すると、リンクは失敗します。サードパーティ・ソフトウェアをプリコンパイル済みプログラムにリンクすると、このような失敗が起こります。これはすべてのサードパーティ・ソフトウェアに Oracle との互換性があるわけではないためです。したがってプログラムをリンクして「共有」にすると、原因不明のエラーが発生することがあります。「スタンドアロン型」または「2 タスク型」でリンクすると問題が解消する場合もあります。

コンパイルとリンクはシステムに依存します。ほとんどのプラットフォームでは、サンプルの *makefiles* またはバッチ・ファイルが提供されています。これらを使用して Pro*C/C++ アプリケーションのプリコンパイル、コンパイルおよびリンクができます。各システムのマニュアルを参照してください。

サンプル表

このガイドのプログラミング例は、その大部分が 2 つのサンプル・データベース表、DEPT および EMP を使用しています。次にそれらの定義を示します。

```
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))
```

サンプル・データ

DEPT 表と EMP 表には、それぞれ次のデータ行が含まれています。

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

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

サンプル・プログラム：単純な問合せ

Pro*C/C++ および埋込み SQL をよく理解する方法の 1 つは、プログラム例を学習することです。次に示すプログラムは、Pro*C/C++ の *demo* ディレクトリにあるファイル *sample1.pc* なのでオンラインでも使用可能です。

プログラムは Oracle に接続し、ループし、従業員番号の入力をユーザーに求めます。サンプル・プログラムは、データベースに従業員名、給与および職務を問い合わせ、その情報を表示した後、ループを続行します。情報はホスト構造体に戻されます。さらに、SELECT で選択された出力値のいずれかが「NULL」であるかどうかを示す、パラレル標識構造体もあります。

サンプル・プログラムは、プリコンパイラ・オプション `MODE=ORACLE` を使用してプリコンパイルします。

```
/*
 * 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 20
#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;
    float commission;
} 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;
int 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\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);
    printf("%.s\n", msg_len, err_msg);
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}

```

データベースの概念

この章では、CONNECT 文およびそのオプション（ALTER AUTHORIZATION、自動接続および分散処理など）について説明します。また Net8 やネットワーク接続で使用する文についても説明します。

次に、トランザクションの処理方法について説明します。Oracle データへの変更の確定または取消しを制御する方法など、データベースの整合性を維持するための基本的な技術を学習します。

この章は次のトピックで構成されています。

- データベースへの接続
- アドバンスド・コネクション・オプション
- トランザクション用語の定義
- トランザクションがデータベースを保護する方法
- トランザクションの開始・終了方法
- COMMIT 文の使用方法
- SAVEPOINT 文の使用方法
- ROLLBACK 文の使用方法
- RELEASE オプションの使用方法
- SET TRANSACTION 文の使用方法
- デフォルト・ロックのオーバーライド
- COMMIT をまたぐ FETCH
- 分散トランザクションの処理
- ガイドライン

データベースへの接続

次のいくつかの項にわたって、CONNECT 文の完全な構文について説明します。構文は次のとおりです。

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

データの問合せまたは操作をする前に、Pro*C/C++ プログラムをデータベースに接続する必要があります。ログインするには、単純に CONNECT 文を使用してください。

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

username および *password* は、**CHAR** または **VARCHAR** ホスト変数です。

または、この文は次のようにも指定できます。

```
EXEC SQL CONNECT :usr_pwd;
```

このホスト変数 *usr_pwd* には、スラッシュ文字 (/) で区切られたユーザー名およびパスワードが含まれます。

CONNECT 文の簡略化されたサブセットもあります。詳細は、この章の次の項または F-26 ページの「[CONNECT \(実行可能埋込み SQL 拡張要素\)](#)」を参照してください。

CONNECT 文は、プログラムが実行する最初の SQL 文である必要があります。つまり、プリコンパイル・ユニット内で他の SQL 文を CONNECT 文の前に物理的に置くことはできませんが、論理的に置くことはできません。

Oracle8 のユーザー名とパスワードを別々に入力する場合は、2 つのホスト変数を文字列または VARCHAR として定義します。(ユーザー名とパスワードの両方を含むユーザー名を入力する場合、必要なホスト変数は 1 つのみです。)

CONNECT を実行する前に、ユーザー名とパスワードの変数を設定する必要があります。設定していない場合、CONNECT は失敗します。設定しておけば、プログラムの指示に従って値を入力したり、次の例のように値をハードコードできます。

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

ただし、ユーザー名およびパスワードはいずれも CONNECT 文にハードコードすることはできません。また、引用されるリテラルも使用できません。たとえば、次の文は両方とも無効です。

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


ALTER AUTHORIZATION 句を使用したパスワードの変更

Pro*C/C++ のクライアント・アプリケーションでは、EXEC SQL CONNECT 文を拡張し、実行時にユーザーのパスワードを変更できます。

この項では、ALTER AUTHORIZATION 句を様々に変化させて使用した場合に予想される結果について説明します。

標準 CONNECT

次の文がアプリケーションから発行されると、

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

通常の接続が実行されます。この場合、次の結果が予想されます。

- アプリケーションが問題なく接続されます。
- アプリケーションは接続されるが、パスワードについての警告が出されます。この警告は、パスワードが期限切れになっているが、まだログインできる期間にあることを示します。この期間内にパスワードを変更してください。そうしないと、アカウントがロックされます。
- アプリケーションが接続されません。次の原因が考えられます。
 - パスワードが間違っています。
 - アカウントが期限切れになっているか、ロック状態になっています。

CONNECT 文でのパスワードの変更

次の CONNECT 文を使用します。

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

この文は、アプリケーションがアカウントのパスワードを newpswd で指定された値に変更することを示します。パスワードを変更すると、user/newpswd として接続試行が実行されます。この場合、次の結果が予想されます。

- アプリケーションが問題なく接続されます。
- アプリケーションが接続されません。次のどちらかの原因が考えられます。
 - なんらかの理由でパスワードを認識できませんでした。パスワードは元のままです。
 - アカウントがロックされています。パスワードは変更できません。

Net8 を利用した接続

Net8 ドライバを使用して接続するには、*tnsnames.ora* 構成ファイルまたは Oracle Names で定義されているサービス名を使用します。

Oracle Names を使用する場合、名前サーバーはネットワーク定義データベースからサービス名を取得します。

注意：SQL*Net V1 は Oracle8 では動作しません。

Net8 の詳細は『Oracle8i Net8 管理者ガイド』を参照してください。

自動接続

ユーザー名を使用して Oracle8 に自動的に接続できます。

OPS\$username

この場合、*username* はカレント・オペレーティング・システムのユーザー名で、OPS\$username は有効な Oracle8 データベース・ユーザー名です。(OPS\$ の実際の値は、INIT.ORA パラメータ・ファイルに定義されています。) 次のようにして Pro*C/C++ プリコンパイラにスラッシュ文字を渡します。

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

これによって自動的に OPS\$username というユーザーとして接続します。たとえば、オペレーティング・システムでのユーザー名が RHILL で、OPS\$RHILL が有効な Oracle8 のユーザー名の場合、'/' を使用した接続で Oracle8 へユーザー OPS\$RHILL として自動的にログインできます。

プリコンパイラにも文字列の中で '/' を渡すことができます。ただし、その文字列に後続ブランクを入れないでください。たとえば、次の CONNECT 文は失敗します。

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

AUTO_CONNECT プリコンパイラ・オプション

AUTO_CONNECT=YES で、最初の実行 SQL 文を処理するときにアプリケーションがまだデータベースに接続されていないければ、アプリケーションは次のユーザー ID を使用して接続を試みます。

OPS\$<username>

この場合、*username* はカレント・オペレーティング・システムのユーザー名またはタスク名で、*OPPS\$username* は有効な Oracle8 ユーザー ID です。AUTO_CONNECT のデフォルト値は NO です。

AUTO_CONNECT=NO のとき、Oracle に接続するにはプログラム内で CONNECT 文を使用する必要があります。

SYSDBA または SYSOPER システム権限

Oracle8i がリリースされる前、SYSDBA または SYSOPER システム権限を得るには、ここで説明するような句を使用する必要はありませんでしたが、Oracle8i 以降では使用する必要があります。

SYSDBA または SYSOPER のいずれかのシステム権限でログインするには、次のオプション文字列を他のすべての句に追加します。

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

たとえば、次のとおりです。

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

このオプションに適用される制限事項は次のとおりです。

- プリコンパイラのオプション設定が AUTO_CONNECT=YES になっている場合、このオプションは使用できません。
- CONNECT 文で ALTER AUTHORIZATION キーワードを使用している場合、このオプションは使用できません。（詳細は、3-3 ページの「[ALTER AUTHORIZATION 句を使用したパスワードの変更](#)」を参照してください。）

アドバンスト・コネクション・オプション

予備知識

ネットワーク内の通信ポイントは「ノード」と呼ばれます。Net8 では、ネットワーク上のあるノードから別のノードへ情報（SQL 文、データおよび状態コード）を送信できます。

「プロトコル」とはネットワークにアクセスするための一連の規則です。その規則は障害後のリカバリ手順、データの転送およびエラーの検査のフォーマットなどを規定します。

ローカル・ドメイン内のデフォルトのデータベースに接続するための Net8 の構文では、そのデータベースのサービス名を使用するのみで済みます。

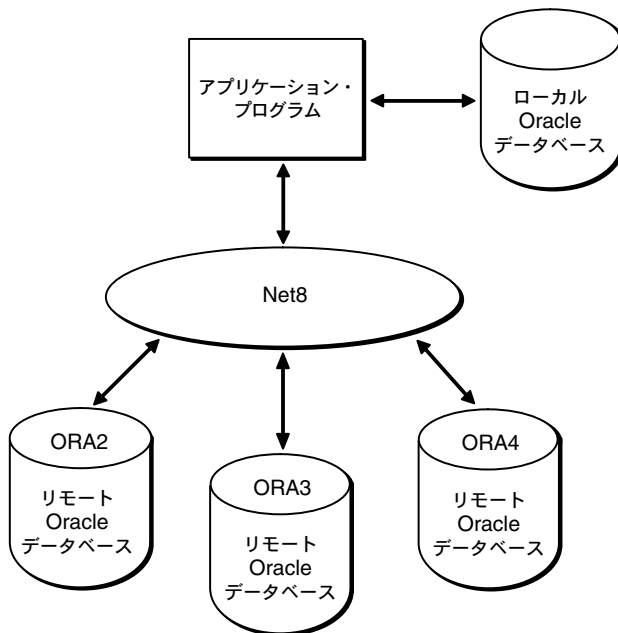
そのサービス名がデフォルト（ローカル）ドメイン内にない場合は、グローバル指定（すべてのドメインの指定）を使用する必要があります。たとえば、次のとおりです。

```
HR.US.ORACLE.COM
```

同時ログイン

Pro*C/C++ は Net8 経由で分散処理をサポートします。アプリケーションは、ローカル・データベースとリモート・データベースの任意の組合せに同時にアクセスしたり、同じデータベースへの複数の接続を確立できます。図 3-1 ではアプリケーション・プログラムは Oracle8 のローカル・データベース 1 つとリモート・データベース 3 つと通信しています。ORA2、ORA3 および ORA4 は CONNECT 文中で使用される論理名です。

図 3-1 Net8 経由の接続



ネットワーク上で異なるマシンとオペレーティング・システム間の境界を排除することによって、Net8 は、Oracle8 ツールに分散処理環境を提供します。ここでは、Pro*C/C++ が Net8 経由で分散処理をサポートする方法を説明します。アプリケーションから可能な操作は次のとおりです。

- 他のデータベースへの直接または間接アクセス
- ローカルおよびリモート・データベースの任意の組合せへの同時アクセス
- 同一のデータベースへの複数接続

Net8 インストールの詳細と使用可能なデータベースの確認は『Oracle8i Net8 管理者ガイド』とシステム固有の Oracle ドキュメントを参照してください。

デフォルトのデータベースおよび接続

それぞれのノードには「デフォルト」のデータベースがあります。CONNECT 文で、データベース名のみを指定してドメイン名を指定しないと、指定したローカル・ノードまたはリモート・ノード上のデフォルトのデータベースに接続されます。

「デフォルト」接続は AT 句のない CONNECT 文によって行われます。ローカルまたはリモート・ノードでデフォルトのデータベースにも非デフォルトのデータベースにも接続できます。AT 句のない SQL 文はデフォルト接続に対して実行されます。逆に、非デフォルト接続は AT 句を持つ CONNECT 文によって行われます。AT 句付きの SQL 文は非デフォルト接続に対して実行されます。

データベース名はすべて一意である必要があります。しかし、2 つ以上のデータベース名で同じ接続を指定できます。つまり任意のノード上のデータベースに複数の接続を持つことができます。

明示的接続

通常は、Oracle8 への接続を次のように確立します。

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

また、次の接続も使用できます。

```
EXEC SQL CONNECT :usr_pwd;
```

usr_pwd には *username/password* が含まれます。

次のユーザー ID で自動的に Oracle8 に接続することができます。

```
OPS$username
```

この場合、*username* はカレント・オペレーティング・システムのユーザー名またはタスク名で、OPS\$username は有効な Oracle8 ユーザー ID です。次に示すとおり、プリコンパイラにスラッシュ (/) 文字を渡します。

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

これによって自動的に OPS\$username というユーザーとして接続します。

データベースおよびノードを指定しないと、カレント・ノードでデフォルトのデータベースに接続されます。異なるデータベースに接続する場合、そのデータベースを明示的に識別する必要があります。

「明示的接続」で、別のデータベースに直接接続し、接続には SQL 文で参照される名前を付けます。同時にいくつかのデータベースに接続することも、同じデータベースに複数回接続することもできます。

単一の明示的接続

次の例では、リモート・ノードで単一の非デフォルトのデータベースに接続します。

```
/* 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 non-default database */
EXEC SQL CONNECT :username IDENTIFIED BY :password
      AT DB_NAME USING :db_string;
```

この例の識別子は次の目的で使用されています。

- ホスト変数 *username* および *password* は有効ユーザーを識別します。
- ホスト変数 *db_string* には、リモート・ノードの非デフォルト・データベースに接続するための Net8 構文が含まれています。
- 未宣言の識別子 *DB_NAME* は非デフォルト接続に名前を付けます。Oracle が使用する識別子であり、ホストまたはプログラム変数ではありません。

USING 句には *DB_NAME* と対応付けるネットワーク、マシンおよびデータベースを指定します。その後、AT 句 (*DB_NAME* 付き) を使用している SQL 文は、*db_string* によって指定されたデータベースで実行されます。

もう 1 つの方法として、次の例に示すように、AT 句で文字ホスト変数を使用できます。

```
/* 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 non-default database using db_name */
EXEC SQL CONNECT :username IDENTIFIED BY :password
      AT :db_name USING :db_string;
...
```

db_name がホスト変数の場合、DECLARE DATABASE 文は必要ありません。*DB_NAME* が未宣言の識別子のときのみ、CONNECT...AT *DB_NAME* 文を実行する前に DECLARE *DB_NAME* DATABASE 文を実行する必要があります。

SQL 操作 権限が与えられている場合、非デフォルト接続で SQL DML 文を実行できます。たとえば次のように入力します。

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

次の例では、*db_name* はホスト変数です。

```
EXEC SQL AT :db_name DELETE ...
```

db_name がホスト変数の場合、SQL 文によって参照されるすべてのデータベース表は DECLARE TABLE 文で定義する必要があります。そうしないと、プリコンパイラは警告を発行します。

詳細な情報は、D-4 ページの「[DECLARE TABLE の使用](#)」および F-39 ページの「[DECLARE TABLE \(Oracle 埋込み SQL 宣言文\)](#)」を参照してください。

PL/SQL ブロック PL/SQL ブロックは、AT 句を使用して実行できます。次に構文例を示します。

```
EXEC SQL AT :db_name EXECUTE
begin
    /* PL/SQL block here */
end;
END-EXEC;
```

カーソルの制御

カーソル制御文 OPEN、FETCH、CLOSE などは例外です。これらは AT 句を使用しません。カーソルと明示的に識別されたデータベースとを対応付ける場合は、次に示すとおり、DECLARE CURSOR 文で AT 句を使用してください。

```
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;
```

db_name がホスト変数のとき、DECLARE されたカーソルを参照するすべての SQL 文の有効範囲内で宣言する必要があります。たとえば、あるサブプログラムでカーソルを OPEN し、その後別のサブプログラムでそのカーソルから FETCH する場合、*db_name* をグローバルとして宣言する必要があります。

カーソルから OPEN、CLOSE および FETCH を実行する場合、AT 句は使用しません。SQL 文が実行されるのは、DECLARE CURSOR 文の AT 句で名前を付けられたデータベースか、カーソル宣言で AT 句が使用されていない場合はデフォルトのデータベースです。

AT: *host_variable* 句を使用すると、カーソルと対応付けられた接続を変更できます。しかし、カーソルがオープンされているときは対応を変更できません。次の例を考えてみます。

```
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 ...
```

これは、2 番目の OPEN 文を実行するときに *emp_cursor* がまだオープンされているので、無効となります。異なる複数の接続に対して複数のカーソルが別個に維持されるのではありません。*emp_cursor* は 1 つしかなく、*emp_cursor* は、別の接続に対して再度オープンする前にクローズする必要があります。この例のデバッグをするには、カーソルを再度オープンする前に次のようにクローズするのみで済みます。

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

動的 SQL

動的 SQL 文は、文中では AT 句が使用されないカーソル制御文に類似しています。

動的 SQL 方法 1 では、非デフォルト接続で文を実行する場合、AT 句を使用する必要があります。次に例を示します。

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

方法 2、3、4 では、非デフォルト接続で文を実行する場合、DECLARE STATEMENT 文でのみ AT 句を使用します。PREPARE、DESCRIBE、OPEN、FETCH および CLOSE のようなその他の動的 SQL 文ではすべて AT 句を使用しません。次の例は方法 2 を示します。

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

次の例は方法 3 を示します。

```
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;
```


複数の明示的接続

単一の明示的接続の場合と同様に、複数の明示的接続に `AT db_name` 句を使用できます。次の例では、2つの非デフォルトのデータベースに同時に接続しています。

```
/* 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 non-default 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;
```

識別子 `DB_NAME1` および `DB_NAME2` を宣言して、2つの非デフォルト・ノードのデフォルトのデータベースに名前を指定します。その結果、SQL 文は後でデータベースを名前によって参照できます。

もう1つの方法として、次の例に示すように、`AT` 句でホスト変数を使用できます。

```
/* declare needed host variables */
char  username[10]   = "scott";
char  password[10]   = "tiger";
char  db_name[20];
char  db_string[20];
int   n_defs = 3;    /* number of connections to make */
...
for (i = 0; i < n_defs; i++)
{
    /* get next database name and Net8 string */
    printf("Database name: ");
    gets(db_name);
    printf("Net8) string: ");
    gets(db_string);
    /* do the connect */
    EXEC SQL CONNECT :username IDENTIFIED BY :password
          AT :db_name USING :db_string;
}
```

また、次の例に示すように、この方法を使用して同じデータベースに複数の接続が行えます。

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

複数の接続に同じ Net8 文字列を使用する場合も、個々に異なるデータベース名を使用する必要があります。ただし、データベース名はデフォルトおよび非デフォルトのデータベースを識別するため、1つのデータベース名で同じデータベースに2回接続できます。

データの整合性の確認

アプリケーション・プログラムは、複数のリモート・データベースでデータを操作するトランザクションの整合性を確認する必要があります。つまり、プログラムはトランザクションのすべての SQL 文をコミットするか、またはロールバックする必要があります。このことはネットワーク接続が失敗した場合、またはシステムの1つが破損した場合は不可能です。

たとえば、2つの会計データベースで作業をしているとします。一方のデータベースで会計を借方に記入し、他方のデータベースで貸方に記入します。それから、それぞれのデータベースで COMMIT を発行します。両方のトランザクションがコミットまたはロールバックされたことは、プログラム側で確認してください。

暗黙的接続

暗黙的接続は Oracle8 の分散問合せ機能を通じてサポートされます。この機能は明示的接続を必要としませんが、SELECT 文しかサポートしません。分散問合せを使用すると、単一の SELECT 文によって複数の非デフォルトのデータベース上のデータにアクセスできます。

分散問合せ機能はデータベース・リンクを利用します。ここでは、接続そのものではなく CONNECT 文に名前を割り当てます。実行時に、埋込み SELECT 文は指定した Oracle8 Server によって実行され、必要なデータを得るために非デフォルトのデータベースに「暗黙的」に接続します。

単一の暗黙的接続

次の例では、単一の実デフォルトのデータベースに接続します。最初にプログラムは次の文を実行して、データベース・リンクを定義します。(データベース・リンクは通常、DBA またはユーザーによって対話的に確立されます。)

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

それから、次に示すとおり、プログラムはデータベース・リンクを使用して非デフォルトの EMP 表を問い合わせることができます。

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

データベース・リンクは、埋込み SQL 文の AT 句で使用されているデータベース名とは関係がありません。単に、非デフォルト・データベースの位置、データベースへのパス、使用する Oracle8 ユーザー名とパスワードを Oracle8 に伝えるのみです。データベース・リンクは明示的に削除されるまで、データ・ディクショナリに格納されます。

上の例で、デフォルトの Oracle8 は、データベース・リンク *db_link* を使用し、Net8 を介して非デフォルトのデータベースにログインします。問合せはデフォルトのサーバーに送られますが、非デフォルトのデータベースに転送されて実行されます。

データベース・リンクを簡単に参照するため、次のようにシノニムを作成できます。(通常、対話的に確立されます。)

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

さらに、プログラムは次のようにして、非デフォルト EMP 表を問い合わせることができます。

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

これによって *emp* に位置が透過的になります。

複数の暗黙的接続

次の例では、2つの非デフォルトのデータベースに同時に接続しています。まず、2つのデータベース・リンクを定義し、2つのシノニムを作成するために次の順序の文を実行します。

```
EXEC SQL CREATE DATABASE LINK db_link1
      CONNECT TO username1 IDENTIFIED BY password1
      USING 'NYNON';
EXEC SQL CREATE DATABASE LINK db_link2
      CONNECT TO username2 IDENTIFIED BY password2
      USING 'CHINON';
```

```
EXEC SQL CREATE SYNONYM emp FOR emp@db_link1;  
EXEC SQL CREATE SYNONYM dept FOR dept@db_link2;
```

プログラムからは、次のようにして非デフォルトの EMP 表および DEPT 表の間合せができます。

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

Oracle8 は、*db_link1* の非デフォルトの EMP 表と *db_link2* の非デフォルトの DEPT 表とを結合して間合せを実行します。

トランザクション用語の定義

トランザクションの本題に入る前に、この項に定義されている用語を理解する必要があります。

Oracle が管理するジョブおよびタスクを「セッション」と呼びます。アプリケーション・プログラムまたは SQL*Forms などのツールを起動してデータベースに接続すると、「ユーザー・セッション」が開始されます。

Oracle では、ユーザー・セッションを同時に機能させ、コンピュータ・リソースを共有させることができます。このために、Oracle は同時実行性、つまり多くのユーザーが同じデータにアクセスすることを制御する必要があります。適切な同時制御を行わないと、「データの整合性」が損われることがあります。つまり、データまたは構造への変更が誤った順序で行われるおそれがあります。

Oracle は「ロック」（「エンキュー」と呼ばれることもあります）を使用して、データへの同時アクセスを制御します。ロックにより、データの表や行などのデータベース・リソースのユーザーに一時的な所有権が与えられます。つまり、このユーザーがデータの変更を終了するまで他のユーザーは同じデータの変更はできません。

デフォルトのロック機構によって Oracle のデータおよび構造が保護されているため、リソースを明示的にロックする必要はありません。ただし、デフォルトのロックを変更した方がよい場合、表または行単位で「データ・ロック」を要求できます。「行共有」および「行排他」など、いくつかのロックの「モード」から選択できます。

複数のユーザーが同一のデータベース・オブジェクトにアクセスすると、「デッドロック」が発生することがあります。たとえば、2 人のユーザーが同一の表を更新している場合、互いに相手側が現在ロックしている行を更新しようとする、待ち状態になります。それぞれのユーザーが相手側の使用しているリソースを待っているため、Oracle がデッドロックを解除するまで、両者とも処理を続行できません。Oracle は最低作業量を完了した参加トランザクションにエラーの信号を出し、「リソース待機中にデッドロックが検出されました」という Oracle エラー・コードが SQLCA の *sqlcode* に戻されます。

表が 1 人のユーザーによって問い合わせられ、同時に別のユーザーによって更新されているとき、問合せ用表データの「読取り一貫性」ビューが生成されます。つまり一度問合せが開始

されてから処理がそのまま続行しているときは、問合せによって読み込まれるデータは変化しません。更新アクティビティが継続する際、表データの「スナップショット」が作成され、ロールバック・セグメントの変更が記録されます。Oracle はこのロールバック・セグメント内の情報をもとに読取り一貫性問合せの結果を作成します。（または、必要に応じて変更を取り消します。）

トランザクションがデータベースを保護する方法

Oracle はトランザクション指向です。つまり、トランザクションを使用してデータの整合性を維持します。トランザクションとは、あるタスクを完了するために定義した 1 つ以上の論理的に対応付けられた SQL 文です。Oracle はこの一連の SQL 文を 1 つの単位として扱うため、これらの文による変更をすべて同時に「コミット」するか「ロールバック」します。アプリケーション・プログラムがトランザクションの途中で異常終了すると、データベースは自動的に前の状態（トランザクション処理の前の状態）にロールバックされます。

以後の項では、トランザクションの設計および制御方法について説明します。次の操作方法を学習します。

- データベースへの接続
- 同時接続
- トランザクションの開始および終了
- COMMIT 文を使用したトランザクションの確定
- ROLLBACK TO 文とともに SAVEPOINT 文を使用してのトランザクションの部分的な取消し
- ROLLBACK 文を使用してのトランザクション全体の取消し
- RELEASE オプションを指定してのリソースの解放およびデータベースのログアウト
- SET TRANSACTION 文を使用した読取り専用トランザクションの設定
- FOR UPDATE 句または LOCK TABLE 文を使用してのデフォルト・ロックの変更（オーバーライド）

この章で説明する SQL 文の詳細は『Oracle8i SQL リファレンス』を参照してください。

トランザクションの開始・終了方法

プログラム内の最初の実行 SQL 文（CONNECT 以外）によってトランザクションを開始します。1 つのトランザクションが終了すると、次の実行 SQL 文が自動的に別のトランザクションを開始します。つまり、すべての実行文はトランザクションの一部です。宣言 SQL 文はロールバックできません。またこの文はコミットする必要もないため、トランザクションの一部とはみなしません。

トランザクションは、次のいずれかの方法で終了します。

- COMMIT または ROLLBACK 文を記述します。RELEASE オプションは、付けても付けなくてもかまいません。これらの文はデータベースへの変更を「明示的」に確定または取り消します。
- 実行の前後両方に自動 COMMIT を発行するデータ定義文（たとえば、ALTER、CREATE または GRANT など）を記述します。これはデータベースへの変更を「暗黙的」に確定します。

システム障害が発生した場合、またはソフトウェア上の問題、ハードウェア上の問題、強制割込みなどが原因で予期しないセッション停止が発生した場合にも、トランザクションは終了します。Oracle によってそのトランザクションはロールバックされます。

プログラムがトランザクションの途中で異常終了すると、Oracle によってエラーが検出されるとともにそのトランザクションがロールバックされます。オペレーティング・システムに障害が発生すると、データベースが元（トランザクション処理の前）の状態にリストアされます。

COMMIT 文の使用法

プログラムを COMMIT 文または ROLLBACK 文で分割しなければ、Oracle ではそのプログラム全体を 1 つのトランザクションとみなします。（ただし、そのプログラムにデータ定義文が指定されている場合は例外です。そのときはデータ定義文が自動 COMMIT を発行します。）

COMMIT 文を使用するとデータベースへの変更を確定できます。変更を COMMIT するまで、他のユーザーは変更されたデータにアクセスできません。つまり、他のユーザーはこのトランザクションを開始する前の状態のデータを見えています。COMMIT 文は、次のことを明確に行います。

- カレント・トランザクション中に行ったデータベースに対する変更をすべて確定します。
- これらの変更を他のユーザーに対し明確にします。
- すべてのセーブポイントを消去します（次の項の「SAVEPOINT 文の使用」を参照）。
- 解析ロック以外の行および表のロックをすべて解除します。
- CURRENT OF 句内で参照されているカーソル、または MODE=ANSI のときは COMMIT 文に指定されている接続の明示的カーソルをすべてクローズします。
- トランザクションを終了します。

COMMIT 文はホスト変数の値にもプログラム内の制御の流れにも影響しません。

MODE=ORACLE のとき、CURRENT OF 句内で参照されない明示的カーソルはコミットの前後でオープンしたままです。これによってパフォーマンスが向上します。例は 3-24 ページの「[COMMIT をまたぐ FETCH](#)」を参照してください。

これらの処理は通常の処理の一部になっているため、COMMIT 文はプログラムのメイン・パスにインラインで設定する必要があります。プログラムを終了する前に、必ず保留状態の変更を明示的に COMMIT してください。そうしないと、Oracle はそれらの変更をロールバックします。次の例では、トランザクションをコミットし Oracle への接続を切断します。

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

オプション指定のキーワード WORK は、ANSI 互換性のために提供されています。RELEASE オプションは、プログラムが使用している Oracle のリソース（ロックおよびカーソル）をすべて解放してデータベースをログアウトします。

データ定義文は実行の前後両方で自動コミットを発行するため、データ定義文の後に COMMIT 文を記述する必要はありません。したがってデータ定義文が正常終了しても異常終了しても、その前のトランザクションがコミットされます。

SAVEPOINT 文の使用法

SAVEPOINT 文を使用すると、トランザクションの処理のカレント・ポイントにマークを設定し名前を指定できます。マークを設定したそれぞれの点を「セーブポイント」と呼びます。たとえば、次の文により *start_delete* というセーブポイントを設定します。

```
EXEC SQL SAVEPOINT start_delete;
```

セーブポイントによって長いトランザクションを分割できるため、より複雑なプロシージャを制御できます。たとえば、単一のトランザクションが複数の機能を実行しているときに、それぞれの関数の前にセーブポイントを設定できます。この結果、ある関数が異常終了したときに、簡単に Oracle データを前の状態にリストアし、後でこの関数を再実行できます。

トランザクションの一部を取り消すには、ROLLBACK 文とその TO SAVEPOINT 句によってセーブポイントを指定します。次の例では、MAIL_LIST 表をアクセスして新しいリストを挿入し、古いリストを更新してから、使用されていないリストを削除します。この削除後に、削除された行数を知るために SQLCA 内の *sqlerrd* の 3 番目の要素を調べます。削除された行数が必要以上に大きいときは、セーブポイントの *start_delete* までロールバックしてこの削除を取り消します。

```
...
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);
```

あるセーブポイントまでロールバックすることによって、そのセーブポイント以降のすべてのセーブポイントが消去されます。ただしロールバックしたセーブポイントはそのまま残ります。たとえば、5つのセーブポイントを設定しているときに3番目のセーブポイントまでロールバックすると、4番目と5番目のセーブポイントのみが消去されます。

2つのセーブポイントに同じ名前を指定すると、前のセーブポイントが消去されます。COMMIT 文または ROLLBACK 文を実行すると、すべてのセーブポイントが消去されます。

WHENEVER 宣言文の詳細は、9-24 ページの「[WHENEVER 宣言文の使用](#)」を参照してください。

ROLLBACK 文の使用法

ROLLBACK 文を使用すると、保留状態になっているデータベースへの変更を取り消すことができます。たとえば表から行を誤って削除してしまったときなどは、ROLLBACK 文を使用できれば元のデータをリストアできます。TO SAVEPOINT 句を使用すると、カレント・トランザクションの途中の文までロールバックできます。したがって、変更をすべて取り消す必要はありません。

また、未完成のトランザクションを開始したとき（たとえば、SQL 文が正常に実行されないなど）は、ROLLBACK によって起点まで戻するため、データベースの整合性は維持されます。具体的には、ROLLBACK 文は次の処理を行います。

- カレント・トランザクション中に実行されたデータベースへの変更を取り消します。
- すべてのセーブポイントを消去します。
- トランザクションを終了します。
- 解析ロック以外の行および表のロックをすべて解除します。
- CURRENT OF 句内で参照されたカーソル、または MODE=ANSI のときはすべての明示的カーソルをクローズします。

ROLLBACK 文は、ホスト変数の値やプログラム内の制御の流れには影響を与えません。

MODE=ORACLE のときは、CURRENT OF 句内で参照されない明示的カーソルはロールバックの前後でオープンしたままになります。

ROLLBACK TO SAVEPOINT 文は、具体的には次の処理を行います。

- 指定されたセーブポイント以降のデータベースへの変更を取り消します。
- 指定したセーブポイント以降のセーブポイントをすべて消去します。
- 指定したセーブポイントが設定された位置以後に取得された行および表のロックをすべて解除します。

ROLLBACK TO SAVEPOINT 文では RELEASE オプションを指定できないことに注意してください。

ROLLBACK 文は例外処理の一部になっているため、プログラムのメイン・パスではなくエラー処理ルーチン内に指定する必要があります。次の例では、トランザクションをロールバックして Oracle への接続を切り離します。

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

オプション指定のキーワード WORK は、ANSI 互換性のために提供されています。RELEASE オプションは、プログラムによって使用されているリソースをすべて解放してデータベースから切断します。

WHENEVER SQLERROR GOTO 文からエラー処理ルーチンに分岐するときに、そのルーチンに ROLLBACK 文が指定されていると、ROLLBACK 文でエラーが発生したときにプログ

ラムが無限ループに陥るおそれがあります。次の例に示すように、ROLLBACK 文の前に WHENEVER SQLERROR CONTINUE を記述することによって、このような無限ループを回避できます。

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;

for (;;)
{
    printf("Employee number? ");
    gets(temp);
    emp_number = atoi(temp);
    printf("Employee name? ");
    gets(emp_name);
    EXEC SQL INSERT INTO emp (empno, ename)
        VALUES (:emp_number, :emp_name);
    ...
}
...
sql_error:
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK WORK RELEASE;
printf("Processing error\n");
exit(1);
```

プログラムが異常終了すると、Oracle によってトランザクションが自動的にロールバックされます。3-21 ページの「[RELEASE オプションの使用法](#)」を参照してください。

文レベルのロールバック

どの SQL 文を実行する場合でも、その前に Oracle によって暗黙的セーブポイント（Oracle 用なので使用は不可）が設定されます。したがって、この文にエラーが発生したときに文が自動的にロールバックされ、適切なエラー・コードが SQLCA 内の *sqlcode* に戻ります。たとえば、INSERT 文が一意の索引内に同じ値を挿入しようとしたためエラーが発生すると、この文はロールバックの対象になります。

Oracle は、デッドロックを解除するために単一の SQL 文をロールバックすることもあります。Oracle は、この原因となっているトランザクションの 1 つにエラーを通知して、そのトランザクションのカレント文をロールバックします。

エラーとなった SQL 文が開始した作業のみが失われます。つまり、カレント・トランザクション内のこの文の前に行われた作業はすべて保存されます。したがってデータ定義文がエラーとなった場合でも、それ以前の自動コミットは取り消されません。

SQL 文は実行前に解析されます。つまり、構文規則に従っているかどうか、および有効なデータベース・オブジェクトを参照しているかどうかを検証されます。SQL 文の実行中にエラーが検出されるとロールバックが発生しますが、SQL 文の解析中にエラーが検出されても文はロールバックされません。

RELEASE オプションの使用法

プログラムが異常終了すると、変更は Oracle によって自動的にロールバックされます。作業の明示的コミットもロールバックもせずに、RELEASE オプションを指定して Oracle から切断すると、プログラムは異常終了します。プログラムが正しく処理を実行し、オープン状態のカーソルをクローズし、明示的に作業をコミットまたはロールバックし、Oracle から切断した後、最後に制御をユーザーに戻すとプログラムは正常終了します。

実行される最後の SQL 文が次のいずれかの場合、プログラムは正常終了します。

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

または

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

トークンの WORK はオプションです。前述以外の場合は、ユーザー・セッションが取得したロックおよびカーソルは、プログラム終了後も、Oracle がこのユーザー・セッションの停止を認識するまでそのまま保持されます。このため、マルチユーザー環境では、ロックされたリソースを他のユーザーが待機する時間が長くなることがあります。

SET TRANSACTION 文の使用法

SET TRANSACTION 文を使用すると、読取り専用トランザクションを開始できます。これにより、繰り返し読取りが可能になるため、読取り専用トランザクションは 1 つ以上の表に対して複数の問合せを実行するのに便利です。一方、他のユーザーは同じ表を更新します。SET TRANSACTION 文の例を次に示します。

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

SET TRANSACTION 文は、読取り専用トランザクション内の最初の SQL 文である必要があります。また、1 つのトランザクション内で一度しか使用することができません。READ ONLY パラメータは必須です。READ ONLY パラメータを使用しても他のトランザクションに影響はありません。

読取り専用トランザクション内で使用できるのは SELECT 文、COMMIT 文および ROLLBACK 文のみです。したがって、INSERT 文、DELETE 文または SELECT FOR UPDATE OF 文などを使用するとエラーが発生します。

読取り専用トランザクションの実行中は、すべての問合せがデータベース内の同一のスナップショットを参照するため、複数の表の複数の問合せの読取り一貫性ビューが生成されます。その他のユーザーは、通常どおりデータの問合せまたは更新を続行できます。

読取り専用トランザクションは、COMMIT 文、ROLLBACK 文またはデータ定義文によって終了します。（データ定義文は、暗黙的 COMMIT を発行することを思い出してください。）

次の例は、店の管理者が読取り専用トランザクションを使用して1日、1週間前および1か月前の販売状態をチェックして、要約レポートを生成する様子を示しています。このレポートは、このトランザクションの実行中にデータベースを更新する他のユーザーによる影響を受けません。

```
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 */
```

デフォルト・ロックのオーバーライド

デフォルトでは、Oracle によって暗黙的に（自動的に）多数のデータ構造がロックされます。ただし、デフォルトのロックを無効にして、別のロックを有効にするときは、行や表を特定して、そこにデータ・ロックを要求できます。明示的ロックによって、トランザクション中に表へのアクセスを共有または拒絶したり、複数の表および複数の問合せの読取り一貫性を保持できます。

SELECT FOR UPDATE OF 文を使用すると、表の特定行を明示的にロックすることによって、UPDATE または DELETE が実行されるまでそれらの行が変更されないように制御できます。ただし、Oracle では UPDATE 時または DELETE 時に行レベルのロックが自動的に取得されます。したがって、UPDATE または DELETE の前に行をロックするときにかぎり FOR UPDATE OF 句を使用してください。

LOCK TABLE 文を使用すると、表全体を明示的にロックできます。

FOR UPDATE OF の使用方法

UPDATE 文または DELETE 文の CURRENT OF 句内で参照されるカーソルを DECLARE するときに、FOR UPDATE OF 句を使用すると行の排他ロックを取得できます。SELECT FOR UPDATE OF によって、まず更新または削除対象とする行が決定し、次にこのアクティブ・セット内のそれぞれの行がロックされます。この設定は、ある行内の既存の値に従って更新処理を行うときなどに便利です。更新前に別のユーザーによってその行が更新されないようにする必要があります。

FOR UPDATE OF 句はオプションです。たとえば次のように記述するかわりに、

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

FOR UPDATE OF 句を削除すると、コードが単純になります。

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

CURRENT OF 句は、必要に応じて FOR UPDATE 句を追加するようプリコンパイラに指示します。CURRENT OF 句を使用して、カーソルから FETCH した最後の行を参照します。例は 6-16 ページの「[CURRENT OF 句の使用方法](#)」を参照してください。

制限

FOR UPDATE OF 句を使用すると、複数の表を参照できません。

明示的な FOR UPDATE OF または暗黙的な FOR UPDATE によって行の排他ロックが取得されます。行はすべて、FETCH 時ではなく OPEN 時にロックされます。COMMIT または ROLLBACK すると、行のロックは解除されます。(セーブポイントに ROLLBACK するときには解除されません。) したがって、COMMIT 後に FOR UPDATE カーソルから FETCH を実行することはできません。

LOCK TABLE の使用方法

LOCK TABLE 文を使用すると、指定したロック・モードで 1 つ以上の表をロックできます。たとえば、次の文は「行共有」モードで EMP 表をロックします。行共有ロックによって表への同時アクセスが可能となります。つまり、他のユーザーがその表全体をロックして排他使用することはできません。

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

ロック・モードによって、その表に設定できる他のロックが決定されます。たとえば、同時に多数のユーザーが 1 つの表に対して行共有ロックを取得できる一方で、「排他」ロックを取得できるのは一度に 1 ユーザーのみです。あるユーザーが表を排他ロックしている間は、他のユーザーはその表において行を INSERT、UPDATE または DELETE できません。

ロック・モードの詳細は『Oracle8i 概要』を参照してください。

オプションのキーワード NOWAIT を指定すると、別のユーザーがこの表をロックしているときはその表の解放を待たないよう Oracle に指示できます。制御はすぐにプログラムに戻されます。このため、プログラムではロックの取得を再度試みるまでの間に別の作業ができます。(SQLCA 内の *sqlcode* をチェックすれば、LOCK TABLE が失敗したかどうかわかります。) NOWAIT を省略すると、Oracle はその表が使用可能になるまで待ち状態になります。この待ち時間には制限がありません。

表ロックによって他のユーザーからの表の問合せが禁止されることはありません。このため、問合せで表ロックは取得されません。したがって、問合せが他の問合せまたは更新を妨げることはありません。2 つの異なるトランザクションが同じ行を更新するときのみ、一方のトランザクションは他方のトランザクションが終了するまで待ち状態になります。

LOCK TABLE 文は暗黙にすべてのカーソルを閉じます。

トランザクションが COMMIT または ROLLBACK を発行すると、表ロックは解除されます。

COMMIT をまたぐ FETCH

COMMIT と FETCH を併用するときは、CURRENT OF 句は使用しないでください。そのかわりに、まず各行の ROWID を SELECT します。次にその値を使用して、更新または削除中のカレント行を確定します。次に例を示します。

```
...
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;
    ...
}
```

ただし、FETCH された行はロックされません。つまり、ある行を読み込んでも、その行を更新または削除する前に別のユーザーがその行を変更すると、結果に矛盾が生じる場合があります。

分散トランザクションの処理

「分散データベース」とは、異なるノード上の複数の物理データベースで構成される単一の論理データベースです。「分散文」とは、データベース・リンクによってリモート・ノードにアクセスする任意の SQL 文です。「分散トランザクション」には、分散データベースの複数のノードでデータを更新するための分散文が少なくとも 1 つ設定されています。その更新が 1 つのノードにのみ影響するときは、トランザクションは分散型ではありません。

COMMIT を発行すると、分散トランザクションによる影響を受けたそれぞれのデータベースへの変更が確定されます。COMMIT のかわりに ROLLBACK を発行すると、すべての変更が取り消されます。ただし、コミットまたはロールバック中にネットワークまたはマシンで障害が発生すると、分散トランザクションの状態は不明または「インダウト」になることがあります。そのようなときに FORCE TRANSACTION システム権限があれば、FORCE 句によってローカル・データベースでトランザクションを手動でコミットまたはロールバックできます。このトランザクションは、トランザクション ID を引用符付きリテラルで囲んで指定する必要があります。トランザクション ID はデータ・ディクショナリ・ビュー DBA_2PC_PENDING に収められています。次にいくつかの例を示します。

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

FORCE は指定されたトランザクションのみをコミットまたはロールバックするため、カレント・トランザクションには影響しません。未確定のトランザクションはセーブポイントに手動でロールバックできません。

COMMIT 文中の COMMENT 句を使用すると、分散トランザクションと対応付けるためのコメントを指定できます。トランザクションがインダウトになると、Oracle は COMMENT で指定済みのテキストを、トランザクション ID とともにデータ・ディクショナリ・ビュー DBA_2PC_PENDING 内に格納します。このテキストは長さが 50 文字以下の引用符付きリテラルである必要があります。次に例を示します。

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

分散トランザクションの詳細は『Oracle8i 概要』を参照してください。

ガイドライン

次のガイドラインに従うと、いくつかの問題を回避できます。

アプリケーションの設計

アプリケーションを設計するときは、論理的に関連する処理を 1 つのトランザクション内にグループ化してください。うまく設計されたトランザクションには特定のタスクを完了するのに必要なすべてのステップが含まれています。余分なものも不足もありません。

表内で参照するデータは一貫している必要があります。したがって、トランザクション内の SQL 文は一貫した方法に従ってデータを変更する必要があります。たとえば、2 つの銀行口座間の資金移動には一方の口座の借方勘定ともう一方の貸方勘定が含まれています。どちらの処理も同時に正常終了または失敗する必要があります。一方の口座への新規預金などといった、この取引とは無関係の更新取引はトランザクションには取り込まないでください。

ロックの取得

アプリケーション・プログラム内に SQL のロック文を指定しているときは、ロックを要求する Oracle ユーザーにそのロックを取得する権限があることを確認してください。データベース管理者（DBA）はどの表でもロックできます。DBA 以外のユーザーは、自分が所有する表または権限を持つ表（ALTER、SELECT、INSERT、UPDATE、DELETE など）のみをロックできます。

PL/SQL の使用方法

PL/SQL ブロックがトランザクションの一部になっている場合、そのブロック内の COMMIT と ROLLBACK は、そのトランザクション全体に影響します。次の例では、ROLLBACK は UPDATE および INSERT による変更を取り消します。

```
EXEC SQL INSERT INTO EMP ...
EXEC SQL EXECUTE
  BEGIN
    UPDATE emp ...
    ...
  EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
      ROLLBACK;
    ...
  END;
END-EXEC;
...
```

データ型とホスト変数

この章では、Pro*C/C++ プログラムを作成する際に必要な基本的な情報について説明します。この章のトピックは次のとおりです。

- [Oracle のデータ型](#)
- [ホスト変数](#)
- [標識変数](#)
- [VARCHAR 変数](#)
- [カーソル変数](#)
- [CONTEXT 変数](#)
- [汎用 ROWID](#)
- [ホスト構造体](#)
- [ポインタ変数](#)
- [各国語サポート](#)
- [NCHAR 変数](#)

この章には、学習用の完全なサンプル・プログラムもいくつか記載されています。これらのプログラムには、この章で説明する技法の使用例が示されています。これらは *demo* ディレクトリに入っており、オンラインで使用できるため、コンパイル、実行および必要に応じた修正もできます。

Oracle のデータ型

Oracle では、「内部データ型」と「外部データ型」の 2 種類のデータ型が識別されます。内部データ型には、Oracle がデータベース表に列値を格納する方法と、擬似列値（NULL、SYSDATE、USER など）を表すのに使用する形式を指定します。外部データ型には、入力ホスト変数および出力ホスト変数に値を格納するのに使用する形式を指定します。

Oracle の内部データ型（ビルトインとも呼ばれます）の詳細は『Oracle8i SQL リファレンス』を参照してください。

内部データ型

データベースの列に保存される値については、Oracle は [表 4-1](#) に示す内部データ型を使用します。

表 4-1 Oracle 内部データ型

名前	説明
VARCHAR2	可変長文字列、<=4000 バイト
NVARCHAR2 または NCHAR VARYING	可変長シングルスバイト文字列または固定幅マルチバイト文字列、<=4000 バイト
NUMBER	100 をベースとして表された、精度とスケールのある数値
LONG	可変長文字列、<=2**31-1 バイト
ROWID	バイナリ値
DATE	固定長の日付 + 時刻値、7 バイト
RAW	可変長バイナリ・データ、<=2000 バイト
LONG RAW	可変長バイナリ・データ、<=2**31-1 バイト
CHAR	固定長文字列、<=2000 バイト
NCHAR	固定長シングルスバイト文字列または固定幅マルチバイト文字列、<=2000 バイト
BFILE	外部ファイル・バイナリ・データ、<=4G バイト
BLOB	バイナリ・データ、<= 4G バイト
CLOB	文字データ、<=4G バイト
NCLOB	マルチバイト・キャラクタ・データ、<=4G バイト

これらの内部データ型は、C のデータ型とはかなり異なる場合があります。たとえば、C には Oracle NUMBER データ型に相当するデータ型はありません。ただし、NUMBER は、いくらかの制限はありますが、**float** および **double** などの C のデータ型の間で変換できます。たとえば、Oracle NUMBER データ型は小数点以下最大 38 桁の精度で指定できますが、カレント C 処理系では、**倍精度値** をそこまでの精度で表せるデータ型はありません。

Oracle の NUMBER データ型は値を正確に（精度の制限内で）表しますが、浮動小数点形式では 10.0 などの値を正確に表すことができません。

構造化されていないデータ（テキストおよびグラフィック・イメージ、ビデオ・クリップ、サウンド波形）を格納するには、LOB データ型を使用します。BFILE データは、データベース外部のオペレーティング・システム・ファイルに格納されます。LOB 型には、データの位置を指定する「ロケータ」が格納されます。詳細は、[第 16 章「ラージ・オブジェクト \(LOB\)」](#)を参照してください。

NCHAR と NVARCHAR2 は、NLS（各国語サポート）文字データの格納に使用されます。これらのデータ型の説明は 4-45 ページの「[各国語サポート](#)」を参照してください。

外部データ型

表 4-2 に示すように、外部データ型にはすべての内部データ型と、C の構文とほぼ一致するいくつかのデータ型が含まれています。たとえば、STRING 外部データ型は、C では NULL 終了記号付きの文字列にあたります。

表 4-2 Oracle の外部データ型

名前	説明
VARCHAR2	可変長文字列、<=65535 バイト
NUMBER	100 をベースとして表された、10 進数
INTEGER	符号付き整数
FLOAT	実数
STRING	NULL 終了記号を付けた可変長文字列
VARNUM	10 進数、NUMBER と同様だが表現の長さのコンポーネントが含まれる
LONG	固定長文字列、2**31-1 バイトまで
VARCHAR	可変長文字列、<= 65533 バイト
ROWID	バイナリ値、外部の長さはシステムによって異なる
DATE	固定長日付 / 時刻値、7 バイト
VARRAW	可変長バイナリ・データ、<=65533 バイト
RAW	固定長バイナリ・データ、<=65533 バイト
LONG RAW	固定長バイナリ・データ、<=2**31-1 バイト
UNSIGNED	符号なし整数
LONG VARCHAR	可変長文字列、<=2**31-5 バイト
LONG VARRAW	可変長バイナリ・データ、<=2**31-5 バイト
CHAR	固定長文字列、<=65535 バイト
CHARZ	NULL 終了記号を付けた固定長文字列、<=65534 バイト
CHARF	CHAR のデフォルトを VARCHAR2 または CHARZ ではなく CHAR にするために、TYPE 文または VAR 文で使用する

次に Oracle データ型を簡単に説明します。

VARCHAR2

VARCHAR2 データ型を使用して、可変長文字列を格納します。VARCHAR2 値の最大長は 64KB です。

VARCHAR2(*n*) 値の最大長は、文字数ではなくバイト数で指定します。そのため、VARCHAR2(*n*) 変数がマルチバイトの文字を格納している場合、最大長は *n* 文字より少なくなります。

CHAR_MAP=VARCHAR2 オプションを使用してプリコンパイルすると、Oracle は **char[n]** または **char** として宣言しているすべてのホスト変数に、VARCHAR2 データ型を割り当てます。

入力時 Oracle は入力ホスト変数に指定されたバイト数を読み込み、後続する空白を取り除き、格納先データベース列に入力値を格納します。注意が必要です。未初期化ホスト変数には、一連の NULL が含まれている場合があります。したがって、必ず、文字入力ホスト変数の宣言長まで空白埋込みを行ってください。NULL 終了記号は付けなくても構いません。

入力値がデータベース列の定義より長い場合、Oracle はエラーを生成します。入力値がすべて空白である場合、Oracle はこれを 1 つの NULL として扱います。

文字値が有効な数を表す場合、Oracle は文字値を NUMBER 列値に変換できます。その他の場合、Oracle はエラーを生成します。

出力時 Oracle は出力ホスト変数に指定されたバイト数を戻し、必要に応じて空白埋込みを行ってから、出力値を格納先ホスト変数に割り当てます。NULL が戻されると、Oracle はホスト変数に空白を埋め込みます。

出力値がホスト変数の宣言長より長い場合、Oracle はその値の長すぎる分を切り捨ててからホスト変数に割り当てます。標識変数がそのホスト変数に対応付けられている場合、Oracle は標識変数を出力値の元の長さに設定します。

Oracle は NUMBER 列値を文字値に変換できます。文字ホスト変数の長さによって精度が決定します。ホスト変数の長さがその数に対して短すぎる場合は、科学表記法が使用されます。たとえば、列値 123456789 を長さ 6 文字の文字ホスト変数に SELECT すると、Oracle は値 '1.2E08' を戻します。

NUMBER

NUMBER データ型を使用して、固定小数点数または浮動小数点数を格納します。精度およびスケールを指定できます。NUMBER 値の最大精度は 38 桁です。マグニチュード範囲は 1.0E-130 から 9.99...9E125 (9 が 38 個に続けて 0 が 88 個) です。スケールの範囲は -84 から 127 までです。

NUMBER 値は可変長形式で格納されます。つまり、1 バイトの指数部に 19 バイトの仮数部が続きます。指数部のバイトの上位 1 ビットは符号ビットであり、正数のときに設定します。下位 7 ビットは絶対値を表します。

仮数は 38 桁の数値を形成し、各バイトは 100 をベースとする 2 桁を表します。仮数の符号は、最初（左端）のバイトの値で指定されます。101 より大きければ仮数は負であり、その 1 桁目は左端のバイトから 101 を差し引いた値と等しくなります。

出力時、ホスト変数には Oracle で内部表現された数が入ります。予想される最大の数に対応するためには、出力ホスト変数を 22 バイトの長さにする必要があります。数を表現するのに使用されるバイトのみが戻されます。Oracle は、出力値に空白を埋め込むことも、NULL 終了記号を付けることもしません。戻された値の長さを知る必要がある場合、かわりに VARNUM データ型を使用します。

この外部データ型を使用する必要はほとんどありません。

INTEGER

INTEGER データ型を使用して、小数部分のない数を格納します。整数は符号付きの 2 バイトまたは 4 バイトの 2 進数です。ワード内のバイトの並びはシステムによって異なります。入力および出力ホスト変数に長さを指定する必要があります。出力時は、列値が実数の場合、Oracle は小数部分を切り捨てます。

FLOAT

FLOAT のデータ型を使用して、小数部分を持つ数または INTEGER データ型の容量を超える数を格納します。数は使用しているコンピュータの浮動小数点形式を使用して表示されます。一般的には、記憶領域に 4 バイトまたは 8 バイトを必要とします。入力および出力ホスト変数に長さを指定する必要があります。

Oracle では数の内部形式が 10 進数であるため、大半の浮動小数点処理系よりも高い精度で数を表現できます。したがって、FLOAT 変数へのフェッチを行うと、精度が低下する可能性があります。

STRING

STRING データ型は、VARCHAR2 データ型に似ていますが、STRING 値が常に NULL 文字で終了する点で異なります。CHAR_MAP=STRING オプションを使用してプリコンパイルすると、Oracle は char[n] または char として宣言しているすべてのホスト変数に、STRING データ型を割り当てます。

入力時 Oracle は指定された長さを使用して、NULL 終了記号のスキンを制限します。NULL 終了記号が見つからない場合、Oracle はエラーを生成します。長さの指定がなければ、Oracle は最大長を 2000 バイトと想定します。STRING 値の最小長は 2 バイトです。最初の文字が NULL 終了記号で、指定された長さが 2 の場合、Oracle は列が NOT NULL と定義されていなければ NULL を挿入します。列が NOT NULL と定義されている場合はエラーが発生します。空白のみからなる値は、そのまま格納されます。

出力時 Oracle は戻された最後の文字に NULL バイトを 1 つ追加します。文字列長が指定された長さを超える場合、Oracle は出力値を切り捨て、NULL バイトを追加します。NULL が SELECT される場合、Oracle は最初の文字間調整に NULL バイトを戻します。

VARNUM

VARNUM データ型は NUMBER データ型に似ていますが、VARNUM 変数の最初のバイトにその表現の長さが格納される点で異なります。

入力では、ホスト変数の最初のバイトを値の長さに設定する必要があります。出力では、ホスト変数には長さに続いて Oracle の内部表現で示した数が設定されています。この数が最大になっても支障がないように、ホスト変数の長さは 22 バイト必要です。VARNUM ホスト変数に列値を SELECT した後で、最初のバイトをチェックして値の長さを得ることができません。

通常、このデータ型はほとんど使用されません。

LONG

LONG データ型を使用して、固定長文字列を格納します。

LONG データ型は VARCHAR2 データ型に似ていますが、LONG 値の最大長が 2147483647 バイトつまり 2GB である点で異なります。

VARCHAR

VARCHAR データ型を使用して、可変長文字列を格納します。VARCHAR 変数では、2 バイト長のフィールドの後に 65533 バイト以下の文字列フィールドが続きます。しかし、VARCHAR 配列要素では、文字列フィールドの最大長は 65530 バイトです。VARCHAR 変数の長さを指定するときは、長さフィールド用に 2 バイト含まれていることを確認してください。もっと長い文字列には、LONG VARCHAR データ型を使用してください。

ROWID

Oracle8 がリリースされる前は、ROWID データ型は各表の行の物理アドレスを 16 進数として格納するのに使用されました。ROWID に含まれる行の物理アドレスにより、一度のブロック・アクセスによって行を効率的に取得できます。

Oracle8 から論理 ROWID が導入されました。索引構成表の行には、恒久物理アドレスは設定されていません。論理 ROWID には、物理 ROWID の場合と同じ構文を使用してアクセスすることができます。このため、物理 ROWID は、「データ・オブジェクト番号」（同じセグメントのスキーマ・オブジェクト）を格納できるように拡張されています。

論理 ROWID と物理 ROWID の両方（Oracle 以外の表の ROWID を含む）をサポートするために、汎用 ROWID が定義されました。

文字ホスト変数は、rowid を可読形式で格納するのに使用できます。SELECT または FETCH で rowid を文字ホスト変数に代入すると、Oracle はその 2 進値を 18 バイトの文字列に変換し、それを次の形式で戻します。

```
BBBBBBBB.RRRR.FFFF
```

BBBBBBBB はデータベース・ファイルのブロック、RRRR はブロック内の行（最初の行は 0）、FFFF はデータベース・ファイルです。これらの値は 16 進数です。たとえば、rowid

```
00000000E.000A.0007
```

は 7 番目のデータベース・ファイルの 15 番目のブロックの 11 行目を示します。

アプリケーションでの汎用 ROWID の使用方法の詳細は、4-36 ページの「汎用 ROWID」を参照してください。

一般的に、FETCH で rowid を文字ホスト変数に代入し、そのホスト変数を UPDATE 文または DELETE 文の WHERE 句の ROWID 疑似列と比較します。そのようにして、カーソルによってフェッチされた最終行を識別できます。例は 8-26 ページの「CURRENT OF の疑似実行」を参照してください。

注意：完全な移植性が必要な場合、あるいはアプリケーションが Oracle Open Gateway テクノロジーを使用して Oracle 以外のデータベースと通信する場合には、ホスト変数を宣言するときに最大長を 256（18 ではなく）に指定してください。ホスト変数の内容については何も仮定できませんが、ホスト変数は SQL 文内で通常どおりにふるまいます。

DATE

DATE データ型を使用して、7 バイト固定長フィールドに日時を格納します。表 4-3 に示すように、世紀、年、月、日、時間（24 時間フォーマット）、分、秒がこの順番で左から右に格納されています。

表 4-3 DATE 書式

バイト	1	2	3	4	5	6	7
意味	世紀	年	月	日	時	分	秒
例	119	194	10	17	14	24	13
1994 年 10 月 17 日 午後 1 時 23 分 12 秒							

世紀および年のバイトは 100 を加えた表記です。時刻、分、秒のバイトは 1 を加えた表記です。紀元前（B.C.E.）は 100 未満です。原点は紀元前 4712 年 1 月 1 日です。この日付では、世紀バイトは 53、年バイトは 88 です。時間バイトの範囲は 1 から 24 です。分と秒のバイトの範囲は 1 から 60 までです。時間のデフォルトは真夜中（1、1、1）です。

通常、このデータ型はほとんど使用されません。

RAW

RAW データ型を使用して、バイナリ・データまたはバイト文字列を格納します。RAW 値の最大長は 65535 バイトです。

RAW データは CHARACTER データに似ていますが、Oracle では RAW データは意味がないものと解釈され、RAW データをあるシステムから別のシステムに転送するときにキャラクタ・セットは変換されないという点で異なります。

VARRAW

VARRAW データ型を使用して、可変長のバイナリ・データまたはバイト文字列を格納します。VARRAW データ型は RAW データ型と似ていますが、VARRAW 変数は 2 バイト長のフィールドの後に長さが 65533 バイト以下のデータ・フィールドが続いています。もっと長い文字列には、LONG VARRAW データ型を使用してください。

VARRAW 変数の長さを指定するときは、長さフィールド用に 2 バイト含まれていることを確認してください。変数の最初の 2 バイトは整数として解釈できる必要があります。

VARRAW 変数の長さを得るには、長さフィールドを参照するのみで済みます。

LONG RAW

LONG RAW データ型を使用して、バイナリ・データまたはバイト文字列を格納します。LONG RAW 値の最大長は、2147483647 バイトつまり 2GB です。

LONG RAW データは LONG データに似ていますが、Oracle では LONG RAW データは意味がないものと解釈され、LONG RAW データをあるシステムから別のシステムに転送するときにキャラクタ・セットは変換されないという点で異なります。

UNSIGNED

UNSIGNED データ型を使用して、符号なし整数を格納します。符号なし整数は 2 バイトまたは 4 バイトの 2 進数です。ワード内のバイトの並びはシステムによって異なります。入力および出力ホスト変数に長さを指定する必要があります。出力時は、列値が浮動小数点数の場合、Oracle は小数部分を切り捨てます。

LONG VARCHAR

LONG VARCHAR データ型を使用して、可変長文字列を格納します。LONG VARCHAR 変数では、4 バイトの長さフィールドに文字列フィールドが続きます。文字列フィールドの最大長は 2147483643 (2**31-5) バイトです。VAR 文または TYPE 文で使用する LONG VARCHAR の長さを指定するときは、4 バイトの長さフィールドを含めないでください。

LONG VARRAW

LONG VARRAW データ型を使用して、可変長のバイナリ・データまたはバイト文字列を格納します。LONG VARRAW 変数では、4 バイトの長さフィールドにデータ・フィールドが

続きます。データ・フィールドの最大長は 2147483643 バイトです。LONG VARRAW を VAR 文あるいは TYPE 文で使用するために長さを指定するときには、長さフィールドの 4 バイトは含めないでください。

CHAR

CHAR データ型を使用して、固定長文字列を格納します。CHAR 値の最大長は 65535 バイトです。

入力時 Oracle は入力ホスト変数に指定されたバイト数を読み込み、後続する空白を削除しないで入力値を格納先データベースの列に格納します。

入力値がデータベース列の定義された幅より長い場合、Oracle はエラーを生成します。入力値が空白のみからなる場合、Oracle はそれを文字値として扱います。

出力時 Oracle は出力ホスト変数に指定されたバイト数を戻し、必要に応じて空白埋込みを行ってから、出力値を格納先ホスト変数に割り当てます。NULL が戻されると、Oracle はホスト変数に空白を埋め込みます。

出力値がホスト変数の宣言長より長い場合、Oracle はその値の長すぎる分を切り捨ててからホスト変数に割り当てます。標識変数が使用可能な場合、Oracle は標識変数を出力値の元の長さに設定します。

CHARZ

デフォルトでは、DBMS=V7 または V8 のとき、Oracle は Pro*C/C++ プログラムのすべての文字ホスト変数に CHARZ データ型を割り当てます。CHARZ データ型は、NULL 終了記号を付けた固定長文字列を示します。CHARZ 値の最大長は 65534 バイトです。

入力時、CHARZ データ型および STRING データ型は同じように機能します。入力値には NULL 終了記号を付ける必要があります。NULL 終了記号は、文字列の区切り記号としての役割のみを果たし、格納データの一部にはなりません。

出力時、CHARZ ホスト変数には必要に応じて空白埋込みが行われてから、NULL 終了記号が付けられます。この出力値には、データの長すぎる分が切り捨てられる場合でも、必ず NULL 終了記号が付けられます。

CHARF

CHARF データ型は、EXEC SQL TYPE 文および EXEC SQL VAR 文で使します。V7 または V8 に設定された DBMS オプションでプリコンパイルするときに、TYPE 文または VAR 文で外部データ型 CHAR を指定すると、C の型または C の変数は固定長で NULL 終了記号付きのデータ型 CHARZ と同値化されます。

しかし、これらの型に同値化するより、固定長の外部型 CHAR に同値化することが必要な場合もあります。外部型 CHARF を使用すると、C の型または C の変数は DBMS 値に関係なく常に固定長の ANSI データ型 CHAR と同値化されます。CHARF によって C の型が VARCHAR2 または CHARZ に同値化されることはありません。かわって、

CHAR_MAP=CHARF オプションを設定すると、char[n] または char として宣言されたホスト変数はすべて CHAR 文字列に同値化されます。

ホスト変数

ホスト変数は、ホスト・プログラムと Oracle 間の通信において重要な役割を果たします。通常、プリコンパイラ・プログラムがホスト変数から Oracle にデータを入力し、Oracle がプログラム内のホスト変数にデータを出力します。Oracle は入力データをデータベース列に格納し、出力データをプログラムのホスト変数に格納します。

ホスト変数には、スカラー型として解決される任意の C の式を指定してもかまいません。しかし、ホスト変数は *lvalue* です。ほとんどのホスト変数のホスト配列もサポートされています。詳細は、4-43 ページの「[ポインタ変数](#)」を参照してください。

ホスト変数の宣言

ホスト変数は、Oracle プログラム・インタフェースがサポートする C データ型を指定して、C の規則に従って宣言します。この C のデータ型は、ソースまたはターゲットのデータベース列のデータ型と互換性が必要です。

MODE=ORACLE の場合、特別な宣言文でホスト変数を宣言する必要はありません。ただし宣言文が ANSI SQL 標準の一部である場合に宣言文を使用しないと、FIPS フラガーがそのことを警告します。CODE=CPP (C++ コードをコンパイル中)、PARSE=NONE または PARSE=PARTIAL である場合、宣言文を使用する必要があります。

[表 4-4](#) に C のデータ型とホスト変数を宣言するときに使用できる疑似型を示します。これらのデータ型のみがホスト変数に使用できます。

表 4-4 ホスト変数の C のデータ型

C のデータ型または疑似型	説明
char	単一の文字
char[n]	n 文字配列 (文字列)
int	整数
short	小さい整数
long	大きい整数
float	浮動小数点数 (通常は単精度)
double	浮動小数点数 (常に倍精度)
VARCHAR[n]	可変長文字列

表 4-5 に互換性のある Oracle の内部データ型を示します。

表 4-5 C と Oracle のデータ型の互換性

内部型	C 型	説明
VARCHAR2(Y) (注意 1)	char	単一の文字
CHAR(X) (注意 1)	char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの変長文字配列
	int	整数
	short	小さい整数
	long	大きい整数
	float	浮動小数点数
	double	倍精度浮動小数点 数値
NUMBER	int	整数
NUMBER(P,S) (注意 2)	short	小さい整数
	int	整数
	long	大きい整数
	float	浮動小数点数
	double	倍精度浮動小数点 数値
	char	単一の文字
	char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの変長文字配列
DATE	char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの変長文字配列
LONG	char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの変長文字配列
RAW(X) (注意 1)	unsigned char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの変長文字配列
LONG RAW	unsigned char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの変長文字配列

表 4-5 C と Oracle のデータ型の互換性

内部型	C 型	説明
ROWID	unsigned char[n]	n バイトの文字配列
	VARCHAR[n]	n バイトの可変長文字配列

注意：

1. X の範囲は 1 から 2000 までです。1 がデフォルト値です。Y の範囲は 1 から 4000 までです。
2. P の範囲は 1 から 38 までです。S の範囲は -84 から 127 までです。

Oracle データ型の詳細は 4-2 ページの「[Oracle のデータ型](#)」を参照してください。

単純な C の型の 1 次元配列は、ホスト変数としての役割も果たします。char[n] および VARCHAR[n] の場合、n には配列内にある文字列の数ではなく、文字列の最大長を指定します。2 次元配列は、char[m][n] および VARCHAR[m][n] の場合にのみ指定できます。m には配列内にある文字列の数を指定し、n には文字列の最大長を指定します。

単純な C の型へのポインタがサポートされています。char[n] および VARCHAR[n] 変数へのポインタは、char または VARCHAR（長さの指定なし）へのポインタとして宣言する必要があります。ただし、ポインタの配列はサポートされていません。

記憶クラス指定子

Pro*C/C++ では、ホスト変数の宣言時に **auto**、**extern** および **static** 記憶クラス指定子を使用することができます。ただし、プリコンパイラはホスト変数の前にアンパサンド（&）を置くことでアドレスを取得するため、**register** 記憶クラス指定子を使用してホスト変数を保存することはできません。C の規則に従えば、**auto** 記憶クラス指定子を使用することができるのは、ブロック内のみです。

次の例に示すように、Pro*C/C++ プリコンパイラでは **extern char[n]** ホスト変数を宣言するとき、最大長の指定の有無は任意です。これにより ANSI C 規格準拠となっています。

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

ただし、いずれの場合でも最大長は指定してください。前述の例で、あるプリコンパイル・ユニットで宣言された出力ホスト変数 *msg* が他のプリコンパイル・ユニットで定義された場合、プリコンパイラにはその最大長を知る方法がありません。2 番目のプリコンパイル・ユニットの *msg* に十分な記憶域を割り当てておかないと、メモリーが壊れることがあります。（通常、「十分な」というのはホスト変数に SELECT されたり FETCH されたりする可能性のある最長の列値のバイト数に、NULL 終了記号がつく可能性があるので 1 バイトを加えた値です。）

extern char[] ホスト変数の最大長を指定し忘れると、プリコンパイラは警告メッセージを出します。また、ホスト変数には CHARACTER 列値を格納するものと見なされますが、こ

の値の長さは 255 文字以内にしてください。したがって、長さ 255 文字を超える VARCHAR2 または LONG 列値をホスト変数に SELECT または FETCH する場合は、最大長を指定する必要があります。

型修飾子

ホスト変数を宣言する場合は、**const** および **volatile** 型修飾子も使用できます。

const ホスト変数は定数値を持つ必要があります。つまり、プログラム中ではその初期値を変えられません。**volatile** ホスト変数は値をプログラムからはわからない方法で（たとえばシステムに接続された装置によって）変更されることがあります。

ホスト変数の参照

ホスト変数は、SQL DML 文で使用します。ホスト変数は SQL 文の中では先頭にコロン (:) を付ける必要がありますが、C の文の中ではコロンを先頭に付けないでください。次に例を示します。

```
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;
```

混乱する場合もありますが、次の例のように、ホスト変数に Oracle の表または列と同じ名前を付けることもできます。

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

制限

ホスト変数名は C 識別子なので、宣言および参照する際は、大文字 / 小文字の使用を一致させる必要があります。ホスト変数を列または表、SQL 文その他の Oracle オブジェクトで置換することはできません。また Oracle 予約語も使用できません。付録 B「予約語、キーワードおよび名前領域」を参照してください。

ホスト変数は、プログラムのアドレスへ設定してください。このため、関数コールおよび数式をホスト変数に指定することはできません。次のコードは無効です。

```
#define MAX_EMP_NUM    9000
...
int get_dept();
...
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
(:MAX_EMP_NUM + 10, 'CHEN', :get_dept());
```

標識変数

個々のホスト変数は任意指定の標識変数と対応付けることができます。標識変数は、2 バイトの整数として定義する必要があります。また、SQL 文の中では、標識変数の前にコロンを付け、(キーワード INDICATOR を使用しないかぎり) ホスト変数の直後に指定する必要があります。宣言文を使用する場合は、宣言文の中にも標識変数を指定する必要があります。

注意: これはリレーショナル列に適用されるもので、オブジェクト型には適用されません。これについては第 17 章「オブジェクト」で説明しています。

キーワード INDICATOR の使用

判読しやすくするため、それぞれの標識変数の前に INDICATOR というオプションのキーワードを置くこともできます。その場合も、標識変数の前にはコロンを付ける必要があります。正しい構文は次のとおりです。

```
:host_variable INDICATOR :indicator_variable
```

これと同じ意味を、次のように表すこともできます。

```
:host_variable:indicator_variable
```

ホスト・プログラムでは、両方の形式の式を使用できます。

考えられる標識の値とその意味は、次のとおりです。

0	操作は成功しました。
-1	NULL が戻されたか、挿入されたか、更新されました。
-2	"long" 型の文字ホスト変数の出力は、長すぎる分が切り捨てられましたが、元の列の長さはわかりません。
>0	SELECT または FETCH の結果文字ホスト変数に入ったデータは、長すぎる分が切り捨てられました。この場合、そのホスト変数が NLS マルチバイト変数であれば、標識の値は、元の列の長さを文字数で表したものになります。そのホスト変数が NLS 変数でないと、標識の長さは、元の列の長さをバイト数で表したものになります。

例

通常、標識変数は、入力ホスト変数への NULL の割当てと、出力ホスト変数に入っている NULL 値または切捨て値の検出に使用します。次の例では、3 つのホスト変数と 1 つの標識変数を宣言したあと、SELECT 文を使用して、ホスト変数 *emp_number* の値に一致する従業員番号をデータベースで探します。一致する行が見つかり、Oracle は出力ホスト変数 *salary* および *commission* をその行の SAL 列および COMM 列の値に設定し、リターン・コードを標識変数 *ind_comm* に格納します。それに続く文では、*ind_comm* を使用して、その後の処置を選択しています。

```
EXEC SQL BEGIN DECLARE SECTION;
    int    emp_number;
    float  salary, commission;
    short ind_comm; /* indicator variable */
EXEC SQL END DECLARE SECTION;
    char temp[16];
    float pay;      /* not used in a SQL statement */
...
printf("Employee number? ");
gets(temp);
emp_number = atof(temp);
EXEC SQL SELECT SAL, COMM
    INTO :salary, :commission:ind_comm
FROM EMP
    WHERE EMPNO = :emp_number;
if(ind_comm == -1) /* commission is null */
    pay = salary;
else
    pay = salary + commission;
```

標識変数の使用に関する詳細は 6-3 ページの「[標識変数の使用](#)」を参照してください。

ガイドライン

標識変数の宣言および参照では、次のガイドラインに従ってください。標識変数は、必ず次のようにします。

- 2 バイトの整数として、明示的に（宣言文があればそこに）宣言します。
- SQL 文の中では、コロン (:) を前に付けます。
- SQL 文および PL/SQL ブロックの中では、そのホスト変数の直後に指定します。（キーワード INDICATOR を前置する場合は除きます。）

標識変数は、次のようにしないでください。

- ホスト言語文の中で、コロンを前に付けないでください。
- ホスト言語文の中で、そのホスト変数の直後に指定しないでください。
- Oracle の予約語にしないでください。

Oracle 制限事項

DBMS=V7 または DBMS=V8 の場合は、標識変数に対応付けられていないホスト変数に NULL を SELECT または FETCH すると、Oracle は次のエラー・メッセージを発行します。

ORA-01405: 取り出した列の値が NULL です。

MODE=ORACLE および DBMS=V7 または DBMS=V8 と指定してプリコンパイルする場合は、UNSAFE_NULL=YES と指定して ORA-01405 メッセージを使用禁止にできます。詳細は、10-39 ページの「[UNSAFE_NULL](#)」を参照してください。

VARCHAR 変数

VARCHAR 疑似型は、可変長の文字列を宣言するのに使用できます。プログラムで扱う文字列が VARCHAR2 列または LONG 列からの出力、またはその列への入力である場合、標準の C 文字列のかわりに VARCHAR ホスト変数を使用した方が便利なこともあります。データ型名 VARCHAR は、すべて大文字でも、すべて小文字でも構いませんが、大文字と小文字を混在させないでください。このマニュアルでは、VARCHAR が C 固有のデータ型と異なっていることを強調するため、大文字を使用しています。

VARCHAR 変数の宣言

VARCHAR は、C の拡張型または宣言済の**構造体**と考えてください。たとえばプリコンパイラは VARCHAR 宣言を拡張します。

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

プリコンパイラはこれを配列メンバーおよび長さメンバーを持つ次の**構造体**に展開します。

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

VARCHAR 変数の利点は、SELECT または FETCH の後に VARCHAR 構造体の長さメンバーを明示的に参照できることです。Oracle は、選択された文字列の長さを長さメンバーに配置します。それからこのメンバーを、NULL（つまり '\0'）終了記号を加えることなどに使用できます。

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

または次の例のように、長さは *strncpy* 文または *printf* 文でも指定できます。

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

VARCHAR 変数の最大長は、その宣言中で指定します。長さは 1 から 65,533 の範囲にする必要があります。たとえば、次の宣言は長さが指定されていないため、無効です。

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

長さメンバーは配列メンバーに格納される値の現行の長さを保持します。

次の例のように、1 つの行に複数の VARCHAR も宣言できます。

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

VARCHAR の長さ指定子としては、**#define** による定義済マクロか、プリコンパイル時に整数で解決できる任意の複合式を使用できます。

VARCHAR データ型へのポインタも宣言できます。詳細は 5-7 ページの「[VARCHAR 変数およびポインタ](#)」の項を参照してください。

注意: 次のような typedef 文は使用しないでください。

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

前述のような文を使用すると、C コンパイル・エラーが発生します。

VARCHAR 変数の参照

SQL 文では、次の例が示すように、コロンを接頭辞として付けた**構造体名**を使用して、VARCHAR 変数を参照します。

```
...
int      part_number;
VARCHAR  part_desc[40];
...
main()
{
    ...
    EXEC SQL SELECT pdesc INTO :part_desc
           FROM parts
           WHERE pnum = :part_number;
    ...
}
```

問合せが実行された後、データベースから取り出され、*part_desc.arr* に格納された文字列の実際の長さが *part_desc.len* に保持されます。

C の文では、次の例で示すようにコンポーネント名を使用して VARCHAR 変数を参照します。

```
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);
```

VARCHAR 変数に NULL を戻す

Oracle は、VARCHAR 出力ホスト変数の長さのコンポーネントを自動的に設定します。SELECT または FETCH で VARCHAR 変数に NULL 値を入れた場合、サーバーは長さメンバーも配列メンバーも変更しません。

注意： NULL を VARCHAR ホスト変数に選択したとき、対応する標識変数がないと、実行時に ORA-01405 エラーが発生します。これを避けるには、すべてのホスト変数に対して標識変数を記述します。（一時修正としては、UNSAFE_NULL=YES プリコンパイラ・オプションを使用します。詳細は 10-15 ページの「**DBMS**」を参照してください。）

VARCHAR 変数を使用した NULL の挿入

VARCHAR 変数の長さをゼロに設定してから UPDATE 文または INSERT 文を実行すると、列値は NULL に設定されます。列に NOT NULL 制約がある場合、Oracle はエラーを戻します。

VARCHAR 変数を関数に渡す

VARCHAR は構造体であり、ほとんどの C コンパイラでは構造体を値によって関数に渡すこと、および関数からコピーすることによって構造体を戻すことが許可されています。ただし、Pro*C/C++ では参照によって VARCHAR を関数に渡す必要があります。次の例で、VARCHAR 変数を関数に渡す正しい方法を示します。

```
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);
    ...
}
```

VARCHAR 配列コンポーネントの長さを調べる方法

プリコンパイラが VARCHAR 宣言を処理すると、生成される構造体中の配列要素の実際の長さは宣言された長さよりも長くなることがあります。たとえば、Sun Solaris システムで次のような Pro*C/C++ 宣言があるとしたします。

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

この宣言は、プリコンパイラによって次のように展開されます。

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

ただし、プリコンパイラまたはこのシステム上の C コンパイラは、配列コンポーネントの長さを 14 バイトまで埋め込みます。この配列要求は構造体全体の長さを 16 バイトに補充します。埋込み処理された配列が 14 バイト、長さが 2 バイトです。

`SQLVarcharGetLength()`（非スレッド `sqlvcp()` と置換する）関数（SQLLIB ランタイム・ライブラリの一部）によって、配列値の実際の長さ（埋め込まれていることもある）が戻されます。

`SQLVarcharGetLength()` 関数には、VARCHAR ホスト変数または VARCHAR ポインタ・ホスト変数のデータの長さを渡します。すると、`SQLVarcharGetLength()` は、VARCHAR の配列コンポーネントの合計の長さを戻します。合計の長さには、使用している C コンパイラによって追加された可能性のある埋込みが含まれます。

`SQLVarcharGetLength()` の構文は、次のとおりです。

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

単一スレッド・アプリケーションの場合は、`sqlvcp()` を使用してください。VARCHAR の長さを `datlen` パラメータに配置してから、`sqlvcp()` をコールします。関数が戻ると、`totlen` パラメータには配列要素の合計の長さが設定されています。どちらのパラメータも未割当ての `unsigned long` へのポインタのため、参照形式で渡す必要があります。

この機能および SQLLIB の他の機能の詳細は、5-49 ページの「[SQLLIB パブリック関数の新しい名前](#)」を参照してください。

サンプル・プログラム : `sqlvcp()` の使用

次のサンプル・プログラムでは、Pro*C/C++ アプリケーションでの関数の使用方法を示します。このサンプル・プログラムでは、`sqlgls()` 関数も使用しています。この関数の詳細は、[第9章「ランタイム・エラーの処理」](#)を参照してください。このサンプルでは、まず VARCHAR ポインタを宣言し、それから `sqlvcp()` 関数を使用して VARCHAR バッファに必要なサイズを決定します。プログラムは EMP 表から従業員名を FETCH して表示します。最後に、サンプルは `sqlgls()` 関数を使用して SQL 文およびその関数コード、長さの属性を表示します。このプログラムは、`demo` ディレクトリの `sqlvcp.pc` として、オンラインで使用可能です。

```
/*
 * 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.
 *
 * This program also demonstrates the use of the sqlgls()
 * function, to get the text of the last SQL statement executed.
 * sqlgls() is described in the "Error Handling" chapter of
 * The Programmer's Guide to the Oracle Pro*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++)
{
    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++)
{
    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("%.5s\n", msg_len, err_msg);

EXEC SQL ROLLBACK RELEASE;
exit(1);
}
```

カーソル変数

Pro*C/C++ プログラム内では、問合せ用にカーソル変数を使用できます。カーソル変数とは、Oracle（リリース 7.2 以降）サーバー上で PL/SQL を使用して定義しオープンする必要があるカーソルのハンドルです。カーソル変数に関する完全な情報は『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

カーソル変数の利点は、次のとおりです。

- メンテナンスしやすいこと：問合せがカーソル変数をオープンするストアード・プロシージャに集中されます。カーソルを変更する必要がある場合は、ストアード・プロシージャを変更するのみで済みます。各アプリケーションを変更する必要はありません。

- セキュリティに都合がよいこと：アプリケーションのユーザーは Pro*C/C++ アプリケーションがサーバーに接続するときに使用されるユーザー名です。ユーザーには、カーソルをオープンするストアド・プロシージャに対する実行権限が必要です。また、問合せに使用される表に対する読み込み権限は必要ありません。このセキュリティ機能を使用して、表内の列や、他のストアド・プロシージャへのアクセスを制限できます。

カーソル変数の宣言

Pro*C/C++ 疑似型 SQL_CURSOR を使用して、Pro*C/C++ プログラム内にカーソル変数を宣言します。たとえば、次のとおりです。

```
EXEC SQL BEGIN DECLARE SECTION;
    sql_cursor      emp_cursor;          /* a cursor variable */
    SQL_CURSOR      dept_cursor;        /* another cursor variable */
    sql_cursor      *ecp;               /* a pointer to a cursor variable */
    ...
EXEC SQL END DECLARE SECTION;
ecp = &emp_cursor;                     /* assign a value to the pointer */
```

カーソル変数を宣言するとき、型指定にはすべて大文字の SQL_CURSOR またはすべて小文字の sql_cursor のどちらかを使用できますが、小文字と大文字の組合せは使用できません。

カーソル変数は、Pro*C/C++ プログラム内の他のホスト変数と同様です。カーソル変数には範囲があり、C の有効範囲規則に従っています。カーソル変数は、他の関数にパラメータとして渡すことができ、カーソル変数を宣言しているソース・ファイルの外部にある関数にも渡すことができます。また、カーソル変数を戻す関数のみでなくカーソル変数へのポインタを戻す関数も定義できます。

注意：SQL_CURSOR は C の構造体として Pro*C/C++ が生成するコードにインプリメントされます。そこで、いつでも、ポインタを使用してカーソル変数を別の関数に渡したり、関数からカーソル変数へポインタを戻すことができます。ただし、SQL_CURSOR を値によって渡したり戻したりできるのは、ご使用の C コンパイラがそのような操作をサポートしている場合に限ります。

カーソル変数の割当て

カーソル変数をオープンするか、カーソル変数を使用して FETCH する前に、カーソルを割り当てる必要があります。それには、新しいプリコンパイラ・コマンドである ALLOCATE を使用します。たとえば前述の例で宣言した SQL_CURSOR *emp_cursor* を割り当てるには次の文を書きます。

```
EXEC SQL ALLOCATE :emp_cursor;
```

カーソルの割当てには、プリコンパイル時も実行時もサーバーのコールは必要ありません。ALLOCATE 文に誤り（たとえば、未宣言のホスト変数）があると、Pro*C/C++ はプリコンパイル時エラーを出します。カーソル変数を割り当てると、ヒープ・メモリーが使用されます。このため、プログラム・ループでカーソル変数を解放します。（4-28 ページの「[カーソル変数のクローズと解放](#)」を参照してください。）カーソル変数に割り当てられるメモリーは、カーソルがクローズされるときには解放されず、明示的な CLOSE が実行されるか、または接続がクローズされるときにのみ解放されます。

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

カーソル変数のオープン

カーソル変数は Oracle8 Server 上でオープンする必要があります。埋込み SQL OPEN コマンドを使用してもカーソル変数はオープンできません。カーソル変数をオープンする方法の 1 つは、カーソルをオープン（し、それを同じ文の中に定義）する PL/SQL のストアード・プロシージャをコールすることです。もう 1 つの方法は、Pro*C/C++ プログラム内で、無名 PL/SQL ブロックを使用してカーソル変数をオープンし定義することです。

たとえば、次の PL/SQL パッケージがデータベースに格納されている場合を考えます。

```
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;
```

このパッケージが格納された後、ストアード・プロシージャ *open_emp_cur* を Pro*C/C++ プログラムからコールすることによってカーソル *curs* をオープンし、プログラム内のカーソルから FETCH できます。たとえば、次のとおりです。

```
...
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);
}
...

```

Pro*C/C++ プログラム内で無名 PL/SQL ブロックの 1 つを使用してカーソルをオープンするには、無名ブロック内にカーソルを定義します。たとえば、次のとおりです。

```

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;
...

```

前述の各例では、PL/SQL を使用してカーソル変数をオープンしています。埋込み SQL 文で CURSOR 句を指定してもカーソル変数をオープンできます。

```

...
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);

```

前述の文では、`emp_cursor` カーソル変数が最も外側の `select` の最初の列にバインドされています。最初の列は、それ自体が問合せですが、`CURSOR(...)` 変換句が指定されているため、`sql_cursor` ホスト変数と互換性のある形式で表されています。

`CURSOR` 句を含む問合せを指定する場合には、必ず `SELECT_ERROR` オプションを `NO` に設定してください。これは、親カーソルを取り消せないようにするのみでなく、プログラムをエラーなく稼働させます。

スタンドアロン・ストアド・プロシージャでのオープン

前述の例では、参照カーソルがパッケージ内で定義され、カーソルはそのパッケージ内のプロシージャ内でオープンされました。しかし、参照カーソルは、カーソルをオープンするプロシージャと同じパッケージ内に定義する必要はありません。

スタンドアロン・ストアド・プロシージャ内でカーソルをオープンする必要があるときは、別のパッケージ内にカーソルを定義します。そして、カーソルをオープンするスタンドアロン・ストアド・プロシージャ内で、そのパッケージを参照します。例を示します。

```
PACKAGE dummy IS
    TYPE EmpName IS RECORD (name VARCHAR2(10));
    TYPE emp_cursor_type IS REF CURSOR RETURN EmpName;
END;
-- and then define a stand-alone 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;
```

戻り型

PL/SQL ストアド・プロシージャ内に参照カーソルを定義するときは、カーソルが戻す値の型を宣言する必要があります。参照カーソル型とその戻り型の完全な情報は『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

カーソル変数のクローズと解放

カーソル変数をクローズするには、CLOSE コマンドを使用します。たとえば、前述の例でオープンした *emp_cursor* カーソル変数をクローズするには、埋込み SQL 文を使用します。

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

なお、カーソル変数はホスト変数であるため、コロンを前に付ける必要があることに注意してください。

ALLOCATE で割り当てたカーソル変数は再利用できます。アプリケーションで必要な回数のみオープン、FETCH および CLOSE できます。しかし、サーバーとの接続を一度切断してから、再び接続する場合には、カーソル変数を ALLOCATE で再び割り当てる必要があります。

カーソルは、FREE 埋込み SQL 文によって割り当てられます。たとえば、次のとおりです。

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

カーソルがオープンしている場合、カーソルはクローズされ、割り当てられたメモリーは解放されます。

OCI でのカーソル変数の使用（リリース 7 のみ）

Pro*C/C++ カーソル変数は、OCI 関数と共有できます。共有するには、SQLLIB の変換関数 `SQLCDAFromResultSetCursor()`（以前の `sqlcdat()`）および `SQLCDAToResultSetCursor`（以前の `sqlcurt()`）を使用する必要があります。これらの関数を使用して、OCI カーソル・データ領域と Pro*C/C++ カーソル変数との間の変換を行います。

`SQLCDAFromResultSetCursor()` 関数は、割当て済のカーソル変数を OCI カーソル・データ領域に変換します。構文は次のとおりです。

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

各パラメータは次のとおりです。

context	SQLLIB 実行時コンテキストへのポインタ
cda	変換先の OCI カーソル・データ領域へのポインタ
cur	変換元の Pro*C/C++ カーソル変数へのポインタ
retval	エラーがなければ 0、そうでなければ SQLLIB (SQL) のエラー番号

注意：エラーの場合には、CDA の V2 と `rc` リターン・コード・フィールドもエラー・コードを受け取ります。CDA の行の処理済み件数フィールドは設定されません。

非スレッドまたはデフォルト・コンテキスト・アプリケーションでは、定義した定数 `SQL_SINGLE_RCTX` をコンテキストとして渡してください。

注意：コンテキストの説明は 5-49 ページの「[SQLLIB パブリック関数の新しい名前](#)」を参照してください。

`SQLCDAToResultSetCursor()` 関数は、OCI カーソル・データ領域を Pro*C/C++ カーソル変数に変換します。構文は次のとおりです。

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

各パラメータは次のとおりです。

context	SQLLIB 実行時コンテキストへのポインタ
cur	変換先の Pro*C/C++ カーソル変数へのポインタ
cda	変換元の OCI カーソル・データ領域へのポインタ
retval	エラーがないときは 0、エラーがあるときはエラー・コード

注意：SQLCA 構造体はこのルーチンでは更新されません。SQLCA のコンポーネントの設定が行われるのは、変換されたカーソルを使用してデータベース操作が実行された後のみです。

非スレッド・アプリケーションでは、定義した定数 `SQL_SINGLE_RCTX` をコンテキストとして渡してください。

これらの関数に対する ANSI と K&R のプロトタイプは `sql2oci.h` ヘッダー・ファイルに用意されています。これらの関数をコールする前に、`cda` および `cur` のメモリーを割り当てする必要があります。

SQLLIB パブリック関数の詳細は 5-49 ページの「[SQLLIB パブリック関数の新しい名前](#)」の表を参照してください。

制限

カーソル変数の使用には、次の制限が適用されます。

- Pro*C/C++ と OCI V7 で同じカーソル変数を使用する場合、接続後ただちに `SQLLDAGetCurrent()` か `SQLLDAGetName()` のいずれかを使用する必要があります。
- カーソル変数は OCI リリース 8 で対応したものには変換できません。
- カーソル変数は動的 SQL では使用できません。
- カーソル変数は、`ALLOCATE`、`FETCH`、`FREE` および `CLOSE` コマンドのみで使用できます。
- `DECLARE CURSOR` コマンドは、カーソル変数には適用されません。
- `CLOSE` でクローズしたカーソル変数からの `FETCH` はできません。
- `ALLOCATE` で割り当てられていないカーソル変数からの `FETCH` はできません。
- `MODE=ANSI` を指定してプリコンパイルする場合、すでにクローズ済のカーソル変数をクローズするとエラーになります。
- `AT` 句は、カーソル変数を参照する `ALLOCATE` コマンド、`FETCH` コマンド、`CLOSE` コマンドには使用できません。
- カーソル変数は、データベースの列に格納できません。
- パッケージ指定の中では、カーソル変数そのものは宣言できません。パッケージ指定の中で宣言できるのは、カーソル変数の型のみです。
- カーソル変数は、PL/SQL レコードのコンポーネントにはできません。

サンプル・プログラム

次のサンプル・プログラム（PL/SQL スクリプトと Pro*C/C++ プログラム）はカーソル変数の使用方法を示します。これらのソースは *demo* ディレクトリに入っており、オンラインで利用できます。アプリケーションの別のバージョンも参照してください。デモ・ディレクトリの *cv_demo.pc* にあります。

cv_demo.sql

```
-- PL/SQL source for a package that declares and
-- opens a ref cursor
CONNECT SCOTT/TIGER;
CREATE OR REPLACE PACKAGE emp_demo_pkg as
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    PROCEDURE open_cur(curs IN OUT emp_cur_type, dno IN NUMBER);
END emp_demo_pkg;

CREATE OR REPLACE PACKAGE BODY emp_demo_pkg AS
    PROCEDURE open_cur(curs IN OUT emp_cur_type, dno IN NUMBER) IS
    BEGIN
        OPEN curs FOR SELECT *
            FROM emp WHERE deptno = dno
            ORDER BY ename ASC;
    END;
END emp_demo_pkg;
```

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);  
}
```

CONTEXT 変数

「ランタイム・コンテキスト」（通常は単にコンテキストと呼ばれる）は、クライアント・メモリーの領域へのハンドルで、このクライアント・メモリーには 0 またはそれ以上の接続、0 またはそれ以上のカーソル、それらのインライン・オプション（MODE、HOLD_CURSOR、RELEASE_CURSOR、SELECT_ERROR など）および他の状態を示す情報が含まれます。

コンテキスト・ホスト変数を定義するには、擬似型の *sql_context* を使用します。たとえば、次のとおりです。

```
sql_context my_context ;
```

CONTEXT ALLOCATE プリコンパイラ宣言文を使用して、コンテキスト用のメモリーの割当ておよび初期化を行います。

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

この context は、コンテキストへのハンドルであるホスト変数です。たとえば、次のとおりです。

```
EXEC SQL CONTEXT ALOOCATE :my_context ;
```

CONTEXT USE プリコンパイラ宣言文宣言文を使用して、プログラム論理の流れではなく、ソース・ファイルのその点から埋込み SQL 文（CONNECT、INSERT、DECLARE CURSOR など）が使用するコンテキストを定義します。このコンテキストは、別の CONTEXT USE 文に遭遇するまで使用されます。構文は次のとおりです。

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

キーワード DEFAULT によって、以降に実行されるすべての埋込み SQL 文で使用されるデフォルト（またはグローバル）コンテキストが指定されます。このコンテキストは、別の CONTEXT USE 宣言文に遭遇するまで使用されます。たとえば次のようになります。

```
EXEC SQL CONTEXT USE :my_context ;
```

コンテキスト変数 my_context が定義され、割り当てられていない場合は、エラーが戻されます。

CONTEXT FREE 文を使用すると、コンテキストによって使用されたメモリーが必要でなくなった場合、メモリーを解放できます。

```
EXEC SQL CONTEXT FREE :context ;
```

例を示します。

```
EXEC SQL CONTEXT FREE :my_context ;
```

次の例は、ユーザー定義のコンテキストと同じアプリケーションにおけるデフォルト・コンテキストの使用法を示します。

CONTEXT USE の例

```
#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');
    ...
}
```

汎用 ROWID

データベース・サーバーでは、「ヒープ表」および「索引構成表」の2種類の表編成が使用されています。

デフォルトではヒープ表が使用されます。これは、Oracle8 以前のすべての表で使用されていた表編成です。物理行アドレス (ROWID) は、ヒープ表の行を識別するために使用される恒久的なプロパティです。物理 ROWID の外部文字形式は、64 をベースにする 18 バイト文字列です。

索引構成表には、恒久的な識別子としての物理行アドレスはありません。これらの表には、論理 ROWID が定義されています。索引構成表から SELECT ROWID ... 文を使用する場合、ROWID は表の主キー、制御情報、およびオプションの物理的「不確定要素」を含む不明瞭な構造になります。この ROWID を "WHERE ROWID = ..." などの句を含む SQL 文で使用し、表から値を取得することができます。

Oracle 8.1 リリースから汎用 ROWID が導入されています。汎用 ROWID は、物理 ROWID と論理 ROWID の両方で使用できます。表構成の変更はアプリケーションに何も影響を与えないため、汎用 ROWID を使用すると、ヒープ表または索引構成表のデータにアクセスできます。ROWID 用に使用される列データ型は、UROWID (length) です。この length はオプションです。

すべての新しいアプリケーションでこれらの関数を使用します。

汎用 ROWID の詳細は『Oracle8i 概要』を参照してください。

汎用 ROWID 変数の使用方法は、次のとおりです。

- OCIRowid へのタイプ・ポインタとして宣言します。
- 汎用 ROWID 変数用のメモリーを割り当てます。
- 汎用 ROWID をホスト・バインド変数として使用します。
- 終了後メモリーを解放します。

たとえば、次のとおりです。

```
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_urowid ;
...
```

19 (18 バイトと NULL 終了記号) から 4001 までの幅を持つ文字列ホスト変数を汎用 ROWID のホスト・バインド変数として使用することもできます。文字ベースの汎用 ROWID はヒープ表でもサポートされています。ただし下位互換性しかありません。汎用 ROWID は可変長であるため、切り捨てられる場合があります。

文字変数は次のように使用します。

```
/* 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 ;
...
```

汎用 ROWID を使用した位置決定済み更新の例は、6-19 ページの「[位置決定済み更新](#)」を参照してください。

SQLRowidGet()

SQLLIB 関数、SQLRowidGet() を使用すると、最後に挿入、更新または選択された行の汎用 ROWID へのポインタを取得することができます。関数プロトタイプおよびその引数は、次のようになります。

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

rctx (IN)

これは、ランタイム・コンテキストへのポインタです。デフォルト・コンテキストまたは非スレッドの場合には、SQL_SINGLE_RCTX を渡します。

urid (OUT)

これは、汎用 ROWID ポインタへのポインタです。通常の実行が終了すると、有効な ROWID をポイントします。エラーが発生した場合、NULL が戻されます。

注意: 汎用 ROWID ポインタは、SQLRowidGet() をコールするために前もって割り当てておく必要があります。汎用 ROWID では後で FREE を使用してください。

ホスト構造体

C 構造体を使用すると、ホスト変数を組み込むことができます。SELECT 文または FETCH 文の INTO 句に、および INSERT 文の VALUES リストに、ホスト変数が入っている構造体を参照します。ホスト構造体のすべてのコンポーネントは表 4-4 で定義されている、正当な Pro*C/C++ のホスト変数である必要があります。

構造体がホスト変数として使用されると、構造体の名前のみが SQL 文で使用されます。しかし、構造体の各メンバーは Oracle にデータを送信したり、問合せで Oracle からデータを受信します。次の例では、EMP 表に従業員を 1 人追加する際に使用されるホスト構造体を示します。

```
typedef struct
{
    char emp_name[11]; /* one greater than column length */
    int emp_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);
```

メンバーが構造体の中で宣言される順序は、SQL 文中の対応する列の順序と一致する必要があります。また、INSERT 文で列のリストが省略されている場合は、データベース表の列の順序と一致している必要があります。

たとえば、ホスト構造体を次のように使用すると無効になり、ランタイム・エラーが発生します。

```
struct
{
    int empno;
    float salary;           /* struct components in wrong order */
    char emp_name[10];
} emp_record;

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

前述の例は、構造体のコンポーネントが選択リスト中の関連する列とは異なる順序で宣言されているため、無効です。SELECT 文の正しい形は次のとおりです。

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

ホスト構造体と配列

「配列」とは「要素」と呼ばれる関連データ項目の集合で、1つの変数名に対応します。ホスト変数として宣言した配列は、ホスト配列と呼ばれます。同様に、配列として宣言した標識変数は、「標識配列」と呼ばれます。標識配列は任意のホスト配列に対応付けることができます。

ホスト配列によって、単一の SQL 文でデータ項目の集合全体を操作でき、その結果パフォーマンスを向上させることができます。2、3の例外を除けば、スカラーのホスト変数が許可されるところならどこにでもホスト配列を使用できます。さらに、どのホスト配列でも標識との対応付けができます。

ホスト配列の完全な説明については、[第8章「ホスト配列」](#)を参照してください。

ホスト配列は、ホスト構造体のコンポーネントとして使用できます。次の例では、配列を含む構造体を使用して、EMP 表に3つの新しい項目を INSERT します。

```
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);

...
```

PL/SQL レコード

C 構造体は、PL/SQL レコードにバインドできません。

ネストした構造体と共用体

ホスト構造体はネストできません。次の例は無効です。

```
struct
{
    int emp_number;
    struct
    {
        float salary;
        float commission;
    }
}
```

```
        } sal_info;          /* INVALID */
        int dept_number;
    } emp_record;
    ...
EXEC SQL SELECT empno, sal, comm, deptno
        INTO :emp_record
        FROM emp;
```

また、C **共用体**をホスト構造体として使用することも、ホスト構造体として使用される構造体に**共用体**をネストすることもできません。

ホスト標識構造体

標識変数を使用する必要があるのに、ホスト変数がホスト構造体に入っている場合は、ホスト構造体中の各ホスト変数ごとの標識変数を含む 2 番目の構造体を設定します。

たとえば、ホスト構造体 *student_record* を次のように宣言する場合を考えます。

```
struct
{
    char s_name[32];
    int s_id;
    char grad_date[9];
} student_record;
```

次のような問合せで、このホスト構造体を使用するとします。

```
EXEC SQL SELECT student_name, student_idno, graduation_date
        INTO :student_record
        FROM college_enrollment
        WHERE student_idno = 7200;
```

次に、卒業日が NULL であるかどうかを知る必要があります。それから個別ホスト標識構造体を宣言する必要があります。これは、次のように宣言します。

```
struct
{
    short s_name_ind; /* indicator variables must be shorts */
    short s_id_ind;
    short grad_date_ind;
} student_record_ind;
```

SQL 文中の標識構造体は、ホスト標識変数を参照するのと同じ方法で参照してください。

```
EXEC SQL SELECT student_name, student_idno, graduation_date
        INTO :student_record INDICATOR :student_record_ind
        FROM college_enrollment
        WHERE student_idno = 7200;
```


問合せが完了すると、選択された各コンポーネントの NULL/NOT NULL ステータスはホスト標識構造体で使用可能になります。

注意： このマニュアルでは伝統に従ってホスト変数もしくは構造体の名前に *_ind* を付加して標識変数と標識構造体の名前にしています。ただし、標識変数の名前はまったく任意のもので。異なる規則を用いても、規則をまったく使用しなくてもかまいません。

サンプル・プログラム：カーソルとホスト構造体

この項のサンプル・プログラムでは、明示的なカーソルを使用し、データを選択してホスト構造体に格納する問合せを示しています。このプログラムは、*demo* ディレクトリのファイル *sample2.pc* 内で使用可能です。

```
/*
 * sample2.pc
 *
 * This program connects to ORACLE, declares and opens a cursor,
 * 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\nThe company's salespeople are--\n\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);
}
```

ポインタ変数

Cは「ポインタ」をサポートしています。これは他の変数を「指して」います。ポインタには、変数の値ではなく、変数のアドレス（格納場所）が保持されます。

ポインタ変数の宣言

ポインタは、次の例に示されているように、通常の C の形式に従ってホスト変数として定義します。

```
int    *int_ptr;
char   *char_ptr;
```

ポインタ変数の参照

SQL 文では、次の例に示すように、ポインタにはコロンを接頭辞として付けてください。

```
EXEC SQL SELECT intcol INTO :int_ptr FROM ...
```

文字列へのポインタを除き、参照される値のサイズは宣言で指定した型のサイズに基づいて決まります。文字列へのポインタでは、参照値は NULL で終わる文字列とみなされます。参照される値のサイズは、実行時に *strlen()* 関数をコールしたときに決定されます。詳細は 4-45 ページの「[各国語サポート](#)」を参照してください。

ポインタを使用すると、**構造体**のメンバーを参照できます。まず、ポインタをホスト変数であると宣言し、次に例に示されるように必要なアドレスにポインタを設定します。**構造体**メンバーとポインタ変数のデータ型を同一にしてください。両者が一致していないと、ほとんどのコンパイラは警告を出します。

```
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. */
    ...
}
```

構造体ポインタ

ポインタを、構造体へのホスト変数として使用できます。次の例は

- 構造体を宣言します。
- 構造体へのポインタを宣言します。
- 構造体にメモリーを割り当てます。

- 構造体へのポインタを間合せてホスト変数として使用します。
- 構造体のコンポーネントの参照を解除して、結果を印刷します。

```

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);

```

SQL 文では、ホスト構造体へのポインタは、ホスト構造体とまったく同じ方法で参照されます。"address of" 表記 (&) は必須ではありません。実際には使用するとエラーになります。

各国語サポート

広く使用されている 7 ビットまたは 8 ビットの ASCII キャラクタ・セットおよび EBCDIC キャラクタ・セットが英数字を表すのに十分であっても、日本語などのアジアの言語の中には、数千という文字があるものもあります。これらの言語は、各文字を表すのに 16 ビット (2 バイト) を必要とします。Oracle8 は、このように異なる言語をどのように扱っているでしょうか。

Oracle8 には、シングルバイトおよびマルチバイトの文字データを処理し、キャラクタ・セット間で変換できるように、各国語サポート (NLS) が用意されています。これによって、異なる言語環境でアプリケーションを実行できます。NLS では、数値書式と日付書式はユーザー・セッション用に指定された言語変換に自動的に適応します。したがって、NLS により、世界中のユーザーがそれぞれの母国語で Oracle8 と対話できます。

様々な NLS パラメータを指定して、言語によって異なる機能の操作を制御できます。これらのパラメータのデフォルト値は、Oracle8 の初期化ファイルで設定できます。表 4-6 にそれぞれの NLS パラメータが指定するものを示します。

表 4-6 NLS パラメータ

NLS パラメータ	指定
NLS_LANGUAGE	言語によって異なる表記法
NLS_TERRITORY	地域によって異なる表記法
NLS_DATE_FORMAT	日付書式
NLS_DATE_LANGUAGE	日と月の名前に使用する言語
NLS_NUMERIC_CHARACTERS	10 進数文字およびグループ・セパレータ
NLS_CURRENCY	ローカル通貨記号
NLS_ISO_CURRENCY	ISO 通貨記号
NLS_SORT	ソート基準

主なパラメータは NLS_LANGUAGE および NLS_TERRITORY です。NLS_LANGUAGE には言語によって異なる機能のデフォルト値を指定します。この機能には次のものが含まれます。

- サーバー・メッセージのための言語
- 日と月の名前に使用する言語
- ソート基準

NLS_TERRITORY には、地域によって異なる機能のデフォルト値を指定します。この機能には次のものが含まれます。

- 日付書式
- 10 進数文字
- グループ・セパレータ
- ローカル通貨記号
- ISO 通貨記号

パラメータ NLS_LANG を次のように指定して、ユーザー・セッション用に言語ごとに異なる NLS 機能の操作を制御できます。

NLS_LANG = <language>_<territory>.<character set>

language はユーザー・セッション用の NLS_LANGUAGE の値、*territory* は、NLS_TERRITORY の値、*character set* は端末に使用されるコード体系を指定します。「コード体系」（通常はキャラクタ・セットまたはコード・ページと呼ばれる）は、端末で表示できるキャラクタ・セットに対応する数値コードの範囲です。また、これには端末との通信を制御するコードも入っています。

NLS_LANG は、環境変数（または使用しているシステムでこれに相当するもの）として定義します。たとえば、C シェルを使用する UNIX では、NLS_LANG を次のように定義できます。

```
setenv NLS_LANG French_France.WE8ISO8859P1
```

NLS パラメータの値は、Oracle8 のデータベース・セッション中に変更できます。ALTER SESSION 文を次のように指定してください。

```
ALTER SESSION SET <nls_parameter> = <value>
```

Pro*C/C++ は NLS 機能をすべてサポートするので、アプリケーションは Oracle8 データベースに格納されている外国語データを処理できます。たとえば、外国語の文字変数を宣言し、それを文字列関数（INSTRB、LENGTHB および SUBSTRB など）に渡すことができます。これらの関数には、それぞれ INSTR および LENGTH、SUBSTR 関数と同じ構文がありますが、文字単位ではなく、バイト単位を基礎とする操作になります。

関数 NLS_INITCAP、NLS_LOWER および NLS_UPPER を使用して、大 / 小文字の変換の特別なインスタンスを扱えます。さらに、関数 NLSSORT を使用して、バイナリ順序ではなく言語上の順序に基づいて WHERE 句の比較を指定できます。NLS パラメータを TO_CHAR、TO_DATE および TO_NUMBER 関数に渡すこともできます。NLS の詳細は『Oracle8i アプリケーション開発者ガイド』を参照してください。

NCHAR 変数

3 つの内部データベース・データ型は、固定幅マルチバイト文字列を格納できます。それらのデータ型は、NCHAR、NCLOB および NVARCHAR2 です。（NCHAR VARYING とも呼びます。）これらのデータ型は、リレーショナル列でのみ指定できます。これまでの Pro*C/C++ リリースでもマルチバイト NCHAR ホスト変数がサポートされていましたが、意味解釈に多少の違いがあります。

コマンドライン・オプション NLS_LOCAL を YES に設定すると、Oracle7 と同じ以前のセマンティクスのマルチバイト・サポートを SQLLIB が用意します。（引用符で囲まれた文字列から文字 "N" が除かれます。）SQLLIB はブランクの埋込みと除去、標識変数の設定などを行います。

NLS_LOCAL を NO（デフォルト）に設定すると、新しい方法によるマルチバイト文字列が Oracle8 でサポートされます。（引用符付きの文字列の前に文字 "N" が付きます。）SQLLIB ではなくデータベースで、空白の埋込みと除去、標識変数の設定が実行されます。

新しいアプリケーションではすべて NLS_LOCAL=NO を使用してください。

CHARACTER SET [IS] NCHAR_CS

各国文字キャラクタ・セットを保持するホスト変数を指定するには、"*CHARACTER SET [IS] NCHAR_CS*" 句を文字変数宣言に挿入します。これで、その変数に各国文字キャラクタ・セット・データを格納できます。トークン *IS* は省略してもかまいません。NCHAR_CS は、各国文字キャラクタ・セットの名前です。

たとえば、次のとおりです。

```
char character set is nchar_cs *str = "<Japanese_string>;
```

この例では、<Japanese_string> はダブルバイト文字で構成されたもので、各国文字キャラクタ・セットでは JA16EUCFIXED になります。これは変数 NLS_NCHAR で定義されています。

また、コマンドラインに NLS_CHAR=str と入力し、アプリケーションのコードを変更して、同じことを実行することもできます。

```
char *str = "<Japanese_string>"
```

Pro*C/C++ では、このように宣言された変数は、環境変数 NLS_NCHAR で指定されたキャラクタ・セットとして処理されます。NCHAR 変数のサイズは、通常の C の変数と同じようにバイト数で指定されます。

データを選択して str に入れるには、次の簡単な問合せを指定します。

```
EXEC SQL
    SELECT ENAME INTO :str FROM EMP WHERE DEPT = n'<Japanese_string1>';
```

または、次の SELECT 文で str を指定します。

```
EXEC SQL
    SELECT DEPT INTO :dept FROM DEPT_TAB WHERE ENAME = :str;
```

環境変数 NLS_NCHAR

Pro*C/C++ では、NLS_LOCAL=NO の場合にマルチバイト・キャラクタ・セット (NCHAR) をサポートしています。NLS_LOCAL=NO の場合に新しい環境変数 NLS_NCHAR が有効な各国文字キャラクタ・セットに設定されていると、データベース・サーバーで NCHAR がサポートされます。詳細は『Oracle8i リファレンス・マニュアル』の NLS_NCHAR を参照してください。

NLS_NCHAR には、プリコンパイル時と実行時に、有効な（言語名ではなく、NLS_LANG で設定する）固定幅キャラクタ・セットを指定する必要があります。最初の SQL 文の実行時に SQLLIB によるランタイム・チェックが行われます。プリコンパイル時のキャラクタ・セットと実行時のキャラクタ・セットが違っていると、SQLLIB はエラー・コードを戻します。

VAR 内の CONVBUFSZ 句

EXEC SQL VAR 文を使用すると、ホスト変数を Oracle8 の外部データ型に同値化して、デフォルトの割当てをオーバーライドできます。これは「ホスト変数の同値化」と呼ばれます。

EXEC SQL VAR 文にはオプション句 CONVBUFSZ (<size>) を付けられます。Oracle8 ランタイム・ライブラリ内で、指定したホスト変数をキャラクタ・セット間で変換するためのバッファのサイズ <size> をバイト単位で指定します。

新しい構文は、次のいずれかです。

```
EXEC SQL VAR host_variable IS datatype [CONVBUFSZ [IS] (size)] ;
```

または

```
EXEC SQL VAR host_variable [CONVBUFSZ [IS] (size)];
```

この場合、datatype は次のとおりです。

```
type_name [ ( { length | precision, scale } ) ]
```

すべてのキーワード、例および変数の詳細は F-111 ページの「[VAR \(Oracle 埋込み SQL 宣言文\)](#)」を参照してください。

埋込み SQL 内の文字列

埋込み SQL 文内のマルチバイト文字列は、その文字列がマルチバイトであることを示す文字リテラルと、その直後に続く文字列から構成されます。文字列の部分は、通常の引用符 (') で囲みます。

たとえば、次のような埋込み SQL 文があるとします。

```
EXEC SQL SELECT empno INTO :emp_num FROM emp
        WHERE ename = N'<Japanese_string>';
```

前述の文には、マルチバイト文字列が含まれています (<Japanese_string> は、実際には漢字である可能性があります)。つまり、文字列の直前に文字リテラル N が付いているため、これはマルチバイト文字列として識別されます。Oracle8 では大 / 小文字が区別されないため、この例では "n" と "N" のどちらを使用してもかまいません。

文字列の制限事項

データ型の同値化 (TYPE コマンドまたは VAR コマンド) は、NLS マルチバイト文字列には使用できません。

動的 SQL 方法 4 は Pro*C/C++ の NLS マルチバイト文字列ホスト変数では利用できません。

標識変数

標識変数は、(NLS_CHAR オプションで指定された) NLS マルチバイトである文字列ホスト変数とともに使用できます。

上級トピック

この章では、Pro*C/C++ の技術上級編を扱います。この章では、次の事項について説明します。

- 文字データの処理
- データ型変換
- データ型の同値化
- C プリプロセッサ
- プリコンパイル済のヘッダー・ファイル
- Oracle プリプロセッサ
- 数値定数の評価
- OCI リリース 8 の SQLLIB 拡張相互運用性
- OCI リリース 8 とのインタフェース
- 埋込み (OCI リリース 7) Oracle コール
- SQLLIB パブリック関数の新しい名前
- X/Open アプリケーションの開発

文字データの処理

この項では、Pro*C/C++ プリコンパイラが文字ホスト変数処理する方法を説明します。ホスト変数キャラクタ・タイプには次の 4 種類があります。

- 文字配列
- 文字列へのポインタ
- VARCHAR 変数
- VARCHAR へのポインタ

VARCHAR（プリコンパイラが提供するホスト変数データ構造体）と、VARCHAR2（可変長の文字列に対応する Oracle 内部データ型）を混同しないでください。

プリコンパイラ・オプション CHAR_MAP

CHAR_MAP プリコンパイラ・コマンドライン・オプションを使用して、char[n] および char ホスト変数のデフォルトのマッピングを指定できます。Oracle8*i*は、CHARZ にマップします。CHARZ により、ANSI 固定文字形式をインプリメントできます。文字列は、固定長の空白埋めおよびゼロ終了記号付きです。VARCHAR2 値（NULL を含む）は常に固定長の空白埋めです。表 5-1 に CHAR_MAP の可能な設定を示します。

表 5-1 CHAR_MAP 設定

CHAR_MAP の設定	デフォルトとなる場合	説明
VARCHAR2		値がすべて（NULL を含む）固定長の空白埋め。
CHARZ	DBMS=V7、DBMS=V8	固定長の空白埋めおよび NULL 終了記号付き。ANSI 固定キャラクタ・タイプに準拠。
STRING	新しい形式	NULL 終了記号付き。C プログラムで使われている ASCII 形式に準拠。
CHARF	以前は、VAR 宣言または TYPE 宣言が行われた場合のみ	固定長の空白埋め。NULL は空白埋めなし。

デフォルトのマッピングは、これまでの Pro*C/C++ リリースと同じ CHAR_MAP=CHARZ です。

従来の DBMS=V6_CHAR（廃止されます）のかわりに CHAR_MAP=VARCHAR2 を指定します。

CHAR_MAP オプションのインラインでの使用方法

char 変数または char[n] 変数が異なった方法で宣言されていない限り、インライン CHAR_MAP オプションによってそのマッピングが決まります。次のコードの一部分は、このオプションを Pro*C/C++ でインライン設定した結果です。

```
char ch_array[5];

strcpy(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' } */

strcpy (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', '\0', '4', '5' } */

strcpy( 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', ' ', ' ', ' ', ' ' } */
```

DBMS オプションおよび CHAR_MAP オプションの影響

DBMS オプションおよび CHAR_MAP オプションによって、Pro*C/C++ で文字配列および文字列のデータを処理する方法が決まります。これらのオプションによってプログラムは、ANSI の固定長文字列との互換性を維持するか、可変長の文字列を使用する Oracle および Pro*C/C++ の以前のリリースとの互換性を維持できます。DBMS と CHAR_MAP の詳細は、[第 10 章「プリコンパイラのオプション」](#)を参照してください。

DBMS オプションは、入力（ホスト変数から Oracle 表へ）および出力（Oracle 表からホスト変数へ）の両方で文字データに影響を及ぼします。

文字配列および CHAR_MAP オプション

文字配列のマッピングは DBMS オプションとは関連しない CHAR_MAP オプションでも設定できます。DBMS=V7 または DBMS=V8 はどちらも CHAR_MAP=CHARZ を使用します。これは CHAR_MAP=VARCHAR2、STRING または CHARF を指定することでオーバーライドできます。

入力時

文字配列 入力では、DBMS オプションによって、プログラム間でホスト変数文字配列に必要な形式が決まります。CHAR_MAP=VARCHAR2 の場合、ホスト変数文字配列には空白埋込みが必要です。また NULL 終了記号を付けないでください。DBMS=V7 あるいは V8 のときには、文字配列は NULL ターミネート ('\0') である必要があります。

CHAR_MAP オプションが VARCHAR2 に設定されると、値に後続する空白が最初の非空白文字まで除去されてから、値がデータベースに送られます。未初期化文字配列には、NULL 文字が含まれている場合があるので注意してください。NULL が表に挿入されるのを確実に防ぐには、長さに達するまで文字配列に空白を埋め込む必要があります。たとえば、次の文を実行する場合を考えます。

```
char emp_name[10];
...
strcpy(emp_name, "MILLER");      /* WRONG! Note no blank-padding */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

この場合は、文字列「MILLER」が「MILLER\0\0\0\0」（4 つの NULL バイトを末尾に追加）として挿入されています。この値は、次の検索条件を満たすものではありません。

```
... WHERE ename = 'MILLER';
```

CHAR_MAP が VARCHAR2 に設定されている場合に文字配列を INSERT するには、次の文を実行します。

```
strcpy(emp_name, "MILLER    ", 10); /* 4 trailing blanks */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

DBMS=V7 または DBMS=V8 のときは、文字配列中の入力データは NULL で終了する必要があります。データが NULL で終わっているかどうかを確認してください。

```
char emp_name[11]; /* Note: one greater than column size of 10 */
...
strcpy(emp_name, "MILLER");      /* No blank-padding required */
EXEC SQL INSERT INTO emp (empno, ename, deptno) VALUES
    (1234, :emp_name, 20);
```

文字ポインタ ポインタには、入力データを保持できる大きさの、NULL 終了記号付きのバッファをアドレス指定してください。使用しているプログラムで、これを実行するのに十分なメモリーを割り当てる必要があります。

入力時

次の例は、データベースから文字配列に取り出される値に CHAR_MAP オプションの設定が及ぼす効果として考えられるすべての組合せを示しています。

次のデータベースで、

```
TABLE strdbase ( ..., strval VARCHAR2(6));
```

strval 列に次の文字列が入っているとします。

```
" "      -- string of length 0
"AB"     -- string of length 2
"KING"   -- string of length 4
"QUEEN"  -- string of length 5
"MILLER" -- string of length 6
```

Pro*C/C++ プログラムでは、5 文字のホスト配列 *str* を文字「X」で初期化して、列 *strval* のすべての値の取出しに使用します。

```
char  str[5] = {'X', 'X', 'X', 'X', 'X'} ;
short str_ind;
...
EXEC SQL SELECT strval INTO :str:str_ind WHERE ... ;
```

CHAR_MAP が VARCHAR2、CHARF、CHARZ または STRING に設定されると、配列 *str* および標識変数 *str_ind* の結果は、次のようになります。

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	"MILL0" 6
STRING	"0XXXX"	-1 "AB0XX" 0	"KING0" 0	"QUEE0" 5	"MILL0" 6

0 は NULL 文字 '\0' を示します。

出力時

文字配列 出力では、DBMS オプションまたは CHAR_MAP オプションによってホスト変数文字配列のプログラム中の形式を判断します。CHAR_MAP=VARCHAR2 のとき、ホスト変数文字配列は配列の長さには達するまで空白埋込みが行われますが、NULL 終了記号が付けられることはありません。DBMS=V7 または DBMS=V8（あるいは CHAR_MAP=CHARZ）のときは、文字配列は空白埋込みが行われてから、配列の最終位置に NULL 終了記号が付けられます。

次の文字出力の例について考えてみます。

```
CREATE TABLE test_char (C_col CHAR(10), V_col VARCHAR2(10));

INSERT INTO test_char VALUES ('MILLER', 'KING');
```

この表を選択するプリコンパイラ・プログラムには、次のような埋込み SQL が記述されています。

```
...
char name1[10];
char name2[10];
...
EXEC SQL SELECT C_col, V_col INTO :name1, :name2
      FROM test_char;
```

CHAR_MAP=VARCHAR2 を指定してプログラムをプリコンパイルすると、*name1* に次の文字列が入ります。

```
"MILLER####"
```

つまり、名前「MILLER」の後に 4 つの空白が続き、NULL 終了記号は付きません。（*name1* がサイズ 15 で宣言されている場合、名前に続く空白は 9 つなので注意してください。）

name2 には次の文字列が入ります。

```
"KING#####" /* 6 trailing blanks */
```

DBMS=V7 または V8 を指定してプログラムをプリコンパイルすると、*name1* には次の文字列が入ります。

```
"MILLER###\0" /* 3 trailing blanks, then a null-terminator */
```

つまり、名前を含み、列の長さに達するまで空白埋込みが行われ、NULL 終了記号が付けられた文字列です。*name2* には、次の文字が含まれます。

```
"KING#####\0"
```

まとめると、CHAR_MAP=VARCHAR2 のとき、CHARACTER 列または VARCHAR2 列からの出力には、ホスト変数配列の長さに達するまで空白埋込みが行われます。DBMS=V7 または DBMS=V8 のとき、出力文字列には常に NULL 終了記号が付けられます。

文字ポインタ DBMS オプションおよび CHAR_MAP オプションを使用しても、文字データがポインタ・ホスト変数に出力される方法に影響を与えることはありません。

データを文字ポインタ・ホスト変数に出力するとき、ポインタが指し示すバッファには、表からの出力に加えて NULL 終了記号のための 1 バイトを保持できる大きさが必要です。

プリコンパイラのランタイム環境は `strlen()` をコールして、出力バッファのサイズを判断します。したがって、バッファに埋込み NULL ('\\0') が含まれていないことを確認してください。データを入れる前に '\\0' 以外の値でバッファをみだし、NULL で終了してください。

注意：C のポインタは DBMS=V7 もしくは V8 と MODE=ANSI でプリコンパイルした Pro*C/C++ プログラムで使用できます。ただし、ポインタは SQL 標準に対応するプログラムで有効なホスト変数型ではありません。FIPS フラガーはポインタをホスト変数として使用している場合に警告します。

次のコードの一部では、前の項で定義された列および表を使用し、文字ポインタ・ホスト変数に宣言および SELECT を行う方法を示します。

```
...
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;
```

前述の SELECT 文が DBMS または CHAR_MAP 設定で実行されると、フェッチされる値は次のようになります。

```
"MILLER####\\0"      /* 4 trailing blanks and a null terminator */

"KING#####\\0"      /* 6 blanks and null */
```

VARCHAR 変数およびポインタ

次の例では、VARCHAR ホスト変数が宣言される方法を示します。

```
VARCHAR emp_name1[10]; /* VARCHAR variable */
VARCHAR *emp_name2;    /* pointer to VARCHAR */
```

入力時

VARCHAR 変数 VARCHAR 変数を入力ホスト変数として使用すると、プログラムに必要なのは、展開された VARCHAR 宣言（例では `emp_name1.arr`）の配列メンバーに必要な文字列を配置し、長さメンバー（`emp_name1.len`）を設定することのみです。配列に空白を埋め込む必要はありません。`emp_name1.len` の文字が正確に Oracle に送られ、空白および NULL があればカウントします。次の例では、`emp_name1.len` を 8 に設定します。

```
strcpy((char *)emp_name1.arr, "VAN HORN");
emp_name1.len = strlen((char *)emp_name1.arr);
```

VARCHAR のポインタ 入力ホスト変数として VARCHAR へのポインタを使用する場合は、展開される VARCHAR 宣言に十分なメモリーを割り当てる必要があります。その後、次の例に示されているように、必要な文字列を配列メンバーに配置し長さメンバーを設定する必要があります。

```
emp_name2 = malloc(sizeof(short) + 10) /* len + arr */
strcpy((char *)emp_name2->arr, "MILLER");
emp_name2->len = strlen((char *)emp_name2->arr);
```

または、*emp_name2* が既存の VARCHAR（この場合 *emp_name1*）を指すように、割当てを次のように記述できます。

```
emp_name2 = &emp_name1;
```

その後次のように、通常の方法で VARCHAR ポインタを使用します。

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME, DEPTNO)
VALUES (:emp_number, :emp_name2, :dept_number);
```

出力時

VARCHAR 変数 VARCHAR 変数を出力ホスト変数として使用すると、プログラム・インタフェースは長さのメンバーを設定しますが、配列メンバーに NULL 終了記号は付けません。文字配列では、プログラムは VARCHAR 変数の *arr* メンバーに NULL 終了記号を付けてから、*printf()* または *strlen()* のような関数に渡すことができます。次に例を示します。

```
emp_name1.arr[emp_name1.len] = '\0';
printf("%s", emp_name1.arr);
```

または、長さメンバーを使用して、文字列の印刷を制限できます。

次のようになります。

```
printf("%.s", emp_name1.len, emp_name1.arr);
```

文字配列よりも VARCHAR 変数が優れている点は、Oracle によって戻される値の長さがすぐにわかることです。文字配列では、文字列の実際の長さを知るために、後続する空白を自分で削除する必要がある場合もあります。

VARCHAR ポインタ 出力ホスト変数として VARCHAR へのポインタを使用すると、プログラム・インタフェースは変数の最大長を長さメンバー（この例では *emp_name2->len*）を調べることで判断します。このため、プログラムは、毎回フェッチが行われる前にこのメンバー

を設定する必要があります。その後次の例にあるように、フェッチ処理時に長さメンバーは、戻された実際の文字数に設定されます。

```
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);
```

Unicode 変数

Pro*C/C++ では、ホスト char 変数で固定幅の Unicode データ（文字セット Unicode Standard Version 2.0。UCS-2 と呼ばれます）を使用できます。UCS-2 では、1 文字に対して 2 バイトが使用されます。UCS-2 のデータ型は、符号なし 2 バイトです。UCS-2 の SQL 文は、まだサポートされていません。

次のサンプル・コードでは、Unicode 型 *utext* のホスト変数である *employee* は、20Unicode 文字長として宣言されています。表 *emp* は、60 バイト長の列 *ename* で作成されます。このため、マルチ・バイト文字で最大 3 バイトまでの、アジア言語のデータベース文字セットがサポートされます。

```
utext employee[20] ; /* Unicode host variable */
EXEC SQL CREATE TABLE emp (ename CHAR(60));
/* ename is in the current database character set */
EXEC SQL INSERT INTO emp (ename) VALUES ('test') ;
/* 'test' in NLS_LANG encoding converted to database character set */
EXEC SQL SELECT * INTO :employee FROM emp ;
/* Database character set converted to Unicode */
```

パブリック・ヘッダー・ファイル *sqlucs2.h* を、アプリケーション・コードに含める必要があります。次のように記述します。

- 文を含めます。

```
#include oratypes.h
```

- *uvarchar*、つまり "Unicode varchar" は、次のように定義します。

```
struct uvarchar
{
    ub2 len;
    utext arr[1] ;
};
typedef struct uvarchar uvarchar ;
```

- *ulong_varchar*、つまり "Unicode long varchar" は、次のように定義します。

```
struct ulong_varchar
{
    ub4 len ;
    utext arr[1] ;
```

```
    }
    typedef struct ulong_vchar ulong_vchar ;

utext のデフォルト・データ型は、すべての文字変数のデフォルトである CHARZ と同じで、
空白埋めおよび NULL 終了記号付きです。

CHAR_MAP プリコンパイラ・オプションを使用して、次のようにデフォルト・データ型を
変更してください。

#include <sqlca.h>
#include <sqlucs2.h>

main()
{
    utext employee1[20] ;

    /* Change to STRING datatype: */
    EXEC ORACLE OPTION (CHAR_MAP=STRING) ;
    utext employee2[20] ;

    EXEC SQL CREATE TABLE emp (ename CHAR(60)) ;
    ...
    /*****
    Initializing employee1 or employee2 is compiler-dependent.
    *****/
    EXEC SQL INSERT INTO emp (ename) VALUES (:employee1) ;
    ...
    EXEC SQL SELECT ename INTO :employee2 FROM emp;
    /* employee2 is now not blank-padded and is null-terminated */
    ...
}
```

Unicode 変数の使用に関する制限事項

- SQL 文の静的および動的 SQL に Unicode を含めることはできません。次のように記述できません。

```
#include oratypes.h
utext sqlstmt[100] ;
...
/* If sqlstmt contains a SQL statement: */
EXEC SQL PREPARE s1 FROM :sqlstmt ;
EXEC SQL EXECUTE IMMEDIATE :sqlstmt ;
...
```

- utext 変数では、データ型は同値化できません。次のコードを使用できません。

```
typedef utext utext_5 ;
EXEC SQL TYPE utext_5 IS STRING ;
```

- CONVBUSZ は、変換バッファ・サイズとして使用できません。CHAR_MAP オプションを使用してください。詳細は、4-49 ページの「[VAR 内の CONVBUSZ 句](#)」を参照してください。
- Oracle 動的 SQL 方法 4 では、Unicode をサポートしていません。方法 4 の詳細は、[第 14 章「ANSI 動的 SQL」](#)を参照してください。
- オブジェクト型は Unicode をサポートしていません。オブジェクト型の詳細は、[第 17 章「オブジェクト」](#)を参照してください。

データ型変換

プリコンパイル時に、デフォルトの外部データ型がそれぞれのホスト変数に割り当てられます。たとえば、プリコンパイラは **short int** および **int** 型のホスト変数に **INTEGER** 外部データ型を割り当てます。

実行時に、SQL 文で使用されるそれぞれのホスト変数のデータ型コードは Oracle に渡されます。Oracle はそのコードを使用して、内部データ型と外部データ型間で変換を行います。

SELECT された列（または疑似列）値を出力ホスト変数に割り当てるには、Oracle ではソース列の内部データ型をホスト変数のデータ型に変換する必要があります。同様に、入力ホスト変数の値を列に割り当てたり列と比較したりする前に、Oracle はホスト変数の外部データ型をターゲット列の内部データ型に変換する必要があります。

内部データ型と外部データ型間の変換は通常の変換規則に従っています。たとえば、CHAR の値 "1234" を Cshort 値に変換することができます。"65543"（大きすぎる値）または "10F"（10 進数でない値）は、C の short 値には変換できません。同様に、アルファベット文字が含まれる char[n] 値を NUMBER 値には変換できません。

データ型の同値化

データ型の同値化によって、Oracle が入力データを解釈する方法および Oracle が出力データをフォーマットする方法を制御できます。データ型の同値化を使用すると、プリコンパイラが割り当てるデフォルトの外部データ型をオーバーライドできます。サポートされている C のホスト変数データ型は、個々の変数ごとに Oracle の外部データ型に同値化できます。また、ユーザー定義のデータ型を Oracle の外部データ型と同値化することもできます。

ホスト変数の同値化

デフォルトでは、Pro*C/C++ プリコンパイラはすべてのホスト変数に特定の外部データ型を割り当てます。

表 5-2 にデフォルトの割当てを示します。

表 5-2 デフォルトの型割当て

C 型または擬似型	Oracle 外部型
char	VARCHAR2 (CHAR_MAP=VARCHAR2)
char[n]	CHARZ (DBMS=V7、DBMS=V8 のデフォルト)
char*	STRING (CHAR_MAP=STRING)
	CHARF (CHAR_MAP=CHARF)
int,int*	INTEGER
short,short*	INTEGER
long,long*	INTEGER
float,float*	FLOAT
double,double*	FLOAT
VARCHAR*, VARCHAR[n]	VARCHAR

VAR 文を使用すると、ホスト変数を Oracle 外部データ型に同値化することによって、デフォルトの割当てを変更できます。使用する構文は次のとおりです。

```
EXEC SQL VAR host_variable IS type_name [ (length) ];
```

このとき、*host_variable* はすでに宣言済の入力ホスト変数または出力ホスト変数（またはホスト配列）、*type_name* は有効な外部データ型の名前、*length* は有効な長さをバイト数で指定した整数リテラルです。

ホスト変数の同値化には、いくつかの利点があります。たとえば、EMP 表から従業員の名前を SELECT し、それらを NULL 終了記号付きの文字列を受け入れるルーチンに渡すとします。これらの名前に、明示的に NULL 終了記号を付ける必要はありません。次のように、ホスト変数を STRING 外部データ型に同値化するのみで済みます。

```
...
char emp_name[11];
EXEC SQL VAR emp_name IS STRING(11);
```

EMP 表の ENAME 列の長さは 10 文字のため、NULL 終了記号を含めるために新しい *emp_name* に 11 文字割り当てます。ENAME 列から *emp_name* に入れる値を SELECT する場合、プログラム・インタフェースはユーザーにかわって値に NULL 終了記号を付けます。

4-4 ページの「[Oracle の外部データ型](#)」の外部データ型表に記したデータ型は NUMBER 以外（かわりに VARNUM を使用します）のどれを使用することもできます。

ユーザー定義型同値化

また、ユーザー定義のデータ型を Oracle の外部データ型と同値化することもできます。最初に、必要を満たす外部データ型によく似た構造の、新しいデータ型を定義します。次に、TYPE 文を使用して新しいデータ型を外部データ型に同値化します。

TYPE 文を使用すると、ホスト変数のクラス全体に Oracle の外部データ型を割り当てることができます。この場合は、次の構文を使用します。

```
EXEC SQL TYPE user_type IS type_name [ (length) ] [REFERENCE];
```

漢字を使用するために可変長文字列データ型が必要だとします。最初に、**short** 型の長さコンポーネントと、それに続く 65533 バイトのデータ・コンポーネントからなる構造体を宣言します。次に、その構造体に基づく新規のデータ型を、**typedef** を使用して定義します。最後に、次の例のように、新しいユーザー定義のデータ型を VARRAW 外部データ型に同値化します。

```
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
...
```

新しい *graphics* 型に長さ 4000 バイトを指定します。この構造体では、それがデータ・コンポーネントの最大長になるためです。プリコンパイラでは、長さを Oracle サーバーに送るときに *len* コンポーネント（および必要な埋込み）を指定できます。

REFERENCE 句

ユーザー定義型はポインタとして宣言できます。明示的にスカラーまたは構造体型へのポインタとして宣言することも、暗黙的に配列として宣言することもできます。そして、この型を EXEC SQL TYPE 文で使用できます。この場合は、次の例に示すように、文の終わりに REFERENCE 句を使用してください。

```
typedef unsigned char *my_raw;

EXEC SQL TYPE my_raw IS VARRAW(4000) REFERENCE;
my_raw    graphics_buffer;
...
graphics_buffer = (my_raw) malloc(4004);
```

この例では、型の長さ（4000）を上回ってメモリーを余分に割り当ててあります。プリコンパイラは長さ（*short* のサイズ）を戻し、システムのワード位置合わせに関する制限を考慮して長さの後に埋込みを追加できるので、このような割当てが必要になります。使用しているシステムの位置合せの規則がわからない場合は、必ずその長さで埋込みの分として余分に何バイトか割り当ててください。（通常は9バイトで十分です。）例は 4-21 ページの「[サンプル・プログラム:sqlvcp\(\) の使用](#)」を参照してください。

CHARF 外部データ型

CHARF は、固定長文字列です。このデータ型を VAR 文および TYPE 文で使用して、DBMS オプションまたは CHAR_MAP オプションの設定に関係なく C のデータ型を固定長の SQL 標準データ型 CHAR に同値化できます。

DBMS=V7 または DBMS=V8 のとき、VAR 文または TYPE 文で外部データ型 CHARACTER を指定すると、C のデータ型は固定長データ型の CHAR（データ型コード 96）に同値化されます。しかし、CHAR_MAP=VARCHAR2 の場合、C のデータ型は可変長データ型 VARCHAR2（コード 1）と同値化されます。

現在では、VAR 文または TYPE 文で CHARF データ型を使用すれば、C のデータ型を固定長の SQL 標準データ型 CHARACTER と同値化できます。CHARF を使用すると、DBMS オプションまたは CHAR_MAP オプションの設定に関係なく、固定長のキャラクタ・タイプになるように同値化が行われます。

EXEC SQL VAR と TYPE 宣言文の利用

EXEC SQL VAR... 文または EXEC SQL TYPE... 文は、プログラム中のどの位置でも記述できます。これらの文は、その影響を受ける変数のデータ型を、TYPE 文または VAR 文が記述された位置から変数の有効範囲の終わりまでの範囲内で変更する実行文として扱われます。MODE=ANSI でプリコンパイルするときには、宣言文を使用する必要があります。この場合、TYPE 文または VAR 文は宣言文の中に記述する必要があります。詳細は、F-106 ページの「[TYPE \(Oracle 埋込み SQL 宣言文\)](#)」および F-111 ページの「[VAR \(Oracle 埋込み SQL 宣言文\)](#)」を参照してください。

Sample4.pc: データ型の同値化

この項のサンプル・プログラムは、Pro*C/C++ プログラムでデータ型の同値化を使用する方法を示しています。このプログラムは、demo ディレクトリの sample4.pc と同じもので、LONG VARRAW 外部データ型を使用してデータ型の同値化を行います。異なるシステムに移植できる実用的な例を示すために、このプログラムでは、バイナリ・ファイルをデータベースに挿入し、そのファイルをデータベースから取り出します。

このプログラムでは、LOB 埋込み SQL 文が使用されます。ラージ・オブジェクト (LOB) の使用方法の詳細は、16-1 ページの「ラージ・オブジェクト (LOB)」を参照してください。

このプログラムの目的については、導入部分の説明を参照してください。

```

/*****
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 recreate) 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)
-----	-----
hello	5508

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
 * 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. */
#ifdef __STDC__
void do_connect(void);
void create_table(void);
void sql_error(char *);
void list_executables(void);
void print_menu(void);
void do_insert(vchar *, char *);
void do_retrieve(vchar *, char *);
void do_delete(vchar *);
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(/* vchar *, char */);
void do_retrieve(/* vchar *, char */);
void do_delete(/* vchar */);
ub4 read_file(/* char *, OCIBlobLocator */);
void write_file(/* char *, OCIBlobLocator */);
#endif

void main()
{
    char reply[20], filename[100];
    vchar key[20];
    short ok = 1;

    /* Connect to the database. */
    do_connect();

    printf("Do you want to create (or recreate) 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\n");
            printf("full path: ");
            gets(filename);
            do_retrieve((varchar *)&key, filename);
            break;
        case 'L': case 'l':
            /* User selected list - just call the list routine. */
            list_executables();
            break;
        case 'D': case 'd':
            /* User selected delete - get the key for the executable to delete. */
            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 SQL WHENEVER SQLERROR 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");
}

/* 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;
    ub4      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, 0L, 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, 0L, 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;
}
```



```
/* Generic error handler. The 'routine' parameter should contain the name
   of the routine executing when the error occurred. 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);
}

/* 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          *out_fd;          /* File descriptor for the output file */
    ub4           amt;
    ub4           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;

    /* Write the Final Piece (or First and Only Piece if not Polling) */
    (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: ");
}

```

C プリプロセッサ

Pro*C/C++ は C プリプロセッサ宣言文の大半をサポートしています。Pro*C/C++ プリプロセッサを使用すると、次のような作業を実行できます。

- **#define** 宣言文を使用した、定数およびマクロの定義、および VARCHAR などの Pro*C/C++ のデータ型宣言をパラメータ化する場合の定義済エンティティの使用。
- **#include** 宣言文を使用した、プリコンパイラに必要な *sqlca.h* などのファイルの読取り。
- 個別のファイルにある定数、マクロの定義およびプリコンパイラでの **#include** 宣言文を使用したこのファイルの読取り。

Pro*C/C++ プリプロセッサの機能

Pro*C/C++ プリプロセッサは、ほとんどの C プリプロセッサ・コマンドを認識し、必要なマクロ置換、ファイルの組込みおよび条件付きソース・テキストの組込みまたは削除を効率的に実行します。Pro*C/C++ プリプロセッサは、この事前処理によって取得した値を使用し、ソース出力テキスト（生成される .c 出力ファイル）を変更します。

この点について、次にプログラム例を示します。仮に次のプログラムを書いたとします。

```

#include "my_header.h"
...
VARCHAR name[VC_LEN];           /* a Pro*C-supplied datatype */
char   another_name[VC_LEN];     /* a pure C datatype */
...

```

カレントディレクトリ中のファイル *my_header.h* には、特に次の行が含まれていると仮定します。

```
#define VC_LEN 20
```

プリコンパイラはファイル *my_header.h* を読み込み、VC_LEN の定義済の値（つまり 20）を使用して、*name* の構造体を VARCHAR[20] として宣言します。

char はネイティブ型です。プリコンパイラは、*another_name*[VC_LEN] の宣言時に 20 を代入しません。

プリコンパイラは C のデータ型の宣言を処理する必要があるため、そのデータ型がホスト変数として指定されてもかまいません。実際にファイル *my_header.h* を組み込んで、*another_name* の宣言内で VC_LEN に 20 を代入するのは C コンパイラのプリプロセッサの仕事です。

プリプロセッサ宣言文

Pro*C/C++ がサポートするプリプロセッサ宣言文は次のとおりです。

- **#define:** プリコンパイラおよび C または C++ のコンパイラが使用するためのマクロを作成します。
- **#include:** プリコンパイラが使用する他のソース・ファイルを読み込みます。
- **#if:** 定数式が 0 に評価される場合にのみ、ソース・テキストをプリコンパイルおよびコンパイルします。
- **#ifdef:** 定義済定義の有無に応じてソーステキストをプリコンパイルおよびコンパイルします。
- **#ifndef:** ソーステキストを条件付きで実行します。
- **#endif:** **#if** コマンド、**#ifdef** コマンドまたは **#ifndef** コマンドを終了します。
- **#else:** **#if**、**#ifdef** または **#ifndef** の条件が満たされない場合に、プリコンパイルおよびコンパイルされる代替ソース・テキストを選択します。
- **#elif:** 定数またはマクロ引数の値に応じて、プリコンパイルおよびコンパイルされる代替ソース・テキストを選択します。

無視される宣言文

Pro*C/C++ プリプロセッサが使用しない C プリプロセッサ宣言文もあります。これらの宣言文のほとんどは、プリコンパイラとは無関係です。たとえば **#pragma** は C コンパイラに対する宣言文です。プリコンパイラは処理をしません。プリコンパイラが処理しない C プリプロセッサの宣言文は、次のとおりです。

- **#:** プリプロセッサのマクロ・パラメータを文字列定数に変換します。
- **##:** 2 つのプリプロセッサ・トークンを 1 つのマクロ定義にマージします。

- **#error**: コンパイル時にエラー・メッセージを生成します。
- **#pragma**: 処理系に依存する情報を C コンパイラに渡します。
- **#line**: C コンパイラ・メッセージに行番号を提供します。

C コンパイラのプリプロセッサがこれらの宣言文をサポートしている場合でも、Pro*C/C++ はこれらの宣言文を使用しません。これらの宣言文のほとんどはプリコンパイラによって使用されません。コンパイラでサポートされている場合はこれらの宣言文を Pro*C/C++ プログラムで使用できますが、C または C++ コード以外の埋込み SQL 文や、プリコンパイラが提供する VARCHAR などのデータ型を使用した変数の宣言では使用できません。

ORA_PROC マクロ

Pro*C/C++ では、ORA_PROC という C プリプロセッサのマクロが事前定義されています。このマクロを使用すると、プリコンパイラがコードの不要な部分や不適切な部分を処理するのを防ぎます。プリコンパイルの時点では必要のない情報を提供する大きなヘッダー・ファイルが、アプリケーションに組み込まれている場合があります。このようなヘッダー・ファイルは、ORA_PROC マクロを使用して条件付きで除外すれば、プリコンパイラが読み込むことはありません。

次の例では、ORA_PROC マクロを使用して *irrelevant.h* ファイルを除外します。

```
#ifndef ORA_PROC
#include <irrelevant.h>
#endif
```

プリコンパイル時に ORA_PROC が定義されているため、*irrelevant.h* ファイルは組み込まれません。

ORA_PROC マクロは、**#ifdef** や **#ifndef** などの C プリプロセッサ宣言文に対してのみ使用できます。EXEC ORACLE 条件文は、C プリプロセッサのマクロと同じ名前領域を共有しません。したがって、次の例の条件は事前定義済の ORA_PROC マクロを使用しません。

```
EXEC ORACLE IFNDEF ORA_PROC;
    <section of code to be ignored>
EXEC ORACLE ENDIF;
```

この条件付きコード部分が正常に処理されるには、DEFINE オプションまたは EXEC ORACLE DEFINE 文を使用して ORA_PROC を設定する必要があります。

ヘッダー・ファイルの格納場所の指定

それぞれのシステムの Pro*C/C++ プリコンパイラはプリプロセッサで読み込まれるヘッダー・ファイル、たとえば *sqlca.h*、*oraca.h* および *sqlda.h* が標準的な場所にあると仮定します。たとえば、一般的な UNIX システムにおける標準的な場所は、`$ORACLE_HOME/precomp/public` です。使用しているシステムでのデフォルトの格納場所については、各システムの Oracle マニュアルを参照してください。組み込む必要のあるヘッ

ダー・ファイルがデフォルトの格納場所がない場合、コマンドラインに、または EXEC ORACLE のオプションとして、INCLUDE= オプションを指定する必要があります。プリコンパイラのオプション、EXEC ORACLE オプションの詳細は、[第 10 章「プリコンパイラのオプション」](#)を参照してください。

stdio.h、*iostream.h* などのシステム・ヘッダー・ファイルの格納場所として、Pro*C/C++ にハードコードされているところと違う場所を指定するには、SYS_INCLUDE プリコンパイラ・オプションを使用します。詳細は、[第 10 章「プリコンパイラのオプション」](#)を参照してください。

プリプロセッサの例

#define コマンドを使用して定数を作成できます。ソース・コードの "マジック・ナンバー" のかわりに使用します。VARCHAR[const] など、プリコンパイラに必要な宣言に対して、**#define** で指定した定数を使用できます。たとえば、次のコードは潜在的にバグを含む可能性があります。

```
...
VARCHAR emp_name[10];
VARCHAR dept_loc[14];
...
...
/* much later in the code ... */
f42()
{
    /* did you remember the correct size? */
    VARCHAR new_dept_loc[10];
    ...
}
```

このコードのかわりに、次のように記述できます。

```
#define ENAME_LEN    10
#define LOCATION_LEN 14
VARCHAR new_emp_name[ENAME_LEN];
...
/* much later in the code ... */
f42()
{
    VARCHAR new_dept_loc[LOCATION_LEN];
    ...
}
```

引数を持つプリプロセッサ・マクロは、C オブジェクトで使用する場合と同じようにして、プリコンパイラが処理するオブジェクトで使用できます。たとえば、次のとおりです。


```

#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)];
    ...
}

```

#include、**#ifdef** および **#endif** プリプロセッサ宣言文を使用して、プリコンパイラに必要なファイルを条件付きで含めることができます。たとえば、次のとおりです。

```

#ifdef ORACLE_MODE
#include <sqlca.h>
#else
    long SQLCODE;
#endif

```

#define の使用

Pro*C/C++ での **#define** プリプロセッサ宣言文の使用には制限があります。**#define** 宣言文を使用して実行 SQL 文の中で使用する記号定数を生成することはできません。次の無効な例にこれを示します。

```

#define RESEARCH_DEPT 40
...
EXEC SQL SELECT empno, sal
    INTO :emp_number, :salary /* host arrays */
    FROM emp
    WHERE deptno = RESEARCH_DEPT; /* INVALID! */

```

#define したマクロを有効に使用できる宣言 SQL 文は TYPE 文と VAR 文のみです。したがって、次のマクロ使用例は Pro*C/C++ で有効です。

```

#define STR_LEN      40
...
typedef char asciiz[STR_LEN];
...
EXEC SQL TYPE asciiz IS STRING(STR_LEN) REFERENCE;
...
EXEC SQL VAR password IS STRING(STR_LEN);

```

他のプリプロセッサの制限

プリプロセッサでは、宣言文 **#** および **##** を無視して、プリコンパイラが認識する必要があるトークンを作成します。もちろんこれらのコマンドは（C コンパイラのプリプロセッサがサポートしている場合）純粋な C コードの中では使用できます。プリコンパイラはこれらのコードを処理しません。次の例の場合、プリプロセッサ・コマンド **##** の使用は無効になります。

```
#define MAKE_COL_NAME(A)      col ## A
...
EXEC SQL SELECT MAKE_COL_NAME(1), MAKE_COL_NAME(2)
      INTO :x, :y
      FROM table1;
```

プリコンパイラは **##** を無視するので、この例は無効です。

#include に使用することができない SQL 文

Pro*C/C++ プリコンパイラが **#include** 宣言文を処理する方法は前の項で説明しましたが、この方法のために、**#include** 宣言文を使用して埋込み SQL 文の入ったファイルを組み込むことはできません。**#include** を使用して、純粋に宣言文と宣言文、たとえば *sqlca.h* 内のような **#define** やプリコンパイラが必要とする変数、構造体の宣言しか含まないファイルを組み込みます。

SQLCA、ORACA および SQLDA の組み込み

C/C++ プリプロセッサの **#include** コマンドまたはプリコンパイラの EXEC SQL INCLUDE コマンドを使用すると、*sqlca.h*、*oraca.h* および *sqlda.h* の各宣言ヘッダー・ファイルを Pro*C/C++ プログラムに入れることができます。これらのヘッダー・ファイルの内容に関する完全な情報は、[第 9 章「ランタイム・エラーの処理」](#)を参照してください。たとえば、次の文のように、EXEC SQL オプションを指定して SQL 通信領域構造体（SQLCA）をプログラムに含めることができます。

```
EXEC SQL INCLUDE sqlca;
```

C/C++ プリプロセッサ宣言文を使用して SQLCA を組み込む場合は、次のコードを追加します。

```
#include <sqlca.h>
```

プリプロセッサ **#include** 宣言文を使用する場合は、必ずファイル拡張子（*.h* など）を指定してください。

注意：`#include` 宣言文を使用して複数箇所に `SQLCA` を組み込む必要がある場合には、`#include` の前に宣言文 `#undef SQLCA` を置いてください。`sqlca.h` は次の行から開始されます。

```
#ifndef SQLCA
#define SQLCA 1
```

そして次に、`SQLCA` が定義されていない場合のみ、`SQLCA` 構造体を宣言します。

`#include` 宣言文または `EXEC SQL INCLUDE` 文を含むファイルをプリコンパイルする場合は、組み込まれるすべてのファイルの位置をプリコンパイラに対して指定する必要があります。コマンドラインまたはシステム構成ファイル内、ユーザー構成ファイル内で、`INCLUDE=` オプションを使用することができます。`INCLUDE` プリコンパイラ・オプション、組み込まれたファイルの検索手順、構成ファイルの詳細は、[第 10 章「プリコンパイラのオプション」](#)を参照してください。

`sqlca.h`、`oraca.h` および `sqlda.h` などの標準的なプリプロセッサ・ヘッダー・ファイルのデフォルト位置はプリコンパイラに埋め込まれています。位置はシステムによって変わります。使用しているシステムでのデフォルトの格納場所は、各システムの Oracle マニュアルを参照してください。

Pro*C/C++ が生成する `.c` 出力ファイルをコンパイルする場合、コンパイラおよびオペレーティング・システムが提供するオプションを使用して、`#include` を実行して組み込まれるファイルの格納場所を識別する必要があります。

たとえば、ほとんどの UNIX システムでは、次のコマンドを使用して、生成される C ソース・ファイルをコンパイルできます。

```
cc -o progname -I$ORACLE_HOME/sqllib/public ... filename.c ...
```

VAX/OPENVMS システム上では、インクルード・ディレクトリ・パスを論理 `VAXC$INCLUDE` に事前に設定しておきます。

EXEC SQL INCLUDE および #include の要約

プログラム中で `EXEC SQL INCLUDE` 文を使用する場合、プリコンパイラはソース・テキストを出力（`.c`）ファイル内に組み込みます。したがって、`EXEC SQL INCLUDE` を使用して組み込んだファイル中に、宣言文および実行可能な埋込み `SQL` 文を入れることができます。

`#include` を使用してファイルを組み込む場合、プリコンパイラは単にファイルを読み込み、`#define` で定義したマクロを追跡し記録するのみです。

警告：`VARCHAR` 宣言と `SQL` 文は `#include` されたファイルには使用できません。このため、Pro*C/C++ プリプロセッサの `#include` 宣言文を使用して組み込んだファイルに `SQL` 文を入れることはできません。

定義済マクロ

C コンパイラのコマンドラインでマクロを定義している場合、アプリケーションの要件によってはそのマクロをプリコンパイラのコマンドラインでも定義する必要がある場合があります。たとえば UNIX のコマンドラインで次のようなコンパイルをします。

```
cc -DDEBUG ...
```

この場合は、`DEFINE=` オプションを使用してプリコンパイルする必要があります。つまり、次のようになります。

```
proc DEFINE=DEBUG ...
```

インクルード・ファイル

プリコンパイルを必要とするすべてのインクルード・ファイルの位置は、コマンドラインまたは構成ファイル内で指定する必要があります。（プリコンパイラのオプションと構成ファイルの詳細は、10-24 ページの「[INCLUDE](#)」を参照してください。）

たとえば、UNIX 環境で開発していて、アプリケーションがディレクトリ `/home/project42/include` にインクルード・ファイルを入れている場合、Pro*C/C++ コマンドラインおよび `cc` コマンドライン上の両方でこのディレクトリを指定する必要があります。次のようなコマンドを使用します。

```
proc iname=my_app.pc include=/home/project42/include ...  
cc -I/home/project42/include ... my_app.c
```

または適切なマクロを *makefile* に組み込みます。Pro*C/C++ アプリケーションのコンパイルとリンクに関する詳細は各システムの Oracle マニュアルを参照してください。

プリコンパイル済のヘッダー・ファイル

プリコンパイル済のヘッダー・ファイルを使用すると、`#include` 文が多いヘッダー・ファイルをプリコンパイルすることで、時間とリソースを節約できます。この機能を使用する場合、次の 2 つの手順が実行されます。

- まず、プリコンパイル済のヘッダー・ファイルが作成されます。
- このプリコンパイル済のヘッダーが、次回以降のアプリケーションのプリコンパイル時に自動的に使用されます。

この機能は、多くのモジュールで構成される大きなアプリケーションで使用してください。

プリコンパイラ・オプションを `HEADER=hdr` に設定すると、次のように指定されます。

- プリコンパイル済のヘッダーを使用します。
- 生成される出力ファイルのファイル拡張子は `hdr` です。

このオプションを入力できるのは、構成ファイルまたはコマンドラインのみです。HEADERにはデフォルト値がありません。ただし、入力ヘッダーの拡張子は、*h*である必要があります。

プリコンパイル済のヘッダー・ファイルの作成

`top.h` というヘッダー・ファイルを仮定します。HEADER= `hdr` を次のように指定してそのファイルをプリコンパイルします。

```
proc HEADER=hdr INAME=top.h
```

注意：拡張子は、`'h'` です。INAME 値には、`'/'`、`'.'` などの絶対パス要素または相対パス要素は使用できません。

Pro*C/C++ により、入力ファイル `top.h` がプリコンパイルされ、同じディレクトリに新しいプリコンパイル済のヘッダー・ファイル `top.hdr` が生成されます。出力ファイル `top.hdr` は、`#include` 文が検索を行うディレクトリに移動できます。

注意：ONAME オプションを使用して出力ファイルに名前を付けないでください。この場合、HEADER とともに使用しても無視されます。

プリコンパイル済のヘッダー・ファイルの使用

HEADER オプションの値には、プリコンパイルされるアプリケーションと同じ値を使用してください。`simple.pc` に次のファイルが含まれ、

```
#include <top.h>
...
```

`top.h` に次のファイルが含まれる場合は、

```
#include <a.h>
#include <b.h>
#include <c.h>
...
```

次の方法でプリコンパイルします。

```
proc HEADER=hdr INAME=simple.pc
```

Pro*C/C++ によって `#include top.h` 文が読み込まれると、対応する `'top.hdr'` ファイルが検索され、`'top.h'` を再度プリコンパイルするかわりにそのファイルからデータがインスタンシエートされます。

注意：プリコンパイルされたヘッダー・ファイルは、常に入力ヘッダー・ファイルのかわりに使用されます。入力 (`.h`) ファイルがインクルード・ディレクトリの標準検索階層の最初に表示されている場合も同様です。

例

冗長なファイル・インクルード

ケース 1: 最上位のヘッダー・ファイルのインクルード

プリコンパイル済のヘッダー・ファイルは、`#include` 宣言文を使用してインクルードされた回数にかかわらず、1 度しかインスタシエートできません。

前の例と同様に、HEADER の値を 'hdr' に指定して、最上位のヘッダー・ファイル `top.h` をプリコンパイルすると仮定します。次に、そのヘッダー・ファイルに対して、複数の `#include` 宣言文をプログラムに記述します。

```
#include <top.h>
#include <top.h>
main() {}
```

`top.h` の最初の `#include` が読み込まれると、プリコンパイル済みのヘッダー・ファイル `top.hdr` がインスタシエートされます。同じヘッダー・ファイルの 2 番目の `#include` は、冗長であるため無視されます。

ケース 2: ネストされたヘッダー・ファイルのインクルード

ファイル `a.h` に、次の文が含まれていると仮定します。

```
#include <b.h>
```

前の例と同様に HEADER を指定し、そのヘッダー・ファイルをプリコンパイルします。Pro*C/C++ によって、`a.h` および `b.h` がプリコンパイルされ、`a.hdr` が生成されます。

次に、この Pro*C/C++ プログラムをプリコンパイルします。

```
#include <a.h>
#include <b.h>
main() {}
```

`a.h` の `#include` が読み込まれると、`a.h` が再度プリコンパイルされるかわりに、プリコンパイル済のヘッダー・ファイル `a.hdr` がインスタシエートされます。このインスタシエーションには、`b.h` のファイルもすべて含まれます。

`b.h` は `a.h` のプリコンパイルに含まれ、`a.hdr` はインスタシエートされているので、プログラムに指定されている `b.h` の後続の `#include` は冗長になり、無視されます。

複数のプリコンパイル済ヘッダー・ファイル

Pro*C/C++ では、1 回のプリコンパイルで複数の異なるプリコンパイル済のヘッダー・ファイルをインスタシエートすることができます。ただし、複数のプリコンパイル済ヘッ

ダー・ファイルが共通のヘッダー・ファイルを共有している場合、次の点に注意してください。

たとえば、topA.h に次の行が含まれ、

```
#include <a.h>
#include <c.h>
```

topB.h に次の行が含まれていると仮定します。

```
#include <b.h>
#include <c.h>
```

topA.h および topB.h では、共通のヘッダー・ファイル c.h がインクルードされています。topA.h および topB.h を同じ HEADER 値でプリコンパイルすると、topA.hdr および topB.hdr が生成されますが、両方に c.h のすべての内容が含まれています。

次のような Pro*C/C++ プログラムがあると仮定します。

```
#include <topA.h>
#include <topB.h>
main() {}
```

プリコンパイル済ヘッダー・ファイル topA.hdr および topB.hdr は、前の例と同様にインスタンスエートされます。ただし、これらのヘッダー・ファイルでは、ヘッダー・ファイル c.h を共有しているため、属しているファイルは2回インスタンスエートされます。

Pro*C/C++ では、プリコンパイル済ヘッダー・ファイル間のファイルの共有を判断できません。各プリコンパイル済ヘッダー・ファイルには、固有のヘッダー・セットをインクルードします。ファイルの共有はできる限り避けてください。共有すると、プリコンパイル速度の低下およびメモリー使用量の増加の原因となり、プリコンパイル済のヘッダー・ファイルを使用する意味がなくなります。

オプションの効果

アプリケーションのプリコンパイル時に、次のプリコンパイラ・オプションを使用することができます。

DEFINE および INCLUDE オプション

プリコンパイル済ヘッダーを使用してプリコンパイルするときは、DEFINE および INCLUDE の値を、プリコンパイル済のヘッダー・ファイルの作成時と同じ値にする必要があります。DEFINE および INCLUDE の値を変更した場合は、プリコンパイル済のヘッダー・ファイルを再作成する必要があります。

開発環境を変更した場合も、プリコンパイル済のヘッダー・ファイルを再作成する必要があります。

シングル・ユーザーの場合

シングル・ユーザーの場合を考えてみます。DEFINE または INCLUDE オプションの値を変更した場合、プリコンパイル済のヘッダー・ファイルの内容は、その後に実行される Pro*C/C++ プリコンパイルでは正常に使用できなくなります。

DEFINE または INCLUDE オプションの値を変更したため、プリコンパイル済のヘッダー・ファイルの内容は、`#include` 宣言文の対応する `.h` ファイルが正常にプリコンパイルされた場合の標準的な結果と一致しなくなります。

つまり、DEFINE または INCLUDE オプションの値を変更した場合、プリコンパイル済のヘッダー・ファイルをすべて再作成し、そのファイルを使用する Pro*C/C++ プログラムを再度プリコンパイルする必要があります。

詳細は、10-17 ページの「[DEFINE](#)」および 10-24 ページの「[INCLUDE](#)」を参照してください。

マルチ・ユーザーの場合

A および B という、2 人のユーザーがいる場合について考えてみます。A と B はまったく別の環境で開発したため、DEFINE および INCLUDE オプションの値もまったく異なっています。

ユーザー A は、共通ヘッダー・ファイル `common.h` をプリコンパイルして、プリコンパイル済のヘッダー・ファイル `common.hdrA` を作成しました。ユーザー B も同じヘッダー・ファイルをプリコンパイルして、`common.hdrB` を作成しました。しかし、ユーザー A とユーザー B の環境が異なるため、2 人のユーザーが使用した DEFINE および INCLUDE オプションの値も異なります。ユーザー A と B が作成した `common.hdr` の内容が同じになることは限りません。

コードは次のようになります。

```
A> proc HEADER=hdrA DEFINE=<A macros> INCLUDE=<A dirs> common.h
```

```
B> proc HEADER=hdrB DEFINE=<B macros> INCLUDE=<B dirs> common.h
```

異なる環境で作成されたため、生成されたプリコンパイル済のヘッダー・ファイル `common.hdrA` は、`common.hdrB` と同等でない場合があります。つまり、ユーザー A とユーザー B が、他のユーザーによって作成された `common.hdr` を使用してプリコンパイルしても、それぞれの開発環境で、Pro*C/C++ プログラムが正常にプリコンパイルされるとは限りません。

このため、プリコンパイル済のヘッダー・ファイルを、異なるユーザー間および異なる開発環境間で共有または交換する場合には注意が必要です。

CODE および PARSE オプション

Pro*C/C++ では、`hpp`、`h++` などの拡張子が付いた C++ ヘッダー・ファイルは検索されません。このため、ヘッダー・ファイルのプリコンパイル時には、`CODE=CPP` を使用しないで

ください。ソース・コードに *h* ヘッダー・ファイルのみが含まれる場合に限り、アプリケーションのプリコンパイル時に CPP 値を使用できます。10-13 ページの「[CODE](#)」を参照してください。

プリコンパイル済のヘッダー・ファイルの作成時またはモジュールのプリコンパイル時には、PARSE オプションの値として FULL または PARTIAL 以外は使用できません。値 FULL は、PARTIAL よりも高い値とみなされます。モジュールのプリコンパイル時に使用する PARSE の値は、プリコンパイル済のヘッダー・ファイルの作成時以下にする必要があります。

注意： PARSE=FULL を指定してプリコンパイル済のヘッダー・ファイルをプリコンパイルしてから、PARSE=PARTIAL を指定してモジュールをプリコンパイルする場合は、ホスト変数を宣言文の中で宣言しておく必要があります。C++ コードは、PARSE=PARTIAL を指定した場合にのみ読み込まれます。PARSE オプションの詳細は、12-4 ページの「[コードの解析](#)」および 10-34 ページの「[PARSE](#)」を参照してください。

PARTIAL に次の PARSE オプションを指定してヘッダー・ファイルをプリコンパイルすると仮定します。

```
proc HEADER=hdr PARSE=PARTIAL file.h
```

次に、PARSE に FULL を指定し、そのヘッダー・ファイルを含むプログラムをプリコンパイルします。

```
proc HEADER=hdr PARSE=FULL program.pc
```

file.h は、PARSE オプションに PARTIAL を指定してプリコンパイルしたため、一部のヘッダー・ファイルは処理されません。このため、未処理部分が参照された場合、Pro*C/C++ プログラムのプリコンパイル時にエラーが発生する可能性があります。

具体例として、file.h に次のコードが含まれ、

```
#define LENGTH 10
typedef int myint;
```

program.pc に、次の小さなプログラムが含まれると仮定します。

```
#include <file.h>
main()
{
    VARCHAR ename[LENGTH];
    myint empno = ...;
    EXEC SQL SELECT ename INTO :ename WHERE JOB = :empno;
}
```

file.h のプリコンパイル時に PARSE オプションに PARTIAL が指定されているため、typedef は処理されずに LENGTH マクロのみが処理されます。

VARCHAR の宣言および宣言以後のホスト変数としての使用は、正常に行われます。ただし、Pro*C/C++ では myint 型宣言が処理されないため、empno ホスト変数を使用できません。

PARSE オプションに FULL を指定してヘッダー・ファイルをプリコンパイルしてから、PARSE オプションに PARTIAL を指定してプログラムをプリコンパイルすると、正常に動作します。ただし、ホスト変数は、明示的な宣言文の内部で宣言する必要があります。

使用上の注意

プリコンパイル済のヘッダーから生成された出力ファイルの形式は、次のリリースでは異なることがあります。Pro*C/C++ では、プリコンパイル済のヘッダー・ファイルの出力ファイルの生成に使用されたプリコンパイラのバージョンを識別できません。

このため、プリコンパイル済のヘッダー・ファイルを使用したプリコンパイル時に、エラーまたは他の予期しない動作を回避するために、Pro*C/C++ のバージョンのアップグレード時には、対応するヘッダー・ファイルを再度プリコンパイルしてファイルを再生成することをお勧めします。

ヘッダー・ファイルをプリコンパイルして生成された出力ファイルには、移植性がありません。つまり、ヘッダー・ファイルのプリコンパイルによって生成された出力ファイルを、プラットフォーム間で転送したり、生成後に別のヘッダー・ファイルまたは Pro*C/C++ プログラムをプリコンパイルしているときにそのファイルを使用することはできません。

Oracle プリプロセッサ

コードの条件文は、環境および処理を定義する EXEC ORACLE 宣言文によってマークされます。これらの条件文には、C の文、および埋込み SQL 文と宣言文を記述できます。次の EXEC ORACLE 宣言文で、プリコンパイルの条件を制御します。

```
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
```

EXEC ORACLE 文の終わりには、必ずセミコロンを付けてください。

記号の定義

記号を定義するには 2 通りの方法があります。次の文をインクルードします。

```
EXEC ORACLE DEFINE symbol;
```

次の構文を使用してコマンドラインで記号を定義します。

```
... INAME=filename ... DEFINE=symbol
```

このとき *symbol* の部分は 大 / 小文字区別がありません。

警告：#define プリプロセッサ宣言文は EXEC ORACLE DEFINE コマンドと同一のものではありません。

Pro*C/C++ プリコンパイラをシステムにインストールするときに、ポート固有の記号がいくつか事前定義されます。

Oracle プリプロセッサの例

次の例では、記号 *site2* が定義されているときのみ SELECT 文がプリコンパイルされます。

```
EXEC ORACLE IFDEF site2;
      EXEC SQL SELECT DNAME
              INTO :dept_name
              FROM DEPT
              WHERE DEPTNO = :dept_number;
EXEC ORACLE ENDIF;
```

次の例に示すように条件ブロックはネストできます。

```
EXEC ORACLE IFDEF outer;
      EXEC ORACLE IFDEF inner;
      ...
      EXEC ORACLE ENDIF;
EXEC ORACLE ENDIF;
```

C または埋込み SQL コードは IFDEF と ENDIF の間に置いて、そのシンボルを定義しないことで「コメント・アウト」できます。

数値定数の評価

以前の Pro*C/C++ では、ホスト変数 (char、VARCHAR など) のサイズの宣言で数値リテラルや数値リテラルを含む単純な定数式を指定できました。その例を示します。

```
#define LENGTH 10
VARCHAR v[LENGTH];
char c[LENGTH + 1];
```

現在は次のような数値定数の宣言も使用することができます。

```
const int length = 10;
VARCHAR v[length];
char c[length + 1];
```

このような定数宣言をサポートする ANSI コンパイラや C++ コンパイラを使用しているプログラマにとって、この機能は最適です。

これまでの Pro*C/C++ では、評価可能な定数式での定数の評価は実行されていましたが、数値定数はどのような定数式にも宣言できませんでした。

Pro*C/C++ では、マクロが数値リテラルに展開されていれば、通常の数値リテラルやマクロを使用する位置であれば、どこでも数値定数を宣言できます。

これが主に使用されるのは、SQL 文で使用するバインド変数の配列のサイズを宣言する場合です。

Pro*C/C++ での数値定数の使用

Pro*C/C++ では、数値定数が宣言された位置を検索する場合は、C の標準の有効範囲規則に従います。

```
const int g = 30;      /* Global declaration to both function_1()
                        and function_2() */

void function_1()
{
    const int a = 10; /* Local declaration only to function_1() */
    char x[a];
    exec sql select ename into :x from emp where job = 'PRESIDENT';
}

void function_2()
{
    const int a = 20; /* Local declaration only to function_2() */
    VARCHAR v[a];
    exec sql select ename into :v from emp where job = 'PRESIDENT';
}

void main()
{
    char m[g];          /* The global g */
    exec sql select ename into :m from emp where job = 'PRESIDENT';
}
```

数値定数の規則および例

特定の静的な型を持つ変数は、**static** で定義し、初期化する必要があります。Pro*C/C++ で数値定数を宣言する場合は、必ず次の規則に従ってください。

- 定数の宣言時に **const** 修飾子を指定します。
- 定数値の初期化時に初期化指定子を指定します。初期化指定子は必ずプリコンパイル時に評価可能である必要があります。

有効な初期化指定子が指定された定数宣言で解決できない識別子を使用すると、エラーとみなされます。

次の例に、無効な指定方法とその指定が許可されない理由を示します。

```
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]; /* Missing initializer */
VARCHAR v6[f]; /* Bad initializer.. b is not a constant */
```

OCI リリース 8 の SQLLIB 拡張相互運用性

OCI 環境ハンドルは、*sql_context* 型の Pro*C/C++ ランタイム・コンテキストに関連付けられます。つまり、アプリケーション実行時に SQLLIB に保存されている Pro*C/C++ ランタイム・コンテキストは、複数の OCI 環境ハンドルと関連付けることはできません。ランタイム・コンテキスト単位に、複数のデータベース接続を行うことができ、各ランタイム・コンテキストは、その OCI 環境ハンドルに関連付けられます。

ランタイム・コンテキストと OCI リリース 8 環境の確立と終了

EXEC SQL CONTEXT USE 文には、Pro*C/C++ プログラムで使用するランタイム・コンテキストを指定します。このコンテキストは、所定の Pro*C/C++ ファイル内で次に指定された EXEC SQL CONTEXT USE 文までそのコンテキストの後に続く実行 SQL 文すべてに適用されます。ソース・ファイルに EXEC SQL CONTEXT USE が現れない場合は、デフォルトの「グローバル」コンテキストが仮定されます。このように、カレント・ランタイム・コンテキストと、それに関連付けられているカレント OCI 環境ハンドルは、プログラム内のどの場所にあっても特定できます。

Pro*C/C++ では EXEC SQL CONNECT 文を使用してデータベースへのログインが実行されると、ランタイム・コンテキストとそれに関連付けられている OCI 環境ハンドルが初期化されます。

EXEC SQL CONTEXT FREE 文を使用して Pro*C/C++ ランタイム・コンテキストを解放すると、それに関連付けられている OCI 環境ハンドルが終了し、すべてのリソース（各 OCI ハンドルおよび各 LOB ロケータのための領域など）の割当てが解除されます。このコマンドは、Pro*C/C++ ランタイム・コンテキストに関連付けられているその他のメモリーもすべて解放します。デフォルトの「グローバル」実行時に確立される OCI 環境ハンドルは Pro*C/C++ プログラムが終了するまで割り当てられたまま残ります。

OCI リリース 8 環境ハンドルのパラメータ

Pro*C/C++ で確立されている OCI 環境では、次のパラメータが使用されます。

- メモリーの割当て、メモリーの解放、テキスト・ファイルへの書き込みおよび出力バッファのフラッシュに対してその環境で使用されるコールバック関数は、それぞれ `malloc()`、`free()`、`fprintf(stderr,...)` および `fflush(stderr)` をコールする通常の関数です。
- 言語は、NLS 環境変数 `NLS_LANG` から取得されます。
- エラー・メッセージ・バッファは、スレッドに固有の記憶域に割り当てられます。

OCI リリース 8 とのインタフェース

SQLLIB ライブラリでは、Pro*C/C++ プログラムで確立されているデータベース接続に対して OCI 環境ハンドルおよびサービス・コンテキスト・ハンドルを取得するためのルーチンがいくつか用意されています。OCI ハンドルを取得すると、ユーザーは様々な OCI ルーチンをコールすることができます。たとえばクライアント側で日付算術を実行したり、オブジェクトに対してナビゲーション操作を行ったりできます。詳細は、[第 17 章「オブジェクト」](#)を参照してください。これらの SQLLIB 関数は次で説明します。これらの関数のプロトタイプはパブリック・ヘッダー・ファイル `sql2oci.h` 内に用意してあります。

埋込み SQL と他の Oracle プログラム・インタフェースのコールを混在させる Pro*C/C++ ユーザーは十分に注意する必要があります。たとえば、ユーザーが OCI インタフェースを使用して直接接続を終了すると、SQLLIB で同期のとれていない状態が発生します。このような場合、Pro*C/C++ プログラム内の後続の SQL 文の動作は未定義になります。

リリース 8.0 からは、Oracle8 OCI との相互操作性を提供する次の新しい SQLLIB 関数が、ヘッダー・ファイル `sql2oci.h` 内で宣言されています。

- `SQLEnvGet()` 所定の SQLLIB ランタイム・コンテキストに関連付けられている OCI 環境ハンドルへのポインタを戻します。シングルスレッド環境とマルチスレッド環境の両方で使用できます。
- `SQLSvcCtxGet()` Pro*C/C++ データベース接続用の OCI サービス・コンテキスト・ハンドルを戻します。シングルスレッド環境とマルチスレッド環境の両方で使用できます。
- シングルスレッド・ランタイム・コンテキストを使用するときに、いずれかの関数の最初のパラメータとして `sql2oci.h` をインクルードするときは、`(dvoid *)0` として定義された定数 `SQL_SINGLE_RCTX` を渡します。

SQLEnvGet()

SQLLIB ライブラリ関数 `SQLEnvGet()` (SQLLIB による OCI 環境の取得) を指定すると、所定の SQLLIB ランタイム・コンテキストに関連付けられている OCI 環境ハンドルへのポインタが戻されます。次は、この関数のプロトタイプです。

```
sword SQLEnvGet(dvoid *rctx, OCIEnv **oeh);
```

パラメータは次のとおりです。

説明	<i>oeh</i> をランタイム・コンテキストに対応している OCIEnv に設定します。
パラメータ	<i>rctx</i> (IN) SQLLIB ランタイム・コンテキストへのポインタ。 <i>oeh</i> (OUT) OCIEnv へのポインタ。
戻り値	成功した場合は SQL_SUCCESS。 失敗した場合は SQL_ERROR。
注意	Pro*C/C++ での通常のエラー・ステータス変数 (SQLCA、SQLSTATE など) は、この関数をコールしても影響を受けません。

SQLSvcCtxGet()

SQLLIB ライブラリ関数 *SQLSvcCtxGet()* (SQLLIB による OCI サービス・コンテキストの取得) を指定すると、Pro*C/C++ データベース接続用の OCI サービス・コンテキストが戻されます。この後、OCI サービス・コンテキストを使用して、OCI 関数を直接コールできます。次は、この関数のプロトタイプです。

```
sword SQLSvcCtxGet(dvoid *rctx, text *dbname,
                   sb4 dbnamelen, OCISvcCtx **svc);
```

パラメータは次のとおりです。

説明	<i>svc</i> をランタイム・コンテキストに対応している OCI サービス・コンテキストに設定します。
パラメータ	<i>rctx</i> (IN) = SQLLIB ランタイム・コンテキストへのポインタ。 <i>dbname</i> (IN) = この接続の「論理」名を含んだバッファ。 <i>dbnamelen</i> (IN) = <i>dbname</i> バッファの長さ。 <i>svc</i> (OUT) = OCISvcCtx ポインタのアドレス。
戻り値	成功した場合は SQL_SUCCESS。 失敗した場合は SQL_ERROR。

注意

1. Pro*C/C++ での通常のエラー・ステータス変数 (SQLCA、SQLSTATE など) は、この関数をコールしても影響を受けません。
2. *dbname* は、埋込み SQL 文に指定された AT 句で使用されている識別子と同じもので構成されます。
3. *dbname* が NULL ポインタで構成されていたり、*dbnamelen* に 0 が指定されていると、SQL 文に AT 句が指定されていない場合と同様に、デフォルトのデータベース接続とみなされます。
4. *dbnamelen* に -1 が指定されると、*dbname* はゼロ終了記号付き文字列で構成されていることになります。

OCI コールの埋込み

OCI リリース 8 コールを Pro*C/C++ プログラムに埋め込む手順は、次のとおりです。

1. パブリック・ヘッダー `sql2oci.h` をインクルードします。
2. 環境ハンドル (type `OCIEnv *`) を Pro*C/C++ プログラムで宣言します。

```
OCIEnv *oeh;
```

3. コールする OCI ファンクションが `ServiceContext` ハンドルを必要とする場合、オプションとして、サービス・コンテキスト・ハンドル (type `OCISvcCtx *`) を Pro*C/C++ プログラムで宣言します。

```
OCISvcCtx *svc;
```

4. エラー・ハンドル (type `OCIError *`) を Pro*C/C++ プログラムで宣言します。

```
OCIError *err;
```

5. 埋込み SQL 文 `CONNECT` を使用して Oracle に接続します。接続には、OCI を使用しないでください。

```
EXEC SQL CONNECT ...
```

6. 希望するランタイム・コンテキストに伴う OCI 環境ハンドルを `SQLEnvGet` ファンクションを使用して取得します。

シングルスレッド・アプリケーションの場合

```
retcode = SQLEnvGet(SQL_SINGLE_RCTX, &oeh);
```

マルチスレッド・アプリケーションの場合

```
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. 取り出した環境ハンドルを使用して OCI エラー・ハンドルを割り当てます。

```
retcode = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
                          (ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0);
```

8. オプションとして、使用中の OCI コールを必要とする場合、SQLSvcCtxGet コールを使用して OCIServiceContext ハンドルを取得します。

シングルスレッド・アプリケーションの場合

```
retcode = SQLSvcCtxGet(SQL_SINGLE_RCTX, (text *)dbname, (ub4)dbnlen, &svc);
```

マルチスレッド・アプリケーションの場合

```
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);
```

注意: Pro*C/C++ 接続が AT 句によって名付けられていないと、NULL ポインタが *dbname* として渡されることがあります。

埋込み (OCI リリース 7) Oracle コール

Pro*C/C++ プログラムに OCI コールを埋め込む手順は次のとおりです。

- OCI ログイン・データ領域 (LDA) を Pro*C/C++ プログラム内に宣言します。
(MODE=ANSI を指定してプリコンパイルする場合は、宣言文の外に宣言します。)
LDA は OCI ヘッダー・ファイル *oci.h* の中で定義された構造体です。詳細は、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。
- OCI の *orlon()* または *onblon()* コールではなく、埋込み SQL 文 CONNECT を使用して Oracle8 に接続します。
- SQLLIB ランタイム・ライブラリ関数 *sqllda()* をコールして LDA.SQLLIB 関数を設定します。

こうすることで、Pro*C/C++ プリコンパイラと OCI はデータを共有しながら動作していることを「知り」ます。ただし、Oracle8 のカーソルは共有されません。

Oracle8 ランタイム・ライブラリによって接続が管理され、HDA がメンテナンスされるので、OCI ホスト・データ領域 (HDA) の宣言を意識する必要はありません。

LDA の設定

OCI コールを発行することによって、LDA を設定します。

```
sqllda(&lda);
```

このとき、*lda* は LDA データ構造を識別します。

設定が失敗すると、*lda* 内の *lda_rc* フィールドはエラーを示す 1012 に設定されます。

リモートの複数接続

sqllda() をコールすることによって、最後に実行された SQL 文で使用する接続のための LDA が設定されます。追加接続に必要な別の LDA を設定するには、それぞれの CONNECT の後に別の *lda* で *sqllda()* をコールしてください。次の例では、2 つの非デフォルトのデータベースを同時に接続しています。

```
#include <ocidfn.h>
Lda_Def lda1;
Lda_Def lda2;

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 non-default database */
EXEC SQL CONNECT :username IDENTIFIED BY :password;
      AT DB_NAME1 USING :db_string1;
/* set up first LDA */
sqllda(&lda1);
/* connect to second non-default database */
EXEC SQL CONNECT :username IDENTIFIED BY :password;
      AT DB_NAME2 USING :db_string2;
/* set up second LDA */
sqllda(&lda2);
```

DB_NAME1 および DB_NAME2 は、C の変数ではなく、SQL 識別子です。識別子 DB_NAME1 および DB_NAME2 は、2 つの非デフォルト・ノードのデフォルトのデータベースに名前を指定するためにのみに使用します。これによって SQL 文は後でデータベースを名前で参照できます。

SQLLIB パブリック関数の新しい名前

表 5-3 は、Oracle8i の SQLLIB 関数の新しい名前の一覧です。これらの SQLLIB 関数はスレッド・アプリケーションでも非スレッド・アプリケーションでも利用できます。たとえば、従来は、*sqlglm()* は、この関数の非スレッド・バージョンまたはデフォルト・コンテキスト・バージョンで、*sqlglmt()* は、スレッド・バージョンまたは非デフォルト・コンテキスト・バージョンでした。*sqlglm()* および *sqlglmt()* の名前は Oracle8 でも有効です。新しい関数 *SQLErrorGetText()* には、*sqlglmt()* などの引数が必要です。非スレッドまたはデフォルト・コンテキスト・アプリケーションでは、定義した定数 `SQL_SINGLE_RCTX` をコンテキストとして渡してください。`SQL_SINGLE_RCTX` の詳細は、5-44 ページの「[OCI リリース 8 とのインタフェース](#)」を参照してください。

標準 SQLLIB パブリック関数は、いずれもスレッドに対して安全であり、ランタイム・コンテキストを最初の引数として受け入れます。次に、*SQLErrorGetText()* の構文の例を示します。

```
void SQLErrorGetText (dvoid *context,  char  *message_buffer,
                      size_t *buffer_size,
                      size_t *message_length);
```

つまり、古い関数名は既存のアプリケーションでそのまま使用できます。新たに作成するアプリケーションでは、新しい関数名を使用できます。

表 5-3 はすべての SQLLIB パブリック関数とそれに対応する構文の一覧です。非スレッド関数またはデフォルト・コンテキストの使用法についてのクロス・リファレンスも示していますので、より詳細な説明が必要なときに参照してください。

表 5-3 SQLLIB パブリック関数 -- 新しい名前

旧名	新しい関数プロトタイプ	クロス・リファレンス
sqlalldt()	struct SQLDA *SQLSQLDAAlloc(dvoid *context, unsigned int maximum_variables, unsigned int maximum_name_length, unsigned int maximum_ind_name_length);	15-5 ページの「 SQLDA の割当て 」も参照してください。
sqlcdat()	void SQLCDAFromResultSetCursor(dvoid *context, Cda_Def *cda, void *cursor, sword *return_value);	4-29 ページの「 OCI でのカーソル変数の使用 (リリース 7 のみ) 」も参照してください。
sqlclut()	void SQLSQLDAFree(dvoid *context, struct SQLDA *descriptor_name);	15-33 ページの「 記憶域の割当て解除 」も参照してください。
sqlcurt()	void SQLCDAToResultSetCursor(dvoid *context, void *cursor, Cda_Def *cda, sword *return_value)	4-29 ページの「 OCI でのカーソル変数の使用 (リリース 7 のみ) 」も参照してください。
sqlglmt()	void SQLErrorGetText(dvoid *context, char *message_buffer, size_t *buffer_size, size_t *message_length);	9-23 ページの「 エラー・メッセージの全文の取得 」も参照してください。
sqlglst()	void SQLStmtGetText(dvoid *context, char *statement_buffer, size_t *statement_length, size_t *sqlfc);	9-32 ページの「 SQL 文のテキスト取得 」も参照してください。
sqlld2t()	void SQLLDAGetName(dvoid *context, Lda_Def *lda, text *cname, int *cname_length);	5-53 ページの「 OCI コール (リリース 7 のみ) 」も参照してください。
sqlldat()	void SQLCDAGetCurrent(dvoid *context, Lda_Def *lda);	5-48 ページの「 リモートの複数接続 」も参照してください。
sqlnult()	void SQLColumnNullCheck(dvoid *context, unsigned short *value_type, unsigned short *type_code, int *null_status);	15-16 ページの「 NULL/NOT NULL データ型の処理 」も参照してください。
sqlprct()	void SQLNumberPrecV6(dvoid *context, unsigned long *length, int *precision, int *scale);	15-15 ページの「 精度とスケールの抽出 」も参照してください。

旧名	新しい関数プロトタイプ	クロス・リファレンス
sqlpr2t()	void SQLNumberPrecV7(dvoid *context, unsigned long *length, int *precision, int *scale);	15-15 ページの「 精度とスケールの抽出 」も参照してください。
sqlvcpt()	void SQLVarcharGetLength(dvoid *context, unsigned long *data_length, unsigned long *total_length);	4-20 ページの「 VARCHAR 配列コンポーネントの長さを調べる方法 」も参照してください。
N/A	sword SQLEnvGet(dvoid *context, OCIEnv **oeh);	5-44 ページの「 SQLEnvGet() 」を参照してください。
N/A	sword SQLSvcCtxGet(dvoid *context, text *dbname, int dbnamelen, OCISvcCtx **svc);	5-45 ページ「 SQLSvcCtxGet() 」を参照してください。
N/A	void SQLRowidGet(dvoid *context, OCIRowid **urid);	4-37 ページ「 SQLRowidGet() 」を参照してください。
N/A	void SQLExtProcError(dvoid *context, char *msg, size_t msglen);	外部プロシージャで使用する場合は 7-30 ページの「 SQLExtProcError() 」を参照してください。

注意：これらの関数の引数リストで使用する特定のデータ型については、プラットフォームごとの *sqlcpr.h* ヘッダー・ファイルを参照してください。

X/Open アプリケーションの開発

X/Open アプリケーションは分散トランザクション処理（DTP）環境で動作します。抽象モデルでは、X/Open アプリケーションはリソース・マネージャ（RM）に各種のサービスの提供をコールします。たとえば、データベース・リソース・マネージャはデータベース内のデータにアクセスします。リソース・マネージャは、アプリケーションのためにすべてのトランザクションを制御するトランザクション・マネージャ（TM）と対話します。

図 5-1 仮定される DTP モデル

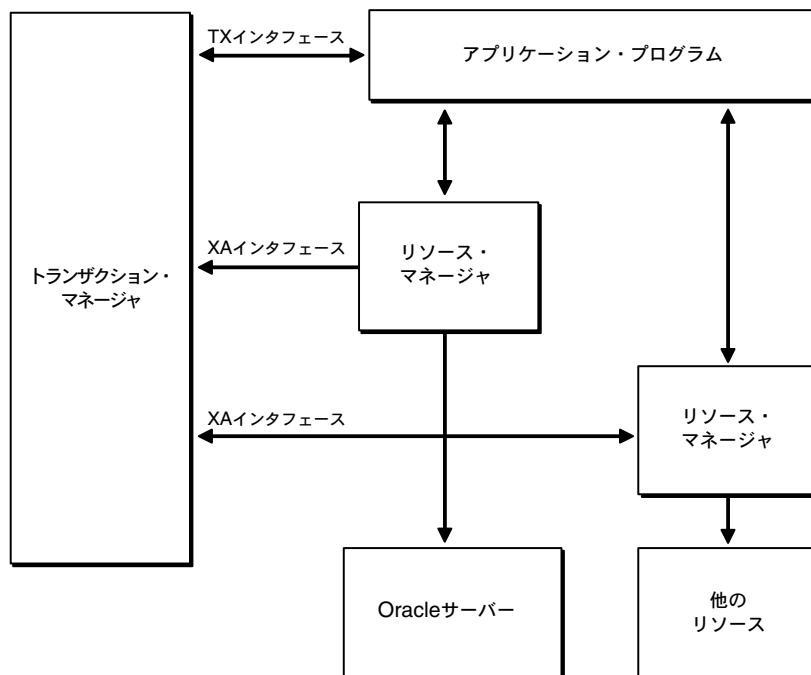


図 5-1 に DTP モデルのコンポーネントが Oracle8 データベースのデータに効率的なアクセスを行うよう相互作用できる方法のひとつを示します。この DTP モデルでは、リソース・マネージャとトランザクション・マネージャとの間に XA インタフェースが指定されています。Oracle は XA 準拠ライブラリを提供します。このライブラリは、X/Open アプリケーションにリンクする必要があります。また、アプリケーション・プログラムとリソース・マネージャ間で「ネイティブ・インタフェース」を指定する必要もあります。

DTP モデルはトランザクション・マネージャとリソース・マネージャがアプリケーション・プログラムとやりとりする方法を指定します。これは X/Open ガイドの『Distributed Transaction Processing Reference Model』と関連出版物に記載されています。これは次の宛て先に書面で要求すれば入手できます。

X/Open Company Ltd.
1010 El Camino Real, Suite 380
Menlo Park, CA 94025

XA インタフェースの使用方法は、『トランザクション・プロセッシング (TP) モニター・ユーザーガイド』を参照してください。

Oracle 固有の項目

プリコンパイラを使用して、X/Open 規格に準拠したアプリケーションを開発できます。ただし、次の必要条件を満たす必要があります。

Oracle8 への接続

X/OPEN アプリケーションはデータベースとの接続の確立および維持しません。かわりに、トランザクション・マネージャおよび XA インタフェースが Oracle によって供給され、データベースとの接続と接続の切離しを透過的に取り扱います。したがって通常、X/Open 準拠のアプリケーションは CONNECT 文を実行しません。

トランザクション制御

X/OPEN アプリケーションで、グローバル・トランザクションに影響を与える COMMIT、ROLLBACK、SAVEPOINT、SET TRANSACTION などの文を実行しないでください。たとえば、アプリケーションで COMMIT 文を実行しないでください。トランザクション・マネージャがコミットを処理するためです。さらに、アプリケーションで CREATE、ALTER、RENAME などの SQL データ定義文を実行しないでください。それらは暗黙の COMMIT を発行するためです。

アプリケーションで後続の SQL 操作を妨げるエラーが検出された場合、内部 ROLLBACK 文を実行できます。しかし、今後リリースされる XA インタフェースでは変更になる可能性があります。

OCI コール（リリース 7 のみ）

X/Open アプリケーションで OCI コールを発行するときは、ランタイム・ライブラリ・ルーチン `sqlld2()` を使用する必要があります。このルーチンは、XA インタフェースを介して確立された指定の接続のために、LDA を設定します。`sqlld2()` コールの説明については、『Oracle Call Interface Programmer's Guide, Release 7』を参照してください。

次の OCI コールは X/Open アプリケーションからは発行できないことに注意してください。OCOM、OCON、OCOF、ONBLON、ORLON、OLON、OLOGOF

OCI リリース 8 コールの Pro*C/C++ での使用方法については 5-44 ページの「[OCI リリース 8 とのインタフェース](#)」を参照してください。

リンク

XA 機能を利用するには、XA ライブラリを X/Open アプリケーション・オブジェクト・モジュールにリンクする必要があります。具体的な指示は、使用中のシステムの Oracle8 マニュアルを参照してください。

埋込み SQL

この章では、埋込み SQL プログラミングの基本的な技法およびその適用方法について説明します。この章では、次の事項について説明します。

- ホスト変数の使用
- 標識変数の使用
- 基本的な SQL 文
- DML 戻り句
- カーソルの使用方法
- オプティマイザ・ヒント
- CURRENT OF 句の使用方法
- すべてのカーソル文の使用方法
- 完全な例

ホスト変数の使用

Oracle はホスト変数を使用してデータおよびステータス情報をプログラムに引き渡します。同様にプログラムはホスト変数を使用してデータを Oracle に引き渡します。

出力ホスト変数および入力ホスト変数

ホスト変数はその使用方法によって、出力ホスト変数または入力ホスト変数と呼ばれます。

SELECT 文または FETCH 文の INTO 句内のホスト変数は、Oracle によって出力される列の値が入るので出力ホスト変数と呼ばれます。Oracle は列の値を INTO 句内の対応する出力ホスト変数に割り当てます。

SQL 文のその他のホスト変数の値は、プログラムがそれを Oracle に入力するため、すべて入力ホスト変数と呼ばれます。たとえば、INSERT 文の VALUES 句内および UPDATE 文の SET 句内では入力ホスト変数を使用します。入力ホスト変数は WHERE 句、HAVING 句、FOR 句内でも使用されます。入力ホスト変数は、SQL 文で値または式を使用できる位置であればどこにでも使用できます。

注意: ORDER BY 句では、ホスト変数を使用できますが、定数もしくはリテラルとして扱われます。このためホスト変数の内容には何の効力もありません。たとえば、次のような SQL 文があるとしたします。

```
EXEC SQL SELECT ename, empno INTO :name, :number FROM emp ORDER BY :ord;
```

この文には、:ord という入力ホスト変数が入っています。しかし、この場合のホスト変数は定数として扱われるので、:ord の値が何であっても、並べ替えは行われません。

SQL キーワードまたはデータベース・オブジェクトの名前を指定する場合には、入力ホスト変数を使用できません。つまり、ALTER、CREATE、DROP などのデータ定義文内で入力ホスト変数は使用できません。次の例の DROP TABLE 文は無効です。

```
char table_name[30];

printf("Table name? ");
gets(table_name);

EXEC SQL DROP TABLE :table_name; -- host variable not allowed
```

データベース・オブジェクト名を実行時に変更する必要があるときは、動的 SQL を使用します。動的 SQL に関する情報は 13-1 ページの「[Oracle の動的 SQL](#)」を参照してください。

入力ホスト変数を含む SQL 文を Oracle で実行する前に、それらの入力ホスト変数に値を割り当てる必要があります。次に例を示します。

```
int    emp_number;
char   temp[20];
VARCHAR emp_name[20];
```

```
/* get values for input host variables */
printf("Employee number? ");
gets(temp);
emp_number = atoi(temp);
printf("Employee name? ");
gets(emp_name.arr);
emp_name.len = strlen(emp_name.arr);
```

```
EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
VALUES (:emp_number, :emp_name);
```

INSERT 文の VALUES 句内の入力ホスト変数の前にコロンが付いていることに注意してください。

標識変数の使用

オプションのホスト変数にオプションの標識変数を関連付けることができます。標識変数に関連付けたホスト変数を SQL 文内で使用するたびに、結果コードが対応する標識変数内に格納されます。つまり、標識変数によってホスト変数を監視できます。

VALUES 句または SET 句中の標識変数を使用して、入力ホスト変数に NULL 値を割り当てたり、INSERT 句中の標識変数を使用して、出力ホスト変数内の NULL 値または切り捨てられた値を検出できます。

入力時 次は、プログラムでの標識変数への割当て値として考えられる値と、その意味です。

-1 Oracle によってその列に NULL 値が割り当てられます。このホスト変数の値は無視されます。

>=0 Oracle によってこのホスト変数の値がその列に割り当てられます。

出力時 次は、Oracle での標識変数への割当て値として考えられる値と、その意味です。

-1	列の値は NULL です。したがってこのホスト変数の値は予測不能です。
0	Oracle によって列の値がそのままこのホスト変数に割り当てられました。
>0	Oracle によって切り捨てられた列の値がこのホスト変数に割り当てられました。標識変数により戻された整数は、この列の値の元の長さを示し、SQLCA 内の SQLCODE はゼロに設定されます。
-2	Oracle によって、切り捨てられた列の値がこのホスト変数に割り当てられましたが、元の列の値（たとえば、LONG 列）が特定できません。

なお、標識変数は 2 バイトの整数として定義してください。また、SQL 文内では、標識変数の前にコロンを付けて、ホスト変数の直後に置く必要があります。

NULL 値の挿入

標識変数を使用すると、NULL 値を INSERT できます。INSERT の前に、次の例に示すように NULL 値を設定する各列に対して適切な標識変数を -1 に設定しておきます。

```
set ind_comm = -1;
```

```
EXEC SQL INSERT INTO emp (empno, comm)
VALUES (:emp_number, :commission:ind_comm);
```

標識変数 *ind_comm* は COMM 列に NULL 値を格納することを示します。

次のようにして、NULL 値をハードコードすることもできます。

```
EXEC SQL INSERT INTO emp (empno, comm)
VALUES (:emp_number, NULL);
```

この方法は柔軟性に欠けますが、非常に理解しやすい方法です。一般的には、次の例に示すように条件的に NULL を挿入します。

```
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);
```

戻された NULL 値の処理

標識変数を使用すると、次の例が示すように、戻された NULL 値を操作することもできます。

```
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;
```

NULL 値のフェッチ

DBMS=V7 または DBMS=V8 のときは、SELECT または FETCH した NULL 値を標識変数と関連付けられていないホスト変数に入れると、Oracle は次のエラー・メッセージを出します。

ORA-01405: 取り出した列の値がNULLです。

DBMS オプションの詳細は、10-15 ページの「[DBMS](#)」を参照してください。

NULL のテスト

WHERE 句で次の例のように標識変数を使用して、NULL をテストできます。

```
EXEC SQL SELECT ename, sal
        INTO :emp_name, :salary
        FROM emp
        WHERE :commission INDICATOR :ind_comm IS NULL ...
```

しかし、関係演算子を使用して NULL と NULL、または NULL と他の値を比較することはできません。たとえば、COMM 列が 1 つ以上の NULL を持つ場合、次の SELECT 文は失敗します。

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

次の例は、値のうちのいくつかが無値である可能性のある場合、値の等価性を比較する方法を示します。

```
EXEC SQL SELECT ename, sal
        INTO :emp_name, :salary
        FROM emp
```

```
WHERE (comm = :commission) OR ((comm IS NULL) AND
(:commission INDICATOR :ind_comm IS NULL));
```

切り捨てられた値のフェッチ

DBMS=V7 または V8 のとき、標識変数と関連付けられていないホスト変数で切り捨てられた列の値を SELECT または FETCH すると、エラーではなく警告が生成されます。

基本的な SQL 文

実行 SQL 文を使用すると、Oracle データの問合せ、操作および制御ができ、表、ビュー、索引などの Oracle オブジェクトを作成、定義およびメンテナンスできます。この章では、データの問合せおよび操作を行う文を重点的に説明しています。

INSERT、UPDATE または DELETE などの DML 文を実行する場合、入力ホスト変数の値を設定すること以外に考慮すべき事項は、その文が正常終了したか異常終了したかのみです。これは、SQLCA を調べることでわかります。(SQL 文を実行すると、SQLCA 変数が設定されます。) 次の 2 通りの方法で調べることができます。

- WHENEVER 文による暗黙的なチェック
- SQLCA 変数の明示的なチェック

SQLCA と WHENEVER 文についての情報は、[第 9 章「ランタイム・エラーの処理」](#)を参照してください。

SELECT 文（問合せ）を実行するときは、戻されるデータの行も処理する必要があります。問合せは次のように分類できます。

- 行を戻さない問合せ（有無を調べるのみ）
- 1 行のみを戻す問合せ
- 複数の行を戻す問合せ

複数の行を戻す問合せの場合は、カーソルを明示的に宣言するか、「ホスト配列」（配列として宣言されたホスト変数）を使用する必要があります。

注意：ホスト配列は行のバッチを処理することを可能にします。詳細は、[第 8 章「ホスト配列」](#)を参照してください。この章ではスカラー・ホスト変数の使用を想定しています。

次の埋込み SQL 文を使用すると、Oracle データの問合せおよび操作ができます。

SELECT	1 つ以上の表から行を戻します。
INSERT	表に新しい行を追加します。
UPDATE	表内の行を変更します。
DELETE	表から行を削除します。

次の埋込み SQL 文を使用すると、明示的なカーソルの定義および操作ができます。

DECLARE	カーソルに名前を付け、問合せに関連付けます。
OPEN	問合せを実行してアクティブ・セットを決定します。
FETCH	カーソルを移動してアクティブ・セット内の各行を 1 つずつ取り出します。
CLOSE	カーソルを使用禁止にします。(アクティブ・セットは未定義になります。)

以降の項では、最初に INSERT、UPDATE、DELETE および単一行の SELECT 文を記述する方法を説明します。その後、複数行の SELECT 文の説明に進みます。それぞれの文と句の詳細な説明は付録 F-1 ページの「[埋込み SQL 文および宣言文](#)」および『Oracle8i SQL リファレンス』を参照してください。

SELECT 文の使用方法

データベースへの問合せは日常的な SQL 処理です。問合せを発行するには、SELECT 文を使用します。次の例では、EMP 表を問い合わせます。

```
EXEC SQL SELECT ename, job, sal + 2000
INTO :emp_name, :job_title, :salary
FROM emp
WHERE empno = :emp_number;
```

キーワード SELECT の後に続く列名および式が「選択リスト」を構成します。この例の選択リストには 3 つの項目が含まれています。WHERE 句（および存在する場合は後続する句）内で指定した条件に基づいて、Oracle は INTO 句内のホスト変数に列の値を戻します。

選択リスト内の項目の数は INTO 句のホスト変数の数と一致している必要があります。これにより、戻された値すべてに格納場所が確保されます。

最も簡単なのは、問合せが 1 行のみを戻す場合で、前述の例の形式をとります。問合せが複数の行を戻す場合は、カーソルを使用してこれらの行を FETCH するか、これらの行をホスト変数の配列内に SELECT する必要があります。カーソルと FETCH 文についてはこの章の後半で説明します。配列処理については[第 8 章「ホスト配列」](#)で説明します。

1 行のみを戻すように作成した問合せが実際には複数行を戻す場合、SELECT の結果は予測不能です。これがエラーの原因かどうかは、SELECT_ERROR オプションの指定方法によって異なります。デフォルトの設定である「YES」の場合は、複数行が戻されるとエラーが発生します。

使用できる句

SELECT 文内では、次の標準的な SQL 句をすべて使用することができます。

SELECT 文：

- INTO
- FROM
- WHERE
- CONNECT BY
- START WITH
- GROUP BY
- HAVING
- ORDER BY
- FOR UPDATE OF

INTO 句以外の場合、埋込み SELECT 文のテキストは、SQL*Plus を使用して対話形式で実行およびテストできます。SQL*Plus では、入力ホスト変数のかわりに置換変数または定数を使用します。

INSERT 文の使用方法

INSERT 文を使用すると、表またはビューに行を追加できます。次の例では、EMP 表に 1 行追加します。

```
EXEC SQL INSERT INTO emp (empno, ename, sal, deptno)
VALUES (:emp_number, :emp_name, :salary, :dept_number);
```

「列リスト」内に指定する各列は、INTO 句で指定した表に含まれている必要があります。VALUES 句には挿入される行の値を指定します。これらの値は、定数、ホスト変数、SQL 式、SQL 関数（USER、SYSDATE など）またはユーザー定義の PL/SQL 関数のうちの、どの値であってもかまいません。

VALUES 句内の値の数は列リスト内の名前の数に等しい必要があります。ただし、表に定義されている順序で、VALUES 句に表内の各列に対する値がすべて指定されている場合は、この列リストを省略できます。

詳細は、F-65 ページの「[INSERT（実行可能埋込み SQL）](#)」を参照してください。

副問合せの使用方法

「副問合せ」はネストされた SELECT 文です。副問合せを使用すると、マルチパートの検索を実行できます。これを使用することができるのは次の場合です。

- WHERE および HAVING で比較のための値を与えます。
- START WITH 句内の比較のための値を与えます。
- DELETE 文
- CREATE TABLE または INSERT 文によって挿入される行の集合を定義します。
- UPDATE 文の SET 句に対して値を定義します。

次の例では、INSERT 文内に副問合せを使用して、1 つの表から別の表に行をコピーします。

```
EXEC SQL INSERT INTO emp2 (empno, ename, sal, deptno)
SELECT empno, ename, sal, deptno FROM emp
WHERE job= :job_title ;
```

INSERT 文で中間結果を得るために副問合せを使用していることに注意してください。

UPDATE 文の使用方法

UPDATE 文を使用すると、表またはビュー内の指定した列の値を変更できます。次の例では、EMP 表の SAL および COMM 列を更新します。

```
EXEC SQL UPDATE emp
SET sal = :salary, comm = :commission
WHERE empno = :emp_number;
```

オプションの WHERE 句を使用すると、行を更新する条件を指定できます。6-10 ページの「[WHERE 句の使用方法](#)」を参照してください。

SET 句には、値を指定する必要がある 1 つ以上の列の名前の並びを指定します。次の例に示すように、副問合せを使用すると値を指定できます。

```
EXEC SQL UPDATE emp
SET sal = (SELECT AVG(sal)*1.1 FROM emp WHERE deptno = 20)
WHERE empno = :emp_number;
```

INSERT および DELETE 文のように UPDATE 文ではオプションで戻り句を指定できます。戻り句はオプションの WHERE 条件の後にのみ置くことができます。

詳細は、F-107 ページの「[UPDATE（実行可能埋込み SQL）](#)」を参照してください。

DELETE 文の使用方法

DELETE 文を使用すると、表またはビューから行を削除できます。次の例では、EMP 表から指定した部内の全従業員を削除します。

```
EXEC SQL DELETE FROM emp
WHERE deptno = :dept_number ;
```

オプションの WHERE 句を使用して、行を削除する条件を指定しています。

戻り句オプションは DELETE 文でも使用できます。オプションの WHERE 条件の後に指定します。前述の例では、削除する各従業員のフィールド値を記録しておくことをお勧めします。

詳細は、F-42 ページの「[DELETE（実行可能埋込み SQL）](#)」を参照してください。

WHERE 句の使用方法

WHERE 句を使用すると、表またはビュー内の検索条件を満たす行のみを SELECT、UPDATE または DELETE できます。WHERE 句の検索条件はブール式であり、この式には、スカラー・ホスト変数およびホスト配列（SELECT 文内を除く）、副問合せ、ユーザー定義のストアド・ファンクションを組み込みます。

WHERE 句を省略した場合、表またはビュー内のすべての行が処理されます。UPDATE あるいは DELETE 文で WHERE 句を省略すると、SQLCA の *sqlwarn*[4] に「W」が設定され、すべての行が処理されることが警告されます。

DML 戻り句

INSERT、UPDATE および DELETE 文では、オプションで DML 戻り句を設定し、標識変数 *iv* を付けて列値の式 *expr* をホスト変数 *hv* に戻すことができます。DML 戻り句は次のように設定します。

```
{RETURNING | RETURN} {expr [,expr]}
INTO { :hv [[INDICATOR]:iv] [, :hv [[INDICATOR]:iv]] }
```

式の数とホスト変数の数と同一にしてください。この句を使用すると、アプリケーションに情報として記録する必要がある場合に、INSERT または UPDATE の後、あるいは DELETE の前に行を選択する必要がありません。戻り句を使用すると、非効率的なネットワークの往復回数や余分な処理を削減し、サーバーのメモリーを節約できます。

Oracle 動的 SQL 方法 4 では DML 戻り句はサポートされませんが、ANSI 動的 SQL 方法 4 ではサポートされます。第 14 章「[ANSI 動的 SQL](#)」を参照してください。

カーソルの使用方法

検索で複数の行が戻されると、カーソルを明示的に定義することで、次のことができます。

- 問合せによって戻された最初の行の後を処理します。
- 現在どの行が処理されているかを追跡し記録します。

ホスト配列を使用することもできます。第8章「ホスト配列」を参照してください。

カーソルは、問合せによって戻された行の集合内のカレント行を示します。これによって、プログラムは一度に1行ずつ処理できます。次の文を使用してカーソルを定義および操作します。

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

最初に、DECLARE CURSOR 文を使用してカーソルに名前を付け、問合せに関連付けます。

OPEN 文によって問合せが実行され、この問合せの検索条件を満たす行がすべて判別されます。これらの行は、カーソルのアクティブ・セットと呼ばれる集合を形成します。このカーソルを OPEN した後、対応する問合せによって戻された行を取り出すことができます。

アクティブ・セットの行は1行ずつ取り出されます（ホスト配列を使用していない場合）。FETCH 文を使用してアクティブ・セット内のカレント行を取り出します。FETCH は、すべての行が取り出されるまで繰り返し実行できます。

アクティブ・セットからの行の FETCH が終了すると、このカーソルを CLOSE 文によって使用禁止にします。（アクティブ・セットは未定義になります。）

以降の項では、アプリケーション・プログラム内でのこれらのカーソル制御文の使用方法について説明します。

DECLARE CURSOR 文の使用法

次の例に示すように、DECLARE CURSOR 文を使用してカーソルに名前を付け、問合せに関連付けて定義できます。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
      SELECT ename, empno, sal
      FROM emp
      WHERE deptno = :dept_number;
```

カーソル名はホスト変数やプログラム変数ではなく、プリコンパイラが使用する識別子なので、宣言文では定義しないでください。カーソル名で、ハイフンは使用できません。長さは

任意ですが、意味があるのは先頭の 31 文字までです。ANSI 互換性を維持するため、カーソル名は 18 文字までにしてください。

カーソルに関連付けられた SELECT 文に INTO 句を含めることはできません。INTO 句および出力ホスト変数のリストは FETCH 文の一部として指定します。

DECLARE CURSOR 文は宣言部分なので、カーソルを参照する他のすべての SQL 文より（論理的にではなく）物理的に前に配置する必要があります。つまり、カーソルの前方参照は許可されていません。次の例では OPEN 文の位置が誤っています。

```
...
EXEC SQL OPEN emp_cursor;

EXEC SQL DECLARE emp_cursor CURSOR FOR
SELECT ename, empno, sal
FROM emp
WHERE ename = :emp_name;
```

カーソル制御文（DECLARE、OPEN、FETCH、CLOSE）はすべて同一のプリコンパイル・ユニット内で指定する必要があります。たとえば、ファイル A の中でカーソルを DECLARE してファイル B で OPEN することはできません。

ホスト・プログラムではカーソルを任意の数のみ DECLARE できます。ただし、指定されたファイル内ではそれぞれの DECLARE 文は一意である必要があります。つまり、カーソルの適用範囲は 1 つのファイル全体なので、1 つのプリコンパイル・ユニット内には、別のブロックまたはプロシージャ内であっても、同じ名前のカーソルを 2 つ DECLARE することはできません。

カーソルを数多く使用するときは、MAXOPENCURSORS オプションの指定が必要な場合があります。詳細な情報は、[第 10 章「プリコンパイラのオプション」](#) および付録 C「パフォーマンスの最適化」を参照してください。

OPEN 文の使用方法

OPEN 文を使用すると、問合せを実行しアクティブ・セットを決定できます。次の例では、*emp_cursor* という名前のカーソルを OPEN します。

```
EXEC SQL OPEN emp_cursor;
```

OPEN によって、カーソルはアクティブ・セットの最初の行の直前に位置付けられます。さらに SQLCA 内の SQLERRD の第 3 要素に保存されている処理済の行カウントが 0 に設定されます。ただし、この時点では実際に取り出された行はありません。行の取出しは FETCH 文によって行われます。

カーソルを OPEN すると、問合せの入力ホスト変数はカーソルを再度 OPEN するまでは再度検査されません。つまり、アクティブ・セットは変更されません。アクティブ・セットを変更するには、そのカーソルを再度 OPEN します。

通常、カーソルは再度 OPEN する前に CLOSE する必要があります。しかし、MODE=ORACLE（デフォルト）を指定する場合は、カーソルを再度 OPEN する前に CLOSE する必要はありません。これによって、パフォーマンスが向上します。詳細は、[付録 C「パフォーマンスの最適化」](#)を参照してください。

OPEN によって行われる作業量は 3 つのプリコンパイラ・オプション HOLD_CURSOR、RELEASE_CURSOR および MAXOPENCURSORS の値によって変化します。詳細は、10-11 ページの「[プリコンパイラ・オプションの使用](#)」の項を参照してください。

FETCH 文の使用方法

FETCH 文を使用すると、アクティブ・セットから行を取り出し、結果を格納する出力ホスト変数を指定できます。カーソルに関連付けられた SELECT 文には INTO 句を組み込めないことを思い出してください。INTO 句および出力ホスト変数のリストは FETCH 文の一部として指定します。次の例では、3 つの出力ホスト変数に対して FETCH INTO を実行します。

```
EXEC SQL FETCH emp_cursor
INTO :emp_name, :emp_number, :salary;
```

カーソルはあらかじめ DECLARE し OPEN しておく必要があります。最初に FETCH 文を実行すると、アクティブ・セットの最初の行より前にあるカーソルがその最初の行に移動します。この行がカレント行になります。その後 FETCH を実行するたびに、カーソルはアクティブ・セットの次の行に進みます（カレント行を変更します）。カーソルはアクティブ・セット内を順方向にしか進みません。すでに FETCH を完了した行に戻るには、このカーソルを再度 OPEN して、その後このアクティブ・セットの最初の行からもう一度始めます。

アクティブ・セットを変更する場合は、カーソルと関連付けられた問合せ内の入力ホスト変数に新しい値を割り当て、それからカーソルを再 OPEN してください。MODE=ANSI のときは、カーソルを再 OPEN する前にいったん CLOSE する必要があります。

次の例に示すとおり、出力ホスト変数の異なる集合を使用して、同じカーソルから FETCH できます。しかし、各 FETCH 文の INTO 句内の対応するホスト変数は、同じデータ型を持つ必要があります。

```
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;
    ...
}
```

アクティブ・セットが空か、それ以上の行を含んでいない場合、FETCH は「データが見つかりません」というエラー・コードを SQLCA の *sqlcode* もしくは SQLCODE または SQLSTATE 状態変数に戻します。出力ホスト変数のステータスは予測不能です。（通常のプログラムでは、WHENEVER NOT FOUND 文でこのエラーを検出します。）このカーソルを再び使用するには、カーソルを再度 OPEN する必要があります。

次の場合、カーソル上での FETCH はエラーになります。

- カーソルをオープン（OPEN）する前
- 「データが見つかりません」条件の後
- カーソルをクローズ（CLOSE）した後

CLOSE 文の使用方法

アクティブ・セットから行の FETCH を終了すると、そのカーソルを CLOSE して、カーソルの OPEN によって取得していたリソース（記憶域など）を解放します。カーソルがクローズされると、解析ロックは解放されます。どのリソースが解放されるかは、HOLD_CURSOR および RELEASE_CURSOR オプションの設定によって異なります。次の例では、*emp_cursor* という名前のカーソルを CLOSE します。

```
EXEC SQL CLOSE emp_cursor;
```

クローズしたカーソルからは、そのアクティブ・セットが未定義になっているため、FETCH はできません。必要に応じて、カーソルを OPEN し直す（たとえば、入力ホスト変数に新しい値を指定するなど）ことができます。

MODE=ORACLE のときに、COMMIT または ROLLBACK を出すと、CURRENT OF 句内で参照されたカーソルがクローズします。他のカーソルは COMMIT または ROLLBACK によって影響されません。オープンである場合は、オープンのままです。ただし、MODE=ANSI のときは、COMMIT または ROLLBACK を出すと、すべての明示的カーソルがクローズされます。COMMIT と ROLLBACK の詳細は、[第3章「データベースの概念」](#)を参照してください。また、CURRENT OF 句の詳細は、次の項を参照してください。

CLOSE_ON_COMMIT プリコンパイラ・オプション

CLOSE_ON_COMMIT マイクロ・プリコンパイラ・オプションを使用すると、マクロ・オプション MODE=ANSI で COMMIT が実行されるときにすべてのカーソルをクローズするかどうか選択できます。MODE=ANSI のとき、CLOSE_ON_COMMIT のデフォルト値は YES です。CLOSE_ON_COMMIT=NO と明示的に設定すると、COMMIT が実行されてもカーソルはクローズされないで、カーソルを再オープンして解析する必要がなくなるためパフォーマンスが向上します。

マイクロ・オプションのマクロ・オプションに対する影響については、10-5 ページの「[マクロ・オプションおよびマイクロ・オプション](#)」を参照してください。この機能の完全な説明は、10-12 ページの「[CLOSE_ON_COMMIT](#)」を参照してください。

PREFETCH プリコンパイラ・オプション

プリコンパイラ・オプション PREFETCH を使用すると、一定の行数を事前に取り出すことによって、より効率的に問合せが行えます。そうすることで、サーバーへの往復回数を減らすことができます。構成ファイルまたはコマンドラインから PREFETCH オプション値で設定する行数は、標準的な慣例に従い、明示カーソルに関するすべての問合せに使用されます。PREFETCH オプションをインラインで使用する場合、次のカーソル文よりも先に指定する必要があります。

- EXEC SQL OPEN *cursor*
- EXEC SQL OPEN *cursor* USING *host_var_list*
- EXEC SQL OPEN *cursor* USING DESCRIPTOR *desc_name*

OPEN を実行すると、問合せ実行時に事前取得する行数が PREFETCH の値により指定されます。0（事前取得なし）から 65535 までの値を設定できます。デフォルトは 1 です。

このオプションは、選択およびフェッチ対象の配列を手動で指定するかわりに使用できます。プリフェッチは宣言された配列のサイズにかかわらず実行されます。つまり、PREFETCH=100 でサイズが 5 の配列にフェッチすると、ネットワークで 100 行が生成されます。5 がプログラムに配布されます。次のネットワーク・ラウンドトリップは 100 行すべてがフェッチされたとき（5 のバッチ）に発生します。

オブティマイザ・ヒント

Pro*C/C++ プリコンパイラは、SQL 文中のオブティマイザ・ヒントをサポートしています。「オブティマイザ・ヒント」とは、Oracle SQL オブティマイザへの提案機能であり、通常行われる最適化アプローチをオーバーライドできます。ヒントを使用して、次の事項を指定できます。

- SQL 文のための最適化アプローチ
- 参照されているそれぞれの表へのアクセス・パス
- 結合のための結合順序
- 表を結合するための方法

ヒントによって、ルールベースの最適化およびコストベースの最適化のどちらかを選択できます。コストベースの最適化を使用する場合は、この他にスループットまたは応答速度を最大にするためのヒントを使用できます。

ヒントの発行

オプティマイザ・ヒントは、SELECT、DELETE、UPDATE コマンドの直後に、C 形式または C++ 形式のコメントの中で発行できます。コメント開始記号の後に間にスペースを空けずにプラス記号 (+) を入力し、コメントに 1 つまたは複数のヒントが含まれていることを示します。たとえば、次の文では最善のスループットを得るために文のコストベース・アプローチの最適化を行う ALL_ROWS ヒントを使用しています。

```
EXEC SQL SELECT /*+ ALL_ROWS (cost-based) */ empno, ename, sal, job
        INTO :emp_rec FROM emp
        WHERE deptno = :dept_number;
```

この文で示されているように、コメントには、オプティマイザ・ヒントのみでなく、他のコメントも組み込めます。

コストベースのオプティマイザとオプティマイザ・ヒントの詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

CURRENT OF 句の使用法

DELETE 文または UPDATE 文で CURRENT OF *cursor_name* 句を使用すると、指定したカーソルから最後にフェッチした行を参照できます。カーソルをオープンし、行に位置付けておく必要があります。FETCH が実行されていない場合、またはそのカーソルがオープンされていない場合は、CURRENT OF 句の結果はエラーとなり、行は処理されません。

UPDATE 文または DELETE 文の CURRENT OF 句で参照されたカーソルを DECLARE する場合、FOR UPDATE OF 句はオプション指定です。CURRENT OF 句は、必要に応じて FOR UPDATE 句を追加するようプリコンパイラに指示します。詳細は、3-22 ページの「[FOR UPDATE OF の使用法](#)」を参照してください。

次の例では、CURRENT OF 句を使用して、*emp_cursor* という名前のカーソルから最後に FETCH した行を参照します。

```
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;
}
```


制限

CURRENT OF 句を索引構成表に使用することはできません。

明示的な FOR UPDATE OF または暗黙的な FOR UPDATE によって行の排他ロックが取得されます。すべての行は FETCH されるときではなく、OPEN されるときにロックされます。COMMIT または ROLLBACK すると、行ロックは解放されます。したがって、COMMIT 後に FOR UPDATE カーソルから FETCH を実行することはできません。FETCH を試みると、Oracle は 01002 エラー・コードを戻します。

また、ホスト配列は CURRENT OF 句と一緒に使用できません。別の方法は 8-26 ページの「[CURRENT OF の疑似実行](#)」を参照してください。

さらに、関連付けられた FOR UPDATE OF 句で複数の表は参照できません。つまり、CURRENT OF 句とは結合できないということです。

最後に、動的 SQL は CURRENT OF 句と一緒に使用できません。

すべてのカーソル文の使用法

次の例に、アプリケーション・プログラムでのカーソル制御文の一般的な順序を示します。

```
...
/* 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;
...
```

完全な例

次に、カーソルと FETCH 文を使用した完全なプログラム例を示します。このプログラムでは部門番号の入力要求が行われ、その後、その部内の全従業員の名前が表示されます。

最後の FETCH を除くすべての FETCH は 1 行を戻し、さらにその FETCH の実行中にエラーが検出されなければ正常処理を示すステータス・コードを戻します。最後の FETCH は失敗して、" データが見つかりません " というエラー・コードを *sqlca.sqlcode* に戻します。実際に FETCH された行の累積数は、SQLCA 内の *sqlerrd[2]* に示されます。

```
#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");
/* 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);
}

```

位置決定済み更新

次の略例は、汎用 ROWID を使用した位置決定済み更新を示しています。これは、4-36 ページの「汎用 ROWID」に定義されています。

```

#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;
...
```

埋込み PL/SQL

この章では、PL/SQL トランザクション処理ブロックをプログラム内に埋め込むことによりパフォーマンスを改善する方法について説明します。最初に PL/SQL の利点を挙げ、その後で次の事項について説明します。

- PL/SQL の利点
- 埋込み PL/SQL ブロック
- ホスト変数の使用
- 標識変数の使用
- ホスト配列の使用
- 埋込み PL/SQL のカーソルの使用方法
- ストアド PL/SQL および Java サブプログラム
- 外部プロシージャ
- 動的 SQL の使用

PL/SQL の利点

この項では、PL/SQL によって提供される次のような機能および利点について説明します。

- パフォーマンス向上
- Oracle との統合
- カーソル FOR ループ
- プロシージャとファンクション
- パッケージ
- PL/SQL 表
- ユーザー定義のレコード

PL/SQL の詳細は、『Oracle8i PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

パフォーマンス向上

PL/SQL によって、オーバーヘッドの削減、パフォーマンスの改善、生産性の向上が図れます。たとえば、PL/SQL を使用しないと、Oracle は SQL 文を 1 度に 1 つずつしか処理できません。その結果、各 SQL 文によってサーバーへの別のコールが発生し、オーバーヘッドが増加します。しかし、PL/SQL を使用すると、サーバーに SQL 文のブロック全体を送ることができます。これによって、アプリケーションと Oracle との間の通信は最小限になります。

Oracle との統合

PL/SQL は、Oracle Server と密接に統合されています。たとえば、PL/SQL データ型の大部分は、Oracle データ・ディクショナリにとっても固有なデータ型です。さらに、次の例に示すとおり、データ・ディクショナリ内に格納された列定義に基づいて変数を宣言するための %TYPE 属性を指定できます。

```
job_title emp.job%TYPE;
```

したがって、列の厳密なデータ型を知る必要はありません。しかも、列定義を変更すれば、変数宣言もそれに応じて自動的に変更されます。このことによって、データ独立性を提供し、メンテナンス・コストを削減し、データベース変更時にプログラムが順応できます。

カーソル FOR ループ

PL/SQL を使用すれば、カーソルを定義し操作するのに DECLARE 文、OPEN 文、FETCH 文および CLOSE 文を指定する必要はありません。かわりに、カーソル FOR ループを指定できます。カーソル FOR ループは、ループ索引をレコードとして暗黙的に宣言し、指定された問合せに関連付けられているカーソルをオープンし、データを繰り返しカーソルからフェッチしてレコードに入れてから、カーソルをクローズします。次に例を示します。

```

DECLARE
...
BEGIN
    FOR emprec IN (SELECT empno, sal, comm FROM emp) LOOP
        IF emprec.comm / emprec.sal > 0.25 THEN ...
        ...
    END LOOP;
END;

```

ドット表記法を使用すると、レコード内のコンポーネントを参照できます。

プロシージャとファンクション

PL/SQL には「プロシージャ」および「ファンクション」と呼ばれる 2 種類のサブプログラムがあります。これらを使用すると、各処理を個別に行えるため、アプリケーション開発が容易になります。一般的には、プロシージャを使用して処理を行い、ファンクションを使用して値を計算します。

プロシージャとファンクションには「拡張性」があります。つまりプロシージャとファンクションを使用することにより、PL/SQL 言語を必要に応じて調整できます。たとえば、新しい部門を作成するプロシージャが必要な場合、次のようにコーディングします。

```

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;

```

このプロシージャをコールすると、新しい部門名および新しい位置が確立され、部門番号データベース順序内のその次の値が選択され、新しい番号、名前および位置が *dept* 表の中に挿入されます。次に、新しい番号がコール側に戻ります。

仮パラメータの動作を定義するには、パラメータ・モードを使用します。パラメータ・モードは 3 種類あります。IN (デフォルト)、OUT と IN OUT です。IN パラメータを使用すると、コールされるサブプログラムに値を渡すことができます。OUT パラメータを使用すると、サブプログラムのコール側に値を戻すことができます。IN OUT パラメータを使用すると、コールされるサブプログラムに初期値を渡し、更新された値をコール側に戻すことができます。

各実パラメータのデータ型は、対応する仮パラメータのデータ型に変換する必要があります。表 7-1 はデータ型間の正当な変換を示しています。

パッケージ

PL/SQL では、論理的に関連する型、プログラム・オブジェクトおよびサブプログラムを 1 つのパッケージにまとめることができます。プロシージャ・データベース拡張要素がある場合、パッケージをコンパイルして、Oracle データベースに格納でき、そのデータベースの内容は多くのアプリケーションで共有できます。

パッケージには通常 2 つの部分があります。仕様部と本体です。仕様部とは、アプリケーションへのインタフェースです。仕様部には、使用可能な型、定数、変数、例外、カーソルおよびサブプログラムが宣言されます。本体には、カーソルおよびサブプログラムが定義され、その中で仕様部がインプリメントされます。次の例では 2 つの手順を「パッケージ」にしています：

```
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;
```

パッケージ仕様部内の宣言のみが参照可能で、アプリケーションからアクセスできます。パッケージ本体中の詳細な内容は隠されていてアクセスできません。

PL/SQL 表

PL/SQL には TABLE という名前のコンポジット・データ型が用意されています。TABLE 型のオブジェクトは「PL/SQL 表」と呼ばれ、データベース表をモデルとしています。（まったく同じではありません。）PL/SQL 表は 1 列からなり、主キーを使用して、配列と同じ方法で行にアクセスします。列は、なんらかのスカラー型（CHAR、DATE または NUMBER など）に属してもかまいませんが、主キーは BINARY_INTEGER 型に属している必要があります。

ブロック、プロシージャ、ファンクションまたはパッケージのいずれかの宣言部分で PL/SQL 表型を宣言できます。次の例では、*NumTabTyp* と呼ばれる TABLE 型を宣言しています。


```

...
DECLARE
    TYPE NumTabTyp IS TABLE OF NUMBER
        INDEX BY BINARY_INTEGER;
...
BEGIN
    ...
END;
...

```

次の例に示すように、一度 *NumTabTyp* 型を定義すれば、その型の PL/SQL 表を宣言できます。

```
num_tab NumTabTyp;
```

識別子 *num_tab* は PL/SQL 表全体を表しています。

配列に似た構文を使用して PL/SQL 表の中の行を参照し、主キーの値を指定します。たとえば、*num_tab* という名前の PL/SQL 表の 9 番目の行を参照するには次のようにします。

```
num_tab(9) ...
```

ユーザー定義のレコード

%ROWTYPE 属性を使用して、表の中の行を表すレコード、またはカーソルによってフェッチされる行を表すレコードを宣言できます。しかし、レコード内のコンポーネントのデータ型は指定できません。また、ユーザー独自のコンポーネントも定義できません。コンポジット・データ型 RECORD を提供することによって、これらの制限事項を削除することができます。

RECORD 型のオブジェクトは「レコード」と呼ばれます。PL/SQL 表とは異なり、レコードは固有の名前を持つコンポーネントで構成されています。各コンポーネントのデータ型はそれぞれ異なってもかまいません。たとえば、ある従業員について異なる種類のデータ（名前、給料、雇用日など）があるとします。これらのデータは、型は異なっていますが論理的に関連しています。従業員の名前、給料および雇用日などのコンポーネントが入っているレコードによって、1 つの論理単位としてデータを処理できます。

ブロック・プロシージャ、ファンクションまたはパッケージのいずれかの宣言部分で、レコード型およびレコード・オブジェクトを宣言できます。次の例では、*DeptRecTyp* と呼ばれる RECORD 型を宣言しています。

```

DECLARE
TYPE DeptRecTyp IS RECORD
    (deptno NUMBER(4) NOT NULL, -- default is NULL allowed
     dname  CHAR(9),
     loc    CHAR(14));

```

コンポーネント宣言は変数宣言に似ているので注意してください。各コンポーネントには、固有の名前と特定のデータ型があります。どのコンポーネント宣言にも NOT NULL オプションを追加できます。これによって、コンポーネントへの NULL の割当てを防止します。

次の例に示すように、一度 *DeptRecTyp* を定義すれば、その型のレコードを宣言できます。

```
dept_rec DeptRecTyp;
```

識別子 *dept_rec* はレコード全体を表しています。

ドット表記法を使用すると、レコード内の個別コンポーネントを参照できます。たとえば、次のように *dept_rec* レコードの中で *dname* コンポーネントを参照します。

```
dept_rec.dname ...
```

埋込み PL/SQL ブロック

Pro*C/C++ プリコンパイラは PL/SQL ブロックを 1 つの埋込み SQL 文と同様に取り扱います。したがって、PL/SQL ブロックは、プログラム内の SQL 文を記述できる位置であればどこにでも記述できます。

PL/SQL ブロックを Pro*C/C++ プログラム内に埋め込むには、次のように、EXEC SQL EXECUTE および END-EXEC キーワードで PL/SQL ブロックを囲むのみで済みます。

```
EXEC SQL EXECUTE
DECLARE
...
BEGIN
    ...
END;
END-EXEC;
```

キーワード END-EXEC の後には、セミコロンを付ける必要があります。

プログラムを作成した後、通常の方法でソース・ファイルをプリコンパイルします。

プログラムに埋込み PL/SQL が含まれている場合、PL/SQL は Oracle Server によって解析される必要があるため、必ず SQLCHECK=SEMANTICS コマンドライン・オプションを指定してください。サーバーに接続する場合には、SQLCHECK=SEMANTICS のみでなく USERID オプションも指定してください。詳細は、10-11 ページの「[プリコンパイラ・オプションの使用](#)」を参照してください。

ホスト変数の使用

ホスト変数はホスト言語と PL/SQL ブロック間の通信を仲介します。ホスト変数と PL/SQL は共有できます。これにより、PL/SQL でのホスト変数の設定および参照が可能になります。

たとえば、ユーザーに対して情報を提供し、ホスト変数を使用してその情報を PL/SQL ブロックに渡すことができます。これにより、PL/SQL を使用してデータベースにアクセスし、ホスト変数を介してその結果をホスト・プログラムに戻すことができます。

PL/SQL ブロック内ではホスト変数はブロック全体のグローバル変数として扱われ、PL/SQL 変数を使用できる位置であればどこにでも使用できます。SQL 文内におけるホスト変数と同様、PL/SQL ブロック内のホスト変数も先頭にコロンを付ける必要があります。コロンはホスト変数と PL/SQL 変数およびデータベース・オブジェクトとを区切ります。

例

次の例では PL/SQL と使用したホスト変数の使用方法について説明します。プログラムはユーザーに従業員番号の入力を求め、その番号に応じて、従業員の役職名、雇用日、給与を表示します。

```
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;
    }
}
/*----- begin PL/SQL block -----*/
EXEC SQL EXECUTE
BEGIN
    SELECT job, hiredate, sal
```

```
        INTO :job_title, :hire_date, :salary
        FROM emp
        WHERE empno = :emp_number;
    END;
    END-EXEC;
/*----- end PL/SQL block -----*/

    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 SQL ROLLBACK WORK RELEASE;
printf("Processing error\n");
exit(1);
```

ホスト変数 *emp_number* は PL/SQL ブロックが入力される前に設定され、ホスト変数 *job_title*、*hire_date* および *salary* はブロック内で設定されることに注意してください。

より複雑な例

次の例では、ユーザーに銀行口座番号、取引の種類、取引金額の入力を求め、次に口座への記帳を行います。口座が存在しない場合は、例外が発生します。取引が完了すると、そのステータスが表示されます。

```
#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);

/*----- begin PL/SQL block -----*/
    EXEC SQL EXECUTE
    DECLARE
        old_bal      NUMBER(9,2);
        err_msg      CHAR(70);
        nonexistent  EXCEPTION;

    BEGIN
        :trans_type := UPPER(:trans_type);
        IF :trans_type = 'C' THEN      -- credit the account
            UPDATE accts SET bal = bal + :trans_amt
            WHERE acctid = :acct_num;
            IF SQL%ROWCOUNT = 0 THEN  -- no rows affected
                RAISE nonexistent;
            ELSE
                :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;
END IF;
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);
}

```

VARCHAR 疑似型

VARCHAR データ型を使用して、可変長の文字列を宣言できることを思い出してください。VARCHAR が入力ホスト変数の場合は、どのくらいの長さを予期すればよいかを Oracle に通知する必要があります。したがって、文字列コンポーネントに格納される値の実際の長さ、長さコンポーネントを設定してください。

VARCHAR が出力ホスト変数の場合は、Oracle が自動的に長さコンポーネントを設定します。しかし、PL/SQL ブロックで VARCHAR 出力ホスト変数を使用するには、ブロックを入力する前に長さコンポーネントを初期化する必要があります。したがって、次の例に示す

とおり、長さコンポーネントは VARCHAR の宣言された（最大の）長さに設定してください。

```
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;
...
```

制限

PL/SQL ブロックでは、C ポインタまたは配列構文を使用しないでください。PL/SQL コンパイラは C ホスト変数の式を認識できないため、解析できません。たとえば、次の SQL 文は無効です。

```
EXEC SQL EXECUTE
  BEGIN
    :x[5].name := 'SCOTT';
    ...
  END;
END-EXEC;
```

構文エラーを避けるには、プレースホルダ（一時変数）を使用して、構造体への移入を行う構造体フィールドのアドレスを保持します。たとえば、次の使用方法は有効です。

```
name = x[5].name ;
EXEC SQL EXECUTE
  BEGIN
    :name := ...;
    ...
  END;
END-EXEC;
```

標識変数の使用

PL/SQL では NULL 値を操作できるため、標識変数を必要としません。たとえば、PL/SQL 内で IS NULL 演算子を使用すると、次のように NULL 値をテストすることができます。

```
IF variable IS NULL THEN ...
```

その後、次のように代入演算子 (:=) を使用して、NULL 値を指定できます。

```
variable := NULL;
```

ただし、C のようなホスト言語は NULL 値を操作できないので、標識変数を必要とします。埋込み PL/SQL でこの要件を満たすには、次の用途に標識変数を使用します。

- ホスト・プログラムから NULL 入力値を受け入れます。
- NULL 値または切り捨てられた値をホスト・プログラムに出力します。

PL/SQL ブロックで標識変数を使用する場合は、次の規則に従ってください。

- 標識変数自体は参照できません。対応するホスト変数に追加する必要があります。
- 標識変数でホスト変数を参照する場合は、必ず同じブロックで参照します。

次の例では、標識変数 *ind_comm* が SELECT 文でそのホスト変数 *commission* とともに表示されているので、IF 文でも同じように表示する必要があります。

```
...
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;
```

PL/SQL では *:commission :ind_comm* はその他の単純な変数と同じように扱われるので注意してください。PL/SQL ブロック内の標識変数は直接参照できませんが、PL/SQL により、ブロックに入力されるときに標識変数の値がチェックされ、ブロックから出るときにその値が正しく設定されます。

NULL の処理

ブロックに入るとき、標識変数の値が -1 であれば、PL/SQL により NULL がホスト変数に自動的に割り当てられます。ブロックから出るとき、ホスト変数が NULL であれば、PL/SQL により値 -1 が標識変数に自動的に割り当てられます。次の例で、PL/SQL ブロック

に入力される前に *ind_sal* の値が -1 であった場合は、*salary_missing* 例外が発生します。「例外」とは、名前が指定されたエラー条件です。

```
...
EXEC SQL EXECUTE
BEGIN
    IF :salary :ind_sal IS NULL THEN
        RAISE salary_missing;
    END IF;
...
END;
END-EXEC;
...
```

切捨て値の処理

PL/SQL では、切り捨てられた文字列の値がホスト変数に割り当てられても、例外と見なされません。しかし、標識変数を指定している場合には、PL/SQL によりその標識変数が文字列の元の長さに設定されます。次の例では、ホスト・プログラムで *ind_name* の値をチェックして、切り捨てられた値が *emp_name* に割り当てられているかどうかを通知できます。

```
...
EXEC SQL EXECUTE
DECLARE
...
new_name CHAR(10);
BEGIN
    ...
    :emp_name:ind_name := new_name;
    ...
END;
END-EXEC;
```

ホスト配列の使用

入力ホスト配列および標識配列は、PL/SQL ブロックに渡せます。入力ホスト配列および標識配列は、BINARY_INTEGER 型の PL/SQL 変数またはその型と互換性のあるホスト変数を使用して索引付けができます。通常は、ホスト配列全体が PL/SQL に渡されます。しかし、より小さい配列サイズを指定するには、ARRAYLEN 文（後述）を使用できます。

さらに、ホスト配列のすべての値を PL/SQL 表の複数の行に割り当てるには、プロシージャ・コールが使用できます。配列の添字範囲が $m..n$ とすると、対応する PL/SQL 表の索引範囲は常に $1..n-m+1$ になります。たとえば、配列の添字範囲が 5..10 の場合、対応する PL/SQL 表の索引範囲は $1..(10-5+1)$ または $1..6$ です。

次の例では、*salary* という名前の配列を PL/SQL ブロックに渡し、そのブロックのファンクション・コール内でその配列を使用しています。この関数は一連の数値の中央値を検出する

ので、*median* という名前が付けられています。この関数の仮パラメータには、*num_tab* という PL/SQL 表が含まれています。この関数コールは、実パラメータ *salary* 内のすべての値を仮パラメータ *num_tab* 内の行に割り当てます。

```
...
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;
...
```

警告：動的 SQL 方法 4 では、“table” タイプのパラメータを使用して、ホスト配列を PL/SQL プロシージャにバインドすることはできません。詳細は、13-23 ページの「[方法 4 の使用方法](#)」を参照してください。

PL/SQL 表のすべての行の値をホスト配列の対応する要素に割り当てする場合にも、プロシージャ・コールを使用できます。例は、7-19 ページの「[ストアド PL/SQL および Java サブプログラム](#)」の項を参照してください。

表 7-1 に PL/SQL 表の行の値とホスト配列のエレメント間での有効な変換を示します。たとえば、LONG 型のホスト配列は、VARCHAR2 型、LONG 型、RAW 型または LONG RAW 型の PL/SQL 表と互換性があります。しかし、CHAR 型の PL/SQL 表とは互換性がないので注意してください。

表 7-1 正当なデータ型変換

PL/SQL 表 > ホスト配列	CHAR	DATE	LONG	LONG RAW	NUMBER	RAW	ROWID	VARCHAR2
CHARF	X							
CHARZ	X							
DATE		X						
DECIMAL					X			
DISPLAY					X			
FLOAT					X			
INTEGER					X			
LONG	X		X					
LONG VARCHAR			X	X		X		X
LONG VARRAW				X		X		
NUMBER					X			
RAW				X		X		
ROWID							X	
STRING			X	X		X		X
UNSIGNED					X			
VARCHAR			X	X		X		X
VARCHAR2			X	X		X		X
VARNUM					X			
VARRAW				X		X		

Pro*C/C++ プリコンパイラでは、ホスト配列の使用方法はチェックされません。たとえば、索引範囲チェックは実行されません。

ARRAYLEN 文

入力ホスト配列を PL/SQL ブロックに渡して処理するとします。デフォルトで、入力ホスト配列をバインドすると、Pro*C/C++ プリコンパイラはその宣言されたサイズを使用します。しかし、その配列全体を処理しない場合もあります。この場合には、ARRAYLEN 文を使用して、より小さい配列サイズを指定できます。ARRAYLEN 文はホスト配列をホスト変数と対応付け、そのホスト変数がより小さいサイズを格納します。文の構文は次のとおりです。

```
EXEC SQL ARRAYLEN host_array (dimension) [EXECUTE];
```

dimension は 4 バイト整数型のホスト変数です。リテラルや式ではありません。

EXECUTE はオプションのキーワードです。

ARRAYLEN 文は *host_array* および *dimension* の宣言とともに（ただし、それらの宣言よりも後に）表示する必要があります。ホスト配列の中にはオフセットを指定できません。しかし、この目的に C の機能が使用できる場合もあります。次の例では、ARRAYLEN を使用して、*bonus* という名前の C ホスト配列のデフォルトのサイズを指定変更しています。

```
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;
```

ARRAYLEN でホスト配列のサイズが 100 要素から 25 要素に減少するので、25 の配列要素のみが PL/SQL ブロックに渡されます。結果として、PL/SQL ブロックが実行のために Oracle に送られるとき、より小さくなったホスト配列と一緒に送られます。これは、時間を節約し、ネットワーク化された環境でのネットワーク通信量を削減します。

オプション・キーワード EXECUTE

動的 SQL 方法 2 の EXEC SQL EXECUTE 文で使用するホスト配列には、オプション・キーワード EXECUTE の有無によって、異なる 2 つの解釈があります。13-12 ページの「[方法 2 の使用方法](#)」を参照してください。

デフォルト (ARRAYLEN 文に EXECUTE キーワードがないとき) :

- ホスト配列は PL/SQL ブロックが実行される回数が決定されるとき考慮されます。(最小値の配列が使用されます。)
- ホスト配列は PL/SQL 索引表に結合されません。

キーワード EXECUTE が存在しているとき :

- ホスト配列は索引表に結合されます。
- PL/SQL ブロックは一回のみ実行されます。
- EXEC SQL EXECUTE 文で指定されているすべてのホスト変数は、次のいずれかです。
 - ARRAYLEN EXECUTE 文で指定される。
 - スカラー。

たとえば、次の PL/SQL プロシージャを仮定します。

```
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;
```

次の Pro*C/C++ ファンクションは特定の PL/SQL ブロックを実行する回数を決定するためにホスト配列を使用する方法を示しています。この場合、PL/SQL ブロックは emp 表に新しい行が 3 行あるために 3 回実行されます。

```
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;
}
```

次の Pro*C/C++ ファンクションは動的方法 2 によってホスト配列を PL/SQL 索引表に結合する方法を示しています。すべてのホスト配列の ARRAYLEN...EXECUTE 文の存在が EXEC SQL EXECUTE 文内で指定されていることに注意してください。

```
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;  
}
```

次の Pro*C/C++ ファンクションは int_arr の ARRAYLEN...EXECUTE 文がないために、プリコンパイル時警告を生じます。

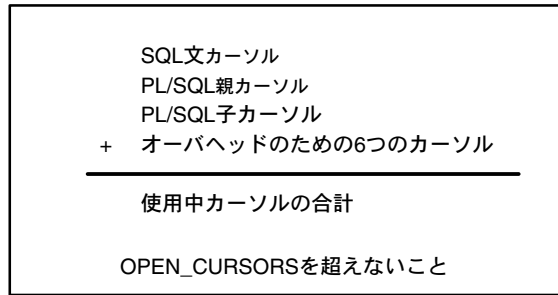
```
func3()  
{  
    int int_arr[3];  
    int int_tab[3] = {1,2,3};  
    int dim = 3;  
    EXEC SQL ARRAYLEN int_tab (dim) EXECUTE;  
  
    char *stmt3 = "begin pkg.proc1(:v1, :v2, :v3); end; ";  
  
    EXEC SQL PREPARE s3 FROM :stmt3;  
    EXEC SQL EXECUTE s3 USING :int_tab, :int_arr, :int_tab;  
}
```

注意: ホスト配列の完全な説明は、[第8章「ホスト配列」](#)を参照してください。

埋込み PL/SQL のカーソルの使用方法

プログラムで同時に使用できるカーソルの最大数は、データベース初期化パラメータ OPEN_CURSORS によって設定されます。埋込み PL/SQL ブロックの実行中に、1つのカーソル、親カーソルがブロック全体に関連付けられ、1つのカーソル、子カーソルが埋込み PL/SQL ブロックの各 SQL 文に関連付けられます。親カーソルおよび子カーソルはともに OPEN_CURSORS 制限値までカウントします。[図 7-1「利用されているカーソルの最大数」](#)に利用されているカーソルの最大数を計算する方法を示します。

図 7-1 利用されているカーソルの最大数



プログラムで、OPEN_CURSORS によって設定された制限を超える数のカーソルが使用された場合、エラーが発生します。C-11 ページの「[埋込み PL/SQL の考慮事項](#)」を参照してください。

ストアド PL/SQL および Java サブプログラム

無名ブロックとは異なり、PL/SQL サブプログラム（プロシージャおよびファンクション）および Java メソッドは別個にコンパイルされ、Oracle データベースに格納されて起動されます。

SQL*Plus などの Oracle ツールを使用して明示的に作成したサブプログラムはストアド・サブプログラムと呼ばれます。ストアド・サブプログラムは、一度コンパイルしてデータ・ディクショナリに格納しておく、再コンパイルしなくても再実行できるデータベース・オブジェクトになります。

PL/SQL ブロックまたはストアド・プロシージャ内のサブプログラムがアプリケーションによって Oracle に送られると、それはインライン・サブプログラムと呼ばれます。Oracle は、インライン・サブプログラムをコンパイルし、システム・グローバル領域（SGA）にキャッシュしますが、データ・ディクショナリへのソース・コードまたはオブジェクト・コードの格納はしません。

パッケージ内で定義されているサブプログラムはそのパッケージの一部とみなされ、そのためパッケージ・サブプログラムと呼ばれます。パッケージで定義されていないストアド・サブプログラムはスタンドアロン・サブプログラムと呼ばれます。

ストアド・サブプログラムの生成

次の例に示すように、SQL 文 CREATE FUNCTION、CREATE PROCEDURE および CREATE PACKAGE をホスト・プログラムに埋め込むことができます。

```
EXEC SQL CREATE
FUNCTION sal_ok (salary REAL, title CHAR)
RETURN BOOLEAN AS
min_sal  REAL;
max_sal  REAL;
BEGIN
    SELECT losal, hisal INTO min_sal, max_sal
    FROM sals
    WHERE job = title;
    RETURN (salary >= min_sal) AND
           (salary <= max_sal);
END sal_ok;
END-EXEC;
```

埋込み CREATE {FUNCTION | PROCEDURE | PACKAGE} 文がハイブリッドであることに注意してください。他のすべての CREATE 埋込み文と同様に、キーワード EXEC SQL (EXEC SQL EXECUTE ではない) で始まります。しかし、PL/SQL 終了記号の END-EXEC で終了する点は他の CREATE 埋込み文と異なります。

次の例では、emp 表から 1 組の行を取り出す *get_employees* という名前のプロシージャを含むパッケージを作成します。バッチ・サイズは、プロシージャのコール側によって定義されます。コール側は、別のストアド・サブプログラムやクライアント・アプリケーションであってもかまいません。

プロシージャは 3 つの PL/SQL 表を OUT 仮パラメータとして宣言し、その後雇用データのバッチを PL/SQL 表にフェッチします。対応する実パラメータはホスト配列です。プロシージャは終了時に、PL/SQL 表のすべての行の値を、ホスト配列の対応する要素に自動的に割り当てます。

```
EXEC SQL CREATE OR REPLACE PACKAGE emp_actions AS
    TYPE CharArrayType IS TABLE OF VARCHAR2(10)
        INDEX BY BINARY_INTEGER;
    TYPE NumArrayType IS TABLE OF FLOAT
        INDEX BY BINARY_INTEGER;
    PROCEDURE get_employees(
        dept_number IN    INTEGER,
        batch_size  IN    INTEGER,
        found       IN OUT INTEGER,
        done_fetch  OUT   INTEGER,
        emp_name    OUT   CharArrayType,
        job_title   OUT   CharArrayType,
        salary      OUT   NumArrayType);
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(
    dept_number IN      INTEGER,
    batch_size  IN      INTEGER,
    found       IN OUT  INTEGER,
    done_fetch  OUT     INTEGER,
    emp_name    OUT     CharArrayType,
    job_title   OUT     CharArrayType,
    salary      OUT     NumArrayType) IS

BEGIN
    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;

```

CREATE 文で REPLACE 句を指定すると、パッケージの削除、再作成および権限再付与を実行しなくても既存のパッケージを再定義できます。CREATE 文の完全な構文は『Oracle8i SQL リファレンス』を参照してください。

埋込み CREATE {FUNCTION | PROCEDURE | PACKAGE} 文が失敗した場合には、Oracle はエラーではなく、警告を生成します。

ストアド PL/SQL または Java サブプログラムのコール

ホスト・プログラムからストアド・サブプログラムをコールするには、無名 PL/SQL ブロックまたは CALL 埋込み SQL 文のいずれかを使用できます。

無名 PL/SQL ブロック

次の例では、*raise_salary* という名前のスタンドアロン・プロシージャをコールします。

```
EXEC SQL EXECUTE
  BEGIN
    raise_salary(:emp_id, :increase);
  END;
END-EXEC;
```

ストアド・サブプログラムにはパラメータを組み込めることに注意してください。この例では、実パラメータ *emp_id* および *increase* は C のホスト変数です。

次の例では、プロシージャ *raise_salary* は *emp_actions* という名前のパッケージに格納されます。したがって、プロシージャ・コールを完全に修飾するにはドット表記法を使用する必要があります。

```
EXEC SQL EXECUTE
  BEGIN
    emp_actions.raise_salary(:emp_id, :increase);
  END;
END-EXEC;
```

IN 実パラメータは、リテラル、スカラー・ホスト変数、ホスト配列、PL/SQL 定数または PL/SQL 変数、PL/SQL 表、PL/SQL ユーザー定義レコード、プロシージャ・コールあるいは式のいずれかにしてもかまいません。しかし、OUT 実パラメータは、リテラル、プロシージャ・コールまたは式にしないでください。

埋込み PL/SQL ブロックでプリコンパイラ・オプション `SQLCHECK=SEMANTICS` を使用する必要があります。

次の例では、仮パラメータのうち 3 つが PL/SQL 表であり、対応する実パラメータはホスト配列です。プログラムはストアド・プロシージャ *get_employees* (7-19 ページの「[ストアド・サブプログラムの生成](#)」を参照してください。) をコールして、従業員のデータを一群ずつ表示して、データがなくなるまで繰り返します。このプログラムは、*demo* ディレクトリの *sample9.pc* ファイルに入っており、オンラインで利用できます。CALLDEMO ストアド・パッケージ作成用の SQL スクリプトは、*calldemo.sql* ファイルに入っています。

```
/*****
Sample Program 9: Calling a stored procedure
```

```
This program connects to ORACLE using the SCOTT/TIGER
account. The program declares several host arrays, then
calls a PL/SQL stored procedure (GET_EMPLOYEES in the
CALLDEMO package) that fills the table OUT parameters. The
PL/SQL procedure returns up to ASIZE values.
```

```
Sample9 keeps calling GET_EMPLOYEES, getting ASIZE arrays
each time, and printing the values, until all rows have been
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;
int      dept_no;           /* which department to query? */
vc2_arr  emp_name[10];      /* array of returned names */
vc2_arr  job[10];
float    salary[10];
int      done_flag;
int      array_size;
int      num_ret;           /* number of rows returned */
EXEC SQL END DECLARE SECTION;

long      SQLCODE;

void print_rows();           /* produces program output */
void sql_error();           /* handles unrecoverable errors */

main()
{
    int i;
    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\n", "Employee", "Job", "Salary");
printf("%-10.10s%-10.10s\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.
 * 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);
}

```

それぞれの実パラメータのデータ型は、対応する仮パラメータのデータ型に変換可能である必要があります。また、ストアード・プロシージャの終了前には、すべての OUT 仮パラメータは割り当てられた値である必要があります。そうでなければ、対応する実パラメータの値は未確定です。

無名 PL/SQL ブロックを使用する場合、SQLCHECK=SEMANTICS は必須です。

リモート・アクセス

PL/SQL を使用すると、データベース・リンクを経由してリモート・データベースにアクセスできます。一般的に、データベース・リンクは、データベース管理者 (DBA) によって確立され、Oracle データ・ディクショナリに格納されます。データベース・リンクは、リモート・データベースが位置付けられる場所、リモート・データベースへのパス、使用する Oracle ユーザー名とパスワードを Oracle に伝えます。次の例では、データベース・リンク *dallas* を使用して、*raise_salary* プロシージャをコールします。

```

EXEC SQL EXECUTE
BEGIN
    raise_salary@dallas(:emp_id, :increase);
END;

```

```
END-EXEC;
```

次の例に示すように、シノニムを作成して、リモート・サブプログラムに位置の透過性を提供できます。

```
CREATE PUBLIC SYNONYM raise_salary  
FOR raise_salary@dallas;
```

CALL 文

前述の埋込み PL/SQL ブロックの概念は CALL 文にも当てはまります。CALL 埋込み SQL 文のフォームは次のとおりです。

```
EXEC SQL  
    CALL [schema.] [package.]stored_proc[@db_link] (arg1, ...)  
    [INTO :ret_var [[INDICATOR]:ret_ind]] ;
```

パラメータは次のとおりです。

schema

プロシージャを含むスキーマ

package

プロシージャを含むパッケージ

stored_proc

コールする Java または PL/SQL ストアド・プロシージャ

db_link

オプション・リモート・データベース・リンク

arg1...

渡された引数（変数、リテラル、式）

ret_var

結果を受け取るオプション・ホスト変数

ind_var

ret_var のオプション標識変数

CALL 文には、SQLCHECK=SYNTAX または SEMANTICS のいずれかを使用できます。

CALL の例

入力として整数値をとる PL/SQL 関数 fact（パッケージ mathpkg に格納されています）を作成し、その階乗の整数値を戻します。

```
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.
```

次に、その CALL 文を使用している Pro*C/C++ アプリケーションで、fact を使用します。

```
...
int num, fact;
...
EXEC SQL CALL mathpkge.fact(:num) INTO :fact ;
...
```

CALL 文の詳細は、F-17 ページの「[CALL \(実行可能埋込み SQL\)](#)」を参照してください。引数の渡し方およびその他の事例の詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』、「外部ルーチン」の章を参照してください。

ストアド・サブプログラムに関する情報を得る方法

[第4章「データ型とホスト変数」](#)ではホスト・プログラムに OCI コールを埋め込む方法を説明しています。ライブラリ・ルーチン SQLLDA をコールして LDA を設定したあと、OCI コール *odessp* を使用して、ストアド・サブプログラムに関する役立つ情報を取得します。*odessp* をコールするときには、有効な LDA とサブプログラム名を渡す必要があります。パッケージ・サブプログラムの場合は、パッケージ名も渡す必要があります。*odessp* は、各サブプログラム・パラメータに関する情報（データ型、サイズ、位置など）を戻します。詳細は『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

DBMS_DESCRIBE パッケージでは、DESCRIBE_PROCEDURE ストアド・プロシージャを使用できます。このプロシージャの詳細は、『Oracle8i アプリケーション開発者ガイド』を参照してください。

外部プロシージャ

PL/SQL では、外部プロシージャの C 関数をコールできます。外部プロシージャ（外部ルーチンとも呼ばれます）は、動的リンク・ライブラリ（DLL）または Solaris の .so ライブラリなどに格納されています。

外部プロシージャをサーバーで実行する場合、同じトランザクションで SQL および PL/SQL が実行されるようにサーバーにコールバックできます。サーバーで外部ルーチンを実行すると、クライアントで実行した場合よりも処理速度が速く、外部システムとデータ・

ソース、およびデータベース・サーバー間のインタフェースとして使用することができます。

サーバー側の外部 C 関数を実行する場合、関数内で REGISTER CONNECT 埋込み SQL 文を使用する必要があります。文の構文は次のとおりです。

```
EXEC SQL REGISTER CONNECT USING :epctx [RETURNING :host_context] ;
```

epctx は、OCIExtProcContext への型ポインタの外部プロシージャ・コンテキストです。epctx は PL/SQL によってプロシージャに渡されます。

host_context は、外部プロシージャによって戻されるランタイム・コンテキストです。カレント行の設定は、デフォルト（グローバル）コンテキストです。

REGISTER CONNECT 文により、カレント Oracle8 接続およびトランザクションに関連付けられた OCI ハンドル・セット（OCIEnv、OCISvcCtx および OCIError）が戻されます。これらのハンドルは、グローバル SQLLIB ランタイム・コンテキストの Pro*C/C++ デフォルト名前なし接続の定義に使用されます。このため、REGISTER CONNECT が、CONNECT 文のかわりに使用されます。

後続の埋込み SQL 文では、この OCI ハンドル・セットを使用します。後続の埋込み SQL 文は、グローバル SQLLIB ランタイム・コンテキスト、名前なし接続に対して実行されます。別々にプリコンパイルされたプログラム・ユニットにある埋込み SQL 文も同様です。コミットされていない変更は無効です。今後のバージョンでは、（非デフォルト）ランタイム・コンテキストはオプションの RETURNING 句に戻されます。

グローバル・ランタイム・コンテキストのアクティブなデフォルト接続はまだありません。すでに接続が確立しているときに REGISTER CONNECT を使用した場合、ランタイム・エラーが戻されます。

OCI 関数の詳細は『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

ここで、簡単な外部プロシージャの例を挙げます。実際の業務では、外部プロシージャは複数の異なるアプリケーションから再使用できるようにすることをお勧めします。

外部プロシージャの制限

外部プロシージャには次の規則があります。

- 外部プロシージャは C でのみ使用できます。C++ 外部プロシージャはサポートされていません。
- 外部プロシージャ・コンテキストに接続した場合、接続を追加できません。ランタイム・エラーが発生します。
- マルチスレッドの外部プロシージャはサポートされていません。EXEC SQL ENABLE THREADS 文は使用できません。ランタイム・エラーが戻されます。Pro*C/C++ では、ここで説明している外部プロシージャを使用しない場合は、アプリケーションでのマルチスレッドをサポートしています。

- DDL 文を使用できません。ランタイム・エラーが発生します。
- EXEC SQL COMMIT および EXEC SQL ROLLBACK などのトランザクション制御文を使用できません。
- EXEC SQL OBJECT などのオブジェクト・ナビゲーション文を使用できません。
- EXEC SQL LOB 文のポーリングを行うことはできません。
- EXEC TOOLS 文を使用できません。ランタイム・エラーが発生します。

外部プロシージャの作成

外部プロシージャ `extp1` を作成する場合の簡単な例を示します。

外部 C プロシージャを格納するには、コードのコンパイルおよびリンクを行い、NT 上の DLL などのライブラリに格納します。

次の SQL コマンドを一回実行して、外部プロシージャ `extp1` を登録します。

```
CREATE OR REPLACE PROCEDURE extp1
AS EXTERNAL NAME "extp1"
LIBRARY mylib
WITH CONTEXT
PARAMETERS (CONTEXT) ;
```

`mylib` は、プロシージャ `extp1` が格納されるライブラリです。WITH CONTEXT が指定されているので、このプロシージャは、引数型 `OCIExtProcContext*` で暗黙的にコールされます。このコールでは、コンテキストが省略されていますが、プロシージャには渡されません。ただし、CREATE 文のキーワード CONTEXT は、プレース・マーカーとして指定されています。

このコンテキスト・パラメータは、`extp1` の内側の EXEC SQL REGISTER CONNECT 文で参照されます。

外部プロシージャのコールのバックグラウンドは、『Oracle8i PL/SQL ユーザーズ・ガイド およびリファレンス』を参照してください。

外部プロシージャは、SQL*Plus から次のようにコールされます。

```
SQL>
BEGIN
    INSERT INTO emp VALUES (9999, 'JOHNSON', 'SALESMAN', 7782, sysdate, 1200, 150, 10);
    extp1;
END;
```

`extp1.pc` のリストです。

```
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 empno = 9999;
return;
err: SQLExtProcError(SQL_SINGLE_RCTX, sqlca.sqlerrm.sqlerrmc, sqlca.sqlerrm.sqlerrml);
return;
}
```

SQLExtProcError()

SQLLIB 関数 *SQLExtProcError()* を使用すると、外部 C プロシージャでエラーが発生した場合に PL/SQL に制御を戻すことができます。関数とその引数は次のとおりです。

SQLExtProcError (ctx, msg, msglen)

パラメータは次のとおりです。

ctx (IN) sql_context *

この関数は、REGISTER CONNECT 文のターゲット SQLLIB ランタイム・コンテキストです。REGISTER CONNECT 文は、この関数が起動される前に実行する必要があります。現在、グローバル・ランタイム・コンテキストのみがサポートされています。

msg (OUT) char *

エラー・メッセージのテキスト

msglen (OUT) size_t

メッセージのバイト単位の長さ

この関数が実行されると、SQLLIB により OCI サービス関数 *OCIExtProcRaiseExcpWithMsg* がコールされます。

メッセージは、SQLCA の構造体 *sqlerrm* から出力されます。SQLCA および *sqlerrm* の説明は、9-20 ページの「[SQLCA の構造](#)」を参照してください。

SQLExtProcError() の使用方法の例です。

```
void extpl (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 empno = 9999;
    return;
err:
    SQLExtProcError (SQL_SINGLE_RCTX, sqlca.sqlerrm.sqlerrmc,
                    sqlca.sqlerrm.sqlerrml);
}
```

```
printf("\n%s*s\n", sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);  
return;  
}
```

動的 SQL の使用

プリコンパイラでは、PL/SQL ブロック全体が 1 つの SQL 文として処理されます。つまり、PL/SQL ブロックをホスト変数の文字列に格納できます。この場合、ブロックにホスト変数が含まれない場合は、動的 SQL 方法 1 を使用して、PL/SQL 文字列を EXECUTE できます。ブロックにホスト変数が含まれる場合、変数の数がわかっているときは、動的 SQL 方法 2 を使用して PL/SQL 文字列を PREPARE および EXECUTE できます。ブロックにホスト変数が含まれる場合、変数の数がわからないときは、動的 SQL 方法 4 を使用する必要があります。

詳細は、[第 13 章「Oracle の動的 SQL」](#)、[第 14 章「ANSI 動的 SQL」](#) および [第 15 章「Oracle の動的 SQL 方法 4」](#) を参照してください。

警告：動的 SQL 方法 4 では、"table" タイプのパラメータを使用して、ホスト配列を PL/SQL プロシージャにバインドすることはできません。詳細は、13-23 ページの「[方法 4 の使用方法](#)」を参照してください。

ホスト配列

この章では、配列を使用してコーディングを簡略化しプログラムのパフォーマンスを改善する方法について説明します。配列を使用して Oracle データを操作する方法、単一の SQL 文を使用して配列内のすべての要素を処理する方法、処理対象となる配列の要素数を制限する方法について説明します。次の事項を扱います。

- 配列を使用する理由
- ホスト配列の宣言
- SQL 文での配列の使用
- 配列への選択
- 配列での挿入
- 配列での更新
- 配列での削除
- FOR 句の使用方法
- WHERE 句の使用方法
- 構造体配列
- CURRENT OF の疑似実行
- sqlca.sqlerrd[2] の使用方法

配列を使用する理由

配列を使用すると、プログラミングの所要時間が短縮され、パフォーマンスを改善できます。

配列によって、単一の SQL 文で配列全体を操作できます。このため、特にネットワーク化された環境では、Oracle 通信のオーバーヘッドが著しく軽減されます。実行時間の大部分は、ネットワーク上でクライアント・プログラムとサーバー・データベースとの間を往復することに費やされます。配列を使用すると、往復回数が減少します。

たとえば、およそ 300 人の従業員に関する情報を EMP という表に挿入する必要があるとします。配列がないと、プログラムは 300 の個々の INSERT（各従業員に 1 つ）を実行する必要があります。配列を使用すれば、INSERT の実行は 1 回で済みます。

ホスト配列の宣言

次の例では 3 つのホスト配列を宣言するとともに、それぞれ要素の最大数を 50 に設定しています。

```
char emp_name[50][10];
int emp_number[50];
float salary[50];
```

VARCHAR の配列も有効です。次の宣言は、有効なホスト言語宣言です。

```
VARCHAR v_array[10][30];
```

制限

オブジェクト型を除き、ポインタのホスト配列は宣言できません。

文字配列（文字列）を除き、SQL 文内で参照できるホスト配列は 1 次元に限定されています。したがって、次の例で宣言されている 2 次元配列は無効です。

```
int hi_lo_scores[25][25]; /* not allowed */
```

配列の最大サイズ

1 回のフェッチでアクセス可能な SQL 文の配列要素の最大数は 32K（あるいはプラットフォームと使用可能なメモリーによっては 32K 以上も可能）です。最大サイズを超えたホスト配列にアクセスすると、「パラメータ範囲外」ランタイム・エラーが生じます。文が無名 PL/SQL ブロックの場合、アクセス可能な配列要素数は 32512 をデータ型のサイズで割った値までに制限されます。

SQL 文での配列の使用

ホスト配列を INSERT 文、UPDATE 文、DELETE 文では入力変数として、また SELECT 文および FETCH 文の INTO 句では出力変数として使用できます。

ホスト配列に使用される埋込み SQL 構文は、単純ホスト変数に使用される埋込み SQL 構文とほとんど同じです。ただし、オプションの FOR 句で配列処理を制御できるという点では違いがあります。また、ホスト配列と単純ホスト変数を単一の SQL 文内で併用するときにも制限があります。

以降の項では、DML 文内でホスト配列を使用する方法を説明します。

ホスト配列の参照

単一の SQL 文で複数のホスト配列を使用する場合、要素の数は同じにする必要があります。同じでない場合には、「配列サイズ不一致」警告メッセージがプリコンパイル時に出了ます。この警告を無視すると、プリコンパイラは SQL 操作で最小数の要素を使用します。

次の例では、INSERT されるのは 25 行のみです。

```
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);
```

SQL 文のホスト配列に添字を付け、それをループで使用するによりデータを挿入またはフェッチできます。たとえば、次のようなループを使用して、配列内の 5 番目の要素ごとに INSERT できます。

```
for (i = 0; i < 50; i += 5)
      EXEC SQL INSERT INTO emp (empno, deptno)
            VALUES (:emp_number[i], :dept_number[i]);
```

ただし、処理する必要のある配列要素が連続している場合、ループでホスト配列を処理しないでください。単に、添字の付いていない配列名を SQL 文で使用してください。要素数 n のホスト配列を含む SQL 文は、 n 個の異なるスカラー変数を持つ同じ SQL 文として n 回実行するのと同様に扱われます。

標識配列の使用法

標識配列は、NULL を割り当ててホスト配列を入力し、出力ホスト配列で NULL または切り捨てられた値（文字列のみ）を検出する場合に使用できます。次の例は、標識配列で INSERT を行う方法を示しています。

```
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);
```

Oracle 制限事項

VALUES、SET、INSERT または WHERE 句でスカラーのホスト変数とホスト配列を併用することはできません。ホスト変数のうち 1 つでも配列があれば、すべてのホスト変数が配列である必要があります。

ホスト配列は UPDATE 文もしくは DELETE 文で CURRENT OF 句とともに使用できません。

ANSI 制限事項および要件

配列インタフェースは、ANSI/ISO の埋込み SQL 標準に対する Oracle の拡張要素です。ただし、MODE=ANSI でプリコンパイルしても、配列の SELECT および FETCH は許可されます。配列の指定では、必要に応じて FIPS フラガー・プリコンパイラ・オプションを使用してフラグできます。

配列を SELECT および FETCH するときは、必ず標識配列を使用します。このようにして、関連する出力ホスト配列内に NULL があるかどうかをテストできます。

プリコンパイラ・オプション DBMS=V7 または DBMS=V8 を指定してプリコンパイルする場合、NULL が SELECT または FETCH され、関連付けられた標識変数を持たないホスト変数に格納されると、Oracle は処理を停止し、*sqlca.sqlerrd[2]* を処理済の行数に設定して、エラーを戻します。

DBMS=V7 または DBMS=V8 の場合、Oracle は切捨てをエラーとみなしません。

配列への選択

ホスト配列は SELECT 文内の出力変数として使用できます。SELECT によって戻される最大行数が分かっている場合、その要素数（最大行数）でホスト配列を宣言してください。次の例では 3 つのホスト配列内にデータを直接選び出します。このとき SELECT が戻すのは 50 行以下と分かっているため、配列は要素数 50 で宣言します。

```
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;
```

この例では SELECT 文は 50 行までの行を戻します。選択される行数が 50 行に満たないとき、または 50 行のみを取り出すときはこの方法を使用します。ただし選択される行数が 50 行を超える場合は、この方法ではすべての行を取り出せません。この SELECT 文をもう 1 度実行しても、他に選択対象の行があっても最初の 50 行のみがまた戻されます。この場合は大きな配列を宣言するか、FETCH 文で使用するカーソルを宣言する必要があります。

宣言した要素数を超える行数が SELECT INTO 文によって戻されると、SELECT_ERROR=NO を指定していないかぎりエラー・メッセージが出されます。SELECT_ERROR オプションの詳細は 10-11 ページの「[プリコンパイラ・オプションの使用](#)」を参照してください。

カーソルのフェッチ

SELECT が戻す最大行数がわからない場合には、カーソルを宣言して、バッチでフェッチすることができます。

ループ内の一括フェッチによって多数の行を簡単に取り出せます。FETCH を実行するたびに、次に続く行の集まりがカレント・アクティブ・セットから戻ります。次の例では、20 行ずつまとめて行をフェッチします。

```
int emp_number[20];
float salary[20];

EXEC SQL DECLARE emp_cursor CURSOR FOR
      SELECT empno, sal FROM emp;

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 */
    ...
}
...
```

最後のフェッチで実際に戻された行数を必ずチェックして処理してください。8-6 ページの「[FETCH される行数](#)」を参照してください。

***sqlca.sqlerrd[2]* の使用方法**

INSERT 文、UPDATE 文、DELETE 文および SELECT INTO 文の場合は、*sqlca.sqlerrd[2]* に処理済の行数が記録されます。FETCH 文の場合は、処理した行の累積数が記録されます。

FETCH でホスト配列を使用しているときに、最後の繰返し実行によって戻された行数を確認するには、*sqlca.sqlerrd[2]* の現在の設定値を（別の変数内に保存した）前の値から減算します。次の例では、最後の FETCH が戻した行数を判断します。

```
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];
}
...
```

配列の処理中にエラーが発生したときにも *sqlca.sqlerrd[2]* が役に立ちます。処理はエラーを引き起こした行で停止するので、*sqlerrd[2]* には正常に処理された行数が格納されます。

FETCH される行数

FETCH はそれぞれ、最大でも配列の合計行数を戻します。次のような場合は最大行数より少ない行が戻ります。

- アクティブ・セットの最後に達したとき。「データが見つかりません」 Oracle エラー・コードは SQLCA 内で SQLCODE に戻されます。たとえば、要素数 100 の配列に行をフェッチしたとき 20 行しか戻されなかった場合にこれが起こります。
- フェッチ対象の行が行の集まり全体よりも少ないとき。たとえば、要素数 20 の配列に 70 行をフェッチするときにこの状態となります。つまり、3 回目のフェッチの後にはフェッチ対象の行は 10 行しか残っていません。
- 行の処理中にエラーが検出されたとき。FETCH は失敗に終わり、適用可能な Oracle エラー・コードが SQLCODE に戻ります。

戻された行の累積数は、このガイドでは *sqlerrd[2]* と記載している SQLCA 内の *sqlerrd* の 3 番目の要素に保存されます。これはオープン状態のすべてのカーソルに適用されます。次の例では、各カーソルの状態がそれぞれ更新されている様子がわかります。

```
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) */
```

サンプル・プログラム 3: ホスト配列

この項のデモ・プログラムでは、Pro*C/C++ で問合せを作成するときの配列の使用法を示しています。特に、SQLCA (*sqlca.sqlerrd[2]*) の「処理済の行数」に注目してください。SQLCA の詳細は、[第 9 章「ランタイム・エラーの処理」](#)を参照してください。このプログラムは、*demo* ディレクトリのファイル *sample3.pc* として、オンラインで使用可能です。

```
/*
 * 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
{
    int    emp_number[ARRAY_LENGTH];
    char   emp_name[ARRAY_LENGTH][NAME_LENGTH];
    float  salary[ARRAY_LENGTH];
} emp_rec;

/* Declare this program's functions. */
void print_rows();           /* produces program output */
void sql_error();           /* handles unrecoverable errors */

main()
{
    int  num_ret;             /* number of rows returned */

    /* 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 SQL OPEN c1;

    /* Initialize the number of rows. */
    num_ret = 0;

    /* Array fetch loop - ends when NOT FOUND becomes true. */
    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);
        num_ret = sqlca.sqlerrd[2];          /* Reset the number. */
    }
/* 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");

/* Disconnect from the database. */
    EXEC SQL COMMIT WORK RELEASE;
    exit(0);
}

void
print_rows(n)
int n;
{
    int i;

    printf("\nNumber      Employee      Salary");
    printf("\n-----      -          -");
    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]);
}

void
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);
}

```

制限

副問合せ文中を除き、SELECT 文の WHERE 句にホスト配列を使用することはできません。例は、8-15 ページの「WHERE 句の使用方法」を参照してください。

また SELECT 文または FETCH 文の INTO 句内では、単純ホスト変数とホスト配列の併用はできません。ホスト変数のうち 1 つでも配列があれば、すべてのホスト変数が配列である必要があります。

表 8-1 に、SELECT INTO 文で有効なホスト配列の使用を示します。

表 8-1 SELECT INTO で有効なホスト配列

INTO 句	WHERE 句	有効 / 無効
配列	配列	無効
スカラー	スカラー	有効
配列	スカラー	有効
スカラー	配列	無効

NULL 値のフェッチ

配列を SELECT および FETCH するときは、必ず標識配列を使用します。このようにして、関連する出力ホスト配列内に NULL があるかどうかをテストできます。

DBMS = V7 のときに、標識配列に関連付けられていないホスト配列に NULL 列値を SELECT または FETCH すると、処理が停止し、*sqlerrd[2]* が処理済行数に設定されてエラー・メッセージが出されます。

切り捨てられた値のフェッチ

DBMS=V7 のときは、切り捨てによって警告メッセージが発行されますが、処理は継続されます。

また、配列を SELECT および FETCH するときは必ず標識配列を使用します。そうすれば、Oracle が 1 つ以上の切り捨てられた列値を出力ホスト配列に割り当てたときに、列値の元の長さが対応する標識配列内に保存されます。

配列での挿入

ホスト配列は INSERT 文内の入力変数として使用できます。プログラムが INSERT 文を実行する前に、プログラム内にデータが含まれている配列があることを確認してください。

配列内に不適切な要素があるときは、FOR 句を使用して INSERT 対象の行数を制御できます。詳細は 8-13 ページの「FOR 句の使用方法」を参照してください。

ホスト配列による INSERT の例を次に示します。

```
char   emp_name[50][20];
int    emp_number[50];
float  salary[50];
/* populate the host arrays */
...
EXEC SQL INSERT INTO EMP (ENAME, EMPNO, SAL)
      VALUES (:emp_name, :emp_number, :salary);
```

挿入された行の累積数は処理済行数 *sqlca.sqlerrd[2]* に保存されます。

次の例では、1 度に 1 行ずつ INSERT されます。このとき挿入するそれぞれの行についてサーバーをコールする必要があるため、この方法は前の例に比べるとかなり効率は悪くなります。

```
for (i = 0; i < array_size; i++)
    EXEC SQL INSERT INTO emp (ename, empno, sal)
          VALUES (:emp_name[i], :emp_number[i], :salary[i]);
```

制限

INSERT 文の VALUES 句内ではポインタの配列は使用できません。つまり配列要素はすべてデータ項目である必要があります。

INSERT 文の VALUES 句では、スカラー・ホスト変数とホスト配列は併用できません。ホスト変数のうち 1 つでも配列があれば、すべてのホスト変数が配列である必要があります。

配列での更新

次の例に示すように、ホスト配列を UPDATE 文内の入力変数として使用できます。

```
int    emp_number[50];
float  salary[50];
/* populate the host arrays */
EXEC SQL UPDATE emp SET sal = :salary
      WHERE EMPNO = :emp_number;
```

更新された行の累積数は *sqlerrd[2]* に保存されます。その累積数には更新カスケードによって処理された行は含まれていません。

配列内に不適切な要素がある場合は、埋込み SQL の FOR 句を使用して更新対象の行数を制御できます。

前の例では一意キー (EMP_NUMBER) を使用した一般的な更新を示しています。このとき配列要素はそれぞれ更新対象の行数を 1 行に制限しています。次の例では、それぞれの配列要素の削除対象の行数が複数行になっています。

```
char job_title [10] [20];
float commission[10];

...

EXEC SQL UPDATE emp SET comm = :commission
    WHERE job = :job_title;
```

制限

UPDATE 文の SET 句または WHERE 句内で、単純ホスト変数とホスト配列を併用することはお薦めできません。ホスト変数のうち 1 つでも配列があれば、すべてのホスト変数が配列である必要があります。さらに、SET 句でホスト配列を使用するときには、WHERE 句の要素数と同じ数のものを使用してください。

UPDATE 文の CURRENT OF 句ではホスト配列は使用できません。別の方法は 8-26 ページの「[CURRENT OF の疑似実行](#)」を参照してください。

[表 8-2](#) では、UPDATE 文で有効なホスト配列の使用を示しています。

表 8-2 UPDATE 文で有効なホスト配列

SET 句	WHERE 句	有効 / 無効
配列	配列	有効
スカラー	スカラー	有効
配列	スカラー	無効
スカラー	配列	無効

配列での削除

DELETE 文内でもホスト配列を入力変数として使用できます。これは、WHERE 句内のホスト配列の連続した要素を使用して DELETE 文を繰り返し実行するときと同様です。つまり 1 回の実行で表から 0 行、1 行または複数行が削除されます。

ホスト配列による削除の例を次に示します。

```
...
int emp_number[50];

/* populate the host array */
...
EXEC SQL DELETE FROM emp
    WHERE empno = :emp_number;
```

削除された行の累積数は *sqlerrd[2]* に保存されます。その累積数には、削除カスケードによって処理された行は含まれていません。

この例では一意キー（EMP_NUMBER）を使用した一般的な削除を示しています。このとき配列要素はそれぞれ削除対象の行数を 1 行に制限しています。次の例では、それぞれの配列要素の削除対象の行数が複数行になっています。

```
...
char job_title[10][20];

/* populate the host array */
...
EXEC SQL DELETE FROM emp
      WHERE job = :job_title;
...
```

制限

DELETE 文の WHERE 句内では、単純ホスト変数とホスト配列を併用することはできません。ホスト変数のうち 1 つでも配列があれば、すべてのホスト変数が配列である必要があります。

DELETE 文の CURRENT OF 句ではホスト配列は使用できません。別の方法については 8-26 ページの「[CURRENT OF の疑似実行](#)」を参照してください。

FOR 句の使用法

埋込み SQL でオプションの FOR 句を使用すると、次に示す SQL 文が処理する配列要素の数を設定できます。

- DELETE
- EXECUTE
- FETCH
- INSERT
- OPEN
- UPDATE

特に UPDATE 文、INSERT 文および DELETE 文内で FOR 句を使用すると便利です。これらの文に配列全体を使用する必要がないときもあります。次の例に示すように、FOR 句を使用すると、使用する要素数を任意の数に制限できます。

```
char emp_name[100][20];
float salary[100];
int rows_to_insert;
```

```
/* populate the host arrays */
rows_to_insert = 25;          /* set FOR-clause variable */
EXEC SQL FOR :rows_to_insert /* will process only 25 rows */
    INSERT INTO emp (ename, sal)
    VALUES (:emp_name, :salary);
```

FOR 句では、配列要素をカウントするための整数型のホスト変数を使用できる他、整数リテラルも使用できます。整数値をとる C の複合式を使用することはできません。たとえば、整数式を使用する次の文は無効です。

```
EXEC SQL FOR :rows_to_insert + 5          /* illegal */
    INSERT INTO emp (ename, empno, sal)
    VALUES (:emp_name, :emp_number, :salary);
```

FOR 句の変数によって、処理される配列の要素数が指定されます。この値が最小の配列サイズを超過していないことを確認してください。この値は内部的には記号のついていない数量として扱われます。記号付きホスト変数を使用して負の値を渡そうとすると、予期できない動作が起こる結果になります。

制限

ふたつの制限が FOR 句の意味を明確に保ちます。FOR 句は SELECT 文では使用できません。また CURRENT OF 句とともに使用することはできません。

SELECT 文中

SELECT 文中で FOR 句を使用すると、エラー・メッセージが戻されます。

FOR 句は意味があいまいなため、SELECT 文中では使用できません。「この SELECT 文を n 回実行する」という意味でしょうか。それとも、「この SELECT 文を 1 回実行し、 n 行戻す」という意味でしょうか。前者の解釈の問題は、各実行が複数の行を戻すことです。後者の解釈では、次に示すように、カーソルを宣言してから FETCH 文中で FOR 句を使用することをお勧めします。

```
EXEC SQL FOR :limit FETCH emp_cursor INTO ...
```

CURRENT OF 句を使用する場合

次の例に示すように、UPDATE 文または DELETE 文中で CURRENT OF 句を使用すると、FETCH 文によって戻される最後の行を参照できます。

```
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;
```

ただし、CURRENT OF 句と FOR 句の併用はできません。次の文は *limit* の論理値が 1 に限定されているため無効です。（つまりカレント行を 1 回のみ更新または削除できます。）

```
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;
```

WHERE 句の使用法

Oracle では要素数 n のホスト配列を含む SQL 文を、同一の SQL 文を n 個の異なるスカラー変数（個々の配列要素）で n 回実行するのと同様に扱います。このような扱いがあいまいなときにかぎり、プリコンパイラによってエラー・メッセージが発行されます。

たとえば次の宣言をしたとき、

```
int mgr_number[50];
char job_title[50][20];
```

次の文を、

```
EXEC SQL SELECT mgr INTO :mgr_number FROM emp
WHERE job = :job_title;
```

次の架空の文のように処理した場合、不明瞭となります。

```
for (i = 0; i < 50; i++)
    SELECT mgr INTO :mgr_number[i] FROM emp
    WHERE job = :job_title[i];
```

WHERE 句の検索条件を満たす行が複数あっても、データの受取りに使用できる出力変数が 1 つしかないためです。したがって、エラー・メッセージが発行されます。

一方、次の文を

```
EXEC SQL UPDATE emp SET mgr = :mgr_number
    WHERE empno IN (SELECT empno FROM emp
    WHERE job = :job_title);
```

次の架空の文のように処理した場合には、不明瞭とはなりません。

```
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]);
```

これは、それぞれの *job_title* が複数の行に一致する場合でも、WHERE 句内の *job_title* に一致する各行について SET 句内に *mgr_number* が指定されているためです。それぞれの *job_title* に一致する行すべてに、同一の *mgr_number* を SET できます。したがってエラー・メッセージは発行されません。

構造体配列

スカラーの配列を使用すると、1つの列で複数行を操作できます。またスカラーの構造体を使用すると、1つの行を複数の列で操作できます。

しかし、複数の列で複数の行を操作するときは、これまでは複数のスカラーの並列配列を個別にまたは1つの構造体の中でカプセル化して割り当てる必要がありました。この方法よりも、このデータ構造体を複数の構造体からなる1つの配列として再編成の方が便利です。

Pro*C/C++ では構造体の配列をサポートし、アプリケーション・プログラマはC 構造体の配列を使用して複数行および複数列の操作を実行できます。この拡張要素は、ユーザー・データの処理をより簡単にするために、Pro*C/C++ ではスカラーの構造体の単純な配列を埋込み SQL 文でバインド変数として処理できます。これで、プログラミングがさらに直観的になり、データ編成もはるかに自由にできます。

Pro*C/C++ では、構造体の配列がバインド変数としてサポートされるのみでなく、標識変数構造体の配列を構造体配列の宣言と併用できます。

注意： 構造体を PL/SQL レコードにバインドすることと、構造体の配列を PL/SQL レコードの表にバインドすることはこの新機能の一部ではありません。また、構造体の配列を埋込み PL/SQL ブロック内で使用することもできません。これ以外の制限は 8-17 ページの「[構造体の配列の制限](#)」を参照してください。

構造体の配列は、複数の列で複数の行を操作するために使用され、通常次のように使用されると考えてください。

- SELECT 文または FETCH 文での出力バインド変数として。
- INSERT 文の VALUES 句での入力バインド変数として。

構造体の配列の使用方法

構造体の配列という概念は、C のプログラマにとって特に目新しいものではありません。しかし、複数の並列配列からなる1つの構造体と比較してみると、データの格納方法に考え方の違いがあります。

複数の並列配列からなる1つの構造体では、個々の列のデータが連続して格納されます。一方、構造体の配列では、列のデータは**インタリーブ**されます。この場合、配列内の各列の間は、その構造体内の他の列に必要な空白で区切られます。この空白は「ストライド (*stride*)」と呼ばれます。

構造体の配列の制限

Pro*C/C++ では、構造体の配列の使用方法に次の制限が適用されます。

- 構造体の配列（通常の構造体による）を埋込み PL/SQL ブロック内で使用できません。
- 構造体の配列を WHERE 句や FROM 句で使用できません。
- 構造体の配列は Oracle 動的 SQL 方法 4 では使用できません。ANSI 動的 SQL では使用できます。詳細は、[第 14 章「ANSI 動的 SQL」](#)を参照してください。
- 構造体の配列を UPDATE 文の SET 句で使用できません。

構造体の配列の宣言には実際的な違いはありません。しかし、構造体の配列を使用する場合には留意事項がいくつかあります。

構造体の配列の宣言

Pro*C/C++ アプリケーションで使用する構造体の配列を宣言する場合、プログラマは必ず次の点に留意してください。

- 構造体には必ず構造体タグを付けてください。例は次のとおりです。

```
struct person {  
    char name[15];  
    int  VARCHARage;  
} people[10];
```

このコード・セグメントでは、person 変数が、構造体のタグです。このタグによって、プリコンパイラは構造体の名前を使用してストライドのサイズを計算できます。

- 構造体のメンバーを配列にしないでください。ただし、**CHAR** や **VARCHAR** などのキャラクタ・タイプの場合には、この規則は適用されません。つまり、このような型の変数の宣言で配列の構文が使用されるためです。
- **CHAR** および **VARCHAR** のメンバーは 2 次元でない可能性があります。
- ネストされた構造体は構造体の配列のメンバーになることはできません。Pro*C/C++ の旧リリースでは、ネストされた構造体はサポートされていなかったため、この制限は新しいものではありません。
- 構造体自体のサイズは、符号付き 4 バイト数に許される最大値を超えないようにしてください。通常、この最大値は 2GB です。

構造体の配列の使用方法に前述の制限が適用されている場合、Pro*C/C++ では次の宣言は有効です。

```
struct department {  
    int deptno;  
    char dname[15];  
    char loc[14];
```

```
} dept[4];
```

一方、次の宣言は無効です。

```
struct {                /* the struct is missing a structure tag */
    int empno[15];       /* struct members may not be arrays */
    char ename[15][10]; /* character types may not be 2-dimensional */
    struct nested {
        int salary; /* nested struct not permitted in array of structs */
    } sal_struct;
} bad[15];
```

データ型の同値化は構造体の配列自体や構造体内の個々のフィールドには適用できないことも重要な留意点です。たとえば、`empno` が前述の無効な構造体内の配列として宣言されていなければ、次の文は無効です。

```
exec sql var bad[3].empno is integer(4);
```

プリコンパイラには、構造体の配列の中の個々の構造体要素を追跡し記録する機能はありません。しかし、次を実行すれば、思いどおりの結果が得られます。

```
typedef int myint;
exec sql type myint is integer(4);

struct equiv {
    myint empno; /* now legally considered an integer(4) datatype */
    ...
} ok[15];
```

Pro*C/C++ の以前のバージョンでは、個別の配列項目の同値化がサポートされていなかったため、これは予測できたことといえます。たとえば、次のスカラー配列宣言は何が有効で、何が有効でないかを示しています。

```
int empno[15];
exec sql var empno[3] is integer(4); /* illegal */

myint empno[15]; /* legal */
```

基本的には、個々の配列項目を同値化することはできません。

標識変数の使用方法

標識変数は、構造体の配列の宣言でも、通常の構造体の宣言の場合とほとんど同じはたらしをします。構造体の標識配列の宣言は 8-17 ページの「[構造体の配列の宣言](#)」で述べた構造体の配列の規則に従いながら、さらに次の規則にも従います。

- 標識変数構造体に含まれるフィールド数は、対応する構造体の配列に含まれるフィールド数と同じか、またはそれ以下である必要があります。
- フィールドの順序は、対応する構造体の配列のメンバーの順序に一致する必要があります。
- 標識構造体に含まれるすべての要素のデータ型は、**SHORT** である必要があります。
- 標識配列のサイズは、ホスト変数で宣言されたサイズと同じか、またはそれ以上である必要があります。ホスト変数で宣言されたサイズより大きい値は指定できますが、それより小さい値は指定できません。

これらの規則のほとんどには、Pro*C/C++ の以前のバージョンでの構造体使用の規則が反映されています。配列制限も以前使用されたスカラーの配列に対するものと同じです。

これらの規則が適用されている場合に、次のように構造体を宣言するものとします。

```
struct department {
    int deptno;
    char dname[15];
    char loc[14];
} dept[4];
```

次の標識変数の構造体の宣言は有効です。

```
struct department_ind {
    short deptno_ind;
    short dname_ind;
    short loc_ind;
} dept_ind[4];
```

一方、次は標識変数として無効です。

```
struct{
    /* missing indicator structure tag */
    int deptno_ind; /* indicator variable not of type short */
    short dname_ind[15]; /* array element forbidden in indicator struct */
    short loc_ind[14]; /* array element forbidden in indicator struct */
} bad_ind[2]; /* indicator array size is smaller than host array */
```

構造体の配列へのポインタの宣言

構造体の配列へのポインタを宣言した方が適切な場合があります。これにより、構造体の配列へのポインタを他の関数に渡したり、埋込み SQL 文で直接指定できます。

注意：構造体の配列へのポインタが参照する配列の長さはプリコンパイルの間は不明です。このため、埋込み SQL 文で構造体の配列へのポインタ型になっているバインド変数を使用するときには、明示的な FOR 句を使用してください。

ただし、FOR 句は埋込み SQL SELECT 文では指定できません。したがって、データを取り出して構造体の配列へのポインタに入れる場合には、必ずカーソルと FETCH 文を FOR 句とともに明示的に指定してください。

例

次の例は、Pro*C/C++ での構造体の配列の機能について、様々な使用方法を示しています。

例 1 スカラーの構造体の単純な配列

次の構造体の宣言を指定したとします。

```
struct department {  
    int deptno;  
    char dname[15];  
    char loc[14];  
} my_dept[4];
```

次のように dept データを選択して my_dept に入れることができます。

```
exec sql select * into :my_dept from dept;
```

また、最初に my_dept を移入してから、dept 表に一括して挿入できます。

```
exec sql insert into dept values (:my_dept);
```

標識変数を指定するには、構造体のパラレル標識配列を宣言します。

```
struct department_ind {  
    short deptno_ind;  
    short dname_ind;  
    short loc_ind;  
} my_dept_ind[4];
```

データの選択に使用される問合せは、標識変数の指定が追加されたこと以外は同じです。

```
exec sql select * into :my_dept indicator :my_dept_ind from dept;
```

同様に、データの挿入時にも標識を指定できます。

```
exec sql insert into dept values (:my_dept indicator :my_dept_ind);
```


例 2 スカラーの配列と構造体の配列との組合せ使用

Pro*C/C++ の旧リリースと同様に、ユーザー・データのバルク処理機能に複数の配列を使用する場合は、各配列のサイズを同じにする必要があります。同じにしないと、最も小さい配列のサイズが選択され、残りの配列は変更されません。

次の宣言を指定したとします。

```
struct employee {
    int empno;
    char ename[11];
} emp[14];

float sal[14];
float comm[14];
```

ただ 1 つの問合せで、すべての列に対して複数の行を選択できます。

```
exec sql select empno, ename, sal, comm into :emp, :sal, :comm from emp;
```

また、コミッションの列の値が NULL かどうかを確認する必要があります。次のように宣言すれば、1 つの標識配列を指定するのみで済みます。

```
short comm_ind[14];
...
exec sql select empno, ename, sal, comm
    into :emp, :sal, :comm indicator :comm_ind from emp;
```

問合せからの標識情報をすべてカプセル化した構造体の標識配列を 1 つのみ宣言することはできないので注意してください。次の例を考えます。

```
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;
```

この列は無効です。(また望ましくもありません。) 前述の文には SELECT...INTO リスト全体ではなく、comm 列しかない標識配列が対応づけられています。

構造体の配列と sal、comm および comm_ind 配列に希望するデータを挿入する場合、その挿入方法は非常に簡単です。

```
exec sql insert into emp (empno, ename, sal, comm)
    values (:emp, :sal, :comm indicator :comm_ind);
```

例 3 複数の構造体の配列と 1 つのカーソルとの組合せ使用

次の宣言をこの例として使用します。

```
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];
```

Oracle のプログラムでは、構造体の配列を次のように指定できます。

```
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;
```

FOR 句の使用法 Oracle では FOR 句を使用した場合も、取り出す行数について FETCH を指示できます。FOR 句は、SELECT 文が使用されている場合には指定できませんが、INSERT 文や FETCH 文の場合には指定できます。

元の宣言に次を追加します。

```
int limit = 10;
```

次は、それに対応したコードの例です。

```
exec sql for :limit
```

```
fetch c into :emp, :wage indicator :wage_ind;
```

例 4 個々の配列メンバーおよび構造体メンバーの参照

これまでの Pro*C/C++ リリースで、配列の参照は、構造体の配列内の単一の構造体に対して許可されています。これによりバインド式はスカラーの単純な構造体で解決できるため、次は有効です。

```
exec sql select * into :dept[3] from emp;
```

次の例に示すように、構造体の配列内の特定の構造体のスカラーのメンバーを 1 つずつ参照できます。

```
exec sql select dname into :dept[3].dname from dept where ...;
```

この場合には当然、問合せは 1 行で指定する必要があるため、1 行のみ選択してこのバインド式で表される変数に代入します。

例 5 標識変数の使用（特殊な場合）

これまでの Pro*C/C++ リリースでは、標識変数構造体のフィールド数はそれに対応するバインド変数構造体と同じである必要があります。構造体を通常に指定する場合には、この制限は緩和されています。前述の構造体の標識配列についてのガイドラインに従えば、次の例を構成できます。

```
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;
```

標識変数はバインド値と 1 対 1 でマップされます。これらは、最初のフィールドから、対応した順番にマップされます。

ただし、他のフィールドでフェッチされた値が NULL であったり、標識が付いていなかったりすると、次のエラーが発生するので注意してください。

ORA-01405: 取り出した列の値が NULL です。

エラーが発生します。例として、sal には標識がないため、sal が NULL になる場合があります。

次のように構造体の配列を変更したとします。

```
struct employee {  
    int empno;  
    char ename[10];  
    float sal;  
    float comm;  
} emp[15];
```

しかし、前述と同じ構造体の標識配列がまだ使用されています。

標識のマッピングは対応した順番に実行されるので、comm バインド変数に標識がないまま comm 標識が empno フィールドにマップされ、再度 ORA-01405 エラーとなります。

通常、対応するバインド変数の構造体よりもフィールド数の少ない標識変数構造体を指定したときに ORA-01405 エラーが発生しないようにするには、最初に NULL になる可能性がある属性を持ってきて順番に並べる必要があります。

この例は、配列なし構造体を使用すれば複数の列を 1 行フェッチできるように簡単に変更でき、標識変数構造体が次のように宣言されている場合と同等のはたらきをすると考えられます。

```
struct employee_ind {  
    short comm;  
    short sal;  
    short empno;  
    short ename;  
} emp_ind;
```

Pro*C/C++ では標識変数構造体のフィールド数と対応する値の構造体のフィールド数が同じである必要がなくなったため、前述の例はこれまで Pro*C/C++ で無効でしたが現在は有効になりました。

Oracle の標識変数構造体は、次のように簡単に指定できます。

```
struct employee_ind {  
    short comm;  
} emp_ind;
```

次のように配列なし構造体である emp および emp_ind を使用すると、1 つの行をフェッチできます。

```
exec sql fetch comm, sal, empno, ename
into :emp indicator :emp_ind from emp;
```

この場合にも comm 標識が comm バインド変数にどうマップされるかに注意してください。

例 6 構造体の配列へのポインタの使用

この例では、構造体の配列へのポインタの使用方法を示します。

次の型の宣言を考えます。

```
typedef struct dept {
    int deptno;
    char dname[15];
    char loc[14];
} dept;
```

その型の構造体の配列へのポインタを操作すると、様々な処理を実行できます。たとえば、構造体の配列へのポインタを他の関数に渡すことができます。

```
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;
}
```

このような関数を起動するには、例に示すように構造体の配列のアドレスを渡します。

```
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' */
```

一部の埋込み SQL 文では、構造体の配列へのポインタを直接指定するのみで関数を起動できます。

```
exec sql for :n insert into dept values (:dptr);
```

FOR 句の使用方法を間違えないように、十分に注意してください。

CURRENT OF の疑似実行

DELETE 文または UPDATE 文で CURRENT OF *cursor* 句を使用すると、カーソルから最後にフェッチされた行を参照できます。（詳細は、6-16 ページの「[CURRENT OF 句の使用方法](#)」を参照してください。）ただし、CURRENT OF 句とホスト配列の併用はできません。かわりに各行の ROWID を選択しておいて、更新または削除するときにその値を使用してカレント行を識別してください。次に例を示します。

```
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;
...
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;
}
```

ただし、ここでは FOR UPDATE OF 句が使用されていないため、フェッチされた行はロックされません。（CURRENT OF 句なしでは FOR UPDATE OF を使用することができません。）したがって、ある行を読み取ってからその行を削除する前に、別のユーザーがその行を変更すると結果に矛盾が生じることがあります。

ランタイム・エラーの処理

アプリケーション・プログラムでは、ランタイム・エラーを予想し、そのエラーをリカバリするように対処する必要があります。この章では、エラーの報告およびリカバリについて詳しく説明します。SQL 通信領域 (SQLCA)、WHENEVER 宣言文および SQLSTATE 状態変数を使用してエラーおよび状態の変化を処理する方法を説明します。さらに Oracle 通信領域 (ORACA) による問題点の診断方法も紹介します。この章は次のトピックで構成されています。

- エラー処理の必要性
- エラー処理の代替手段
- SQLSTATE 状態変数
- SQLCODE の宣言
- SQLCA を使用したエラー報告の主要コンポーネント
- SQL コミュニケーション領域 (SQLCA) の使用
- エラー・メッセージの全文の取得
- WHENEVER 宣言文の使用
- SQL 文のテキスト取得
- ORACLE コミュニケーション領域 (ORACA) の使用

エラー処理の必要性

どのようなアプリケーション・プログラムでも、その大部分をエラー処理に当てる必要があります。エラー処理の主な目的は、エラーが発生してもプログラムの処理を続行できるようにすることです。エラーは設計ミス、コーディングの誤り、ハードウェア障害、誤ったユーザー入力をはじめ、様々な原因で発生します。

潜在的なエラーをすべて予測するのは無理ですが、プログラムにとって意味のある特定の種類のエラーについてアクションを考えることができます。Pro*C/C++ プリコンパイラでは、エラー処理は SQL 文の実行エラーの検出およびリカバリを意味します。

「値が切り捨てられました」などの警告や「データの終わり」などの状態変化も処理できるよう準備しておくことができます。

INSERT 文、UPDATE 文または DELETE 文では表内の処理対象行をすべて処理する前にエラーが発生することがあるため、SQL DML 文を実行するたびにエラー条件および警告条件を調べることに重要です。

エラー処理の代替手段

アプリケーションの状態の変化およびエラーを検出するにはいくつかの手段があります。この章ではそれらの手段について説明しますが、特定の手段はオススメしません。最終的にどの手段を使用するかは、作成中のアプリケーション・プログラムまたはツールがどのように設計されているかによって決まります。

状態変数

状態変数 SQLSTATE または SQLCODE を別に宣言し、実行 SQL 文の後の値をそれぞれ調べて、適切なアクションを取ることができます。アクションとしては、まずエラー報告関数をコールし、エラーがリカバリ不能なときはプログラムを終了します。または、データを調整するか変数を制御して、アクションを再試行することもできます。これらの状態変数に関する完全な情報は 9-3 ページの「[SQLSTATE 状態変数](#)」および 9-15 ページの「[SQLCODE の宣言](#)」を参照してください。

SQL コミュニケーション領域

もう 1 つの手段は、プログラムに SQL 通信領域構造体 (*sqlca*) を組み込むことです。実行時に SQL 文が Oracle によって処理されると、この構造体に含まれるコンポーネントに値が格納されます。

注意：このマニュアルでは *sqlca* 構造体は一般に「SQL コミュニケーション領域」の頭字語 (SQLCA) を使用して表記します。このガイドで C の **構造体** の特定のコンポーネントに言及するときには、構造体の名称 (*sqlca*) を使用します。

SQLCA はヘッダー・ファイル *sqlca.h* に定義されています。次の文のどちらかを使用して SQLCA をプログラムに組み込みます。

- EXEC SQL INCLUDE SQLCA;
- #include <sqlca.h>

Oracle はすべての実行 SQL 文の後で SQLCA を更新します。(SQLCA の値は宣言文の後では変更されません。) SQLCA に格納されている Oracle リターン・コードをチェックすることによって、プログラムは SQL 文の結果を判断できます。この判断には次の 2 通りの方法があります。

- WHENEVER 宣言文による暗黙的なチェック
- SQLCA コンポーネントの明示的なチェック

WHENEVER 宣言文を使用するか、SQLCA コンポーネントの明示的なチェックを記述できます。またはその両方を同時に使用できます。

SQLCA で最も頻繁に使用されるコンポーネントは、状態変数 (*sqlca.sqlcode*) およびエラー・コード (*sqlca.sqlerrm.sqlerrmc*) に関連するテキストです。他のコンポーネントには警告フラグおよび SQL 文の処理に関する各種の情報が格納されます。SQLCA 構造体の完全な情報は 9-17 ページの「[SQL コミュニケーション領域 \(SQLCA\) の使用](#)」を参照してください。

注意: SQLCODE (大文字) は常に個別の状態変数のことです。SQLCA のコンポーネントではありません。SQLCODE は、**long** 型整数として宣言されます。SQLCA のコンポーネント *sqlcode* を参照する場合は、常に完全修飾名 *sqlca.sqlcode* が使用されます。

ランタイム・エラーについて、SQLCA に格納されている情報よりも詳細な情報が必要なときに ORACA を使用できます。ORACA は、Oracle の通信を処理する C の構造体です。この中にはカーソルの統計情報、カレント SQL 文に関する情報、オプションの設定、システムの統計情報が含まれます。ORACA の完全な情報は 35 ページの「[ORACLE コミュニケーション領域 \(ORACA\) の使用](#)」を参照してください。

SQLSTATE 状態変数

プリコンパイラのコマンドライン・オプション MODE は、ANSI/ISO への準拠を制御します。MODE=ANSI のとき、SQLCA データ構造体の宣言はオプションです。ただし、SQLCODE という状態変数は別に宣言する必要があります。SQL92 では、SQLSTATE という類似した状態変数が指定されています。SQLSTATE は、SQLCODE とともに使用しても別々に使用してもかまいません。

SQL 文の実行後、Oracle Server は現在の適用範囲内の SQLSTATE 変数にステータス・コードを戻します。ステータス・コードは、SQL 文が正常に実行されたか例外 (エラーまたは警告条件) が発生したかを示します。「相互操作性」(システム間で情報を簡単に交換する機能) を高めるために、共通の SQL 例外がすべて SQL92 によってあらかじめ定義されています。

SQLCODE にはエラー・コードのみが格納されるのに対し、SQLSTATE にはエラー・コードおよび警告コードが格納されます。さらに、SQLSTATE の報告のメカニズムには、標準化されたコード化方式が採用されています。このため、SQLSTATE は状態変数として優先されます。SQL92 では SQLCODE は SQL89 との互換性を保つためにのみ維持されている「否定的機能」で、標準の将来のバージョンからは削除される可能性があります。

SQLSTATE の宣言

MODE=ANSI のときは、SQLSTATE または SQLCODE を宣言する必要があります。SQLCA の宣言はオプションです。MODE=ORACLE のときは、SQLSTATE を宣言しても無視されません。

SQLCODE では符号付き整数を格納し、宣言文の外で宣言できますが、SQLSTATE では NULL 終了記号を付けた 5 文字の文字列を格納し、宣言文の中で宣言する必要があります。次のように SQLSTATE を宣言します。

```
char SQLSTATE[6]; /* Upper case is required. */
```

注意：SQLSTATE は 6 文字ちょうどのディメンションで宣言される必要があります。

SQLSTATE の値

SQLSTATE ステータス・コードは、2 文字のクラス・コードおよびその後続く 3 文字のサブクラス・コードで構成されます。クラス・コード 00（「正常終了」）以外のときには、クラス・コードは例外のカテゴリを示します。また、サブクラス・コード 000（「適用外」）以外では、サブクラス・コードはそのカテゴリ内の特定の例外を示します。たとえば、SQLSTATE の値「22012」はクラス・コード 22（「データ例外」）とサブクラス・コード 012（「ゼロ除算」）を示します。

SQLSTATE 値の 5 文字はそれぞれ数字（0～9）または大文字の英文字（A～Z）で構成されます。0～4 の範囲の数字、または A～H の範囲の文字で始まるクラス・コードは、事前定義済の条件（SQL92 で定義されている）用に確保されています。他のすべてのクラス・コードは処理系定義の条件用に確保されています。事前定義済のクラスのうち、0～4 の範囲の数字および A～H の範囲の文字で始まるサブクラス・コードは、事前定義済の副条件用に確保されています。他のサブクラス・コードはすべて実装時定義済の副条件用に確保されています。[図 9-1](#) にコーディングの概要を示します。

図 9-1 SQLSTATE コーディング概要

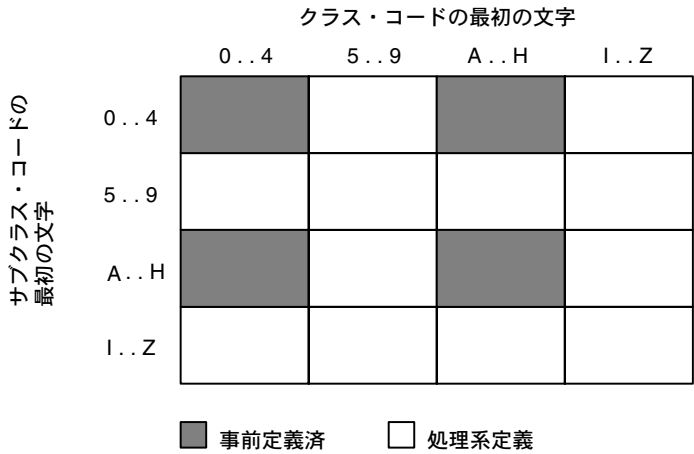


表 9-1 に SQL92 で事前定義済のクラスを示します。

表 9-1 事前定義済クラス

CLASS	条件
00	正常終了
01	警告
02	データなし
07	動的 SQL エラー
08	接続例外
0A	サポートされていない機能
21	制約違反
22	データ例外
23	整合性制約違反
24	カーソル状態が無効
25	トランザクション状態が無効
26	SQL 文名が無効
27	トリガー・データの変更違反
28	認証の指定が無効
2A	直接 SQL 構文エラーまたはアクセス規則違反
2B	依存権限記述子がまだ存在している
2C	キャラクタ・セット名が無効
2D	トランザクションの終了が無効
2E	接続名が無効
33	SQL 記述子名が無効
34	カーソル名が無効
35	条件番号が無効
37	動的 SQL 構文エラーまたはアクセス規則違反
3C	カーソル名があいまい
3D	カタログ名が無効
3F	スキーマ名が無効
40	トランザクションのロールバック

表 9-1 事前定義済クラス

CLASS	条件
42	構文エラーまたはアクセス規則違反
44	WITH_CHECK_OPTION 指定違反
HZ	リモート・データベース・アクセス

注意：クラス・コード HZ は国際標準 ISO/IEC DIS 9579-2、「リモート・データベース・アクセス」で定義された条件に予約されています。

表 9-2 に SQLSTATE ステータス・コードと条件の Oracle エラーとの対応を示します。範囲 60000 ～ 99999 のステータス・コードは実装時に定義されています。

表 9-2 SQLSTATE ステータス・コード

コード	条件	Oracle エラー
00000	正常終了	ORA-00000
01000	警告	
01001	カーソル操作の競合	
01002	切断エラー	
01003	集合関数で NULL 値が削除されている	
01004	文字列データの右側切捨て	
01005	項目記述子領域が不十分	
01006	権限が取り消されていない	
01007	権限が付与されていない	
01008	暗黙のゼロビットの埋込み	
01009	情報スキーマの検索条件が長すぎる	
0100A	情報スキーマの問合せ式が長すぎる	
02000	データなし	ORA-01095 ORA-01403
07000	動的 SQL エラー	
07001	USING 句がパラメータの指定と一致しない	
07002	USING 句がターゲットの指定と一致しない	
07003	カーソル仕様が実行できない	
07004	動的パラメータには USING 句が必要	
07005	作成された文がカーソル仕様ではない	
07006	制限付きのデータ型属性違反	
07007	結果コンポーネントには USING 句が必要、 記述子の数が無効	
07008	記述子の数が無効	SQL-02126
07009	記述子の索引が無効	
08000	接続例外	
08001	SQL クライアントは SQL 接続を確立できない	
08002	接続名を使用中	

表 9-2 SQLSTATE ステータス・コード

コード	条件	Oracle エラー
08003	接続が存在しない	SQL-02121
08004	SQL サーバーが SQL 接続を拒否した	
08006	接続障害	
08007	トランザクションの結果が不明	
0A000	サポートされていない機能	ORA-03000 ～ 03099
0A001	複数サーバー・トランザクション	
21000	制約違反	ORA-01427 SQL-02112
22000	データ例外	
22001	文字列データの右側切捨て。	ORA-01406
22002	NULL 値 - 標識パラメータなし	SQL-02124
22003	数値が範囲外	ORA-01426
22005	割当てのエラー	
22007	日時書式が無効	
22008	日時フィールドのオーバーフロー	ORA-01800 ～ 01899
22009	時間帯の変位値が無効	
22011	副文字列のエラー	
22012	0 による除算	ORA-01476
22015	間隔フィールドのオーバーフロー	
22018	キャストの文字値が無効	
22019	エスケープ文字が無効	ORA-00911
22021	レポートリに文字がない	
22022	標識のオーバーフロー	ORA-01411
22023	パラメータ値が無効	ORA-01025 ORA-04000 ～ 04019
22024	C 文字列が未終了	ORA-01479 ORA-01480

表 9-2 SQLSTATE ステータス・コード

コード	条件	Oracle エラー
22025	エスケープ文字の列が無効	ORA-01424 ORA-01425
22026	文字列データの長さが不一致	ORA-01401
22027	切捨てエラー	
23000	整合性制約違反	ORA-02290 ～ 02299
24000	カーソル状態が無効	ORA-01002 ORA-01003 SQL-02114 SQL-02117
25000	トランザクション状態が無効	SQL-02118
26000	SQL 文名が無効	
27000	トリガー・データの変更違反	
28000	認証の指定が無効	
2A000	直接 SQL 構文エラーまたはアクセス規則違反	
2B000	依存権限記述子がまだ存在している	
2C000	キャラクタ・セット名が無効	
2D000	トランザクションの終了が無効	
2E000	接続名が無効	
33000	SQL 記述子名が無効	
34000	カーソル名が無効	
35000	条件番号が無効	
37000	動的 SQL 構文エラーまたはアクセス規則違反	
3C000	カーソル名があいまい	
3D000	カタログ名が無効	
3F000	スキーマ名が無効	
40000	トランザクションのロールバック	ORA-02091 ORA-02092
40001	シリアル化の障害	

表 9-2 SQLSTATE ステータス・コード

コード	条件	Oracle エラー
40002	整合性制約違反	
40003	文の完了が不明	
42000	構文エラーまたはアクセス規則違反	ORA-00022 ORA-00251 ORA-00900 ～ 00999 ORA-01031 ORA-01490 ～ 01493 ORA-01700 ～ 01799 ORA-01900 ～ 02099 ORA-02140 ～ 02289 ORA-02420 ～ 02424 ORA-02450 ～ 02499 ORA-03276 ～ 03299 ORA-04040 ～ 04059 ORA-04070 ～ 04099
44000	WITH_CHECK_OPTION 指定違反	ORA-01402
60000	システム・エラー	ORA-00370 ～ 00429 ORA-00600 ～ 00899 ORA-06430 ～ 06449 ORA-07200 ～ 07999 ORA-09700 ～ 09999
61000	マルチスレッド・サーバーおよび分離プロセス のエラー	ORA-00018 ～ 00035 ORA-00050 ～ 00068 ORA-02376 ～ 02399 ORA-04020 ～ 04039
62000	マルチスレッド・サーバーおよび分離プロセス のエラー	ORA-00100 ～ 00120 ORA-00440 ～ 00569

表 9-2 SQLSTATE ステータス・コード

コード	条件	Oracle エラー
63000	Oracle*XA および 2 タスク・インタフェースの エラー	ORA-00150 ～ 00159
		ORA-02700 ～ 02899
		ORA-03100 ～ 03199
		ORA-06200 ～ 06249
		SQL-02128
64000	制御ファイル、データベース・ファイルおよび REDO ファイルのエラー；アーカイブおよびメ ディア・リカバリのエラー	ORA-00200 ～ 00369
		ORA-01100 ～ 01250
65000	PL/SQL のエラー	ORA-06500 ～ 06599
66000	Net8 ドライバ・エラー	ORA-06000 ～ 06149
		ORA-06250 ～ 06429
		ORA-06600 ～ 06999
		ORA-12100 ～ 12299
		ORA-12500 ～ 12599
67000	ライセンス許可エラー	ORA-00430 ～ 00439
69000	SQL*Connect のエラー	ORA-00570 ～ 00599
		ORA-07000 ～ 07199
72000	SQL 実行フェーズのエラー	ORA-00001
		ORA-01000 ～ 01099
		ORA-01400 ～ 01489
		ORA-01495 ～ 01499
		ORA-01500 ～ 01699
		ORA-02400 ～ 02419
		ORA-02425 ～ 02449
		ORA-04060 ～ 04069
		ORA-08000 ～ 08190
		ORA-12000 ～ 12019
		ORA-12300 ～ 12499
		ORA-12700 ～ 21999

表 9-2 SQLSTATE ステータス・コード

コード	条件	Oracle エラー
82100	メモリー不足 (割り当てられない)	SQL-02100
82101	カーソル・キャッシュが矛盾 (UCE/CUC が不一致)	SQL-02101
82102	カーソル・キャッシュが矛盾 (UCE の CUC エントリがない)	SQL-02102
82103	カーソル・キャッシュが矛盾 (CUC 参照の範囲外)	SQL-02103
82104	カーソル・キャッシュが矛盾 (使用可能な CUC がない)	SQL-02104
82105	カーソル・キャッシュが矛盾 (キャッシュに CUC エントリがない)	SQL-02105
82106	カーソル・キャッシュが矛盾 (カーソル番号が無効)	SQL-02106
82107	ランタイム・ライブラリに対してプログラムが古すぎる ; 再ブリコンパイルが必要	SQL-02107
82108	ランタイム・ライブラリに無効な記述子が渡された	SQL-02108
82109	ホスト・キャッシュが矛盾 (SIT 参照が範囲外)	SQL-02109
82110	ホスト・キャッシュが矛盾 (SQL の型が無効)	SQL-02110
82111	ヒープ一貫性のエラー	SQL-02111
82113	コード生成の内部整合性の障害	SQL-02115
82114	リエントラント・コード・ジェネレータが無効なコンテキストを与えた	SQL-02116
82117	この接続での OPEN または PREPARE が無効	SQL-02122
82118	アプリケーション・コンテキストが見つからない	SQL-02123
82119	エラー・メッセージのテキストを取り出せない	SQL-02125
82120	プリコンパイラと SQLLIB のバージョンが不一致	SQL-02127
82121	NCHAR エラー ; フェッチされたバイト数が奇数	SQL-02129
82122	EXEC TOOLS インタフェースが使用できない	SQL-02130

表 9-2 SQLSTATE ステータス・コード

コード	条件	Oracle エラー
82123	ランタイム・コンテキストが使用中	SQL-02131
82124	ランタイム・コンテキストが割り当てられない	SQL-02132
82125	スレッドで使用するためのプロセスを初期化できない	SQL-02133
82126	ランタイム・コンテキストが無効	SQL-02134
HZ000	リモート・データベース・アクセス	

SQLSTATE の使用

次の規則は、オプションの設定を MODE=ANSI にしてプリコンパイルするときに、SQLSTATE を SQLCODE または SQLCA と併用する場合に適用されます。SQLSTATE は宣言文内で宣言する必要があります。宣言文内で宣言しなければ無視されます。

SQLSTATE を宣言する場合

- SQLCODE の宣言はオプションです。宣言文の内部で SQLCODE を宣言すると、SQL 処理を実行するたびに Oracle Server は SQLSTATE および SQLCODE にステータス・コードを戻します。ただし、宣言文の外部で SQLCODE を宣言すると、Oracle は SQLSTATE のみにステータス・コードを戻します。
- SQLCA の宣言はオプションです。SQLCA を宣言すると、Oracle は SQLSTATE および SQLCA にステータス・コードを戻します。このときコンパイルのエラーを防ぐために、SQLCODE を宣言しないでください。

SQLSTATE を宣言しない場合

- 宣言文の内部または外部で、SQLCODE を宣言する必要があります。SQL 処理を実行するたびに、Oracle Server は SQLCODE にステータス・コードを戻します。
- SQLCA の宣言はオプションです。SQLCA を宣言すると、Oracle は SQLCODE および SQLCA にステータス・コードを戻します。

独自のコードを作成して SQLSTATE を明示的にチェックするか、WHENEVER SQLERROR 宣言文を使用して SQLSTATE を暗黙的にチェックすることによって、最新の実行 SQL 文の結果が得られます。実行 SQL 文および PL/SQL 文の後にのみ SQLSTATE をチェックしてください。

SQLCODE の宣言

MODE=ANSI で、SQLSTATE 状態変数を宣言していない場合は、宣言文の内側または外側で SQLCODE という **long** 型の整数の変数を宣言する必要があります。次に例を示します。

```
/* 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 */
long SQLCODE;
```

MODE=ORACLE の場合は、SQLCODE を宣言しても無視されます。

複数の SQLCODE を宣言できます。ローカルな SQLCODE へのアクセスは、プログラム内の適用範囲によって制限されます。

SQL 処理を実行するたびに、Oracle は現在の適用範囲内にある SQLCODE にステータス・コードを戻します。したがって、プログラムは、SQLCODE を明示的にチェックするか、WHENEVER 宣言文を暗黙的に指定することによって、最新の SQL 処理の結果を知ることができます。

特定のコンパイル・ユニット内で SQLCA のかわりに SQLCODE を宣言すると、プリコンパイラはそのユニット用に内部 SQLCA を割り当てます。ホスト・プログラムは内部 SQLCA にはアクセスできません。SQLCA および SQLCODE の両方を宣言すると、Oracle は SQL 操作を実行するたびに同じステータス・コードを両方に戻します。

SQLCA を使用したエラー報告の主要コンポーネント

エラー報告は SQLCA 内の変数によって異なります。この項ではエラー報告の主要コンポーネントを扱います。また、この次の項では SQLCA を詳しく説明します。

ステータス・コード

すべての実行 SQL 文は、SQLCA 変数 *sqlcode* にステータス・コードを戻します。戻されたステータス・コードは WHENEVER 文によって暗黙的に、または独自のコードによって明示的にチェックできます。

0（ゼロ）のステータス・コードは、Oracle がエラーまたは例外を検出せずに文を実行したことを意味します。正のステータス・コードは、Oracle が例外を検出した上で文を実行したことを意味します。負のステータス・コードは、エラーが発生したために Oracle が SQL 文を実行しなかったことを意味します。

警告フラグ

警告フラグは SQLCA 変数の `sqlwarn[0]` から `sqlwarn[7]` に戻されます。これは暗黙的にも明示的にもチェックできます。警告フラグは、Oracle がエラーとみなさない実行時の条件をチェックするのに便利です。標識変数がなければ、Oracle はエラー・メッセージを発行します。

処理済の行数

最後に実行した SQL 文で処理された行数は、SQLCA 変数 `sqlca.sqlerrd[2]` に戻されます。これは明示的にチェックできます。

厳密にはこの変数はエラー報告用ではなく、誤りを防止するためのものです。たとえば、表から約 10 行を削除するとします。削除処理後、`sqlca.sqlerrd[2]` をチェックすると、75 行が削除されていました。このような場合、安全のため、削除処理をロールバックして WHERE 句の検索条件を確認できます。

解析エラー・オフセット

SQL 文は実行前に必ず解析されます。つまり、構文規則に従っているかどうかおよび有効なデータベース・オブジェクトを参照しているかどうかを検証されます。Oracle がエラーを検出すると、SQLCA 変数 `sqlca.sqlerrd[4]` にオフセットが格納されます。これは明示的にチェックできます。解析エラーの始まりを示す SQL 文中の文字間調整がオフセットとして指定されます。通常の C 文字列と同様に、先頭の文字間調整はゼロです。たとえば、オフセットが 9 のとき、解析エラーは 10 番目の文字から始まります。

デフォルトでは、静的 SQL 文はプリコンパイル時に構文エラーをチェックします。したがって、`sqlca.sqlerrd[4]` は、プログラムが実行時に受け入れるまたは作成する動的 SQL 文のデバッグに最も適しています。

解析エラーは、キーワードの脱落、キーワードの位置指定の誤り、キーワードのスペルミス、無効なオプション、存在しない表などが原因で発生します。次に例を示します。

```
"UPDATE emp SET jib = :job_title WHERE empno = :emp_number"
```

この動的 SQL 文は解析エラーになります。

ORA-00904: 列名が無効です。

原因は列名 JOB のスペルミスです。誤った列名 JIB が 16 番目の文字で始まっているため、`sqlca.sqlerrd[4]` の値が 15 になっています。

SQL 文に解析エラーがなければ、Oracle は `sqlca.sqlerrd[4]` に 0（ゼロ）を設定します。解析エラーが先頭の文字（文字間調整はゼロ）で始まっているときも、Oracle は、`sqlca.sqlerrd[4]` に 0（ゼロ）を設定します。したがって `sqlca.sqlcode` が負のときのみ、`sqlca.sqlerrd[4]` をチェックします。負はエラーが発生したことを意味しています。

エラー・メッセージ・テキスト

Oracle エラーのエラー・コードおよびメッセージは SQLCA 変数 SQLERRMC に格納されています。テキストの先頭から最大 70 文字 (バイト) 分が格納されています。70 文字を超えるメッセージのテキスト全体を格納するには、*sqlglm()* 関数を使用します。詳細は 9-23 ページの「[エラー・メッセージの全文の取得](#)」を参照してください。

SQL コミュニケーション領域 (SQLCA) の使用

SQLCA はデータ構造体です。そのコンポーネントには、エラー、警告および SQL 文を実行するたびに更新される状態情報が格納されます。つまり、SQLCA は常に最新の SQL 処理の結果を反映します。この結果を判定するために、SQLCA 内の変数をチェックできます。

プログラムでは複数の SQLCA を使用できます。たとえば、1 つのグローバル SQLCA と複数のローカル SQLCA を指定できます。ローカルな SQLCA へのアクセスは、プログラム内の適用範囲によって制限されます。Oracle は適用範囲内にある SQLCA のみに情報を戻します。

注意： ローカルとリモートのデータベースに同時にアクセスするためにアプリケーションが Net8 を使用しているとき、すべてのデータベースはひとつの SQLCA に書き込みます。つまり、データベースごとに異なる SQLCA があるわけではありません。詳細は、3-5 ページの「[アドバンスト・コネクション・オプション](#)」の項を参照してください。

SQLCA の宣言

MODE=ORACLE のときは、SQLCA の宣言が必要です。SQLCA を宣言するには、次に示すように INCLUDE または **#include** 文を使用して SQLCA をプログラム内にコピーします。

```
EXEC SQL INCLUDE SQLCA;
```

または

```
#include <sqlca.h>
```

宣言文を使用するときは、SQLCA は宣言文の外側で宣言する必要があります。SQLCA を宣言しないとコンパイル時にエラーが発生します。

プログラムをプリコンパイルすると、INCLUDE SQLCA 文は複数の変数宣言に置換されます。この変数宣言によって、Oracle はそのプログラムと通信できます。

MODE=ANSI のときは、SQLCA の宣言はオプションです。ただしこのとき、SQLCODE または SQLSTATE 状態変数は宣言する必要があります。SQLCODE (必ず大文字) の型は **long** です。特定のコンパイル・ユニットで SQLCA のかわりに SQLCODE または SQLSTATE を宣言した場合には、プリコンパイラはその単位用に内部 SQLCA を割り当てます。Pro*C/C++ プログラムは、内部 SQLCA にはアクセスできません。SQLCA および SQLCODE の両方を宣言すると、Oracle は SQL 操作を実行するたびに同じステータス・コードを両方に戻します。

注意：SQLCA の宣言は MODE=ANSI のときにはオプションですが、WHENEVER SQLWARNING 宣言文は SQLCA がなければ使用できません。したがって、WHENEVER SQLWARNING 宣言文を使用する場合は、SQLCA を宣言する必要があります。

注意：このガイドでは SQLCODE 状態変数のことを SQLCODE と言います。また SQLCA 構造体のコンポーネントのことを明示する場合には *sqlca.sqlcode* と言います。

SQLCA に含まれているもの

SQLCA 内には SQL 文の実行結果に関する次の情報が格納されます。

- Oracle エラー・コード
- 警告フラグ
- イベント情報
- 処理済の行数
- 診断情報

sqlca.h ヘッダー・ファイルは次のとおりです。

```
/*
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 programming practice not to have uninitialized variables. However, some C compilers/OS's 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/OS 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.

```

*/
#endif SQLCA
#define SQLCA 1
struct sqlca
{
    /* ub1 */ char    sqlcaid[8];
    /* b4  */ long    sqlabc;
    /* b4  */ long    sqlcode;
    struct
    {
        /* ub2 */ unsigned short sqlerrml;
        /* ub1 */ char          sqlerrmc[70];
    } sqlerrm;
    /* ub1 */ char    sqlerrp[8];
    /* b4  */ long    sqlerrd[6];
    /* ub1 */ char    sqlwarn[8];
    /* ub1 */ char    sqlext[8];
};
#endif SQLCA_NONE
#ifdef SQLCA_STORAGE_CLASS
SQLCA_STORAGE_CLASS struct sqlca sqlca
#else
    struct sqlca sqlca
#endif
#ifdef SQLCA_INIT
= {
    {'S', 'Q', 'L', 'C', 'A', ' ', ' ', ' ', ' '},
    sizeof(struct sqlca),
    0,
    { 0, {0}},
    {'N', 'O', 'T', ' ', 'S', 'E', 'T', ' ', ' '},
    {0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0}
}

```

```
    }  
#endif  
;  
#endif  
#endif
```

SQLCA の構造

この項では SQLCA の構造体、そのコンポーネントおよびコンポーネントに格納できる値について説明します。

sqlcaid

この文字列コンポーネントは「SQLCA」に初期化されて、SQL 通信領域を示します。

sqlcabc

この整数コンポーネントには SQLCA 構造体の長さがバイト単位で格納されます。

sqlcode

この整数コンポーネントには最後に実行された SQL 文のステータス・コードが格納されます。SQL 操作の結果を示すステータス・コードは次の数字のどれかになります。

0	エラーまたは例外の検出なしで文が実行されたことを示します。
>0	文は実行されたが、例外が検出されたことを示します。WHERE 句の検索条件を満たす行が見つからないとき、または SELECT INTO や FETCH で行が戻されなかったときにこの状態が発生します。

MODE=ANSI のときは、どの行も INSERT できなければ +100 が *sqlcode* に戻ります。副問合せで処理に行が戻されなかったときにこの状態が発生します。

<0	データベース、システム、ネットワークまたはアプリケーションのエラーが原因で、文が実行されなかったことを示します。このようなエラーは致命的です。こうしたエラーが発生すると、ほとんどの場合はカレント・トランザクションがロールバックされます。
----	--

負のリターン・コードは『Oracle8i エラー・メッセージ』に一覧を示したエラー・コードに対応します。

sqlerrm

この埋込み構造体には次の 2 つのコンポーネントがあります。

sqlerrml	この整数コンポーネントには、sqlerrmc 内に保存されているメッセージ・テキストの長さが格納されます。
sqlerrmc	この文字列コンポーネントには、sqlcode 内に保存されているエラー・コードに対応したメッセージ・テキストが格納されます。文字列は NULL では終了しません。長さを調べるには sqlerrml コンポーネントを使用します。

このコンポーネントには最大 70 文字 (バイト) まで格納できます。70 文字を超えるメッセージ・テキスト全体を保存するには、この後で説明する *sqlglm* 関数を使用します。

sqlerrmc を参照する前に、*sqlcode* が負であることを確認する必要があります。*sqlcode* が 0 (ゼロ) のときに *sqlerrmc* を参照すると、前の SQL 文と対応するメッセージ・テキストが得られます。

sqlerrp

この文字列コンポーネントは将来使用するために確保されています。

sqlerrd

この 2 進整数の配列には 6 つの要素があります。*sqlerrd* 内のコンポーネントの説明を次に示します。

sqlerrd[0]	このコンポーネントは将来使用するために確保されています。
sqlerrd[1]	このコンポーネントは将来使用するために確保されています。
sqlerrd[2]	このコンポーネントには、最後に実行した SQL 文によって処理された行数が格納されます。ただし、SQL 文が失敗すると、1 つの例外を除き <i>sqlca.sqlerrd[2]</i> の値は未定義となります。配列処理中にエラーが発生すると、そのエラーの発生した行で処理は停止します。そのため、 <i>sqlca.sqlerrd[2]</i> は正常に処理された行数を示します。

処理済の行数は OPEN 文の後に 0 (ゼロ) に設定され、FETCH 文の後に増分されます。処理済の行数は、EXECUTE 文、INSERT 文、UPDATE 文、DELETE 文および SELECT INTO 文について、正常に処理された行数を反映します。この数字には UPDATE または DELETE CASCADE によって処理された行は含まれません。たとえば WHERE 句の条件を満たす 20

行が削除された後で、列制約条件に違反する 5 行が削除されたときの処理済の行数は、25 ではなく 20 となります。

sqlerrd[3] このコンポーネントは将来使用するために確保されています。

sqlerrd[4] このコンポーネントは、最後に実行された SQL 文中で解析エラーの始まりを示す文字間調整を指定するオフセットを保持します。先頭の文字間調整は 0 (ゼロ) です。

sqlerrd[5] このコンポーネントは将来使用するために確保されています。

sqlwarn

この 1 文字の配列には 8 つの要素があります。これらの要素は警告フラグとして使用されます。Oracle はそれに文字値「W」(警告) を割り当てることで、フラグを設定します。

フラグは例外条件の発生を警告します。たとえば、切り捨てられた列値を Oracle が出力ホスト変数に代入すると、警告フラグが設定されます。

sqlwarn のコンポーネントの説明を次に示します。

sqlwarn[0] このフラグは別の警告フラグが設定されていることを示します。

sqlwarn[1] このフラグは、切り捨てられた列値が出力ホスト変数に代入されたときに設定されます。これは文字データにのみ適用されます。つまり Oracle は、警告を設定することも負の sqlcode を返すこともなく特定の数値データを切り捨てます。

列値が切り捨てられたかどうか、またどのくらい切り捨てられたかを調べるには、出力ホスト変数に対応する標識変数をチェックします。標識変数によって戻された値が正の整数のときは、その値は列値の元の長さを示します。その値に応じてホスト変数の長さを増やすことができます。

sqlwarn[2]	AVG() や SUM() などの SQL グループ関数の結果に NULL 列が使用されない場合、このフラグが設定されます。
sqlwarn[3]	問合せの選択リスト内の列の数が SELECT 文または FETCH 文の INTO 句内のホスト変数の数と一致しないときに、このフラグが設定されます。戻される項目の数は両者のうち少ない方の数となります。
sqlwarn[4]	このフラグは現在使用されていません。
sqlwarn[5]	このフラグが設定されるのは、EXEC SQL CREATE {PROCEDURE FUNCTION PACKAGE PACKAGE BODY} 文が PL/SQL コンパイル・エラーのために失敗したときです
sqlwarn[6]	このフラグは現在使用されていません。
sqlwarn[7]	このフラグは現在使用されていません。

sqltext

この文字列コンポーネントは将来使用するために確保されています。

PL/SQL の考慮事項

プリコンパイラ・アプリケーションで埋込み PL/SQL ブロックを実行するときに、SQLCA のすべてのコンポーネントが設定されるわけではありません。たとえばブロックが複数の行をフェッチするときは、処理済の行数 (*sqlerrd[2]*) には 1 しか設定されません。PL/SQL ブロックの実行後は、SQLCA の *sqlcode* および *sqlerrm* コンポーネントのみを使用してください。

エラー・メッセージの全文の取得

SQLCA は最高で 70 文字（バイト）までの長さのエラー・メッセージを格納できます。それ以上長い（またはネストされた）エラー・メッセージのテキスト全体を取得するには、*sqlglm()* 関数が必要です。構文は次のとおりです。

```
void sqlglm(char    *message_buffer,
              size_t *buffer_size,
              size_t *message_length);
```

パラメータは次のとおりです。

message_buffer	エラー・メッセージを格納するためのテキスト・バッファです（バッファの残りの部分には空白が埋め込まれます）。
----------------	---

<code>buffer_size</code>	バッファの最大サイズをバイト数で示したスカラー変数です。
<code>message_length</code>	切り捨てられていない場合は、Oracle によって格納されたエラー・メッセージの実際の長さを示すスカラー変数です。

注意： `sqlglm()` 関数の最後のふたつの引数の型は一般的な `size_t` ポインタとして示してあります。しかし、プラットフォームによって型が異なることがあります。たとえば、多くの UNIX ワークステーション・ポートでは、`unsigned int *` になります。

これらのパラメータのデータ型を判別するには、システムの標準インクルード・ディレクトリにある `sqlcpr.h` ファイルをチェックしてください。

Oracle エラー・メッセージの最大長は 512 文字です。これにはエラー・コード、ネストされたメッセージ、表名や列名などといったメッセージ挿入情報が含まれます。`sqlglm` によって戻されるエラー・メッセージの最大長は、`buffer_size` に指定した値によって決まります。

次の例では、`sqlglm` をコールして、200 文字以内の長さのエラー・メッセージを取得します。

```
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 */
    printf("%.s\n\n", msg_len, msg);
    exit(1);
}
```

`sqlglm` は、SQL エラーが発生したときにのみコールできます。`sqlglm` をコールする前に、`SQLCODE`（または `sqlca.sqlcode`）の値がゼロでないことを必ず確認してください。`SQLCODE` が 0（ゼロ）のときに `sqlglm` をコールすると、前の SQL 文に対応したメッセージ・テキストが得られます。

WHENEVER 宣言文の使用

デフォルトでは、プリコンパイルされたプログラムは Oracle エラーおよび警告条件を無視し、可能であれば処理を続行します。自動条件チェックおよびエラー処理を実行するには `WHENEVER` 宣言文が必要です。

WHENEVER 宣言文によって、Oracle がエラーや警告条件、「見つからない」条件を検出したときにとる行動を指定することができます。これらのアクションには、次の文の継続実行、ルーチンのコール、ラベル付き文への分岐、停止などがあります。

WHENEVER 宣言文の構文は次のとおりです。

```
EXEC SQL WHENEVER <condition> <action>;
```

条件

次の条件の有無を調べるために Oracle で SQLCA を自動的にチェックできます。

SQLWARNING

sqlwarn[0] が設定されるのは、Oracle が警告（警告フラグ *sqlwarn[1]* から *sqlwarn[7]* までのどれか 1 つも設定されます）を戻したか、SQLCODE が +1403 以外の正の値になっているときです。たとえば *sqlwarn[0]* は、Oracle が切り捨てた列値を出力ホスト変数に割り当てると設定されます。

MODE=ANSI のときは、SQLCA の宣言はオプションです。ただし WHENEVER SQLWARNING を使用するには、SQLCA を宣言する必要があります。

SQLERROR

Oracle がエラーを戻したため、SQLCODE に負の値が設定されています。

NOT FOUND

Oracle が WHERE 句の検索条件を満たす行を検出できなかったか、SELECT INTO または FETCH が行を戻さなかったため、SQLCODE に +1403 が設定されています（MODE=ANSI のときは +100）。

MODE=ANSI のときは、どの行も INSERT できなければ +100 が SQLCODE に戻ります。

アクション

前述の条件のいずれかが検出されたときは、プログラムで次のアクションを実行できます。

CONTINUE

可能であれば、プログラムは次の文からの実行を継続します。これはデフォルトの動作で、WHENEVER 宣言文を使用しない場合と同じです。このアクションを使用すると条件チェックを終了できます。

DO

制御をプログラムのエラー処理関数に移します。ルーチンの最後に達すると、制御は失敗した SQL 文に続く文に移ります。

関数への出入口について、通常の規則が適用されます。EXEC SQL WHENEVER ... DO ... 宣言文に呼び出されるエラー・ハンドラにパラメータを渡し、関数によって値を戻すことができます。

DO BREAK

実際の「break」文はプログラム中に置かれます。このアクションはループ内に使用します。WHENEVER 条件が成立すると、プログラムは入っていたループを終了します。

DO CONTINUE

実際の「continue」文はプログラム中に置かれます。このアクションはループ内に使用します。WHENEVER 条件が成立すると、プログラムは入っていたループの次の繰返しに移ります。

GOTO label_name

プログラムはラベル付き文に分岐します。ラベル名の長さに制限はありませんが、有効なのは最初の 31 文字のみです。異なる最大長を必要とする C コンパイラもあります。使用している C コンパイラのユーザーズ・ガイドを参照してください。

STOP

プログラムは実行を停止し、COMMIT されていない作業がロールバックされます。

STOP は、実際には条件が発生するたびに *exit()* コールを生成するのみです。注意が必要です。STOP アクションは Oracle からの切断前に何もメッセージを表示しません。

例

たとえばプログラムで、

- 「データが見つかりません」条件が発生したときには、*close_cursor* へ進みます。
- 警告が発生したときは、次の文を続行します。
- エラーが発生したときは、*error_handler* に進みます。

最初の実行 SQL 文の前に次の WHENEVER 宣言文を指定するのみで済みます。

```
EXEC SQL WHENEVER NOT FOUND GOTO close_cursor;  
EXEC SQL WHENEVER SQLWARNING CONTINUE;  
EXEC SQL WHENEVER SQLERROR GOTO error_handler;
```

次の例では、WHENEVER...DO 宣言文を使用して特定のエラーを処理します。


```

...
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
    {
        ...
    }
    ...
}

```

プロシージャで SQLCA の変数をチェックしてアクションの過程を決定する方法に注目してください。

DO BREAK と DO CONTINUE の利用

この例では、コミッションを受け取っている従業員の分のみ、従業員の名前、給料、コミッションを表示する方法を示しています。

```
#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;
        /* An ORA-1405 would cause the 'continue' to occur. So only employees with */
        /* 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);
}
```

WHENEVER 文の適用範囲

WHENEVER 文は宣言文のため、その適用範囲は論理的なものではなく位置的なものになります。つまり、WHENEVER 文はプログラム論理の流れではなく、ソース・ファイル内で物理的に WHENEVER 文に続く実行 SQL 文をすべてテストします。したがって WHENEVER 宣言文は、テストする最初の実行 SQL 文の前に指定する必要があります。

ある WHENEVER 宣言文は、同じ条件をチェックする別の WHENEVER 宣言文に置き換えられるまでの間は有効です。

次の例では、最初の WHENEVER SQLERROR 宣言文は 2 番目のものに置き換えられます。したがって、この宣言文の制御は CONNECT 文のみに適用されます。2 番目の WHENEVER SQLERROR 宣言文は、*step1* から *step3* への制御の流れに関係なく、UPDATE 文および DROP 文の両方に適用されます。

```
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;
    ...
```

WHENEVER のガイドライン

この項では、一般的な問題点を回避するためのガイドラインを示します。

文の位置

通常、WHENEVER 宣言文はプログラム内の最初の実行 SQL 文の前に指定します。この位置に指定した WHENEVER 宣言文はファイルの最後まで有効になるため、発生するすべてのエラーを確実にトラップできます。

データの終わり条件の処理

カーソルを使用して行をフェッチするときは、プログラムでデータの終わり条件を対処できるようにする必要があります。FETCH がデータを戻さないときは、プログラムは次のように FETCH ループを終了します。

```
EXEC SQL WHENEVER NOT FOUND DO break;
for (;;)
{
    EXEC SQL FETCH...
```

```
}  
EXEC SQL CLOSE my_cursor;  
...
```

行が挿入されていない場合は、INSERT は「NOT FOUND」を戻します。この条件を取り上げない場合は、INSERT の前に EXEC SQL WHENEVER NOT FOUND CONTINUE を使用します。

```
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;  
...
```

無限ループの回避

WHENEVER SQLERROR GOT 宣言文が、実行 SQL 文を含むエラー処理ルーチンに分岐しているときに、その SQL 文にエラーが発生するとプログラムが無限ループに陥る恐れがあります。無限ループを回避するには、次に示すように SQL 文の前で WHENEVER SQLERROR CONTINUE を記述します。

```
EXEC SQL WHENEVER SQLERROR GOTO sql_error;  
...  
sql_error:  
    EXEC SQL WHENEVER SQLERROR CONTINUE;  
    EXEC SQL ROLLBACK WORK RELEASE;  
...
```

WHENEVER SQLERROR CONTINUE 文を指定しなければ、ROLLBACK エラーが発生したときにこのルーチンが再び実行されるため、結果として無限ループに陥ります。

WHENEVER 句は、注意して使用しないと問題が発生することがあります。たとえば、検索条件を満たす行がないために DELETE 文が NOT FOUND を設定すると、次のコードは無限ループに陥ります。

```
/* 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;
    ...
```

次の例では、GOTO のターゲットを再設定することによって、NOT FOUND 条件を適切に処理します。

```
/* 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:
    ...
```

アドレス指定可能度の維持

WHENEVER GOTO 宣言文によって制御されるすべての SQL 文が、必ず GOTO ラベルに分岐するようにしてください。次のコードは *func1* 内の *labelA* が *func2* 内の INSERT 文の範囲内がないため、コンパイル時エラーが発生します。

```
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);
    ...
}
```

WHENEVER GOTO 宣言文の分岐先のラベルは、この文と同じプリコンパイル・ファイル内にする必要があります。

エラー後の戻り

エラーの処理後にプログラムに戻る必要がある場合は、`DO routine_call` アクションを使用します。または次の例に示すように、`sqlcode` の値をテストしてもかまいません。

```
...
EXEC SQL UPDATE emp SET sal = sal * 1.10;
if (sqlca.sqlcode < 0)
{ /* handle error */

EXEC SQL DROP INDEX emp_index;
```

アクティブな `WHENEVER GOTO` 宣言文または `WHENEVER STOP` 宣言文がないことを確認してください。

SQL 文のテキスト取得

多くのプリコンパイラ・アプリケーションでは、処理中の文のテキスト、その長さ、指定されている SQL コマンド (`INSERT` や `SELECT` など) を把握していると役に立ちます。これは、動的 SQL を使用するアプリケーションでは特に重要です。

`SQLStmtGetText()` 関数 (古くは `sqlgls()` 関数) —`SQLLIB` ランタイム・ライブラリの一部 — は次の情報を戻します。

- 最後に解析された SQL 文のテキスト
- 文の有効な長さ
- 文で使用されている SQL コマンドの機能コード

`SQLStmtGetText()` はスレッド・セーフです。静的 SQL 文を発行した後に `SQLStmtGetText()` 関数をコールできます。動的 SQL 方法 1 のときは、SQL 文が実行された後に `SQLStmtGetText()` をコールします。動的 SQL 方法 2、3 および 4 のときは、文が `PREPARE` されるとすぐに `SQLStmtGetText()` をコールできます。

`SQLLIB` 関数の新しい名前は 5-49 ページの「[SQLLIB パブリック関数の新しい名前](#)」を参照してください。

`SQLStmtGetText()` のプロトタイプは、次のとおりです。

```
void SQLStmtGetText (dvoid *context, char *sqlstm, size_t *stmrlen, size_t *sqlfc);
```

コンテキスト・パラメータはランタイム・コンテキストです。コンテキストの定義と使用方法は 4-34 ページの「[CONTEXT 変数](#)」を参照してください。

`sqlstm` パラメータは文字バッファです。このバッファには、SQL 文の戻されたテキストが格納されます。プログラムでは静的にバッファを宣言するか、動的にバッファのメモリーを割り当てる必要があります。

stmrlen パラメータは *size_t* 変数です。SQLStmtGetText() をコールする前に、このパラメータに *sqlstm* バッファの実際のサイズをバイト単位で設定してください。SQLStmtGetText() が戻ると、*sqlstm* バッファには SQL 文テキストが入り、その後はバッファの長さまで空白が埋め込まれます。*stmrlen* パラメータは、戻された文テキストの実際のバイト数を返します。これは埋め込まれた空白のバイト数を含みません。*stmrlen* の最大値はポート固有で、通常は最大整数サイズになります。

sqlfc パラメータは、文の SQL コマンドの SQL 関数コードを返す *size_t* 変数です。表 9-3 にコマンドの SQL 関数コードを示します。

表 9-3 SQL コード

コード	SQL 関数	コード	SQL 関数	コード	SQL 関数
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
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	(使用されていない)	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	(使用されてい ない)	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

表 9-3 SQL コード

コード	SQL 関数	コード	SQL 関数	コード	SQL 関数
19	CREATE USER	44	CREATE DATABASE	69	(使用されていない)
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

エラーが発生すると、長さパラメータ (*stmrlen*) はゼロを戻します。発生する可能性のあるエラー条件は、次のとおりです。

- SQL 文が解析されていません。
- 無効なパラメータ (たとえば、負の長さのパラメータ) を渡しました。
- SQLLIB で内部の例外が生じました。

制限

SQLStmtGetText() は、次のコマンドを含む文のテキストは戻しません。

- CONNECT
- COMMIT
- ROLLBACK
- FETCH

これらのコマンドの SQL 関数コードはありません。

サンプル・プログラム

第 4 章「データ型とホスト変数」に示すサンプル・プログラム *sqlvcp.pc* は *sqlgls()* 関数の利用方法を実演しています。このプログラムは、*demo* ディレクトリにあり、オンラインでも利用できます。

ORACLE コミュニケーション領域（ORACA）の使用

SQLCA が標準的な SQL 通信を処理するのに対し、ORACA は Oracle 通信を処理します。SQLCA が提供するランタイム・エラーおよび状態の変化に関する情報よりもさらに詳しい情報が必要なときは ORACA を使用します。ORACA には豊富な診断ツールが用意されています。ただし ORACA の使用はランタイム・オーバーヘッドを増加させるため、あくまでもオプションです。

問題の診断に役立つうえに、ORACA はプログラムの Oracl リソース、たとえば SQL 文エグゼキュータやカーソル・キャッシュなどの利用を監視することを可能にします。

プログラムでは複数の ORACA を使用できます。たとえば、1 つのグローバル ORACA と複数のローカル ORACA を指定できます。ローカルな ORACA へのアクセスは、プログラム内の適用範囲によって制限されます。Oracle は適用範囲内の ORACA のみに情報を戻します。

ORACA の宣言

ORACA を宣言するには、次に示すように、INCLUDE 文または **#include** プリプロセッサ宣言文を使用して ORACA を自分のプログラムにコピーします。

```
EXEC SQL INCLUDE ORACA;
```

または

```
#include <oraca.h>
```

ORACA が必ず **extern** 記憶クラスである場合、プログラムに次のように ORACA_STORAGE_CLASS を定義します。

```
#define ORACA_STORAGE_CLASS extern
```

プログラムで宣言文を使用するときは、ORACA を宣言文の外側で定義する必要があります。

ORACA を使用可能にする

ORACA を使用可能にするには、コマンドラインに次のように ORACA オプションを指定する必要があります。

```
ORACA=YES
```

またはインラインで次のように指定します。

```
EXEC ORACLE OPTION (ORACA=YES);
```

その後に ORACA 内のフラグを設定することによって、適切なランタイム・オプションを選択する必要があります。

ORACA に含まれているもの

ORACA 内には次に示すように、オプションの設定、システムの統計情報および高度な診断情報が保存されています。

- SQL 文のテキスト (そのテキストを保存するときに指定できます)
- エラーが発生したファイルの名称 (サブルーチンの使用時に便利です)
- ファイル内のエラーの位置
- カーソル・キャッシュのエラーおよび統計情報

oraca.h の一部を次に示します。

```
/*
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
    uninitialized variables. However, some C compilers/OS's 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/OS 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    1

struct    oraca
{
    char oracaid[8];    /* Reserved          */
    long oracabc;       /* Reserved          */

    /*    Flags which are setable by User.    */

    long oracchf;       /* <> 0 if "check cur cache consistncy" */
    long oradbgf;       /* <> 0 if "do DEBUG mode checking"    */
    long orahchf;       /* <> 0 if "do Heap consistency check" */
    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 */
    long orahoc;        /* highest max open OraCurs requested */
    long oramoc;        /* max open OraCursors required */
    long oracoc;        /* current OraCursors open */
    long oranor;        /* nr of OraCursor re-assignments */
    long oranpr;        /* nr of parses */
    long oranex;        /* nr of executes */
    };

#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}},
        0,
        0,0,0,0,0,0
    }
#endif
;

#endif

#endif

/* end oraca.h */

```

ランタイム・オプションの選択

ORACA にはいくつかのオプション・フラグがあります。これらのフラグにゼロ以外の値を設定することにより、次のことが可能になります。

- SQL 文のテキストの保存
- DEBUG 処理の有効化
- カーソル・キャッシュの一貫性チェック（「カーソル・キャッシュ」とは、カーソル管理に使用されるメモリーで継続的に更新される領域を指します）
- ヒープの一貫性チェック（ヒープとは動的変数のために予約されるメモリー領域を指します）
- カーソルの統計情報の収集

次の説明はオプションを選択するときの参考になります。

ORACA の構造

この項では、ORACA の構造体とそのコンポーネントおよび格納できる値について説明します。

oracaid

この文字列コンポーネントは Oracle 通信領域を示すために「ORACA」に初期化されます。

oracabc

この整数コンポーネントには ORACA データ構造体の長さがバイト単位で格納されています。

oracchf

マスター DEBUG フラグ (*oradbgf*) が設定されていると、このフラグによってカーソル・キャッシュの統計情報の収集と、各カーソル操作前のカーソル・キャッシュの一貫性チェックができます。

Oracle ランタイム・ライブラリは一貫性チェックを行い、エラー・メッセージを発行することがあります。これはマニュアル『Oracle8i エラー・メッセージ』に一覧にしてあります。それらのメッセージは、Oracle エラー・メッセージと同様に SQLCA に戻ります。

このフラグは次のいずれかを設定します。

- キャッシュ一貫性チェックを使用禁止にします（デフォルト）。
- キャッシュ一貫性チェックを使用可能にします。

oradbgf

このマスター・フラグを使用すると、DEBUG オプションをすべて選択できます。このフラグには次のいずれかを設定します。

すべての DEBUG 処理を使用禁止にします (デフォルト)。

すべての DEBUG 処理を使用可能にします。

orahchf

マスター DEBUG フラグ (*oradbgf*) が設定されていると、Oracle ランタイム・ライブラリでは、プリコンパイラが動的にメモリーを割り当てたりメモリーを解放するたびにヒープの一貫性がチェックされます。これはメモリー障害を起こすプログラムのバグを検出するのに役立ちます。

このフラグは CONNECT コマンドを発行する前に設定する必要があります。また、このフラグは一度設定すると解除できなくなります。つまり設定後にこのフラグの変更要求があっても無視されます。このフラグには次のいずれかを設定します。

- ヒープ一貫性チェックを使用禁止にします (デフォルト)。
- ヒープ一貫性チェックを使用可能にします。

orastxtf

このフラグを使用すると、カレント SQL 文のテキストを保存するタイミングを指定できます。このフラグには次のいずれかを設定します。

- SQL 文のテキストを保存しません (デフォルト)。
- SQLERROR の SQL 文のテキストのみを保存します。
- SQLERROR または SQLWARNING の SQL 文のテキストのみを保存します。
- 常に SQL 文のテキストを保存します。

SQL 文のテキストは、*orastxt* という名前の ORACA 埋込み構造体に保存されます。

診断情報

ORACA は高度な診断情報を提供します。次の変数によってエラーの位置をすばやく特定できます。

orastxt

この埋込み構造体は、問題のある SQL 文を見つけるために使用します。これによって、Oracle が最後に解析した SQL 文のテキストを保存できます。この構造体には次の 2 つのコンポーネントが収められています。

orastxtl この整数コンポーネントにはカレント SQL 文の長さが格納されます。

orastxtc この文字列コンポーネントにはカレント SQL 文のテキストが格納されます。先頭から最長 70 文字分のテキストが保存されます。文字列は NULL では終了しません。文字列を印刷するときは、**orastxtl** 長さコンポーネントを使用します。

CONNECT、FETCH、COMMIT などの文は、プリコンパイラによって解析されますが、ORACA には保存されません。

orasfnm

この埋込み構造体は、カレント SQL 文が格納されているファイルを識別します。このため、1 つのアプリケーション用に複数のファイルをプリコンパイルするときにエラーを検出できます。この構造体には次の 2 つのコンポーネントが収められています。

orasfnml この整数コンポーネントには、**orasfnmc** に保存されているファイル名の長さが格納されます。

orasfnmc この文字列コンポーネントにはファイル名が格納されます。先頭から最長 70 文字分が保存されます。

oraslnr

この整数コンポーネントはカレント SQL 文がある行またはその付近の行を識別します。

カーソル・キャッシュ統計情報

マスター DEBUG フラグ (*oradb_{gf}*) およびカーソル・キャッシュ・フラグ (*oracch_f*) が設定されているときは、次の変数を使用するとカーソル・キャッシュ統計情報を収集できます。これらの変数は、プログラムが COMMIT コマンドまたは ROLLBACK コマンドを発行するたびに自動的に設定されます。

内部的には、CONNECT されているデータベース別にこの変数のセットがあります。ORACA 内のカレント設定値は、最後に COMMIT または ROLLBACK が行われたデータベースに関係します。

orahoc

この整数コンポーネントは、プログラムの実行中に MAXOPENCURSORS に設定された最大値を記録します。

oramoc

この整数コンポーネントは、プログラムの要求によってオープンされた Oracle カーソルの最大数を記録します。MAXOPENCURSORS に設定されている数が小さすぎると、この数は *orahoc* よりも大きくなる場合があります。このとき、プリコンパイラによってカーソル・キャッシュが拡張されます。

oracoc

この整数コンポーネントは、プログラムの要求によってオープンされている Oracle カーソルのカレント数を記録します。

oronor

この整数コンポーネントは、プログラムの要求によって再度割り当てられたカーソル・キャッシュの数を記録します。この数字はカーソル・キャッシュの「スラッシング」の程度を示します。可能なかぎり低く保ってください。

oranpr

この整数コンポーネントは、プログラムの要求によって解析された SQL 文の数を記録します。

oranex

この整数コンポーネントは、プログラムの要求によって実行された SQL 文の数を記録します。この数値の *oranpr* に対する割合は、できる限り高く保ってください。つまり、不要な再解析は回避する必要があります。ヘルプについては [付録 C「パフォーマンスの最適化」](#) を参照してください。

ORACA の使用例

次のプログラムは部門番号の入力を要求し、その部内の各従業員の名前および給与を 2 つの表のどちらかに挿入してから、ORACA からの診断情報を表示します。このプログラムは *oraca.pc* として *demo* ディレクトリにあり、オンラインで利用できます。

```
/* 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.oradbfg = 1;          /* enable debug operations */
    oraca.oracchf = 1;         /* gather cursor cache statistics */
    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
("Cursor Cache Statistics\n-----\n");
    printf
("Maximum value of MAXOPENCURSORS:      %d\n", oraca.oraohoc);
    printf
("Maximum open cursors required:        %d\n", oraca.oramoc);
    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);
}
```

プリコンパイラのオプション

この章では、Pro*C/C++ プリコンパイラの実行方法とプリコンパイラ・オプションの拡張セットについて詳しく説明します。

この章では、次の事項について説明します。

- プリコンパイラのコマンド
- プリコンパイラのオプション
- 早見表
- オプションの入力
- プリコンパイラ・オプションの使用

プリコンパイラのコマンド

プリコンパイラの位置はシステムによって異なります。システム管理者またはデータベース管理者は、通常は論理装置名またはその別名を指定するか、その他のシステム固有の手段を使用することによって、Pro*C/C++ 実行ファイルをアクセス可能にします。

Pro*C/C++ プリコンパイラを実行するには、次のコマンドを入力します。

```
proc option=value...
```

注意：オプション値は必ずオプション名に続く等号（前後のスペースなし）の後に指定します。

たとえば次のコマンドを入力すると、

```
proc INAME=test_proc
```

カレント・ディレクトリのファイル *test_proc.pc* がプリコンパイルされます。これはプリコンパイラがファイル名の拡張子を *.pc* とみなすためです。INAME= 引数は、プリコンパイルされるソース・ファイルを指定します。INAME オプションはコマンドラインの最初のオプションでなくてもかまいませんが、最初にくる場合はオプション指定を省略できます。したがって、次のコマンドは

```
proc myfile
```

次のオプションに相当します。

```
proc INAME=myfile
```

注意：特定の OS オブジェクトの名前がついていないオプション名とオプション値では大文字小文字は区別されません。このマニュアル中の例では、オプション名は大文字で記述し、オプション値は通常は小文字で記述しています。大文字小文字の区別に関して、Pro*C/C++ プリコンパイラ実行ファイル自体も含めてファイル名を入力するときには、オペレーティング・システムの表記規則に従ってください。

UNIX など一部のプラットフォームでは、値の文字列の前に特定の「エスケープ文字」が必要です。プラットフォームに固有のマニュアルを参照してください。

大文字と小文字の区別

一般的に、プリコンパイラ・オプションの名前および値には、大文字または小文字のどちらを指定してもかまいません。ただし、UNIX のように大 / 小文字を区別するオペレーティング・システムの場合は、大文字および小文字を正しく組み合わせて、Pro*C/C++ 実行ファイルの名前を含むファイル名を指定してください。

プリコンパイラのオプション

各オプションを使用すると、リソースの使用方法、エラーのレポート方法、入出力のフォーマット方法およびカーソルの管理方法を制御できます。

オプションの値はリテラルです。これらの値はテキスト値または数値です。たとえば次のオプションでは、

```
... INAME=my_test
```

値はファイル名を表す文字列リテラルです。

次のオプション MAXOPENCURSORS では、

```
...MAXOPENCURSORS=20
```

値は数値です。

一部のオプションはブール値をとり、文字列 *yes* または *no*、*true* または *false*、あるいは整数リテラル 1 または 0 で表すことができます。たとえば、次のオプションは

```
... SELECT_ERROR=yes
```

次のオプションに相当します。

```
... SELECT_ERROR=true
```

または

```
... SELECT_ERROR=1
```

これらはすべて、SELECT エラーが実行時に検出されることを意味しています。

構成ファイル

構成ファイルは、プリコンパイラ・オプションを格納するテキスト・ファイルです。ファイル内の各レコード（行）には、オプション 1 つと、それに対応付けられた 1 つ以上の値が含まれます。1 行に複数のオプションを入力すると、2 番目以降のオプションは無視されます。たとえば、INTYPE ファイルに次の行が含まれる場合があります。

```
FIPS=YES  
MODE=ANSI  
CODE=ANSI_C
```

これらの行は、FIPS、MODE および CODE オプションにデフォルト値を設定します。

それぞれのインストレーションごとにシステム構成ファイルが 1 つあります。システム構成ファイルの名前は *pcscfg.cfg* で、構成ファイルの位置はシステム固有です。

Pro*C/C++ のユーザーはそれぞれ、1 つ以上のプライベート構成ファイルを持つことができます。構成ファイルの名前は、必ず CONFIG= プリコンパイラ・オプションを使用して指定してください。10-11 ページの「[プリコンパイラ・オプションの使用](#)」を参照してください。

注意： 構成ファイルはネストできません。つまり、構成ファイル内では、CONFIG= は有効ではありません。

オプション値の優先順位

オプションの値は、優先順位の低いものから順に、次のように決定されます。

- プリコンパイラに組み込まれている値
- Pro*C/C++ システム構成ファイル内の値の集合
- Pro*C/C++ ユーザー構成ファイル内の値の集合
- コマンドラインで設定される値
- インラインで設定される値

たとえば、オプション MAXOPENCURSORS はキャッシュ内のオープン・カーソルの最大数を指定します。このオプションのプリコンパイラに組み込まれたデフォルト値は 10 です。ただし、システム構成ファイルで MAXOPENCURSORS=32 を指定すると、デフォルトは 32 になります。ユーザー構成ファイルはこれをさらに別の値に設定することができます。これはシステム構成ファイルの値より優先されます。

最終的には、インライン指定が前述のすべてのデフォルト値よりも優先されます。構成ファイルに関する詳細は 10-6 ページの「[プリコンパイル中の状況](#)」を参照してください。

USERID などの一部のオプションには、プリコンパイラ・デフォルト値がありません。オプションでビルトイン・デフォルト値のあるものを表 10-2 と 10-11 ページの「[プリコンパイラ・オプションの使用](#)」に示します。

注意： プリコンパイラのデフォルト値については、システム固有のマニュアルを参照してください。プラットフォーム上では、この章で示された値から変更されている可能性もあります。

カレント設定値を調べる

コマンドラインで疑問符 (?) を使用すると、1 つ以上のオプションのカレント設定値を対話形式で調べることができます。たとえば次のコマンドを発行すると、

```
proc ?
```

すべてのオプションが、それらのカレント設定値とともに端末に出力（表示）されます。(UNIX システムで C シェルを使用しているときには ? をバックスラッシュでエスケープしてください。) この場合、値はプリコンパイラに組み込まれているもので、システム構成ファイルの値によってオーバーライドされています。

しかしコマンド

proc config=my_config_file.h ?

を入力したときに、カレント・ディレクトリに *my_config_file.h* というファイルがあると、すべてのオプションがリストで表示されます。ユーザー構成ファイルの値によって、不足している値が補われ、Pro*C/C プリコンパイラに組み込まれている値またはシステム構成ファイルに指定されている値を置き換えます。

構成ファイルでは、オプションを1行に1つずつ指定する必要がありますので注意してください。1行に複数のオプションを入力すると、2つ目以降のオプションは無視されます。

オプション名を指定して、その後ろに =? を付けるのみでどれか1つのオプションのカレント値を調べることもできます。たとえば、次のとおりです。

proc maxopencursors=?

このように入力すると、MAXOPENCURSORS オプションのカレント・デフォルト値が出力されます。

たとえば、

proc

と入力すると、表 10-2 「プリコンパイラのオプション」 のような短いサマリーが表示されます。

マクロ・オプションおよびマイクロ・オプション

MODE オプションは複数のオプションを同時に制御します。MODE はマクロ・オプションとも呼ばれます。CLOSE_ON_COMMIT、DYNAMIC および TYPE_CODE などのより新しいオプションは1つの関数のみを制御し、マイクロ・オプションとして知られています。マクロ・オプションは、10-4 ページの「[オプション値の優先順位](#)」に示されるリストで高い優先順位が付けられている場合に限り、マイクロ・オプションに対して優先されます。

次の表は、マクロ・オプション値によって設定されるマイクロ・オプションの値を示しています。

表 10-1 マクロ・オプション値によりマイクロ・オプション値が設定される方法

マクロ・オプション	マイクロ・オプション
MODE=ANSI ISO	CLOSE_ON_COMMIT=YES
	DYNAMIC=ANSI
	TYPE_CODE=ANSI

表 10-1 マクロ・オプション値によりマイクロ・オプション値が設定される方法

マクロ・オプション	マイクロ・オプション
MODE=ORACLE	CLOSE_ON_COMMIT=NO
	END_OF_FETCH=1403
	DYNAMIC=ORACLE
	TYPE_CODE=ORACLE

ユーザー構成ファイルで MODE=ANSI と CLOSE_ON_COMMIT=NO の両方を指定すると、COMMIT してもカーソルはクローズしません。構成ファイルで MODE=ORACLE を指定し、コマンドラインで CLOSE_ON_COMMIT=YES を指定すると、カーソルはクローズします。

プリコンパイル中の状況

プリコンパイル時に、ホスト・プログラムに埋め込まれている SQL 文は、Pro*C/C++ が生成する C または C++ のコードに置換されます。生成されたコードには、データ型、データ長、ホスト変数のアドレスを示すデータ構造の他、ランタイム・ライブラリである SQLLIB に必要なその他の情報も含まれています。またこのコードには、埋込み SQL の処理を実行する SQLLIB ルーチンのコールも入っています。

注意：プリコンパイラは Oracle コール・インターフェイス（OCI）ルーチンの呼び出しは生成しません。

プリコンパイラは警告およびエラー・メッセージを発行する場合があります。これらのメッセージは、『Oracle8i エラー・メッセージ』で説明しています。

表 10-2 は主なプリコンパイラ・オプションの早見表です。10-11 ページの「[プリコンパイラ・オプションの使用](#)」のセクションがまとめられています。受け付けられても効力を持たないオプションは、この表には載っていません。

オプションの適用範囲

プリコンパイル単位は C コードと 1 つ以上の埋込み SQL 文を含むファイルです。特定のプリコンパイル・ユニットに対して指定したオプションは、そのプリコンパイル・ユニットにのみ効力を持ちます。たとえば、ユニット A に対して HOLD_CURSOR=YES および RELEASE_CURSOR=YES を指定し、ユニット B には指定しなければ、ユニット A の SQL 文はこれらの HOLD_CURSOR 値および RELEASE_CURSOR 値を使用して実行されますが、ユニット B の SQL 文はデフォルト値を使用して実行されます。

早見表

表 10-2 は Pro*C/C++ オプションの早見表です。アスタリスクでマークされたオプションはインラインで入力できます。

表 10-2 プリコンパイラのオプション

構文	デフォルト値	仕様
AUTO_CONNECT={YES NO}	NO	最初の実行文の前の自動 OPS\$ アカウント接続
CHAR_MAP={VARCHAR2 CHARZ STRING CHARF}	CHARZ	文字配列および文字列のマッピング
CLOSE_ON_COMMIT={YES NO}	NO	COMMIT 時にすべてのカーソルをクローズ
CODE={ANSI_C KR_C CPP}	KR_C	生成される C コードの種類
COMP_CHARSET={MULTI_BYTE SINGLE_BYTE}	MULTI_BYTE	C/C++ コンパイラがサポートするキャラクタ・セットの型
CONFIG= <i>filename</i>	なし	ユーザーのプライベート構成ファイル
CPP_SUFFIX= <i>extension</i>	なし	出力ファイルのデフォルトのファイルの拡張子を指定します。
DBMS={V7 NATIVE V8}	NATIVE	互換性 (Oracle7、Oracle8i またはプリコンパイル時に接続されていたデータベースのバージョン)
DEF_SQLCODE={YES NO}	NO	#define SQLCODE に対するマクロを生成します。
DEFINE= <i>name</i> *	なし	Pro*C/C++ プリコンパイラで使用する名前を定義します。
DURATION={TRANSACTION SESSION}	TRANSACTION	キャッシュ内のオブジェクトの確保継続時間を設定します。
DYNAMIC={ANSI ORACLE}	ORACLE	Oracle または ANSI SQL の意味を指定します。
ERRORS={YES NO}	YES	エラー・メッセージの送り先 (NO を指定すると、リスト・ファイルのみに送られ、端末には送られません)
ERRTYPE= <i>filename</i>	なし	Intype ファイル・エラー・メッセージのリスト・ファイル名
FIPS={NO SQL89 SQL2 YES} *	なし	ANSI/ISO 非標準拠を切り替えるかどうか
HEADER= <i>extension</i>	なし	プリコンパイルされたヘッダー・ファイルのファイル拡張子
HOLD_CURSOR={YES NO} *	NO	カーソル・キャッシュが SQL 文を処理する方法
[INAME=] <i>filename</i>	なし	入力ファイルの名前

表 10-2 プリコンパイラのオプション

構文	デフォルト値	仕様
INCLUDE= <i>pathname</i> *	なし	EXEC SQL INCLUDE 文または #include 文のディレクトリ・パス
INTYPE= <i>filename</i>	なし	型情報の入力ファイル名
LINES={YES NO}	NO	#line 宣言文を生成するかどうか
LNAME= <i>filename</i>	なし	リスト・ファイルの名前
LTYPE={NONE SHORT LONG}	なし	生成するリスト・ファイルの型（生成する場合）
MAXLITERAL=10 ～ 1024	1024	生成される C コードの文字列リテラルの最大長（バイト）
MAXOPENCURSORS=5 ～ 255 *	10	同時にキャッシュされるオープン・カーソルの最大数
MODE={ANSI ISO ORACLE}	ORACLE	ANSI/ISO または Oracle の動作
NLS_CHAR=(<i>var1</i> ..., <i>varn</i>)	なし	NLS 文字変数を指定する
NLS_LOCAL={YES NO}	NO	NLS 文字の意味を制御する
OBJECTS={YES NO}	YES	オブジェクト型をサポートする
[ONAME]= <i>filename</i>	<i>iname.c</i>	出力（コード）ファイルの名前
ORACA={YES NO} *	NO	ORACA を使用するかどうか
PAGELN=30..256	80	リスト・ファイルのページ長
PARSE={NONE PARTIAL FULL}	FULL	Pro*C/C++ が（C 解析機能で）.pc ソース・コードを解析するかどうか
PREFETCH=0..65535	1	任意の行数を事前に取得して問い合わせをスピード・アップする
RELEASE_CURSOR={YES NO} *	NO	カーソル・キャッシュからのカーソルの解放を制御します。
SELECT_ERROR={YES NO} *	YES	SELECT エラーのフラグ付け
SQLCHECK={SEMANTICS SYNTAX} *	SYNTAX	プリコンパイル時の SQL チェック量
SYS_INCLUDE= <i>pathname</i>	なし	iostream.h などのシステム・ヘッダー・ファイルがあるディレクトリ
THREADS={YES NO}	NO	マルチスレッド・アプリケーションを指定します。

表 10-2 プリコンパイラのオプション

構文	デフォルト値	仕様
TYPE_CODE={ORACLE ANSI}	ORACLE	動的 SQL の Oracle または ANSI 型コードの使用方法
UNSAFE_NULL={YES NO}	NO	UNSAFE_NULL=YES と指定すると ORA-01405 メッセージが使用禁止になります。
USERID= <i>username/password</i> [@ <i>dbname</i>]	なし	<i>username/password</i> [@ <i>dbname</i>] 接続文字列
VARCHAR={YES NO}	NO	暗黙的 VARCHAR 構造体の使用を許可するかどうか
VERSION={ANY LATEST RECENT} *	RECENT	どのバージョンのオブジェクトを戻すか

オプションの入力

どのプリコンパイラ・オプションも、コマンドラインに入力できます。また、その多くは、EXEC ORACLE OPTION 文を使用してプリコンパイラ・プログラムのソース・ファイルにインライン入力できます。

コマンドライン

プリコンパイラ・オプションをコマンドラインに入力するには、次の構文を使用します。

```
... [OPTION_NAME=value] [OPTION_NAME=value] ...
```

それぞれのオプション = 値の指定は、1 つ以上のスペースで区切ります。たとえば、次のように入力します。

```
... CODE=ANSI_C MODE=ANSI
```

インライン

次の構文を使用して EXEC ORACLE 文を記述すると、オプションをインライン入力できます。

```
EXEC ORACLE OPTION (OPTION_NAME=value);
```

たとえば、次のように記述します。

```
EXEC ORACLE OPTION (RELEASE_CURSOR=yes);
```

EXEC ORACLE の用途

EXEC ORACLE 機能は、プリコンパイル中にオプション値を変更するのに特に便利です。たとえば、HOLD_CURSOR と RELEASE_CURSOR を 1 文単位で変更する場合があります。[付録 C「パフォーマンスの最適化」](#)にインライン・オプションを使用してランタイム性能を最適化する方法を示します。

また、コマンドラインで入力できる文字数が、使用しているオペレーティング・システムで制限されているときは、オプションをインラインまたは構成ファイルで指定すると便利です。

EXEC ORACLE の適用範囲

EXEC ORACLE 文は、同一オプションを指定した別の EXEC ORACLE 文によってオプション指定値（テキスト）が変更されるまで有効です。次の例では、HOLD_CURSOR=NO は HOLD_CURSOR=YES が指定されるまで有効です。

```
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);
```

プリコンパイラ・オプションの使用

この項は、プリコンパイラ・オプションを簡単に参照できるように構成されています。プリコンパイラ・オプションはアルファベット順に示します。また、オプション別にその用途、構文およびデフォルト値も示してあります。さらに「使用上の注意」の欄で、オプションの働きを説明します。

AUTO_CONNECT

用途

OPS\$ アカウントへの自動接続を可能にします。

構文

AUTO_CONNECT={YES | NO}

デフォルト値

NO

使用上の注意

入力できるのは、コマンドラインまたは構成ファイルからのみです。

AUTO_CONNECT=YES で、最初の実行 SQL 文を処理するときにアプリケーションがまだデータベースに接続されていない場合、アプリケーションは次のユーザー ID を使用して接続を試みます。

OPS\$*username*

このとき *username* はカレント・オペレーティング・システムのユーザー名またはタスク名です。OPS\$*username* は有効な Oracle ユーザー ID です。

AUTO_CONNECT=NO のとき、Oracle に接続するにはプログラム内で CONNECT 文を使用する必要があります。

CHAR_MAP

用途

char 型または char[n] 配列型の C ホスト変数およびそれらへのポインタの SQL へのデフォルトのマッピングを指定します。

構文

CHAR_MAP={VARCHAR2 | CHARZ | STRING | CHARF}

デフォルト値

CHARZ

使用上の注意

リリース 8.0 より前のバージョンでは、SQL DECLARE 文を使用して CHAR などの char または char[n] ホスト変数を宣言する必要がありました。外部データ型 VARCHAR2 および CHARZ が Oracle7 のデフォルト文字マッピングでした。

CHAR_MAP 設定の表、データ型の説明、それがデフォルトになる場所は 4-45 ページの「[各国語サポート](#)」を参照してください。Pro*C/C++ での CHAR_MAP の使用例は 5-3 ページの「[CHAR_MAP オプションのインラインでの使用方法](#)」にあります。

CLOSE_ON_COMMIT

用途

文のコミット時にすべてのカーソルをクローズするかどうかを指定します。

構文

CLOSE_ON_COMMIT={YES | NO}

デフォルト値

NO

使用上の注意

入力できるのは、コマンドラインまたは構成ファイルからのみです。

MODE が CLOSE_ON_COMMIT よりも高いレベルに指定されている場合、MODE が優先されます。たとえば、デフォルトで MODE=ORACLE および CLOSE_ON_COMMIT=NO と設定されている場合、ユーザーがコマンドラインで MODE=ANSI と指定すると、コミット時にすべてのカーソルがクローズされます。

COMMIT または ROLLBACK を発行すると、明示カーソルがすべてクローズされます。MODE=ORACLE のときにコミットまたはロールバックを出すと、CURRENT OF 句内で参照されたカーソルのみがクローズします。

詳細は 6-14 ページの「[CLOSE_ON_COMMIT プリコンパイラ・オプション](#)」を参照してください。このオプションの優先順位の詳細は 10-5 ページの「[マクロ・オプションおよびマイクロ・オプション](#)」を参照してください。

CODE

用途

Pro*C/C++ プリコンパイラが生成する C 関数プロトタイプの形式を指定します。（関数プロトタイプを使用して関数およびその引数のデータ型を宣言します。）プリコンパイラは SQL ライブラリ・ルーチン用の関数プロトタイプを生成します。それによって、C コンパイラは外部参照を解決できます。CODE オプションによってプロトタイプの生成を制御します。

構文

CODE={ANSI_C | KR_C | CPP}

デフォルト値

KR_C

使用上の注意

コマンドラインで入力することはできますが、インラインにはできません。

ANSI C 規格 X3.159-1989 は、関数プロトタイプを規定しています。CODE=ANSI_C のとき、Pro*C/C++ は完全な関数プロトタイプを生成します。この関数プロトタイプは ANSI C 規格に準拠するものです。次に例を示します。

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

プリコンパイラを使用して他の ANSI 準拠のコード（**const** 型修飾子など）も生成できます。

CODE=KR_C（デフォルト）のとき、生成された関数プロトタイプの引数リストは次に示すようなコメントになります。

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

C コンパイラが X3.159 規格に準拠していなければ、CODE=KR_C を指定します。

CODE=CPP のとき、プリコンパイラは C++ 互換コードを生成します。このオプション値を使用するすべての結果は 12-3 ページの「[コードの生成](#)」を参照してください。

COMP_CHARSET

用途

使用するコンパイラがマルチバイト・キャラクタ・セットをサポートするかどうかを Pro*C/C++ プリコンパイラに指定します。このオプションはマルチバイトのクライアント側環境（たとえば NLS_LANG がマルチバイト・キャラクタ・セットに設定されているとき）で作業する開発者向けです。

構文

COMP_CHARSET={MULTI_BYTE | SINGLE_BYTE}

デフォルト値

MULTI_BYTE

使用上の注意

入力できるのは、コマンドラインでのみです。

COMP_CHARSET=MULTI_BYTE（デフォルト）のとき、Pro*C/C++ はマルチバイトの NLS キャラクタ・セットをサポートしているコンパイラがコンパイルする C コードを生成します。

COMP_CHARSET=SINGLE_BYTE が指定されたときに Pro*C/C++ が生成する C コードはシングルスバイト系コンパイラ用です。マルチバイト文字列中の 2 バイト文字の 1 バイトは、円記号 (¥) に対応する ASCII 文字で、この文字が面倒な問題を引き起こすことがあります。前述で生成された C コードによりこの問題に対処できます。この場合、円記号 (¥) は先行するもう 1 つの円記号でエスケープされます。

注意： この機能は古い C コンパイラを使用してシフト JIS 環境で開発をするときには一般に必要となります。

NLS_LANG がシングルスバイト・キャラクタ・セットに設定されていると、このオプションは効力をもちません。

CONFIG

用途

ユーザー構成ファイルの名前を指定します。

構文

CONFIG=*filename*

デフォルト値

なし

使用上の注意

入力できるのは、コマンドラインでのみです。

ユーザー構成ファイルの名前および位置を Pro*C/C++ に通知する方法は、このオプション以外にありません。

CPP_SUFFIX

用途

CPP_SUFFIX オプションを使用すると、CODE=CPP オプションが指定されているときに生成される C++ 出力ファイルにプリコンパイラが付加するファイルの拡張子を指定できます。

構文

CPP_SUFFIX=*filename_extension*

デフォルト値

システム固有

使用上の注意

ほとんどの C コンパイラでは、入力ファイルのデフォルト拡張子は ".c" とみなされます。しかし、C++ コンパイラでは、想定されるファイルの拡張子がコンパイラごとに異なる場合があります。CPP_SUFFIX オプションを使用すると、プリコンパイラが生成するファイルの拡張子を指定できます。このオプションの値は、引用符もピリオドも付けない文字列です。たとえば、CPP_SUFFIX=cc または CPP_SUFFIX=C のように指定します。

DBMS

用途

Oracle で Oracle8i、Oracle7、もしくは Oracle のシステム固有なバージョン（つまりアプリケーションが接続されているバージョン）の意味規則と構文規則に従うかを指定します。

構文

DBMS=NATIVE | V7 | V8

デフォルト値

NATIVE

使用上の注意

入力できるのは、コマンドラインまたは構成ファイルからのみです。

DBMS オプションの指定により Oracle のバージョンに固有の動作を制御できます。
DBMS=NATIVE（デフォルト値）のとき、Oracle はアプリケーションが接続しているデータベース・バージョンの意味上および構文上の規則に従います。

DBMS=V8 または DBMS=V7 のときは、Oracle はそれぞれ Oracle8i の規則（Oracle7 より変更なし）に従います。

Oracle8i では V6_CHAR はサポートされておらず、この機能はプリコンパイラ・オプション CHAR_MAP（10-11 ページの「[CHAR_MAP](#)」を参照してください。）に用意されています。

表 10-3 DBMS と MODE の相互作用

状況	DBMS=V7 V8 MODE=ANSI	DBMS=V7 V8 MODE=ORACLE
「データが見つかりません」警告コード	+100	+1403
標識変数を使用しない NULL のフェッチ	エラー -1405	エラー -1405
標識変数を使用しない切捨て値のフェッチ	エラーはなし。 sqlwarn[1] が設定されます。	エラーはなし。sqlwarn[1] が設定されます。
COMMIT または ROLLBACK によるカーソルのクローズ	すべて明示的に	CURRENT OF のみ
すでにオープンしているカーソルのオープン	エラー -2117	エラーにならない
すでにクローズしているカーソルのクローズ	エラー -2114	エラーにならない
SQL グループ関数による NULL の無視	警告なし	警告なし
複数行の間合せて SQL グループ関数をコールするタイミング	FETCH 時	FETCH 時
SQLCA 構造体の宣言	オプション	必須
SQLCODE または SQLSTATE ステータス変数の宣言	必須	指定できるが Oracle は無視
整合性制約	有効	有効
ロールバック・セグメント用 PCTINCREASE	使用不可	使用不可
MAXEXTENTS 記憶領域パラメータ	使用不可	使用不可

DEF_SQLCODE

用途

Pro*C/C++ プリコンパイラにより SQLCODE の **#define** が生成されるかどうかを制御します。

構文

DEF_SQLCODE={NO | YES}

デフォルト値

NO

使用上の注意

入力できるのは、コマンドラインまたは構成ファイルからのみです。

DEF_SQLCODE=YES のとき、プリコンパイラは生成するソース・コードに SQLCODE を次のように定義します。

```
#define SQLCODE sqlca.sqlcode
```

この定義があれば、SQLCODE を使用して実行 SQL 文の結果をチェックできます。

DEF_SQLCODE オプションは、SQLCODE の使用を義務づけている規格への準拠を目的としています。

また次の行のどちらかを入力することによって、ソース・コードに SQLCA を組み込む必要があります。

```
#include <sqlca.h>
```

または

```
EXEC SQL INCLUDE SQLCA;
```

SQLCA を組み込まなければ、このオプションを使用するとプリコンパイル時にエラーが発生します。

DEFINE

用途

#ifdef および **#ifndef** Pro*C/C++ プリコンパイラ宣言文で使用できる名称を定義します。定義された名称は EXEC ORACLE IFDEF と EXEC ORACLE IFNDEF 文でも使用できます。

構文

DEFINE=*name*

デフォルト値

なし

使用上の注意

コマンドラインまたはインラインで入力できます。DEFINE では名称しか定義できません。マクロは定義できません。たとえば次のような定義は無効です。

```
proc my_prog DEFINE=LEN=20
```

DEFINE 文を正しく使用すれば、次のような指定ができます。

```
proc my_prog DEFINE=XYZZY
```

すると *my_prog.pc* に次のコードを記述できます。

```
#ifdef XYZZY
...
#else
...
#endif
```

または、単純にコーディングできます。

```
EXEC ORACLE IFDEF XYZZY;
...
EXEC ORACLE ELSE;
...
EXEC ORACLE ENDIF;
```

次の例は無効です。

```
#define XYZZY
...
EXEC ORACLE IFDEF XYZZY
...
EXEC ORACLE ENDIF;
```

EXEC ORACLE DEFINE または DEFINE オプションでマクロが定義されているときにかぎり、EXEC ORACLE 条件文が有効となります。

DEFINE= を使用して名前を定義してから Pro*C/C++ プリコンパイラの **#ifdef**（または **#ifndef**）宣言文を使用して条件を含めた（または除外した）場合、C コンパイラを実行するときにその名前が定義されていることを確認します。たとえば、UNIX の *cc* では、**-D** オプションを使用して C コンパイラの名前を定義する必要があります。

DURATION

用途

後続の EXEC SQL OBJECT CREATE 文と EXEC SQL OBJECT DEREf 文に使用される保持期間を設定します。キャッシュ内のオブジェクトは、保持期間の終わりに暗黙的に解放されます。

構文

DURATION={TRANSACTION | SESSION}

デフォルト値

TRANSACTION

使用上の注意

EXEC ORACLE OPTION 文を使用してインライン入力できます。

TRANSACTION は、オブジェクトがトランザクションの完了時に暗黙的に解放されることを意味します。

SESSION は、オブジェクトが接続の終了時に暗黙的に解放されることを意味します。

DYNAMIC

用途

このマイクロ・オプションは、動的 SQL 方法 4 の記述子の動作を指定します。MODE の設定により DYNAMIC の設定が決定されます。

構文

DYNAMIC={ORACLE | ANSI}

デフォルト値

ORACLE

使用上の注意

EXEC ORACLE OPTION 文を使用してインライン入力できません。

DYNAMIC オプション設定は 14-11 ページの「[表 14-2](#)」を参照してください。

ERRORS

用途

エラー・メッセージを端末とリスト・ファイルの両方に送信（YES）するか、リスト・ファイルのみに送信（NO）するかを指定します。

構文

ERRORS={YES | NO}

デフォルト値

YES

使用上の注意

入力できるのは、コマンドラインまたは構成ファイルからのみです。

ERRTYPE

用途

型ファイルの処理時に生成されたエラーを書き込む出力ファイルを指定します。このオプションを省略すると、エラーは画面に出力されます。（10-25 ページの「[INTYPE](#)」を参照してください。）

構文

ERRTYPE=*filename*

デフォルト値

なし

使用上の注意

エラー・ファイルが1つのみ生成されます。複数の値を入力すると、最後の値がプリコンパイラに使用されます。

FIPS

用途

ANSI SQL の拡張要素にフラグを立てる（FIPS フラガーで）かどうかを指定します。拡張要素とは、ANSI の形式または構文規則（権限付与規則は除く）に違反する SQL 要素を指します。

構文

FIPS={SQL89 | SQL2 | YES | NO}

デフォルト値

なし

使用上の注意

インラインまたはコマンドラインで入力できます。

FIPS=YES のとき、FIPS フラガーは使用可能になります。このとき ANSI SQL の Oracle 拡張要素または非標準拠の方法で ANSI SQL 機能を使用すると、警告メッセージ（エラーはありませんでした）が発行されます。プリコンパイル時にフラグ付けされる ANSI SQL 拡張要素には次のものがあります。

- FOR 句を含む配列インタフェース
- SQLCA、ORACA および SQLDA データ構造体
- DESCRIBE 文を含む動的 SQL
- 埋込み PL/SQL ブロック
- 自動データ型変換
- DATE、NUMBER、RAW、LONGRAW、VARRAW、ROWID、VARCHAR2 および VARCHAR データ型
- ポインタ・ホスト変数
- ランタイム・オプション指定用の Oracle OPTION 文
- ユーザー・イグジットの IAF 文
- CONNECT 文
- TYPE および VAR データ型の同等文
- AT *db_name* 句
- DECLARE...DATABASE 文、...STATEMENT 文および ...TABLE 文

- WHENEVER 文での SQLWARNING 条件
- WHENEVER 文における DO *function_name*()、「do break」アクションおよび「do continue」アクション
- COMMIT 文での COMMENT 句と FORCE TRANSACTION 句
- ROLLBACK 文での FORCE TRANSACTION 句および TO SAVEPOINT 句
- COMMIT 文および ROLLBACK 文での RELEASE パラメータ
- WHENEVER...GOTO ラベルおよび INTO 句のホスト変数の前に任意指定で付加するコロン

HEADER

用途

プリコンパイル済ヘッダー・ファイルを許可します。プリコンパイルされたヘッダー・ファイルのファイル拡張子を指定します。

構文

HEADER=*extension*

デフォルト値

なし

使用上の注意

ヘッダー・ファイルをプリコンパイルする場合、このオプションは必須で、ヘッダー・ファイルのプリコンパイルによって生成される出力ファイルのファイル拡張子を指定するために使用されます。

通常の Pro*C/C++ プログラムをプリコンパイルする場合、このオプションはオプションで設定します。指定すると、Pro*C/C++ プログラムのプリコンパイル時にヘッダーのプリコンパイル機能が使用できます。

どちらの場合も、このオプションで #include 宣言文を処理するときに使用するファイル拡張子を指定できます。指定した拡張子の #include ファイルが存在する場合、そのファイルは Pro*C/C++ で以前に生成されたプリコンパイルされたヘッダー・ファイルとみなされます。#include 宣言文を処理してそこに含まれるヘッダー・ファイルをプリコンパイルするかわりに、ファイルのデータがインスタンス化されます。

このオプションは、コマンドラインまたは構成ファイルでのみ使用できます。インラインでは使用できません。このオプションを使用する場合、ファイル拡張子のみを指定します。ファイル・セパレータは含めないでください。たとえば、拡張子にピリオド (.) は含めません。

使用方法は 5-34 ページの「[プリコンパイル済のヘッダー・ファイル](#)」を参照してください。

HOLD_CURSOR

用途

カーソル・キャッシュでの SQL 文および PL/SQL ブロック用カーソルの取扱い方法を指定します。

構文

HOLD_CURSOR={YES | NO}

デフォルト値

NO

使用上の注意

インラインまたはコマンドラインで入力できます。

HOLD_CURSOR を使用すると、プログラムのパフォーマンスを改善できます。詳細は[付録 C「パフォーマンスの最適化」](#)を参照してください。

SQL DML 文を実行すると、その文に対応付けられたカーソルがカーソル・キャッシュ内のエントリにリンクされます。続いてカーソル・キャッシュ・エントリは Oracle プライベート SQL 領域にリンクされます。この領域には SQL 文の実行に必要な情報が格納されます。HOLD_CURSOR はカーソルとカーソル・キャッシュの間のリンクで発生する処理を制御します。

HOLD_CURSOR=NO のとき、Oracle が SQL 文を実行し、カーソルがクローズされた後に、プリコンパイラがそのリンクに再利用可能のマークを付けます。このリンクは、それが示すカーソル・キャッシュ・エントリが別の SQL 文に必要なになると、すぐに再利用されます。これにより、プライベート SQL 領域に割り当てられたメモリーが解放され、解析ロックが解除されます。

HOLD_CURSOR=YES で RELEASE_CURSOR=NO のときは、リンクが保持されます。つまりプリコンパイラはそのリンクを再利用しません。これは、以降の実行速度を改善するので頻繁に実行される SQL 文には役立ちます。文の再解析や Oracle プライベート SQL 領域用のメモリー割当ては不要です。

暗黙カーソル用としてインラインで使用するときは、SQL 文の実行前に HOLD_CURSOR を設定してください。明示カーソル用としてインラインで使用するときは、カーソルをクローズする前に HOLD_CURSOR を設定してください。

RELEASE_CURSOR=YES で HOLD_CURSOR=YES がオーバーライドされ、
HOLD_CURSOR=NO で RELEASE_CURSOR=NO がオーバーライドされます。このふたつのオプションの相互作用方法を示す情報は C-12 ページの「表 C-1」を参照してください。

INAME

用途

入力ファイル名を指定します。

構文

INAME=*path_and_filename*

デフォルト値

なし

使用上の注意

入力できるのは、コマンドラインでのみです。

すべての入力ファイル名はプリコンパイル時に一意である必要があります。

ファイル名拡張子が *.pc* の場合は、ファイル名拡張子を省略できます。入力ファイル名がコマンドラインの先頭オプションになっているときは、オプションの INAME= の部分を省略できます。たとえば、次のとおりです。

```
proc sample1 MODE=ansi
```

この例では、ANSI モードを使用してファイル *sample1.pc* をプリコンパイルします。このコマンドは次のコマンドに相当します。

```
proc INAME=sample1 MODE=ansi
```

INCLUDE

用途

#include または EXEC SQL INCLUDE 宣言文によってインクルードするファイルのディレクトリ・パスを指定します。

構文

INCLUDE=*pathname* または INCLUDE=(*path_1,path_2,...,path_n*)

デフォルト値

Pro*C/C++ にインクルードするカレント・ディレクトリおよびパス

使用上の注意

インラインまたはコマンドラインで入力できます。

INCLUDE はインクルードされたファイルに対するディレクトリ・パスを指定するために使用します。プリコンパイラは次の順序でディレクトリを検索します。

1. カレント・ディレクトリ
2. SYS_INCLUDE プリコンパイラ・オプションに指定されているシステム・ディレクトリ
3. INCLUDE オプションで指定された入力順のディレクトリ
4. 標準ヘッダー・ファイル用の組込みディレクトリ

通常は Oracle 固有のヘッダー・ファイル *sqlca.h* や *sqllda.h* などのディレクトリ・パスを指定する必要はありません。

注意：インクルードするファイルに Oracle 固有のファイル名を拡張子なしで指定すると、Pro*C/C++ は拡張子 *.h* を仮定します。このため、インクルードするファイルには、*.h* でなくても拡張子を指定する必要があります。

他のすべてのヘッダー・ファイルの場合、プリコンパイラは *.h* 拡張子を想定しません。

非標準ファイルの場合は、カレント・ディレクトリに格納されていない限り、INCLUDE を使用してディレクトリ・パスを指定する必要があります。次に示すように、コマンドラインに複数のパスを指定できます。

```
... INCLUDE=path_1 INCLUDE=path_2 ...
```

警告：インクルードするファイルが別ディレクトリに存在している場合には、同じ名前のファイルをカレント・ディレクトリに存在させないでください。

INCLUDE オプションを使用してディレクトリ・パスを指定する構文はシステムによって異なります。各オペレーティング・システムの規則に従ってください。

INTYPE

用途

OTT に生成された型ファイルを 1 つ以上指定します。(アプリケーションでオブジェクト型が使用される場合にのみ必要です。)

構文

```
INTYPE=(file_1,file_2,...,file_n)
```

デフォルト値

なし

使用上の注意

Pro*C/C++ コードにはオブジェクト型 1 つにつき、1 つの型ファイルが存在します。

LINES

用途

Pro*C/C++ プリコンパイラがその出力ファイルに **#line** プリプロセッサ宣言文を追加するかどうかを指定します。

構文

LINES={YES | NO}

デフォルト値

NO

使用上の注意

入力できるのは、コマンドラインでのみです。

LINES オプションはデバッグに便利です。

LINES=YES のとき、Pro*C/C++ プリコンパイラはその出力ファイルに **#line** プリプロセッサ宣言文を追加します。

通常、C コンパイラはそれぞれの入力行を処理するたびに行カウントを増分します。**#line** 宣言文はコンパイラの入力行カウンタを強制的にリセットして、プリコンパイラが生成したコードを数えないようにします。さらに、入力ファイルの名前が変わったときに、次の **#line** 宣言文が新しいファイル名を指定します。

C コンパイラは行番号およびファイル名を使用してエラーの発生位置を示します。したがって、C コンパイラによって発行されたエラー・メッセージは、修正済（プリコンパイル済）のソース・ファイルではなく、元のソース・ファイルを常に参照します。これにより、大半のデバッグを使用した元のソース・コードを参照できます。

LINES=NO（デフォルト）のときは、プリコンパイラは出力ファイルに **#line** 宣言文を追加しません。

注意：5-28 ページの「無視される宣言文」では、Pro*C/C++ プリコンパイラは **#line** 宣言文をサポートしていないことが明示されています。これは、プリコンパイラ・ソースでは **#line** 宣言文を直接コーディングできないことを意味します。ただし、**LINES=** オプションを使用すれば、プリコンパイラに **#line** 宣言文を挿入させることができます。

LNAME

用途

リスト・ファイル名を指定します。

構文

LNAME=*filename*

デフォルト値

なし

使用上の注意

入力できるのは、コマンドラインでのみです。

リスト・ファイルのデフォルトのファイル名の拡張子は *.lis* です。

LTYPE

用途

生成されるリスト・ファイルの型を指定します。

構文

LTYPE={NONE | SHORT | LONG}

デフォルト値

SHORT

使用上の注意

コマンドラインまたは構成ファイルで入力できます。

リスト・ファイル生成時のデフォルトの形式は LONG です。LTYPE=LONG のとき、すべてのソース・コードが解析順に出力されます。また、メッセージも生成順に出力されます。さらに、現在有効な Pro*C/C++ オプションを出力します。

LTYPE=SHORT を指定すると、生成されたメッセージのみが出力され、ソース・コードは出力されません。ソース・ファイルへの参照行はメッセージ条件を生成したコードを突き止めるのに役立ちます。

LTYPE=NONE のとき、LNAME オプションでリスト・ファイル名を明示的に指定しない限り、リスト・ファイルは作成されません。後者の場合、リスト・ファイルは LTYPE=LONG を想定して生成されます。

MAXLITERAL

用途

プリコンパイラによって生成される文字列リテラルの最大長を指定します。これによってコンパイラの制限を超えないようにします。

構文

MAXLITERAL=*integer* (範囲は 10 ～ 1024)

デフォルト値

1024

使用上の注意

インラインでは入力できません。

MAXLITERAL に指定できる最大値はコンパイラによって異なります。たとえば、C コンパイラには 512 文字を超える文字列リテラルを扱えないものがあるため、そのような場合は MAXLITERAL=512 と指定します。

MAXLITERAL で指定した長さを超える文字列はプリコンパイル中に分割され、実行時に再び結合（連結）されます。

インラインで MAXLITERAL を入力することはできますが、プログラムで値を設定できるのは 1 回のみで EXEC ORACLE 文を最初の EXEC SQL 文の前に指定する必要があります。そうにしない場合、Pro*C/C++ により警告メッセージが発行されます。余分または誤って指定した EXEC ORACLE 文は無視され、処理が続行されます。

MAXOPENCURSORS

用途

プリコンパイラがキャッシュしたままにしておく同時オープン・カーソルの数を指定します。

構文

MAXOPENCURSORS=*integer*

デフォルト値

10

使用上の注意

インラインまたはコマンドラインで入力できます。

MAXOPENCURSORS を使用すると、プログラムのパフォーマンスを改善できます。詳細は、[付録 C「パフォーマンスの最適化」](#)を参照してください。

分割プリコンパイルをするときは、MAXOPENCURSORS を 2-9 ページの「[プログラミングのガイドライン](#)」に記述された方法で使用してください。

MAXOPENCURSORS オプションには、SQLLIB カーソル・キャッシュの初期サイズを指定します。

暗黙的文が実行され、HOLD_CURSOR=NO のとき、あるいはカーソルがクローズされたとき、カーソル・エントリには再利用可能のマークが付けられます。この文が再度発行され、このカーソル・エントリが別の文で使用されていない場合は再利用されます。

新しいカーソルが必要で、割当て済カーソル数が MAXOPENCURSORS より少ない場合は、キャッシュ内の次のカーソルが割り当てられます。MAXOPENCURSORS を超えると Oracle はまず以前のエントリを再利用しようとします。空いているエントリがない場合、追加キャッシュ・エントリが割り当てられます。Oracle はプログラムがメモリーを消費するか、データベース・パラメータの OPEN_CURSORS が超過するまで継続してこれを行います。

通常処理で、HOLD_CURSOR=NO と RELEASE_CURSOR=NO（デフォルト）を使用する場合は、MAXOPENCURSORS をデータベース・パラメータの OPEN_CURSORS よりも少なくとも 6 以上小さい数を設定し、データ辞書で使用するカーソルが文を処理できるようにすることをお勧めします。

プログラムが同時オープン・カーソル数を増す必要がある場合には、MAXOPENCURSORS を必要な数まで増やして指定し直すことがあります。45 ～ 50 の値を指定することは珍しくありませんが、ユーザー・プロセスのメモリー領域にカーソル 1 つにつきプライベート SQL 領域が必要です。デフォルト値の 10 は、大半のプログラムには適切な値です。

MODE

用途

プログラムが Oracle の動作規則に従うかどうか、またはカレント ANSI/ISO SQL 規格に準拠するかどうかを指定します。

構文

MODE={ANSI | ISO | ORACLE}

デフォルト値

ORACLE

使用上の注意

入力できるのは、コマンドラインまたは構成ファイルからのみです。

ISO は ANSI のシノニムです。

MODE=ORACLE（デフォルト）のとき、埋込み SQL プログラムは Oracle の動作規則に従います。たとえば、宣言文はオプションで、空白セルは削除されます。

MODE=ANSI のときは、プログラムは完全に ANSI SQL 規格に準拠します。この設定変更で次のように処理が変化します。

- COMMIT または ROLLBACK を発行すると、明示カーソルがすべてクローズされます。
- すでにオープンされているカーソルの OPEN や、クローズされているカーソルの CLOSE はできません。（MODE=ORACLE のときは、再解析を避けるためにオープン状態のカーソルを再度 OPEN できます。）
- すべての EXEC SQL 文のスコープ内に `SQLCODE` という名前の **long** 変数または **char** `SQLSTATE[6]` 変数（どちらの変数も大文字にする必要があります）を宣言する必要があります。すべての場合に同一の `SQLCODE` または `SQLSTATE` 変数を使用する必要はありません。つまり、変数はグローバル変数でなくてもかまいません。
- `SQLCA` の宣言はオプションです。`SQLCA` を挿入する必要はありません。
- `SQLCODE` に戻される「データが見つかりません」という Oracle 警告コードは +1403 から +100 に変わりました。メッセージ文字列は変更されていません。
- ホスト変数に宣言文が必要です。

NLS_CHAR

用途

プリコンパイラで各国語サポート（NLS）マルチバイト文字変数として扱われる C ホスト文字変数を指定します。

構文

NLS_CHAR=*varname* または NLS_CHAR=(*var_1,var_2,...,var_n*)

デフォルト値

なし

使用上の注意

入力できるのは、コマンドラインまたは構成ファイルからのみです。

このオプションを使用すると、プリコンパイラで各国語文字変数として扱う必要があるホスト変数をプリコンパイル時に指定できます。このオプションでは、C の *char* 変数または Pro*C/C++ の VARCHAR 変数のみを指定できます。

オプション・リストにプログラムで宣言していない変数を指定すると、プリコンパイラはエラーを発生しません。

NLS_LOCAL

用途

プリコンパイラのランタイム・ライブラリ SQLLIB、またはデータベース・サーバーのどちらで NLS 文字変換を実行するかを指定します。

構文

NLS_LOCAL={YES | NO}

デフォルト値

NO

使用上の注意

YES を指定すると、Pro*C/C++ および SQLLIB ライブラリによってローカルのマルチバイト・サポートが提供されます。オプション NLS_CHAR を使用して、マルチバイトの C ホスト変数を指定する必要があります。

NO を指定すると、Pro*C/C++ はマルチバイトの処理にデータベース・サーバーのサポートを使用します。すべての新しいアプリケーションでは NLS_LOCAL を NO に設定してください。

環境変数 NLS_NCHAR には有効な固定幅各国キャラクタ・セットを設定する必要があります。変数幅各国キャラクタ・セットはサポートされていません。

入力できるのは、コマンドラインまたは構成ファイルからのみです。

OBJECTS

用途

オブジェクト型のサポートを要求します。

構文

OBJECTS={YES | NO}

デフォルト値

YES

使用上の注意

コマンドラインからのみ入力できます。

ONAME

用途

出力ファイル名を指定します。出力ファイルは、プリコンパイラが生成する C コードのファイルです。

構文

ONAME=*path_and_filename*

デフォルト値

.c 拡張子付きの INAME

使用上の注意

入力できるのは、コマンドラインのみです。このオプションを使用すると、入力（.pc）ファイルとは異なるパス名を使用して、出力ファイルの完全パス名を指定できます。たとえば、次のコマンドを発行する場合を考えます。

```
proc iname=my_test
```

デフォルト出力名は *my_test.c* です。出力ファイル名を *my_test_1.c* にする場合は、次のコマンドを発行します。

```
proc iname=my_test oname=my_test_1.c
```

ONAME で指定するファイルにはデフォルトで拡張子が追加されないため、.c を付加する必要があります。

注意：出力ファイル名をデフォルトのまま使用せず、ONAME を使用して明示的に名前を付けることをお勧めします。拡張子なしで ONAME 値を指定すると、生成ファイルの名前に拡張子は付きません。

ORACA

用途

プログラムが Oracle 通信領域（ORACA）を使用できるかどうかを指定します。

構文

```
ORACA={YES | NO}
```

デフォルト値

NO

使用上の注意

インラインまたはコマンドラインで入力できます。

ORACA=YES のときは、プログラムに EXEC SQL INCLUDE ORACA 文または **#include oraca.h** 文を記述する必要があります。

PAGELEN

用途

リスト・ファイルの物理ページの行数を指定します。

構文

PAGELEN=*integer*

デフォルト値

80

使用上の注意

インラインでは入力できません。値の許容範囲は 30 ～ 256 です。

PARSE

用途

Pro*C/C++ プリコンパイラでソース・ファイルを解析する方法を指定します。

構文

PARSE={FULL | PARTIAL | NONE}

デフォルト値

FULL

使用上の注意

PARSE オプションの詳細は 12-4 ページの「[コードの解析](#)」を参照してください。

C++ 互換コードを生成するには、PARSE オプションに NONE または PARTIAL のどちらかを指定する必要があります。

PARSE=NONE または PARSE=PARTIAL の場合、すべてのホスト変数は宣言文の内側で宣言する必要があります。2-9 ページの「[宣言文](#)」を参照してください。

変数 SQLCODE を宣言文の内側で宣言しない場合、エラー検出の信頼性がなくなります。使用しているプラットフォームの PARSE のデフォルト値をチェックしてください。

PARSE=FULL のときは C 解析機能が動作して、コードにある C++ クラスなどの構造体を認識しません。

PARSE=FULL または PARSE=PARTIAL を指定すると、Pro*C/C++ は `#define`、`#ifdef` などの C プリプロセッサ宣言文を完全にサポートします。ただし、PARSE=NONE の場合には EXEC ORACLE 文により条件プリプロセッシングがサポートされています。2-13 ページの「[条件付きプリコンパイル](#)」を参照してください。

注意：一部のプラットフォームでは、PARSE のデフォルト値が FULL ではないので注意してください。各システムのマニュアルを参照してください。

PREFETCH

用途

任意の行数を事前に取得して問合せをスピード・アップします。

構文

PREFETCH=*integer*

デフォルト値

1

使用上の注意

構成ファイルまたはコマンドラインで入力できます。優先順位の規則に従い、明示カーソルを使用するすべての問合せに整数の値が実行に使用されます。

インラインで使用する場合、明示カーソルのある OPEN 文の前に置く必要があります。OPEN が実行されるときに事前にフェッチされる行数は、最後のインラインの有効な PREFETCH オプションにより指定されます。

値の許容範囲は 0 ～ 65535 です。

RELEASE_CURSOR

用途

カーソル・キャッシュでの SQL 文および PL/SQL ブロック用カーソルの取扱方法を指定します。

構文

RELEASE_CURSOR={YES | NO}

デフォルト値

NO

使用上の注意

インラインまたはコマンドラインで入力できます。

RELEASE_CURSOR を使用すると、プログラムのパフォーマンスを改善できます。詳細は、[付録 C「パフォーマンスの最適化」](#)を参照してください。

SQL DML 文を実行すると、その文に対応付けられたカーソルがカーソル・キャッシュ内のエントリにリンクされます。続いてカーソル・キャッシュ・エントリは Oracle プライベート SQL 領域にリンクされます。この領域には SQL 文の実行に必要な情報が格納されます。

RELEASE_CURSOR はカーソル・キャッシュとプライベート SQL 領域の間のリンクで発生する処理を制御します。

RELEASE_CURSOR=YES に指定すると、Oracle が SQL 文を実行し、カーソルがクローズされた後、プリコンパイラによってこのリンクはただちに解除されます。これにより、プライベート SQL 領域に割り当てられたメモリーが解放され、解析ロックが解除されます。カーソルの CLOSE 時に関連リソースを確実に解放するには、RELEASE_CURSOR=YES を指定する必要があります。

RELEASE_CURSOR=NO の場合、リンクは保持されます。オープン・カーソルの数が MAXOPENCURSORS の設定値を超えないかぎり、プリコンパイラはリンクを再利用しません。この設定によって後に続く処理の実行速度が向上するため、これは実行頻度の高い SQL 文には便利です。文の再解析や Oracle プライベート SQL 領域用のメモリー割当ては不要です。

暗黙カーソル用としてインラインで使用するときは、SQL 文の実行前に RELEASE_CURSOR を設定してください。明示カーソル用としてインラインで使用するときは、カーソルを CLOSE する前に RELEASE_CURSOR を設定してください。

RELEASE_CURSOR=YES は HOLD_CURSOR=YES をオーバーライドすることに注意してください。これらの 2 つのオプションの相互作用を示す表は [付録 C「パフォーマンスの最適化」](#) を参照してください。

SELECT_ERROR

用途

SELECT 文が複数行を戻すとき、またはホスト配列の許容範囲を超える数の行を戻すときにプログラムがエラーを生成するかどうかを指定します。

構文

SELECT_ERROR={YES | NO}

デフォルト値

YES

使用上の注意

インラインまたはコマンドラインで入力できます。

SELECT_ERROR=YES のときは、行単位の SELECT 文が戻した行数が過大であったり、配列単位の SELECT 文が戻した行がホスト配列に入りきらなかったときにエラーが生成されます。SELECT 文の結果は未定義です。

SELECT_ERROR=NO のときは、行単位の SELECT 文が戻した行数が過大であっても、配列単位の SELECT 文が戻した行がホスト配列に入りきらなくてもエラーは生成されません。

YES を指定しても NO を指定しても、行は表から無作為に選択されます。選択する行の順序を特定するには、SELECT 文に ORDER BY 句を指定する以外に方法はありません。

SELECT_ERROR=NO のとき ORDER BY 句を指定すると、配列からの選択時に Oracle は先頭行または先頭の *n* 行を戻します。SELECT_ERROR=YES を指定すると、ORDER BY 句の有無を問わず、戻る行数が多すぎる場合にエラーが生成されます。

SQLCHECK

用途

構文検査および意味検査の種類と範囲を指定します。

構文

SQLCHECK={SEMANTICS | FULL | SYNTAX}

デフォルト値

SYNTAX

使用上の注意

SEMANTICS は FULL と同じです。

インラインまたはコマンドラインで入力できます。

完全な詳細は [付録 D「構文検査と意味検査」](#) を参照してください。

SYS_INCLUDE

用途

システム・ヘッダー・ファイルの位置を指定します。

構文

SYS_INCLUDE=*pathname* | (*path1*, ..., *pathn*)

デフォルト値

システム固有

使用上の注意

Pro*C/C++ は、プラットフォーム固有の標準的な位置で *stdio.h* などの標準のシステム・ヘッダー・ファイルを検索します。たとえば、ほとんどの UNIX システムでは *stdio.h* ファイルのフルパス名は */usr/include/stdio.h* です。

ただし、C++ コンパイラには、*stdio.h* などのシステム・ヘッダー・ファイルが標準的なシステム位置にないものがあります。SYS_INCLUDE コマンドライン・オプションを使用すると、Pro*C/C++ がシステム・ヘッダー・ファイルを検索するためのディレクトリ・パスのリストを指定できます。たとえば、次のとおりです。

```
SYS_INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)
```

SYS_INCLUDE を使用して指定した検索パスは、デフォルトのヘッダー位置より優先されます。

PARSE=NONE の場合、Pro*C/C++ はシステム・ヘッダー・ファイルをプリコンパイルに含める必要がないので、SYS_INCLUDE で指定した値は無視されます。（もちろん、それでも Oracle 特有のヘッダー *sqlca.h* やシステム・ヘッダー・ファイルなどは、コンパイラによるプリプロセッシングのために *#include* 宣言文を付けて、インクルードする必要があります。）

プリコンパイラは次の順序でディレクトリを検索します。

1. カレント・ディレクトリ
2. SYS_INCLUDE プリコンパイラ・オプションで指定されたシステム・ディレクトリ
3. INCLUDE オプションで指定されたディレクトリを入力順に
4. 標準ヘッダー・ファイル用の組込みディレクトリ

手順 3 があるので、通常は *sqlca.h* と *sqllda.h* などの標準ヘッダー・ファイルのディレクトリ・パスを指定する必要はありません。

THREADS

用途

THREADS=YES のとき、プリコンパイラはコンテキスト宣言を検索します。

構文

THREADS={YES | NO}

デフォルト値

NO

使用上の注意

インラインでは入力できません。

マルチスレッド・サポートを必要とするプログラムにはすべてこのプリコンパイラ・オプションを指定する必要があります。

THREADS=YES の場合、プリコンパイラは最初のコンテキストが表示され、実行 SQL 文が検出される前に EXEC SQL CONTEXT USE 宣言文が現れないときにエラーを生成します。詳細は、[第 11 章「マルチスレッド・アプリケーション」](#)を参照してください。

TYPE_CODE

用途

動的 SQL 方法 4 で ANSI または Oracle データ型を使用するかどうかをマイクロ・オプションで指定します。設定は MODE オプションの設定と同じです。

構文

TYPE_CODE={ORACLE | ANSI}

デフォルト値

ORACLE

使用上の注意

インラインでは入力できません。

可能なオプション設定は 14-11 ページの「[表 14-3](#)」を参照してください。

UNSAFE_NULL

用途

UNSAFE_NULL=YES を指定すると、標識変数を使用しないで NULL をフェッチしても ORA-01405 メッセージは生成されません。

構文

UNSAFE_NULL={YES | NO}

デフォルト値

NO

使用上の注意

インラインでは入力できません。

MODE=ORACLE のときにかぎり、UNSAFE_NULL=YES を指定できます。

埋込み PL/SQL ブロックのホスト変数では UNSAFE_NULL オプションは何の効果もありません。ORA-01405 エラーの発生を避けるためには、必ず標識変数を使用してください。

USERID

用途

Oracle ユーザー名およびパスワードを指定します。

構文

USERID=*username/password[@dbname]*

デフォルト値

なし

使用上の注意

入力できるのは、コマンドラインのみです。

自動接続機能を使用している場合は、このオプションを指定しないでください。自動接続機能では、先頭に OPS\$ の付いた Oracle ユーザー名しか受け付けません。"OPS\$" 文字列の実際の値は、INIT.ORA ファイルのパラメータとして設定されます。

SQLCHECK=SEMANTICS のとき、Oracle に接続してデータ・ディクショナリにアクセスしてプリコンパイラに必要な情報を取得させるには、USERID もあわせて指定する必要があります。

VARCHAR

用途

いくつかの構造体を VARCHAR ホスト変数として解釈するよう、Pro*C/C++ プリコンパイラに指示します。

構文

VARCHAR={NO | YES}

デフォルト値

NO

使用上の注意

入力できるのは、コマンドラインのみです。

VARCHAR=YES のとき、次のように C の構造体を記述します。

```
struct {  
    short len;  
    char  arr[n];  
} name;
```

すると、プリコンパイラによって VARCHAR[n] 型のホスト変数として解釈されます。

VARCHAR は NLS_CHAR オプションと一緒に使用して各国語変数を指定できます。

VERSION

用途

EXEC SQL OBJECT Deref 文によって、戻されるオブジェクトのバージョンを決めます。

構文

VERSION={RECENT | LATEST | ANY}

デフォルト値

RECENT

使用上の注意

EXEC ORACLE OPTION 文を使用してインライン入力できます。

RECENT は、カレント・トランザクション内でオブジェクトがオブジェクト・キャッシュに選択されている場合は、そのオブジェクトが戻されることを意味します。シリアル化可能モードで実行中のトランザクションの場合、このオプションの動作は LATEST と同じですが、ネットワーク往復回数はそれほど多くありません。ほとんどのアプリケーションは、RECENT が適切です。

LATEST は、オブジェクトがオブジェクト・キャッシュに存在しない場合は、データベースから取り出されることを意味します。オブジェクト・キャッシュに存在する場合は、サーバーからリフレッシュされます。LATEST の場合は、ネットワーク往復回数が最大になるため、慎重に使用してください。LATEST を使用するのには、オブジェクト・キャッシュとサーバー側のバッファ・キャッシュとを可能な限り一致させる必要がある場合のみです。

ANY は、オブジェクトがすでにオブジェクト・キャッシュに存在している場合は、そのオブジェクトが戻されることを意味します。常駐していなければ、そのオブジェクトはサーバーから取り出されます。ANY を指定すると、ネットワーク往復回数は最小になります。この値を使用するのは、アプリケーションが読取り専用オブジェクトにアクセスする場合や、ユーザーがオブジェクトに排他アクセスする場合です。

マルチスレッド・アプリケーション

使用中の開発プラットフォームでスレッドがサポートされない場合、この章は無視してください。

この章のトピックは次のとおりです。

- [スレッド](#)
- [Pro*C/C++ のランタイム・コンテキスト](#)
- [ランタイム・コンテキストの使用モデル](#)
- [マルチスレッド・アプリケーションのユーザー・インタフェース](#)
- [マルチスレッドの例](#)

スレッド

マルチスレッド・アプリケーションでは、共有のアドレス・スペースで複数のスレッドが実行されます。スレッドはプロセス内で実行される軽量のサブプロセスです。コードとデータ・セグメントは共有しますが、独自のプログラム・カウンタ、マシン・レジスタおよびスタックがあります。グローバル変数と静的変数はすべてのスレッドに共通であり、通常、アプリケーション内の複数のスレッドからこれらの変数へのアクセスを管理するには、相互排他メカニズムが必要です。mutex は、データの整合性が保たれることを保証するための同期化メカニズムです。

mutex の詳細は、マルチスレッドに関するテキストを参照してください。マルチスレッド・アプリケーションの詳細は、スレッドのマニュアルを参照してください。

Pro*C/C++ プリコンパイラは、マルチスレッドの Oracle8i Server アプリケーションの開発を（マルチスレッド・アプリケーションをサポートするプラットフォームで）サポートします。サポートされている機能は次のとおりです。

- スレッドに対して安全なコードを生成するコマンドライン・オプション
- マルチスレッド処理をサポートする埋込み SQL 文および宣言文
- スレッドに対して安全な SQLLIB と、その他のクライアント側 Oracle8i ライブラリ

注意：プラットフォームが特定のスレッド・パッケージをサポートしている場合でもプラットフォーム固有の Oracle ドキュメンテーションを参照して、Oracle8i がそれをサポートしているかどうか判断してください。

次の各トピックでは、前述の機能を使用してマルチスレッド Pro*C/C++ アプリケーションを開発する方法を示します。

- マルチスレッド・アプリケーションのランタイム・コンテキスト
- ランタイム・コンテキストを使用する 2 つのモデル
- マルチスレッド・アプリケーションのユーザー・インタフェース
- Pro*C/C++ でマルチスレッド・アプリケーションを記述する場合の考慮事項
- Pro*C/C++ マルチスレッド・アプリケーションの例

Pro*C/C++ のランタイム・コンテキスト

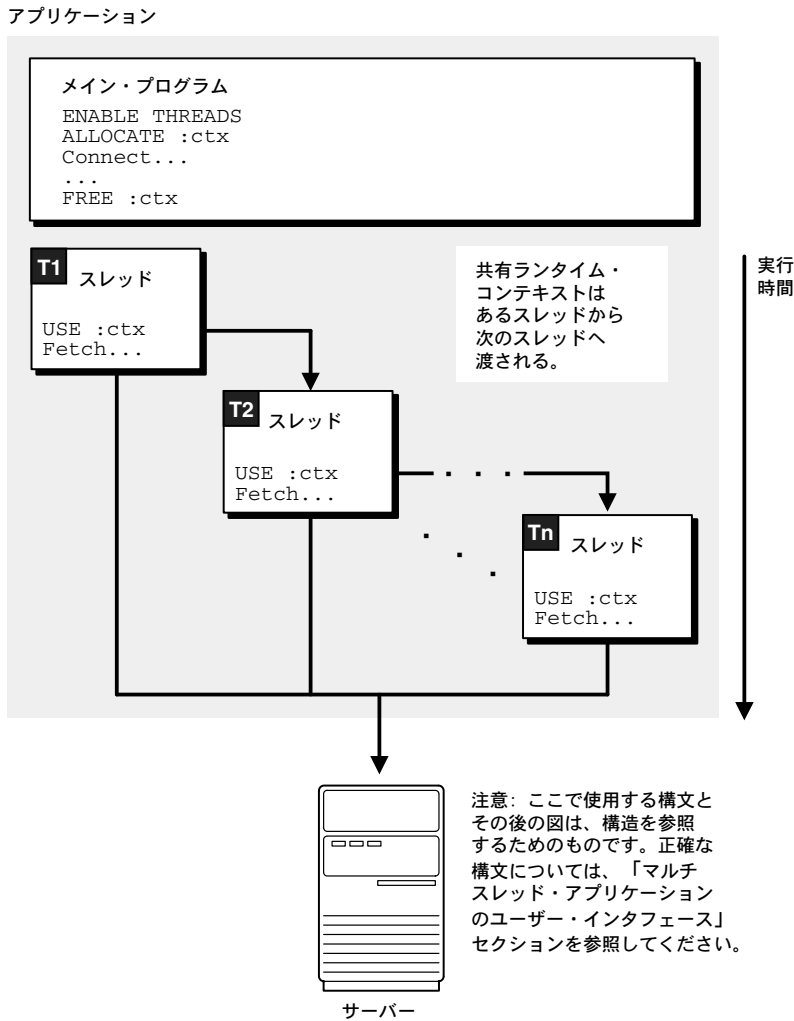
Pro*C/C++ には、スレッドと接続を疎結合するために、ランタイム・コンテキストという概念が導入されています。ランタイム・コンテキストには、次のリソースと現在の状態が含まれています。

- 1 つ以上の Oracle8i Server への 0 個以上の接続
- サーバーへの接続に使用される 0 個以上のカーソル
- MODE、HOLD_CURSOR、RELEASE_CURSOR、SELECT_ERROR などのインライン・オプション

Pro*C/C++ プリコンパイラを使用すると、スレッドと接続を疎結合せずに、スレッドをランタイム・コンテキストに疎結合できます。また、Pro*C/C++ では、アプリケーションでランタイム・コンテキストのハンドルを定義して、そのハンドルをあるスレッドから別のスレッドに渡すことができます。

たとえば、対話形式のアプリケーションで、スレッド T1 を作成し、問合せを実行して先頭の 10 行をアプリケーションに戻します。その後、T1 は終了します。必要なユーザー入力取得されると、別のスレッド T2 が作成され（または既存のスレッドが使用され）、T1 のランタイム・コンテキストが T2 に渡されます。T2 は同じカーソルを処理して次の 10 行をフェッチできます。[図 11-1 の「接続とスレッドの疎結合」](#)を参照してください。

図 11-1 接続とスレッドの疎結合



ランタイム・コンテキストの使用モデル

マルチスレッド Pro*C/C++ アプリケーションでランタイム・コンテキストを使用した 2 つの可能なモデルを次に示します。

- 単一のランタイム・コンテキストを共有する複数のスレッド
- 複数のランタイム・コンテキストを使用した複数のスレッド

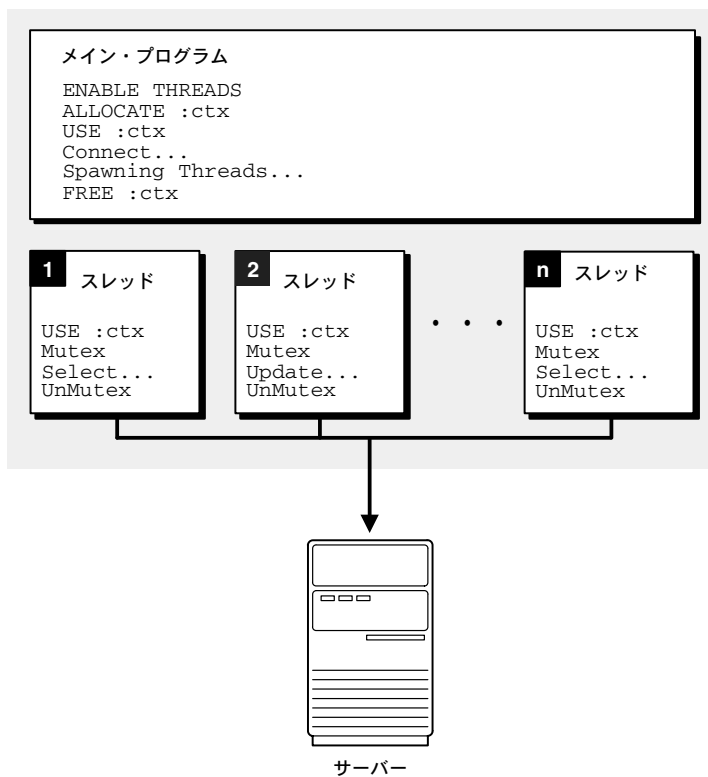
使用するランタイム・コンテキストのモデルにかかわらず、1 つのランタイム・コンテキストを複数のスレッドで同時に共有することはできません。複数のスレッドで同じランタイム・コンテキストを同時に使用すると、ランタイム・エラーが発生します。

単一のランタイム・コンテキストを共有する複数のスレッド

[図 11-2](#) にマルチスレッド環境で実行されるアプリケーションを示します。複数のスレッドが単一のランタイム・コンテキストを共有して 1 つまたは複数の SQL 文を処理します。なお、ランタイム・コンテキストを同時に複数のスレッドで共有することはできません。[図 11-2](#) の mutex は同時使用を防ぐ方法を示しています。

図 11-2 スレッド間のコンテキスト共有

アプリケーション

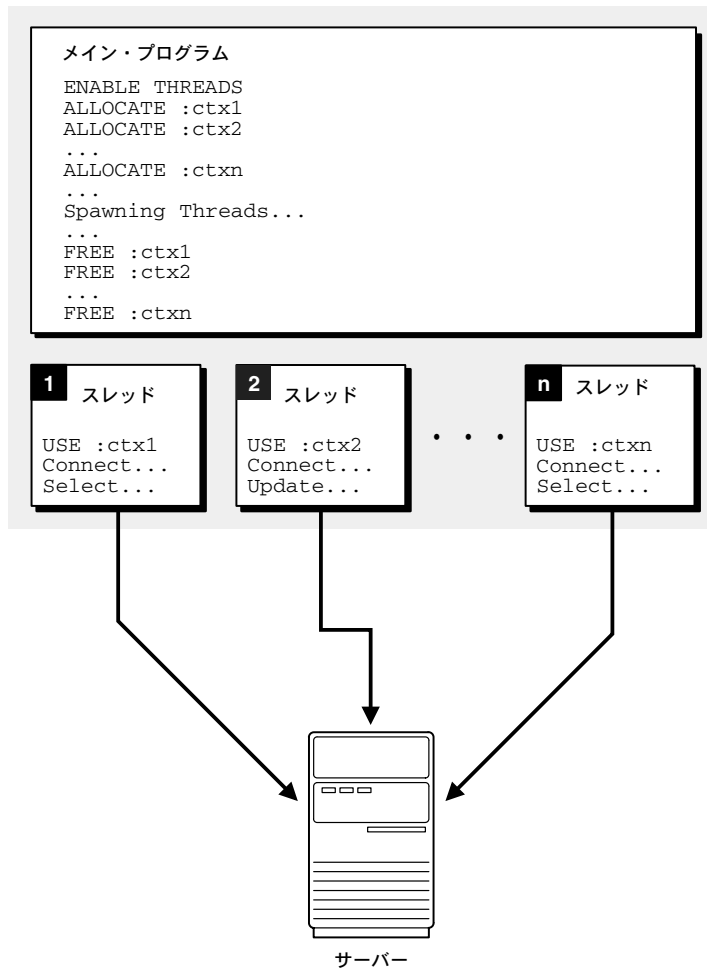


複数のランタイム・コンテキストを共有する複数のスレッド

図 11-3 に複数のランタイム・コンテキストを使用して複数スレッドを実行するアプリケーションを示します。この場合、各スレッドに専用のランタイム・コンテキストがあるため、アプリケーションに mutex は必要ありません。

図 11-3 スレッド間のコンテキスト非共有

アプリケーション



マルチスレッド・アプリケーションのユーザー・インタフェース

Pro*C/C++ プリコンパイラは、次に示すユーザー・インタフェース機能によってマルチスレッド・アプリケーションをサポートしています。

- コマンドライン・オプション THREADS=YES|NO
- 埋込み SQL 文と宣言文
- スレッドに対して安全な SQLLIB パブリック関数

THREADS オプション

THREADS=YES をコマンドラインに指定すると、11-12 ページの「[プログラミングの考慮事項](#)」で述べたガイドラインに従っていれば、Pro*C/C++ プリコンパイラは生成したコードがスレッドに対して安全なことを保証します。THREADS=YES と指定すると、Pro*C/C++ はすべての SQL 文がユーザー定義のランタイム・コンテキストの範囲内で実行されるかどうかを検証します。プログラムがこの要件を満たさない場合は、プリコンパイラ・エラーが戻されます。

埋込み SQL 文と宣言文

次の埋込み SQL 文および宣言文は、ランタイム・コンテキストおよびスレッドの定義と使用をサポートしています。

- EXEC SQL ENABLE THREADS;
- EXEC SQL CONTEXT ALLOCATE :context_var;
- EXEC SQL CONTEXT USE { :context_var | DEFAULT};
- EXEC SQL CONTEXT FREE :context_var;

これらの EXEC SQL 文では、*context_var* がランタイム・コンテキストのハンドルです。*context_var* は、次のように **sql_context** 型として宣言する必要があります。

```
sql_context <context_variable>;
```

DEFAULT を使用すると、別の CONTEXT USE 文でオーバーライドされるまで、字句的に続くすべての埋込み SQL 文でデフォルト（グローバル）ランタイム・コンテキストが使用されます。

EXEC SQL ENABLE THREADS

この実行 SQL 文は、複数のスレッドをサポートするプロセスを初期化します。この SQL 文は、マルチスレッド・アプリケーション内の最初の実行 SQL 文にしてください。詳細は、F-49 ページの「[ENABLE THREADS（実行可能埋込み SQL 拡張要素）](#)」を参照してください。

EXEC SQL CONTEXT ALLOCATE

この実行 SQL 文は指定されたランタイム・コンテキストにメモリーを割り当てて初期化します。ランタイム・コンテキスト変数は `sql_context` 型として宣言する必要があります。詳細は、F-28 ページの「[CONTEXT ALLOCATE \(実行可能埋込み SQL 拡張要素\)](#)」を参照してください。

EXEC SQL CONTEXT USE

この宣言文はプリコンパイラに、後に続く実行 SQL 文で指定したランタイム・コンテキストを利用するように指示します。指定するランタイム・コンテキストを、EXEC SQL CONTEXT ALLOCATE 文を使用して事前に割り当てする必要があります。

EXEC SQL CONTEXT USE 宣言文は、EXEC SQL WHENEVER 宣言文と同様に、任意のソース・ファイル内でこの宣言文に後続するすべての実行 SQL 文に作用し、C の標準の有効範囲規則には従いません。次の例では、`function2()` の UPDATE 文はグローバルなランタイム・コンテキスト `ctx1` を使用しています。

```
sql_context ctx1;                /* declare global context ctx1    */

function1()
{
    sql_context :ctx1;           /* declare local context ctx1    */
    EXEC SQL CONTEXT ALLOCATE :ctx1;
    EXEC SQL CONTEXT USE :ctx1;
    EXEC SQL INSERT INTO ... /* local ctx1 used for this stmt */
    ...
}

function2()
{
    EXEC SQL UPDATE ... /* global ctx1 used for this stmt */
}
```

ローカル・コンテキストを使用した後でグローバル・コンテキストを使用するには、次のコードを `function1()` に追加します。

```
function1()
{
    sql_context :ctx1;           /* declare local context ctx1    */
    EXEC SQL CONTEXT ALLOCATE :ctx1;
    EXEC SQL CONTEXT USE :ctx1;
    EXEC SQL INSERT INTO ... /* local ctx1 used for this stmt */
    EXEC SQL CONTEXT USE DEFAULT;
    EXEC SQL INSERT INTO ... /* global ctx1 used for this stmt */
    ...
}
```

次の例に、グローバル・ランタイム・コンテキストはありません。プリコンパイラは、UPDATE 文に対して生成されたコードで実行時コンテキスト *ctx1* を参照します。しかし、*function2()* の有効範囲にコンテキスト変数がないため、コンパイル時にエラーが発生します。

```
function1()
{
    sql_context ctx1;          /* local context variable declared */
    EXEC SQL CONTEXT ALLOCATE :ctx1;
    EXEC SQL CONTEXT USE :ctx1;
    EXEC SQL INSERT INTO ...    /* ctx1 used for this statement */
    ...
}
function2()
{
    EXEC SQL UPDATE ...        /* Error! No context variable in scope */
}
```

詳細は、F-32 ページの「[CONTEXT USE \(Oracle 埋込み SQL 宣言文\)](#)」と F-28 ページの「[CONTEXT ALLOCATE \(実行可能埋込み SQL 拡張要素\)](#)」を参照してください。

EXEC SQL CONTEXT FREE

この実行 SQL 文は、指定したランタイム・コンテキストに関連付けられたメモリーを解放し、ホスト・プログラム変数に NULL ポインタを入れます。詳細は、F-29 ページの「[CONTEXT FREE \(実行可能埋込み SQL 拡張要素\)](#)」を参照してください。

CONTEXT USE の例

次に示すコード部分では、2つの典型的なプログラミング・モデルに埋込み SQL 文とプリコンパイラ宣言文を使用する方法を示します。*thread_create()* を使用してスレッドを作成します。

最初の例では、複数のスレッドが複数のランタイム・コンテキストを使用する場合を示します。

```
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 SQL CONTEXT USE :ctx;
    /* execute executable SQL statements on runtime context ctx1!!! */
    ...
}

void function2(sql_context ctx)
{
    EXEC SQL CONTEXT USE :ctx;
    /* execute executable SQL statements on runtime context ctx2!!! */
    ...
}

```

次の例では、共通のランタイム・コンテキストを共有する複数のスレッドを使用する方法を示します。*function1()* および *function2()* で実行される SQL 文は同時に実行される可能性があるため、すべての実行 EXEC SQL 文を *mutex* で囲むことによって、データ操作を逐次的すなわち安全に行うことが必要です。

```

main()
{
    sql_context ctx;                /* declare runtime context */
    EXEC SQL CONTEXT ALLOCATE :ctx;
    ...
    /* spawn thread, execute function1 (in the thread) passing ctx */
    thread_create(..., function1, ctx);
    /* spawn thread, execute function2 (in the thread) passing ctx */
    thread_create(..., function2, ctx);
    ...
}

void function1(sql_context ctx)
{
    EXEC SQL CONTEXT USE :ctx;
    /* Execute SQL statements on runtime context ctx.                */
    ...
}

void function2(sql_context ctx)
{
    EXEC SQL CONTEXT USE :ctx;

```

```
/* Execute SQL statements on runtime context ctx.          */
...
}
```

プログラミングの考慮事項

Oracle8i は SQLLIB コードがスレッドに対して安全なことを保証しますが、Pro*C/C++ のソース・コードがスレッドで正しく働くように設計するのは設計者の責任です。たとえば、静的変数とグローバル変数の利用については慎重に考慮してください。

また、マルチスレッドには次の設計上の判断が必要です。

- SQLCA をスレッドに対して安全な構造体として宣言します。通常は自動変数として、ランタイム・コンテキストごとに1つずつ宣言します。
- SQLDA をスレッドに対して安全な構造体として宣言します (SQLCA と同様)。通常は自動変数として、ランタイム・コンテキストごとに1つずつ宣言します。
- ホスト変数をスレッドに対して安全であるように宣言します。つまり、静的ホスト変数およびグローバルなホスト変数の使用方法を慎重に検討します。
- 複数のスレッドでランタイム・コンテキストを同時に使用するのを避けます。
- デフォルトのデータベース接続を使用するか、あるいは AT 句を使用して明示的に定義するかを判断します。

さらに、複数の実行可能な埋込み SQL 文 (EXEC SQL UPDATE など) をランタイム・コンテキストで同時に未解決にしないでください。

プリコンパイルしたアプリケーションに対する既存の要件も適用されます。たとえば、ある特定のカーソルへの参照はすべて同じソース・ファイル内で指定する必要があります。

マルチスレッドの例

次のプログラムは、マルチスレッドの埋込み SQL アプリケーションを作成する方法の1つです。このプログラムはスレッド数と同じ数のセッションを作成します。それぞれのスレッドは0個以上のトランザクションを実行します。これは「レコード」と呼ばれる一時的な構造体によって指定されます。

注意： このプログラムは Solaris を動作している Sun ワークステーション専用に開発されています。このプログラムでは、DCE または Solaris のスレッド・パッケージを使用できます。スレッド・パッケージの可用性は、プラットフォーム固有のマニュアルを参照してください。

```
/*
 * Name:          Thread_example1.pc
 *
 * Description: This program illustrates how to use threading in
 *              conjunction with precompilers. 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'.
* Requirements:
*      The program requires a table 'ACCOUNTS' to be in the schema
*      scott/tiger. The description of ACCOUNTS is:
* SQL> desc accounts
*      Name                               Null?    Type
* -----
* 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>
```

```
#define      _EXC_OS_      _EXC_UNIX
#define      _CMA_OS_      _CMA_UNIX
```

```
#ifndef 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++)
    {
        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++)
    {
        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++)
    {
        /*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);
        else
            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))
#else
    if (mutex_destroy(&mutex))
#endif
#endif
    {
        printf("Can't destroy mutex\n");
        exit(1);
    }
}

```

```

/*****
 * Function: do_transaction
 *
 * 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)))
    {
        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
 *
 * Description: This routine prints out the most recent error
 *
 *****/
void      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
 *
 * Description: Logs on to the database as USERNAME/PASSWORD
 *
 *****/
void      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");
}

```

```

/*****
 * 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))
#else
    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))
#endif
        printf("Can't unlock mutex\n");
}

```

C++ アプリケーション

この章では、Pro*C/C++ プリコンパイラを使用して C++ の埋込み SQL アプリケーションをプリコンパイルする方法と、Pro*C/C++ が C++ 互換コードを生成する仕組みについて説明します。

この章では、次の事項について説明します。

- C++ サポートの理解
- C++ のプリコンパイル
- サンプル・プログラム

C++ サポートの理解

Pro*C/C++ で C++ がどのようにサポートされているかを理解するには、Pro*C/C++ の基本機能を理解する必要があります。特に、Pro*C/C++ と Pro*C バージョン 1 との違いを認識する必要があります。

Pro*C/C++ の基本機能は、次のとおりです。

- C プリプロセッサの完全サポート。Pro*C/C++ プログラム内で **#define**、**#include**、**#ifdef** およびその他のプリプロセッサ宣言文を使用して、プリコンパイラ自体が処理する必要がある構文を取り扱うことができます。詳細は、4-2 ページの「[Oracle のデータ型](#)」を参照してください。
- C の固有の構造体をホスト変数として使用可能。構造体（または構造体へのポインタ）をホスト変数として関数に渡す機能や、ホスト構造体または構造体ポインタを戻す書込み関数などがあります。詳細は、4-44 ページの「[構造体ポインタ](#)」を参照してください。

C プリプロセッサの機能をサポートし、特殊な宣言文の外でホスト変数を宣言できるようにするために、Pro*C/C++ には完全な C 解析機能が組み込まれています。Pro*C/C++ 解析機能は C の解析機能であり、C++ コードは解析できません。

したがって、C++ をサポートするには、C 解析機能を完全または部分的に使用禁止にする必要があります。C 解析機能を使用禁止にするために、Pro*C/C++ プリコンパイラには、Pro*C/C++ がソース・コードに対して行う C 解析の範囲を制御できるコマンドライン・オプションが組み込まれています。これらのオプションの詳細は 2 ページの「[C++ のプリコンパイル](#)」を参照してください。

特殊なマクロ処理は不要

C++ を Pro*C/C++ とともに使用するのには、特殊な事前処理や、Pro*C/C++ 外部の特殊なマクロ・プロセッサは必要ありません。プリコンパイラの出力に対してマクロ・プロセッサを実行しなくても、C++ との互換性を実現できます。

Pro*C/C++ プリコンパイラの旧リリースのユーザーで、プリコンパイラの出力にマクロ・プロセッサを使用していた場合は、コードを変更しないで Pro*C/C++ を使用して C++ アプリケーションをプリコンパイルできます。

C++ のプリコンパイル

C++ に対応できるようにプリコンパイルを制御するには、次の 4 点を考慮する必要があります。

- プリコンパイラによるコード出力
- 解析機能
- 出力ファイル名の拡張子
- システムのヘッダー・ファイルの位置

コードの生成

プリコンパイラで生成されるコードの種類（C 互換コードまたは C++ 互換コード）を指定する必要があります。デフォルトでは、Pro*C/C++ によって C のコードが生成されます。C++ は、C の完全なスーパーセットではありません。生成されるコードを C++ のコンパイラでコンパイルするには、コードを多少変更する必要があります。

たとえば、プリコンパイラにより、アプリケーション・コードが出力されるのみでなく、ランタイム・ライブラリ SQLLIB に対するコールを挿入します。SQLLIB 内の関数は、C 関数です。特殊な C++ 版の SQLLIB はありません。このため、C++ コンパイラを使用して生成したコードをコンパイルするには、SQLLIB 内でコールした関数を Pro*C/C++ により C 関数として宣言する必要があります。

C の出力では、プリコンパイラは次のようなプロトタイプを生成します。

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

ただし、C++ 互換コードの場合には、プリコンパイラで次のようなコードを生成する必要があります。

```
extern "C" {
void sqlora(unsigned long *, void *);
};
```

Pro*C/C++ によって生成されるコードの種類は、CODE というプリコンパイラ・オプションを使用して制御します。このオプションの値は、CPP、KR_C および ANSI_C です。これらのオプション間の違いは、SQLLIB の関数 *sqlora* の宣言方法が CODE オプションの 3 つの値で異なることを考慮すると説明できます。

```
void sqlora( /*_ unsigned long *, void * _*/); /* K&R C */

void sqlora(unsigned long *, void *);          /* ANSI C */

extern "C" {                                   /* CPP */
void sqlora(unsigned long *, void *);
};
```

CODE=CPP を指定すると、プリコンパイラは次を行います。

- C++ 互換コードを生成します。
- 出力ファイルに、標準の ".c" 拡張子ではなく、".C" や ".cc" など、プラットフォーム固有のファイル拡張子（接尾辞）を付けます。（この設定は、CPP_SUFFIX オプションを使用してオーバーライドできます。）
- PARSE オプションの値をデフォルトの PARTIAL にします。また、PARSE=NONE も指定できます。PARSE=FULL を指定すると、プリコンパイル時にエラーが発生します。
- C++ 形式の // コメントをコード内で使用可能にします。CODE=CPP のときは、この形式のコメントを SQL 文および PL/SQL ブロックの中でも使用できます。

- Pro*C/C++ は `/**` で始まる SQL オプティマイザ・ヒントを認識できます。
- OTT (オブジェクト型トランスレータ) によって生成される C のヘッダー・ファイルは、宣言文の中に含める必要があります。

CODE オプションの値 KR_C と ANSI_C の詳細は、10-13 ページの「[CODE](#)」を参照してください。

コードの解析

使用しているコードへの Pro*C/C++ の C 解析機能の効果を制御する必要があります。この制御は PARSE プリコンパイラ・オプションを使用することによって可能になります。このオプションでは、プリコンパイラの C 解析機能がコードの取扱方法を制御できます。

PARSE オプションの値と効果を次に示します。

PARSE=NONE	値 NONE の効果は次のとおりです。 <ul style="list-style-type: none">■ C プリプロセッサ宣言文は、宣言文の中にある場合のみ解釈されます。■ ホスト変数はすべて、宣言文の中に宣言する必要があります。■ プリコンパイラ・リリース 1.x の動作。
PARSE=PARTIAL	値 PARTIAL の効果は次のとおりです。 <ul style="list-style-type: none">■ プリプロセッサ宣言文はすべて解釈されます。■ ホスト変数はすべて、宣言文の中に宣言する必要があります。 このオプション値は、CODE=CPP のときデフォルトです。
PARSE=FULL	値 FULL の効果は次のとおりです。 <ul style="list-style-type: none">■ プリコンパイラの C 解析機能が使用しているコードで稼働します。■ プリプロセッサ宣言文はすべて解釈されます。■ ホスト変数は、C で有効に宣言できる位置ならばどこにでも宣言できます。

このオプション値は、CODE オプションの値が CPP 以外のときのデフォルト値です。CODE=CPP のときに PARSE=FULL を指定すると、エラーになります。

C++ 互換コードを生成するには、PARSE オプションに NONE または PARTIAL のどちらかを指定する必要があります。PARSE=FULL のときは C 解析機能が稼働し、コードにある C++ クラスなどの構文を認識しません。

出力ファイル名の拡張子

ほとんどの C コンパイラでは、入力ファイルのデフォルト拡張子は ".c" とみなされます。しかし、C++ コンパイラでは、想定されるファイルの拡張子がコンパイラごとに異なる場合があります。CPP_SUFFIX オプションを使用すると、プリコンパイラが生成するファイルの拡張子を指定できます。このオプションの値は、引用符もピリオドも付けない文字列です。たとえば、CPP_SUFFIX=cc または CPP_SUFFIX=C のように指定します。

システム・ヘッダー・ファイル

Pro*C/C++ は、プラットフォーム固有の標準的な位置で *stdio.h* などの標準のシステム・ヘッダー・ファイルを検索します。Pro*C/C++ では、*hpp* または *h++* などの拡張子が付いたヘッダー・ファイルは検索されません。たとえば、ほとんどの UNIX システムでは、*stdio.h* ファイルのフル・パス名は */usr/include/stdio.h* です。

しかし、C++ コンパイラは独自のバージョンの *stdio.h* を持っており、これはシステムの標準位置にはありません。C++ でのプリコンパイル時は、Pro*C/C++ がシステム・ヘッダー・ファイルを検索するためのディレクトリ・パスを、SYS_INCLUDE プリコンパイラ・オプションを使用して指定する必要があります。たとえば、次のとおりです。

```
SYS_INCLUDE=(/usr/lang/SC2.0.1/include,/usr/lang/SC2.1.1/include)
```

システム・ヘッダー・ファイル以外の位置を指定するには、INCLUDE プリコンパイラ・オプションを使用します。10-24 ページの「[INCLUDE](#)」を参照してください。SYS_INCLUDE オプションで指定したディレクトリは、INCLUDE オプションで指定したディレクトリよりも前に検索されます。

PARSE=NONE のときは、Pro*C/C++ はシステム・ヘッダー・ファイルを組み込む必要がないので、システム・ファイルについて SYS_INCLUDE および INCLUDE で指定した値は無視されます。（ただし、当然ながら、*sqlca.h* などの Pro*C/C++ 固有のヘッダーは、EXEC SQL INCLUDE 文を使用して組み込むことができます。）

サンプル・プログラム

この項には、C++ 構文を含んでいるサンプルの Pro*C/C++ プログラムを 3 つ記載しています。これらのプログラムはそれぞれ *demo* ディレクトリにオンラインで利用可能な形で入っています。

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
#define     PWD_LEN       40

// Declare section is required when CODE=CPP and/or
// PARSE={PARTIAL|NONE}
EXEC SQL BEGIN DECLARE SECTION;
    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 SQL END DECLARE SECTION;

// Define a C++ class object to match the desired
// struct from the above 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&);
};

emp::emp(empdat& dat, empind& ind)
{
    strcpy(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 *) 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: "

```

```

        << (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
    FROM EMP // Comments in SQL statements.
    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."
        << endl << endl;
} // end while(1)

cout << endl << "Total rows returned was "
    << total_queried << endl;
cout << "Have a nice day!" << endl << endl;

// 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);
}

```

cppdemo2.pc

次のアプリケーションは、簡単なモジュラーの例です。最初に SQL*Plus の次の SQL スクリプト cppdemo2.sql を実行します。

```

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
/

```

ヘッダー・ファイル empclass.h では、クラス emp が定義されます。

```

// 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:
    emp();    // Constructor: ALLOCATE Cursor Variable
    ~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 via 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;
};

```

empclass.pc のコードには、emp メソッドが含まれています。

```

#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
        BEGIN
            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);
}

```

メイン・プログラム cppdemo2.pc では、カーソル変数が使用されます。

```

// 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.

    delete e;           // Invoke Destructor - FREE Cursor Variable.
```

```

EXEC SQL ROLLBACK WORK RELEASE;
return (0);
}

```

cppdemo3.pc

```

/*
 * cppdemo3.pc : An example of C++ Inheritance
 *
 * 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&);
};

// 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;
}

void print(char *ename, int sal, int comm)
{
    salesman man(ename, sal, comm);
    cout << man;
}

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);
    }
    EXEC SQL CLOSE c;
    exit(0);

error:
    cout << endl << sqlca.sqlerrm.sqlerrmc << endl;
    exit(1);
}
```

Oracle の動的 SQL

この章では、アプリケーションに柔軟性と機能性を持たせる高度なプログラミング技法である Oracle 動的 SQL の使用方法について説明します。動的 SQL の長所と短所を検討した後で、ランタイムに作動される SQL 文を受け入れ、処理するプログラムの作成方法を単純なものから複雑なものまで 4 通り紹介します。それぞれの方法の必要条件および制限事項、さらに実行するジョブに対する適切な方法の選択方法についても説明します。

注意： Oracle 動的 SQL は、オブジェクト型、カーソル変数、構造体の配列、DML 戻り句、Unicode 変数、および LOB をサポートしていません。かわりに ANSI 動的 SQL 方法 4 を使用してください。第 14 章「ANSI 動的 SQL」を参照してください。

この章のトピックは、次のとおりです。

- 動的 SQL
- 動的 SQL の長所と短所
- 動的 SQL の使用
- 動的 SQL 文の要件
- 動的 SQL 文の処理方法
- 動的 SQL の使用方法
- 方法 1 の使用方法
- 方法 2 の使用方法
- 方法 3 の使用方法
- 方法 4 の使用方法
- DECLARE STATEMENT 文の使用方法
- PL/SQL の使用方法

動的 SQL

ほとんどのデータベース・アプリケーションでは、ある特定のジョブが実行されます。たとえば、ユーザーに従業員番号の入力を要求して、その後 EMP および DEPT という表の行を更新するという単純なプログラムがあります。この場合は、プリコンパイル時に UPDATE 文の構成がわかっています。つまり、変更する表、それぞれの表および列に定義されている制約、更新する列、それぞれの列のデータ型がわかっています。

しかし一部のアプリケーションでは、様々な SQL 文を実行時に受け入れ（または作成し）、処理する必要があります。たとえば汎用レポート・ライターでは、生成するレポートについてそれぞれ別の SELECT 文を作成する必要があります。この場合は、文の構成はランタイムまではわかりません。このような文は実行のたびに異なる可能性があります。このような文を動的 SQL 文と呼びます。

静的 SQL 文とは異なり、動的 SQL 文はソース・プログラム内には埋め込まれません。そのかわり、これらの文は実行時にプログラムに入力される（または、プログラムによって作成される）文字列に格納されます。動的 SQL 文は対話形式で入力できるだけでなく、ファイルから読み込むこともできます。

動的 SQL の長所と短所

通常の埋込み SQL プログラムと比べると、動的に定義された SQL 文を受け入れて処理するホスト・プログラムの方が柔軟性は高くなります。動的 SQL 文は、SQL の知識がほとんどないユーザーでも対話形式で作成できます。

たとえば、SELECT 文、UPDATE 文または DELETE 文の WHERE 句内で使用する検索条件の入力をユーザーに求めるという単純なプログラムがあります。さらにプログラムが複雑になると、ユーザーは SQL 処理、表およびビューの名前、列の名前などが表示されているメニューから選択できます。このように、動的 SQL を使用すると柔軟性に富んだアプリケーションを記述できます。

ただし、動的問合せの中には複雑なコーディング、特殊なデータ構造体の使用、実行時の処理時間が必要になるものもあります。処理時間が増えることは気にならない場合もありますが、動的 SQL の概念および技法を完全に理解するまで、コーディングは難しく感じられるかもしれません。

動的 SQL の使用

実際は静的 SQL によって、プログラミング要件のほとんどを満たすことができます。動的 SQL は、その高度な柔軟性が必要とされる場合にのみ使用してください。プリコンパイル時に次の項目のいずれかが不明の場合は、動的 SQL の使用が適しています。

- SQL 文のテキスト（コマンド、句など）
- ホスト変数の数
- ホスト変数のデータ型
- データベース・オブジェクトの参照（列、索引、順序、表、ユーザー名、ビューなど）

動的 SQL 文の要件

動的 SQL 文を記述するには、文字列に有効な SQL 文を示すテキストが格納される必要がありますが、このとき EXEC SQL 句、文終了記号または次に示す埋込み SQL コマンドを含まないでください。

- ALLOCATE
- CLOSE
- DECLARE
- DESCRIBE
- EXECUTE
- FETCH
- FREE
- GET
- INCLUDE
- OPEN
- PREPARE
- SET
- WHENEVER

ほとんどの場合、この文字列にはダミーのホスト変数を含むことができます。これらは SQL 文内に実際のホスト変数のための場所を確保します。ダミーのホスト変数はプレースホルダにすぎないため宣言する必要はなく、しかも任意の名前を指定できます。たとえば、Oracle では次の 2 つの文字列は区別されません。

```
'DELETE FROM EMP WHERE MGR = :mgr_number AND JOB = :job_title'  
'DELETE FROM EMP WHERE MGR = :m AND JOB = :j'
```

動的 SQL 文の処理方法

一般にアプリケーション・プログラムでは、SQL 文のテキストおよびその文で使用するホスト変数の値をユーザーが入力する必要があります。SQL 文が入力されると、Oracle によって解析されます。つまりこの SQL 文が構文規則に従うとともに、有効なデータベース・オブジェクトを参照していることを Oracle は確認します。解析では、データベース・アクセス権限のチェック、必要なリソースの確保および最適なアクセス・パスの検索も行われます。

次にこのホスト変数は Oracle によって SQL 文にバインドされます。つまり Oracle がホスト変数のアドレスを取得するので、値の読込みと書込みができます。

その後、この SQL 文が実行されます。つまり Oracle はこの SQL 文が要求する処理（表からの行の削除など）を実行します。

これらのホスト変数に別の値を指定することによって、この SQL 文を繰り返し実行できます。

動的 SQL の使用方法

この項では、動的 SQL 文の定義に使用できる 4 つの方法を紹介します。まずそれぞれの方法の機能と制限事項を簡単に説明した後、適切な方法を選択するためのガイドラインを示します。この後の項でこれらの方法の使用方法を説明します。また学習用にサンプル・プログラムを示します。

この 4 つの方法は番号が大きくなるに従って対象が広がります。つまり方法 2 は方法 1 を包含し、方法 3 は方法 1 と方法 2 を包含します。ただし、[表 13-1](#) で示すように、それぞれの方法は特定の種類の SQL 文を処理するのに適しています。

表 13-1 動的 SQL の使用方法

方法	SQL 文の種類
1	ホスト変数のない非問合せ
2	入力ホスト変数の数がわかっている非問合せ
3	選択リスト項目の数と入力ホスト変数の数がわかっている問合せ
4	選択リスト項目の数または入力ホスト変数の数が不明の問合せ

注意： 選択リスト項目には、SAL * 1.10 および MAX (SAL) などの列名と式が含まれます。

方法 1

この方法を使用すると、プログラムは動的 SQL 文を受け入れ（または作成し）、EXECUTE IMMEDIATE コマンドを使用してその文をすぐに実行します。この SQL 文を問合せ（SELECT 文）にしないでください。またこの SQL 文中には入力ホスト変数のプレースホルダを指定できません。たとえば次のホスト文字列は有効です。

```
'DELETE FROM EMP WHERE DEPTNO = 20'
'GRANT SELECT ON EMP TO scott'
```

方法 1 では、SQL 文は実行のたびに解析されます。

方法 2

この方法を使用すると、プログラムは動的 SQL 文を受け入れ（または作成し）、PREPARE および EXECUTE コマンドを使用してその文を処理します。SQL 文を問合せにしないでください。プリコンパイル時に 入力ホスト変数のプレースホルダの数および入力ホスト変数のデータ型を明確にする必要があります。たとえば次のホスト文字列はこのカテゴリに該当します。

```
'INSERT INTO EMP (ENAME, JOB) VALUES (:emp_name, :job_title)'  
'DELETE FROM EMP WHERE EMPNO = :emp_number'
```

方法 2 では SQL 文の解析は 1 度しか行われませんが、ホスト変数に異なる値を指定して、この SQL 文を複数回実行できます。SQL データ定義文（CREATE や GRANT など）は、PREPARE の際に実行されます。

方法 3

この方法を使用すると、プログラムは動的問合せを受け入れ（または作成し）、DECLARE、OPEN、FETCH および CLOSE カーソル・コマンドとともに PREPARE コマンドを使用してその問合せを 処理します。プリコンパイル時に選択リスト項目の数、入力ホスト変数のプレースホルダの数および入力ホスト変数のデータ型を明確にする必要があります。たとえば次のホスト文字列は有効です。

```
'SELECT DEPTNO, MIN(SAL), MAX(SAL) FROM EMP GROUP BY DEPTNO'  
'SELECT ENAME, EMPNO FROM EMP WHERE DEPTNO = :dept_number'
```

方法 4

この方法を使用すると、プログラムは動的 SQL 文を受け入れ（または作成し）、記述子を使用して処理します（13-23 ページの「[方法 4 の使用方法](#)」で説明されています）。選択リスト項目の数、入力ホスト変数のプレースホルダの数および入力ホスト変数のデータ型は実行時まで不明にできます。たとえば次のホスト文字列はこのカテゴリに該当します。

```
'INSERT INTO EMP (<unknown>) VALUES (<unknown>)'  
'SELECT <unknown> FROM EMP WHERE DEPTNO = 20'
```

方法 4 は、選択リスト項目の数または入力ホスト変数の数が不明の動的 SQL 文を実行するときに必要です。

ガイドライン

4つの方法はいずれも、動的 SQL 文を文字列に格納する必要があります。このとき指定する文字列は、ホスト変数または引用符で囲んだりテラルにする必要があります。SQL 文を文字列に格納する際に、キーワード EXEC SQL と ';' の文の終了記号を省略します。

方法2および方法3のときは、入力ホスト変数のプレースホルダの数と入力ホスト変数のデータ型をプリコンパイル時には明確にしておいてください。

方法の番号が大きくなるほどアプリケーションへの制約は少なくなりますが、コードの記述は難しくなります。原則として、できるだけ簡単な方法を使用してください。ただし動的 SQL 文を方法1で繰り返し実行する場合は、実行のたびにその文が再解析されるのを避けるために方法2を使用します。

方法4は最も柔軟性に富んでいますが、複雑なコード記述方法および動的 SQL の概念の完全な理解が求められます。一般には方法1、2、3を使用できないときにのみ方法4を使用します。

図 13-1 の決定論理を基に、適切な方法を選ぶことができます。

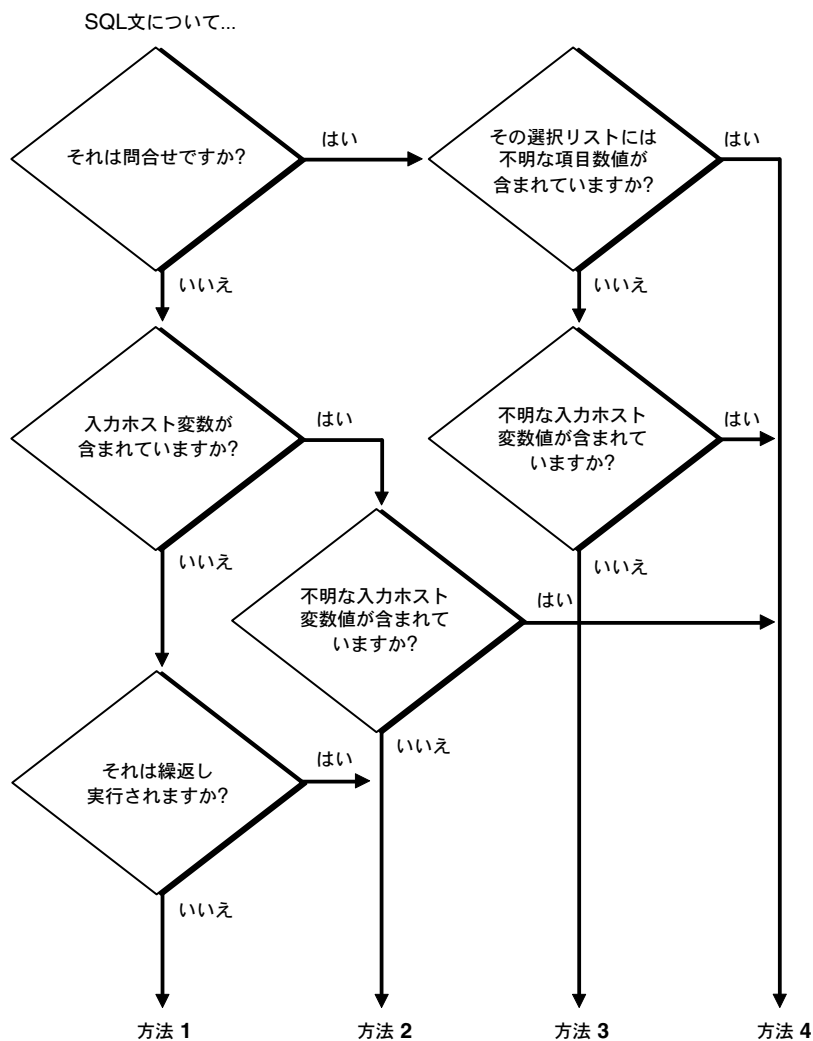
一般的なエラーの回避

コマンドライン・オプション DBMS=V6_CHAR と指定してプリコンパイルするときは、SQL 文を格納する前に配列を空白で埋めてください。こうして余分な文字を消去します。別の SQL 文を格納するために配列を再利用するときにこの処理が特に重要となります。原則として、SQL 文を格納する前に必ずホスト文字列を初期化（または再初期化）してください。ホスト文字列は NULL で終了しないでください。Oracle は NULL 終了記号を文字列の終了と見なしていません。Oracle は NULL を SQL 文の一部と見なします。

コマンドライン・オプション DBMS=V8 と指定してプリコンパイルするときは、PREPARE 文または EXECUTE IMMEDIATE 文を実行する前に、文字列が NULL で終了していることを確認してください。

DBMS の値が何であっても、VARCHAR 変数を使用して動的 SQL 文を格納するときは、PREPARE 文または EXECUTE IMMEDIATE 文を実行する前に、VARCHAR の長さが正しく設定（または再設定）されていることを確認してください。

図 13-1 適切な方法の選択



方法 1 の使用方法

最も単純な動的 SQL 文では、結果が「成功」か「失敗」のどちらかで、ホスト変数は使用されません。次にいくつかの例を示します。

```
'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'
```

方法 1 では、SQL 文を解析すると、EXECUTE IMMEDIATE コマンドを使用してその文をすぐに実行します。コマンドには実行用の SQL 文を含む文字列（ホスト変数またはリテラル）が続きます。この文を問合せにしないでください。

EXECUTE IMMEDIATE 文の構文は次のとおりです。

```
EXEC SQL EXECUTE IMMEDIATE { :host_string | string_literal };
```

次の例では、ホスト変数 *dyn_stmt* を使用して、ユーザーの入力する SQL 文を格納します。

```
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;
}
...
```

次の例で示すように、文字列リテラルを使用することも可能です。

```
EXEC SQL EXECUTE IMMEDIATE 'REVOKE RESOURCE FROM MILLER';
```

EXECUTE IMMEDIATE は入力されている SQL 文を実行するたびに解析するため、方法 1 は 1 回しか実行しない文に最も適しています。一般に、データ定義言語がこのカテゴリに該当します。

サンプル・プログラム : 動的 SQL 方法 1

次のプログラムでは動的 SQL 方法 1 を使用して、表の作成、行の挿入、挿入のコミット、表の削除を実行します。このプログラムは demo ディレクトリの *sample6.pc* ファイルにあるので、オンラインで利用できます。

```
/*
 * 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
 * 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";
    char    *password = "TIGER";
    char    *dynstmt1;
    char    dynstmt2[10];
    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
 * 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 SQL 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.orasfnn.orasfnnl,
        oraca.orasfnn.orasfnnmc);

/* 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);
}
```

方法 2 の使用方法

方法 1 では 1 段階で実行することを、方法 2 では 2 段階に分けて実行します。動的 SQL 文（問合せは不可）は、まず PREPARE（名前の指定と解析）され、次に EXECUTE されます。

方法 2 では、SQL 文には入力ホスト変数と標識変数のプレースホルダを指定できます。この SQL 文は 1 度 PREPARE すれば、ホスト変数に別の値を指定して繰り返し EXECUTE できます。したがって、（ログアウトして再接続しないかぎり）COMMIT または ROLLBACK の後に再度 SQL 文を PREPARE する必要はありません。

方法 4 では、非問合せに EXECUTE を使用できるので注意してください。

PREPARE 文の構文は次のとおりです。

```
EXEC SQL PREPARE statement_name
      FROM { :host_string | string_literal };
```

PREPARE はこの SQL 文を解析して名前を指定します。

statement_name は、ホスト変数やプログラム変数ではなく、プリコンパイラにより使用される識別子であり、そのため宣言文で宣言されません。これは EXECUTE の対象として PREPARE した文を示しているにすぎません。

EXECUTE 文の構文は次のとおりです。

```
EXEC SQL EXECUTE statement_name [USING host_variable_list];
```

host_variable_list は次の構文に従います。

```
:host_variable1[:indicator1] [, host_variable2[:indicator2], ...]
```

解析した SQL 文は、それぞれの入力ホスト変数に指定済の値を使用して EXECUTE によって実行されます。

次の例では、入力された SQL 文にプレースホルダ *n* が含まれています。

```
...
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;
}
...
```

方法 2 では、プリコンパイル時に入力ホスト変数のデータ型を明確にする必要があります。最後の例では、*emp_number* が **int** として宣言されています。Oracle では **float** や **char** などの全データ型の Oracle 内部 NUMBER データ型への変換がサポートされるため、*emp_number* が float または char として宣言することも可能です。

USING 句

SQL 文が EXECUTE されると、USING 句の入力ホスト変数は、PREPARE された動的 SQL 文内の該当するプレースホルダに置換されます。

PREPARE された動的 SQL 文のプレースホルダは、それぞれが必ず USING 句の個別のホスト変数に対応している必要があります。つまり、PREPARE された文に同じプレースホルダが 2 回以上現れるときは、それぞれが USING 句のホスト変数に対応している必要があります。

プレースホルダの名前はホスト変数名と一致する必要はありません。ただし PREPARE された動的 SQL 文のプレースホルダの順序は、USING 句の対応するホスト変数の順序と一致する必要があります。

USING 句のホスト変数のうち 1 つでも配列があれば、すべてのホスト変数が配列である必要があります。

NULL 値を指定するために、標識変数を USING 句のホスト変数と関連付けることができます。詳細は、6-3 ページの「[標識変数の使用](#)」を参照してください。

サンプル・プログラム：動的 SQL 方法 2

次のプログラムでは、動的 SQL 方法 2 を使用して 2 つの行を EMP 表へ挿入し、その後それらの行を削除しています。このプログラムは demo ディレクトリのファイル *sample7.pc* があるので、オンラインで利用できます。

```
/*
 * 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 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 use of
 * the ORACA.
 */
EXEC ORACLE OPTION (ORACA=YES);

char    *username = USERNAME;
char    *password = PASSWORD;
VARCHAR dynstmt[80];
int      empno    = 1234;
int      deptno1  = 97;
int      deptno2  = 99;

/* Handle SQL runtime errors. */
void dyn_error();
```

```
main()
{
/* 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);

/* Display the SQL statement and its current input host
 * variables.
 */
    puts((char *) dynstmt.arr);
    printf("    v1 = %d, v2 = %d\n", empno, deptno1);

/* 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.

 * 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
* 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
* contains no placeholders.
*
* 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);
    printf("on line %d of %.*s.\n\n",
        oraca.oraslnr, oraca.orasfnn.orasfnnl,
        oraca.orasfnn.orasfnnmc);

/* Disable ORACLE error checking to avoid an infinite loop
 * should another error occur within this routine.
 */
    EXEC SQL WHENEVER SQLERROR CONTINUE;

/* Roll back any pending changes and
 * disconnect from Oracle.
 */
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}

```

方法 3 の使用方法

方法 3 は方法 2 に似ていますが、PREPARE 文をカーソルの定義および処理に必要な文と結合する点で異なります。これによって、プログラムで問合せを受け入れて処理できます。実際、動的 SQL 文が問合せの場合は、方法 3 または 4 を使用する必要があります。

方法 3 では、プリコンパイル時に問合せ選択リストの列数と入力ホスト変数のプレースホルダの数を明確にする必要があります。ただし、表および列などのデータベース・オブジェクトの名前は実行時まで指定する必要はありません。データベース・オブジェクトの名前はホ

スト変数に指定できません。問合せ結果を限定、分類、ソートする句（WHERE、GROUP BY、ORDER BY など）も実行時に指定できます。

方法 3 では、埋込み SQL 文を次のような順序で使用します。

```
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;
```

各文の機能を次に説明します。

PREPARE

PREPARE はこの動的 SQL 文を解析し、名前を指定します。次の例では、PREPARE は文字列 *select_stmt* 内の問合せを解析し、これに *sql_stmt* という名前を指定します。

```
char select_stmt[132] =
    "SELECT MGR, JOB FROM EMP WHERE SAL < :salary";
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
```

一般的には、この問合せの WHERE 句は実行時に端末から入力するか、またはアプリケーションによって生成されます。

識別子 *sql_stmt* はホスト変数でもプログラム変数でもありませんが、一意にする必要があります。*sql_stmt* は特定の動的 SQL 文を指定します。

次の文も正しい文です。

```
EXEC SQL PREPARE sql_stmt FROM SELECT MGR, JOB FROM EMP WHERE SAL < :salary;
```

'%' ワイルドカードを使用する次の PREPARE 文も正しい文です。

```
EXEC SQL PREPARE S FROM select ename FROM test WHERE ename LIKE 'SMIT%';
```

DECLARE

DECLARE は、カーソルに名前を指定するとともにこれを特定の問合せに関連付けてカーソルを定義します。前述の例に続いて、DECLARE は次のように *emp_cursor* という名前のカーソルを定義してから、これを *sql_stmt* に関連付けます。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

識別子 *sql_stmt* および *emp_cursor* はホスト変数でもプログラム変数でもありませんが、一意にする必要があります。同じ文名で 2 つのカーソルを宣言すると、プリコンパイラは 2 つのカーソル名をシノニムと見なします。

たとえば次の文を実行します。

```
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;
```

emp_cursor を OPEN すると、*select_stmt* に格納されている動的 SQL 文ではなく、*delete_stmt* に格納されている動的 SQL 文が処理されます。

OPEN

OPEN はアクティブ・セットを識別して、Oracle カーソルを割り当て、入力ホスト変数をバインドし、問合せを実行します。OPEN はさらに、アクティブ・セットの最初の行にカーソルを位置付け、SQLCA 内の *sqlerrd* の 3 番目の要素に保存される処理済の行数を 0（ゼロ）に設定します。USING 句内の入力ホスト変数は、PREPARE された動的 SQL 文内の対応するプレースホルダに置換されます。

前述の例に続いて、OPEN は次に示すように *emp_cursor* を割り当て、ホスト変数 *salary* を WHERE 句に割り当てます。

```
EXEC SQL OPEN emp_cursor USING :salary;
```

FETCH

FETCH はアクティブ・セットから行を戻し、選択リスト内の列の値を INTO 句内の対応するホスト変数に割り当ててから、カーソルを次の行に進めます。他に行がない場合は、FETCH により "データが見つかりません" という Oracle エラー・コードが *sqlca.sqlcode* に戻されます。

次の例では、FETCH はアクティブ・セットから 1 行を戻して、MGR および JOB の列の値をホスト変数の *mgr_number* および *job_title* に割り当てます。

```
EXEC SQL FETCH emp_cursor INTO :mgr_number, :job_title;
```

CLOSE

CLOSE はカーソルを使用禁止にします。一度カーソルをクローズすると、それ以降は FETCH できません。

例では、次のように CLOSE により *emp_cursor* が使用禁止になります。

```
EXEC SQL CLOSE emp_cursor;
```

サンプル・プログラム：動的 SQL 方法 3

次のプログラムは、動的 SQL 方法 3 を使用して EMP 表から指定された部門のすべての従業員の名前を検索します。このプログラムは demo ディレクトリのファイル *sample8.pc* があるので、オンラインで利用できます。

```
/*
 * 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
 * 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];
int      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,
 * 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 can not 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
 * contains no placeholders. OPEN places the cursor at the
 * 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
 * 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.orasfnc.orasfnc[oraca.orasfnc.orasfncml] = '\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.orasfnc.orasfnc);

/* 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);
}

```

方法 4 の使用方法

この項では、動的 SQL 方法 4 の概要を説明します。Oracle 動的 SQL 方法 4 は、オブジェクト型、結果セット、構造体の配列、および LOB をサポートしていません。詳細は、[第 15 章「Oracle の動的 SQL 方法 4」](#)を参照してください。

ANSI SQL では、すべてのデータ型がサポートされます。すべての新しいアプリケーションでは、ANSI SQL を使用してください。ANSI 動的 SQL 方法 4 の詳細は、14-1 ページの[「ANSI 動的 SQL」](#)を参照してください。

動的 SQL 文には、方法 3 を使用してもプログラムで処理できないものがあります。選択リスト項目の数または入力ホスト変数のプレースホルダの数が実行時まで不明な場合は、プログラムで記述子を使用する必要があります。記述子とはプログラムおよび Oracle が動的 SQL 文内の変数の完全な記述を保存するためのメモリー領域です。

複数行の問合せのときに、宣言済の出力ホスト変数のリスト内に選択した列の値を FETCH INTO したことを思い出してください。この選択リストが不明な場合は、プリコンパイル時に INTO 句でホスト変数リストを作成できません。たとえば次の問合せでは、2 つの列値が戻されます。

```
SELECT ename, empno FROM emp WHERE deptno = :dept_number;
```

ただし、この選択リストをユーザーに定義させる場合、その問合せによって戻される列の数は不明です。

SQLDA の必要性

このような種類の動的問合せを処理するには、プログラムで DESCRIBE SELECT LIST コマンドを発行するとともに、SQL 記述子領域 (SQLDA) というデータ構造体を宣言する必要があります。この構造体は問合せ選択リストの列の記述を保持しているため、選択記述子とも呼ばれます。

また、動的 SQL 文で入力ホスト変数のプレースホルダの数が明確でない場合、プリコンパイル時に USING 句でホスト変数リストを作成できません。

動的 SQL 文を処理するには、プログラムで DESCRIBE BIND VARIABLES コマンドを発行し、バインド記述子と呼ばれる別の SQLDA を宣言することによって、入力ホスト変数のプレースホルダの記述を保存する必要があります。(入力ホスト変数はバインド変数とも呼ばれます。)

プログラム内にアクティブな SQL 文が複数ある (たとえばプログラムが複数のカーソルを OPEN している) ときは、それぞれの文に専用の SQLDA が必要になります。ただし、カーソルが同時に実行されなければ SQLDA を再利用できます。なお、1 つのプログラム内の SQLDA の数に制限はありません。

DESCRIBE 文

DESCRIBE は選択リスト項目または入力ホスト変数の記述を保存するために記述子を初期化します。

選択記述子を指定すると、PREPARE した動的問合せのそれぞれの選択リスト項目が DESCRIBE SELECT LIST 文によってチェックされます。これによって、選択リスト項目の名前、データ型、制約、長さ、スケールおよび精度が決定されます。続いて、この情報がその選択記述子に格納されます。

バインド記述子を指定すると、PREPARE 文で作成された動的問合せの各プレースホルダが DESCRIBE BIND VARIABLES 文によってチェックされます。これによって、プレースホルダの名前と長さ、プレースホルダに関連付けられている入力ホスト変数のデータ型が決定されます。続いて、この情報がそのバインド記述子に格納されます。たとえばプレースホルダ名を使用することによって、ホスト変数の値の入力をユーザーに要求できます。

SQLDA

SQLDA はホスト・プログラムのデータ構造体です。この構造体は選択リスト項目または入力ホスト変数の記述を保持します。

SQLDA 変数は DECLARE SECTION では定義されません。

選択 SQLDA には問合せ選択リストに関する次の情報が格納されています。

- DESCRIBE できる列の最大数
- 実際に DESCRIBE で検出された列数
- 列値を格納するバッファのアドレス
- 列値の長さ
- 列の値のデータ型
- 標識変数の値のアドレス
- 列名を格納するバッファのアドレス
- 列名を格納するバッファのサイズ
- 列名の現行の長さ

バインド SQLDA には SQL 文の入力ホスト変数に関する次の情報が格納されています。

- DESCRIBE できるプレースホルダの最大数
- 実際に DESCRIBE で検出されたプレースホルダの数
- 入力ホスト変数のアドレス
- 入力ホスト変数の長さ
- 入力ホスト変数のデータ型
- 標識変数のアドレス
- プレースホルダ名を格納するバッファのアドレス
- プレースホルダ名を格納するバッファのサイズ
- プレースホルダ名の現行の長さ
- 標識変数名を格納するバッファのアドレス
- 標識変数名を格納するバッファのサイズ
- 標識変数名の現行の長さ

SQLDA 構造体と変数名については、[第 15 章「Oracle の動的 SQL 方法 4」](#) を参照してください。

Oracle 方法 4 の実行

Oracle 方法 4 では、一般に次の順序で埋込み SQL 文を使用します。

```
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;
```

ただし、選択記述子とバインド記述子が同時に動作する必要はありません。したがって、問合せ選択リストの列数が明確でも入力ホスト変数のプレースホルダの数が不明な場合は、方法 4 の OPEN 文とともに次の方法 3 の FETCH 文を使用できます。

```
EXEC SQL FETCH emp_cursor INTO host_variable_list;
```

逆に、入力ホスト変数のプレースホルダの数は明確でも問合せ選択リストの列数が不明な場合は、次の方法 3 の OPEN 文

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

を方法 4 の FETCH 文とともに使用できます。

方法 4 では EXECUTE を非問合せにも使用できることに注意してください。

制限

動的 SQL 方法 4 では、"table" タイプのパラメータを使用して、ホスト配列を PL/SQL プロシージャにバインドすることはできません。

DECLARE STATEMENT 文の使用方法

方法 2、3 および 4 では、次の文を使用する必要がある場合があります。

```
EXEC SQL [AT db_name] DECLARE statement_name STATEMENT;
```

db_name および *statement_name* は、プリコンパイラによって使用される識別子で、ホスト変数でもプログラム変数でもありません。

DECLARE STATEMENT によって動的 SQL 文の名前が宣言されます。すると、この動的 SQL 文は PREPARE、EXECUTE、DECLARE CURSOR および DESCRIBE で参照できます。

デフォルト以外のデータベースで動的 SQL 文を実行するときにこの文が必要になります。方法 2 での使用例を次に示します。

```
EXEC SQL AT remote_db DECLARE sql_stmt STATEMENT;  
EXEC SQL PREPARE sql_stmt FROM :dyn_string;  
EXEC SQL EXECUTE sql_stmt;
```

この例では、どこで SQL 文を EXECUTE するかを *remote_db* によって Oracle に指示します。

方法 3 および方法 4 では、次の例に示すように DECLARE CURSOR 文が PREPARE 文の前にあるときにも DECLARE STATEMENT が必要です。

```
EXEC SQL DECLARE sql_stmt STATEMENT;  
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;  
EXEC SQL PREPARE sql_stmt FROM :dyn_string;
```

一般的な文の順序は次のとおりです。

```
EXEC SQL PREPARE sql_stmt FROM :dyn_string;  
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

ホスト配列の使用

静的 SQL および動的 SQL 内でのホスト配列の使用方法は似ています。たとえば、動的 SQL 方法 2 で入力ホスト配列を使用するには、次の構文を使用します。

```
EXEC SQL EXECUTE statement_name USING host_array_list;
```

host_array_list には 1 つ以上のホスト配列が格納されます。

同様に、方法 3 で入力ホスト配列を使用するには、次の構文を使用します。

```
OPEN cursor_name USING host_array_list;
```

また方法 3 で出力ホスト配列を使用するには、次の構文を使用します。

```
FETCH cursor_name INTO host_array_list;
```

方法 4 では、オプションの FOR 句を使用して Oracle に入力ホスト配列または出力ホスト配列のサイズを指示する必要があります。これについては、[第 15 章「Oracle の動的 SQL 方法 4」](#)で説明します。

PL/SQL の使用方法

Pro*C/C++ プリコンパイラは PL/SQL ブロックを単一の SQL 文として取り扱います。したがって SQL 文と同様に、PL/SQL ブロックを文字列ホスト変数またはリテラルに格納できます。文字列に PL/SQL ブロックを格納する場合は、キーワード EXEC SQL EXECUTE、キーワード END-EXEC および文の終了記号 ';' を省略します。

ただし、プリコンパイラによる SQL と PL/SQL の処理方法には次の 2 つの違いがあります。

- PL/SQL ホスト変数の PL/SQL ブロック内での役割が入力ホスト変数、出力ホスト変数、その両方のうちどれであっても、プリコンパイラは PL/SQL ホスト変数をすべて入力ホスト変数として扱います。
- PL/SQL ブロックに格納できる SQL 文の数には制限がないため、PL/SQL ブロックからは FETCH できません。

方法 1 の場合

PL/SQL ブロックにホスト変数が含まれている場合は、方法 1 で通常どおり PL/SQL 文字列を EXECUTE できます。

方法 2 の場合

PL/SQL ブロック内の入力ホスト変数および出力ホスト変数の数が明確である場合は、方法 2 で通常どおり PL/SQL 文字列を PREPARE および EXECUTE できます。

USING 句には、すべてのホスト変数を指定する必要があります。この PL/SQL 文を EXECUTE すると、USING 句内のホスト変数は PREPARE された文字列内の対応するプレースホルダに置換されます。プリコンパイラが PL/SQL ホスト変数をすべて入力ホスト変数として扱っても、値は正しく代入されます。入力（プログラム）値は入力ホスト変数に代入されます。また出力（列）値は出力ホスト変数に代入されます。

PREPARE された PL/SQL 文字列中のプレースホルダは、それぞれ USING 句のホスト変数に対応している必要があります。したがって、PREPARE された文に同じプレースホルダが 2 回以上現れるときは、それぞれが USING 句の個別のホスト変数に対応している必要があります。

方法 3 の場合

方法 3 は、FETCH が使用できることを除けば方法 2 と同じです。PL/SQL ブロックからの FETCH はできないので、方法 2 を使用してください。

Oracle 方法 4 の場合

PL/SQL ブロックに不明数の入力または出力ホスト変数が含まれる場合は、方法 4 を使用する必要があります。

方法 4 を使用するには、すべての入力ホスト変数および出力ホスト変数について 1 つのバインド記述子を設定します。DESCRIBE BIND VARIABLES を実行すると、入力ホスト変数および出力ホスト変数に関する情報がそのバインド記述子に格納されます。プリコンパイラは PL/SQL ホスト変数をすべて入力ホスト変数として扱うため、DESCRIBE SELECT LIST を実行しても効果はありません。

警告：動的 SQL 方法 4 では、"table" タイプのパラメータを使用して、ホスト配列を PL/SQL プロシージャにバインドすることはできません。

方法 4 でバインド記述子を使用する方法は、[第 15 章「Oracle の動的 SQL 方法 4」](#)を参照してください。

注意

ANSI では行終了文字が無視されるため、動的に処理される PL/SQL ブロックでは ANSI 形式のコメント (--) は使用しないでください。ANSI で記述すると、行の終わりではなくブロックの終わりまでコメントが続いてしまいます。ANSI 形式のコメントではなく、C 形式のコメント (/... */) を使用してください。

ANSI 動的 SQL

この章では、Oracle ANSI 動的 SQL (SQL92 動的 SQL と呼びます) のインプリメンテーションについて説明します。ANSI 動的 SQL は新しい方法 4 アプリケーションで使用されます。これは、旧バージョンの Oracle 動的 SQL 方法 4 の拡張版です。詳細は、[第 15 章「Oracle の動的 SQL 方法 4」](#)を参照してください。

ANSI 方法 4 では、すべての Oracle 型がサポートされます。旧バージョンの Oracle 方法 4 では、オブジェクト型、カーソル変数、構造体の配列、DML 戻り句、Unicode 変数および LOB はサポートされませんでした。

ANSI 動的 SQL では、記述子は Oracle によって内部的に保持されます。一方、旧バージョンの Oracle 動的 SQL 方法 4 では、記述子はユーザーにより Pro*C/C++ プログラムで定義されます。どちらの場合も、方法 4 では Pro*C/C++ プログラムを使用してホスト変数を含む動的 SQL 文を受け取ったり作成することができます。含まれるホスト変数の個数は様々です。

オブジェクト・オプション付きの Oracle8i Enterprise Edition を購入した場合に限り、オブジェクト型およびオブジェクト型トランスレータがサポートされます。

この章は主に次の項で構成されています。

- [ANSI 動的 SQL の基本](#)
- [ANSI SQL 文の概要](#)
- [Oracle 拡張機能](#)
- [ANSI 動的 SQL プリコンパイラ・オプション](#)
- [動的 SQL 文の完全な構文](#)
- [サンプル・プログラム](#)

ANSI 動的 SQL の基本

次の SQL 文について考えます。

```
SELECT ename, empno FROM emp WHERE deptno = :deptno_data
```

ANSI 動的 SQL を使用するステップは、次のとおりです。

- 変数および実行する文を保持する文字列の宣言。
- 入力および出力変数の記述子の割当て。
- 文の準備。
- 入力記述子の入力の記述。
- 入力記述子（上の例の入力ホスト・バインド変数は、deptno_data）の設定。
- 動的カーソルの宣言およびオープン。
- 出力記述子（上の例の出力ホスト変数は、ename および empno）の設定。
- データを繰り返しフェッチします。GET DESCRIPTOR を使用して各行から ename および empno データ・フィールドを取り出すことにより、データをフェッチします。
- 取り出したデータの利用（データの出力など）。
- 動的カーソルのクローズと入力および出力記述子の割当ての解除。

プリコンパイラのオプション

マイクロ・プリコンパイラ・オプションを DYNAMIC から ANSI に設定するか、マクロ・オプションを MODE から ANSI に設定してください。これにより、DYNAMIC のデフォルト値が ANSI に設定されます。DYNAMIC のもう一つの設定値は、ORACLE です。

ANSI 型コードを使用するには、プリコンパイラ・マイクロ・オプションを TYPE_CODE から ANSI に設定するか、マクロ・オプションを MODE から ANSI に設定します。これにより、デフォルト値が TYPE_CODE から ANSI に変更されます。TYPE_CODE を ANSI に設定する場合、DYNAMIC も ANSI に設定する必要があります。

14-4 ページの表 14-1「ANSI SQL データ型」に記載されている ANSI SQL 型の Oracle によるインプリメンテーションは、ANSI 規格と完全には合致していません。たとえば、INTEGER として宣言された列の記述では、NUMERIC のコードが戻されます。Oracle を ANSI 規格に近づけると、動作にわずかな変更が必要になる場合があります。ご使用のアプリケーションをデータベース・プラットフォーム間で移植できるようにし、可能な限り ANSI 準拠にする場合、TYPE_CODE プリコンパイラ・オプションを設定した ANSI 型を使用してください。このような変更ができない場合は、TYPE_CODE を ANSI に設定しないでください。

ANSI SQL 文の概要

動的 SQL 文で記述子を使用する前に、記述子領域を割り当てます。

ALLOCATE DESCRIPTOR 文の構文は次のとおりです。

```
EXEC SQL ALLOCATE DESCRIPTOR [GLOBAL | LOCAL] { :desc_name | string_literal }  
    [WITH MAX { :occurrences | numeric_literal } ] ;
```

グローバル記述子は、プログラム内のどのモジュールでも使用できます。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。デフォルト値は、Local です。

記述子名の desc_name には、引用符で囲んだりテラルまたはホスト変数に格納した文字値を代入できます。

occurrences は、記述子が保持できるバインド変数または列数の最大値（デフォルトは 100）です。

記述子が必要でない場合、割当てを解除するとメモリーを節約できます。それ以外の場合には、アクティブなデータベース接続がなくなった時点で自動的に割当てが解除されます。

割当て解除文は次のとおりです。

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL] { :desc_name | string_literal } ;
```

準備済の SQL 文の情報を取得するには、DESCRIBE 文を使用します。準備済の動的文のバインド変数を記述するには、DESCRIBE INPUT を使用します。出力列の数、型および長さを取得するには、DESCRIBE OUTPUT（デフォルト）を使用します。構文を簡略化すると次のようになります。

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] sql_statement  
    USING [SQL] DESCRIPTOR [GLOBAL | LOCAL] { :desc_name | string_literal } ;
```

SQL 文に入力値および出力値がある場合、記述子を 2 つ割り当てる必要があります。1 つは入力用にもう 1 つは出力用に割り当てます。次に例を示します。

```
SELECT ename, empno FROM emp ;
```

入力値がない場合には、入力記述子は必要ありません。

INSERTS、UPDATES、DELETES および SELECT 文の WHERE 句の入力値を指定するには、SET DESCRIPTOR 文を使用します。入力記述子内に DESCRIBE していないときに入力バインド変数の数（COUNT に格納されています）を設定するには、SET DESCRIPTOR 文を使用します。

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] { :desc_name | string_literal }  
    COUNT = { :kount | numeric_literal } ;
```

kount には、ホスト変数または数値リテラル（5 など）を設定できます。SET DESCRIPTOR 文を使用して、各ホスト変数に少なくともデータ・ソースを指定してください。

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] { :desc_nam | string_literal }  
VALUE item_number DATA = :hv3 ;
```

また入力ホスト変数の型および長さも設定できます。

注意：TYPE_CODE=ORACLE のとき、SET 文を介して明示的にまたは DESCRIBE OUTPUT によって暗黙的に TYPE および LENGTH を指定していない場合は、プリコンパイラではホスト変数から導出された値が使用されます。TYPE_CODE=ANSI のときは、[表 14-1「ANSI SQL データ型」](#)の値を使用して TYPE を設定する必要があります。また ANSI デフォルト長はホスト変数に合致しないことがあるため、LENGTH も設定する必要があります。

```
EXEC SQL SET DESCRIPTOR [GLOBAL | LOCAL] { :desc_nam | string_literal }  
VALUE item_number TYPE = :hv1, LENGTH = :hv2, DATA = :hv3 ;
```

hv1、hv2 および hv3 といった識別子は、ホスト変数から値を供給する必要があることをユーザーが忘れないようにするために使用します。item_number は、入力変数の SQL 文内での位置を表します。

TYPE_CODE が ANSI に設定されている場合、TYPE は次の表から選択されるタイプ・コードになります。

表 14-1 ANSI SQL データ型

データ型	タイプ・コード
CHARACTER	1
CHARACTER VARYING	12
DATE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
NUMERIC	2
REAL	7
SMALLINT	5

Oracle タイプ・コードについては、15-13 ページの[表 15-2「Oracle の外部データ型とデータ型コード」](#)を参照してください。

DATA は、入力されるホスト変数の値です。

標識、精度およびスケールなどの他の入力値も設定できます。可能な記述子項目名の完全な説明は、14-17 ページの「[SET DESCRIPTOR](#)」を参照してください。

SET DESCRIPTOR 文の数値は、**int** または **short int** のいずれかで宣言する必要があります。ただし、記述子および戻された長さの値は **short int** として宣言する必要があります。

たとえば、次の例で empno を取得する場合、empno は動的 SQL 文の 2 番目の出力ホスト変数なので、値を VALUE = 2 に設定します。ホスト変数 empno_ttyp は、3 (Oracle タイプの整数値) に設定します。ホスト整数の長さを表す empno_len は、4 に設定します。この値はホスト変数のサイズです。DATA はホスト変数 empno_data と等しくなります。この変数は値をデータベース表から受け取ります。コードの一部は、次のようになります。

```
...
char *dyn_statement = "SELECT ename, empno FROM emp
    WHERE deptno = :deptno_number" ;
int empno_data ;
int empno_ttyp = 3 ;
int empno_len = 4 ;
...
EXEC SQL SET DESCRIPTOR 'out' VALUE 2  TYPE = :empno_ttyp, LENGTH = :empno_len,
    DATA = :empno_data ;
```

入力値を設定後、入力記述子を使用して文を実行またはオープンします。文中に出力値がある場合、FETCH を行う前に出力値を設定してください。DESCRIBE OUTPUT した場合、DESCRIBE の実行によってホスト変数の外部型および長さとは異なる内部型および長さが生成されるため、ホスト変数の実際の型および長さをリセットする必要があります。

出力記述子を FETCH した後、戻されたデータにアクセスするには、GET DESCRIPTOR を使用します。簡略化された構文は次のとおりです。構文の詳細は、この章の後半部分を参照してください。

```
EXEC SQL GET DESCRIPTOR [GLOBAL | LOCAL] { :desc_nam | string_literal }
    VALUE item_number :hv1 = DATA, :hv2 = INDICATOR, :hv3 = RETURNED_LENGTH ;
```

desc_nam および item_number には、リテラルまたはホスト変数を指定できます。記述子には、'out' などリテラルの名前を指定できます。項目番号には、2 などの数値リテラルを指定できます。

hv1、hv2 および hv3 は、ホスト変数です。これらはホスト変数であり、リテラルではありません。例では、ホスト変数が 3 つのみ使用されています。戻されたデータから取得できるすべての項目の一覧については、14-15 ページの表 14-4 「[GET DESCRIPTOR の記述子項目名の定義](#)」を参照してください。

数値すべてに **long**、**int** または **short** のいずれかを指定します。ただし、記述子または戻された長さの値は **short** にする必要があります。

サンプル・コード

次の例で ANSI 動的 SQL の使用例を示します。ここでは入力記述子 ('in') および出力記述子 ('out') を割り当てて SELECT 文を実行します。入力値は SET DESCRIPTOR 文を使用して設定します。カーソルはオープンおよびフェッチされ、結果の出力値は GET DESCRIPTOR 文を使用して取得されます。

```
...
char* dyn_statement = "SELECT ename, empno FROM emp WHERE deptno = :deptno_data" ;
int deptno_type = 3, deptno_len = 2, deptno_data = 10 ;
int ename_type = 97, ename_len = 30 ;
char ename_data[31] ;
int empno_type = 3, empno_len = 4 ;
int empno_data ;
long 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' ;
    ...
}
```

Oracle 拡張機能

次の拡張機能について説明します。

- SET 文のデータ項目の参照セマンティクス
- 配列を使用したバルク操作
- オブジェクト型、NCHAR 列および LOB のサポート

参照セマンティクス

ANSI 規格により値構文が指定されます。パフォーマンスを改善するために、Oracle によりこの規格に参照セマンティクスが含まれました。

値構文ではホスト変数データのコピーが生成されます。参照セマンティクスでは、ホスト変数のアドレスを使用することによりコピーを回避しています。このように参照セマンティクスを使用すると、大容量データを処理する際のパフォーマンスが向上します。

フェッチ速度を速めるには、データ句の前に REF キーワードを使用してください。

```
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 ;
```

ホスト変数は、取得結果を受け取ります。GET 文は必要ありません。取得されたデータは、FETCH のたびに ename_data および empno_data に直接書き込まれます。

次のコード例のように、REF キーワードは、DATA、INDICATOR および RETURNED_LENGTH 項目（フェッチされた行ごとに異なります）の前でのみ使用できます。

```
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 ;
```

フェッチするたびに returnLen によって ename フィールドの実際に取得した長さが保持されます。このフィールドは CHAR または VARCHAR2 データで使用する则便利です。

ename_len は、戻された長さを受け取りません。また FETCH 文によっても変更されません。DESCRIBE 文とそれに続く GET 文を使用して、データ行をフェッチする前に列の最大幅を調べてください。

処理速度を速くするために、SQL 文の SELECT 以外の型で REF キーワードを使用することもできます。参照セマンティクスの場合、記述子領域にコピーされた値よりもむしろホスト変数が使用される点に注意してください。SET 時ではなく、SQL 文の実行時のホスト変数データが使用されます。例を示します。

```
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 */
```

詳細は、14-17 ページの「[SET DESCRIPTOR](#)」を参照してください。

配列を使用したバルク操作

Oracle によりバルク操作機能が追加されて、SQL92 ANSI 動的規格が拡張されました。バルク操作を行うには、配列サイズを指定した FOR 句を使用して処理する入力データ数または行数を指定します。

データまたは行数の最大値を指定するには、ALLOCATE 文で FOR 句を使用します。たとえば、最大配列サイズを 100 に指定する場合、次のようにします。

```
EXEC SQL FOR 100 ALLOCATE DESCRIPTOR 'out' ;
```

または

```
int array_size = 100 ;
...
EXEC SQL FOR :array_size ALLOCATE DESCRIPTOR 'out' ;
```

FOR 句は、記述子にアクセスする後続の文で使用されます。出力記述子での FETCH 文の配列サイズは、ALLOCATE 文で指定した配列サイズ以下にする必要があります。

```
EXEC SQL FOR 20 FETCH c1 USING DESCRIPTOR 'out' ;
```

同じ記述子の後続の GET 文では、FETCH 文と同じ配列サイズを指定する必要があります。GET 文を使用すると、DATA、INDICATOR または RETURNED_LENGTH 値を取得できます。

```
int val_data[20] ;
short val_indi[20] ;
...
EXEC SQL FOR 20 GET DESCRIPTOR 'out' VALUE 1 :val_data = DATA,
      :val_indi = INDICATOR ;
```

ただし、行ごとに異なることのない他の項目（LENGTH、TYPE、COUNT など）を参照する GET 文では、FOR 句を使用しないでください。

```
int cnt, len ;
...
EXEC SQL GET DESCRIPTOR 'out' :cnt = COUNT ;
EXEC SQL GET DESCRIPTOR 'out' VALUE 1 :len = LENGTH ;
```

これは参照セマンティクスを使用する SET 文でも同様です。FETCH 文の前に置かれ、DATA、INDICATOR または RETURNED_LENGTH の参照セマンティクスを使用する SET 文には、FETCH と同じ配列サイズを指定する必要があります。

```
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 ;
```

同様に、入力に使用する記述子を使用して、たとえば行の集まりを挿入する場合、EXECUTE または OPEN 文には、ALLOCATE 文で使用するサイズ以下の配列サイズを指定する必要があります。値構文と参照セマンティクス両方の SET 文では、EXECUTE 文と同じサイズの配列を使用する必要があります。この SET 文を使用して DATA、INDICATOR または RETURNED_LENGTH にアクセスします。

FOR 句は、DEALLOCATE 文または PREPARE 文では使用されません。

次のサンプル・コードでは、出力記述子がないバルク操作について説明します。このサンプル・コードでは出力は行われず、emp 表への挿入操作のみが実行されます。COUNT の値は 2 です。INSERT 文には ename_arr と empno_arr の 2 つのホスト変数があります。データ配列 ename_arr には、順番に "Tom"、"Dick" および "Harry" という 3 つの文字列が保持されます。標識配列 ename_ind の 2 番目の要素には、-1 という値が指定されるため、"Dick" のかわりに NULL が挿入されます。データ配列 empno_arr には、従業員番号が 3 つ含まれています。DML 戻り句を使用すると、実際に挿入された名前を確認できます。詳細は、6-10 ページの「[DML 戻り句](#)」を参照してください。

```
...
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' ;
...
```

上のコードを実行すると、次の値が挿入されます。

```
EMPNO  ENAME
8001    Tom
8002
8003    Harry
```

制限や注意事項は、8-13 ページの「[FOR 句の使用方法](#)」を参照してください。

構造体配列のサポート

HOST_STRIDE_LENGTH を構造体のサイズに、INDICATOR_STRIDE_LENGTH を標識構造体のサイズに、そして RETURNED_LENGTH_STRIDE を戻された長さの構造体のサイズに設定する必要があります。

これらの項目の詳細は、14-15 ページの表 14-5「[Oracle 拡張機能により追加された GET DESCRIPTOR の記述子項目名の定義](#)」を参照してください。

構造体の配列は、ANSI 動的 SQL によってサポートされていますが、旧バージョンの Oracle 動的 SQL ではサポートされていません。

オブジェクト型のサポート

独自に定義したオブジェクト型では、Oracle TYPE を 108 にして使用してください。オブジェクト型の列では、DESCRIBE 文を使用して USER_DEFINED_TYPE_VERSION、USER_DEFINED_TYPE_NAME、USER_DEFINED_TYPE_NAME_LENGTH、USER_DEFINED_TYPE_SCHEMA および USER_DEFINED_TYPE_SCHEMA_LENGTH を取得します。

DESCRIBE 文を使用しないでこれらの値を取得する場合、SET DESCRIPTOR 文を使用して自分で設定を行う必要があります。

ANSI 動的 SQL プリコンパイラ・オプション

マクロ・オプションの MODE (10-30 ページの「[MODE](#)」を参照) を使用すると、ANSI と互換性のある特性を設定したり多くの機能を制御できます。このオプションでは ANSI または ORACLE の値を使用できます。個々の機能に対しては、マクロ・オプションを使用します。このオプションは MODE 設定をオーバーライドします。

動的 SQL での記述子の動作を指定する場合、プリコンパイラ・マクロ・オプション DYNAMIC を使用します。ANSI または Oracle のどちらのデータ型を使用するかを指定する場合、プリコンパイラ・マクロ・オプション TYPE_CODE を使用します。

マクロ・オプション MODE を ANSI に設定すると、マクロ・オプション DYNAMIC も自動的に ANSI になります。同様に MODE を ORACLE に設定すると、DYNAMIC も ORACLE になります。

DYNAMIC と TYPE_CODE はインラインでは使用できません。

次の表に機能と DYNAMIC の設定がその機能に与える影響を示します。

表 14-2 DYNAMIC オプションの設定

機能	DYNAMIC = ANSI	DYNAMIC = ORACLE
記述子の生成	ALLOCATE 文を使用する必要があります。	関数 SQLSQLDAAlloc() を使用する必要があります。5-49 ページの「 SQLLIB パブリック関数の新しい名前 」を参照してください。
記述子の破壊	DEALLOCATE 文が使用可能です。	関数 SQLLDADeFree() が使用可能です。5-49 ページの「 SQLLIB パブリック関数の新しい名前 」を参照してください。
データの取得	FETCH と GET 文の両方が使用可能です。	FETCH 文のみ使用する必要があります。
入力データの設定	DESCRIBE INPUT 文が使用可能です。SET 文を使用する必要があります。	コードに記述子の値を設定する必要があります。DESCRIBE BIND VARIABLES 文を使用する必要があります。
記述子の表現	引用符で囲んだりテラルまたは記述子名を含むホスト識別子。	SQLDA へのポインタであるホスト変数。
使用可能なデータ型	BIT 以外のすべての ANSI 型およびすべての Oracle 型。	オブジェクト、LOB、構造体の配列およびカーソル変数以外の Oracle 型。

マイクロ・オプション TYPE_CODE は、プリコンパイラによってマイクロ・オプション MODE と同じ設定にされます。TYPE_CODE が ANSI に等しくなるのは、DYNAMIC が ANSI に等しい場合のみです。

TYPE_CODE 設定に対応する機能は次のとおりです。

表 14-3 TYPE_CODE オプション設定

機能	TYPE_CODE = ANSI	TYPE_CODE = ORACLE
動的 SQL へのデータ型コード番号の入力および戻り値	ANSI 型が存在する場合は ANSI コードを使用します。それ以外の場合は、負の Oracle コード番号を使用します。 DYNAMIC = ANSI のときのみ有効です。	Oracle コード番号を使用します。 DYNAMIC の設定に関係なく使用可能です。

動的 SQL 文の完全な構文

ここで説明する構文の詳細は、付録 F-1 ページの「[埋込み SQL 文および宣言文](#)」でアルファベット順に列挙されている情報を参照してください。

ALLOCATE DESCRIPTOR

用途

この文を使用して SQL 記述子領域を割り当てます。記述子、ホスト・バインド項目発生数の最大値および配列サイズを指定します。この文は、ANSI 動的 SQL でのみ使用できます。

構文

```
EXEC SQL [FOR [:]array_size] ALLOCATE DESCRIPTOR [GLOBAL | LOCAL]
        { :desc_nam | string_literal } [WITH MAX occurrences] ;
```

変数

array_size

これは配列処理をサポートするオプション句（Oracle 拡張機能）です。この句によって配列処理で記述子が使用可能であることがプリコンパイラに通知されます。

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

識別子の名前。ローカル記述子は、モジュール内で一意にしてください。前回の割当てを解除せずに記述子を割り当てた場合、ランタイム・エラーが生成されます。グローバル記述子は、アプリケーション全体で一意にしてください。そうでない場合はランタイム・エラーが発生します。

occurrences

記述子で使用可能なホスト変数の最大値です。この値は、0 から 64K までの整数定数にする必要があります。それ以外の場合はエラーが戻されます。デフォルト値は 100 です。この句はオプションです。これらの規則に違反するとプリコンパイラ・エラーが戻されます。

例

```
EXEC SQL ALLOCATE DESCRIPTOR 'SELDES' WITH MAX 50 ;
```

```
EXEC SQL FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25 ;
```


DEALLOCATE DESCRIPTOR

用途

以前に割り当てられた SQL 記述子の割当てを解除してメモリーを解放する場合、この文を使用します。この文は、ANSI 動的 SQL でのみ使用できます。

構文

```
EXEC SQL DEALLOCATE DESCRIPTOR [GLOBAL | LOCAL] { :desc_nam | string_literal } ;
```

変数

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

同じ名前および適用範囲の記述子が割り当てられていない場合または割り当てられていたが割当てが解除されている場合、ランタイム・エラーが発生します。

例

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES' ;
```

```
EXEC SQL DEALLOCATE DESCRIPTOR :binddes ;
```

GET DESCRIPTOR

用途

この文を使用して SQL 記述子領域から情報を取得します。

構文

```
EXEC SQL [FOR [:]array_size] GET DESCRIPTOR [GLOBAL | LOCAL]  
  { :desc_nam | string_literal }  
  { :hv0 = COUNT | VALUE item_number  
    :hv1 = item_name1 [ { , :hvN = item_nameN } ] } ;
```

変数

array_size

FOR array_size は、オプションの Oracle 拡張機能です。array_size は、FETCH 文の array_size フィールドと等しくする必要があります。

COUNT

バインド変数の総数。

desc_nam

識別子の名前。

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

VALUE item_number

SQL 文内での項目の位置。item_number には変数または定数を指定できます。
item_number の値が COUNT より大きい場合、"no data found" 条件が戻されます。
item_number は 0 より大きくする必要があります。

hv1 .. hvN

値が転送されるホスト変数。

item_name1 .. item_nameN

ホスト変数に対応する記述子項目名。使用可能な ANSI 記述子項目名は 次のとおりです。

表 14-4 GET DESCRIPTOR の記述子項目名の定義

記述子項目名	意味
TYPE	ANSI タイプ・コードは、14-4 ページの表 14-1「ANSI SQL データ型」を参照してください。Oracle タイプ・コードは、15-13 ページの表 15-2「Oracle の外部データ型とデータ型コード」を参照してください。ANSI データ型が表にない場合および TYPE_CODE=ANSI の場合、負の値の Oracle 型コードを使用してください。
LENGTH	列データの長さ。NCHAR では文字数、その他の場合はバイト数で表されます。DESCRIBE OUTPUT によって設定されます。
OCTET_LENGTH	データの長さ。バイト数で表します。
RETURNED_LENGTH	FETCH 後の実際のデータ長。
RETURNED_OCTET_LENGTH	戻されたデータの長さ。バイト数で表します。
PRECISION	桁数。
SCALE	正確な数値型の場合は、小数点以下の桁数。
NULLABLE	1 の場合、列に NULL 値を指定できます。0 の場合、列には NULL 値を指定できません。
INDICATOR	関連付けられた記述子の値。
DATA	データの値。
NAME	列の名前。
CHARACTER_SET_NAME	列のキャラクタ・セット。

追加された Oracle 記述子項目の名前は次のとおりです。

表 14-5 Oracle 拡張機能により追加された GET DESCRIPTOR の記述子項目名の定義

記述子項目名	意味
NATIONAL_CHARACTER	2 の場合、NCHAR または NVARCHAR2 です。1 の場合、文字です。0 の場合は文字以外の値です。
INTERNAL_LENGTH	内部の長さ。バイト数で表します。
HOST_STRIDE_LENGTH	ホスト構造体のサイズ。バイト数で表します。
INDICATOR_STRIDE_LENGTH	標識構造体のサイズ。バイト数で表します。
RETURNED_LENGTH_STRIDE	戻された長さの構造体のサイズ。バイト数で表します。

表 14-5 Oracle 拡張機能により追加された GET DESCRIPTOR の記述子項目名の定義（続き）

USER_DEFINED_TYPE_VERSION	オブジェクト型バージョンを表す文字。
USER_DEFINED_TYPE_NAME	オブジェクト型の名前。
USER_DEFINED_TYPE_NAME_LENGTH	オブジェクト型の名前の長さ。
USER_DEFINED_TYPE_SCHEMA	オブジェクト・スキーマを表す文字。
USER_DEFINED_TYPE_SCHEMA_LENGTH	USER_DEFINED_TYPE_SCHEMA の長さ。

使用上の注意

FOR 句は、DATA、INDICATOR および RETURNED_LENGTH 項目を含む GET DESCRIPTOR 文でのみ使用してください。

内部型は、DESCRIBE OUTPUT 文によって指定されます。その型を入力と出力の両方でホスト変数の外部型に設定する必要があります。TYPE は、14-4 ページの表 14-1「ANSI SQL データ型」に記載されている ANSI コードです。ANSI コードが表に含まれていない場合、15-13 ページの表 15-2「Oracle の外部データ型とデータ型コード」に記載されている負の値の Oracle 型コードを使用してください。

LENGTH には、固定幅の各国語文字（NLC）セットを持つフィールドの列の長さを表す文字数が含まれます。他の文字列はバイト数で表されます。DESCRIBE OUTPUT で設定されます。

RETURNED_LENGTH は、FETCH 文によって設定された実際のデータ長です。これは、LENGTH の場合と同様にバイト数または文字数で表されます。フィールド OCTET_LENGTH および RETURNED_OCTET_LENGTH の長さはバイト数で表されます。

NULLABLE = 1 は、列で NULL を使用できることを意味し、NULLABLE = 0 は NULL を使用できないことを意味します。

CHARACTER_SET_NAME は、文字列でのみ使用します。他の型については定義されていません。DESCRIBE OUTPUT 文を使用して値を取得します。

DATA および INDICATOR は、列のデータおよび標識の状態を表します。data = NULL でも標識が要求されていない場合、ランタイム・エラーが生成されます。("DATA EXCEPTION, NULL VALUE, NO INDICATOR PARAMETER")

Oracle 固有の記述子項目名

列が NCHAR または NVARCHAR2 の場合、NATIONAL_CHARACTER = 2 に設定されます。列が文字列（ただし各国文字ではない）の場合、この項目は 1 に設定されます。文字以外の列の場合、DESCRIBE OUTPUT の実行後この項目は 0 になります。

INTERNAL_LENGTH は、Oracle 動的メソッド 4 との互換性を保つためのものです。この項目には Oracle SQL 記述子領域の長さメンバーと同じ値が設定されています。（第 15 章「Oracle の動的 SQL メソッド 4」を参照してください。）

次の 3 つの項目は DESCRIBE OUTPUT 文によって戻されません。

- ホスト変数構造体のサイズを示す HOST_STRIDE_LENGTH
- 標識変数構造体のサイズを示す INDICATOR_STRIDE_LENGTH
- 戻された長さの変数構造体のサイズを示す RETURNED_LENGTH_STRIDE

次の項目は、プリコンパイラ・オプション OBJECTS が YES に設定されているときにのみオブジェクト型に適用されます。

- タイプ・バージョンを表す文字を含む USER_DEFINED_TYPE_VERSION
- 型の名前を表す文字を示す USER_DEFINED_TYPE_NAME
- 型の名前の長さをバイト数で示す USER_DEFINED_TYPE_NAME_LENGTH
- 型のスキーマ名を表す文字を示す USER_DEFINED_TYPE_SCHEMA
- 型のスキーマ名の長さを文字数で示す USER_DEFINED_TYPE_SCHEMA_LENGTH

例

```
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 ;
```

SET DESCRIPTOR

用途

この文を使用してホスト変数からの情報を記述子領域に設定します。SET DESCRIPTOR 文は、項目名のホスト変数のみをサポートしています。

構文

```
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}]}
```

変数

array_size

この Oracle オプション句で配列を使用できるのは、記述子項目 DATA、INDICATOR および RETURNED_LENGTH の設定時のみです。FOR 句を含む SET DESCRIPTOR では他の項目を使用できません。ホスト変数配列サイズはすべて一致している必要があります。FETCH 文で使用するのと同じ配列サイズを SET 文で使用してください。

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。この項目には、ALLOCATE DESCRIPTOR の規則が適用されます。

COUNT

バインド（入力）または定義（出力）変数の数。

VALUE item_number

ホスト変数の動的 SQL 文における位置。

hv1 .. hvN

設定したホスト変数（定数ではありません）。

item_name1

desc_item_name では、GET DESCRIPTOR 構文（14-13 ページの「[GET DESCRIPTOR](#)」を参照）と同様の方法でこれらの値が使用されます。

表 14-6 SET DESCRIPTOR の記述子項目名

記述子項目名	意味
TYPE	ANSI タイプ・コードは、14-4 ページの表 14-1「 ANSI SQL データ型 」を参照してください。Oracle タイプ・コードは、15-13 ページの表 15-2「 Oracle の外部データ型とデータ型コード 」を参照してください。対応する ANSI 型がない場合、負の値の Oracle 型を使用します。
LENGTH	列データの最大長。
INDICATOR	関連付けられた記述子の値。参照セマンティクス用に設定します。
DATA	設定されるデータの値。参照セマンティクス用に設定します。
CHARACTER_SET_NAME	列のキャラクタ・セット。

Oracle 拡張機能により追加された記述子項目の名前は次のとおりです。

表 14-7 Oracle 拡張機能により追加された SET DESCRIPTOR の記述子項目名の定義

記述子項目名	意味
RETURNED_LENGTH	FETCH 後に戻された長さ。参照セマンティクスを使用している場合に設定します。
NATIONAL_CHARACTER	入力ホスト変数が NCHAR または NVARCHAR2 型の場合、2 に設定します。
HOST_STRIDE_LENGTH	ホスト変数構造体のサイズ。バイト数で表します。
INDICATOR_STRIDE_LENGTH	標識変数のサイズ。バイト数で表します。
RETURNED_LENGTH_STRIDE	戻された長さの構造体のサイズ。バイト数で表します。
USER_DEFINED_TYPE_NAME	オブジェクト型の名前。
USER_DEFINED_TYPE_NAME_LENGTH	オブジェクト型の名前の長さ。
USER_DEFINED_TYPE_SCHEMA	オブジェクト・スキーマを表す文字。
USER_DEFINED_TYPE_SCHEMA_LENGTH	USER_DEFINED_TYPE_SCHEMA の長さ。

使用上の注意

参照セマンティクスは、パフォーマンスを向上させる別の Oracle 拡張機能です。REF キーワードは、次の記述子項目名の前にのみ使用します。DATA、INDICATOR および RETURNED_LENGTH です。REF キーワードの使用時には、GET 文を使用する必要はありません。複合データ型（オブジェクト型、コレクション型、構造体の配列および DML 戻り句）はすべて、SET DESCRIPTOR の REF 形式を必要とします。詳細は、6-10 ページの「DML 戻り句」を参照してください。

REF の使用時には、関連付けられたホスト変数自体が SET で使用されます。この場合、GET は必要ありません。値ではなく REF を使用している場合に限り、RETURNED_LENGTH を設定できます。

FETCH 文で使用するのと同じ配列サイズを SET 文または GET 文で使用してください。

NCHAR ホスト入力値用に NATIONAL_CHAR フィールドを 2 に設定します。

オブジェクト型の特性の設定時には、USER_DEFINED_TYPE_NAME および USER_DEFINED_TYPE_NAME_LENGTH を設定する必要があります。

この操作を省略すると、USER_DEFINED_TYPE_SCHEMA および USER_DEFINED_TYPE_SCHEMA_LENGTH のデフォルトがカレント接続になります。

クライアント側 Unicode サポートには、CHARACTER_SET_NAME を UTF16 に設定します。

例

バルク配列の例については、14-8 ページの「[配列を使用したバルク操作](#)」を参照してください。

```
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 ;
...
```

PREPARE の使用

用途

この方法で使用されている PREPARE 文は、他の動的 SQL 方法で使用されている PREPARE 文と同じです。Oracle 拡張機能により、変数と同様に SQL 文で引用符付きの文字列を使用できます。

構文

```
EXEC SQL PREPARE statement_id FROM :sql_statement ;
```

変数

statement_id

これを宣言しないでください。これは未宣言の SQL 識別子です。

sql_statement

埋込み SQL 文を保持する文字列（定数または変数）。

例

```
char* statement = "SELECT ENAME FROM emp WHERE deptno = :d" ;
EXEC SQL PREPARE S1 FROM :statement ;
```


DESCRIBE INPUT

用途

この文はバインド変数についての情報を戻します。

構文

```
EXEC SQL DESCRIBE INPUT statement_id USING [SQL] DESCRIPTOR  
[GLOBAL | LOCAL] {:desc_nam | string_literal};
```

変数

statement_id

PREPARE および DESCRIBE OUTPUT と同じように使用します。これを宣言しないでください。これは未宣言の SQL 識別子です。

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。

使用上の注意

DESCRIBE INPUT では、COUNT および NAME 項目のみが設定されます。

例

```
EXEC SQL DESCRIBE INPUT S1 USING SQL DESCRIPTOR GLOBAL :binddes ;  
EXEC SQL DESCRIBE INPUT S2 USING DESCRIPTOR 'input' ;
```

DESCRIBE OUTPUT

用途

PREPARE された文の出力列についての情報を取得する場合、この文を使用します。ANSI 構文は、旧バージョンの Oracle 構文と異なります。SQL 記述子領域に格納された情報には、戻り値の数、型、長さおよび名前など関連付けられた情報が含まれます。

構文

```
EXEC SQL DESCRIBE [OUTPUT] statement_id USING [SQL] DESCRIPTOR  
    [GLOBAL | LOCAL] {:desc_nam | string_literal} ;
```

変数

statement_id

PREPARE で使用されているものと同じです。これを宣言しないでください。これは未宣言の SQL 識別子です。

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。

OUTPUT はデフォルトで、省略可能です。

例

```
char* desname = "SELDES" ;  
EXEC SQL DESCRIBE S1 USING SQL DESCRIPTOR 'SELDES' ; /* Or, */  
EXEC SQL DESCRIBE OUTPUT S1 USING DESCRIPTOR :desname ;
```

EXECUTE

用途

EXECUTE 文では、まず準備済の SQL 文の入力および出力変数を一致させ、それから文が実行されます。EXECUTE の ANSI バージョンは、1 つの文中に 2 つの記述子を割り当てることにより DML 戻り句をサポートできる点で旧バージョンの EXECUTE 文とは異なります。

構文

```
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}] ;
```

変数

array_size

文によって処理される行数。

statement_id

PREPARE で使用されているものと同じです。これを宣言しないでください。これは未宣言の SQL 識別子です。この項目にリテラルを指定することも可能です。

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。

使用上の注意

INTO 句により INSERT、UPDATE および DELETE の DML 戻り句がインプリメントされます。6-10 ページの「[DML 戻り句](#)」を参照してください。

例

```
EXEC SQL EXECUTE S1 USING SQL DESCRIPTOR GLOBAL :binddes ;
```

```
EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES' ;
```

EXECUTE IMMEDIATE の使用

用途

リテラルまたは SQL 文を含むホスト変数文字列を実行します。この文の ANSI SQL 形式は、旧バージョンの Oracle 動的 SQL と同じです。

構文

```
EXEC SQL EXECUTE IMMEDIATE { :sql_statement | string_literal }
```

変数

sql_statement

文字列内の SQL 文または PL/SQL ブロック。

例

```
EXEC SQL EXECUTE IMMEDIATE :statement ;
```

DYNAMIC DECLARE CURSOR の使用

用途

問合せ文と関連付けられたカーソルを宣言します。これは汎用 DECLARE CURSOR 文の形式です。

構文

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_id;
```

変数

cursor_name

カーソル変数（ホスト変数ではなく SQL 識別子）。

statement_id

未宣言の SQL 識別子。

例

```
EXEC SQL DECLARE C1 CURSOR FOR S1 ;
```

OPEN カーソル

用途

OPEN 文は、入力パラメータとカーソルを関連付け、それからカーソルをオープンします。

構文

```
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}]] ;
```

変数

array_size

記述子の割当て時には、この制限は指定数以下になります。

dyn_cursor

カーソル変数。

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

記述子名。

使用上の注意

カーソルに関連付けられた準備済の文にコロンの疑問符が含まれる場合、USING 句を指定する必要があります。指定しない場合、ランタイム・エラーが発生します。DML 戻り句がサポートされています。6-10 ページの「[DML 戻り句](#)」を参照してください。

例

```
EXEC SQL OPEN C1 USING SQL DESCRIPTOR :binddes ;
```

```
EXEC SQL FOR :limit OPEN C2 USING DESCRIPTOR :b1, :b2  
      INTO SQL DESCRIPTOR :seldes ;
```

FETCH

用途

動的 DECLARE 文で宣言されたカーソルの行をフェッチします。

構文

```
EXEC SQL [FOR :array_size] FETCH cursor INTO [SQL] DESCRIPTOR  
      [GLOBAL | LOCAL] { :desc_nam | string_literal } ;
```

変数

array_size

文によって処理される行数。

cursor

以前に宣言された動的カーソル。

GLOBAL | LOCAL

デフォルトではオプション句の適用範囲は、LOCAL に設定されています。ローカル記述子には、割り当てられたファイル内でのみアクセスできます。グローバル記述子は、コンパイル・ユニット内のどのモジュールでも使用できます。

desc_nam

識別子の名前。

使用上の注意

FOR 句の `array_size` オプションは、`ALLOCATE DESCRIPTOR` 文で指定された数以下にする必要があります。

例

```
EXEC SQL FETCH FROM C1 INTO DESCRIPTOR 'SELDES' ;
```

```
EXEC SQL FOR :arsz FETCH C2 INTO DESCRIPTOR :desc ;
```

動的カーソルの CLOSE

用途

動的カーソルをクローズします。構文は、旧バージョンの Oracle 方法 4 から変更されていません。

構文

```
EXEC SQL CLOSE cursor ;
```

変数

cursor

以前に宣言された動的カーソル。

例

```
EXEC SQL CLOSE C1 ;
```

旧バージョンの Oracle 動的 method 4 との相違点

ANSI 動的 SQL インタフェースは、Oracle 動的 method 4 がサポートしたすべてのデータ型に加え、次の機能をサポートしています。

- ANSI 動的 SQL によるオブジェクト型、結果セットおよび LOB 型を含むすべてのデータ型のサポート。
- ANSI モードによる内部 SQL 記述子領域を使用した入力および出力情報の格納。この内部 SQL 記述子領域は、旧バージョンの Oracle 動的 method 4 で使用された外部 SQLDA の拡張版です。

- 次の新しい埋込み SQL 文の導入。ALLOCATE DESCRIPTOR、DEALLOCATE DESCRIPTOR、DESCRIBE、GET DESCRIPTOR および SET DESCRIPTOR。
- DESCRIBE 文では、ANSI 動的 SQL の標識変数名は戻されません。
- ANSI 動的 SQL では、戻された列名または式の最大値を指定できません。デフォルト・サイズは 128 です。
- 記述子名は、引用符で囲んだ識別子または前にコロンが付いたホスト変数のいずれかにする必要があります。
- 出力時、DESCRIBE 文の SELECT LIST FOR 句はオプション・キーワード OUTPUT に置き換えられます。INTO 句は USING DESCRIPTOR 句に置き換えられます。この句には、オプション・キーワード SQL を含めることができます。
- 入力時、DESCRIBE 文のオプションの BIND VARIABLES FOR 句をキーワード INPUT に置き換えることができます。INTO 句は USING DESCRIPTOR 句に置き換えられます。この句には、オプション・キーワード SQL を含めることができます。
- オプション・キーワード SQL を EXECUTE、FETCH および OPEN 文の USING 句のキーワード DESCRIPTOR の前に置くことができます。

制限

ANSI 動的 SQL に対する制限は、次のとおりです。

- 同じモジュール内で ANSI および Oracle 動的 SQL を組み合わせて使用できません。
- プリコンパイラ・オプション DYNAMIC を ANSI に設定する必要があります。DYNAMIC が ANSI に設定されている場合に限り、プリコンパイラ・オプション TYPE_CODE を ANSI に設定できます。
- SET 文は、項目名としてホスト変数のみをサポートしています。

サンプル・プログラム

次の 2 つのプログラムは、デモ・ディレクトリにあります。

ansidyn1.pc

このプログラムは、ANSI 動的 SQL を使用した SQL 文の処理方法を示します。この SQL 文は実行時まで不明です。このプログラムは、ANSI 動的 SQL を使用した最も簡単なプログラミング（最も効率的というわけではありません）を紹介することを目的としています。ここでは、ANSI と互換性のある値構文および ANSI 型コードを使用しています。ANSI SQLSTATE は、エラー番号用に使用されています。記述子名はリテラルです。すべての入力および出力には、ANSI 可変キャラクタ・タイプが使用されます。

このプログラムでは、自分のユーザー名とパスワードを使用して ORACLE に接続した後、SQL 文を入力します。埋込み型ではなく通常の SQL 構文を使用して有効な SQL または PL/SQL 文を入力し、各文の終わりにセミコロンを付けてください。入力した文が処理されます。問合せのときはフェッチされた行が表示されます。

複数行の文を入力できます。最大 1023 文字まで入力できます。変数のサイズには制限があり、MAX_VAR_LEN が 255 に定義されています。このプログラムでは、バインド変数は 40、選択リスト項目も 40 まで処理できます。DML 戻り句およびユーザー定義型は、値構文ではサポートされていません。

次のように mode = ansi に設定してプログラムをプリコンパイルします。

```
proc mode=ansi ansidyn1
```

mode=ansi に指定すると、動的および type_code が ANSI に設定されます。

```
/*  
ANSI Dynamic Demo 1:  ANSI Dynamic SQL with value semantics,  
                      literal descriptor names  
                      and ANSI type codes
```

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 via 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 semicolon. Your statement will be processed. If it is a query, the fetched rows are displayed.

You can enter multi-line 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 statements and user defined types are not supported with value semantics.

Precompile the program with mode=ansi, i.e:

```
proc mode=ansi ansidyn1
```

Using mode=ansi will set dynamic and type_code to ansi.

```
*/
```



```
#include <stdio.h>
#include <string.h>
#include <setjmp.h>
#include <stdlib.h>
#include <sqlcpr.h>

#define MAX_OCCURENCES 40
#define MAX_VAR_LEN 255
#define MAX_NAME_LEN 31

#ifdef NULL
#define NULL 0
#endif

/* Prototypes */
#ifdef __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;

        /* Prepare the statement and declare a cursor. */
        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);

        /* Open the cursor and execute the statement. */
        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, plsqli;

    for (plsqli = 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 ((strcmp(linebuf, "SELECT", 6) == 0) ||
            (strcmp(linebuf, "select", 6) == 0))
        {
            select_found=1;;
        }
    }
}
```

```

        if ((strcmp(linebuf, "EXIT", 4) == 0) ||
            (strcmp(linebuf, "exit", 4) == 0))
        {
            return -1;
        }

        else if (linebuf[0] == '?' ||
            (strcmp(linebuf, "HELP", 4) == 0) ||
            (strcmp(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;                                /* occurrence 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++)
    {
        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++)
        {
            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()
{
    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;

connect_error:
    fprintf(stderr, "Cannot connect to ORACLE as user %s\\n", username.arr);
    return -1;
}

```

ansidyn2.pc

このプログラムは、ANSI 動的 SQL を使用した SQL 文の処理方法を示します。この SQL 文は実行時まで不明です。このプログラムでは、バッチ処理および参照セマンティクスの Oracle 拡張機能が使用されます。

このプログラムでは、自分のユーザー名とパスワードを使用して ORACLE に接続した後、SQL 文を入力します。埋込み型ではなく対話型の SQL 構文を使用して有効な SQL または PL/SQL 文を入力し、各文の終わりにセミコロンを付けてください。入力した文が処理されます。問合せのときはフェッチされた行が表示されます。

複数行の文を入力できます。最大 1023 文字まで入力できます。変数のサイズには制限があり、MAX_VAR_LEN が 255 に定義されています。このプログラムでは、バインド変数は 40、選択リスト項目も 40 まで処理できます。

次のように dynamic = ansi に設定してプログラムをプリコンパイルします。

```
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 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.

```

```

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 statement 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 multi-line 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, i.e:

```



```

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

#ifdef NULL
#define NULL 0
#endif

/* Prototypes */
#ifdef __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;

int  in_array_size;          /* size of input batch, i.e., number of rows */
int  out_array_size;         /* size of input batch, i.e., 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();

    /* Allocate the input and output descriptors. */
    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;

        /* Prepare the statement and declare a cursor. */

```

```
    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);

    /* Open the cursor and execute the statement. */
    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, plsqli;
```

```

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 ((strcmp(linebuf, "SELECT", 6) == 0) ||
        (strcmp(linebuf, "select", 6) == 0))
    {
        select_found=1;;
    }

    if ((strcmp(linebuf, "EXIT", 4) == 0) ||
        (strcmp(linebuf, "exit", 4) == 0))
    {
        return -1;
    }

    else if (linebuf[0] == '?' ||
        (strcmp(linebuf, "HELP", 4) == 0) ||
        (strcmp(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; /* occurrence 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)    /* 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;
    occurs = i + 1; /* occurrence is one based */
    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++)
    for (i=0; i < output_count; i++)
        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++)
        {
            /* 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]);
            }
        }
    return(0);
}

void rows_processed()
{
    int i;
    for (i = 0; i < 8; i++)
    {
        if (strcmp(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 の動的 SQL 方法 4

この章では、Oracle 動的 SQL 方法 4 を実現する方法について説明します。この方法を使用すると、ホスト変数を含む動的 SQL 文を受け取ったり作成することができます。含まれるホスト変数の個数は様々です。この方法を使用して既存のアプリケーションをサポートしてください。新しいアプリケーションではすべて ANSI 動的 SQL 方法 4 を使用してください。

注意：動的 SQL 方法 1、2、3 の詳細および方法 4 の概要は、[第 13 章「Oracle の動的 SQL」](#)を参照してください。

Oracle 動的 SQL 方法 4 は、オブジェクト型、カーソル変数、構造体の配列、DML 戻り句、Unicode 変数および LOB をサポートしていません。かわりに ANSI 動的 SQL 方法 4 を使用してください。[第 14 章「ANSI 動的 SQL」](#)を参照してください。

この章のトピックは、次のとおりです。

- [方法 4 の特殊要件](#)
- [SQLDA の説明](#)
- [SQLDA 変数の使用](#)
- [予備知識](#)
- [基本ステップ](#)
- [各ステップの詳細](#)
- [サンプル・プログラム : 動的 SQL 方法 4](#)

方法 4 の特殊要件

方法 4 の必要条件を学習する前に、選択リスト項目とプレースホルダという用語に慣れておく必要があります。選択リスト項目とは、問合せ内でキーワード **SELECT** の後に続く列または式のことです。たとえば、次の動的問合せは 3 つの選択リスト項目を含んでいます。

```
SELECT ename, job, sal + comm FROM emp WHERE deptno = 20
```

プレースホルダはダミーのバインド変数です。プレースホルダは実際のバインド変数用に SQL 文内に場所を確保するためのものです。プレースホルダは宣言が不要で、任意の名前を指定できます。

バインド変数のプレースホルダは **SET** 句、**VALUES** 句および **WHERE** 句で最もよく使用されます。たとえば、次の動的 SQL 文はそれぞれ 2 つのプレースホルダを含んでいます。

```
INSERT INTO emp (empno, deptno) VALUES (:e, :d)
DELETE FROM dept WHERE deptno = :num OR loc = :loc
```

方法 4 が特別な理由

方法 1、2、3 とは異なり、動的 SQL 方法 4 ではプログラムで次のことができます。

- 選択リスト項目数とプレースホルダ数が不明な動的 SQL 文の受取りまたは作成
- Oracle と C の間のデータ型変換の明示的な制御

プログラムにこのような柔軟性をもたせるには、Oracle ランタイム・ライブラリに補足情報を追加する必要があります。

Oracle に必要な情報

Pro*C/C++ プリコンパイラはすべての実行可能な動的 SQL 文について Oracle コールを生成します。動的 SQL 文に選択リスト項目またはプレースホルダが指定されていない場合は、Oracle にはその文を実行するための補足情報は必要ありません。次の **DELETE** 文がこのカテゴリに該当します。

```
DELETE FROM emp WHERE deptno = 30
```

ただし、ほとんどの動的 SQL 文には、次の **UPDATE** 文のように、選択リスト項目、またはバインド変数のプレースホルダが含まれています。

UPDATE 文：

```
UPDATE emp SET comm = :c WHERE empno = :e
```

バインド変数のためのプレースホルダまたは選択リスト項目を含む動的 SQL 文を実行するには、入力（バインド）値および問合せ実行するときに FETCH された値を保持するプログラム変数についての情報が必要です。Oracle は次の情報を必要とします。

- バインド変数の数と選択リスト項目の数
- 各バインド変数と選択リスト項目の長さ
- 各バインド変数と選択リスト項目のデータ型
- 各バインド変数のアドレスと、各選択リスト項目の値が設定される出力変数のアドレス

情報の格納位置

選択リスト項目またはバインド変数のプレースホルダについて Oracle で必要な情報は、それらの値以外はすべて、SQL 記述子領域（SQLDA）というプログラム・データ構造体に格納されます。SQLDA 構造体は *sqlda.h* ヘッダー・ファイルに定義されています。

選択リスト項目の記述は選択記述子に格納されます。また、バインド変数に対するプレースホルダの記述はバインド記述子に格納されます。

選択リスト項目の値は出力変数に代入されます。これに対し、バインド変数の値は入力変数に代入されます。これらの変数のアドレスを選択 SQLDA またはバインド SQLDA に格納すると、出力値を書き込む位置および入力値を読み込む位置が Oracle に認識されます。

値はどのようにしてこれらのデータ変数に代入されるのでしょうか。出力値は、カーソルを使用して FETCH されます。入力値は、通常はユーザーが対話形式で入力した情報をもとに、プログラムによって代入されます。

SQLDA の参照方法

バインド記述子および選択記述子は、通常はポインタによって参照されます。動的 SQL プログラムでは、少なくとも 1 つのバインド記述子と 1 つの選択記述子のポインタを次のように宣言する必要があります。

```
#include <sqlda.h>
...
SQLDA *bind_dp;
SQLDA *select_dp;
```

この後に *SQLSQLDAAlloc()* 関数を使用すると、次のように記述子を割り当てることができます。

```
bind_dp = SQLSQLDAAlloc(runtime_context, size, name_length, ind_name_length);
```

Oracle8 以前のバージョンでは、*SQLSQLDAAlloc()* は *sqlldt()* に相当します。

定数 *SQL_SINGLE_RCTX* は、*(dvoid*)0* として定義されます。これはアプリケーションが単一のスレッドを扱う場合、*runtime_context* 用に使用します。

この機能および SQLLIB の他の機能の詳細は、表 15-3「SQL データ型の精度とスケール」を参照してください。SQLSQLDAAlloc() およびそのパラメータの詳細は、15-5 ページの「SQLDA の割当て」を参照してください。

情報の取得方法

DESCRIBE 文を使用すると、Oracle に必要な情報が得られます。

DESCRIBE SELECT LIST 文は、各選択リスト項目を検査して名前とその長さを判断します。次にこの情報を使用できるように選択 SQLDA に格納します。たとえば、選択リスト名を印刷時に列のヘッダーとして使用できます。選択リスト項目の合計数も DESCRIBE 文によって SQLDA に格納されます。

DESCRIBE BIND VARIABLES 文はそれぞれのプレースホルダを調べて、名前と長さを決定します。続いてこの情報を使用できるように入力バッファとバインド SQLDA に格納します。たとえば、プレースホルダ名を使用してバインド変数の値をユーザーに入力要求できます。

SQLDA の説明

この項では SQLDA のデータ構造を詳しく説明します。SQLDA の宣言方法、格納されている変数、初期化の方法、プログラム内での使用方法を理解できます。

SQLDA の用途

選択リスト項目の数またはバインド変数のプレースホルダの数が不明の動的 SQL 文には方法 4 を使用する必要があります。このような動的 SQL 文を処理するには、プログラムで SQLDA（記述子とも呼ばれる）を明示的に宣言する必要があります。記述子はそれぞれ**構造体**になっています。記述子はプログラムにコピーまたはハードコードする必要があります。

選択記述子は選択リスト項目の記述、および選択リスト項目の名前と値が格納されている出力バッファのアドレスを保持します。

注意： 選択リスト項目の " 名前 " は、列名、列の別名あるいは *sal+comm* などの表現テキストになります。

バインド記述子は、バインド変数と標識変数の記述、およびバインド変数と標識変数の名前と値が格納されている入力バッファのアドレスを保持します。

複数の SQLDA

プログラムにアクティブな動的 SQL 文が 2 つ以上ある場合は、それぞれの文が専用の SQLDA を持つ必要があります。別の名前で任意の数の SQLDA を宣言できます。たとえば、*sel_desc1*、*sel_desc2* および *sel_desc3* と名前を付けられた 3 つの選択 SQLDA を宣言すると、3 つの現在の OPEN カーソルから FETCH できます。ただし、カーソルが同時に実行されなければ SQLDA を再利用できます。

SQLDA の宣言

SQLDA を宣言するには、*sqllda.h* ヘッダー・ファイルを組み込みます。SQLDA の内容は、次のとおりです。

```
struct SQLDA
{
    long    N;           /* Descriptor size in number of entries */
    char **V;           /* Ptr to Arr of addresses of main variables */
    long    *L;          /* Ptr to Arr of lengths of buffers */
    short   *T;          /* Ptr to Arr of types of buffers */
    short **I;          /* Ptr to Arr of addresses of indicator vars */
    long    F;           /* Number of variables found by DESCRIBE */
    char **S;           /* Ptr to Arr of variable name pointers */
    short   *M;          /* Ptr to Arr of max lengths of var. names */
    short   *C;          /* Ptr to Arr of current lengths of var. names */
    char **X;           /* Ptr to Arr of ind. var. name pointers */
    short   *Y;          /* Ptr to Arr of max lengths of ind. var. names */
    short   *Z;          /* Ptr to Arr of cur lengths of ind. var. names */
};
```

SQLDA の割当て

SQLDA の宣言後、次の構文のライブラリ関数 *SQLSQLDAAlloc()* (Oracle8 以前のバージョンでは *sqlalldt()* に相当する) を使用して、記憶領域を割り当てます。

```
descriptor_name = SQLSQLDAAlloc (runtime_context, max_vars, max_name, max_ind_name);
```

パラメータは次のとおりです。

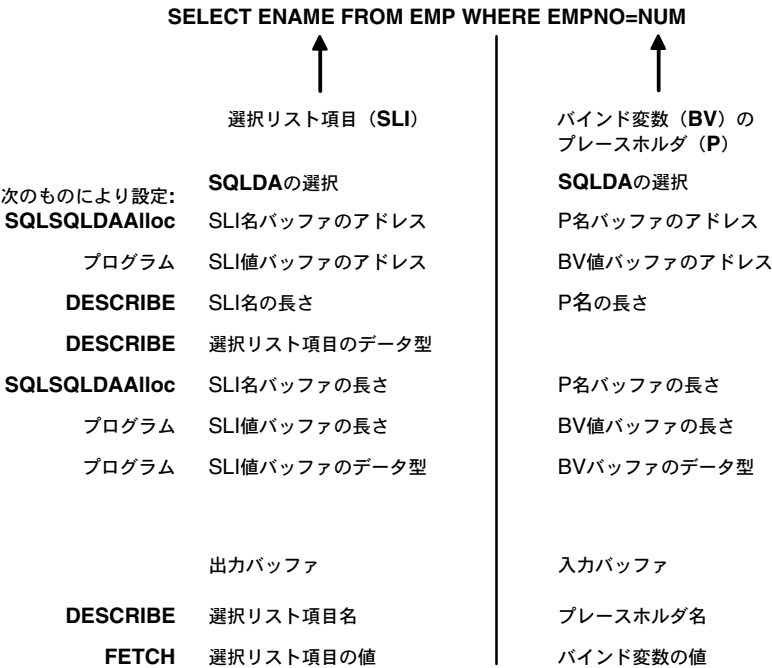
<i>runtime_context</i>	ランタイム・コンテキストへのポインタ。
<i>max_vars</i>	記述子が記述できる選択リスト項目またはプレースホルダの最大数。
<i>max_name</i>	選択リスト名またはプレースホルダ名の最大長。
<i>max_ind_name</i>	オプション指定でプレースホルダ名に付加される標識変数名の最大長。このパラメータはバインド記述子専用です。したがって、選択記述子を割り当てるときはこのパラメータを 0 (ゼロ) に設定します。

記述子の他にも、*SQLSQLDAAlloc()* は記述子変数が指すデータ・バッファも割り当てます。

SQLSQLDAAlloc() の詳細は、15-6 ページの「[SQLDA 変数の使用](#)」および 15-20 ページの「[記述子用の記憶領域の割当て](#)」を参照してください。

図 15-1 に、変数が `SQLSQLDAAlloc()` コール、`DESCRIBE` コマンド、`FETCH` コマンドあるいはプログラム割当てのどの方法で設定されるかを示します。

図 15-1 変数の設定方法



SQLDA 変数の使用

この項では、SQLDA 内の各変数の用途と使用方法を説明します。

N 変数

`N` は、`DESCRIBE` 可能な選択リスト項目またはブレースホルダの最大数を指定します。つまり、`N` によって記述子配列に含まれる要素の数が決まります。

オプションの `DESCRIBE` コマンドを発行する前に、ライブラリ関数 `SQLSQLDAAlloc()` を使用して `N` を記述子配列のディメンションに設定する必要があります。`DESCRIBE` コマンドの発行後は、`N` を `DESCRIBE` された変数の実際の数に再設定する必要があります。この数は `F` 変数に格納されています。

V 変数

V は、選択リストまたはバインド変数の値を格納するデータ・バッファのアドレスからなる配列のポインタです。

記述子を割り当てると、`SQLSQLDAAlloc()` によってアドレスの配列にある `V[0]` から `V[N-1]` の要素が 0 (ゼロ) に設定されます。

選択記述子の場合、FETCH コマンドを発行する前にデータ・バッファを割り当て、この配列を設定する必要があります。例を次に示します。

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

この文は FETCH された選択リストの値を、`V[0]` から `V[N-1]` が指しているデータ・バッファに格納するように Oracle に指示します。Oracle は *i* 番目の選択リストの値を、`V[i]` が指しているデータ・バッファに格納します。

バインド記述子の場合、OPEN コマンドを発行する前に、この配列を設定する必要があります。例を次に示します。

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

この文は、`V[0]` から `V[N-1]` が指しているバインド変数の値を使用して動的 SQL 文を実行するよう Oracle に指示します。Oracle は、`V[i]` が指し示しているデータ・バッファで *i* 番目のバインド変数の値を参照します。

L 変数

L は、データ・バッファに格納されている選択リストまたはバインド変数の値の長さからなる配列のポインタです。

選択記述子の場合、DESCRIBE SELECT LIST によって、長さの配列は各選択リスト項目に予想される最大値に設定されます。しかし、FETCH コマンドを発行する前に長さを再設定することもあります。FETCH は、最大 *n* 文字を戻します。この場合、*n* は FETCH を発行する前の `L[i]` の値です。

長さの形式は Oracle データ型によって異なります。CHAR または VARCHAR2 の選択リスト項目については、DESCRIBE SELECT LIST は `L[i]` を選択リスト項目の最大長に設定します。NUMBER 型の選択リスト項目については、スケールと精度が変数の下位バイトとその次の上位側バイトにそれぞれ戻されます。精度とスケールの値を `L[i]` から抽出するには、ライブラリ関数 `SQLNumberPrecV6()` を使用できます。詳細は 15-15 ページの「[精度とスケールの抽出](#)」を参照してください。

FETCH する前に、`L[i]` を必要なデータ・バッファの長さに再設定する必要があります。たとえば、NUMBER を C の `char` 文字列に強制変換する場合、`L[i]` を符号と小数点の精度の数値に 2 を加えたものに設定します。NUMBER 型を C の `float` 型に強制変換する場合は、`L[i]` をシステム上の `float` 型の長さに設定します。強制変換したデータ型の長さの詳細は、15-11 ページの「[データの変換](#)」を参照してください。

バインド記述子の場合、OPEN コマンドを発行する前に、配列の長さを設定する必要があります。たとえば、`strlen()` を使用してユーザーが入力したバインド変数文字列の長さを取得してから、適切な配列要素を設定します。

Oracle は `V[i]` に格納されているアドレスを使用して間接的にデータ・バッファにアクセスします。このためバッファ内の値の長さは Oracle には認識されません。*i* 番目の選択リストまたはバインド変数の値について Oracle が使用する長さを変更するときは、`L[i]` を必要な長さに再設定してください。入力バッファまたは出力バッファにはそれぞれ異なる長さを指定できます。

T 変数

T は、選択リストまたはバインド変数の値のデータ型コードからなる配列のポインタです。これらのデータ型コードは、V 配列の要素が指示するデータ・バッファに Oracle データが格納されるときデータの変換方法を決定します。詳細は、15-11 ページの「[データの変換](#)」を参照してください。

選択記述子の場合、DESCRIBE SELECT LIST はデータ型コードの配列を選択リスト内の項目の内部データ型 (CHAR、NUMBER、DATE など) に設定します。

Oracle データ型の内部形式は処理が複雑なため、FETCH する前にデータ型をいくつか再設定する必要がある場合があります。表示用データのときは、一般には選択リストの値のデータ型を VARCHAR2 または STRING に強制変換することをお勧めします。計算用データのときは、Oracle の数値を C の形式に強制変換する必要がある場合があります。詳細は 15-14 ページの「[データ型の強制変換](#)」を参照してください。

`T[i]` の上位ビットの設定は、*i* 番目の選択リスト項目の NULL/NOT NULL ステータスを示します。OPEN コマンドまたは FETCH コマンドを発行する前に、常にこのビットをオフにする必要があります。データ型コードを取り出し、NULL/NOT NULL ビットを消去するには、ライブラリ関数 `SQLColumnNullCheck()` を使用します。詳細は 15-16 ページの「[NULL/NOT NULL データ型の処理](#)」を参照してください。

Oracle の NUMBER 内部データ型は、`V[i]` が指示する C のデータ・バッファと互換性のある外部データ型に変更する必要があります。

バインド記述子の場合、データ型コードの配列は DESCRIBE BIND VARIABLES によって 0 に設定されます。OPEN コマンドを発行する前に、各要素に格納されたデータ型を設定する必要があります。コードは、`V[i]` が指すデータ・バッファの外部 (C) データ型を表します。バインド変数の値が文字列に格納され、データ型配列の要素が 1 (VARCHAR2 データ型コード) に設定されることがよくあります。データ型コード 5 (STRING) を使用することもできます。

i 番目の選択リストまたはバインド変数の値のデータ型を変更するには、`T[i]` を変更するデータ型に再設定してください。

I 変数

I は標識変数を格納するデータ・バッファのアドレスの配列へのポインタです。

アドレスの配列の *I*[0] から *I*[*N* - 1] の要素を設定する必要があります。

選択記述子の場合、FETCH コマンドを発行する前に、アドレスの配列を設定する必要があります。Oracle が次の文を実行する際、

```
EXEC SQL FETCH ... USING DESCRIPTOR ...
```

戻された *i* 番目の選択リスト値が NULL の場合は、*I*[*i*] が指す標識変数値が -1 に設定されます。それ以外の場合は、0 (値が NULL でない) または正の整数 (値が切り捨てられている) に設定されます。

バインド記述子の場合、OPEN コマンドを発行する前に、アドレスの配列とそれに対応付けられた標識変数を設定する必要があります。Oracle が次の文を実行する際、

```
EXEC SQL OPEN ... USING DESCRIPTOR ...
```

I[*i*] が指しているデータ・バッファは *i* 番目のバインド変数の値が NULL であるかどうかを決定します。標識変数の値が -1 のとき、関連するバインド変数の値は NULL です。

F 変数

F は、DESCRIBE によって検出される選択リスト項目またはプレースホルダの実際の個数です。

F は DESCRIBE によって設定されます。*F* の値が負のときは、割り当てられた記述子のサイズに対して DESCRIBE が検出した選択リスト項目またはプレースホルダが多すぎることを示しています。たとえば、*N* を 10 に設定したときに DESCRIBE で 11 個の選択リスト項目またはプレースホルダが検出されると、*F* は -11 に設定されます。この機能を使用すると、必要に応じて選択リスト項目またはプレースホルダに大きい記憶領域を動的に再割り当てすることができます。

S 変数

S は、データ・バッファのアドレスの配列へのポインタです。動的 SQL 文で選択リストまたはプレースホルダの名前が見つかったとそれらをそのデータ・バッファに格納します。

SQLSQLDAAlloc() を使用してデータ・バッファを割り当ててから、*S* 配列にそれらのアドレスを格納します。

DESCRIBE は *S*[*i*] が指すデータ・バッファ内に *i* 番目の選択リスト項目またはプレースホルダの名前を格納するように Oracle に指示します。

M 変数

M は、選択リストまたはプレースホルダの名前を格納するデータ・バッファの最大長からなる配列へのポインタです。このデータ・バッファのアドレスは、S 配列の要素によって指定されます。

記述子を割り当てると、`SQLSQLDAAlloc()` は要素 `M[0]` から `M[N - 1]` までを最大長の配列に設定します。`S[i]` が指しているデータ・バッファに格納するとき、*i* 番目の名前は必要に応じて `M[i]` の長さに切り捨てられます。

C 変数

C は、選択リストまたはプレースホルダの名前の現行の長さからなる配列へのポインタです。

DESCRIBE は、現行の長さの配列の中に `C[0]` から `C[N - 1]` の要素を設定します。DESCRIBE 文を実行すると、配列には選択リスト名またはプレースホルダ名の文字数が格納されます。

X 変数

X は、標識変数の名前を格納するデータ・バッファのアドレスからなる配列へのポインタです。標識変数の値を選択リスト項目およびバインド変数に関連付けることができます。ただし、標識変数の名前はバインド変数にしか関連付けられません。したがって X はバインド記述子専用です。

データ・バッファを割り当て、そのアドレスを X 配列に保存するには、`SQLSQLDAAlloc()` を使用します。

DESCRIBE BIND VARIABLE は、*i* 番目の標識変数の名前を `X[i]` が指すデータ・バッファに格納するように Oracle に指示します。

Y 変数

Y は、標識変数の名前を格納するデータ・バッファの最大長からなる配列へのポインタです。X と同様に、Y もバインド記述子専用です。

`SQLSQLDAAlloc()` を使用して要素 `Y[0]` から `Y[N - 1]` までを最大長の配列に割り当てます。`Y[i]` が指しているデータ・バッファに格納するとき、*i* 番目の名前は必要に応じて `Y[i]` の長さに切り捨てられます。

Z 変数

Z は、標識変数の名前の現行の長さからなる配列へのポインタです。X および Y と同様に、Z もバインド記述子専用です。

DESCRIBE BIND VARIABLES は、現行の長さの配列に `Z[0]` から `Z[N - 1]` の要素を設定します。DESCRIBE を実行すると、それぞれの標識変数名の文字数がこの配列に格納されます。

予備知識

動的 SQL 方法 4 を実現するには次の処理についての知識が必要です。

- データの変換
- データ型の強制変換
- NULL/NOT NULL データ型の処理

データの変換

この項では *T* (データ型) の記述子配列について詳しく説明します。データ型の同値化と動的 SQL 方法 4 のどちらも使用しないホスト・プログラムでは、Oracle の内部データ型と外部データ型との変換方法はプリコンパイル時に決定されます。デフォルトでは、プリコンパイラは宣言文内のそれぞれのホスト変数に特定の外部データ型を割り当てます。たとえば、プリコンパイラは *int* 型のホスト変数に INTEGER 外部データ型を割り当てます。

しかし方法 4 を使用すると、データの変換および形式を制御できます。データの変換方法を指定するには、*T* 記述子配列にデータ型コードを設定します。

内部データ型

内部データ型は、Oracle がデータベース表に列値を格納するための形式と、疑似列値を表すための形式を指定します。

DESCRIBE SELECT LIST コマンドを発行すると、Oracle はそれぞれの選択リスト項目に対する内部データ型コードを *T* 記述子配列に戻します。たとえば、*i* 番目の選択リスト項目に対するデータ型コードは *T[i]* に戻されます。

表 15-1 に、Oracle の内部データ型とそのコードを示します。

表 15-1 Oracle 内部データ型

Oracle 内部データ型	コード
VARCHAR2	1
NUMBER	2
LONG	8
ROWID	11
DATE	12
RAW	23
LONG RAW	24
CHARACTER (または CHAR)	96

外部データ型

外部データ型には、入力ホスト変数および出力ホスト変数に値を格納するのに使用する形式を指定します。

DESCRIBE BIND VARIABLES コマンドはデータ型コードの T 配列を 0（ゼロ）に設定します。このため、OPEN コマンドを発行する前にそれらのコードを再設定する必要があります。これらのコードは、各種バインド変数について想定される外部データ型を Oracle に指示します。 i 番目のバインド変数については、必要な外部データ型を $T[i]$ に再設定してください。

表 15-2 に、Oracle の外部データ型とそのコード、および各外部データ型で通常使用する C のデータ型を示します。

表 15-2 Oracle の外部データ型とデータ型コード

外部データ型	コード	C データ型
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
LONG	8	char[n]
VARCHAR	9	char[n+2]
ROWID	11	char[n]
DATE	12	char[n]
VARRAW	15	char[n]
RAW	23	符号なし char[n]
LONG RAW	24	符号なし char[n]
UNSIGNED	68	符号なし INT
DISPLAY	91	char[n]
LONG VARCHAR	94	char[n+4]
LONG VARRAW	95	符号なし char[n+4]
CHAR	96	char[n]
CHARF	96	char[n]
CHARZ	97	char[n+1]

Oracle のデータ型とその書式の詳細は、このガイド 4-2 ページの「[Oracle のデータ型](#)」および『Oracle8i SQL リファレンス』を参照してください。

データ型の強制変換

選択記述子の場合、`DESCRIBE SELECT LIST` は Oracle8 の内部データ型をどれでも戻すことができます。文字データの場合と同じく、内部データ型は使用する外部データ型と正確に対応している場合が多くあります。ただし、外部データ型にマップされるいくつかの内部データ型は処理が困難です。そのため、`T` 記述子配列の一部の要素を再設定する必要がある場合があります。たとえば、`NUMBER` 値を `C` の浮動小数点数に相当する `float` 値にリセットする場合があります。Oracle は、内部データ型と外部データ型の間の変換に必要な処理を `FETCH` 時に行います。このため、データ型のリセットは必ず `DESCRIBE SELECT LIST` の後、`FETCH` の前に行ってください。

バインド記述子の場合、`DESCRIBE BIND VARIABLES` によってバインド変数のデータ型は戻されません。バインド変数の数と名前のみが戻されます。したがって、データ型コードの `T` 配列を明示的に設定することによって、それぞれのバインド変数の外部データ型を Oracle に通知する必要があります。Oracle は、`OPEN` 時に外部データ型と内部データ型間で必要な変換をすべて実行します。

`T` 記述子配列でデータ型コードをリセットすると、「データ型を強制変換」することになります。たとえば、 i 番目の選択リスト値を `STRING` に強制変換するには、次の文を使用します。

```
/* Coerce select-list value to STRING. */
select_des->T[i] = 5;
```

データ表示用に `NUMBER` の選択リスト値を `STRING` に強制変換するときは、値の精度とスケールのバイトを抽出し、それらを使用して最大表示長を算出する必要もあります。`FETCH` の前に、`L` (長さ) 記述子配列の該当する要素をリセットし、使用するバッファの長さを Oracle に通知する必要があります。詳細は 15-15 ページの「[精度とスケールの抽出](#)」を参照してください。

たとえば、`DESCRIBE SELECT LIST` によって i 番目の選択リスト項目のデータ型が `NUMBER` 型であると確認し、このとき `float` 型で宣言されている `C` 変数に戻り値を格納する場合は、`T[i]` には 4 を、`L[i]` にはシステムが定める `float` の長さを設定するのみです。

注意

`DESCRIBE SELECT LIST` によって戻される内部データ型が、期待する結果と合わない場合があります。`DATE` 型と `NUMBER` 型がその例です。`DATE` 型の選択リスト項目を `DESCRIBE` すると、Oracle はデータ型コード 12 を `T` 記述子配列に戻します。`FETCH` の前にコードをリセットしないかぎり、日付の値はその 7 バイト内部形式で戻されます。日付を文字形式 (`DD-MON-YY`) で取得するには、12 に設定されているデータ型コードを 1 (`VARCHAR2`) または 5 (`STRING`) に変更し、7 に設定されている `L` 値を 9 または 10 に増やします。

`NUMBER` 型の選択リスト項目を同じ要領で `DESCRIBE` すると、Oracle はデータ型コード 2 を `T` 配列に戻します。`FETCH` の前にコードをリセットしないかぎり、数値はその内部形式で戻されるので、おそらく求めている値とは異なります。そのときは、2 に設定されているコードを 1 (`VARCHAR2`) または 3 (`INTEGER`)、4 (`FLOAT`)、5 (`STRING`)、あるいはその他の適切なデータ型に変更します。

精度とスケールの抽出

ライブラリ関数 `SQLNumberPrecV6()` (従来の `sqlnvl()`) は、精度とスケールを抽出します。一般には、この関数は `DESCRIBE SELECT LIST` の後に使用します。その最初の引数は `L[i]` です。次の構文で、`SQLNumberPrecV6()` をコールします。

```
SQLNumberPrecV6(dvoid *runtime_context, long *length, int *precision, int *scale);
```

注意： プラットフォームの正しいプロトタイプは、プラットフォーム固有の `SQLNumberPrecV6` ヘッダー・ファイルを参照してください。

パラメータは次のとおりです。

<i>runtime_context</i>	ランタイム・コンテキストへのポインタ。
<i>length</i>	Oracle の NUMBER 値を保存する長い整数へのポインタ。長さは <code>L[i]</code> に格納されます。値のスケールと精度は低バイトと次の上位バイトにそれぞれ格納されます。
<i>precision</i>	NUMBER 値の精度を戻す整数へのポインタ。精度とは有効桁数を指します。選択リスト項目がサイズの指定されていない NUMBER を参照している場合、精度は 0 (ゼロ) に設定されます。このときはサイズ指定がないため、最大の精度 (38) を想定してください。
<i>scale</i>	NUMBER 値のスケールを戻す整数へのポインタ。スケールには四捨五入する位置を指定します。たとえばスケールが 2 のときは、1/100 の倍数の近似値に値が四捨五入される (3.456 は 3.46 になる) ことを意味します。またスケールが -3 のときは、1000 の倍数の近似値に値が四捨五入される (3456 が 3000 になる) ことを意味します。

スケールが負の場合は、その絶対値を長さに追加してください。たとえば、精度に 3、スケールに -2 を指定すると、99900 までの値が有効になります。

次の例に、`SQLNumberPrecV6()` を使用して、STRING に強制変換する NUMBER 値の最大値表示長を計算する方法を示します。

```
/* Declare variables for the function call. */
sqllda      *select_des; /* pointer to select descriptor */
int          prec;        /* precision */
int          scal;        /* scale */
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;
```

この関数コールの最初の引数は長さの配列の *i* 番目の要素を指します。また、パラメータは 3 つともすべてアドレスなので注意してください。

`SQLNumberPrecV6()` 関数では、一部の SQL データ型の精度とスケールの値に 0（ゼロ）が戻されます。`SQLNumberPrecV7()` 関数も同様に、次に示す SQL データ型の場合を除けば引数リストも戻り値も同じです。

表 15-3 SQL データ型の精度とスケール

SQL データ型	2 進数精度	スケール
FLOAT	126	-127
FLOAT(N)	N（範囲は 1 ～ 126）	-127
REAL	63	-127
DOUBLE PRECISION	126	-127

NULL/NOT NULL データ型の処理

すべての選択リスト列（式は不可）について、`DESCRIBE SELECT LIST` は選択記述子のデータ型配列 *T* に `NULL/NOT NULL` 標識を戻します。*i* 番目の選択リスト列に `NOT NULL` 制約が指定されていると、*T[i]* の上位ビットはオフにされます。それ以外の場合は上位ビットが設定されます。

`OPEN` 文または `FETCH` 文でデータ型を使用する前に、すでに `NULL/NOT NULL` ビットが設定されているときは、そのビットをオフにする必要があります。（このビットは絶対にオンにしないでください。）

列に `NULL` が有効かどうかを調べデータ型の `NULL/NOT NULL` ビットを消去するには、ライブラリ関数 `SQLColumnNullCheck()`（従来の `sqlnul()`）を使用します。次の構文で、`SQLColumnNullCheck()` を呼び出します。

```
SQLColumnNullCheck(dvoid *context, unsigned short *value_type,
    unsigned short *type_code, int *null_status);
```

パラメータは次のとおりです。

- `context` ランタイム・コンテキストへのポインタ。
- `value_type` 選択リスト列のデータ型コードを格納する符号なし short integer 変数へのポインタ。データ型は *T[i]* に格納されます。

<i>type_code</i>	選択リスト列のデータ型コードを戻す符号なし short integer 変数へのポインタ。上位ビットはオフにされています。
<i>null_status</i>	選択リスト列の NULL 状態を戻す integer 変数へのポインタ。1 は列が NULL を許可し、0 は許可しないことを意味します。

次の例に、`SQLColumnNullCheck()` の使用方法を示します。

```
/* Declare variables for the function call. */
sqlda *select_des; /* pointer to select descriptor */
unsigned short dtype; /* datatype without null bit */
int nullok; /* 1 = null, 0 = not null */
extern void SQLColumnNullCheck(); /* Declare library function. */
/* Find out whether column is not null. */
SQLColumnNullCheck(SQL_SINGLE_RCTX, (unsigned short *)&(select_des->T[i]), &dtype,
&nullok);
if (nullok)
{
    /* Nulls are allowed. */
    ...
    /* Clear the null/not null bit. */
    SQLColumnNullCheck(SQL_SINGLE_RCTX, &(select_des->T[i]), &(select_des->T[i]),
&nullok);
}
```

`SQLColumnNullCheck()` 関数の 2 回目のコールで指定されている 1 番目と 2 番目の引数は、データ型配列の *i* 番目の要素を指します。また、パラメータは 3 つともすべてアドレスであることに注意してください。

基本ステップ

方法 4 はあらゆる動的 SQL 文に使用できます。次の例には問合せの処理が示されているので、入力ホスト変数と出力ホスト変数の両方の処理方法が理解できます。

このサンプル・プログラムでは次のステップに従って動的問合せを処理します。

1. 問合せのテキストを保持するためのホスト文字列を宣言文で宣言します。
2. 選択 SQLDA とバインド SQLDA を宣言します。
3. 選択記述子とバインド記述子に対する記憶領域を割り当てます。
4. DESCRIBE できる選択リスト項目とプレースホルダの最大数を設定します。
5. 問合せのテキストをホスト文字列に設定します。
6. ホスト文字列から問合せを PREPARE します。

7. 問合せ用の（FOR）カーソルを DECLARE します。
8. バインド記述子に（INTO）バインド変数を DESCRIBE します。
9. プレースホルダの最大数を DESCRIBE によって実際に検出された数に再設定します。
10. DESCRIBE で検出されたバインド変数の値を取得し、それらの変数に対する記憶領域を割り当てます。
11. バインド記述子を使用して（USING）カーソルを OPEN します。
12. 選択記述子に（INTO）選択リストを DESCRIBE します。
13. 選択リスト項目の最大数を DESCRIBE によって実際に検出された数にリセットします。
14. 表示用にそれぞれの選択リスト項目の長さとデータ型をリセットします。
15. 選択記述子が指している割当て済のデータ・バッファに（INTO）データベースの行を FETCH します。
16. FETCH によって戻された選択リストの値を処理します。
17. 選択リスト項目、プレースホルダ、標識変数および記述子に対する記憶領域の割当てを解除します。
18. カーソルを CLOSE します。

注意：動的 SQL 文に含まれる選択リスト項目またはプレースホルダの数が明確である場合、一部の手順は必要ありません。

各ステップの詳細

この項ではそれぞれのステップを詳しく説明します。また章の終わりには、方法 4 を使用したコメント付きの完全なプログラム例を示します。

方法 4 では、埋込み SQL 文を次のような順序で使用します。

```
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;
```

動的問合せの選択リスト項目の数が明確な場合は、`DESCRIBE SELECT LIST` を省略するとともに次の方法 3 の `FETCH` 文を使用できます。

```
EXEC SQL FETCH cursor_name INTO host_variable_list;
```

または、動的 SQL 文のバインド変数に対するプレースホルダの数が明確な場合は、`DESCRIBE BIND VARIABLES` を省略するとともに次の方法 3 の `OPEN` 文を使用できます。

```
EXEC SQL OPEN cursor_name [USING host_variable_list];
```

次にこれらの文によって、ホスト・プログラムで記述子を使用して動的 SQL 文を受け入れ、それ进行处理する方法を説明します。

注意：次の説明には、図を使用します。図が複雑になるのを避けるために、次の条件を満たす必要があります。

- 記述子配列は 3 要素までに制限します。
- 名前の最大長は 5 文字までに制限します。
- 値の最大長は 10 文字までに制限します。

ホスト文字列の宣言

プログラムには、動的 SQL 文のテキストを格納するためのホスト変数が必要です。ホスト変数（ここでは `select_stmt`）は文字列として宣言する必要があります。

```
...  
int      emp_number;  
VARCHAR  emp_name [10];  
VARCHAR  select_stmt [120];  
float    bonus;
```

SQLDA の宣言

ここでは、SQLDA のデータ構造体をハードコード化するかわりに、次のように `INCLUDE` を使用することによって SQLDA をプログラムにコピーします。

```
#include <sqlda.h>
```

問合せに含まれる選択リスト項目の数またはバインド変数のプレースホルダの数が不明なので、次のように選択記述子とバインド記述子のポインタを宣言します。

```
sqlda *select_des;  
sqlda *bind_des;
```

記述子用の記憶領域の割当て

記述子用の記憶領域を割り当てるには、`SQLSQLDAAlloc()` ライブラリ関数を使用することを思い出してください。ANSI C 表記法による構文は次のとおりです。

```
SQLDA *SQLSQLDAAlloc(dvoid *context, unsigned int max_vars, unsigned int max_name,
unsigned int max_ind_name);
```

`SQLSQLDAAlloc()` 関数は、記述子構造体およびポインタ変数 `V`、`L`、`T`、`I` によってアドレスされる配列を割り当てます。

`max_name` が 0（ゼロ）以外の場合は、ポインタ変数 `S`、`M`、`C` によってアドレスされる配列が割り当てられます。`max_ind_name` が 0（ゼロ）以外の場合は、ポインタ変数 `X`、`Y`、`Z` によってアドレスされる配列が割り当てられます。`max_name` と `max_ind_name` が 0（ゼロ）の場合は、領域は割り当てられません。

`SQLSQLDAAlloc()` は成功すると、構造体のポインタを戻します。`SQLSQLDAAlloc()` は失敗すると、0（ゼロ）を戻します。

この例では選択記述子とバインド記述子を次のように割り当てます。

```
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);
```

選択記述子には、`X` によってアドレスされる配列に領域が割り当てられないようにするため、常に `max_ind_name` を 0（ゼロ）に設定します。

DESCRIBE への最大数の設定

DESCRIBE できる選択リスト項目またはプレースホルダの最大数を次のように設定します。

```
select_des->N = 3;
bind_des->N = 3;
```

図 15-2 と図 15-3 に、結果として得られる記述子を示します。

注意： 選択記述子の場合（図 15-2）、標識変数名の選択は消されて使用されないことを示します。

図 15-2 初期化された選択記述子

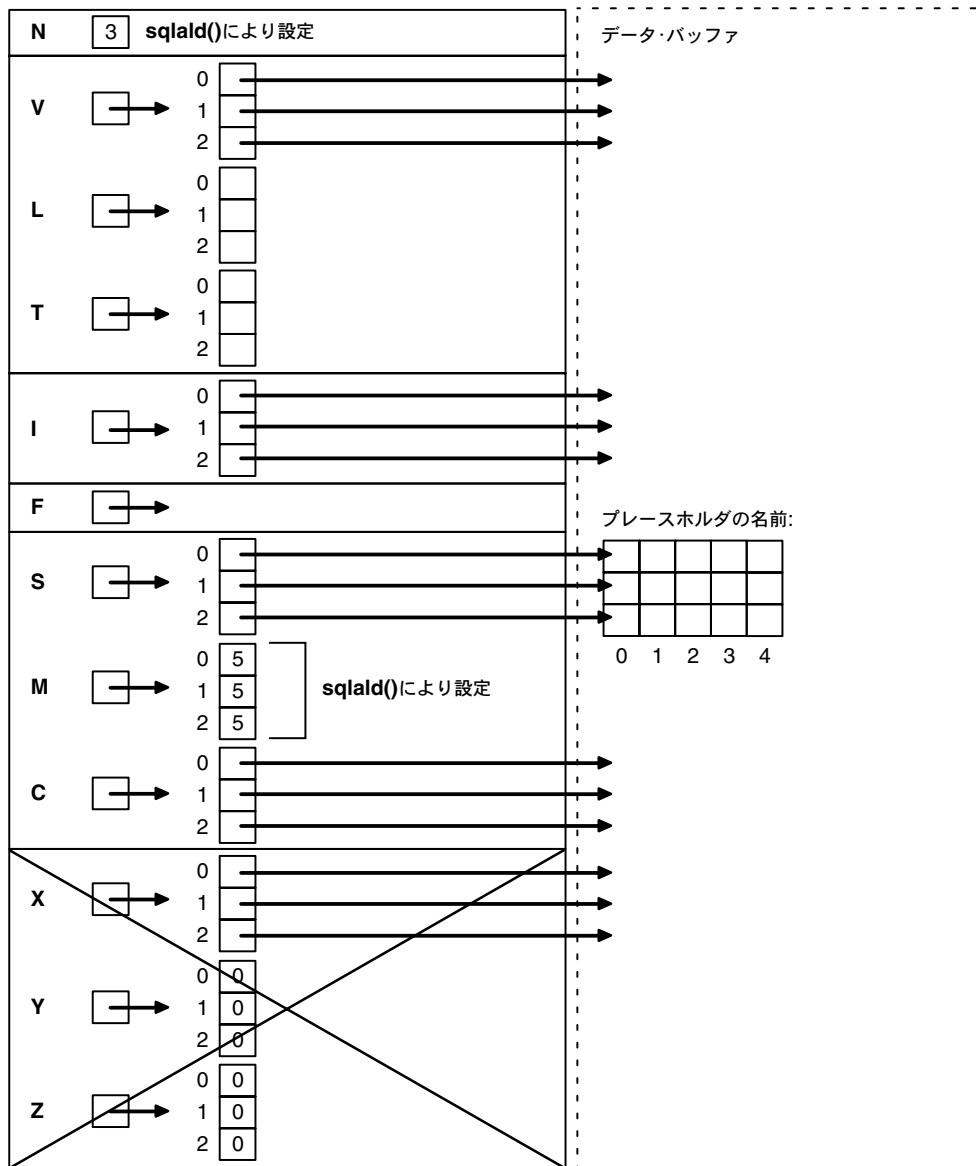
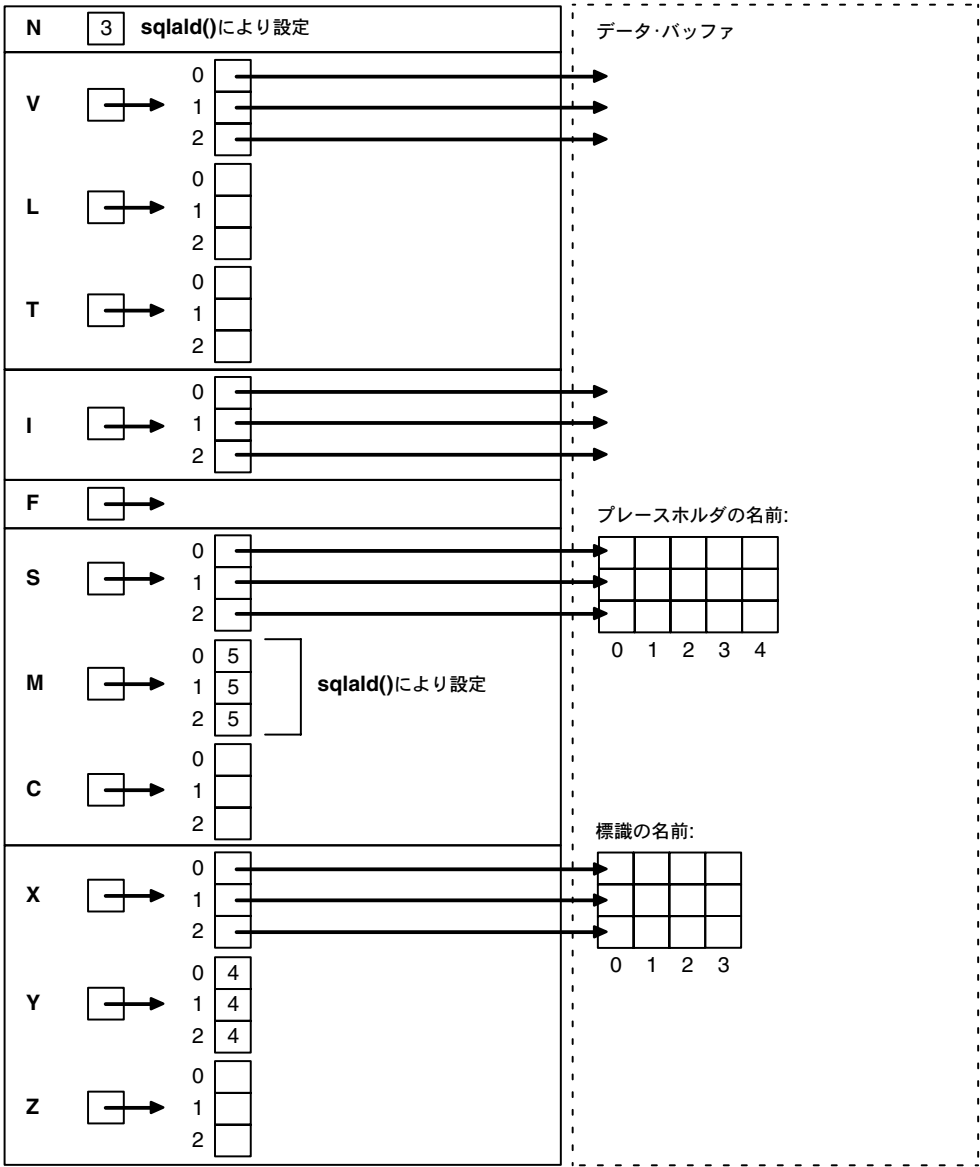


図 15-3 初期化されたバインド記述子



ホスト文字列への問合せテキストの設定

次の例では、ユーザーに SQL 文の入力を求め、入力された文字列を *select_stmt* に格納します。

```
printf("\n\nEnter SQL statement: ");
gets(select_stmt.arr);
select_stmt.len = strlen(select_stmt.arr);
```

このときユーザーが次の文字列を入力したと仮定しています。

```
"SELECT ename, empno, comm FROM emp WHERE comm < :bonus"
```

ホスト文字列からの問合せの PREPARE

PREPARE はこの SQL 文を解析して名前を指定します。例では、PREPARE はホスト文字列 *select_stmt* を解析してから、それに *sql_stmt* という名前を指定します。

```
EXEC SQL PREPARE sql_stmt FROM :select_stmt;
```

カーソルの宣言

DECLARE CURSOR は名前を指定し、特定の SELECT 文に対応付けることによって、カーソルを定義します。

静的問合せ用のカーソルを宣言するには次の構文を使用します。

```
EXEC SQL DECLARE cursor_name CURSOR FOR SELECT ...
```

動的問合せのカーソルを宣言する場合は、静的問合せのかわりに、PREPARE によって動的問合せに付けられた文の名前を指定します。例では、DECLARE CURSOR は *emp_cursor* という名前のカーソルを定義し、このカーソルを *sql_stmt* に対応付けます。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR sql_stmt;
```

注意：問合せのみでなく、すべての動的 SQL 文のカーソルを宣言できます。また、問合せ以外の場合も、カーソルの OPEN によって動的 SQL 文を実行します。

バインド変数の DESCRIBE

DESCRIBE BIND VARIABLES は、バインド記述子にプレースホルダの記述を設定します。例では、DESCRIBE は次のように *bind_des* を準備します。

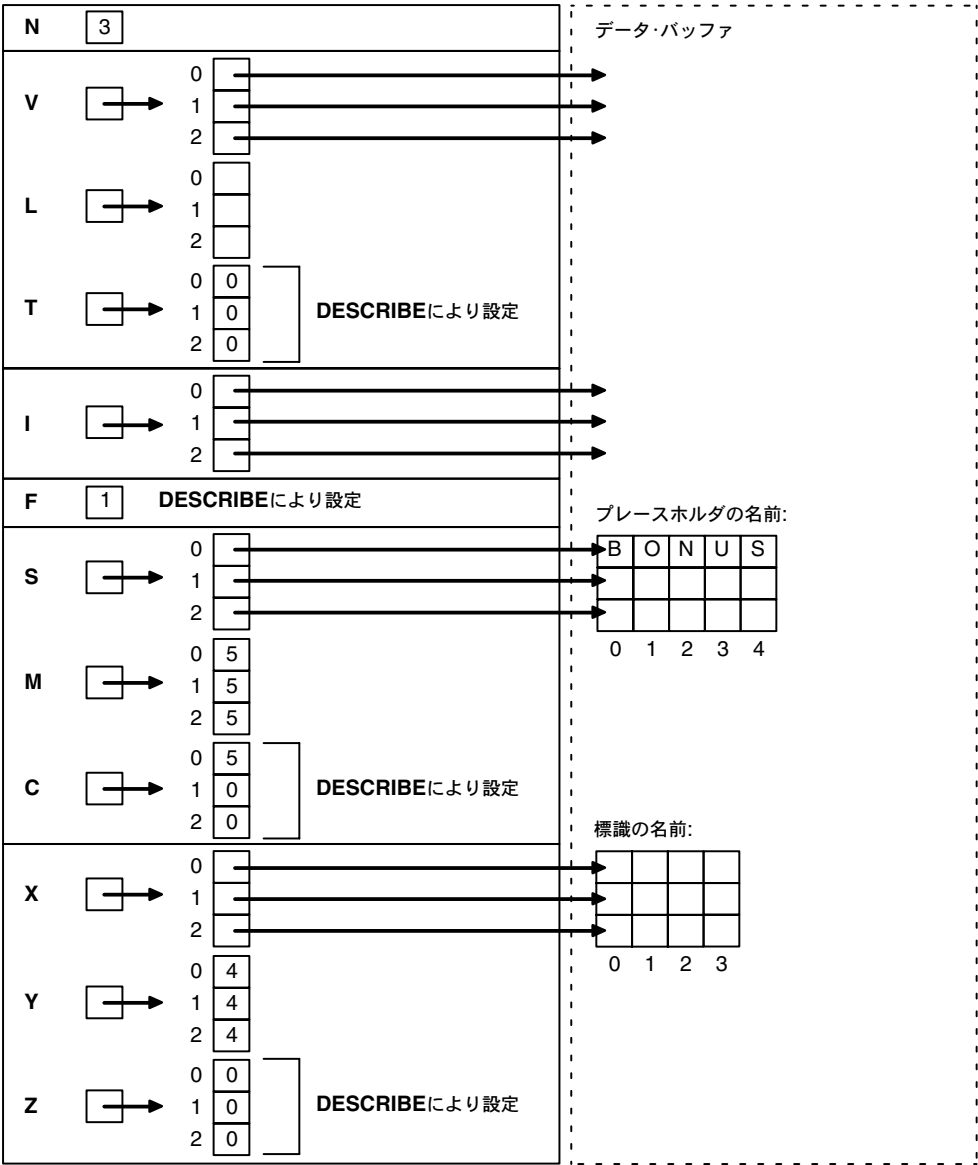
```
EXEC SQL DESCRIBE BIND VARIABLES FOR sql_stmt INTO bind_des;
```

bind_des はコロンで始めることはできません。

DESCRIBE BIND VARIABLES 文は PREPARE 文の後、かつ OPEN 文の前に指定する必要があります。

図 15-4 に、DESCRIBE 実行後のバインド記述子を示します。SQL 文の実行で検出されたプレースホルダの実際の数が、DESCRIBE によって F に設定されています。

図 15-4 DESCRIBE 後のバインド記述子



ブレースホルダの最大数のリセット

次に、ブレースホルダの最大数を DESCRIBE によって実際に検出された数にリセットする必要があります。

```
bind_des->N = bind_des->F;
```

バインド変数の値の取得と記憶領域の割当て

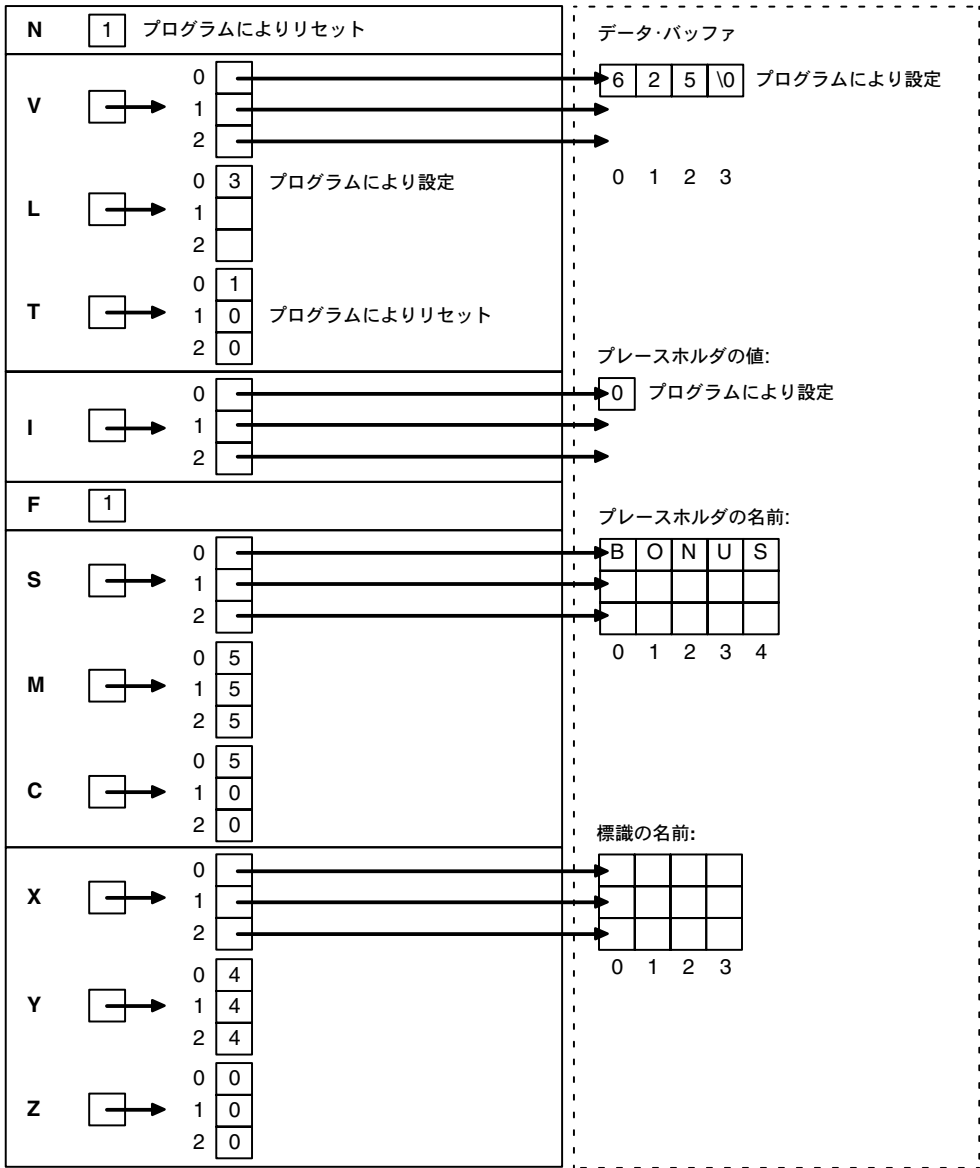
プログラムでは、SQL 文で検出されたバインド変数に対する値を取得し、メモリーを割り当てる必要があります。値はどのように取得してもかまいません。たとえば値をハードコードしたり、ファイルから読み込んだり、または対話形式で入力することもできます。

例では、問合せの WHERE 句のブレースホルダ *bonus* に置換されるバインド変数に値を割り当てる必要があります。そこで、ユーザーに値の入力を求め、入力された値を次のように処理します。

```
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;
}
```

ここでは、ユーザーが *bonus* の値として 625 と入力したと想定します。図 15-5 に、結果として得られるバインド記述子を示します。値は NULL で終わります。

図 15-5 値を割り当てた後のバインド記述子



カーソルの OPEN

動的問合せに使用する OPEN 文は、カーソルがバインド記述子に対応付けられることを除けば静的問合せに使用するものと同じです。実行時に決定され、バインド記述子表の要素でアドレス指定したバッファに格納された値を使用して、SQL 文を評価します。問合せの場合は、アクティブ・セットの識別にも同じ値を使用します。

例では、OPEN は次のように *emp_cursor* を *bind_des* に対応付けます。

```
EXEC SQL OPEN emp_cursor USING DESCRIPTOR bind_des;
```

bind_des はコロンで始めることはできません。

OPEN は SQL 文を実行します。問合せのときは、OPEN はアクティブ・セットを決定するとともにカーソルを先頭行に位置づけます。

選択リストの DESCRIBE

動的 SQL 文が問合せのときは、DESCRIBE SELECT LIST 文は OPEN 文の後、かつ FETCH 文の前に指定する必要があります。

DESCRIBE SELECT LIST は、選択記述子に選択リスト項目の記述を設定します。例では、DESCRIBE は次のように *select_des* を準備します。

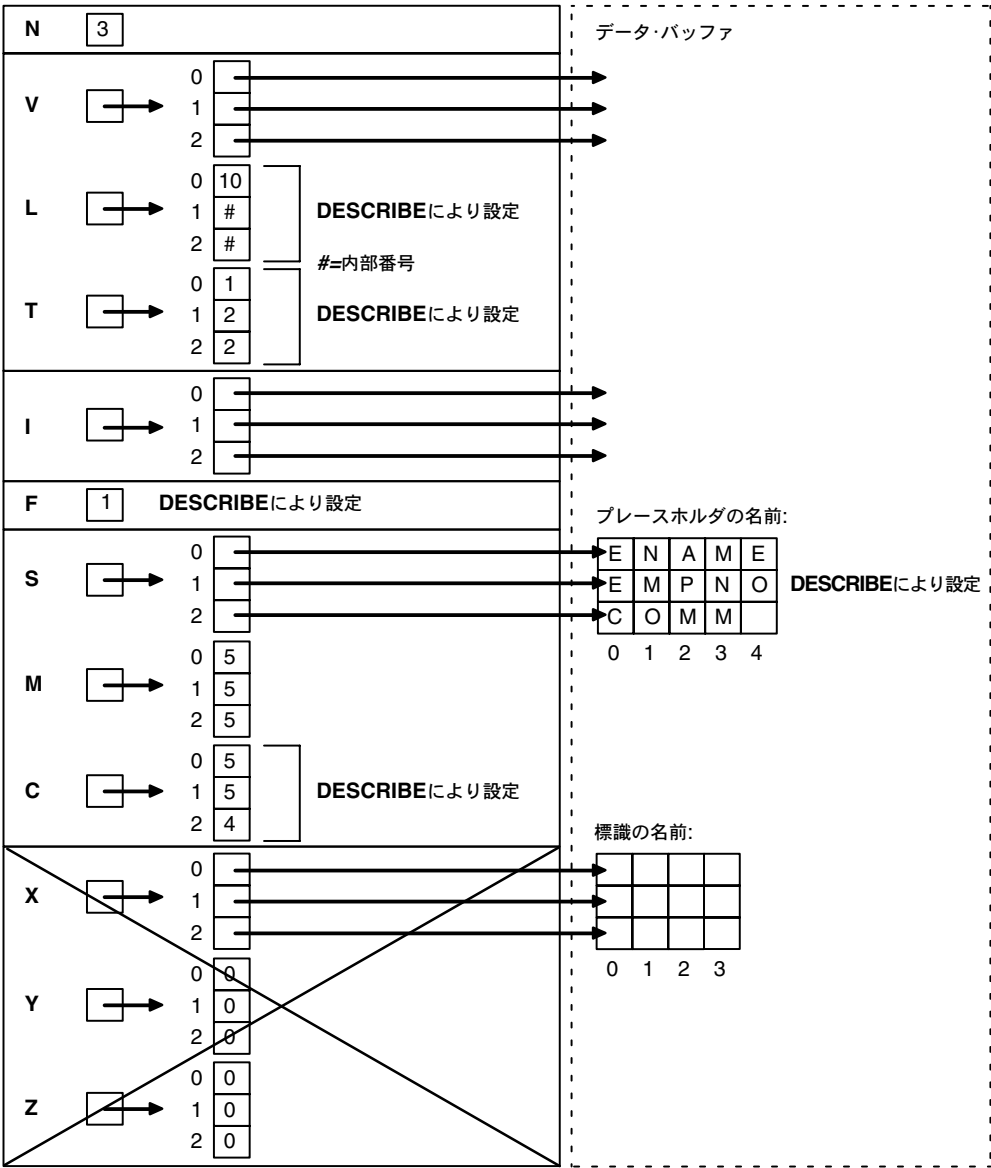
```
EXEC SQL DESCRIBE SELECT LIST FOR sql_stmt INTO select_des;
```

Oracle のデータ・ディクショナリにアクセスすることによって、DESCRIBE は各選択リストの値の長さやデータ型を設定します。

図 15-6 に、DESCRIBE 実行後の選択記述子を示します。問合せの選択リストで検出された項目の実際数が DESCRIBE によって *F* に設定されています。SQL 文が問合せでないときは、*F* はゼロに設定されます。

また、NUMBER 型の長さはまだ使用できません。NUMBER と定義した列には、ライブラリ関数 `SQLNumberPrecV6()` を使用して精度とスケールを抽出する必要があります。15-14 ページの「[データ型の強制変換](#)」を参照してください。

図 15-6 DESCRIBE 実行後の選択記述子



選択リスト項目の最大数のリセット

次に選択リスト項目の最大数を、DESCRIBE によって実際に検出された数にリセットする必要があります。

```
select_des->N = select_des->F;
```

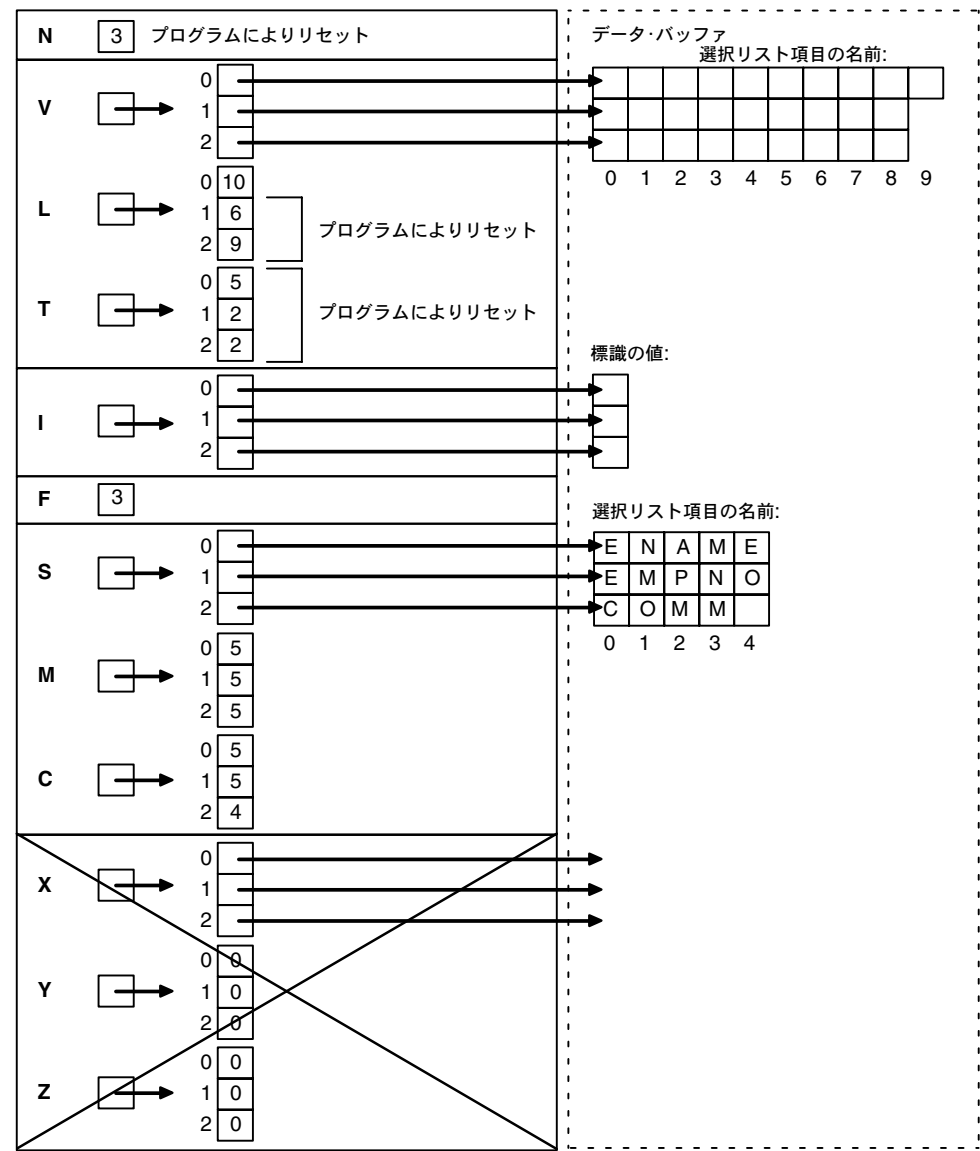
各選択リスト項目の長さとデータ型のリセット

例では、選択リストの値を FETCH する前に、ライブラリ関数 *malloc()* を使用して、記憶領域を割り当てます。また、表示用の長さとデータ型の配列の要素のいくつかをリセットします。

```
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;
            break;
        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;
}
```

図 15-7 に、結果として得られる選択記述子を示します。NUMBER の長さはこのとき使用可能となります。データ型はすべて STRING です。L[1] および L[2] の長さはそれぞれ 6 と 9 になっています。これは、DESCRIBE された長さ 4 と 7 にそれぞれ符号と小数点のための 2 を加算したためです。

図 15-7 FETCH 前の選択記述子



アクティブ・セットからの行の FETCH

FETCH はアクティブ・セットから 1 行を戻し、データ・バッファに選択リストの値を格納してから、カーソルをアクティブ・セットの次の行に進めます。行がなくなると、FETCH は "データが見つかりません" の Oracle エラー・コードを *sqlca.sqlcode* に設定します。例では、FETCH は次のように ENAME、EMPNO および COMM の列の値を *select_des* に戻します。

```
EXEC SQL FETCH emp_cursor USING DESCRIPTOR select_des;
```

図 15-8 に、FETCH 実行後の選択記述子を示します。Oracle は選択リストの値と標識の値を、*V* と *I* の要素によってアドレスされるデータ・バッファに格納しています。

データ型 1 の出力バッファについては、Oracle は *L* 配列に格納された長さを使用し、CHAR または VARCHAR2 のデータを左揃えにしてから、NUMBER データを右揃えにします。データ型 5 (STRING) の出力バッファについては、値を左揃えにし、CHAR、VARCHAR2 および NUMBER のデータに NULL 終了記号を付けます。

値 "MARTIN" は、EMP 表の VARCHAR2(10) 列から取り出されました。*L[0]* の長さを使用して、Oracle は 10 バイトのフィールドの値を左揃えにしてバッファを埋めます。

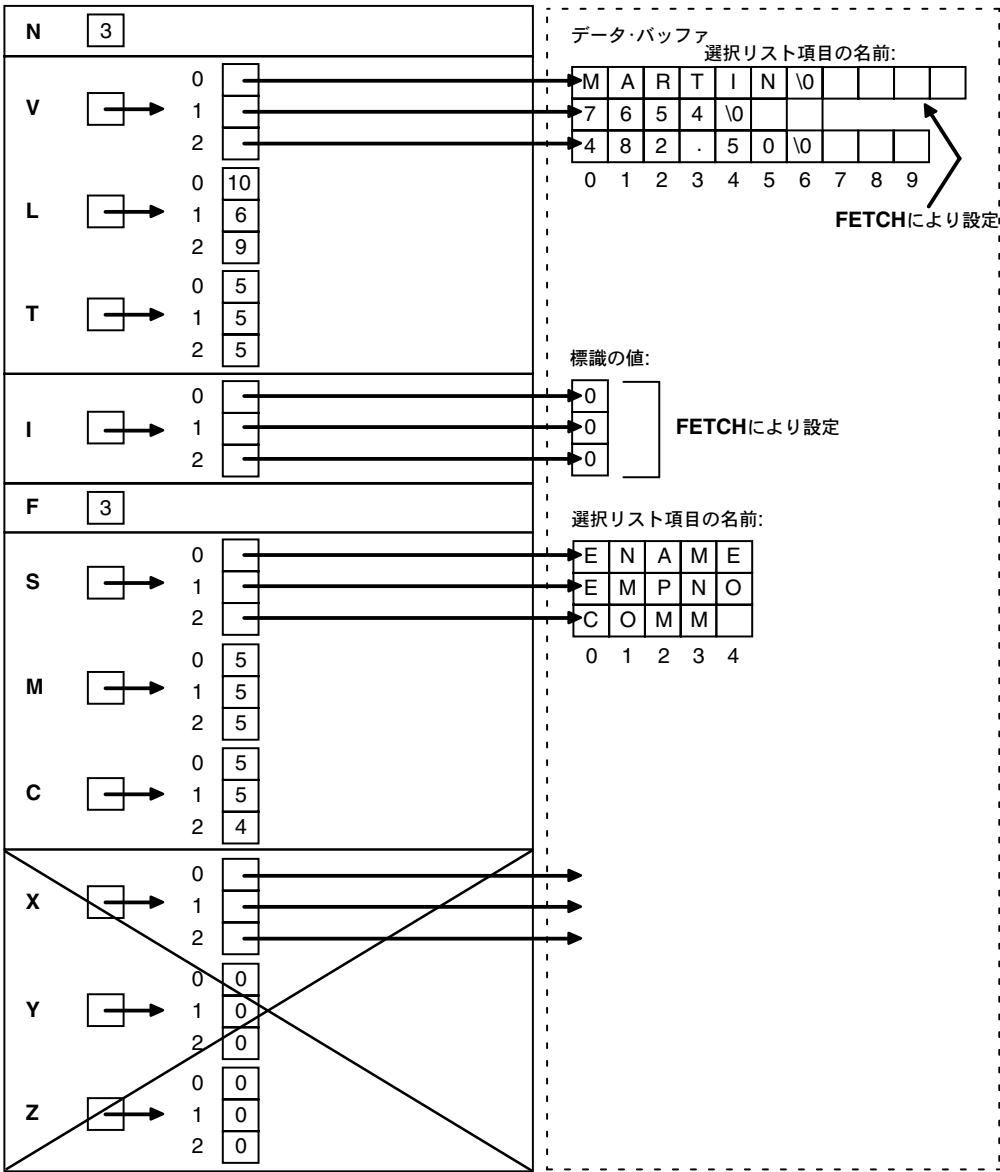
値 7654 は NUMBER(4) 列から取り出され、'7654' に強制変換されています。しかし、符号と小数点を使用可能にするため、*L[1]* の長さが 2 のみ増えています。そこで Oracle は 6 バイトのフィールドの値を左揃えにしてから、NULL 終了記号を付けます。

値 482.50 は NUMBER(7,2) 列から取り出され、'482.50' に強制変換されています。*L[2]* の長さが 2 のみ増えています。Oracle は 9 バイトのフィールドの値を左揃えにしてから、NULL 終了記号を付けます。

選択リストの値の取得と処理

FETCH 後、プログラムで戻り値を処理できます。例では、列 ENAME、EMPNO および COMM の値が処理されます。

図 15-8 FETCH 後の選択記述子



記憶域の割当て解除

`malloc()` によって割り当てられた記憶領域を解除するには、`free()` ライブラリ関数を使用します。構文は次のとおりです。

```
free(char *pointer);
```

例では、選択リスト項目、バインド変数および標識変数の値に対する記憶領域の割当てを次のように解除します。

```
for (i = 0; i < select_des->F; i++)    /* for select descriptor */
{
    free(select_des->V[i]);
    free(select_des->I[i]);
}
for (i = 0; i < bind_des->F; i++)    /* for bind descriptor */
{
    free(bind_des->V[i]);
    free(bind_des->I[i]);
}
```

記述子の記憶領域の割当てを解除するには、次の構文のライブラリ関数 `SQLSQLDAFree()` を使用します。

```
SQLSQLDAFree(context, descriptor_name);
```

記述子は `SQLSQLDAAlloc()` を使用して割り当ててください。そうしないと結果は予測できなくなります。

例では、選択記述子とバインド記述子に対する記憶領域の割当てを次のように解除します。

```
SQLSQLDAFree(SQL_SINGLE_RCTX, select_des);
SQLSQLDAFree(SQL_SINGLE_RCTX, bind_des);
```

カーソルの CLOSE

CLOSE はカーソルを使用禁止にします。例では、CLOSE は次のように `emp_cursor` を使用禁止にします。

```
EXEC SQL CLOSE emp_cursor;
```

ホスト配列の使用

方法 4 で入力ホスト配列または出力ホスト配列を使用するには、オプションの FOR 句を使用してホスト配列のサイズを Oracle に通知する必要があります。FOR 句の詳細は、[第 8 章「ホスト配列」](#) を参照してください。

次の構文を使用して、*i* 番目の選択リスト項目またはバインド変数に記述子エントリを設定する必要があります。

```
V[i] = array_address;  
L[i] = element_size;
```

このとき *array_address* はホスト配列のアドレスで、*element_size* はある配列要素のサイズです。

EXECUTE 文または FETCH 文（どちらか適切なほう）に FOR 句を指定することによって、処理対象の配列要素の数を Oracle に通知する必要があります。Oracle がホスト配列のサイズを認識する方法は他にないので、このプロシージャは必須です。

次の完全なプログラム例では、3つの入力ホスト配列を使用して EMP 表に行を INSERT します。方法 4 による問合せ以外のデータ操作言語文にも EXECUTE を使用できることに注意してください。

```
#include <stdio.h>  
#include <sqlcpr.h>  
#include <sqllda.h>  
#include <sqlca.h>  
  
#define NAME_SIZE    10  
#define INAME_SIZE   10  
#define ARRAY_SIZE   5  
  
/* connect string */  
char *username = "scott/tiger";  
  
char *sql_stmt =  
"INSERT INTO emp (empno, ename, deptno) VALUES (:e, :n, :d)";  
int array_size = ARRAY_SIZE; /* must have a host variable too */  
  
SQLDA *binda;  
  
char names[ARRAY_SIZE][NAME_SIZE];  
int numbers[ARRAY_SIZE], depts[ARRAY_SIZE];  
  
/* Declare and initialize indicator vars. for empno and deptno columns */  
short ind_empno[ARRAY_SIZE] = {0,0,0,0,0};  
short ind_dept[ARRAY_SIZE] = {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->T[0] = 3;
binda->I[0] = ind_empno;

binda->V[1] = (char *) names;
binda->L[1] = (long) NAME_SIZE;
binda->T[1] = 1;
binda->I[1] = (short *)0;

binda->V[2] = (char *) depts;
binda->L[2] = (long) sizeof (int);
binda->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;
ind_dept[3] = -1;          /* set indicator to -1 to insert NULL */
depts[3] = 30;            /* value in depts[3] is ignored */
```

```

        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);
}

```

sample12.pc

配列のフェッチを使用した簡単な動的 SQL の例は、demo ディレクトリの sample12.pc ファイルにあります。

サンプル・プログラム : 動的 SQL 方法 4

このプログラムでは、動的 SQL 方法 4 を使用するために必要な基本ステップを示します。Oracle に接続すると、プログラムは `SQLSQLDAAlloc()` を使用して記述子用のメモリーを割り当ててから、ユーザーに SQL 文の入力を求めます。次に文の `PREPARE`、カーソルの `DECLARE` に続いて、`DESCRIBE BIND` を使用してバインド変数をチェックします。最後にカーソルを `OPEN` して、選択リスト項目を `DESCRIBE` します。入力された SQL 文が問合せのときは、プログラムは各行のデータを `FETCH` してからカーソルを `CLOSE` します。このプログラムは `demo` ディレクトリの `sample10.pc` ファイルにありますので、オンラインで利用できます。

```

/*****
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 multi-line 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 40

/* Maximum lengths of the _names_ of the
   select-list items or indicator variables. */
#define MAX_VNAME_LEN 30
#define MAX_INAME_LEN 30

#ifndef NULL
#define NULL 0
#endif

/* Prototypes */
#ifdef __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 sqllda;
EXEC SQL INCLUDE sqlca;

EXEC SQL BEGIN DECLARE SECTION;
    char    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;

        /* Prepare the statement and declare a cursor. */
        EXEC SQL WHENEVER SQLERROR DO sql_error();

        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;
```



```

/* Set the bind variables for any placeholders in the
   SQL statement. */
set_bind_variables();

/* Open the cursor and execute the statement.
 * If the statement is not a query (SELECT), the
 * 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 (strcmp(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
     * 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.
     *
     * 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->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] == '?' ||
                (strcmp(linebuf, "HELP", 4) == 0) ||
                (strcmp(linebuf, "help", 4) == 0))
        {
            help();
            iter = 1;
            continue;
        }

        if (strstr(linebuf, "BEGIN") ||
            (strstr(linebuf, "begin")))
        {
            plsqli = 1;
        }

        strcat(dyn_statement, linebuf);

        if ((plsqli && (cp = strrchr(dyn_statement, '/')) ||
            (!plsqli && (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;

    /* 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->N = bind_dp->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->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 ((strcmp(bind_dp->V[i], "NULL", 4) == 0) ||
            (strcmp(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 ((strcmp(dyn_statement, "SELECT", 6) != 0) &&
        (strcmp(dyn_statement, "select", 6) != 0))
    {
        select_dp->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;

    /* 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->L[i] = 18;
            break;

        case 12 : /* DATE datatype */
            select_dp->L[i] = 9;
            break;

        case 23 : /* RAW datatype */
            break;

        case 24 : /* LONG RAW datatype */
            select_dp->L[i] = 240;
            break;
    }
}

```

```

    }
    /* 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);
    else
        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);
        else
            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->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 */
        else
            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, ' ');
        else
            printf ("%-*c ", (int)select_dp->L[i], ' ');
    else
        if (select_dp->T[i] == 3) /* int datatype */
            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]);
        else /* 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 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);
}
```

ラージ・オブジェクト (LOB)

この章では、LOB（ラージ・オブジェクト）データ型の埋込み SQL 文で提供されるサポートについて説明します。

4 種類の LOB について紹介し、今までの LONG および LONG RAW データ型と比較します。

Oracle コール・インタフェース API および PL/SQL 言語と同様の機能を提供する Pro*C/C++ の埋込み SQL インタフェースを示します。

LOB 文およびそのオプションとホスト変数を示します。

最後に、使用方法を簡単に示す LOB インタフェースを使用した Pro*C/C++ プログラムの例を挙げています。

主な項は次のとおりです。

- [LOB](#)
- [プログラムでの LOB の使用方法](#)
- [LOB 文のルール](#)
- [LOB 文](#)
- [LOB およびナビゲーション・インタフェース](#)
- [LOB プログラムの例](#)

LOB

LOB（ラージ・オブジェクト）列を使用して ASCII テキスト、各国文字のテキスト、様々なグラフィック形式のファイルおよびサウンド波形などの大量のデータ（最大 4 ギガバイト）を格納します。

内部 LOB

内部 LOB（BLOB、CLOB、NCLOB）はデータベース表領域に格納され、データベース・サーバーからトランザクション・サポート（コミット、ロールバックなどの処理）が有効です。

BLOB（バイナリ LOB）には、ビデオ・クリップなどの非構造化バイナリ（"raw" と呼ばれます）データが格納されます。

CLOB（キャラクタ LOB）には、データベース・キャラクタ・セットのキャラクタ・データの大きいブロックが格納されます。

NCLOB（ナショナル・キャラクタ LOB）には、ナショナル・キャラクタ・セットのキャラクタ・データの大きいブロックが格納されます。

外部 LOB

外部 LOB は、データベース表領域外のオペレーティング・システムのファイルです。データベース・サーバーのトランザクション・サポートは無効です。

BFILE（バイナリ・ファイル）には、外部バイナリ・ファイル形式のデータが格納されます。BFILE には、GIF、JPEG、MPEG、MPEG2、テキストなどの形式があります。

BFILE のセキュリティ

DIRECTORY オブジェクトは、BFILE にアクセスして操作するときに使います。DIRECTORY は、ファイルを格納するサーバー・ファイル・システムの実際の物理ディレクトリの論理的な別名です。ユーザーは、DIRECTORY オブジェクトのアクセス権限が割り当てられている場合に限り、ファイルにアクセスできます。

- DDL（データ定義言語）SQL 文 CREATE、REPLACE、ALTER および DROP は、DIRECTORY データベース・オブジェクトで使います。
- DML（データ管理言語）SQL 文は、DIRECTORY オブジェクトのシステムおよびオブジェクトの READ 権限を GRANT または REVOKE するために使います。

CREATEDIRECTORY 宣言文の例です。

```
EXEC SQL CREATE OR REPLACE DIRECTORY "Mydir" AS '/usr/home/mydir' ;
```

他のユーザーまたはロールは、GRANT などの DML（データ操作言語）で権限が割り当てられている場合に限り、ディレクトリを読み取ることができます。たとえば、ユーザー scott にディレクトリ /usr/home/mydir にある BFILES の読取り権限を割り当てる場合は、次のようにします。

```
EXEC SQL GRANT READ ON DIRECTORY "Mydir" TO scott ;
```

1 セッションで最大 10 個の BFILE を同時にオープンできます。このデフォルト値は、SESSION_MAX_OPEN_FILES パラメータを設定すると変更できます。

DIRECTORY オブジェクトおよび BFILE セキュリティの詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。GRANT コマンドの詳細は、『Oracle8i SQL リファレンス』を参照してください。

LOB 対 LONG および LONG RAW

LOB は今までの LONG および LONG RAW データ型と多くの点で異なります。

- LONG および LONG RAW の最大サイズは 2 ギガバイトですが、LOB は 4 ギガバイトです。
- LOB に対しては順次アクセスに加えてランダム・アクセスも可能です。LONG および LONG RAW に対しては順次アクセスのみが可能です。
- LOB（NCLOB を除く）は、定義対象のオブジェクト型の属性です。
- 表に複数の LOB 列を含めることができますが、LONG および LONG RAW は 1 列しか設定できません。

既存の LONG または LONG Raw 属性を、LOB に移行することをお勧めします。今後のリリースでは、LONG および LONG RAW に対するサポートを終了する予定です。移行の詳細は、『Oracle8i 移行ガイド』を参照してください。

LOB ロケータ

LOB ロケータは、LOB の実際の内容をポイントしています。LOB を取り出すと、LOB の内容ではなくロケータが戻されます。LOB ロケータは、1 つのトランザクションまたはセッションで保存し、別のトランザクションまたはセッションで使用することはできません。

テンポラリ LOB

ローカル変数のように使用できるテンポラリ LOB を作成すると、データベース LOB が使用しやすくなります。テンポラリ LOB は、表に関係付けられません。作成者のみがアクセスできます。また、ロケータを持っており（ロケータを使用してアクセスします）、セッション終了時には削除されます。

テンポラリ BFILES はサポートされていません。INSERT、UPDATE あるいは DELETE 文の WHERE 句でのみ、テンポラリ LOB を入力変数（IN 値）として使用できます。テンポラリ

LOB は、INSERT 文で挿入される値あるいは UPDATE 文の SET 句の値としても使用できます。テンポラリ LOB ではデータベース・サーバーのトランザクション・サポートが無効なので、COMMIT または ROLLBACK を行うことはできません。

テンポラリ LOB ロケータは、複数のトランザクションにまたがって使用することができません。サーバーが異常終了したとき、およびデータベース SQL 処理でエラーが発生したときは削除されます。

LOB バッファリング・サブシステム

LBS (LOB バッファリング・サブシステム) は、クライアント側のアドレス領域に、1 つ以上の LOB のバッファとして提供されるユーザー・メモリーです。

バッファリングには、LOB の特定の領域に対する少量データの読み込みと書き込みを何度も実行するクライアント上のアプリケーションで、特に次のような利点があります。

- LBS を使用すると、LOB に対して読み込み / 書き込みが複数回行われ、バッファがいっぱいになってから、FLUSH ディレクティブが実行されるときにサーバーに書き込まれるため、サーバー往復回数が減少します。
- また、バッファリングを使用すると、サーバー上での LOB 更新の合計回数も減少します。このため、LOB のパフォーマンスが向上し、ディスク領域を節約できます。

Oracle で提供しているバッファリングは、簡単なバッファ・サブシステムで、キャッシュではありません。バッファの内容は、サーバー LOB 値と必ずしも同期していません。実際にサーバー LOB に更新を書き込むには、FLUSH 文を使用します。

LOB のバッファへの読み込み / 書き込みは、ロケータを使用して行われます。バッファリングで使用可能にしたロケータは、書き込みを実行するまで一貫して LOB の読み込み機能を提供します。

ロケータは、WRITE のバッファに使用された後で更新され、バッファリング・サブシステムを介して表示できる最新の LOB に対するアクセス権限が割り当てられます。LOB に対するその後の WRITE は、この更新ロケータを介してのみバッファされます。LOB のバッファリング操作を含むトランザクションは、ユーザー・セッション間で移行することはできません。

LBS は、FLUSH 文を使用してサーバー LOB 値の更新を行うユーザーが管理します。LBS は、シングル・ユーザーおよび単一スレッドです。サーバー LOB の適正さを確保するには、ROLLBACK および SAVEPOINT アクションを使用します。LOB のバッファリング操作のトランザクション・サポートは保証していません。バッファされた LOB の更新のトランザクション・セマンティクスを確実にするには、論理セーブポイントをメンテナンスし、エラーが発生した場合は、ロールバックを実行する必要があります。

LBS の詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

プログラムでの LOB の使用方法

LOB にアクセスする 3 種類の方法

Pro*C/C++ の LOB にアクセスするには、次の 3 種類の方法があります。

- PL/SQL ブロックの DBMS_LOB パッケージ
- OCI (Oracle コール・インタフェース) 機能のコール
- 埋込み SQL 文

SQL 文は、PL/SQL インタフェースと同等の機能を提供するように設計されており、OCI インタフェースほど複雑ではありません。

次の表では、Pro*C/C++ 内での OCI ファンクション・コール、PL/SQL および Pro*C/C++ の埋込み SQL 文による LOB アクセスを比較しています。空欄は機能がないことを示します。

表 16-1 LOB アクセス方法

OCI ¹	PL/SQL ²	Pro*C/C++ 埋込み 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

表 16-1 LOB アクセス方法

OCI ¹	PL/SQL ²	Pro*C/C++ 埋込み SQL
OCILobFileExists	FILEEXISTS()	DESCRIBE
OCILobFileGetName	FILEGETNAME()	DESCRIBE
OCILobFileIsOpen	FILEISOPEN()	DESCRIBE
OCILobFileOpen	FILEOPEN()	OPEN
OCILobFileSetName	BFILENAME()	FILE SET ³
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
OCILobWrite	WRITE()	WRITE
OCILobWriteAppend	WRITEAPPEND()	WRITE

¹ C/C++ ユーザーのみ。これらの関数のプロトタイプは、ociap.h にあります。
² dbmslob.sql を参照します。BFILENAME を除くすべてのルーチンには、「DBMS_LOB.」という接頭辞がついています。
³ SQL 関数に組み込まれている関数 BFILENAME() も使用できます。

注意：LOB の修正または変更を行う新しい文を使用する前に、行を明示的にロックする必要があります。LOB 値を修正する操作は、APPEND、COPY、ERASE、LOAD FROM FILE、TRIM および WRITE です。

アプリケーションの LOB ロケータ

LOB ロケータを Pro*C/C++ アプリケーションで使用する場合は、oci.h ヘッダー・ファイルをインクルードし、BLOB に対して OCIBlobLocator 型のポインタ、CLOB および NCLOB に対して OCIClobLocator、あるいは BFILE に対して OCIBFileLocator を宣言します。

NCLOB の場合は、次のいずれかの操作を行う必要があります。

- C/C++ 宣言で「CHARACTER SET IS NCHAR_CS」句を使用します。
- コマンドラインまたは構成ファイルで、NLS_CHAR プリコンパイラ・オプションをあらかじめ指定しておき、NLS_NCHAR 環境変数を設定する必要があります。

詳細は、10-31 ページの「NLS_CHAR」を参照してください。設定方法は次のとおりです。

```
/* In your precompiler program */
#include <oci.h>
...
OCIClobLocator CHARACTER SET IS NCHAR_CS *a_nclob ;
```

または、Pro*C/C++ をコールするときに、プリコンパイラ・オプション NLS_CHAR を次のように設定している場合は、

```
NLS_CHAR=(a_nclob)
```

コードから、CHARACTER SET 句を削除できます。

```
#include <oci.h>
...
OCIClobLocator *a_nclob ;
```

他に次のように簡潔に宣言します。

```
/* In your precompiler program */
#include <oci.h>
...
OCIBlobLocator *a_blob ;
OCIClobLocator *a_clob ;
OCIBFileLocator *a_bfile ;
```

LOB の初期化

内部 LOB

BLOB を初期化して空にするには、EMPTY_BLOB() 関数を使用するか、ALLOCATE SQL 文を使用します。CLOB および NCLOB の場合は、EMPTY_CLOB() 関数を使用します。EMPTY_BLOB() および EMPTY_CLOB() の詳細は、『Oracle8i SQL リファレンス』を参照してください。

これらの関数は、INSERT 文の VALUES 句または UPDATE 文の SET 句のみで使用できます。

たとえば、次のとおりです。

```
EXEC SQL INSERT INTO lob_table (a_blob, a_clob)
VALUES (EMPTY_BLOB(), EMPTY_CLOB());
```

ALLOCATE 文を実行すると、LOB ロケータが割り当てられ、初期化後に空になります。つまり、次のコードを実行しても、前の例と同じ結果が得られます。

```
#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);
```

外部 LOB

次の方法で、LOB FILE SET 文を使用して BFILE の DIRECTORY 別名および FILENAME を初期化します。

```
#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) ;
```

DIRECTORY オブジェクト・ネーミング規則および DIRECTORY オブジェクト権限の詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

また、INSERT または UPDATE 文で BFILENAME ('ディレクトリ', 'ファイル名') 関数を使用し、BFILE 列または特定の行の属性を初期化してから、実際の物理ディレクトリまたはファイル名を指定することもできます。

```
EXEC SQL INSERT INTO file_table (a_bfile)
VALUES (BFILENAME('lob_dir', 'image.gif'))
RETURNING a_bfile INTO :bfile ;
```

注意: BFILENAME() では、ディレクトリまたはファイル名の権限、および物理ディレクトリの存在は確認されません。BFILE ロケータを使用してファイルにアクセスしたときに、これらが確認され、ファイルにアクセスできない場合にはエラーが戻されます。

テンポラリ LOB

テンポラリ LOB は、埋込み SQL LOB CREATE TEMPORARY 文を使用して最初に作成するときに、初期化されて空になります。テンポラリ LOB には、EMPTY_BLOB() および EMPTY_CLOB() 関数を使用することはできません。

LOB の解放

FREE 文は、ALLOCATE 文によって確保されたメモリーを解放する際に使用します。

```
EXEC SQL FREE :a_blob;
```

LOB 文のルール

LOB 文を使用するときのルールを次に示します。

すべての LOB 文に対するルール

SQL LOB 文で LOB を操作する場合、次の一般的な制限事項および制約が適用されます。

- 埋込み SQL LOB 文では、FOR 句を使用できません。埋込み SQL LOB 文では、LOB ロケータを複数使用することはできません。ただし、ALLOCATE 文および FREE 文では、FOR 句を使用できます。
- 分散 LOB はサポートされていません。新しい埋込み SQL LOB 文では、AT データベース句を使用できますが、同一の SQL LOB 文で、異なるデータベース接続を使用して作成または ALLOCATE された LOB ロケータを混在させることはできません。
- LOB READ および WRITE 操作を行う場合、OCI では、コールバック関数をクライアントから指定できるコールバック・メカニズムを提供しています。このコールバック関数は、LOB 値ピースの読み込みあるいは書き込みが行われるたびに実行されます。埋込み SQL LOB 文では、この機能はサポートされていません。
- OCI では、テンポラリ LOB を作成するときに使用可能な期間を、独自に作成または指定できるメカニズムを提供しています。テンポラリ LOB の READ および WRITE 操作に使用されるバッファ・キャッシュを指定するメカニズムもあります。このインタフェースでは、これらの機能はサポートされていません。

LOB バッファリング・サブシステムに対するルール

LBS では、次のルールに従う必要があります。

- 読み込みアクセスまたは書き込みアクセスのエラーは、次のサーバーへのアクセス時に報告されます。このため、エラー・リカバリのコーディングはユーザーが行う必要があります。
- バッファに書き込まれた LOB を更新するときは、LOB バッファリング・サブシステムを経由せずに更新しないでください。

- バッファリングが使用可能な更新された LOB ロケータを、IN パラメータとして PL/SQL プロシージャに渡すことはできますが、IN OUT または OUT パラメータとして渡すことはできません。エラーが戻されます。更新されたロケータを戻そうとしたときも、エラーが戻されます。
- バッファリングが使用可能な更新されたロケータを、ASSIGN 文で別のロケータに割り当てることはできません。
- バッファされた書込みを LOB 値に追加できますが、LOB の最後の次の 1 文字は開始オフセットにする必要があります。LBS では、APPEND 文を使用して、データベース・サーバーの LOB にゼロ・バイトの充填文字または空白を追加することはできません。
- ホスト・ロケータ・バインド変数およびデータベース・サーバー CLOB のキャラクタ・セットは、同じにしてください。
- バッファリングが使用可能なロケータでは、ASSIGN 文、READ 文および WRITE 文以外は実行できません。
- バッファリングが使用可能なロケータで、APPEND、COPY、ERASE、DESCRIBE (LENGTH のみ) および TRIM 文を実行するとエラーが発生します。ロケータがポイントしている LOB に、別のロケータからバッファ・モードでアクセスしている場合は、バッファリングが使用禁止なロケータを使用してこれらの文を実行するとエラーが戻されます。

注意: 次の処理の前に、LOB バッファリング・サブシステムが使用可能な LOB に対して、FLUSH 文を実行する必要があります。

- トランザクションをコミットするとき
- カレント・トランザクションから別のトランザクションに移行するとき
- LOB に対してバッファ操作を使用禁止にするとき
- 外部プロシージャの実行から PL/SQL ルーチンに戻るとき

注意: ロケータ・パラメータによって PL/SQL ブロックから外部コールアウトがコールされた場合は、ENABLE 文を含むすべてのバッファリングは、コールアウト内で行う必要があります。

次の手順に従います。

- 外部コールアウトをコールします。
- バッファリングのロケータを ENABLE します。
- ロケータを使用して READ または WRITE します。
- LOB に対して FLUSH します (LOB を暗黙的にフラッシュできません)。
- バッファリングのロケータを使用禁止にします。
- PL/SQL のファンクション / プロシージャ / メソッドに戻ります。

LOB は、明示的に FLUSH する必要があります。

ホスト変数に対するルール

LOB 文では、次のルールおよび注意事項に従ってください。

- *src* および *dst* から内部ロケータまたは外部 LOB ロケータを参照できますが、*file* からは外部ロケータのみを参照できます。
- 数値のホスト値 (*amt*、*src_offset*、*dst_offset* など) は、4 バイトの符号なし整数の変数として宣言されます。値は 0 ～ 4 ギガバイトに制限されます。
- NULL は、LOB ロケータで使います。LOB 文では標識変数は必要ありません。NULL は、*amt*、*src_offset* などの数値変数には使用できません。エラーが発生します。
- オフセット値 *src_offset* および *dst_offset* のデフォルト値は 1 です。

LOB 文

文はアルファベット順に並んでいます。*database* は、すべての文でデータベース接続を表しています。

APPEND

用途

この文は LOB 値を別の LOB の最後に追加します。

構文

```
EXEC SQL [AT [:]database] LOB APPEND :src TO :dst ;
```

ホスト変数

src (IN)

ソース LOB を固有に参照する内部 LOB ロケータ。

dst (IN OUT)

宛先 LOB を固有に参照する内部 LOB ロケータ。

使用上の注意

データは、ソース LOB から宛先 LOB の最後にコピーされます。宛先 LOB は最大 4 ギガバイトまで拡張できます。4 ギガバイトを超えて LOB を拡張すると、エラーが発生します。

ソース LOB および宛先 LOB はあらかじめ存在している必要があります。宛先 LOB は初期化されている必要があります。

ソース LOB および宛先 LOB は、同じ内部 LOB 型にする必要があります。ロケータのいずれかの型に対して、LOB バッファリングが使用可能になっている場合はエラーになります。

ASSIGN

用途

LOB または BFILE ロケータを別のロケータに割り当てます。

構文

```
EXEC SQL [AT [:]database] LOB ASSIGN :src to :dst ;
```

ホスト変数

src (IN)

コピー元の LOB または BFILE ロケータ・ソース。

dst (IN OUT)

コピー先の LOB または BFILE ロケータ。

使用上の注意

割当て後には、ロケータは両方とも同じ LOB 値を参照します。宛先 LOB ロケータは初期化された有効な (ALLOCATE で割り当てられた) ロケータにしてください。

内部 LOB の場合は、宛先ロケータが表に格納されている場合に限り、ソース・ロケータの LOB 値が宛先ロケータの LOB 値にコピーされます。Pro*C/C++ の場合は、宛先ロケータを含むオブジェクトに対して FLUSH を発行すると、LOB 値がコピーされます。

BFILE ロケータが内部 LOB ロケータに割り当てられている場合またはその逆の場合には、エラーが戻されます。srcLOB と dstLOB が同じ型でない場合にもエラーになります。

バッファリングが使用可能な内部 LOB に対するソース・ロケータの場合、そのソース・ロケータが LOB バッファリング・サブシステム経由で LOB 値を修正するために使用され、WRITE 後にバッファに対して FLUSH していないときは、ソース・ロケータを宛先ロケータに割り当ててはできません。LOB バッファリング・サブシステムから修正できる LOB 値は、各 LOB につき 1 つのロケータに限られているためです。

CLOSE

用途

オープンしている LOB または BFILE をクローズします。

構文

```
EXEC SQL [AT [:]database] LOB CLOSE :src ;
```

ホスト変数

src (IN OUT)

クローズする LOB または BFILE のロケータ。

使用上の注意

異なるロケータを使用した場合も、同じロケータを使用した場合も、同じ LOB を 2 度クローズするとエラーになります。外部 LOB の場合は、BFILE が存在するのにオープンしていない場合は、エラーは発生しません。

オープンしていたすべての LOB をクローズする前に、トランザクションに対して COMMIT するとエラーになります。トランザクションの ROLLBACK 時にオープンされている LOB は、すべてクローズされずに破棄されます。

COPY

用途

LOB 値の全部または一部を、2 番目の LOB にコピーします。

構文

```
EXEC SQL [AT [:]database] LOB COPY :amt FROM :src [AT :src_offset]  
      TO :dst [AT :dst_offset] ;
```

ホスト変数

amt (IN)

コピーする BLOB の最大バイト数または CLOB および NCLOB の最大文字数。

src (IN)

ソース LOB のロケータ。

src_offset (IN)

CLOB または NCLOB の場合は、文字数。BLOB の場合は、バイト数。LOB の先頭で 1 から始まります。

`dst (IN)`

宛先 LOB のロケータ。

`dst_offset (IN)`

宛先オフセット。`src_offset` と同じルールです。

使用上の注意

宛先のオフセット以降にデータがすでに存在する場合は、そのデータはソース・データにより上書きされます。宛先のオフセットがカレント・データの最後を超えている場合は、ゼロ・バイト充填文字（BLOB の場合）または空白（CLOB の場合）が、カレント・データの最後から新しく書き込まれたソース・データの先頭まで、宛先 LOB に書き込まれます。

新しく書き込まれたデータが現行の宛先 LOB の長さを超える場合は、格納できるように宛先 LOB が拡張されます。この LOB を 4 ギガバイトを超えて拡張すると、ランタイム・エラーになります。

初期化していない LOB からコピーする場合もエラーになります。

ソース LOB および宛先 LOB は、同じ型にする必要があります。いずれのロケータも、LOB バッファリングを使用可能にしないでください。

`amt` 変数は、コピーの最大量です。指定した LOB 値がコピーされる前にソース LOB の最後に到達した場合は、操作はエラーなしで終了します。

テンポラリ LOB を永続 LOB にするには、COPY 文を使用して、テンポラリ LOB を永続 LOB に明示的に COPY する必要があります。

CREATE TEMPORARY

用途

テンポラリ LOB を作成します。

構文

```
EXEC SQL [AT [:] database] LOB CREATE TEMPORARY :src ;
```

ホスト変数

`src (IN OUT)`

実行前は IN で、`src` は、以前に ALLOCAT された LOB ロケータです。

実行後は OUT になり、`src` は、新しい空のテンポラリ LOB をポイントする LOB ロケータです。

使用上の注意

実行が正常に終了すると、ロケータは新しく作成されたテンポラリ LOB をポイントします。テンポラリ LOB は、データベース・サーバーに格納され、表には関係付けられません。テンポラリ LOB は、空で長さはゼロです。

セッションの最後に、すべてのテンポラリ LOB は解放されます。テンポラリ LOB に対する READ および WRITE では、バッファ・キャッシュは経由されません。

DISABLE BUFFERING

用途

LOB ロケータの LOB バッファリングを使用禁止にします。

構文

```
EXEC SQL [AT [:]database] LOB DISABLE BUFFERING :src ;
```

ホスト変数

src (IN OUT)

内部 LOB ロケータ。

使用上の注意

この文では BFILE はサポートされていません。この文以降の読み込みあるいは書き込みでは、LBS は使用されません。

注意: この文では、LOB バッファリング・サブシステムの変更は暗黙的にフラッシュされないため、変更を有効にするには FLUSH BUFFER コマンドを使用します。

ENABLE BUFFERING

用途

LOB ロケータの LOB バッファリングを使用可能にします。

構文

```
EXEC SQL [AT [:]database] LOB ENABLE BUFFERING :src ;
```

ホスト変数

src (IN OUT)

内部 LOB ロケータ。

使用上の注意

この文では BFILE はサポートされていません。この文以降の読み込みおよび書き込みでは、LBS が使用されます。

ERASE

用途

LOB データの任意の値の消去を任意のオフセットから開始します。

構文

```
EXEC SQL [AT [:]database] LOB ERASE :amt FROM :src [AT :src_offset] ;
```

ホスト変数

amt (IN OUT)

入力は、消去するバイト数または文字数です。出力は、実際に消去された数です。

src (IN OUT)

内部 LOB ロケータ。

src_offset (IN)

LOB の先頭からのオフセット。1 から始まります。

使用上の注意

この文では BFILE はサポートされていません。

実行後に消去された実際の文字 / バイト数が amt から戻されます。要求した文字 / バイト数が消去される前に LOB 値の最後に到達した場合は、実際の消去数と要求した消去数が異なることがあります。LOB が空の場合は、amt には、ゼロ文字 / バイトが消去されたことを示します。

BLOB の場合は、消去は既存の LOB 値をゼロ・バイトの充填文字で上書きすることです。CLOB の場合は、消去は既存の LOB 値を空白で上書きすることです。

FILE CLOSE ALL

用途

カレント・セッションでオープンしているすべての BFILES をクローズします。

構文

```
EXEC SQL [AT [:]database] LOB FILE CLOSE ALL ;
```

使用上の注意

クローズ処理が正常に終了しなかったためにセッションにオープンしているファイルが存在する場合は、FILE CLOSE ALL 文を使用して、セッション内でオープンしているファイルをすべてクローズし、最初からファイル操作を再開できます。

FILE SET

用途

DIRECTORY 別名を設定し、BFILE ロケータに FILENAME を設定します。

構文

```
EXEC SQL [AT [:]database] LOB FILE SET :file  
        DIRECTORY = :alias, FILENAME = :filename ;
```

ホスト変数

file (IN OUT)

DIRECTORY 別名および FILENAME が設定されている BFILE ロケータ。

alias (IN)

設定する DIRECTORY 別名。

filename (IN)

設定する FILENAME。

使用上の注意

指定した BFILE ロケータは、この文で使用する前にあらかじめ ALLOCATE される必要があります。

DIRECTORY 別名および FILENAME の両方が必要です。

DIRECTORY 別名の最大長は 30 バイトです。FILENAME の最大長は 255 バイトです。

DIRECTORY 別名および FILENAME 属性は、CHARZ、STRING、VARCHAR、VARCHAR2 および CHARF 以外の外部データ型ではサポートされません。

この文を外部 LOB ロケータ以外で使用するとエラーになります。

FLUSH BUFFER

用途

この LOB のバッファをデータベース・サーバーに書き込みます。

構文

```
EXEC SQL [AT [:]database] LOB FLUSH BUFFER :src [FREE] ;
```

ホスト変数

src (IN OUT)

内部 LOB ロケータ。

使用上の注意

入力ロケータから参照される LOB から、サーバー上のデータベース LOB にバッファ・データを書き込みます。

LOB バッファリングは、入力 LOB ロケータに対してあらかじめ使用可能にする必要があります。

デフォルトでは、バッファ・リソースは、バッファされた別の LOB 操作で再度割り当てられることがあるため、FLUSH 操作では解放されません。ただし、バッファを明示的に解放する場合は、オプションの FREE キーワードを指定すると解放できます。

FREE TEMPORARY

用途

LOB ロケータのテンポラリ領域を解放します。

構文

```
EXEC SQL [AT [:]database] LOB FREE TEMPORARY :src ;
```

ホスト変数

src (IN OUT)

テンポラリ LOB をポイントしている LOB ロケータ。

使用上の注意

入力ロケータは、テンポラリ LOB をポイントする必要があります。出力ロケータは、初期化されていないとマークされ、この文以降の LOB 文で使用できます。

LOAD FROM FILE

用途

BFILE の全部または一部を内部 LOB にコピーします。

構文

```
EXEC SQL [AT [:]database] LOB LOAD :amt FROM FILE :file [AT :src_offset] INTO  
:dst [AT :dst_offset] ;
```

ホスト変数

amt (IN)

ロードされる最大バイト数。

file (IN OUT)

BFILE ロケータのソース。

src_offset (IN)

ファイルの先頭からのオフセットのバイト数。1 から始まります。

dst (IN OUT)

宛先 LOB ロケータ。BLOB、CLOB または NCLOB になります。

dst_offset (IN)

書き込みが開始される宛先 LOB の先頭からのバイト数（BLOB の場合）または文字数（CLOB および NCLOB の場合）。1 から始まります。

使用上の注意

データはソース BFILE から宛先内部 LOB にコピーされます。BFILE データを CLOB または NCLOB にコピーする場合は、キャラクタ・セット変換は行われません。このため、BFILE データは、あらかじめデータベースの CLOB または NCLOB と同じキャラクタ・セットにしておく必要があります。

ソース LOB および宛先 LOB は、あらかじめ存在する必要があります。宛先の開始位置にデータがすでに存在する場合は、ソース・データで上書きされます。宛先の開始位置がカレント・データの最後を超える場合は、データの最後から新しく書き込まれたソース・データの最初まで、ゼロ・バイトの充填文字（BLOB）または空白（CLOB および NCLOB）が宛先 LOB に書き込まれます。

新しく書き込まれたデータが現行の宛先 LOB の長さを超える場合は、格納できるように宛先 LOB が拡張されます。この LOB を 4 ギガバイトを超えて拡張するとエラーになります。

初期化していない BFILE からコピーする場合もエラーになります。

量パラメータは、ロードの最大量を示します。指定した量がロードされる前にソース BFILE の最後に到達した場合は、処理はエラーなしで終了します。

OPEN

用途

読み込みまたは読書きアクセスで使用する LOB または BFILE をオープンします。

構文

```
EXEC SQL [AT [:]database] LOB OPEN :src [ READ ONLY | READ WRITE ] ;
```

ホスト変数

src (IN OUT)

LOB または BFILE の LOB ロケータ。

使用上の注意

LOB または BFILE を OPEN できるデフォルト・モードは、READ ONLY アクセスです。

内部 LOB の場合は、OPEN はロケータではなく LOB に関係付けられます。すでに OPEN されているロケータを別のロケータに割り当てても、新しい LOB を OPEN したとは見なされず、両方のロケータから同じ LOB が参照されます。BFILE の場合は、OPEN はロケータに関係付けられます。

同時に 32 個の LOB を OPEN できます。33 個目の LOB を OPEN するとエラーが戻されます。

書き込み可能な BFILE はサポートされていません。このため、BFILE を READ WRITE モードで OPEN すると、エラーが戻されます。

READ ONLY モードの LOB をオープンして、LOB に WRITE した場合もエラーになります。

READ

用途

LOB または BFILE の全部または一部をバッファに読み込みます。

構文

```
EXEC SQL [AT [:]database] LOB READ :amt FROM :src [AT :src_offset]  
        INTO :buffer [WITH LENGTH :buflen] ;
```

ホスト変数

amt (IN OUT)

入力は、読み込まれる文字数またはバイト数です。出力は、実際の文字数またはバイト数です。

読み込まれるバイト数がバッファ長よりも大きい場合は、LOB はポーリング・モードで READ されると見なされます。入力時にこの値がゼロの場合は、データはポーリング・モードで入力オフセットから LOB の最後まで読み込まれます。

実際に読み込まれたバイト数あるいは文字数は、amt に戻されます。データが分割されて読み込まれた場合は、amt には必ず最後に読み込まれた部分の長さが入ります。

LOB の最後に到達した場合は、「ORA-01403: データが見つかりません。」というエラーが発生します。

ポーリング・モードで読み込む場合は、アプリケーションから LOB READ を繰り返しコールし、データがなくなるまで LOB ピースを読み込む必要があります。ORA-1403 エラーを取得するには、WHENEVER ディレクティブの NOT FOUND 条件を使用してポーリング・モードの使用を制御します。

src (IN)

LOB または BFILE ロケータ。

src_offset (IN)

読み込みを開始する、LOB 値の先頭からの絶対オフセットです。キャラクタ LOB の場合は、LOB の先頭からの文字数です。バイナリ LOB または BFILE の場合は、バイト数です。最初の位置は 1 です。

buffer (IN/OUT)

LOB データが読み込まれるバッファ。バッファの外部データ型の種類は、ソース LOB の型により限定されます。バッファの最大長は、LOB 値の格納に使用されている外部データ型によって決まります。次の表では、有効な外部データ型および対応する最大長を、ソース LOB 型単位に分類してあります。

表 16-2 ソース LOB およびプリコンパイラ・データ型

外部 LOB ¹	内部 LOB	プリコンパイラ外部 データ型	プリコンパイラ 最大長 ²	PL/SQL データ 型	PL/SQL 最 大長
BFILE	BLOB	RAW	65535	RAW	32767
		VARRAW	65533		
		LONG RAW	2147483647		
		LONG VARRAW	2147483643		
	CLOB	VARCHAR2	65535	VARCHAR2	32767
		VARCHAR	65533		
		LONG VARCHAR	2147483643		
	NCLOB	NVARCHAR2	4000	NVARCHAR2	4000

¹ BFILES で使用可能な外部データ型です。

² 長さは文字数ではなくバイト数です。

buflen (IN)

他の方法で指定できないときは、指定したバッファ長が指定されます。

使用上の注意

BFILE はデータベース・サーバーにあらかじめ存在し、入力ロケータを使用してオープンされている必要があります。データベースにはファイルを読み込む権限が、またユーザーにはディレクトリの読み込み権限が必要です。

初期化されていない LOB または BFILE から読み込むと、エラーになります。

バッファ長は次の方法で決まります。

- WITH LENGTH 句がある場合は、buflen で指定します。
- WITH LENGTH 句がない場合は、4-45 ページの「[各国語サポート](#)」のルールに従い、OUT モードのバッファ・ホスト変数の指定によって決定されます。

16-30 ページの「[BLOB の READ およびファイル書込みの例](#)」を参照してください。

TRIM

用途

LOB 値を切り捨てます。

構文

```
EXEC SQL [AT [:] database] LOB TRIM :src TO :newlen ;
```

ホスト変数

src (IN OUT)

内部 LOB の LOB ロケータ。

newlen (IN)

LOB 値の新しい長さ。

使用上の注意

この文は BFILES には使用できません。新しい長さを、現行の長さより大きくすることはできません。エラーが戻されます。

WRITE

用途

バッファの内容を LOB に書き込みます。

構文

```
EXEC SQL [AT [:] database] LOB WRITE [APPEND] [ FIRST | NEXT | LAST | ONE ]  
      :amt FROM :buffer [WITH LENGTH :buflen] INTO :dst [AT :dst_offset] ;
```

ホスト変数

amt (IN OUT)

入力は、書き込まれる文字数またはバイト数です。

出力は、書き込まれる実際の文字数またはバイト数です。

ポーリング・モードで書き込む場合は、WRITE LAST された後の *amt* には、WRITE 文の実行で書き込まれた累計の長さが戻されます。WRITE 文が中断された場合は、*amt* は定義されません。

buffer (IN)

LOB データが書き込まれるバッファ。データ型の長さは、16-21 ページの「[READ](#)」を参照してください。

dst (IN OUT)

LOB ロケータ。

dst_offset (IN)

CLOB および NCLOB の場合は文字数単位、BLOB の場合はバイト数単位の、LOB の先頭からのオフセット（1 から始まります）。

buflen (IN)

他の方法で計算できないときのバッファ長。

使用上の注意

LOB データがすでに存在する場合は、バッファに格納されているデータで上書きされます。指定したオフセットが LOB のカレント・データの最後を超える場合は、ゼロバイトの充填文字または空白が LOB に挿入されます。

WRITE 文に APPEND キーワードを指定すると、データは自動的に LOB の最後に書き込まれます。APPEND を指定した場合は、宛先オフセットは LOB の最後にあると見なされます。WRITE 文に APPEND オプションを指定したときは、宛先オフセットを指定するとエラーになります。

バッファは 1 ピース（デフォルトの ONE キーワードを使用します）で LOB に書き込まれますが、標準ポーリング・モードを使用するとピース単位で書き込まれます。

FIRST でポーリングが始まり、NEXT で後続のピースが書き込まれます。書込みを終了する最後のピースを書き込むには、LAST キーワードを使用します。

このピース単位の書込みモードを使用したときは、各ピースのサイズが異なり、異なる場所から書き込まれる場合、各コールでバッファおよび長さが異なることがあります。

すべての書込みが終了した後、Oracle に渡されるデータの合計量が amt パラメータに指定した量以下の場合、エラーが発生します。

このルールは、READ 文などでバッファ長を決定する場合にも適用されます。16-21 ページの「[READ](#)」を参照してください。

16-31 ページの「[ファイルの読み込みおよび BLOB の WRITE の例](#)」を参照してください。

DESCRIBE

用途

この文は複数の OCI および PL/SQL 文に相当します（このため最後に保存します）。LOB DESCRIBE SQL 文を使用して LOB から属性を取得します。この機能は、OCI および PL/SQL プロシージャに似ています。LOB DESCRIBE 文は次の形式になります。

構文

```
EXEC SQL [AT [:]database] LOB DESCRIBE :src GET attribute1 [{, attributeN}]  
      INTO :hv1 [[INDICATOR] :hv_ind1] [{, :hvN [[INDICATOR] :hv_indN }] ] ;
```

属性は、次のいずれかです。

CHUNKSIZE | DIRECTORY | FILEEXISTS | FILENAME | ISOPEN | ISTEMPORARY | LENGTH

ホスト変数

src (IN)

内部または外部 LOB の LOB ロケータ。

hv1 ... hvN ... (OUT)

属性値を受け取るホスト変数。属性名リストで指定した順に指定します。

hv_ind1 ... hv_indN ... (OUT)

標識 NULL 状態を受け取るオプションのホスト変数。属性名リストで指定した順に指定します。

この表では、属性、対応付けられる LOB、および読み込まれる C 型を説明します。

表 16-3 LOB 属性

LOB 属性	属性の説明	制限	C 型
CHUNKSIZE	LOB 値を格納する LOB チャンクに使用される領域の量 (BLOB の場合はバイト数単位、CLOB または NCLOB の場合は文字数単位です)。このチャンク・サイズの倍数で READ または WRITE 要求を発行すると、パフォーマンスが向上します。WRITE はすべてチャンク単位で行われます。チャンク単位以外の WRITE は行われません。重複して WRITE されることもありません。同一 CHUNK に対して複数の WRITE コールを発行するかわりに、チャンクがいっぱいになるまで WRITE を蓄積できます。	BLOB、CLOB および NCLOB のみ	符号なし INT
DIRECTORY	BFILE の DIRECTORY 別名。最大長は 30 バイトです。	FILE LOB のみ	char * ¹
FILEEXISTS	サーバーの OS のファイル・システム上に、BFILE が存在するかどうかを決定します。FILEEXISTS はゼロでないときは真で、ゼロのときは偽です。	FILE LOB のみ	符号付き INT
FILENAME	BFILE の名前。最大長は 255 バイトです。	FILE LOB のみ	char*
ISOPEN	BFILE の場合は、OPEN 文で入力 BFILE ロケータを使用しなかったときは、BFILE はこのロケータでは OPEN されていないと見なされます。ただし、別の BFILE ロケータによって OPEN されていることもあります。別のロケータを使用して、同一の BFILE に対して複数の OPEN を行うことができます。LOB の場合は、別のロケータにより LOB が OPEN された場合も、その入力ロケータによって OPEN されていると見なされます。ISOPEN はゼロでないときは真で、ゼロのときは偽です。		符号付き INT
ISTEMPORARY	入力 LOB ロケータからテンポラリ LOB を参照するかどうかを決定します。ISTEMPORARY は、ゼロでないときは真で、ゼロのときは偽です。	BLOB、CLOB および NCLOB のみ	符号付き INT

表 16-3 LOB 属性

LOB 属性	属性の説明	制限	C 型
LENGTH	BLOB および BFILE の長さはバイト数単位、CLOB および NCLOB の長さは文字数単位で表されます。BFILE の場合は、EOF が存在するときは、EOF も長さに含まれます。空の内部 LOB は、ゼロ長になります。初期化されていない LOB および BFILE の長さは、定義されません。		符号なし INT

¹ DIRECTORY 属性および FILENAME 属性の場合は、CHARZ、STRING、VARCHAR、VARCHAR2 および CHARF 以外の外部データ型はサポートされません。

使用上の注意

標識変数は、SHORT 型で宣言します。実行が完了すると、sqlca.sqlerrd[2] にはエラーなしで取り出された複数の属性が戻されます。実行エラーが発生した場合は、エラーが発生した LOB の属性は、sqlca.sqlerrd[2] の内容より 1 つ多くなります。

DESCRIBE の例

ここで、任意の BFILE から DIRECTORY および FILENAME 属性を抽出する、簡単な Pro*C/C++ の例を示します。

次の OCIBFileLocator 宣言で型の解決およびコンパイルを正しく行うには、oci.h ヘッダー・ファイルが必要です。

```
#include <oci.h>
...
OCIBFileLocator *bfile ;
char directory[31], filename[256] ;
short d_ind, f_ind ;
```

最後に、LOB 表から BFILE ロケータを選択し、DESCRIBE を実行します。

```
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 ;
```

標識変数は、DIRECTORY および FILENAME 属性で使用するときにのみ有効です。属性値の保存に使用されるホスト変数バッファの大きさが不足している場合は、これらの属性値の文字列が切り捨てられることがあります。切捨てが発生する場合は、標識の値は属性の元の長さに設定されます。

LOB およびナビゲーション・インタフェース

17-12 ページの「[オブジェクトへのナビゲーション・アクセス](#)」で説明したナビゲーション・インタフェースは、LOB を属性として含むオブジェクト型の操作で使用することもできます。

一時オブジェクト

OBJECT CREATE 文を使用して、LOB 属性を持つ一時および永続オブジェクトを作成します。テンポラリ LOB を一時オブジェクトの LOB 属性に ASSIGN し、永続 LOB または永続オブジェクトの LOB 属性に値をコピーしてデータを保存します。または、テンポラリ LOB を LOB 属性に ASSIGN し、FLUSH を使用してデータベースに値を書き込みます。

BFILE 属性の一時オブジェクトを作成して、ディスク上の BFILE からデータを読み込むことができます。テンポラリ BFILE はサポートされていません。

永続オブジェクト

内部 LOB 属性が格納されたオブジェクト・キャッシュに永続オブジェクトを作成すると、LOB 属性は暗黙的に空に設定されます。まず、OBJECT FLUSH 文を使用してこのオブジェクトをフラッシュし、表に行を挿入して空の LOB を作成する必要があります。オブジェクト・キャッシュのオブジェクトを（VERSION=LATEST オプションを使用して）リフレッシュすると、実際のロケータが属性に読み込まれます。

BFILE 属性のオブジェクトを作成すると、BFILE は NULL に設定されます。BFILE を読み込む前に、有効なディレクトリ別名およびファイル名で更新する必要があります。

テンポラリ LOB は、永続オブジェクトの LOB 属性に ASSIGN されることがあります。オブジェクトがフラッシュされると、実際の LOB 値がコピーされます。COPY 文でテンポラリ LOB ロケータおよび LOB 属性のロケータを使用して、テンポラリ LOB の値を永続オブジェクトの LOB 属性に明示的にコピーすることもできます。

ナビゲーション・インタフェースの例

ナビゲーション・インタフェースで LOB を処理する場合は、OBJECT GET および SET 文を使用します。

オブジェクト型の属性の LOB ロケータを取り出し、新しい埋込み SQL LOB 文で使うことができます。OBJECT SET 文を使用して、LOB ロケータをオブジェクト型の属性に戻します。

この場合、直接 LOB ASSIGN 操作を実行した場合と同じ結果になります。型の変更など、LOB ASSIGN が実行された場合に適用されるオブジェクト型に対して、LOB 属性の OBJECT GET または SET を実行した場合にもこのルールが適用されます。

たとえば、次の簡単な型の定義を仮定します。

```
CREATE TYPE lob_type AS OBJECT (a_blob BLOB) ;
```


この例では、この型を有効な（初期化済の）BLOB 属性を持つデータベースの列と見なします。

Pro*C/C++ で使用できる OTT 生成の C 構造体は次のようになります（OTT の INTYPE ファイルの作成および OTT の実行は、19-1 ページの「オブジェクト型トランスレータ」の章で説明します）。

```
struct lob_type
{
    OCIBlobLocator *a_blob ;
} ;
typedef struct lob_type lob_type ;
```

Pro*C/C++ プログラムを作成して、DESCRIBE 文で BLOB 属性を抽出し、BLOB の現行の長さを取り出します。次に、TRIM で BLOB のサイズを半分に調整し、SET OBJECT で属性を元に戻してから、OBJECT FLUSH で変更を有効にします。

まず、oci.h をインクルードし、一部のローカル変数を宣言します。

```
#include <oci.h>
lob_type *lob_type_p ;
OCIBlobLocator *blob = (OCIBlobLocator *)0 ;
unsigned int length ;
```

オブジェクトから BLOB 属性を選択し、OBJECT GET および DESCRIBE を行って BLOB の現行の長さを取得します。

```
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 ;
```

長さを半分にし、BLOB を新しい長さに TRIM します。

```
length = (unsigned int)(length / 2) ;
EXEC SQL LOB TRIM :blob TO :length ;
```

BLOB が変更されると、BLOB 属性をオブジェクトに戻し、変更をサーバーに FLUSH し、コミットします。

```
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 ;
```

LOB プログラムの例

BFILE および BLOB の読み込みおよび書き込みの方法について、2 つの例を挙げます。

BLOB の READ およびファイル書き込みの例

この例では、長さが不明な任意の長さの BLOB からデータをバッファに読み込み、バッファから外部ファイルにそのデータを書き込みます。バッファが小さいため、読み込む BLOB のサイズに応じて、1 つの READ 文で BLOB 値をバッファに読み込める場合もありますが、標準ポーリング・モードを使用する必要がある場合もあります。

まず、oci.h および簡単なローカル変数を宣言します。

```
#include <oci.h>
OCIBlobLocator *blob ;
FILE *fp ;
unsigned int amt, offset = 1 ;
```

BLOB 値を格納し、ファイルに書き込むバッファが必要です。

```
#define MAXBUFLLEN 5000
unsigned char buffer[MAXBUFLLEN] ;
EXEC SQL VAR buffer IS RAW(MAXBUFLLEN) ;
```

BLOB ホスト変数を割り当て、READ する BLOB を選択します。

```
EXEC SQL ALLOCATE :blob ;
EXEC SQL SELECT a_blob INTO :blob FROM lob_table WHERE ... ;
```

BLOB 値を書き込む外部ファイルをオープンします。

```
fp = fopen((const char *)"image.gif", (const char *)"w") ;
```

1 回の READ ですべての LOB 値をバッファに読み込める場合は、シグナル LOB READ の終了に対して NOT FOUND 条件を取得する必要があります。

```
EXEC SQL WHENEVER NOT FOUND GOTO end_of_lob ;
```

最初の READ を行います。量パラメータは、最大値の 4 ギガバイトに設定します。バッファより大きい場合、LOB を読み込めない場合は、ポーリング・モードを使用して READ します。

```
amt = 4294967295 ;
EXEC SQL LOB READ :amt FROM :blob AT :offset INTO :buffer ;
```

この例の場合、バッファの大きさが不足しているため LOB 値をすべて格納できないので、読み込み済のデータはバイナリ I/O を使用して書き込みおよび読み込みを続行します。

```
(void) fwrite((void *)buffer, (size_t)MAXBUFLLEN, (size_t)1, fp) ;
```

標準ポーリング・モードを使用して、無限ループ内で LOB READ によって読み込みを続行します。ループを終了するには、NOT FOUND 条件を設定します。

```
EXEC SQL WHENEVER NOT FOUND DO break ;
while (TRUE)
{
    ポーリング中はオフセットが使用されないため、後続の LOB READ では省略できます。ただし、最後の READ のコールで READ された量が通知されるようにするため、量パラメータを指定します。

    EXEC SQL LOB READ :amt FROM :blob INTO :buffer ;
    (void) fwrite((void *)buffer, (size_t)MAXBUFLen, (size_t)1, fp) ;
}
```

LOB 値の最後に到達しました。量パラメータには、READ された最後のピースの量が保存されます。ポーリング中は、中間の各ピースの量が、MAXBUFLen、つまりバッファの最大サイズに設定されます。

```
end_of_lob:
(void) fwrite((void *)buffer, (size_t)amt, (size_t)1, fp) ;
```

この基本的な構造のコードでは、任意の長さの内部 LOB がローカル・バッファに READ され、外部ファイルに書き込まれます。OCI および PL/SQL をモデルにしています。詳細な情報は、『Oracle コール・インタフェース・プログラマーズ・ガイド』の付録を参照するか、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

ファイルの読み込みおよび BLOB の WRITE の例

この例では、長さが明確な任意の長さのファイルからデータをバッファに読み込み、バッファからデータを内部 LOB に書き込みます。バッファが小さいため、読み込むファイルのサイズに応じて、1つの WRITE 文でファイル・データを LOB に書き込める場合もありますが、標準ポーリング・モードを使用する必要がある場合もあります。

まず、oci.h および簡単なローカル変数を宣言します。

```
#include <oci.h>
OCIBlobLocator *blob ;
FILE *fp ;
unsigned int amt, offset = 1 ;
unsigned filelen, remainder, nbytes ;
boolean last ;
```

ファイル・データを格納し、LOB に書き込むバッファが必要です。

```
#define MAXBUFLen 5000
unsigned char buffer[MAXBUFLen] ;
EXEC SQL VAR buffer IS RAW(MAXBUFLen) ;
```

空の表の空の BLOB を初期化して、ALLOCATE されたロケータにその BLOB を取り出し、ファイルからデータをコピーします。

```
EXEC SQL ALLOCATE :blob ;
EXEC SQL INSERT INTO lob_table (a_blob) VALUES (EMPTY_BLOB())
RETURNING a_blob INTO :blob ;
```

バイナリ・ファイルをオープンして長さを決定します。BLOB に書き込む合計量が、バイナリ・ファイルの実際の長さになります。

```
fp = fopen((const char *)"image.gif", (const char *)"r") ;
(void) fseek(fp, 0L, SEEK_END) ;
filelen = (unsigned int)ftell(fp) ;
amt = filelen ;
```

バッファ・サイズに基づいて読み込むバイト数を決定し、ファイルの初期読み込みを設定します。

```
if (filelen > MAXBUFLLEN)
    nbytes = MAXBUFLLEN ;
else
    nbytes = filelen ;
```

ファイル I/O 操作を発行して n バイトのデータをファイル fp からバッファに読み込み、残りの読み込み量を決定します。ファイルの先頭から読み込みを開始します。

```
(void) fseek(fp, 0L, SEEK_SET) ;
(void) fread((void *)buffer, (size_t)nbytes, (size_t)1, fp) ;
remainder = filelen - nbytes ;
```

残りの読み込み量に応じて、1 ピースでバッファに書き込むか、ポーリングを開始し、複数の小さなピース単位でファイルのデータを書き込むポーリングを開始します。

```
if (remainder == 0)
{
```

この場合は、1 ピースでデータを書き込みます。

```
    EXEC SQL LOB WRITE ONE :amt
    FROM :buffer INTO :blob AT :offset ;
}
else
{
```

ポーリング方法を開始し、ピース単位でデータを LOB に書き込みます。ポーリング方法を開始するには、まず、最初の WRITE で FIRST キーワードを使用します。

```
    EXEC SQL LOB WRITE FIRST :amt
    FROM :buffer INTO :blob AT :offset ;
```

簡単なループを設定し、ポーリング・モードをインプリメントします。

```
last = FALSE ;
EXEC SQL WHENEVER SQLERROR DO break ;
do
{
```

ファイルから読み込み、宛先 LOB に WRITE するバイト数を計算します。また、読み込んだピースが LAST ピースかどうかを判断します。

```
if (remainder > MAXBUFLen)
    nbytes = MAXBUFLen ;
else
{
    nbytes = remainder ;
    last = TRUE ;
}
```

ファイル・システムのファイルから次の nbytes をバッファに読み込みます。ファイル読み中にエラーが発生した場合は、自動的に次の WRITE が LAST になるように設定します。

```
if fread((void *)buffer, (size_t)nbytes, (size_t)1, fp) != 1)
    last = TRUE ;
```

ここで、LAST ピースを WRITE するか、中間の NEXT ピースを WRITE します。NEXT ピースは、ファイルから読み込まれるデータが残っていることを示しています。

```
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)) ;
```

このコード例では、任意の長さのファイルをローカル・バッファに読み込み、LOB に書き込みます。これは、OCI の例をモデルにしています。詳細な情報は、『Oracle コール・インタフェース・プログラマーズ・ガイド』の付録を参照してください。

lobdemo1.pc

このプログラム lobdemo1.pc は、LOB 埋込み SQL 文の例です。ソース・コードは、demo ディレクトリにあります。アプリケーションでは、社会保険番号、氏名、および交通違反を集計したテキストが含まれる CLOB の列で構成される license_table という名前の表を使用します。標準的な自動車部門の簡単な SQL 操作をモデルにしています。

考えられるアクションは次のとおりです。

- 新しいレコードを追加します。
- 社会保険番号順にレコードを一覧にします。
- 任意の社会保険番号のレコードの情報を一覧にします。
- 既存の CLOB の内容に、新しい交通違反を追加します。

/*****

SCENARIO:

We consider the example of a database used to store driver's licenses. The licenses are stored as rows of a table containing three columns: the sss number of a person, his name in text and the text summary of the info found in his license.

The sss number is the driver's unique social security number.

The name is the driver's given name as found on his ID card.

The text summary is a summary of the information on the driver, including his driving record, which can be arbitrarily long and may contain comments and data regarding the person's driving ability.

APPLICATION OVERVIEW:

This example demonstrate how a Pro*C client can handle the new LOB datatypes through PL/SQL routines. Demonstrated are mechanisms for accessing and storing lobbs to tables and manipulating LOBs through the stored procedures available via the dbms_lob package.

*****/

/*****

To run the demo:

1. Execute the script, lobdemo1c.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 lobdemo1c.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 lobdemo1.pc code *
 *****/

#define EX_SUCCESS      0
#define EX_FAILURE      1

#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 BUFLLEN 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 *
 *****/

#ifdef __STDC__
    void GetDate( void );
    void PrintSQLException( 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 PrintSQLException( );
    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 PrintSQLException();

    printf("Connecting to license database account: %s \n", uid);
    EXEC SQL CONNECT :uid;

    GetDate();

    printf("\t*****\n");
    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]={'\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");
        return;
    }
    else
    {
        OCILobLocator *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=====");
        printf("\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 <= 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 lob and lob
*   performance were designed for large data. Therefore,
*   users are encouraged to read and write large chunks of
*   data to lob.

```

```
*/

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] = '\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);
}
```


この章では、Pro*C/C++ のユーザー定義オブジェクトのサポートについて説明します。
内容は次のとおりです。

- オブジェクトの概要
- Pro*C/C++ でのオブジェクト型の使用
- オブジェクト・キャッシュ
- 結合インタフェース
- ナビゲーション・インタフェース
- オブジェクト属性と C 型の変換
- オブジェクト・オプションの設定 / 取得
- オブジェクトに対する新しいプリコンパイラ・オプション
- Pro*C/C++ のオブジェクト例
- ナビゲーション・アクセスのサンプル・コード
- C 構造体の使用
- REF の使用
- OCIDate、OCIString、OCINumber および OCIRaw の使用
- Pro*C/C++ の新しいデータベース型の概要
- 動的 SQL での Oracle8i データ型使用の制限

オブジェクトの概要

Pro*C/C++ では、Oracle8 以降これまでサポートされていた Oracle のリレーショナル・データ型に加えて、次のユーザー定義データ型がサポートされます。

- オブジェクト型
- オブジェクト型の REF
- コレクション・オブジェクト型（第 18 章「コレクション」を参照してください。）

オブジェクト型

オブジェクト型は、ユーザー定義によるデータ型です。これには、CREATE TYPE SQL 文で変数として定義されるデータ型の属性と、オブジェクト型に適用できる動作としての関数およびプロシージャからなるメソッドが含まれます。このマニュアルでは、属性のみを持つオブジェクト型を考えます。

たとえば、次のとおりです。

```
--Defining an object type...
CREATE TYPE employee_type AS OBJECT(
    name    VARCHAR2(20),
    id      NUMBER,
    MEMBER FUNCTION get_id(name VARCHAR2) RETURN NUMBER);
/
--
--Creating an object table...
CREATE TABLE employees OF employee_type;
--Instantiating an object, using a constructor...
INSERT INTO employees VALUES (
    employee_type('JONES', 10042));
```

LONG、LONG RAW、NCLOB、NCHAR および NCHAR 可変幅データ型は、オブジェクト属性では使用できません。

REF

REF（参照）も Oracle8 での新機能です。オブジェクト自体の参照ではなく、データベース表に格納されているオブジェクトを参照します。REF 型は、リレーショナル列に指定できるだけでなく、オブジェクト型のデータ型としても指定できます。たとえば、次のように、表 *employee_tab* にオブジェクト型 *employee_t* 自体の REF を表す列を組み込むことができます。

```
CREATE TYPE employee_t AS OBJECT(
    empname    CHAR(20),
    empno      INTEGER,
    manager    REF employee_t);
/
CREATE TABLE employee_tab OF employee_t;
```

Pro*C/C++ でのオブジェクト型の使用

OTT (オブジェクト型トランスレータ) が生成した C 構造体へのポインタを、Pro*C/C++ アプリケーションでホスト変数と標識変数として宣言します。詳細は、[第 19 章「オブジェクト型トランスレータ」](#)を参照してください。オブジェクト型の場合、標識変数はオプションですが、使用することをお勧めします。

Pro*C/C++ プログラムではオブジェクト型を、OTT を使用してデータベース・オブジェクトから生成された C 構造体として表現します。次の操作を必ず実行してください。

- OTT によって生成されるヘッダー・ファイルに、構造体定義およびそれに対応付けられた NULL 標識構造体、オブジェクト型の REF を表す C 型を指定して、Pro*C/C++ プログラムに組み込みます。
- OTT 生成の型ファイルを Pro*C/C++ の INTYPE コマンドライン・オプションとして入力します。この型ファイルにより、OTT によって生成される C 構造体とそれに対応するデータベース内のオブジェクト型の間およびスキーマと型のバージョン情報との間の対応関係がコード化されます。

NULL 標識

オブジェクト型トランスレータによって、オブジェクト型の NULL ステータスを表す C 構造体が生成されます。生成されたこれらの構造体型をオブジェクト型の標識変数の宣言で使用する必要があります。

その他の Oracle8i 型では、NULL 標識に対して特別な処置は必要ありません。NULL 標識の詳細は、4-15 ページの「[標識変数](#)」を参照してください。

オブジェクト型には内部構造があるので、オブジェクト型を表す NULL 標識にも内部構造があります。コレクション・オブジェクト型以外のオブジェクト型を表す NULL 標識構造体は、オブジェクト型全体を表すアトミック (シングル) NULL ステータスのみでなく、すべての属性の NULL ステータスも提供します。OTT により、オブジェクト型を表す NULL 標識構造体を示す C 構造体が生成されます。NULL 標識構造体の名前は、Object_ttypename_ind です。この場合、Object_ttypename は、データベース内のユーザー定義型を表す C 構造体の名前です。

オブジェクト・キャッシュ

オブジェクト・キャッシュは、プログラムがデータベース・オブジェクトとのインタフェースに使用するために割り当てられたクライアントのメモリー領域です。オブジェクトには、2つのインタフェースが機能します。結合インタフェースはオブジェクトの一時コピーを操作し、ナビゲーション・インタフェースは永続オブジェクトを操作します。

永続オブジェクト対一時コピー

Pro*C/C++ で EXEC SQL ALLOCATE 文を使用してキャッシュに割り当てたオブジェクトは、Oracle データベース内では永続オブジェクトの一時コピーになります。したがって、これらのコピーはフェッチした後にキャッシュ内で更新できますが、その変更をデータベース内で永続的なものにするには、明示的な SQL コマンドを使用する必要があります。この一時コピーもしくは値ベースのオブジェクト・キャッシング・モデルはリレーショナル・モデルの拡張で、ここではリレーショナル表のスカラー列をホスト変数にフェッチすること、その場で更新することおよび更新結果をサーバーに通信することが可能です。

結合インタフェース

結合インタフェースは、オブジェクトの一時コピーを操作します。メモリーは、EXEC SQL ALLOCATE 文を使用してオブジェクト・キャッシュ内で割り当てます。

SQLLIB ランタイム・コンテキストごとに、オブジェクト・キャッシュが1つずつ作成されます。

各オブジェクトは、EXEC SQL SELECT 文または EXEC SQL FETCH 文によって取り出されます。この2つの文では、ホスト変数の属性値が設定されます。NULL 標識が与えられている場合は、それも設定されます。

オブジェクトの挿入、更新または削除には、EXEC SQL INSERT 文、EXEC SQL UPDATE 文および EXEC SQL DELETE 文を使用します。文が実行される前に、オブジェクトのホスト変数の属性を設定する必要があります。

トランザクション文 EXEC SQL COMMIT および EXEC SQL ROLLBACK は、変更をサーバーに永続的に書き込んだり、変更を異常終了するときに使用します。

EXEC SQL FREE 文を使用すると、オブジェクト・キャッシュ内のメモリーを明示的に解放できます。接続の終了時には、その割当て済メモリーが暗黙的に解放されます。

結合インタフェースを使用する場合

次のような場合に使用します。

- 表相互の明示的結合が面倒でないオブジェクトの大規模なコレクションにアクセスする場合。
- 参照できないオブジェクトにアクセスする場合。このようなオブジェクトには、個性がありません。たとえば、リレーショナル列内のオブジェクト型などです。
- 一連のオブジェクトに UPDATE や INSERT などの操作を適用する場合。たとえば、特定部門のすべての従業員に \$1000 のボーナスを追加する場合などです。

ALLOCATE

オブジェクト・キャッシュに領域を割り当てるには、次の文を使用します。構文は次のとおりです。

```
EXEC SQL [AT [:]database] ALLOCATE :host_ptr [[INDICATOR]:ind_ptr] ;
```

入力する変数は、次のとおりです。

database (IN)

前に次の文を使用して確立されたデータベース接続の名前を含むゼロ終了記号付き文字列。

```
EXEC SQL CONNECT :user [AT [:]database];
```

AT 句 AT を省略するか、データベースが空の文字列であれば、デフォルトのデータベース接続とみなされます。

host_ptr (IN)

OTT がオブジェクト型、コレクション・オブジェクト型、もしくは REF に対して生成したホスト構造体のポインタ、または新たな C のデータ型 OCIDate、OCINumber、OCIRaw および OCISString に対してのポインタ。

ind_ptr (IN)

標識変数 *ind_ptr* とキーワード INDICATOR はともにオプションです。構造体の型を持つ標識へのポインタに限り、ALLOCATE 文および FREE 文に指定できます。

host_ptr および *ind_ptr* には、ホスト配列を構成できます。

割当てはセッションが終了するまで有効です。どのインスタンスも、FREE 文で明示的に解放されなくても、セッション（接続）が終了すると解放されます。

詳細な情報は、F-12 ページの「[ALLOCATE（実行可能埋込み SQL 拡張要素）](#)」および F-62 ページの「[FREE（実行可能埋込み SQL 拡張要素）](#)」を参照してください。

FREE

```
EXEC SQL [AT[:]database] [OBJECT] FREE :host_ptr [[INDICATOR]:ind_ptr];
```

オブジェクト・キャッシュに格納されるオブジェクトの領域の割当てを解除するには、FREE 文を使用します。この文に使用する変数は、ALLOCATE 文の場合と同じです。

注意：ホスト変数や標識変数に対するポインタは NULL には設定されません。

CACHE FREE ALL

```
EXEC SQL [AT [:]database] [OBJECT] CACHE FREE ALL;
```

前述の文を使用すると、指定したデータベース接続用のオブジェクト・キャッシュ・メモリーがすべて解放されます。

詳細は、F-16 ページの「[CACHE FREE ALL \(実行可能埋込み SQL 拡張要素\)](#)」を参照してください。

結合インタフェースによるオブジェクトへのアクセス

SQL を使用してオブジェクトにアクセスする場合、Pro*C/C++ アプリケーションは永続オブジェクトの一時コピーを操作します。これは、SELECT、UPDATE および DELETE の各文を使用するリレーショナル・アクセス・インタフェースの直接の拡張要素です。

[図 17-1](#) では、永続オブジェクトの一時コピーに ALLOCATE 文でキャッシュを割り当てます。割り当てられるオブジェクトにデータは含まれていませんが、OTT によって生成される構造体の形式をとります。

```
person *per_p;  
...  
EXEC SQL ALLOCATE :per_p;
```

SELECT 文を実行してキャッシュを移入できます。また、FETCH 文や C 割当てを使用してキャッシュにデータを移入することもできます。

```
EXEC SQL SELECT ... INTO :per_p FROM person_tab WHERE ...
```

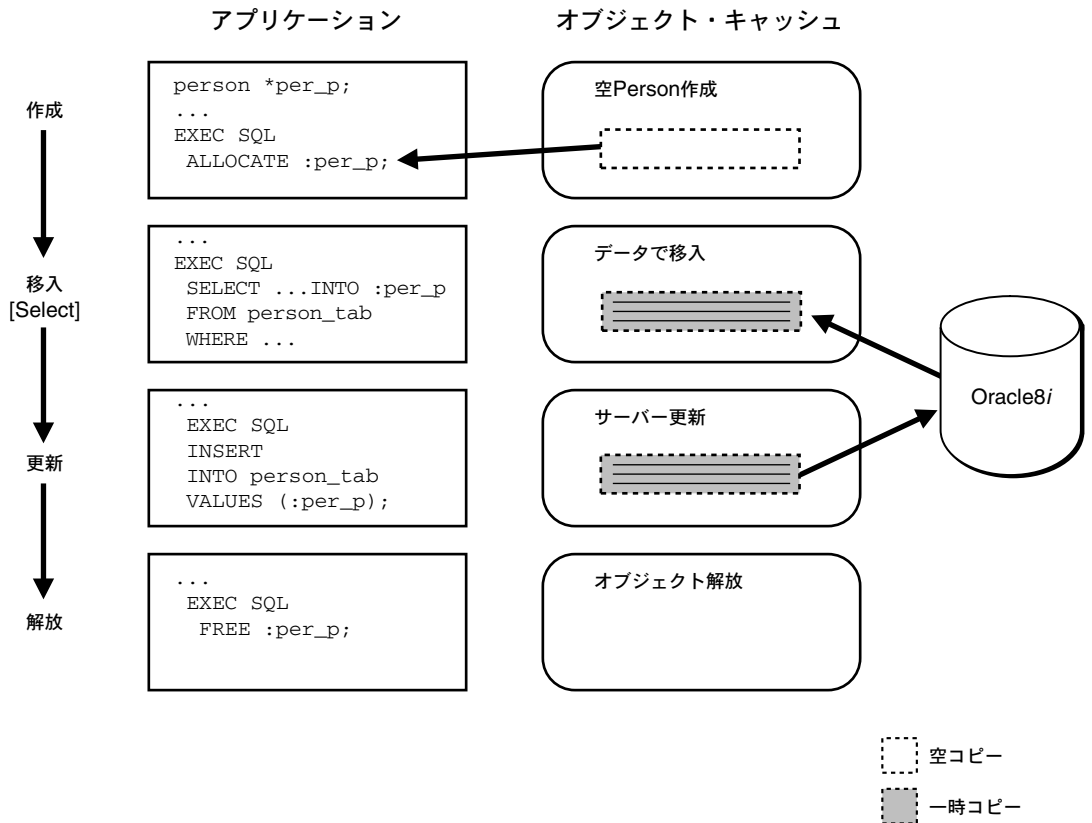
図のように、INSERT、UPDATE または DELETE 文を使用して、サーバー・オブジェクトを変更します。データは、次の INSERT 文を使用して表に挿入できます。

```
EXEC SQL INSERT INTO person_tab VALUES(:per_p);
```

最後に、FREE 文を使用して、オブジェクトのコピーに対応するメモリーを解放します。

```
EXEC SQL FREE :per_p;
```

図 17-1 SQL を使用してオブジェクトにアクセスする



ナビゲーション・インタフェース

ナビゲーション・インタフェースを使用すると、結合インタフェースと同じスキーマにアクセスできます。ナビゲーション・インタフェースはオブジェクトに対する REF の参照を解除して、あるオブジェクトから他のオブジェクトへと横断（ナビゲート）することで（永続および一時）オブジェクトにアクセスします。次に、一部の定義を示します。

Pinning とは、オブジェクトを "Dereferencing" するという意味の用語でしたが、プログラムがオブジェクトにアクセスできるようにすることです。

Unpinning とは、オブジェクトが不要になったことをキャッシュに対して示すことです。

Dereferencing とは、サーバーが REF を使用してクライアント内にそのオブジェクトの別版を作成することであると定義できます。キャッシュ内では、各オブジェクトとそれに対応するサーバー・オブジェクトとの間の対応付けがメンテナンスされますが、自動一貫性は得られません。キャッシュ内の各オブジェクトの内容の正確さと一貫性を確認するのは、ユーザーの責任です。

Releasing とは、オブジェクトのコピーがキャッシュに対してオブジェクトが現在使用されていないことを示すことです。メモリーを解放するために、不要になったオブジェクトをリリースして暗黙の解放にあてはまるようにします。

Freeing とは、オブジェクトのコピーがオブジェクトをキャッシュから削除し、メモリー領域を解放することです。

Marking とは、そのオブジェクト・コピーがキャッシュ内で更新されており、そのフラッシュ時に対応するサーバー・オブジェクトを更新する必要があることを、キャッシュに対して指示することです。

Un-marking とは、オブジェクトが更新されたことを示すマークを削除することです。

Flushing とは、キャッシュ内でマーク設定されたコピーに対するローカルの変更が、サーバー内のそれに対応するオブジェクトに書き込まれることです。この時点で、キャッシュ内のオブジェクト・コピーからもマークが解除されます。

Refreshing とは、オブジェクトのコピーがサーバー内のオブジェクトの最新値に置き換わることです。

ナビゲーション・インタフェースと結合インタフェースは、併用することができます。その点については、17-24 ページの「[ナビゲーション・アクセスのサンプル・コード](#)」にコードで示しています。

キャッシュ・コピーを更新、削除およびフラッシュする（キャッシュ内の変更をサーバーに書き込む）には、EXEC SQL OBJECT 文、ナビゲーション・インタフェースを使用します。

ナビゲーション・インタフェースを使用する場合

次の場合にナビゲーション・インタフェースを使用します。

- 表相互の明示的な結合が面倒な 1 組または小規模なオブジェクト・セットにアクセスする場合。参照 (DEREF) を使用してオブジェクト間でナビゲートする場合は、明示的結合より容易な暗黙的結合を 2 つの表全体の間で実行します。
- 多数の異なるオブジェクトに多数の小規模な変更を加える場合。すべてのオブジェクトをクライアントにフェッチし、変更し、更新済としてマーク設定してから、変更後のすべてのオブジェクトをフラッシュしてサーバーに戻すほうが手軽です。

ナビゲーション文に使用されるルール

埋込み SQL OBJECT 文は次の仮定のもとに説明されています。

- AT 句がない場合、デフォルト（名前なし）接続と見なされます。
- 特に指定する場合を除き、ホスト変数は配列になります。
- 配列ディメンションを明示的に指定するには、FOR 句を使用します。FOR 句がない場合、永続ホスト変数の最小ディメンションが使用されます。
- 文の実行後、SQLCA が状態変数として提供されると、処理されたエレメントの数は `sqlca.sqlerrd[2]` に戻されます。
- パラメータには、入力または出力を示す IN または OUT（あるいはその両方）が指定されています。

SQL OBJECT 文は、[付録 F「埋込み SQL 文および宣言文」](#)でアルファベット順に説明しています。付録 F には、構文図も含まれています。

OBJECT CREATE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT CREATE :obj [INDICATOR]:obj_ind
[TABLE tab] [RETURNING REF INTO :ref] ;
```

この場合、tab は次のとおりです。

```
{:hv | [schema.]table}
```

この文を使用して、オブジェクト・キャッシュ内で参照可能オブジェクトを作成します。オブジェクトの型は、ホスト変数 *obj* に対応します。オプションの型ホスト変数（*:obj_ind*、*:ref*、*:ref_ind*）を指定する場合は、すべてが同じ型に対応している必要があります。

参照可能オブジェクトは、永続オブジェクト（TABLE 句あり）でも一時オブジェクト（TABLE 句なし）でもかまいません。永続オブジェクトは、暗黙的に保持され、更新されたとしてマーク設定されます。一時オブジェクトは、暗黙的に保持されます。

ホスト変数は、次のとおりです。

obj (OUT)

オブジェクト・インスタンス・ホスト変数 *obj* は OTT が生成した構造体へのポインタである必要があります。この変数は、オブジェクト・キャッシュ内で作成される参照可能オブジェクトを判別するために使用されます。正常に実行されると、*obj* は新規作成されたオブジェクトを指します。

obj_ind (OUT)

この変数は OTT が生成した標識構造体を指します。その型は、オブジェクト・インスタンスのホスト変数の型と同じである必要があります。正常に実行されると、*obj_ind* は参照可能なオブジェクトに対するパラレル標識構造体へのポインタとなります。

tab (IN)

table 句を使用して永続オブジェクトを作成します。表名は、ホスト変数 *hv*、または未宣言の SQL 識別子として指定できます。スキーマ名で修飾することもできます。表名を含むホスト変数には、後続ブランクを使用しないでください。

hv (IN)

表を指定するホスト変数。ホスト変数を使用する場合、配列を使用しないでください。また、空白を埋め込まないでください。この文字列では、大文字と小文字を区別します。永続オブジェクトの配列を作成すると、すべてが同じ表に対応付けられます。

table (IN)

大 / 小文字が区別される未宣言の SQL 識別子。

ref (OUT)

参照ホスト変数は、OTT によって生成される参照の型へのポインタである必要があります。*ref* の型は、オブジェクト・インスタンスのホスト変数の型と同じである必要があります。実行後の *ref* には、新規作成されたオブジェクトの *ref* へのポインタが含まれます。

属性は、NULL に初期設定されるので注意してください。オブジェクト・ビューに対する新しいオブジェクトの作成は現在サポートされていません。

OBJECT Deref

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT Deref :ref INTO :obj  
[[:INDICATOR]:obj_ind] [FOR UPDATE [NOWAIT]] ;
```

オブジェクト参照 *ref* を指定すると、OBJECT Deref 文は指定された *ref* に対応するオブジェクトまたはオブジェクトの配列を、オブジェクト・キャッシュ内で保持します。これらのオブジェクトへのポインタは、変数 *obj* および *obj_ind* 内で戻されます。

ホスト変数は、次のとおりです。

ref (IN)

これはオブジェクト参照変数であり、OTT によって生成される参照の型へのポインタである必要があります。この変数（または変数の配列）は間接参照され、キャッシュ内のそれに対応するオブジェクトへのポインタを戻します。

obj (OUT)

オブジェクト・インスタンス・ホスト変数 *obj* は OTT が生成した構造体へのポインタである必要があります。その型は、オブジェクト参照のホスト変数の型と同じである必要があります。正常に実行されると、*obj* にはオブジェクト・キャッシュ内で保持されたオブジェクトへのポインタが含まれます。

obj_ind (OUT)

オブジェクト・インスタンス標識変数 *obj_ind* は OTT が生成した標識構造体へのポインタである必要があります。その型は、オブジェクト参照の標識変数の型と同じである必要があります。正常に実行されると、*obj_ind* には参照可能なオブジェクトに対するパラレル標識構造体へのポインタが含まれます。

FOR UPDATE

この句を指定すると、サーバー内でそれに対応するオブジェクト用に排他ロックが取得されます。

NOWAIT

このオプション設定のキーワードを指定すると、別のユーザーがすでにオブジェクトをロックしていた場合、ただちにエラーが戻されます。

OBJECT RELEASE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT RELEASE :obj ;
```

この文では、オブジェクト・キャッシュ内のオブジェクトが解放されます。オブジェクトが確保されず、更新されなければ、暗黙的な解放の対象になります。

オブジェクトが *n* 回 Dereference された場合は、オブジェクト・キャッシュから暗黙的に解放されるように、*n* 回リリースする必要があります。不要になったオブジェクトは、すべてリリースすることをお勧めします。

OBJECT DELETE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT DELETE :obj ;
```

永続オブジェクトの場合、この文はオブジェクト・キャッシュ内のオブジェクトまたはオブジェクトの配列を削除済としてマーク設定します。オブジェクトがサーバー内で削除されるのは、そのオブジェクトのフラッシュ時またはキャッシュのフラッシュ時です。オブジェクト・キャッシュ内で確保されているメモリーは解放されません。

一時オブジェクトの場合は、そのオブジェクトが削除済としてマーク設定されます。オブジェクト用のメモリーは解放されません。

OBJECT UPDATE

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT UPDATE :obj ;
```

永続オブジェクトの場合、この文はオブジェクト・キャッシュ内でオブジェクトを更新済としてマーク設定します。変更結果は、オブジェクトのフラッシュ時またはキャッシュのフラッシュ時に、サーバーに書き込まれます。

一時オブジェクトの場合、この文は何も操作しません。

OBJECT FLUSH

```
EXEC SQL [AT [:]database] [FOR [:]count] OBJECT FLUSH :obj ;
```

この文は、更新済、削除済または作成済としてマーク設定された永続オブジェクトをサーバーにフラッシュします。

注意：

オブジェクトがフラッシュされると、排他ロックが暗黙的に取得されます。

この文が正常終了すると、オブジェクトのマークが削除されます。

オブジェクトのバージョンが LATEST（次の項を参照）であれば、暗黙的にリフレッシュされます。

オブジェクトへのナビゲーション・アクセス

ナビゲーション・インタフェースの実例は、[図 17-2](#) を参照してください。

ALLOCATE 文を使用して、`person` オブジェクトを指す REF のコピー用に、オブジェクト・キャッシュ内のメモリーを割り当てます。割り当てられた REF には、データは含まれません。

```
person *per_p;  
person_ref *per_ref_p;  
...  
EXEC SQL ALLOCATE :per_p;
```

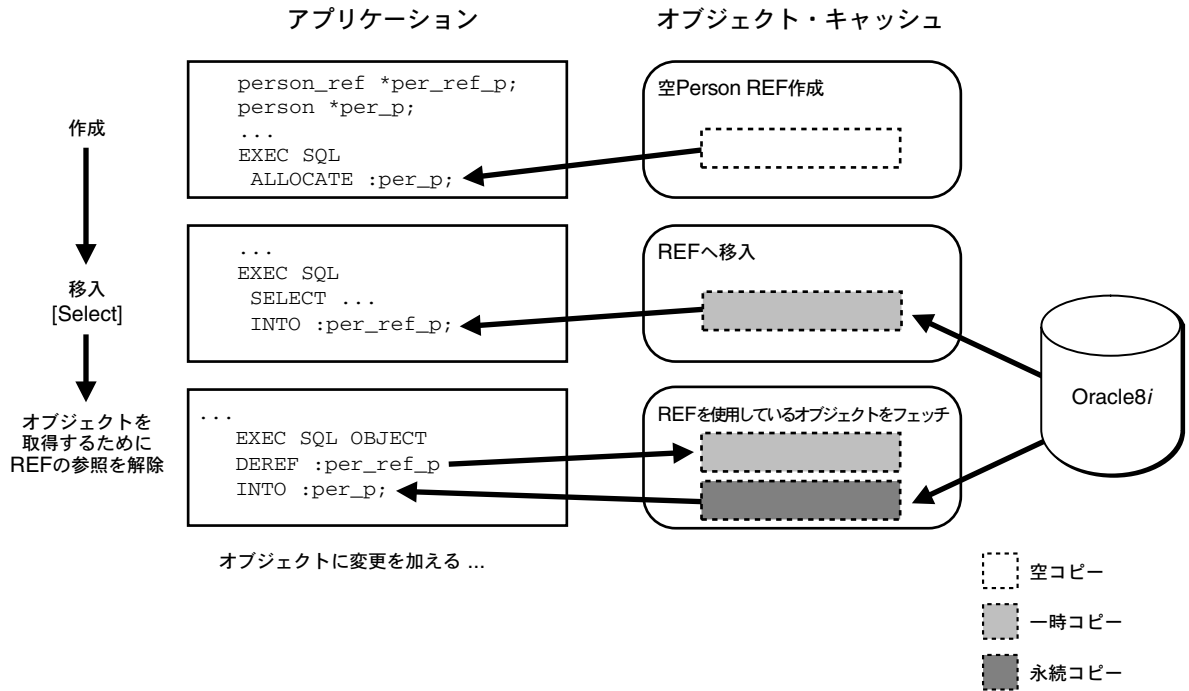
SELECT 文を使用して `person` オブジェクトの REF を取り出すことで、割り当てたメモリーを移入します。（正確なフォーマットはアプリケーション次第です。）

```
EXEC SQL SELECT ... INTO :per_ref_p;
```

次に、オブジェクト内で変更できるように、DEREF 文を使用してキャッシュ内でオブジェクトを確保します。DEREF 文は、ポインタ `per_ref_p` を使用して、クライアント側キャッシュ内で `person` オブジェクトのインスタンスを作成します。`person` オブジェクトへのポインタ `per_p` が戻されます。

```
EXEC SQL OBJECT DEREF :per_ref_p INTO :per_p;
```


図 17-2 ナビゲーション・アクセス



C 代入文を使用するか、または OBJECT SET 文によるデータ変換を使用して、キャッシュ内でオブジェクトを変更します。

次に、オブジェクトを更新済としてマーク設定する必要があります。図 17-3 を参照してください。キャッシュ内のオブジェクトを更新済としてマーク設定し、サーバーにフラッシュできるようにするには、次の文を使用します。

```
EXEC SQL OBJECT UPDATE :per_p;
```

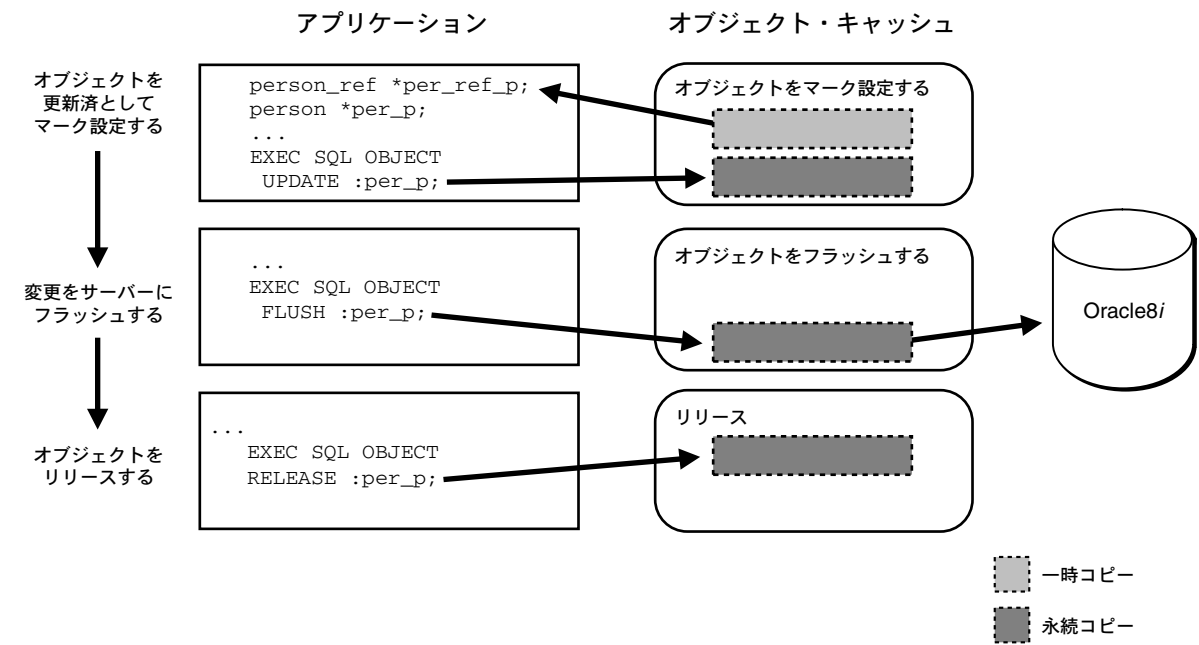
FLUSH 文によって変更結果をサーバーに送ります。

```
EXEC SQL OBJECT FLUSH :per_p;
```

オブジェクトをリリースします。

```
EXEC SQL OBJECT RELEASE :per_p;
```

図 17-3 ナビゲショナル・アクセス（続編）



オブジェクト属性と C の型の間で変換するには、次の項で説明する文を使用します。

オブジェクト属性と C 型の変換

OBJECT SET

```
EXEC SQL [AT [:]database]
  OBJECT SET [ { '*' | {attr[, attr]} } OF]
    :obj [[INDICATOR]:obj_ind]
    TO { :hv [[INDICATOR]:hv_ind]
        [, :hv [INDICATOR]:hv_ind]] ;
```

この文は結合インタフェースとナビゲショナル・インタフェースのどちらで作成したオブジェクトでも使用します。この文によって、オブジェクトの属性が更新されます。永続オブジェクトでは、変更はオブジェクトが更新され、フラッシュされたときにサーバーに書き込まれます。キャッシュのフラッシュは更新済オブジェクトになされたすべての変更をサーバーに書き込みます。

OF 句はオプション設定です。OF 句を指定しなければ、*obj* の属性がすべて設定されます。次のように記述しても同じ結果が得られます。

```
... OBJECT SET * OF ...
```

ホスト変数リストには属性の値を提供するように分解された構造体を組み込むことができます。ただし、*obj* 内の属性の数は、展開する変数リスト内の要素の数と同じである必要があります。

ホスト変数と属性は、次のとおりです。

attr

各属性はホスト変数ではなく、オブジェクトのどの属性が更新されるかを指定する識別子にすぎません。リスト内の最初の属性は、リスト内の最初の式と対になります。次も同様です。属性は OCIStrIng、OCINumber、CIDate もしくは OCIStrIng のどれかにする必要があります。

obj (IN/OUT)

obj では、更新対象となるオブジェクトを指定します。バインド変数 *obj* に配列を使用することはできません。これは OTT が生成した構造体へのポインタである必要があります。

obj_ind (IN/OUT)

更新対象となるパラレル標識構造体です。これは OTT が生成した標識構造体へのポインタである必要があります。

hv (IN)

これは、OBJECT SET 文への入力として使用されるバインド変数です。*hv* は int、float、OCIStrIng*、一次元の文字配列もしくはこれらの型の構造体である必要があります。

hv_ind (IN)

これは、OBJECT SET 文への入力として使用される結合標識です。*hv_ind* は 2 バイト整数スカラーもしくは 2 バイト整数スカラーの構造体である必要があります。

標識変数の使用：

ホスト変数の標識を提示する場合は、オブジェクト標識も提示してください。

hv_ind を -1 に設定すると、それに対応付けられたフィールドが *obj_ind* 内で -1 に設定されます。

次の暗黙的な変換が許されます。

- [OCIStrIng | STRING | VARCHAR | CHARZ] から OCIStrIng へ
- OCIStrIng から OCIStrIng へ
- [OCINumber | int | float | double] から OCINumber へ
- [OCIDate | STRING | VARCHAR | CHARZ] から OCIDate へ

注意：

- ネストされた構造体は実現されません。
- この文を使用して、参照可能オブジェクトをアトミック NULL に設定することはできません。かわりに、NULL 標識の適切なフィールドを設定してください。
- OCIDateTime あるいは OCIInterval データ型と OCISString との間での変換はサポートされていません。

OBJECT GET

```
EXEC SQL [AT [:]database]
  OBJECT GET [ { '*' | {attr[, attr]} } FROM
    :obj [[INDICATOR]:obj_ind]
    INTO { :hv [[INDICATOR]:hv_ind]
      [, :hv [[INDICATOR]:hv_ind]] } ;
```

この文では、オブジェクトの属性がネイティブな C の型に変換されます。

FROM 句はオプションです。FROM 句を指定しなければ、*obj* の属性がすべて変換されます。次のように記述しても同じ結果が得られます。

```
... OBJECT GET * FROM ...
```

ホスト変数リストには属性の値を受け取るように分解された構造体を組み込んでもかまいません。ただし、*obj* 内の属性の数は、展開されるホスト変数リスト内の要素の数と同じである必要があります。

ホスト変数と属性は、次のとおりです。

attr

各属性はホスト変数ではなく、オブジェクトのどの属性が取り出されるかを指定する識別子にすぎません。リスト内の最初の属性は、リスト内の最初の式とペアになっており、2 番目以降の属性と式も同じようにペアになっています。属性は、ベース型を表すもの、つまり OCISString、OCINumber、OCIRef もしくは OCIDate にする必要があります。

obj (IN)

この変数では、属性を取り出すときのソースとして機能するオブジェクトを指定します。バインド変数 *obj* に配列を使用することはできません。

hv (OUT)

これは、OBJECT GET 文からの出力を保持するためのバインド変数です。これは int、float、double、一次元の文字配列もしくはこれらの型の構造体にできます。この文では、このホスト変数に変換済の属性値が戻されます。

hv_ind (OUT)

これは、属性値に対応付けられた標識変数です。これは 2 バイト整数スカラーもしくは 2 バイト整数スカラーの構造体です。

標識変数の使用：

オブジェクト標識を指定しなかった場合、属性は有効とみなされます。オブジェクトがアトミック NULL の場合または、要求した属性が NULL で、オブジェクト標識変数を指定しなかった場合は、オブジェクト属性が C の型に変換されるプログラム・エラーになります。この状況では、Oracle エラーを呼び出せないことがあります。

オブジェクト変数がアトミック NULL の場合、または要求した属性が NULL で、ホスト変数標識 (*hv_ind*) が与えられている場合は、-1 に設定されます。

オブジェクトがアトミック NULL の場合、または要求した属性が NULL で、ホスト変数標識が与えられていない場合は、エラーが発生します。

次の暗黙的な変換ができます。

- OCIStrIng から [STRING | VARCHAR | CHARZ | OCIStrIng] へ
- OCINumber から [int | float | double | OCINumber] へ
- OCIStrRef から OCIStrRef へ
- OCIDate から [STRING | VARCHAR | CHARZ | OCIDate] へ

注意：

- ネストされた構造体は実現されません。
- OCIDateTime あるいは OCIInterval データ型と OCIStrIng との間での変換はサポートされていません。

オブジェクト・オプションの設定 / 取得

ランタイム・コンテキストにはランタイム・コンテキストが生成され、割り当てられたときにデフォルトの値が設定されるオプションがあります。これらのオプションは次の埋込み SQL 宣言文で設定します。

CONTEXT OBJECT OPTION SET

```
EXEC SQL CONTEXT OBJECT OPTION SET {option[, option]} TO { :hv[, :hv] } ;
```

変数は、次のとおりです。

:hv(IN) ...

入力バインド変数 *hv* ... は STRING 型、VARCHAR 型もしくは CHARZ 型です。

option...

ランタイム・コンテキストのどのオプションを更新するかを指定する単純な識別子です。最初のオプションは最初の入力バインド変数と対になります。次も同様です。現在サポートされている値を次に示します。

表 17-1 CONTEXT OBJECT OPTION 値の有効な選択肢

オプション値	指定
DATEFORMAT	Date 属性とコレクション要素のフォーマット
DATELANG	Date 型と Datetime 型すべての言語

例を示します。

```
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;
...
```

注意: 使用可能なフォーマットの完全な説明は、「Oracle8i SQL リファレンス」を参照してください。F-31 ページの「CONTEXT OBJECT OPTION SET (実行可能埋込み SQL 拡張要素)」を参照してください。

CONTEXT OBJECT OPTION GET

影響されるコンテキストは、その時点で使用されているコンテキストと解釈されます。これらのオプションの値を判断するには、次の宣言文を用います。

```
EXEC SQL CONTEXT OBJECT OPTION GET {option[, option]} INTO {:hv[, :hv]} ;
```

オプションの値は表 17-1 「CONTEXT OBJECT OPTION 値の有効な選択肢」にあります。

出力に使用されるバインド変数 hv ... は STRING 型、VARCHAR 型もしくは CHARZ 型です。影響されるコンテキストは、その時点で使用されているコンテキストと解釈されます。

注意: 使用可能なフォーマットの完全な説明は、「Oracle8i SQL リファレンス」を参照してください。F-30 ページの「CONTEXT OBJECT OPTION GET (実行可能埋込み SQL 拡張要素)」を参照してください。

オブジェクトに対する新しいプリコンパイラ・オプション

オブジェクトをサポートするには、次のプリコンパイラ・オプションを使用します。

VERSION

このオプションでは、EXEC SQL OBJECT DEREf 文によってどのバージョンのオブジェクトが戻されるかが決まります。これにより、キャッシュ・オブジェクトとサーバー・オブジェクトの間で、一貫性レベルを変更できます。

EXEC ORACLE OPTION 文を使用してインラインで設定します。設定できる値は、次のとおりです。

RECENT (デフォルト) カレント・トランザクション内でオブジェクトがオブジェクト・キャッシュに選択されている場合は、そのオブジェクトが戻されます。オブジェクトが選択されていない場合は、サーバーから取り出されます。シリアル化された状態で実行中のトランザクションの場合、このオプションの動作は VERSION=LATEST と同じですが、ネットワーク往復回数はそれほど多くありません。この値は、ほとんどの Pro*C/C++ アプリケーションで問題なく使用できます。

LATEST オブジェクトがオブジェクト・キャッシュに存在しない場合は、データベースから取り出されます。オブジェクト・キャッシュに存在している場合は、サーバーからリフレッシュされます。この値を指定すると、ネットワークの往復回数が増大するので、慎重に使用してください。この値を使用するのは、オブジェクト・キャッシュをサーバー側バッファとできるだけ一貫性のある状態にしておく必要がある場合です。

ANY オブジェクトがすでにオブジェクト・キャッシュに存在している場合は、そのオブジェクトが戻されます。オブジェクトがオブジェクト・キャッシュに存在しなければ、サーバーから取り出されます。この値を指定すると、ネットワークの往復回数は最小になります。この値を使用するのは、アプリケーションが読取り専用オブジェクトにアクセスする場合や、ユーザーがオブジェクトに排他アクセスする場合です。

DURATION

このプリコンパイラ・オプションは、後続の EXEC SQL OBJECT CREATE 文と EXEC SQL OBJECT DEREf 文に使用される保持期間を設定するときに使用します。キャッシュ内のオブジェクトは、保持期間の終わりに暗黙的に解放されます。

ナビゲーション・インタフェースでのみ使用します。

このオプションは、EXEC ORACLE OPTION 文で設定できます。設定できる値は、次のとおりです。

TRANSACTION (デフォルト) オブジェクトは、トランザクションの完了時に暗黙的に解放されます。

SESSION オブジェクトは、接続の終了時に暗黙的に解放されます。

OBJECTS

このプリコンパイラ・オプションを指定すると、オブジェクト・キャッシュを使用できます。

DBMS=NATIVE | V8 に対する OBJECTS のデフォルト値は YES です。オブジェクト・キャッシュのデフォルト・サイズは、OCI デフォルト・キャッシュ・サイズと同じく、8 メガバイトです。

10-32 ページの「[OBJECTS](#)」を参照してください。

INTYPE

プログラムでオブジェクト型、コレクション・オブジェクト型または REF 型を使用する場合は、このコマンドライン・オプションで INTYPE ファイルを指定する必要があります。

次の構文で、INTYPE オプションを指定します。

```
INTYPE=filename1 INTYPE=filename2 ...
```

この場合、*filename1*、*filename2...* は、OTT で生成された型ファイルの名前です。これらのファイルは Pro*C/C++ への読取り専用入力ファイルになります。それに含まれる情報は単純なテキスト形式ですが、エンコードされている場合もあり、ユーザーが読める形式になっているとは限りません。

Pro*C/C++ の 1 つのプリコンパイル単位に対する入力ファイルとして、複数の INTYPE ファイルを指定できます。

このオプションは、EXEC ORACLE 文内でインラインで使用することはできません。

OTT は、データベース内で作成されたオブジェクト型を表す C の構造体の宣言を生成し、型ファイルと呼ばれるファイルに型の名前とバージョン情報を書き込みます。

オブジェクト型の名前は、それを表す C の構造体の型や C++ クラスの型と同じであるとは限りません。これには、次の理由が考えられます。

- サーバーに指定されたオブジェクト型の名前に、C または C++ 識別子で無効な文字が含まれている。
- ユーザーが OTT に対して、構造体またはクラスに異なる名前を使用するように要求した。
- ユーザーが OTT に対して、名前の大文字を小文字に、小文字を大文字に変更するように要求した。

前述の状況では、構造体やクラスの宣言からは、その構造体やクラスがどのオブジェクト型と合致するかを推論することができません。この情報は Pro*C/C++ に必要であり、OTT によって型ファイル内で生成されます。

ERRTYPE

ERRTYPE=*filename*

エラーは、画面のみでなく、指定したファイルにも書き込まれます。このオプションを省略すると、エラーは画面にのみ出力されます。ただし、ERRTYPE は 1 つしか指定できません。値が 1 つしかない他のコマンドライン・オプションと同じように、コマンドラインで ERRTYPE に対して複数の値を入力すると、最後の値がすべてに優先します。

このオプションは、EXEC ORACLE 文内でインラインで使用することはできません。

オブジェクトに対する SQLCHECK のサポート

各オブジェクト型とその属性は、Oracle 型の C バインディングに従って C プログラムに表されます。プリコンパイラ・コマンドライン・オプション SQLCHECK を SEMANTICS または FULL に設定すると、Pro*C/C++ はプリコンパイル中に、ホスト変数型がデータベース・スキーマ内でその型に対して必須の C バインディングに準拠しているかどうかを検証します。さらに、Oracle 型がプログラム実行中に正しくマップされているかどうかを検証するためにランタイム・チェックが常に行われます。10-37 ページの「SQLCHECK」を参照してください。

リレーショナル・データ型は通常の方法でチェックされます。

リレーショナル SQL データ型とホスト変数型は、両方の型が同一の場合または両方の型の間で変換が可能な場合に、互換性を持ちます。一方、オブジェクト型が互換性を持つのは、両方の型が同一の場合のみです。それらの型を次のように指定する必要があります。

- 同じ名前にします。
- 同じスキーマに含めます（スキーマが明示的に指定されている場合）。

オプション SQLCHECK=SEMANTICS または FULL を指定すると、Pro*C/C++ はプリコンパイル中に、指定されたユーザー ID とパスワードを使用してデータベースにログインし、構造体の宣言が生成されたオブジェクト型と、埋込み SQL 文に使用されたオブジェクト型が同じかどうかを検証します。

実行時のタイプ・チェック

Pro*C/C++ は、ある型について、入力 INTYPE ファイルからオブジェクト、コレクション・オブジェクトおよび REF ホスト変数の型の名前およびバージョン、可能な場合にはスキーマ情報を収集し、これらの情報を生成したコードに格納します。これにより、実行時にオブジェクトおよび REF バインド変数の型情報にアクセスできます。型が同じでない場合は、固有のエラー・メッセージが戻されます。

Pro*C/C++ のオブジェクト例

簡単なオブジェクトの例を検証してみましょう。型 *person* と表 *person_tab* を生成します。この表にはこれもオブジェクト型の列 *address* が含まれています。

```
create type person as object (  
    lastname      varchar2(20),  
    firstname     char(20),  
    age           int,  
    addr          address  
)  
/  
create table person_tab of person;
```

表にデータを挿入し、処理します。

結合アクセス

Pro*C/C++ を使用して、*lastname* の値を "Smith" から "Smythe" に変更する方法を考えてみましょう。

OTT を実行して、*person* へマップする C の構造体を生成します。Pro*C/C++ プログラムに、OTT によって生成されるヘッダー・ファイルを組み込む必要があります。

アプリケーション内で、クライアント側キャッシュ内の永続メモリーへのポインタ、*person_p* を宣言します。それからメモリーを割り当てて、戻されたポインタを使用します。

```
char *new_name = "Smythe";  
person *person_p;  
...  
EXEC SQL ALLOCATE :person_p;
```

これで、永続オブジェクトのコピーにメモリーが割り当てられます。割当て済オブジェクトには、まだデータは含まれていません。

C の代入文もしくは SELECT あるいは FETCH 文を使用して既存のオブジェクトを取り出して、キャッシュにデータを移入します。

```
EXEC SQL SELECT VALUE(p) INTO :person_p FROM person_tab p WHERE lastname = 'Smith';
```

キャッシュ内のコピーに対する変更結果は、INSERT 文、UPDATE 文および DELETE 文を使用してサーバー・データベースに送信します。

```
EXEC SQL OBJECT SET lastname OF :person_p TO :new_name;  
EXEC SQL INSERT INTO person_tab VALUES(:person_p);
```

次の文を使用してキャッシュ・メモリーを解放します。

```
EXEC SQL FREE :person_p;
```

ナビゲーション・アクセス

オブジェクト・キャッシュ内にオブジェクト `person` への REF のコピーのためにメモリーを割り当てます。ALLOCATE 文は REF へのポインタを戻します。

```
person *person_p;
person_ref *per_ref_p;
...
EXEC SQL ALLOCATE :per_ref_p;
```

割当て済の REF には、データが含まれていません。データを移入するには、オブジェクトの REF を取り出します。

```
EXEC SQL SELECT ... INTO :per_ref_p;
```

それから REF を間接参照して、オブジェクトのインスタンスをクライアント側のキャッシュに入れます。参照 (DEREF) コマンドは、`per_ref_p` を使用して、キャッシュ内で対応するオブジェクトのインスタンスを作成します。

```
EXEC SQL OBJECT DEREF :per_ref_p INTO :person_p;
```

C 割当てを使用するか、または OBJECT GET 文を使用して、キャッシュ内のデータを変更します。

```
/* 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;
```

変更をデータベース内で永続的なものにするために、FLUSH を使用します。

```
EXEC SQL OBJECT FLUSH :person_p;
```

サーバーが変更されたので、オブジェクトをリリースできます。リリースされるオブジェクトが、オブジェクト・キャッシュ・メモリーからただちに解放されるとは限りません。最近の使用頻度が最も低いスタックに置かれます。キャッシュがいっぱいになると、オブジェクトはメモリーからスワップされます。

リリースされるのはオブジェクトのみで、そのオブジェクトを指す REF はキャッシュ内に残留します。REF をリリースするには、REF 用の RELEASE 文を使用します。`person_p` が指すオブジェクトをリリースするには次のようにします。

```
EXEC SQL OBJECT RELEASE :person_p;
```

または、保持継続時間が適切に設定されていれば、トランザクション・コミットを発行すると、キャッシュ内のオブジェクトがすべてリリースされます。

ナビゲーション・アクセスのサンプル・コード

サンプル・コードは3つのオブジェクト型を生成します。budoka は武道の専門家です。

- customer
- budoka
- location

さらに、次の2つの表が作成されます。

- person_tab
- customer_tab

次の SQL ファイル navdemo1.sql は型と表を生成してから、表に値を挿入します。

```
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,
    age        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;

```

OTT (オブジェクト型トランスレータ) の Intype ファイルは 19-8 ページの「[Intype ファイル](#)」で説明しています。このファイルを用意し、OTT への入力として使用します。

次が Intype ファイル navdemo1.typ のリストです。

```

case=lower
type location
type budoka
type customer

```

OTT が生成するヘッダー・ファイル navdemo1.h は、`#include` プリプロセッサ宣言文の付いたプリコンパイラのコードに組み込まれます。

プリコンパイラ・コード内のコメントを読み込みます。プログラムは新しい budoka オブジェクト (Jackie Chan のもの) を追加してから、customer_tab 表のすべての顧客を表示します。

次が Intype ファイル navdemo1.typ のリストです。

```
/*
 *
 * This is a simple Pro*C/C++ program designed to illustrate the
 * Navigational access to objects in the object cache.
 *
 * To build the executable:
 *
 * 1. Execute the SQL script, navdemo1.sql in SQL*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 navdemo1 intype=navdemo1_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 SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL 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 */
    customer_ind *cust_ind; /* Pointer to indicator struct for customer */
    customer_ref *cust_ref; /* Pointer to customer object reference */
    budoka *budo_p; /* Pointer to budoka object */
    budoka_ref *budo_ref; /* Pointer to budoka object reference */
    budoka_ind *budo_ind; /* Pointer to indicator struct for budoka */
}
```

```
/* 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"));

/* Convert native C types to OTS types - Note also the REF type */
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 below */
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;

/* Associatively select a REF to the newly created budoka object */
EXEC SQL SELECT REF(p) INTO :budo_ref FROM person_tab p
    WHERE p.lastname = 'Chan';

/* Pin the budoka REF, returning a pointer to the budoka object */
EXEC SQL OBJECT Deref :budo_ref INTO :budo_p;

/* Mark the new budoka object as deleted in the object cache */
EXEC SQL OBJECT DELETE :budo_p;

/* Flush the changes, effectively deleting the budoka object */
EXEC SQL OBJECT FLUSH :budo_p;
```

```
/* Finally, free all object cache memory and log off */
EXEC SQL OBJECT CACHE FREE ALL;

EXEC SQL COMMIT WORK RELEASE;

exit(EXIT_SUCCESS);
}
```

プログラムの実行結果は次のとおりです。

```
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
```

C 構造体の使用

Oracle8 以前は、Pro*C/C++ の SQL SELECT 文では、C 構造体を 1 つのホスト変数として指定できました。その場合、構造体の各メンバーは、リレーショナル表の 1 つのデータベース列に対応付けられます。つまり、各メンバーは問合せによって戻される選択リスト内の 1 つの項目を表します。

Oracle8i の場合、データベース内のオブジェクト型は、1 つのエンティティであり、1 つの項目として選択できます。このため、Oracle7 の表記法では構造体はスカラー変数のグループなのか、それともオブジェクトなのかといったあいまいさがあることが導き出されます。

Pro*C/C++ では、次の規則を利用してこのあいまいさを解消しています。

OTT を使用して C 宣言が生成された場合に限り、C 構造体のホスト変数がオブジェクト型を表すと見なされます。そのため、型記述は Pro*C/C++ への INTYPE オプションで指定される型ファイルに表示されます。他のすべてのホスト構造体は、データベースに同じ名前のデータ型が存在する場合でも Oracle7 構文が使用されているものと見なされます。

したがって、既存の構造体ホスト変数の型と同じ名前を持つ新しいオブジェクト型を使用する場合は、Pro*C/C++ では INTYPE ファイル内のオブジェクト型定義が使用されることに注意してください。これは、コンパイル・エラーの原因となる場合があります。この修正には、既存のホスト変数の型を改名するか、または OTT を使用してオブジェクト型に新しい名前を付けます。

前述の規則は、OTT 生成のデータ型に対して別名指定されるユーザー定義のデータ型にまで広く適用されます。例として、emptytype がヘッダー・ファイル dbtypes.h 内で OTT によって生成された構造体であり、Pro*C/C++ プログラムに次の文を組み込んだ場合を考えます。

```
#include <dbtypes.h>
typedef emptytype myemp;
myemp *employee;
```

変数 *employee* を表す型の名前 *myemp* は、データベース内で定義されたあるオブジェクト型を表す OTT 生成の型の名前 *emptytype* に対して別名指定されます。これによって、Pro*C/C++ では、変数 *employee* はオブジェクト型を表すものとみなされます。

前述の規則は、OTT 生成の型を持つ C 構造体や、OTT 生成の型に対して別名指定されている C 構造体を、オブジェクト型以外の型データのフェッチに使用できないという意味ではありません。たった 1 つの含意は Pro*C/C++ は自動的にそうした構造体を拡張しないということです。ユーザーは自由に一般的な構文を用いて、構造体の個々のフィールドを単一データベースの列の選択や更新に使用できます。

REF の使用

REF 型はオブジェクト自体ではなく、オブジェクトの参照を示します。REF 型は関係列のみでなく、オブジェクト型の属性としても指定できます。

REF の C 構造体の生成

オブジェクト型の REF の C での表現は、OTT により型の変換中に生成されます。たとえばデータベース内のユーザー定義型 PERSON への参照は C では「Person_ref」型で示されます。正確な型名は型変換時に有効な OTT オプションで決定されます。OTT により生成された型ファイルは Pro*C/C++ の INTYPE プリコンパイラ・オプションで指定する必要があります。また、OTT により生成されたヘッダーは #include を使用して Pro*C/C++ プログラムに組み込む必要があります。このスキーマにより、REF 型に対する適切な型チェックが Pro*C/C++ のプリコンパイル中に間違いなく実行されます。

REF 型では、OTT で特殊な標識構造体を生成する必要がありません。かわりに、2 バイトの符号付きスカラー標識が使用されます。

REF の宣言

Pro*C/C++ で REF 型を表すホスト変数は、該当する OTT 生成の型へのポインタとして宣言する必要があります。

オブジェクト型とは異なり、REF 型を表す標識変数は、2 バイトの符号付きスカラー型 OCInd として宣言されます。標識変数は本来オプションですが、Pro*C/C++ で宣言された各ホスト変数に対してそれぞれ 1 つずつ指定するようにプログラミングしてください。

埋込み SQL での REF の使用

REF 型は、オブジェクト・キャッシュに格納されています。しかし、REF 型を表す標識はスカラーであるため、キャッシュには割り当てられていません。この標識は通常ユーザー・スタックに格納されています。

REF 型を表すホスト構造体を埋込み SQL 文で指定する前に、EXEC SQL ALLOCATE コマンドを使用してオブジェクト・キャッシュ内の領域を割り当ててください。使用後、EXEC SQL FREE または EXEC SQL CACHE FREE ALL コマンドの自由な使用は 17-7 ページの「[ナビゲーション・インタフェース](#)」で説明しています。

スカラー標識変数のためのメモリーはオブジェクト・キャッシュに割り当てられていないので、REF 型を表す標識は ALLOCATE コマンドおよび FREE コマンドには使用できません。OCInd として宣言されたスカラー標識はプログラム・スタックに格納されています。ALLOCATE 文を指定すれば、実行時に、指定されたホスト変数のための領域がオブジェクト・キャッシュに割り当てられます。ナビゲーション・インタフェースでは、C 割当てではなく、EXEC SQL GET と EXEC SQL SET を使用してください。

Pro*C/C++ では、関連する SQL 文および埋込み PL/SQL ブロックでの REF ホスト変数の指定がサポートされています。

OCIDate、OCIStrIng、OCINumber および OCIRaw の使用

これらの OCI 型は、それぞれ日付、可変長ゼロ終了記号付き文字列、Oracle 番号および可変長バイナリ・データを表す新しい C の表現です。これらの型は、いくつかの面でこれまでの C の数量表現よりも機能的になっています。たとえば OCIDate 型はクライアント側のルーチンが日付演算を実行する準備をします。これは以前のリリースではサーバーでの SQL 文を必要としていました。

OCIDate、OCIStrIng、OCINumber、OCIRaw の宣言

OCI* 型は、OTT 生成の構造体でオブジェクト型の属性として表示され、Pro*C/C++ プログラムではオブジェクト型の一部として使用します。オブジェクト型として使用しない場合に、初心者レベルの C および Pro*C/C++ ユーザーは、これらの型のホスト変数を単独で宣言しないでください。経験豊富な Pro*C/C++ ユーザーは、これらの型の高い機能性を生かすように、それぞれの C ホスト変数を宣言してもかまいません。ホスト変数はこれらの型へのポインタ、つまり OCIStrIng*s として宣言される必要があります。対応づけられた（オプションの）標識は、2 バイトの符号付きスカラー、つまり OCInd s_ind として宣言されます。

埋込み SQL での OCI 型の使用

これらの型のホスト変数のための領域は、EXEC SQL ALLOCATE を使用してオブジェクト・キャッシュに割り当てられます。これらの型を表す（スカラー）標識変数は ALLOCATE および FREE コマンドには使用できないので注意してください。このような標識は、スタック上で静的に割り当てるかまたはヒープ上で動的に割り当てます。領域の割当

ては、EXEC SQL FREE 文または EXEC SQL CACHE FREE ALL 文を使用して解除できます。また、セッションの終わりには自動的に解除されます。この点は、17-7 ページの「[ナビゲーション・インタフェース](#)」で説明しています。

OCI 型の操作

年、月、日、時間などの各種日付コンポーネントの個別フィールドを持つ構造体である OCIDate を除いて、その他の OCI 型はカプセル化されています。つまり、外部ユーザーには見えません。現在、VARCHAR のような既存の C の型は Pro*C/C++ で処理されますが、この方法とは対照的に OCI ヘッダー・ファイル oci.h を組み込み、その関数を使用して DATE 算術を実行したり、これらの型と int や char のような C 固有の型の間で変換できます。

Pro*C/C++ の新しいデータベース型の概要

表 17-2 にオブジェクト・サポートのための新たなデータベース型を示します。

表 17-2 Pro*C/C++ での新しいデータベース型の使用

操作 ----- データベース型	DECLARE	ALLOCATE	FREE	MANIPULATE
オブジェクト型	ホスト：OTT で生成された C 構造体へのポインタ インジケータ：OTT で生成された標識構造体へのポインタ	結合インタフェース： EXEC SQL ALLOCATE ナビゲーション・インタフェース： EXEC SQL OBJECT CREATE ... EXEC SQL OBJECT DEREF ホスト変数および標識のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放する、またはセッションの終わりに自動的に解放します。	C ポインタを参照 (DEREF) して各属性を取得します。操作方法は属性の型によって異なります (下記参照)。
コレクション・オブジェクト型 (NESTED TABLE および可変長配列)	ホスト：OTT で生成された C 構造体へのポインタ インジケータ：OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放する、またはセッションの終わりに自動的に解放します。	第 18 章「コレクション」 を参照してください。 OCIColl* 関数 (oci.h に定義されている) を使用して、各要素を取得または設定します。

表 17-2 Pro*C/C++ での新しいデータベース型の使用

REF	ホスト : OTT で生成された C 構造体へのポインタ インジケータ : OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放する、またはセッションの終わりに自動的に解放します。	EXEC SQL OBJECT Deref を使用します。 ナビゲショナル・インタフェースで EXEC SQL OBJECT SET/GET を使用します。
LOB	ホスト : OCIBlobLocator *, OCIClobLocator * または OCIFileLocator * インジケータ : OCIInd	EXEC SQL ALLOCATE malloc() を使用してホスト変数用のメモリーをユーザー・ヒープ内で割り当てます。	EXEC SQL FREE を使用して解放する、またはすべての Pro*C/C++ 接続のクローズ時に自動的に解放します。EXEC SQL CACHE FREE ALL では、オブジェクトの LOB 属性のみ解放されます。	16-1 ページの「 ラージ・オブジェクト (LOB) 」を参照してください。 または、dbms_lob パッケージの埋込み PL/SQL ストアド・プロシージャを使用するか、あるいは oci.h に定義されている OCILob* 関数を使用します。
注意: Pro*C/C++ では、これらの型のホスト配列は、SQL のバルクフェッチまたは挿入操作で宣言および使用できます。				

表 17-3 に Pro*C/C++ での新たな C データ型の利用方法を示します。

表 17-3 Pro*C/C++ での新たな C データ型の使用

操作 ----- C 型	DECLARE	ALLOCATE	FREE	MANIPULATE
OCIDate	ホスト : OCIDate * インジケータ : OCIInd	EXEC SQL ALLOCATE ホスト変数のためのメモリーをオブジェクト・キャッシュに割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解放する、またはセッションの終わりに自動的に解放します。	(1) oci.h に定義された OCIDate* 関数を使用します。 (2) EXEC SQL OBJECT GET/SET を使用します。 または (3) oci.h に定義されている OCINumber* 関数を使用します。

表 17-3 Pro*C/C++ での新たな C データ型の使用

OCINumber	ホスト: OCINumber * インジケータ: OCIInd	EXEC SQL ALLOCATE ホスト変数の ためのメモ リーをオブ ジェクト・ キャッシュに 割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解 放する、またはセッシ ョンの終わりに自動的に解 放します。	(1) EXEC SQL OBJECT GET/SET を使用します。 または (2) oci.h に定義されてい る OCINumber* 関数を 使用します。
OCIRaw	ホスト: OCIRaw * インジケータ: OCIInd	EXEC SQL ALLOCATE ホスト変数の ためのメモ リーをオブ ジェクト・ キャッシュに 割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解 放する、またはセッシ ョンの終わりに自動的に解 放します。	oci.h に定義されている OCIRaw* 関数を使用し ます。
OCIString	ホスト: OCIString * インジケータ: OCIInd	EXEC SQL ALLOCATE ホスト変数の ためのメモ リーをオブ ジェクト・ キャッシュに 割り当てます。	EXEC SQL FREE または EXEC SQL CACHE FREE ALL を使用して解 放する、またはセッシ ョンの終わりに自動的に解 放します。	(1) EXEC SQL OBJECT GET/SET を使用します。 または (2) oci.h に定義されて いる OCIString* 関数を 使用します。
注意: Pro*C/C++ では、これらの型のホスト配列は、SQL のバルクフェッチまたは挿入操作で使用できない場合があります。				

Oracle8 での新しいデータ型は、REF、BLOB、NCLOB、CLOB および BFILE です。これらの型は、オブジェクト内またはリレーショナル列に使用することができます。どちらの場合でも、17-34 ページの「Pro*C/C++ での新しいデータベース型の使用」で示す C バインディングに従い、ホスト変数にマップされます。

動的 SQL での Oracle8i データ型使用の制限

Pro*C/C++ は、現在このような異なる種類の動的 SQL 方法をサポートしています。つまり、方法 1、方法 2、方法 3 および方法 4（ANSI および Oracle）です。これらの方法の詳細は、第 13 章「Oracle の動的 SQL」、第 14 章「ANSI 動的 SQL」および第 15 章「Oracle の動的 SQL 方法 4」を参照してください。

動的 SQL 方法 1、2 および 3 を使用すると、これまでに述べた Pro*C/C++ 拡張要素、つまり新しいオブジェクト型、REF、NESTED TABLE、可変長配列、NCHAR、NCHAR の可変長配列および LOB 型などをすべて処理できます。

旧版の動的 SQL 方法 4 は、基本的にリリース 8.0 以前の Pro*C/C++ でサポートされる Oracle 型に制限されています。NCHAR、NCHAR 可変幅および LOB データ型のホスト変数はサポートされています。動的 SQL 方法 4 は、オブジェクト型、NESTED TABLE、可変長配列および REF 型の処理には使用できません。

その代わり新しいアプリケーションにはすべて、Oracle8i で導入されたすべてのデータ型をサポートする ANSI 動的 SQL 方法 4 を使用してください。

コレクション

この章では、コレクションと呼ばれる他の種類のオブジェクト型と、Pro*C/C++ でコレクションの使用方法を説明します。コレクションとその要素にアクセスする方法を示します。

この章は主に次の項で構成されています。

- [コレクション](#)
- [コレクションの記述子](#)
- [OBJECT GET および SET](#)
- [COLLECTION 文](#)
- [コレクション・サンプル・コード](#)

コレクション

コレクションのオブジェクト型には、NESTED TABLE および VARRAY の 2 種類があります。

コレクション型は、リレーショナル列に指定できるだけでなく、オブジェクト型の属性としても指定できます。すべてのコレクションは、データベース内では必ず名前付きのオブジェクト型である必要があります。VARRAY の場合、最初にデータベース内に名前付きの型を作成して、希望する配列の要素の型と最大配列サイズを指定する必要があります。

NESTED TABLE

NESTED TABLE は、列内での要素と呼ばれる行の集まり（コレクション）です。データベース表の各行には、そのような要素が多数あります。簡単な例として、各従業員の作業リストがあります。この場合、多対 1 リレーションシップを 1 つの表に格納でき、従業員と作業の表を結合する必要はありません。

NESTED TABLE は、次の点で C および C++ 配列と異なります。

- 配列には一定の上限があります。NESTED TABLE は制限がありません。（索引最大値がありません。）
- 配列には連続する添字があり、稠密です。NESTED TABLE は、稠密または疎密で、添字は連続していません。NESTED TABLE がプログラムの配列に取り出されると、間隔はスキップされ、間隔のない稠密な配列が作成されます。

CREATE TYPE 文を使用して表型を定義します。表型は、リレーショナル表の 1 つ以上の列でその他のオブジェクト型の中にネストできます。

たとえば、組織の各部門に複数のプロジェクトを格納する場合を考えます。

```
CREATE TYPE project_type AS OBJECT (  
    pno          CHAR(5),  
    pname       CHAR(20),  
    budget      NUMBER(7,2)) ;  
  
CREATE TYPE project_table AS TABLE OF project_type ;  
  
CREATE TABLE depts (  
    dno          CHAR(5),  
    dname       CHAR(20),  
    budgets_limit NUMBER(15,2),  
    projects     project_table)  
    NESTED TABLE projects STORE AS depts_projects ;
```

VARRAY（可変長配列）

VARRAY 型を作成する場合、NESTED TABLE とは異なり、要素の最大数を指定する必要があります。NESTED TABLE では稠密または粗にできますが、VARRAY は粗にはできません。VARRAY および NESTED TABLE の要素は、両方とも 0 から始まります。

次のような CREATE TYPE 文で、VARRAY を作成します。

```
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
);
```

VARRAY は、リレーショナル表の列またはオブジェクト型の属性として使用します。この場合、各チームが最大 15 レコード（チームの名前を含みます）で構成されるリレーショナル表と比較して、記憶領域を節約できます。

C およびコレクション

C または C++ プログラムでは、NESTED TABLE はコレクションの索引値 0 から読み込みが開始されます。配列に NESTED TABLE を書き込むと、NESTED TABLE の要素は配列索引 0 で格納されます。配列に格納された NESTED TABLE が粗（索引に間隔があります）の場合、間隔はスキップされます。配列が NESTED TABLE に読み込まれると、間隔が再作成されます。

C または C++ プログラムでは、VARRAY は配列に書き込まれます。（索引 0 から開始します。）VARRAY に再び読み込まれると、要素は索引 0 から同じ順序でリストアップされます。このため、配列を使用するとコレクションに簡単にアクセスできます。

コレクションの記述子

NESTED TABLE の C 型は、OCITable へのポインタです。VARRAY の場合、OCIArray へのポインタになります（両方とも OCIColl へのポインタのサブタイプです）。OTT（オブジェクト型トランスレータ）ユーティリティを使用して、アプリケーション・コードに含めるヘッダー・ファイルに typedefs を生成します。19-1 ページの「[オブジェクト型トランスレータ](#)」を参照してください。

コレクションのホスト構造体は記述子で、これを介してコレクションの要素にアクセスできます。この記述子には、実際のコレクション要素は保持されていませんが、その要素へのポ

インタが格納されています。記述子とその関連する要素のためのメモリーは、オブジェクト・キャッシュで入手できます。

オブジェクト型における通常の手順に従って、OTT 生成の型ファイルを、Pro*C/C++ に対する INTYPE プリコンパイラ・オプション内と、`#include` プリプロセッサ宣言文を使用して Pro*C/C++ プログラムに組み込まれた OTT 生成のヘッダー内に指定する必要があります。このスキーマにより、コレクション・オブジェクト型に対する適切な型チェックがプリコンパイル中に間違いなく実行されます。

ただし、他のオブジェクト型とは異なり、コレクション・オブジェクト型は、OTT によって生成される特別な標識構造体を必要としません。かわりにスカラー標識が使用されます。その理由は、アトミック NULL 標識のみで十分にコレクション型全体が NULL かどうかかわるからです。個々のコレクション要素の NULL 状態は、(オプションで) 各要素に対応する別個の標識で表すことができます。

ホスト変数と標識変数の宣言

コレクション・オブジェクト型を表すホスト変数は、その他のオブジェクト型の場合と同様に、該当する OTT 生成の型へのポインタとして宣言する必要があります。

ただし、コレクション・オブジェクト型全体を表す標識変数は、その他のオブジェクト型の場合とは異なり、2 バイトの符号付きスカラー型 `OCIInd` として宣言されます。標識変数はオプションで設定しますが、Pro*C/C++ で宣言された各ホスト変数に対してそれぞれ1つずつ指定するようにプログラミングすることをお勧めします。

コレクションの操作

コレクションを操作する方法は2つあります。自立型エンティティと扱われ、コレクションの要素へのアクセスが発生しない場合と、コレクションの要素に対するアクセス、追加および切り捨てが発生する場合です。

自立型コレクション・アクセス

C コレクション記述子 (`OCITable` または `OCIArray`) を使用した場合、コレクション全体を1つのエンティティとして割り当てること以外ではできません。OBJECT GET 埋込み SQL 文 (17-14 ページの「[オブジェクト属性と C 型の変換](#)」を参照) を使用すると、コレクションが C ホスト変数記述子にバインドされます。OBJECT SET 文では、逆に C ホスト記述子がコレクションにバインドされます。

1つの文の中で複数のコレクションを1つの互換 C 記述子にバインドしたり、コレクションを C 記述子にバインドしている文の中で他のスカラーにバインドすることもできます。

コレクション要素アクセス

C コレクション記述子は、コレクションの要素にアクセスするために使用します。記述子には、始点およびエンド・ポイントなどのコレクションの内部属性およびその他の情報が含まれます。

要素のスライスは、互換データ型を持つホスト配列にバインドされます。コレクションのスライスとは、開始索引と終了索引間の内容のことです。スライスは配列にマップされます。配列のディメンションが、スライス要素数よりも大きいことがあります。

スカラーのバインドは、ホスト配列のディメンションを1にすること、またはオプションのFOR句が1に評価されることと同じことです。

アクセスのルール

自立型アクセス

- コレクションは1つのエンティティとして処理されるため、FOR句を使用することはできません。
- NESTED TABLE と VARRAY の定義方法が異なるため、2つの表の間で割当てを共有することはできません。
- 1つの文の中で、複数のコレクションを様々に組み合わせてC記述子に割り当てることができます。1つの文の中で、コレクションをC記述子に割り当てたり、他のスカラー・データ型をバインドすることもできます。

要素アクセス

- FOR句を使用することができます。省略すると、配列ディメンションの最小値が実行の繰り返し回数になります。
- 一度に複数のコレクションにアクセスすることはできません。

注意: FOR句の変数によって、処理される配列の要素数が指定されます。この値が最小の配列サイズを超過していないことを確認してください。この値は内部的には記号のついていない数量として扱われます。記号付きホスト変数を使用して負の値を渡そうとすると、予期できない動作が起こる結果になります。

標識変数

各アクセス方法で標識変数の使用方法が異なります。

自立型バインディング

1つの標識変数には、コレクションのNULL状態が1つのエンティティとして格納されます。各要素のNULL状態は格納されていません。

要素バインディング

標識変数には、要素がNULLであるかどうか格納されます。コレクション・データのスライスは、独自の標識配列のホスト配列にバインドされている場合は、標識配列には各要素のNULL状態がスライスに格納されます。

コレクション要素型がユーザー定義オブジェクト型の場合、ホスト変数に関連付けられた標識変数に、オブジェクトおよびその属性の NULL 状態が格納されます。

OBJECT GET および SET

OBJECT SET および OBJECT GET というナビゲーションルを使用すると、コレクション属性および定義したオブジェクト型の取出しおよび更新を行うことができます。

次の場合にオブジェクト型の要素に対して OBJECT GET 文を実行すると、オブジェクトのすべての属性がホスト変数に取り出されます。

```
'*' | {attr [, attr]} FROM
```

句が省略されるか「OBJECT GET *FROM」が使用される場合。

```
EXEC SQL [AT [:]database]
  OBJECT GET [ '*' | {attr [, attr]} FROM]
    :object [[INDICATOR] :object_ind]
      INTO { :hv [[INDICATOR] :hv_ind]
        [, :hv [[INDICATOR] :hv_ind]] } ;
```

次の場合に OBJECT SET 文を実行すると、オブジェクトのすべての属性に対してホスト変数を使用した更新が行われます。

```
'*' | {attr [, attr]} OF
```

句が省略されるか「OBJECT SET *OF」が使用される場合。

```
EXEC SQL [AT [:]database]
  OBJECT SET [ '*' | {attr [, attr]} OF]
    :object [INDICATOR] :object_ind]
      TO { :hv [[INDICATOR] :hv_ind]
        [, :hv [[INDICATOR] :hv_ind]] } ;
```

OBJECT GET および OBJECT SET 文の詳細は、17-14 ページの「[オブジェクト属性と C 型の変換](#)」を参照してください。

この表は、これら 2 つの文でオブジェクトとコレクション型をどのようにマッピングするかを示しています。

表 18-1 オブジェクトおよびコレクション属性

属性型	記述	ホスト・データ型
オブジェクト	OTT 生成の構造体	OTT 構造体へのポインタ
コレクション	OCIArray、OCITable (OCIColl)	OCIArray *, OCITable * (OCIColl *)

オブジェクトまたはコレクションはバウンドされる属性と型互換性を持つ必要があります。コレクション属性の型互換性は両方が VARRAY または NESTED TABLE のいずれかで、その要素型に互換性がある場合に限って保たれます。

次の表は 2 つのコレクション型の要素でどのように互換性を保てるかを示しています。

表 18-2 コレクションおよびホスト配列で可能な型変換

コレクション要素型	記述	ホスト・データ型
CHAR、VARCHAR、VARCHAR2	OCIString	STRING、VARCHAR、CHARZ、OCIString
REF	OCIRef	OCIRef
INT、INTEGER、SMALLINT	OCINumber	INT、SHORT、OCINumber
NUMBER、NUMERIC、REAL、FLOAT、DOUBLE PRECISION	OCINumber	INT、FLOAT、DOUBLE、OCINumber
DATE	OCIDate	STRING、VARCHAR、CHARZ、OCIDate

REF が参照するオブジェクトは、バインドされる REF と型互換性を持つ必要があります。

これらの表の場合、OBJECT GET では「記述」列に指定された形式が使用され、左端のデータベース型から「ホスト・データ型」列の形式を使用する内部データ型に変換されます。OBJECT SET では逆の変換が行われます。

明示的タイプ・チェックはサポートされません

プリコンパイラでは、コレクション要素データ型とホスト変数データ型間のバインド時の、明示的タイプ・チェックはサポートされません。タイプ・チェックは実行時に行われます。

COLLECTION 文

COLLECTION GET

用途

COLLECTION GET 文は、OBJECT GET 文に似ていますが、コレクションを対象としています。コレクションの要素を取得し、カレント・スライスを設定し、適切な場合には要素を C 型に変換します。

構文

```
EXEC SQL [AT [:]database] [FOR :num]
  COLLECTION GET :collect [[INDICATOR] :collect_ind]
  INTO :hv [[INDICATOR] :hv_ind] ;
```

変数

num (IN)

要求する要素の数。この句を省略すると、コレクションから取得する要素の数は、ホスト変数の配列サイズ（スカラーは1）により決定されます。

collect (IN)

ホスト変数 C コレクション記述子。

collect_ind (IN)

コレクションの NULL 状態を戻すオプションの標識変数。

hv (OUT)

コレクション要素値を受け取るホスト変数。

hv_ind (OUT)

スカラーまたは配列にスライスの各要素の状態が格納されている場合、hv の NULL 状態を戻すオプションの標識変数。

使用上の注意

最後の COLLECTION GET から実際に戻される要素の数は、sqlca.sqlerrd[2] に設定されます。（すべての GET 累計ではありません。）SQLCA のこのコンポーネントの説明は、9-21 ページの「[sqlerrd](#)」を参照してください。

スライスのエンドポイントのいずれかまたは両方がコレクションの境界を超過したときは、戻される要素数が要求した数よりも少ないことがあります。次の場合にこの状態が発生します。

- コレクション記述子が、正しい構文の ALLOCATE 文で初期化されなかった場合、NULL の場合、その他の理由で無効な場合。
- コレクションが NULL の場合。関連付けられる標識は -1 になります。
- コレクションが空（要素がない）の場合。
- コレクションに残っている要素数以上の要素が要求された場合。
- COLLECTION TRIM 文が実行された結果、カレント・スライスで終了索引が開始索引よりも前になっている場合。

C コレクション記述子が適切に初期化されなかった場合、エラーになります。前述のリスト以外の場合は、「ORA-01403: データが見つかりません。」というエラーが発生します。この場合、エラー発生前に正常に取得できた要素の合計数は、`sqlca.sqlerrd[2]` に格納されたままです。

最初の GET または RESET 後の最初の GET により、スライスは次のようになります。

- スライスの終了索引は、検出された最後の要素の索引になります。要求する要素数によって変わります。コレクションに要求を満たすのみの要素が残っていない場合、最後の索引はコレクションの最後の索引になります。

引き続き GET を実行すると、スライスの索引は次のようになります。

- 直前のスライスのエンドポイントの後に最初に検出された要素の索引が、開始索引になります。直前のスライスのエンドポイントの後に要素が残っていない場合、開始索引はコレクションの最後の要素の索引になります。
- 次のスライスの終了索引は、検出された最後の要素の索引になります。要求した要素数によって変わります。直前のスライスで指定された位置で、要求を満たすのみの要素がコレクションに残っていない場合、終了索引はコレクションの最後の索引になります。

COLLECTION SET

用途

COLLECTION SET 文は、OBJECT SET 文に似ています。コレクションの要素値の更新に使用します。カレント・スライスの要素は、C 固有型から Oracle データ型に変換されます。

構文

```
EXEC SQL [AT [:]database] [FOR :num]
  COLLECTION SET :collect [[INDICATOR] :collect_ind]
  TO :hv [[INDICATOR] :hv_ind];
```

変数

`num` (IN)

このオプションのスカラー値は、スライス内で更新される要素の最大数です。この句を省略すると、コレクションから更新される要素数は、ホスト変数の配列サイズ（スカラーは 1）により決定されます。

`collect` (OUT)

ホスト変数 C コレクション記述子。

`collect_ind` (OUT)

コレクションの NULL 状態を決定するオプションの標識変数。

hv (IN)

コレクション内で更新される値を含むホスト変数。

hv_ind (IN)

ホスト変数の NULL 状態を表す関連付けられた標識変数。

使用上の注意

次の制限が適用されます。

- COLLECTION GET は、COLLECTION SET の前に実行する必要があります。
- スライスの開始索引および終了索引は、常に変わりません。コレクションに格納されている要素数が、カレント・スライスに格納できる要素数以下の場合も変わりません。SET 文では、スライスのエンドポイントは変更されません。カレント・スライスの要素のみが変更されます。
- COLLECTION SET では、カレント・スライスの要素のみが更新されます。COLLECTION SET 文を使用して、新しい要素をコレクションに追加することはできません。
- カレント・スライスに含まれる要素数以上の要素を SET 文で更新すると、既存のスライスに格納されている要素のみが更新され、スライス終点外の残りの要素は更新されません。また、ホスト変数によって指定されたその他の値は使用されません。

オプションの FOR 句で指定したホスト変数または値 num のディメンションにより、コレクションで更新を要求できる要素の最大数が決まります。

変数 `sqlca.sqlerrd[2]` により、直前の SET 文で正常に更新された要素の数が戻されます。(累計ではありません。) 次のような場合、GET 文と同様に、設定要求数よりも少ないことがあります。

- C コレクション記述子が、構文の正しい ALLOCATE 文で正常に初期化されなかった場合、NULL の場合、またはその他の理由で無効の場合。
- コレクションが空の場合。
- コレクションのカレント・スライスの位置で、コレクションの残りの部分の要素数が、設定要求数よりも少なかった場合。
- カレント・スライスの終点を超えた場合。既存のスライスの要素数以上の要素を設定しようとしたときに、この状態になります。
- コレクションに対して TRIM が実行され、コレクションの終了索引の最大値がカレント・スライスの開始索引を下回った場合。

COLLECTION GET または SET の直後に COLLECTION SET を実行した場合は、既存のスライスの要素の値のみが更新されます。COLLECTION SET の直後に COLLECTION GET を実行すると、すでに説明したように次のスライスに移ります。

COLLECTION RESET

用途

コレクション・スライスのエンドポイントをコレクションの最初にリセットします。

構文

```
EXEC SQL [AT [:] database]  
    COLLECTION RESET :collect  
    [ [INDICATOR] :collect_ind ] ;
```

変数

collect (IN/OUT)

エンドポイントをリセットするコレクション。

collect_ind

コレクションの NULL 状態を決定するオプションの標識変数。

使用上の注意

指定したコレクションが NULL または無効の場合、エラーが発生します。

COLLECTION APPEND

用途

コレクションの最後に要素のセット（1 つ以上）を追加します。コレクションのサイズが増加します。

構文

```
EXEC SQL [AT [:] database] [FOR :num]  
    COLLECTION APPEND :src [[INDICATOR] :src_ind]  
    TO :collect [[INDICATOR] :collect_ind ] ;
```

変数

num (IN)

追加する要素数が格納されたスカラー。指定しない場合、配列サイズ *src* が追加する要素数になります。

src (IN)

コレクションに追加する要素のスカラーまたは配列。

src_ind (IN)

追加する要素の NULL 状態を決定するオプションの標識変数（スカラーまたは配列）。

collect (IN OUT)

要素を追加するコレクション。

collect_ind (IN)

コレクションの NULL 状態を決定するオプションの標識変数。

使用上の注意

要素は一度に1つずつ追加されます。（コレクションのサイズは1ずつ増加し、データがその要素にコピーされます。）

変数 `sqlca.sqlerrd[2]` により、最後の APPEND で正常に追加された要素数が戻されます。（累計ではありません。）コレクションの上限を超えて要素を追加したり、NULL コレクションを追加しようとすると、エラーが発生します。上限以内の要素のみが追加されます。

COLLECTION TRIM

用途

コレクションの最後から要素を削除します。

構文

```
EXEC SQL [AT [:] database]
    COLLECTION TRIM :num
    FROM :collect [[INDICATOR] :collect_ind] ;
```

変数

num (IN)

削除する要素数を示すホスト・スカラー変数。最大許容値は2 ギガバイトです。

collect (IN OUT)

切り捨てるコレクション。

collect_ind (IN)

コレクションの NULL 状態を決定するオプションの標識変数。

使用上の注意

次の制限が適用されます。

- FOR 句は使用できません。
- num の最大値は2 ギガバイトです (4 バイト符号付バイナリ変数の最大数)。
- num に標識を使用することはできません。

num がコレクションのサイズよりも大きい場合、エラーが戻されます。TRIM 文でカレント・スライスから要素を削除すると、警告が戻されます。

COLLECTION DESCRIBE

用途

コレクションについての情報が返されます。

構文

```
EXEC SQL [AT [:] database]
  COLLECTION DESCRIBE :collect [[INDICATOR] :collect_ind]
  GET attribute1 [{, attributeN}]
  INTO :hv1 [[INDICATOR] :hv_ind1] [{, hvN [[INDICATOR] :hv_indN]]] ;
```

attributeN は次のとおりです。

DATA_SIZE		TYPECODE		DATA_TYPE		NUM_ELEMENTS
		PRECISION		SCALE		TYPE_NAME
				TYPE_SCHEMA		SIZE
						TABLE_SIZE

変数

collect (IN)

ホスト変数 C コレクション記述子。

collect_ind (IN)

コレクションの NULL 状態を含むオプションの標識変数。

hv1 .. hvN (OUT)

情報が格納される出力ホスト変数。

hv_ind1 .. hv_indN (OUT)

出力ホスト変数の標識変数。

使用上の注意

次の制限が適用されます。

- コレクションを NULL にはできません。
- ホスト変数型は、戻される属性の型と互換性を持つ必要があります。
- 属性の標識変数は、テキストの切捨てが行われる TYPE_NAME および TYPE_SCHEMA 属性値でのみ必要です。
- FOR 句は使用できません。
- 変数 sqlca.sqlerrd[2] では、正常に取り出された属性の数を戻します。DESCRIBE 文でエラーが発生した場合、sqlca.sqlqerrd[2] には、エラー発生前に戻された属性の数が格納されます。エラー発生時の属性数より 1 つ少なくなっています。

次の表は、属性、説明、および取り出される属性の C 型を示しています。

表 18-3 COLLECTION DESCRIBE の属性

属性	説明	C 型	注意
DATA_SIZE	型属性の最大サイズ。戻される長さは、文字列のバイト長です。NUMBER の場合は 22 です。	符号なし SHORT	オブジェクトまたはオブジェクト REF 要素では無効です。
TYPECODE	OCI 型コード。	OCITypeCode	
DATA_TYPE	コレクション項目の内部数値型コード。	符号なし SHORT	
NUM_ELEMENTS	VARRAY の最大要素数。	符号なし INT	VARRAY 型に限り有効です。
PRECISION	数値型属性の精度。戻される値が 0 の場合、記述される項目は初期化されず、データ・ディクショナリは NULL になります。	符号なし CHAR	NUMBER 型の要素に限り有効です。
SCALE	数値型属性のスケール。戻される値が -127 の場合、記述される項目は初期化されず、データ・ディクショナリは NULL になります。	符号付き CHAR	NUMBER 型の要素に限り有効です。
TYPE_NAME	型の名前を含む文字列。オブジェクト型の場合、その名前が戻されます。REF の場合、REF によって参照されるデータ型の名前が戻されます。使用できる外部データ型は CHARZ、STRING および VARCHAR です。	char*	オブジェクトおよびオブジェクト REF 要素に限り有効です。

表 18-3 COLLECTION DESCRIBE の属性

属性	説明	C 型	注意
TYPE_SCHEMA	型を作成するスキーマ名。使用できる外部データ型は CHARZ、STRING および VARCHAR です。	char*	オブジェクトおよびオブジェクト REF 要素に限り有効です。
SIZE	コレクションに実際に格納されている要素数。NESTED TABLE の場合、SIZE により空の要素が格納されます。TRIM 文を実行すると、切り捨てられた要素数のみコレクションの SIZE が減少します。	符号付き INT	
TABLE_SIZE	NESTED TABLE の要素の数。間隔は含みません。	符号付き INT	NESTED TABLE に限り有効です。

表について

Pro*C/C++ の外部データ型では、属性 TYPE_NAME および TYPE_SCHEMA の CHARZ、STRING および VARCHAR のみがサポートされます。

SIZE および TABLE_SIZE を除き、DESCRIBE 属性は、すべてコレクションの要素型に依存し、コレクションの特定のインスタンスには依存しません。一方、SIZE および TABLE_SIZE 属性は、値がコレクションの特定のインスタンスに完全に依存しています。割り当てられたコレクション記述子が再利用されて同じコレクションの異なるインスタンスが参照される場合、SIZE または TABLE_SIZE の値はコレクションのインスタンスごとに変わります。NUM_ELEMENTS、はコレクション型（この例では VARRAY）の属性で、コレクション要素型ではありません。また、コレクションの特定のインスタンスには依存しません。

コレクションを使用する場合の規則

- ホスト変数コレクション記述子は、常に明示的に割り当てする必要があります。
- メタデータ（コレクションおよびその要素型についての、データベースの内部 Oracle データ）は、ALLOCATE の実行中に収集されます。ALLOCATE が実行された接続がクローズしているとき、または ALLOCATE の実行後に型が変更されたときは、メタデータは無効になります。
- 各 C コレクション記述子の使用を開始または終了するには、ALLOCATE または FREE 文を使用します。

コレクション・サンプル・コード

COLLECTION SQL 文を使用する SQL および Pro*C/C++ コードの例を示します。

型および表の作成

scott/tiger として接続し、SQL を使用して次の型を作成すると仮定します。

```
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
) ;
```

オブジェクト型トランスレータによって、ヘッダー・ファイルが生成されます。OTT の完全な説明は、18-1 ページの「[オブジェクト型トランスレータ](#)」を参照してください。OTT の入力として、次の入力ファイル（ファイル名 in.typ）が使用されます。

```
case=lower
type employee
type employees
type department
```

次のコマンドによりヘッダー・ファイルが生成されます。

```
ott intype=in.typ outtype=out.typ hfile=example.h user=scott/tiger code=c
```

このヘッダー・ファイル example.h, は、OTT により生成されます。

```
#ifndef EXAMPLE_ORACLE
# define EXAMPLE_ORACLE

#endif

#ifndef OCI_ORACLE
# include <oci.h>
#endif

typedef OCISRef employee_ref ;
typedef OCIArray employees ;
typedef OCISRef department_ref ;

struct employee
{
```

```

        OCIStr * 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

```

注意：oci.h には、OCIArray を定義した typedef を持つ orl.h が格納されています。typedef は、次の「typedef OCIColl OCIArray;」のようになります。OCIColl は、総称コレクションを表す不透明な構造体です。

次の 1 列が含まれる簡単な表を作成します。

```
CREATE TABLE division ( subdivision department ) ;
```

この表に複数の行を挿入します。

```

INSERT INTO division (subdivision) VALUES
(department('Accounting',
            employees(employee('John', 75000),
                       employee('Jane', 75000)))
);
INSERT INTO division (subdivision) VALUES
(department('Development',
            employees(employee('Peter', 80000),
                       employee('Paula', 80000)))
);
INSERT INTO division (subdivision) VALUES
(department('Research',
            employees(employee('Albert', 95000),
                       employee('Alison', 95000)))
);

```

これらの型定義および表の情報を、次の例で使います。

GET および SET の例

この例では、オブジェクトのコレクション属性から値を取り出し、簡単な修正を加え、コレクションに戻します。

まず、example.h をインクルードし、オブジェクト型の変数を宣言する必要があります。

```
#include <example.h>
department *dept_p ;
```

「開発」部門を部署表から選択します。

```
EXEC SQL ALLOCATE :dept_p ;
EXEC SQL SELECT subdivision INTO :dept_p
FROM division WHERE name = 'Development' ;
```

employee オブジェクト型のチーム VARRAY の変数および単一の employee オブジェクトを表す変数が必要です。開発部門のすべてのメンバーの給料を昇給します。そのための変数が必要です。

```
employees *emp_array ;
employee *emp_p ;
double salary ;
```

作成した VARRAY C コレクションおよび employee オブジェクト記述子に、ALLOCATE を実行する必要があります。ナビゲーション・インタフェースを使用して、オブジェクトから実際のコレクションを取り出します。

```
EXEC SQL ALLOCATE :emp_array ;
EXEC SQL ALLOCATE :emp_p ;
EXEC SQL OBJECT GET team FROM :dept_p INTO :emp_array ;
```

ループを使用し、VARRAY 要素に対して処理を繰り返します。WHENEVER 宣言文を使用してループの終了を制御します。

```
EXEC SQL WHENEVER NOT FOUND DO break ;
while (TRUE)
{
```

まず、コレクションから変更する要素を取り出します。実際の要素型は、employee オブジェクトです。

```
EXEC SQL COLLECTION GET :emp_array INTO :emp_p ;
```

実際のオブジェクト要素を取り出したので、既存のナビゲーション・インタフェースを使用して属性の値を変更します。この例では、全員の給料を 10% 増やします。

```
EXEC SQL OBJECT GET salary FROM :emp_p INTO :salary ;
salary += (salary * .10) ;
EXEC SQL OBJECT SET salary OF :emp_p TO :salary ;
```

変更が終わると、コレクションに現在含まれるオブジェクト要素の属性の値を更新できます。

```
EXEC SQL COLLECTION SET :emp_array TO :emp_p ;
}
```

すべてのコレクション要素に対して処理を繰り返した後、そのコレクションを含むオブジェクトが格納されている表の列を更新する必要があります。

```
EXEC SQL UPDATE division SET subdivision = :dept_p ;
```

次に、FREE を実行してすべてのリソースを解放し、COMMIT を実行してこの一連の操作を終了します。

```
EXEC SQL FREE :emp_array ;
EXEC SQL FREE :emp_p ;
EXEC SQL FREE :dept_p ;
EXEC SQL COMMIT WORK ;
```

簡単な例ですが、必要な処理はすべて含まれています。ナビゲーション OBJECT GET 文を応用して、コレクション属性をオブジェクトから取り出し、C コレクション記述子に格納する方法は、明確に説明されています。さらに、その C 記述子を使用して、実際のコレクションの要素を取出しおよび更新を行うための、新しい COLLECTION GET 文および SET 文の使用方法について説明しました。コレクション・オブジェクト要素型の属性値の変更には、ナビゲーション・インタフェースを使用しています。

DESCRIBE の例

この例では、DECLARE CURSOR 文の使用方を示します。任意のコレクションについての基本情報を検索します。

まず、例で使用されているヘッダー・ファイル、オブジェクト・ポインタおよび SQL コレクション記述子が必要です。

```
#include <example.h>
department *dept_p ;
```

前と同様に、オブジェクト・ポインタの ALLOCATE を実行し、表からオブジェクトを取り出します。

```
EXEC SQL ALLOCATE :dept_p ;
EXEC SQL SELECT subdivision INTO :dept_p
FROM division WHERE name = 'Research' ;
```

検索するコレクション属性情報を格納する Pro*C/C++ 変数を宣言します。

```
int size ;
char type_name[9] ;
employees *emp_array ;
```

コレクション記述子を割り当て、ナビゲーション・インタフェースを使用して、オブジェクトからコレクション属性を取り出します。

```
EXEC SQL ALLOCATE :emp_array ;
EXEC SQL OBJECT GET team FROM :dept_p INTO :emp_array ;
```

最後に、新しい COLLECTION DESCRIBE 文を使用して目的のコレクション属性情報を抽出します。

```
EXEC SQL COLLECTION DESCRIBE :emp_array
      GET SIZE, TYPE_NAME INTO :size, :type_name ;
```

注意：この例に示すように、目的のコレクション属性名と同じホスト変数名を使用することができます。

型 employees は、オブジェクト employee の VARRAY のため、型名を抽出できます。

DESCRIBE が正常に実行されると、SIZE の値は 2（このコレクション・インスタンス Research の場合、2 つの要素、Albert および Alison が存在します）になります。type_name 変数は「EMPLOYEE\0」（デフォルトでは CHARZ）になります。

SQL 記述子およびオブジェクト・ポインタを使用した処理が終了した後に、FREE を実行してリソースを解放します。

```
EXEC SQL FREE :emp_array ;
EXEC SQL FREE :dept_p ;
```

この例では、C コレクション記述子の参照先の基礎となるコレクションについて、記述子から情報を抽出するために使用する DESCRIBE のメカニズムについて説明しました。

RESET の例

開発の従業員の給料を昇給するかわりに、GET および SET の例のように、部署全体の給料を昇給します。

前の例と同様に、オブジェクト型トランスレータで生成されたサンプル・ヘッダー・ファイルなどを処理します。ただし、今回は、カーソルを使用して部署の部門ごとに、一度に 1 部門ずつ繰り返し実行します。

```
#include <example.h>
EXEC SQL DECLARE c CURSOR FOR SELECT subdivision FROM division ;
```

データを操作するローカル変数が必要になります。

```

department *dept_p ;
employees *emp_array ;
employee *emp_p ;
double salary ;
int size ;

```

オブジェクト変数およびコレクション変数を使用する前に、次の ALLOCATE 文を使用して初期化する必要があります。

```

EXEC SQL ALLOCATE :emp_array ;
EXEC SQL ALLOCATE :emp_p ;

```

カーソルを使用して、部署のすべての部門に対して繰り返し処理を行うことができますようになりました。

```

EXEC SQL OPEN c ;
EXEC SQL WHENEVER NOT FOUND DO break ;
while (TRUE)
{
    EXEC SQL FETCH c INTO :dept_p ;

```

ここで、部門オブジェクトを使用します。ナビゲーション・インタフェースを使用して、部門から VARRAY 属性 "team" を抽出する必要があります。

```

EXEC SQL OBJECT GET team FROM :dept_p INTO :emp_array ;

```

コレクションへの参照を開始する前に、RESET 文を使用して、スライスのエンドポイントがカレント・コレクション・インスタンスの始点（前のインスタンスの最後ではありません）に設定されていることを確認してください。

```

EXEC SQL COLLECTION RESET :emp_array ;

```

VARRAY のすべての要素を繰り返し処理し、前と同様に給料を更新します。このループの場合も、既存の WHENEVER 宣言文は引き続き有効です。

```

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 ;

```

処理が完了すると、コレクション属性を更新します。

```

EXEC SQL COLLECTION SET :emp_array TO :emp_p ;
}

```

前の例と同様に、変更が完了したコレクションを含むオブジェクトが格納された表の列を更新する必要があります。

```
EXEC SQL UPDATE division SET subdivision = :dept_p ;
}
```

ループが終了すると、処理は終了です。FREE を実行してすべてのリソースを解放し、COMMIT で作業内容を送信します。

```
EXEC SQL CLOSE c ;
EXEC SQL FREE :emp_p ;
EXEC SQL FREE :emp_array ;
EXEC SQL FREE :dept_p ;
EXEC SQL COMMIT WORK ;
```

この例では、同じコレクション型の異なるインスタンスに対して、ALLOCATE で割り当てられたコレクション記述子の再利用方法について説明しています。COLLECTION RESET 文を実行すると、スライスのエンドポイントは、直前のコレクション・インスタンスの参照中に移動した後の位置のままではなく、カレント・コレクション・インスタンスの始点に戻るよう設定できます。

COLLECTION RESET 文をこのように使用すると、アプリケーション開発者は同じコレクション型の新しいインスタンスを作成するたびに、コレクション記述子に対して明示的に FREE および ALLOCATE を実行する必要がなくなります。

サンプル・プログラム :coldemo1.pc

次のプログラム coldemo1.pc は、demo ディレクトリにあります。

この例では、Pro*C クライアントからコレクション型データベース列を操作する 3 種類の方法について説明しています。この例では、NESTED TABLE を使用していますが、VARRAY にも適用できます。

この例では、SQL*Plus ファイル、coldemo1.sql を使用して、挿入データおよび calidata.sql に格納されているデータを使用する表をセットアップします。

```
REM *****
REM ** This is a SQL*Plus script to demonstrate collection manipulation
REM ** in Pro*C/C++.
REM ** Run this script before executing OTT for the coldemo1.pc program
REM *****

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 January 1, 1996.

@calidata.sql;

REM Commit to save
COMMIT;

```

表の設定方法およびこのプログラムでデモンストレーションする機能の説明については、次のプログラムの最初のコメントを参照してください。

```

/* ***** */
/* 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=coldemol.typ outt=out.typ hfile=coldemol.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).

Scenario:

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 below.

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>
#include <sqlca.h> /* SQL Communications Area */
#include <coldemo1.h> /* OTT-generated header with C typedefs for the */
/* database types city_t and citytbl_t */
#ifndef EXIT_SUCCESS
# define EXIT_SUCCESS 0
#endif
#ifndef EXIT_FAILURE
# define EXIT_FAILURE 1
#endif

#define CITY_NAME_LEN 30
#define COUNTY_NAME_LEN 30
#define MAX_COUNTIES 60

/* 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 PrintSQLException();

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);
                break;
            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] != '\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 be 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()
{
    citytbl_t *cityTable[MAX_COUNTIES];           /* 1 */
    OCIInd    cityInd[MAX_COUNTIES];              /* 4 */
    char      county[MAX_COUNTIES][COUNTY_NAME_LEN + 1];
    int       i, numCounties;
    city_t    *city;

    EXEC SQL ALLOCATE :cityTable;                  /* 2 */
    EXEC SQL ALLOCATE :city;

    EXEC SQL SELECT name, cities
        INTO :county, :cityTable:cityInd FROM county_tbl; /* 3, 4 */

    numCounties = sqlca.sqlerrd[2];

    for (i = 0; i < numCounties; i++)
    {
```

```

        if (cityInd[i] == OCI_IND_NULL)                                /* 4 */
        {
            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
 *   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;

    EXEC SQL ALLOCATE :cityCursor;                         /* 2 */
    EXEC SQL ALLOCATE :city;

    EXEC ORACLE OPTION(select_error=no);
    EXEC SQL SELECT
        CURSOR(SELECT VALUE(c) FROM TABLE(county_tbl.cities) c)
        INTO :cityCursor
        FROM county_tbl
        WHERE county_tbl.name = :countyName;                /* 3 */
    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;

EXEC SQL FREE :cityCursor;                                /* 2 */
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
            WHERE name = :countyName) AS c;                /* 1, 2 */

    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
*   PrintSQLException
* DESCRIPTION
*   Prints an error message using info in sqlca and calls exit.
* COLLECTION FEATURES
*   none
*/
void PrintSQLException()
{
    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);
}
```

オブジェクト型トランスレータ

この章では、Pro*C/C++ アプリケーションで使用するためにデータベース・オブジェクト型、LOB 型およびコレクション型を C 構造体にマップする、オブジェクト型トランスレータ (OTT) について説明します。

この章にある項は次のとおりです。

- [OTT 概要](#)
- [オブジェクト型トランスレータ](#)
- [OCI アプリケーションでの OTT の使用](#)
- [Pro*C/C++ アプリケーションでの OTT の使用](#)
- [OTT 参照](#)

OTT 概要

OTT（オブジェクト型トランスレータ）は、Oracle8i サーバーのユーザー定義型を利用するアプリケーションの開発に役立ちます。

SQL CREATE TYPE 文を使用して、オブジェクト型を作成できます。これらの型の定義をデータベースに格納しておき、データベースの表を作成するときに使用できます。これらの表への移入後は、OCI、Pro*C/C++ または Java のプログラマは表に格納されているオブジェクトにアクセスできます。

オブジェクト・データにアクセスするアプリケーションには、データをホスト言語の書式で表す機能が必要です。これは、オブジェクト型を C 構造体として表現することによって実現できます。プログラマがデータベース・オブジェクト型を表す構造体宣言を、手入力コーディングすることは可能です。しかし、多くの型がある場合、この作業は時間がかかり、エラーを生む原因になりがちです。OTT を使用すると、必要な構造体の宣言が自動的に生成されるので、作業を軽減できます。Pro*C/C++ の場合、アプリケーション側では、OTT によって生成されたヘッダー・ファイルを組み込むのみですみます。OCI の場合は、アプリケーション側では OTT によって生成される初期化関数もコールする必要があります。

OTT は、格納されているデータ型を表す構造体を作成するのみでなく、オブジェクト型またはそのフィールドが NULL かどうかを示すパラレル標識構造体も生成します。

参照項目：オブジェクト型の詳細は、[第 17 章「オブジェクト」](#)を参照してください。[第 18 章「コレクション」](#)および[第 16 章「ラージ・オブジェクト \(LOB\)」](#)も参照してください。

オブジェクト型トランスレータ

オブジェクト型トランスレータ（OTT）は、オブジェクト型と名前付きコレクション型のデータベース定義を、OCI または Pro*C/C++ アプリケーションに組み込める C 構造体宣言に変換します。

OCI と Pro*C/C++ のプログラマは、OTT を明示的に起動してデータベース型を C の表現に変換する必要があります。また、OCI のプログラマは、タイプ・バージョン表というデータ構造を、プログラムに必要なユーザー定義型の情報で初期化する必要があります。この初期化を実行するためのコードは、OTT によって生成されます。Pro*C/C++ では、タイプ・バージョン情報は、Pro*C/C++ にパラメータとして渡される Outtype ファイルに記録されます。

ほとんどのオペレーティング・システムでは、OTT はコマンドラインから起動します。OTT は、*intype* ファイルを入力として受け取り、*outtype* ファイル、1 つ以上の C のヘッダー・ファイルおよびオプションのインプリメンテーション・ファイル（OCI プログラム用）を生成します。次の例は、OTT を起動するコマンドを示しています。

```
ott userid=scott/tiger intype=demo.in.typ outtype=demo.out.typ code=c hfile=demo.h
```

このコマンドを実行すると、OTT はユーザー名 *scott* とパスワード *tiger* を使用してデータベースに接続され、*intype* ファイル *demo.in.typ* 内の指示に従ってデータベース型を C の構

造体に変換します。その結果、構造体は `code` パラメータで指定したホスト言語 (C) 用に、ヘッダー・ファイル *demo.h* に出力されます。outtype ファイル *demoout.typ* には、この変換に関する情報が格納されます。

各パラメータについては、この章のこの後の項で詳しく説明します。

demoin.typ ファイルの例

```
CASE=LOWER
TYPE employee
```

demoout.typ ファイルの例

```
CASE = LOWER
TYPE EMPLOYEE AS employee
    VERSION = "$8.0"
    HFILE = demo.h
```

この例では、demoin.typ ファイルに、TYPE が前に付く変換対象の型 (たとえば、TYPE employee) があります。Outtype ファイルの構造は Intype ファイルに似ていますが、OTT が取得した情報が追加されています。

OTT による変換が完了すると、ヘッダー・ファイルには Intype ファイルで指定した各型を表す C の構造体と、各型に対応する NULL 標識構造体が含まれています。たとえば、Intype ファイルにリストされた employee 型が次のように定義されたとします。

```
CREATE TYPE employee AS OBJECT
(
    name          VARCHAR2(30),
    empno         NUMBER,
    deptno        NUMBER,
    hiredate      DATE,
    salary        NUMBER
);
```

この場合、OTT によって生成されるヘッダー・ファイル (demo.h) には、他の項目とともに次の宣言が含まれます。

```
struct employee
{
    OCIStrng * name;
    OCINumber empno;
    OCINumber deptno;
    OCIDate  hiredate;
    OCINumber salary;
};
typedef struct emp_type emp_type;

struct employee_ind
```

```
{
    OCInd _atomic;
    OCInd name;
    OCInd empno;
    OCInd deptno;
    OCInd hiredate;
    OCInd salary;
};
typedef struct employee_ind employee_ind;
```

注意 : Intype ファイルにおけるパラメータにより、生成された構造体の名前設定の方法が制御されます。この例では、構造体名の `employee` はデータベース型名の `employee` と合致します。Intype ファイルの行で `CASE=lower` と指定されているため、構造体名には小文字を使用します。

構造体宣言で使用するデータ型 (**OCString**、**OCInd** など) は Oracle8 の新しい特殊データ型です。これらの型の詳細は、19-9 ページの「[OTT データ型のマップ](#)」を参照してください。

これ以降の項では、OTT の使用について次の事柄を説明します。

- [データベースにおける型の作成](#)
- [OTT の起動](#)
- [OTT コマンドライン](#)
- [Intype ファイル](#)
- [OTT データ型のマップ](#)
- [NULL 標識構造体](#)
- [Outtype ファイル](#)

これ以降の各項では、OCI および Pro*C/C++ と OTT の併用について説明してから、参考のためにコマンドライン構文、パラメータ、Intype ファイルの構造、ネストされた `#include` ファイルの生成、スキーマ名の使用方法、デフォルト名のマッピングおよび制限事項について説明します。

データベースにおける型の作成

OTT を使用するには、最初にオブジェクト型または名前付きコレクション型を作成して、データベースに格納します。そのためには、SQL `CREATE TYPE` 文を使用します。

関連項目 : オブジェクト型およびコレクション型の作成に関する詳細は、[第 17 章「オブジェクト」](#)を参照してください。

OTT の起動

次のステップは、OTT を起動することです。

OTT のパラメータは、コマンドラインまたは構成ファイル内で指定できます。一部のパラメータは Intype ファイルでも指定できます。

パラメータを数箇所指定すると、優先順位が最も高いのはコマンドラインで指定した値で、次に Intype ファイル内の値、ユーザー定義の構成ファイルの値、デフォルトの構成ファイル内の値の順になります。

グローバル・オプション、つまりコマンドラインのオプションあるいは INTYPE ファイルの TYPE 文の前のオプションについては、コマンドラインの値が INTYPE ファイルの値をオーバーライドします。(INTYPE ファイルでグローバルとして指定できるオプションには、CASE、CODE、INITFILE、OUTDIR および INITFUNC があります。HFILE は含まれません。) TYPE を指定する INTYPE ファイルのオプションは特定の型に限り適用されます。その型に適用される、コマンドラインで指定したそれ以外のオプションはオーバーライドされません。TYPE person HFILE=p.h と入力した場合、オプションは person にのみ適用され、コマンドラインの HFILE はオーバーライドされます。文はコマンドライン・パラメータとは見なされません。

コマンドライン

コマンドラインに設定されたパラメータ（オプションとも呼ばれます）は、他で設定されたパラメータを上書きします。詳細は、19-6 ページの「[OTT コマンドライン](#)」を参照してください。

構成ファイル

構成ファイルは、OTT パラメータが入っているテキスト・ファイルです。ファイル内の空白以外の各行には、1 つのパラメータと、それに関連付けられた 1 つ以上の値が入っています。1 行に 2 つ以上のパラメータを指定した場合は、最初のパラメータのみが使用されます。構成ファイルの空白以外の行では、空白は使用できません。

構成ファイル名は、コマンドラインで指定できます。さらに、デフォルトの構成ファイルは常に読み込まれます。このデフォルトの構成ファイルは、常に存在する必要がありますが、空でもかまいません。デフォルトの構成ファイルの名前は `ottcfg.cfg` で、構成ファイルの位置はシステム固有です。詳細は、プラットフォーム固有のマニュアルを参照してください。

INTYPE ファイル

Intype ファイルは、OTT 用に変換される型のリストを示します。

パラメータの CASE、HFILE、INITFUNC および INITFILE は INTYPE ファイルに入れることができます。詳細は、19-8 ページの「[Intype ファイル](#)」を参照してください。

OTT コマンドライン

ほとんどのプラットフォームでは、OTT はコマンドラインから起動します。入力ファイルおよび出力ファイル、データベース接続情報などを指定できます。プラットフォームで OTT を起動する方法は、使用中のプラットフォームのマニュアルで確認してください。

次の例（例 1）は、コマンドラインから OTT を起動する方法を示しています。

```
ott userid=scott/tiger intype=demo.in.typ outtype=demo.out.typ code=c hfile=demo.h
```

注意：等号（=）の前後には空白は入力できません。

次の各項では、この例で使用しているコマンドラインの要素を説明します。

OTT コマンドラインの様々なオプションの詳細は、19-23 ページの「[OTT 参照](#)」を参照してください。

OTT

OTT を起動します。必ずコマンドラインの先頭に置きます。

Userid

OTT で使用されるデータベース接続情報を指定します。例 1 では、OTT はユーザー名 *scott* およびパスワード *tiger* を使用して接続しようとします。

Intype

使用する *intype* ファイルの名前を指定します。例 1 では、*Intype* ファイルの名前を *demo.in.typ* と指定しています。

Outtype

outtype ファイルの名前を指定します。OTT は、C のヘッダー・ファイルを生成するときに、変換対象の型の情報を *Outtype* ファイルにも書き込みます。このファイルには、変換対象の型ごとに、バージョン文字列と、その C での表現が書き込まれたヘッダー・ファイルが記録されます。

例 1 では、*Outtype* ファイルの名前は *demo.out.typ* です。

注意：*Outtype* のキーワードで指定されたファイルが既存する場合は、OTT の実行時に上書きされますが、次の例外があります。OTT で生成されたファイルの内容がそのファイルの内容と同一の場合、OTT はそのファイルには実際に書き込みません。これにより、ファイルの変更時間を節約でき、UNIX の *Make* および他のプラットフォームでの類似機能で、不必要な再コンパイルが実行されません。

コード

変換の目標言語を指定します。次のオプションがあります。

- C (ANSI_C と等価)
- ANSI_C (ANSI C 対応)
- KR_C (Kernighan & Ritchie C 対応)

現在、デフォルト値はないので、このパラメータは必須です。

構造体宣言は C の両言語で同一です。INITFILE ファイルで定義された初期化関数の定義形式は、KR_C が使用されるかどうかによって決まります。INITFILE オプションを使用しない場合、3 つのオプションはすべて等価です。

Hfile

生成された構造体を書き込む C ヘッダー・ファイルの名前を指定します。例 1 では、生成された構造体はファイル `demo.h` に格納されます。

注意: `hfile` のキーワードで指定されたファイルが既存する場合は、OTT の実行時に上書きされますが、次の例外があります。OTT で生成されたファイルの内容がそのファイルの前の内容と同一の場合、OTT はそのファイルには実際に書き込みません。これにより、ファイルの変更時間を節約でき、UNIX の *Make* および他のプラットフォームでの類似機能で、不必要な再コンパイルが実行されません。

Initfile

型初期化関数を書き込む C ソース・ファイルの使用を指定します。

初期化関数は、OCI プログラム内でのみ必要です。Pro*C/C++ プログラムでは、Pro*C/C++ ランタイム・ライブラリによって型が自動的に初期化されます。

注意: `initfile` のキーワードで指定されたファイルが既存する場合は、OTT の実行時に上書きされますが、次の例外があります。OTT で生成されたファイルの内容がそのファイルの前の内容と同一の場合、OTT はそのファイルには実際に書き込みません。これにより、ファイルの変更時間を節約でき、UNIX の *Make* および他のプラットフォームでの類似機能で、不必要な再コンパイルが実行されません。

Initfunc

`initfile` に定義する初期化関数の名前を指定します。

このパラメータを使用せず初期化関数を生成すると、初期化関数の名前は、`initfile` の基本名と同一になります。

この関数は、OCI プログラム内でのみ必要です。

Intype ファイル

OTT の実行時に、Intype ファイルはどのデータベース型の変換が必要であることを OTT に指示します。また、このファイルを使用して、生成される構造体の命名方法も指定できます。Intype ファイルは、新しく作成しても、前に OTT を起動したときの Outtype ファイルを使用してもかまいません。INTYPE パラメータを使用しない場合、OTT の接続先となるスキーマ内のすべての型が変換されます。

簡単なユーザー作成 intype ファイルの例を次に示します。

```
CASE=LOWER
TYPE employee
    TRANSLATE SALARY$ AS salary
        DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

第 1 行では、CASE キーワードによって、生成された C 識別子を小文字にすることを指示しています。ただし、この CASE オプションは、intype ファイルに明示的に記述していない識別子にのみ適用されます。そのため、*employee* および *ADDRESS* は、C 構造体では常にそれぞれ *employee* および *ADDRESS* となります。これらの構造体のメンバーは、小文字で名前が付けられます。

CASE オプションの詳細は、19-27 ページの「[CASE](#)」を参照してください。

TYPE キーワードで始まる行では、データベース内のどの型を変換するかを指定します。この場合、*EMPLOYEE*、*ADDRESS*、*ITEM*、*PERSON* および *PURCHASE_ORDER* 型です。

TRANSLATE...AS キーワードでは、オブジェクト型を C 構造体に変換するときに、オブジェクト属性の名前を変更することを指定しています。この場合は、*employee* 型の *SALARY\$* 属性が *salary* に変換されます。

最終行の AS キーワードでは、オブジェクト型を構造体に変換するとき、名前を変更することを指定しています。この例では、*purchase_order* というデータベース型を *p_o* という構造体に変換します。

AS を使用して型や属性名を変換しない場合は、その型や属性のデータベース名が C の識別子名として使用されます。ただし、CASE オプションを指定した場合、有効な C の識別子文字にマップできない文字は、アンダースコアに置き換えられます。型または属性名を変換する理由は、次のとおりです。

- 名前に、アルファベット、数字およびアンダースコア以外の文字が含まれる場合。
- 名前が C キーワードと競合する場合。
- 型名が、同一の有効範囲内で別の識別子と競合する場合。この問題が発生するのは、異なるスキーマから同じ名前を持つ 2 つの型がプログラムで使用される場合などです。
- プログラムが別の名前に変更する場合。

OTT では、Intype ファイル内で指定されていない型の変換を必要とする場合があります。これは、OTT は変換を実行する前に、Intype ファイル内の型相互の依存性を分析し、必要であれば他の型も変換するからです。たとえば、ADDRESS 型が Intype ファイル内で指定されていなくても、Person 型が ADDRESS 型の属性を持っている場合、ADDRESS は Person 型の定義に必要なので変換されます。

通常の大 / 小文字の区別がない SQL 識別子は、INTYPE ファイルで、大 / 小文字のどのような組合せのつづりでもかまいません。そして、引用符は付けません。

TYPE "Person" などの引用符を使用して、大 / 小文字を区別して作成された CREATE TYPE "Person" などの SQL 識別子が参照されます。宣言されるとき引用符付きの SQL 識別子は、大 / 小文字が区別されます。また、引用符は、TYPE "CASE" のような OTT 予約語である SQL 識別子を参照するためにも使用できます。その SQL 識別子が CREATE TYPE Case のように大 / 小文字を区別しないで作成されている場合、名前に引用符を付けるときは、その名前は大文字である必要があります。OTT の予約語を使用して SQL 識別子名を参照する場合には、引用符を付けなければ、Intype ファイル内で構文エラーがレポートされます。

関連項目：Intype ファイル構造の仕様の詳細と使用可能なオプションは、19-29 ページの「[Intype ファイルの構造](#)」を参照してください。

OTT データ型のマップ

OTT によってデータベース型から C の構造体が生成されると、その構造体にはオブジェクト型の各属性に対応する要素が 1 つ格納されます。属性のデータ型は、Oracle8i のオブジェクト型において使用される型にマップされます。オブジェクト型やコレクションなどのユーザー定義型の作成をサポートするために、Oracle8i のデータ型には、事前定義済の基本的な型の集合が組み込まれています。

事前定義済のデータ型には、数値型や文字型など、どのプログラマにも馴染みの標準型が含まれます。また、Oracle8 で新しく導入されたデータ型（BLOB や CLOB など）もあります。

Oracle8i には、オブジェクト型の属性を C の構造体で表すための事前定義済の型も用意されています。たとえば、次のようなオブジェクト型定義と、OTT で生成した対応する構造体宣言があるとします。

```
CREATE TYPE employee AS OBJECT
(   name          VARCHAR2(30),
    empno          NUMBER,
    deptno         NUMBER,
    hiredate       DATE,
    salary$        NUMBER);
```

CASE=LOWER で、型または属性名の明示的なマッピングがないと仮定すると、OTT 出力は次のようになります。

```
struct employee
{   OCIStrng * name;
    OCINumber empno;
```

```

    OCINumber department;
    OCIDate   hiredate;
    OCINumber salary_;
};
typedef struct emp_type emp_type;
struct employee_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd empno;
    OCIInd department;
    OCIInd hiredate;
    OCIInd salary_;
}
typedef struct employee_ind employee_ind;
```

標識構造体 (struct employee_ind) は、19-14 ページの「[NULL 標識構造体](#)」で説明しています。

構造体宣言のデータ型 *OCISString*、*OCINumber*、*OCIDate*、*OCIInd* は Oracle8 の新しいオブジェクト型の C マッピングです。オブジェクト型属性のデータ型をマップするためにここで使用されます。たとえば、*empno* 属性の数値データ型は、新しい *OCINumber* データ型にマップします。また、これらの新しいデータ型は、バインド変数および定義変数の型としても使用します。

関連項目 : OCI アプリケーションのオブジェクト・データ型を含むデータ型の使用方法の詳細は、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』を参照してください。

オブジェクト・データ型の C へのマッピング

この項では、オブジェクトの属性型を OTT によって生成された C の型にマップする方法について説明します。19-12 ページの「[OTT 型マッピングの例](#)」には、OTT によって生成されるオブジェクト・データ型の属性として使用できる型からのマッピングが表 19-1 にリストされています。

表 19-1 オブジェクト型属性のオブジェクト・データ型マッピング

オブジェクト属性の型	C マッピング
VARCHAR2(N)	OCISString *
VARCHAR(N)	OCISString *
CHAR(N)、CHARACTER(N)	OCISString *
NUMBER、NUMBER(N)、NUMBER(N,N)	OCINumber
NUMERIC、NUMERIC(N)、NUMERIC(N,N)	OCINumber

表 19-1 オブジェクト型属性のオブジェクト・データ型マッピング

オブジェクト属性の型	C マッピング
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 *
ネスト・オブジェクト型	ネスト・オブジェクト型の C 名
REF	typedef を使用して宣言 ; OCIRef * に相当
RAW(N)	OCIRaw *

表 19-2 では、名前付きのコレクション型の、OTT によって生成されるオブジェクト・データ型へのマッピングを示しています。

表 19-2 コレクション型のオブジェクト型マッピング

名前付きコレクション型	C マッピング
VARRAY (可変長配列)	typedef を使用して宣言 ; OCIArrary * に相当
NESTED TABLE	typedef を使用して宣言 ; OCITable * に相当

注意 : REF、VARRAY および NESTED TABLE 型では、OTT により typedef が生成されます。この typedef で宣言された型は、構造体の宣言でデータ・メンバーの型として使用されます。例は、19-12 ページの「[OTT 型マッピングの例](#)」を参照してください。

オブジェクト型に REF またはコレクション型の属性が含まれる場合、最初に REF またはコレクション型の typedef が生成されます。次に、オブジェクト型に対応する構造体宣言が生成されます。構造体には、REF またはコレクション型へのポインタを型に持つ要素が含まれます。

オブジェクト型が他のオブジェクト型の属性を持つ場合、OTT ではネストされた型が最初に生成されます。次に、オブジェクト型属性が、ネストしたオブジェクト型である型のネストした構造体にマッピングされます。

OTT によってオブジェクトではないデータベース属性型がマップされる C データ型は、構造体です。ただし、OCIDate の場合は不明になります。

OTT 型マッピングの例

次の例は、OTT によって作成される各種の型のマッピングを示しています。

この例では、次のデータベース型を使用します。

```
CREATE TYPE my_varray AS VARRAY(5) of integer;
```

```
CREATE TYPE object_type AS OBJECT
(object_name    VARCHAR2(20));
```

```
CREATE TYPE my_table AS TABLE OF object_type;
```

```
CREATE TYPE many_types AS OBJECT
( the_varchar    VARCHAR2(30),
  the_char       CHAR(3),
  the_blob       BLOB,
  the_clob       CLOB,
  the_object     object_type,
  another_ref    REF other_type,
  the_ref        REF many_types,
  the_varray     my_varray,
  the_table      my_table,
  the_date       DATE,
  the_num        NUMBER,
  the_raw        RAW(255));
```

また、Intype ファイルの内容は次のとおりです。

```
CASE = LOWER
TYPE many_types
```

OTT では、次の C の構造体が生成されます。

注意：構造体の説明の補足コメントを次に示します。これらのコメントは実際の OTT 出力の一部ではありません。

```
#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE

#ifdef OCI_ORACLE
#include <oci.h>
#endif

typedef OCISRef many_types_ref;
typedef OCISRef object_type_ref;
```

```

typedef OCIArray my_varray;          /* part of many_types */
typedef OCITable my_table;          /* part of many_types */
typedef OCIRef other_type_ref;
struct object_type                  /* part of many_types */
{
    OCIStr * object_name;
};
typedef struct object_type object_type;

struct object_type_ind              /*indicator struct for*/
{                                   /*object_types*/
    OCIInd _atomic;
    OCIInd object_name;
};
typedef struct object_type_ind object_type_ind;

struct many_types
{
    OCIStr *      the_varchar;
    OCIStr *      the_char;
    OCIBlobLocator * the_blob;
    OCIClobLocator * the_clob;
    struct object_type the_object;
    other_type_ref * another_ref;
    many_types_ref * the_ref;
    my_varray *    the_varray;
    my_table *     the_table;
    OCIDate        the_date;
    OCINumber      the_num;
    OCIRaw *       the_raw;
};
typedef struct many_types many_types;

struct many_types_ind              /*indicator struct for*/
{                                   /*many_types*/
    OCIInd _atomic;
    OCIInd the_varchar;
    OCIInd the_char;
    OCIInd the_blob;
    OCIInd the_clob;
    struct object_type_ind the_object; /*nested*/
    OCIInd another_ref;
    OCIInd the_ref;
    OCIInd the_varray;
    OCIInd the_table;
    OCIInd the_date;
    OCIInd the_num;
};

```

```
OCIInd the_raw;  
};  
typedef struct many_types_ind many_types_ind;  
  
#endif
```

Intype ファイル内では変換対象の項目は1つしか指定されていませんが、2つのオブジェクト型と2つの名前付きコレクション型が変換されていることに注目してください。19-6 ページの「[OTT コマンドライン](#)」で説明するように、リストされている型の変換を完了するために、OTT により変換対象の型の属性として使用される型がすべて変換されます。

ただし、その型がオブジェクト型の属性内でポインタまたは REF でしかアクセスされない場合は例外です。たとえば、*many_types* 型には属性として *another_ref REF other_type* がありますが、`struct other_type` の宣言は生成されていません。

また、この例では、VARRAY、NESTED TABLE および REF の各型を宣言するための typedefs の使用方法を示しています。

typedefs は始めの部分にあります。

```
typedef OCISRef many_types_ref;  
typedef OCISRef object_type_ref;  
typedef OCIArray my_varray;  
typedef OCISTable my_table;  
typedef OCISRef other_type_ref;
```

構造体 *many_types* では、次のように VARRAY、NESTED TABLE および REF 属性が宣言されています。

```
struct many_types  
{  
    ...  
    other_type_ref *   another_ref;  
    many_types_ref *   the_ref;  
    my_varray *        the_varray;  
    my_table *         the_table;  
    ...  
}
```

NULL 標識構造体

OTT によってデータベース・オブジェクト型を表す C 構造体が生成されるたびに、それに対応する NULL 標識構造体も生成されます。あるオブジェクト型を選択して C 構造体に変換すると、NULL 標識情報をパラレル構造体への変換用を選択できます。

たとえば、前の項の例では次の NULL 標識構造体が生成されました。

```
struct many_types_ind
{
  OCInd _atomic;
  OCInd the_varchar;
  OCInd the_char;
  OCInd the_blob;
  OCInd the_clob;
  struct object_type_ind the_object;
  OCInd another_ref;
  OCInd the_ref;
  OCInd the_varray;
  OCInd the_table;
  OCInd the_date;
  OCInd the_num;
  OCInd the_raw;
};
typedef struct many_types_ind many_types_ind;
```

NULL 構造体のレイアウトは重要です。構造体の第 1 要素 (`_atomic`) は、アトミック NULL 標識です。この値は、オブジェクト型全体の NULL 状態を示します。このアトミック NULL 標識の後に、OTT で生成した、オブジェクト型を表現する構造体の各要素に対応する標識要素が続きます。

オブジェクト型の定義に別のオブジェクト型が含まれている場合（前述の例では *object_type* 属性）、その属性の標識エントリは、ネストされたオブジェクト型に対応する NULL 標識構造体 (`object_type_ind`) になります。

VARRAY と NESTED TABLE には、要素に関する NULL 情報が含まれます。NULL 標識構造体のその他の要素のデータ型は、すべて *OCInd* です。

関連項目：アトミック NULL に関する詳細は、『Oracle8i コール・インタフェース・プログラマーズ・ガイド.』の第 1 章のオブジェクト型の説明を参照してください。

Outtype ファイル

Outtype ファイルは、OTT コマンドラインで名前が付けられます。OTT は、C のヘッダー・ファイルを生成するときに、変換結果を Outtype ファイルにも書き込みます。このファイルには、変換対象の型ごとに、バージョン文字列と、その C での表現が書き込まれたヘッダー・ファイルが記録されます。

OTT を 1 度実行して生成した outtype ファイルは、それ以降に OTT を起動する際の intype ファイルとして使用できます。

たとえば、この章の前半の例で使用した単純な intype ファイルを考えてみます。

```
CASE=LOWER
TYPE employee
```

```
TRANSLATE SALARY$ AS salary
DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE person
TYPE PURCHASE_ORDER AS p_o
```

ここで、ユーザーは OTT によって生成される C の識別子に大 / 小文字のどちらを使用するかを選択し、変換する型のリストを指定しています。そのうち 2 つの型については、ネーミング規則を指定しています。

次の例は、OTT の実行後の Outtype ファイルの内容を示しています。

```
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
```

Outtype ファイルの内容を検証すると、Intype 仕様部で指定されていなかった型がリスト表示されているのを見かけることがあります。たとえば、次のように、Intype ファイルでは *person* 型の変換のみを指定した場合を考えます。

```
CASE = LOWER
TYPE PERSON
```

そして、*person* 型の定義に *address* 型の属性があり、Outtype ファイルに *PERSON* と *ADDRESS* の両方のエントリがあるとします。*person* 型を完全に変換するには、最初に *address* を変換する必要があります。

19-6 ページの「[OTT コマンドライン](#)」で説明したように、OTT は Intype ファイル内の型相互の依存性を分析してから変換を行い、必要に応じて他の型も変換します。

OCI アプリケーションでの OTT の使用

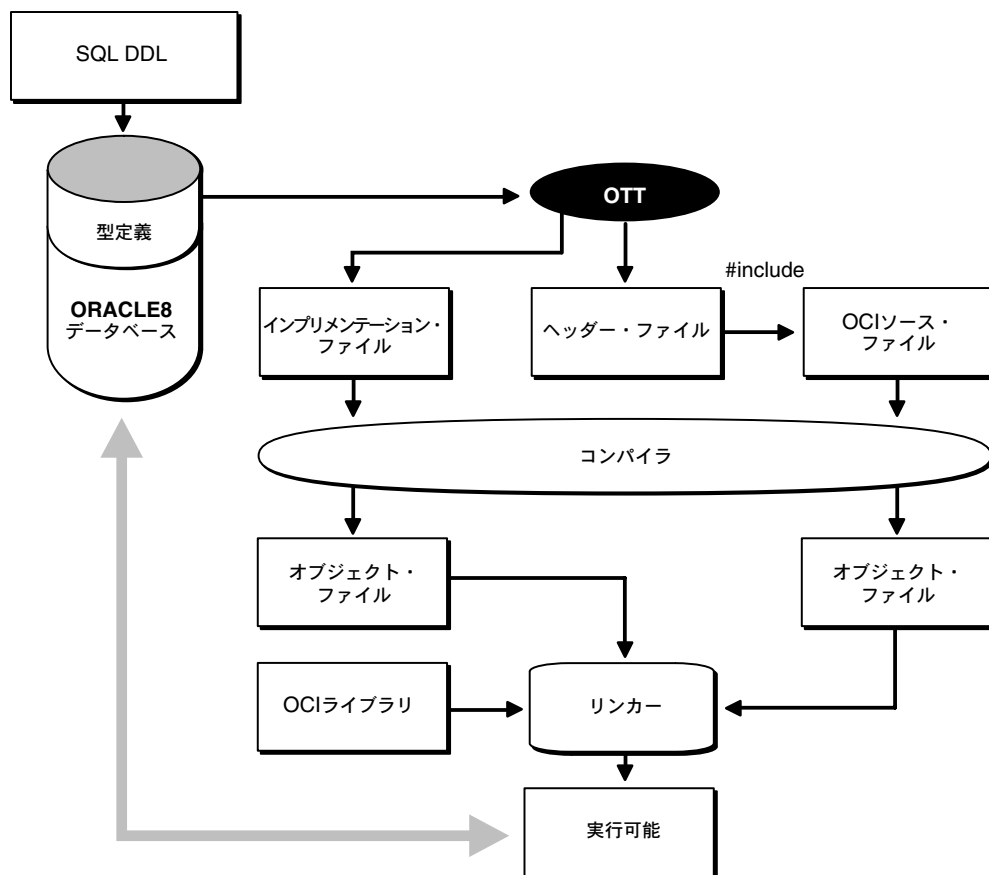
OTT によって生成される C のヘッダー・ファイルとインプリメンテーション・ファイルは、データベース・サーバーのオブジェクトにアクセスする OCI アプリケーションで使うことができます。ヘッダー・ファイルを OCI コードに取り込むには、`#include` 文を使います。

OCI アプリケーションでは、ヘッダー・ファイルを組み込んだ後、ホスト言語形式のオブジェクト・データにアクセスし、操作できます。

図 19-1 は、OCI で OTT を使用するにあたって必要となるステップの説明です。

1. SQL を使用してデータベースに型定義を作成します。
2. OTT を使用して、C 表現によるオブジェクト型と名前付きコレクション型を含むヘッダー・ファイルを生成します。また、`INITFILE` オプションを使用して名前が付けられたインプリメンテーション・ファイルも生成します。
3. アプリケーションを記述します。OCI アプリケーションでユーザーが記述したコードで、`INITFUNC` 関数を宣言してコールします。
4. ヘッダー・ファイルを OCI ソース・コード・ファイルに組み込みます。
5. OCI アプリケーションを OTT 生成のインプリメンテーション・ファイルとともにコンパイルして OCI ライブラリにリンクします。
6. OCI 実行可能ファイルを Oracle8i Server で実行します。

図 19-1 OCI での OTT の使用方法



OCI によるオブジェクトのアクセスと操作

アプリケーション内では、OTT によって生成されたヘッダー・ファイルに表示される型を持つように宣言されたプログラム変数を使用すると、OCI プログラムでバインド操作と定義操作を実行できます。

たとえば、アプリケーションで SQL の SELECT 文を使用してオブジェクトへの REF をフェッチし、適切な OCI 関数を使用してそのオブジェクトを確保します。オブジェクトを確保した後、その他の OCI 関数を使用してそのオブジェクトの属性データにアクセスし、操作できます。

OCI には、データ型のマッピングと操作のための一連の関数が組み込まれています。これらの関数は、オブジェクト型と名前付きコレクション型の属性を操作するために明確に設計されています。

次に、使用可能な関数の一部を示します。

- `OCIStrSize()` は、`OCIString` 文字列のサイズを取得します。
- `OCINumberAdd()` は、2 つの `OCINumber` 数値を加算します。
- `OCILobIsEqual()` は、2 つの LOB ロケータが等しいかを比較します。
- `OCIRawPtr()` は、`OCIRaw` のロー・データ型へのポインタを入手します。
- `OCICollAppend()` は、コレクション型 (`OCIArray` または `OCITable`) に要素を追加します。
- `OCITableFirst()` は、NESTED TABLE (`OCITable`) の最初の既存要素の索引を戻します。
- `OCISRefIsNull()` は、REF (`OCISRef`) が NULL かどうかテストします。

これらの関数の詳細は、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』の次の章で詳しく説明しています。

- 第 2 章 (バインディングと定義付けなどの OCI 概念の説明)
- 第 6 章 (オブジェクトへのアクセスとナビゲーションの説明)
- 第 7 章 (データ型のマップと操作の説明)
- 第 12 章 (データ型のマップと操作の関数のリスト)

初期化関数のコール

OTT では、必要に応じて C の初期化関数が生成されます。初期化関数では、プログラムで使用されている各オブジェクト型について、どのバージョンの型が使用されているかを環境に通知します。初期化関数名は、OTT の起動時に `INITFUNC` オプションを使用して指定できます。また、その関数を含むインプリメンテーション・ファイル (`INITFILE`) の名前に基づいて、OTT でデフォルト名を選択することもできます。

初期化関数には、環境ハンドル・ポインタとエラー・ハンドル・ポインタの 2 つの引数があります。一般的に、使用する初期化関数は 1 つですが、必ずしもそうである必要はありません。プログラムがいくつかの部分に分かれて個別にコンパイルされており、それぞれの部分に異なる型が必要な場合は、必要な部分ごとに、OTT を実行します。この場合は、それぞれの部分ごとに初期化関数を含んだ初期化ファイルが作成されます。

`OCISetEnv()` をコールするなど、明示的な OCI オブジェクト・コールによって環境ハンドルを作成した後に、環境ハンドルごとに初期化関数も明示的にコールする必要があります。これによって、各ハンドルからプログラム全体で使用されるすべてのデータ型にアクセスできます。

`EXEC SQL CONTEXT USE` や `EXEC SQL CONNECT` などの埋込み SQL 文を使用して環境ハンドルを暗黙的に作成する場合、ハンドルは暗黙的に初期化され、初期化関数をコールする必要はありません。これは、Pro*C/C++ アプリケーションまたは Pro*C/C++ と OCI アプリケーションが併用される場合に適用されます。

次に、初期化関数の例を示します。

Intype ファイルの `ex2c.typ` を指定します。次を含みます。

```
TYPE SCOTT.PERSON
TYPE SCOTT.ADDRESS
```

そして、次のコマンドラインを含みます。

```
ott userid=scott/tiger intype=ex2c outtype=ex2co hfile=ex2ch.h initfile=ex2cv.c
```

OTT によって、次のようなファイル `ex2cv.c` が生成されます。

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword ex2cv(OCIEnv *env, OCIError *err)
{
    sword status = OCITypeVtInit(env, err);
    if (status == OCI_SUCCESS)
        status = OCITypeVtInsert(env, err,
            "SCOTT", 5,
            "PERSON", 6,
            "$8.0", 4);
    if (status == OCI_SUCCESS)
        status = OCITypeVtInsert(env, err,
            "SCOTT", 5,
            "ADDRESS", 7,
            "$8.0", 4);
    return status;
}
```

関数 `ex2cv` によってタイプ・バージョン表が作成され、型 `SCOTT.PERSON` および `SCOTT.ADDRESS` が挿入されます。

プログラムで明示的に環境ハンドルを作成する場合、明示的に作成するハンドルごとに初期化関数をコールする必要があるため、すべての初期化関数を生成し、コンパイルし、リンクする必要があります。プログラムが明示的に環境ハンドルを作成しない場合、初期化関数は必要ありません。

OTT で生成したヘッダー・ファイルを使用するプログラムでは、同時に生成された初期化関数も使用する必要があります。具体例として、コンパイルによってプログラム P にリンクされるコードが生成される場合を考えます。OTT によって生成されるヘッダー・ファイルをコンパイル環境に組み込み、プログラム P のどこかで環境ハンドルが明示的に作成される場合は、それと同時に OTT によって生成されるインプリメンテーション・ファイルもコンパイルしてプログラム P にリンクする必要があります。この操作を正しく実行するのは、ユーザー側の責任になります。

初期化関数のタスク

C の初期化関数は、OTT で処理される型のバージョン情報を提供します。C 初期化関数は、OTT で処理する各オブジェクト・データ型の名前とバージョン識別子をタイプ・バージョン表に追加します。

オープン・タイプ・マネージャ (OTM) では、個々のプログラムで使用する型のバージョンを判別するためにタイプ・バージョン表を使用します。OTT によって様々な初期化関数が別々に生成されると、タイプ・バージョン表に同じ型がいくつも追加されることがあります。型が何度も追加されると、そのたびに OTM によって同じバージョンの型が登録されているかどうかを確認されます。

初期化関数の関数プロトタイプの宣言と関数のコールは、プログラマが行います。

注意 : Oracle8i のカレント・リリースでは、それぞれの型のバージョンは 1 つのみです。タイプ・バージョン表の初期化は、Oracle8i の将来のリリースとの互換性のためのみに必要です。

Pro*C/C++ アプリケーションでの OTT の使用

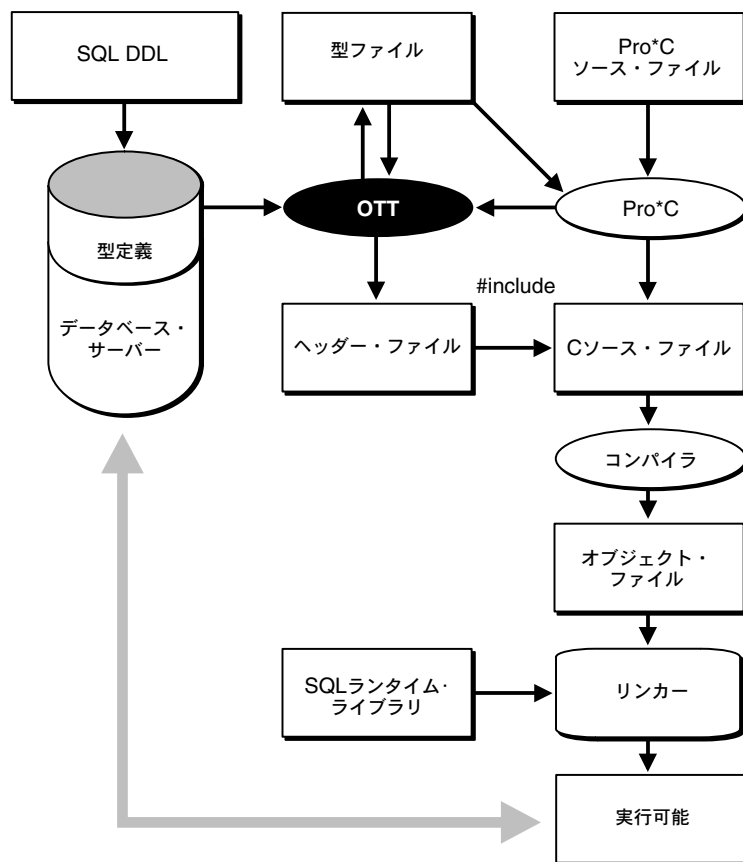
Pro*C/C++ アプリケーションの作成時の型の変換処理は、OCI ベースのアプリケーションの作成時よりも簡単です。これは、プリコンパイラで生成されるコードによって自動的にタイプ・バージョン表が初期化されるためです。

Pro*C/C++ アプリケーションでは、OTT によって生成される C のヘッダー・ファイルを使用して、データベース・サーバーのオブジェクトにアクセスできます。このヘッダー・ファイルは、`#include` 文によりコードに組み込まれます。ヘッダー・ファイルを組み込んだ後は、Pro*C/C++ アプリケーションからホスト言語の書式でオブジェクト・データにアクセスしたりオブジェクト・データを操作できます。

図 19-2 は、Pro*C/C++ で OTT を使用するにあたって必要となるステップを示しています。

1. SQL を使用してデータベースに型定義を作成します。
2. OTT を使用して、C 表現によるオブジェクト型、REF 型および名前付きコレクション型を含むヘッダー・ファイルを生成します。INTYPE パラメータとして Pro*C/C++ に渡される OUTTYPE ファイルも生成されます。
3. ヘッダー・ファイルを Pro*C/C++ ソース・コード・ファイルに組み込みます。
4. Pro*C/C++ アプリケーションをコンパイルし、Pro*C/C++ のランタイム・ライブラリ SQLLIB にリンクします。
5. Pro*C/C++ 実行可能ファイルを Oracle8i Server で実行します。

図 19-2 オブジェクト指向 Pro*C/C++ アプリケーションの作成



前述のステップ 2 が示すように、OTT によって生成される OUTTYPE ファイルは、Pro*C/C++ プログラマにとって特別な用途があります。Pro*C/C++ の起動時に OUTTYPE ファイルを新しい INTYPE コマンドライン・パラメータに渡します。このファイルの内容は、OTT 生成の構造体に対応付けるデータベース型を決定するためにプリコンパイラで使用されます。OCI でプログラミングしている場合は、バインド、定義および型情報へのアクセスのための特別な関数を使用して、明示的にこの対応付けを行う必要があります。

また、プリコンパイラにより、OTT OUTTYPE (Pro*C/C++ INTYPE) ファイルにおいて名前が付けられた型を使用して、タイプ・バージョン表を初期化するコードが生成されます。

注意: 常に OTT からの OUTTYPE ファイルを Pro*C/C++ の INTYPE ファイルとして使用することをお勧めします。Pro*C/C++ の INTYPE ファイルを作成することも可能ですが、エラーが発生する可能性もあるためお勧めしません。

サーバーから取り出されたオブジェクトの属性を操作するには、OCI のデータ型のマッピングと操作のための関数をコールするのも 1 つの方法です。これを実行する前に、アプリケーションで、まず `SQLEnvGet()` をコールして OCI 関数に渡す OCI 環境ハンドルを取得し、次に `SQLSvcCtxGet()` をコールして OCI 関数に渡す OCI サービス・コンテキストを取得する必要があります。Pro*C には、オブジェクト属性の操作に使用できる機能もあります。詳細は、[第 17 章「オブジェクト」](#)を参照してください。

Pro*C/C++ から OCI をコールするプロセスの簡単な説明は、19-18 ページの「[OCI によるオブジェクトのアクセスと操作](#)」を参照してください。詳細は、『Oracle8i コール・インタフェース・プログラマーズ・ガイド』の第 8 章を参照してください。

OTT 参照

OTT の動作を制御するパラメータは、OTT コマンドラインでも CONFIG ファイル内でも指定できます。また、一部のパラメータは、INTYPE ファイルにも指定できます。この項では、次のトピックについて詳しく説明します。

- [OTT コマンドライン構文](#)
- [OTT パラメータ](#)
- [OTT パラメータの使用場所](#)
- [Intype ファイルの構造](#)
- [ネストした #include ファイル生成](#)
- [SCHEMA_NAMES の使用方法](#)
- [デフォルト名のマッピング](#)
- [制限](#)

この章では、次の規則を使用して OTT の構文を説明します。

- イタリック文字列はユーザーが入力する内容です。
- 大文字の文字列は、そのとおりに入力する文字列です。ただし、大 / 小文字の区別はされないで、小文字で入力しても有効です。
- 大カッコ [...] で囲んだ項目は、オプション項目です。
- 1 つの項目（あるいはカッコで囲まれた複数の項目）のすぐ後の省略記号 (...) は、その項目を何度も繰り返し指定できることを示します。
- これ以外の句読点記号は、示されているとおりに入力します。これには、','、'@" などが含まれます。

OTT コマンドライン構文

OTT コマンドライン・インタフェースは、OTT を明示的に起動してデータベース型を C の構造体に変換するときに使用されます。このインタフェースは、オブジェクトを活用する OCI アプリケーションまたは Pro*C/C++ アプリケーションの開発に必須です。

OTT コマンドラインの文は、キーワード *OTT* と、後続の OTT パラメータ・リストで構成されます。

OTT コマンドラインの文で指定できるパラメータは、次のとおりです。

```
[USERID=username/password[@db_name]]  
[INTYPE=in_filename]  
OUTTYPE=out_filename  
CODE={C|ANSI_C|KR_C}  
[HFILE=filename]  
[ERRTYPE=filename]  
[CONFIG=filename]  
[INITFILE=filename]  
[INITFUNC=filename]  
[CASE={SAME|LOWER|UPPER|OPPOSITE}]  
[SCHEMA_NAMES={ALWAYS|IF_NEEDED|FROM_INTYPE}]
```

注意：一般に OTT コマンドの後に続くパラメータはどのような順序でもよく、OUTTYPE および CODE パラメータのみが常に必要とされます。

HFILE パラメータは、ほとんどいつも使用されます。省略すると、INTYPE ファイルのそれぞれの型について、個別に HFILE を指定する必要があります。OTT が Intype ファイルで指定されていない型を変換する必要があると判断すると、エラーがレポートされます。したがって、INTYPE ファイルが以前に OTT OUTTYPE ファイルとして生成された場合のみ、HFILE パラメータは省略できます。

INTYPE ファイルを省略すると、スキーマ全体が変換されます。詳細は、この次の項にあるパラメータの説明を参照してください。

次に OTT コマンドライン文の例を示します（1 行に入力します）。

```
OTT userid=scott/tiger intype=in.typ outtype=out.typ code=c hfile=demo.h  
errtype=demo.tls case=lower
```

OTT コマンドラインの各パラメータについて、この後の各項で説明します。

OTT パラメータ

OTT コマンドラインにパラメータを入力するときの書式は、次のとおりです。

parameter=value

parameter はリテラル・パラメータ文字列であり、*value* は有効なパラメータ設定値です。リテラル・パラメータ文字列は大 / 小文字を区別しません。

コマンドラインのパラメータは、空白またはタブのいずれかを使用して区切ります。

また、パラメータは構成ファイル内でも指定できます。ただし、この場合、行の中に空白を入れることはできないので、各パラメータは独立した行に指定する必要があります。さらに、パラメータの CASE、HFILE、INITFUNC および INITFILE は INTYPE ファイルに入れることができます。

USERID

USERID パラメータでは、Oracle ユーザー名、パスワードおよびオプションのデータベース名 (Net8 のデータベース指定文字列) を指定します。データベース名を省略すると、デフォルトのデータベースが想定されます。このパラメータの構文は、次のとおりです。

USERID=username/password[@db_name]

これが第 1 パラメータである場合は、"USERID=" を省略して、次のように指定できます。

OTT ユーザー名 / パスワード ...

USERID パラメータはオプションです。省略した場合、OTT は自動的にユーザー OPS\$*username* としてデフォルトのデータベースに接続を試みます。このとき *username* はユーザーのオペレーティング・システムのユーザー名になります。

INTYPE

INTYPE パラメータでは、オブジェクト型指定のリストを読み込む元のファイルの名前を指定します。OTT は、このリストに含まれる型を 1 つずつ変換します。このパラメータの構文は、次のとおりです。

INTYPE=filename

USERID が第 1 パラメータ、INTYPE が第 2 パラメータで、"USERID=" を省略した場合は、"INTYPE=" も省略できます。INTYPE が指定されていない場合は、ユーザーのスキーマにおけるすべての型が変換されます。

OTT *username/password filename...*

INTYPE ファイルは、型宣言の Make ファイルと考えることができます。C 構造体宣言の必要な型を INTYPE ファイルにリストします。Intype ファイルの形式は、19-29 ページの「[Intype ファイルの構造](#)」で説明しています。

コマンドラインまたは INTYPE ファイルのファイル名に拡張子が含まれていない場合、"TYP" や "typ" などのプラットフォーム特定の拡張子が追加されます。

OUTTYPE

OTT によって処理されるすべてのオブジェクト・データ型の型情報が書き込まれるファイルの名前を指定します。OUTTYPE ファイルには、INTYPE ファイルで明示的に指定したすべての型が含まれます。それに加えて、変換の対象である他の型の宣言で使用しているために変換された型が含まれる場合もあります。このファイルは、それ以後に OTT を起動するときに Intype ファイルとして使用できます。

OUTTYPE=*filename*

INTYPE パラメータと OUTTYPE パラメータが同一のファイルを参照している場合、INTYPE ファイルの古い情報は、新しい INTYPE の情報に置き換えられます。このことは、型の変更から型宣言の生成、ソースコードの編集、プリコンパイル、コンパイル、デバッグまでのサイクルで、同一の INTYPE ファイルを繰り返し使用するとき便利です。

OUTTYPE は必ず指定します。

コマンドラインまたは INTYPE ファイルのファイル名に拡張子が含まれていない場合、"TYP" や "typ" などのプラットフォーム特定の拡張子が追加されます。

CODE

CODE=C|KR_C|ANSI_C

OTT の出力を表すホスト言語を指定します。この場合、CODE=C、CODE=KR_C、CODE=ANSI_C のいずれかを指定できます。「CODE=C」は「CODE=ANSI_C」と同じ意味です。

このパラメータは、デフォルト値がないので必ず指定する必要があります。

INITFILE

INITFILE パラメータでは、OTT で生成した初期化ファイルを書き込むファイルの名前を指定します。このパラメータを省略すると、初期化関数は生成されません。

Pro*C/C++ プログラムの場合、必要な初期化は SQLLIB ランタイム・ライブラリによって実行されるので、INITFILE は必要ありません。OCI プログラムのユーザーは、INITFILE ファイルをコンパイルおよびリンクし、環境ハンドルの作成時に初期化関数をコールする必要があります。

コマンドラインまたは INTYPE ファイルで指定した INITFILE ファイル名に拡張子を付けなかった場合、".C" または ".c" のようなプラットフォーム固有の拡張子が追加されます。

INITFILE=*filename*

INITFUNC

INITFUNC パラメータが使用されるのは、OCI プログラムの場合のみです。このパラメータでは、OTT 生成の初期化関数の名前を指定します。このパラメータを省略すると、INITFILE の名前から初期化関数の名前が付けられます。

INITFUNC=*filename*

HFILE

Intype ファイルで型の宣言を指定し、インクルード（.h）ファイルを指定しなかった場合に、OTT によって生成されるインクルード・ファイルの名前を指定します。INTYPE ファイルで各型のインクルード・ファイルを個々に指定していない場合は、このパラメータが必要です。INTYPE ファイルに記述していない型を、2 つ以上の異なるファイルで宣言した他の型で使用する場合は、INTYPE ファイルに記述していない型も生成する必要があります。そのような場合もこのパラメータが必要です。

コマンドラインまたは INTYPE ファイルで指定した HFILE ファイル名に拡張子を付けなかった場合、".H" や ".h" のようなプラットフォーム固有の拡張子が追加されます。

HFILE=*filename*

CONFIG

CONFIG パラメータでは、共通に使用されるパラメータ指定を含む OTT 構成ファイルの名前を指定します。また、パラメータ指定は、プラットフォームによって異なる位置にあるシステム構成ファイルから読み込まれます。残りのすべてのパラメータ指定は、コマンドラインまたは INTYPE ファイルで指定する必要があります。

CONFIG=*filename*

注意： CONFIG パラメータは、config ファイルでは使用できません。

ERRTYPE

このパラメータを指定すると、Intype ファイルのリストが、すべての情報メッセージおよびエラー・メッセージとともに ERRTYPE ファイルに書き込まれます。情報メッセージおよびエラー・メッセージは、ERRTYPE を指定したかどうかに関係なく、標準出力に送信されます。

実質的に、ERRTYPE ファイルはエラー・メッセージが追加された INTYPE ファイルのコピーです。ほとんどの場合、エラー・メッセージには、エラーの原因となったテキストへのポインタが含まれます。

コマンドラインまたは INTYPE ファイルの ERRTYPE ファイル名に拡張子が付いていない場合、".TLS" または ".tls" のようなプラットフォーム固有の拡張子が追加されます。

ERRTYPE=*filename*

CASE

このパラメータでは、OTT 生成の特定の C の識別子を大文字で表記するか小文字で表記するかを指定します。CASE の可能な値は、SAME、LOWER、UPPER、OPPOSITE です。CASE = SAME の場合は、データベース型と属性名を C 識別子に変換するとき、文字の大 / 小文字は変更されません。CASE = LOWER とすると、大文字はすべて小文字に変換されます。CASE = UPPER とすると、小文字はすべて大文字に変換されます。CASE = OPPOSITE とすると、大文字はすべて小文字に変換され、小文字はすべて大文字に変換されます。

CASE=[SAME|LOWER|UPPER|OPPOSITE]

このパラメータは、INTYPE ファイル内で指定していない識別子（明示的に指定していない属性または型）にのみ影響します。大 / 小文字の変換は、正当な識別子が生成された後で行われます。

注意: INTYPE で特定された型の C 構造体識別子の大 / 小文字の区別は、INTYPE ファイルにおける大 / 小文字と同じです。たとえば、INTYPE ファイルに次の行が含まれる場合、

```
TYPE Worker
```

OTT によって次のように生成されます。

```
struct Worker {...};
```

一方で、INTYPE ファイルに次のように記述したとします。

```
TYPE wOrKeR
```

OTT によって次のように生成されます。

```
struct wOrKeR {...};
```

これは INTYPE ファイルの大 / 小文字区別どおりです。

INTYPE ファイルに記述されていない、大 / 小文字の区別のない SQL 識別子は、CASE=SAME の場合は大文字で、CASE=OPPOSITE の場合は小文字で指定します。宣言されるとき引用符が付かなかった SQL 識別子は、大 / 小文字の区別はありません。

SCHEMA_NAMES

このパラメータでは、デフォルトのスキーマからの型のデータベース名を Outtype ファイル内のスキーマ名で修飾する操作を制御できます。OTT 生成の Outtype ファイルには、型の名前など、OTT の処理対象となる型の情報が含まれています。

詳細は、19-33 ページの「[SCHEMA_NAMES の使用方法](#)」を参照してください。

OTT パラメータの使用場所

OTT のパラメータは、コマンドライン、またはコマンドラインで指定する config ファイル内で指定します。パラメータの一部は、INTYPE ファイルでも指定できます。

OTT を起動するには、次のように入力します。

```
OTT username/password parameters
```

コマンドラインのパラメータの 1 つが次の場合、

```
CONFIG=filename
```

構成ファイル *filename* からその他のパラメータが読み込まれます。

さらに、パラメータは、プラットフォームによって異なる位置にあるデフォルトの構成ファイルから読み込まれます。このファイルは、常に存在する必要がありますが、空でもかまいません。構成ファイルには、各パラメータを 1 行に 1 つずつ、空白を使用しないで入力する必要があります。

引数を指定しないで OTT を実行すると、オンラインのパラメータ参照が表示されます。

OTT の変換対象の型は、INTYPE パラメータで指定されるファイルにおいて名前が指定されます。パラメータの CASE、INITFILE、INITFUNC および HFILE は、INTYPE ファイルに入れることもできます。OTT 生成の Outtype ファイルには CASE パラメータが含まれ、初期化ファイルが生成されている場合は INITFILE および INITFUNC パラメータが含まれます。OUTTYPE ファイルでは、型ごとにそれぞれ HFILE を指定します。

OTT コマンドの大 / 小文字区別は、プラットフォームによって異なります。

Intype ファイルの構造

Intype および Outtype ファイルでは、OTT により変換される型がリストされ、型または属性名の有効な C 識別子への変換方法を決定するにあたり必要となる全情報が提供されます。これらのファイルには、1 つ以上の型指定を記述します。また、次のオプションを指定する場合もあります。

- CASE
- HFILE
- INITFILE
- INITFUNC

CASE、INITFILE または INITFUNC オプションを指定する場合は、すべての型指定よりも前に指定する必要があります。これらのオプションをコマンドラインと intype ファイルの両方に指定した場合は、コマンドラインの値が使用されます。

簡単なユーザー定義の Intype ファイルと、それから OTT により生成される完全な Outtype ファイルの例は、19-15 ページの「[Outtype ファイル](#)」を参照してください。

Intype ファイルの型指定

INTYPE での型指定によって、これから変換するオブジェクト・データ型の名前を指定します。ユーザー作成 intype ファイルの例を次に示します。

```
TYPE employee
  TRANSLATE SALARY$ AS salary
  DEPTNO AS department
TYPE ADDRESS
TYPE PURCHASE_ORDER AS p_o
```

型指定の構造は、次のとおりです。

```
TYPE type_name [AS type_identifier]
[VERSION [=] version_string]
[HFILE [=] hfile_name]
[TRANSLATE{member_name [AS identifier]}...]

```

type_name の構文は、次のとおりです。

```
[schema_name.]type_name
```

この場合、*schema_name* は、指定されたオブジェクト・データ型を所有するスキーマの名前で、*type_name* はその型の名前です。デフォルト・スキーマは、OTT を実行するユーザーのスキーマです。デフォルト・データベースは、ローカル・データベースです。

型指定のコンポーネントは、次のとおりです。

- *type name* はオブジェクト・データ型の名前です。
- *type identifier* は型を表すのに使用される C 識別子です。省略すると、デフォルトの名前マッピング・アルゴリズムが使用されます。詳細は、19-35 ページの「[デフォルト名のマッピング](#)」を参照してください。
- *version string* は、OTT の前起動によってコードが生成されたときに使用された型のバージョン文字列です。バージョン文字列は OTT によって生成され、**Outtype** ファイルに書き込まれます。このファイルは、後で OTT を実行するときに **Intype** ファイルとして使用することができます。バージョン文字列は OTT の操作には影響を与えませんが、最終的にこれを使用して、起動中のプログラムで使用されるオブジェクト型のバージョンが選択されます。
- *hfile name* は、該当する構造またはクラスの宣言が表れるヘッダー・ファイルの名前です。*hfile name* を省略すると、宣言の生成時にはコマンドラインの HFILE パラメータで指定したファイルが使用されます。
- *member name* は、次の *identifier* に変換される属性（データ・メンバー）の名前です。
- *identifier* はユーザー・プログラムの属性を表すのに使用される C 識別子です。この方法で、必要な数の属性の識別子を指定できます。指定していない属性については、デフォルトの名前マッピング・アルゴリズムが使用されます。

オブジェクト・データ型は、次のいずれかの場合に変換する必要があります。

- INTYPE ファイルに指定されている場合
- 変換が必要な別の型の宣言で使用されている場合

明示的に記述していない型があり、その型が、正確に 1 つのファイルのみに宣言した型で必要だとします。この場合、明示的に記述していない型の変換結果は、それを必要とする明示的に宣言した型と同じファイルに書き込まれます。

明示的に記述していない型があり、その型が、複数の異なるファイルに宣言した型が必要だとします。この場合、要求された型の変換結果は、グローバルな HFILE ファイルに書き込まれます。

ネストした #include ファイル生成

OTT によって生成されるすべての HFILE で、他の必要なファイルが #includes を使用して組み込まれ、そのファイルの名前から組み立てられた記号が #defines を使用して定義されます。この記号は、HFILE がすでに組み込まれているかどうかを判断する場合に使用することができます。たとえば、データベースに次の型があるとします。

```
create type px1 AS OBJECT (col1 number, col2 integer);
create type px2 AS OBJECT (col1 px1);
create type px3 AS OBJECT (col1 px1);
```

intype ファイルは次のとおりです。

```
CASE=lower
type px1
  hfile tott95a.h
type px3
  hfile tott95b.h
```

次のようにして OTT を起動した場合、

```
ott scott/tiger tott95i.typ outtype=tott95o.typ code=c
```

次の 2 つのヘッダー・ファイルが生成されます。

ファイル tott95b.h は次のとおりです。

```
#ifndef TOT95B_ORACLE
#define TOT95B_ORACLE
#endif
#include <oci.h>
#endif
#ifndef TOT95A_ORACLE
#include "tott95a.h"
#endif
typedef OCISRef px3_ref;
struct px3
{
    struct px1 col1;
};
typedef struct px3 px3;
struct px3_ind
{
    OCISInd_atomic;
```

```
    struct px1_ind coll
};
typedef struct px3_ind px3_ind;
#endif
```

ファイル `tott95a.h` は次のとおりです。

```
#ifndef TOTTT95A_ORACLE
#define TOTTT95A_ORACLE
#endif
#include <oci.h>
typedef OCISvcCtx px1_ref;
struct px1
{
    OCISvcCtx coll1;
    OCISvcCtx coll2;
}
typedef struct px1 px1;
struct px1_ind
{
    OCISvcCtx _atomic;
    OCISvcCtx coll1;
    OCISvcCtx coll2;
}
typedef struct px1_ind px1_ind;
#endif
```

このファイルでは、`TOTTT95B_ORACLE` という記号を最初に定義しています。そのため、プログラムは、次の構造体を使用して `tott95b.h` を条件付きで組み込むことができます。その際、`tott95b.h` がインクルード・ファイルに依存しているかどうかを考慮する必要はありません。

```
#ifndef TOTTT95B_ORACLE
#include "tott95b.h"
#endif
```

このテクニックを使用すると、`"foo.h"` などのファイルから `"tott95b.h"` を組み込むことができます。この場合、`"foo.h"` から組み込まれる他のファイルに `"tott95b.h"` が含まれているかどうかを確認する必要はありません。

記号 `TOTTT95B_ORACLE` の定義の後に、ファイル `oci.h` が `#included` によって組み込まれています。OTT 生成のすべての HFILE には、Pro*C/C++ や OCI のプログラムに役立つ型と関数の宣言が含まれた `oci.h` が組み込まれます。これは、OTT で `#include` に山カッコが使用されている場合のみです。

次に、ファイル `tott95a.h` が組み込まれるのは、`tott95a.h` に必要な `"struct px1"` の宣言が含まれているためです。Intype ファイルにより型宣言の複数ファイルへの書込みが要求される場

合、OTT により、その他各 HFILE に組み込む必要があるファイルが決定され、必要な `#includes` が生成されます。

この `#include` では引用符が使用されるので注意してください。`tott95b.h` を含むプログラムをコンパイルすると、`tott95a.h` の検索はソース・プログラムの検出位置から開始され、それ以降はインプリメンテーション定義の検索規則に従って実行されます。この方法で `tott95a.h` が検出されない場合は、INTYPE ファイル内で完全ファイル名（/ で始まる UNIX の絶対パス名など）を使用して、`tott95a.h` の位置を指定する必要があります。

SCHEMA_NAMES の使用方法

このパラメータでは、OTT の接続先となるデフォルト・スキーマに含まれる型の名前を、OUTTYPE ファイル内のスキーマ名で修飾するかどうかを指定します。

デフォルト・スキーマ以外のスキーマに基づく型の名前は、OUTTYPE ファイル内のスキーマ名で常に修飾されます。

スキーマ名で修飾するかしないかで、プログラム実行中に型がどのスキーマで検索されるかが決定します。

次の 3 通りの設定があります。

- `SCHEMA_NAMES=ALWAYS` (デフォルト)

OUTTYPE ファイル内のすべての型名をスキーマ名で修飾します。

- `SCHEMA_NAMES=IF_NEEDED`

デフォルト・スキーマに属する OUTTYPE ファイル内の型名はスキーマ名で修飾しません。デフォルト・スキーマ以外のスキーマに属する型名は、スキーマ名で修飾します。

- `SCHEMA_NAMES=FROM_INTYPE`

INTYPE ファイルに記述されている型は、INTYPE ファイル内のスキーマ名で修飾されている場合のみ、OUTTYPE ファイル内のスキーマ名で修飾されます。デフォルト・スキーマ内の型は、INTYPE ファイル内で指定されていなくても、型相互の依存性によっては生成する必要があります。このような型がスキーマ名で修飾されるのは、その型に依存する型として OTT で最初に検出された型がスキーマ名で修飾されていた場合のみです。ただし、OTT の接続先となるデフォルト・スキーマ内で設定されていない型は、常に明示的なスキーマ名で修飾されます。

OTT 生成の OUTTYPE ファイルは、Pro*C/C++ の INTYPE ファイルになります。このファイルは、データベース型名を C 構造体名と対応付けます。この情報は、構造体内で正しいデータベース型が確実に選択されるようにするために、実行時に使用されます。OUTTYPE ファイル (Pro*C/C++ INTYPE ファイル) 内のスキーマ名で型が指定される場合、その型は、プログラム実行中に名前付きスキーマ内で検索されます。型がスキーマ名なしで表示される場合、そのファイルはプログラムの接続先となるデフォルト・スキーマ内で検出されます。このデフォルト・スキーマは、OTT で使用されたデフォルト・スキーマと異なる場合があります。

例

SCHEMA_NAMES に FROM_INTYPE を設定し、INTYPE ファイルで読み込みます。

```
TYPE Person
TYPE joe.Dept
TYPE sam.Company
```

この場合、OTT 生成の構造体を使用する Pro*C/C++ アプリケーションでは、sam.Company、joe.Dept および Person の 3 つの型を使用します。Person 型にはスキーマ名が付いていないので、このアプリケーションが接続されているスキーマ内の Person 型が参照されます。

OTT とアプリケーションの両方がスキーマ joe に接続すると、アプリケーションでは OTT と同じ型 (joe.Person) が使用されます。OTT がスキーマ joe に接続しても、アプリケーションがスキーマ mary に接続した場合、このアプリケーションでは型 mary.Person が使用されます。この動作が有効なのは、スキーマ joe とスキーマ mary で同じ "CREATE TYPE Person" 文が実行された場合のみです。

一方、アプリケーションでは、どのスキーマに接続されているかに関係なく、joe.Dept 型が使用されます。この動作のためには、INTYPE ファイルに型名とともにスキーマ名を必ず記述する必要があります。

場合によっては、ユーザーが明示的に指定しなかった型が OTT によって変換されます。たとえば、次の SQL 宣言があるとして。

```
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
);
```

ここで、その OTT がスキーマ joe に接続し、SCHEMA_NAMES=FROM_INTYPE が指定され、ユーザーの INTYPE ファイルは次のいずれかを含むとします。

```
TYPE Person or TYPE joe.Person
```

型 joe.Address は指定されていない場合は、型 *joe.Person* にネストされたオブジェクト型として使用されています。"TYPE joe.Person" が INTYPE ファイルに記述されている場合は、OUTTYPE ファイルには "TYPE joe.Person" と "TYPE joe.Address" が表示されます。INTYPE

ファイルに "Type Person" が記述されている場合は、"TYPE Person" および "TYPE Address" が OUTTYPE ファイルに表示されます。

joe.Address 型が OTT によって変換された複数の型に埋め込まれていた場合にも、INTYPE ファイル内で明示的に指定されていなければ、埋め込まれた *joe.Address* 型が最初に検出されたときに、スキーマ名を使用するかどうかが決まります。なんらかの理由で、型 *joe.Address* にはスキーマ名を付け、型 *Person* にはスキーマ名を付けない場合は、次のように明示的に要求する必要があります。

```
TYPE      joe.Address
```

これは INTYPE FILE で要求します。

通常は、それぞれの型が 1 つのスキーマで宣言されるので、INTYPE ファイル内ですべての型の名前をスキーマ名で修飾するのが最も安全な方法です。

デフォルト名のマッピング

OTT は、オブジェクト型または属性を表す C の識別子名を作成するときに、その名前をデータベースのキャラクタ・セットから有効な C の識別子に変換します。最初に、名前はデータベースのキャラクタ・セットから OTT で使用されるキャラクタ・セットに変換されます。次に、その変換された名前の変換内容が INTYPE ファイルに供給される場合は、その変換内容が使用されます。それ以外の場合、名前は CASE オプションで指定したコンパイラのキャラクタ・セットに 1 文字ずつ変換されます。これについての詳細を次に説明します。

OTT によってデータベース・エンティティ名が読み込まれると、データベースのキャラクタ・セットから OTT で使用されるキャラクタ・セットに自動的に変換されます。OTT がデータベース・エンティティ名を正常に読み込むには、その名前のすべての文字が OTT のキャラクタ・セット内で検出される必要はありませんが、2 つのキャラクタ・セット間で文字コードが異なることがあります。

必要な文字が OTT で使用されるキャラクタ・セットにすべて含まれていることを保証するには、データベースのキャラクタ・セットと同じものを使用するのが最も簡単な方法です。ただし、OTT のキャラクタ・セットは、コンパイラのキャラクタ・セットのスーパーセットである必要があります。つまり、コンパイラのキャラクタ・セットが 7 ビット ASCII の場合、OTT のキャラクタ・セットはサブセットとして 7 ビット ASCII を含む必要があります。コンパイラのキャラクタ・セットが 7 ビット EBCDIC の場合は、OTT のキャラクタ・セットはサブセットとして 7 ビット EBCDIC を含む必要があります。OTT で使用されるキャラクタ・セットを指定するには、NLS_LANG 環境変数を設定する方法と、プラットフォーム固有の他のメカニズムを使用する方法があります。

データベース・エンティティの名前は、OTT によって読み込まれると、OTT で使用されるキャラクタ・セットから、コンパイラのキャラクタ・セットに変換されます。名前の変換が Intype ファイルで指定されていれば、その変換が使用されます。

それ以外の場合、名前は次のように変換されます。

1. 最初に、OTT のキャラクタ・セットがマルチバイト・キャラクタ・セットの場合、その名前にある、等価のシングルバイト文字を持つマルチバイト文字は、シングルバイト文字に変換されます。
2. 次に、その名前は、OTT のキャラクタ・セットからコンパイラのキャラクタ・セットに変換されます。コンパイラのキャラクタ・セットは、US7ASCII のように、シングルバイト・キャラクタ・セットです。
3. 最後に、有効になっている CASE オプションに従って、文字の大 / 小文字が設定されます。そして、C 識別子で無効な文字、またはコンパイラのキャラクタ・セットに変換内容がない文字は、アンダースコアに置き換えられます。1 文字でもアンダースコアに置き換えられると、OTT から警告メッセージが表示されます。名前に含まれる文字がすべてアンダースコアに置き換えられると、エラー・メッセージが表示されます。

文字単位の名前の変換では、コンパイラのキャラクタ・セットにあるアンダースコア、数字またはシングルバイト文字は変更されません。したがって、有効な C 識別子は変更されません。

たとえば、名前の変換では、ウムラウト (¨) の付いた "ö"、または抑音符 (˘) の付いた "a" などのシングルバイトのアクセント文字を、"o" または "a" に変換できます。そして、マルチバイト文字を等価のシングルバイト文字に変換できます。名前の変換は、その名前に等価のシングルバイトがないマルチバイト文字がある場合、通常は失敗します。この場合、ユーザーは、INTYPE ファイルでの名前の変換を指定する必要があります。

OTT では、同じ C の名前に複数のデータベース識別子がマップされたために発生する名前の重複は検出されません。また、データベース識別子が C のキーワードにマップされる場合に発生する命名の問題も検出されません。

制限

OTT を使用する場合、次の制限が適用されます。

ファイル名比較

現在、OTT では、2 つのファイルが同じかどうかは、ユーザーがコマンドラインまたは Intype ファイルで指定したファイル名を比較して判断されています。しかし、2 つのファイル名が同じファイルを参照するかどうかを OTT に認識させる必要がある場合は、問題が発生する可能性があります。たとえば、OTT 生成のファイル `foo.h` に、`foo1.h` に書き込まれた型の宣言と、`/private/smith/foo1.h` に書き込まれた別の型の宣言が必要な場合、OTT は 2 つのファイルが同じであれば `#include` を 1 つ、異なっていれば `#include` を 2 つ生成する必要があります。しかし、実際には OTT は 2 つのファイルが異なるものと見なして、次のように 2 つの `#include` を生成します。

```
#ifndef FOO1_ORACLE
#include "foo1.h"
#endif
```

```
#ifndef FOO1_ORACLE
#include "/private/smith/fool.h"
#endif
```

ファイル `fool.h` とファイル `/private/smith/fool.h` が異なっていれば、最初のファイルのみが組み込まれます。ファイル `fool.h` とファイル `/private/smith/fool.h` が同じであれば、`#include` は重複して記述されます。

そのため、コマンドラインまたは INTYPE ファイルでファイルを複数回記述するときは、各記述で正確に同じファイル名を使用する必要があります。

ユーザー・イグジット

この章では、Oracle Tools アプリケーション用のユーザー・イグジットの作成方法を説明します。C のサブルーチンを使用すると、SQL*Forms および Oracle Forms よりも迅速で、しかも簡単に特定の作業を実行できることがわかります。この章は、次のトピックで構成されています。

- ユーザー・イグジット
- ユーザー・イグジットを作成する理由
- ユーザー・イグジットの開発
- ユーザー・イグジットの作成
- ユーザー・イグジットのコール
- ユーザー・イグジットへのパラメータの引渡し
- フォームへの値のリターン
- 例
- ユーザー・イグジットのプリコンパイルおよびコンパイル
- サンプル・プログラム: ユーザー・イグジット
- GENXTB ユーティリティの使用方法
- ユーザー・イグジットの SQL*Forms へのリンク
- ガイドライン
- EXEC TOOLS 文

この章の内容は補足説明です。ユーザー・イグジットの詳細は、『SQL*Forms デザイナーズ・リファレンス』、『Oracle Forms リファレンス・マニュアル』、および各システム固有の Oracle ドキュメントを参照してください。

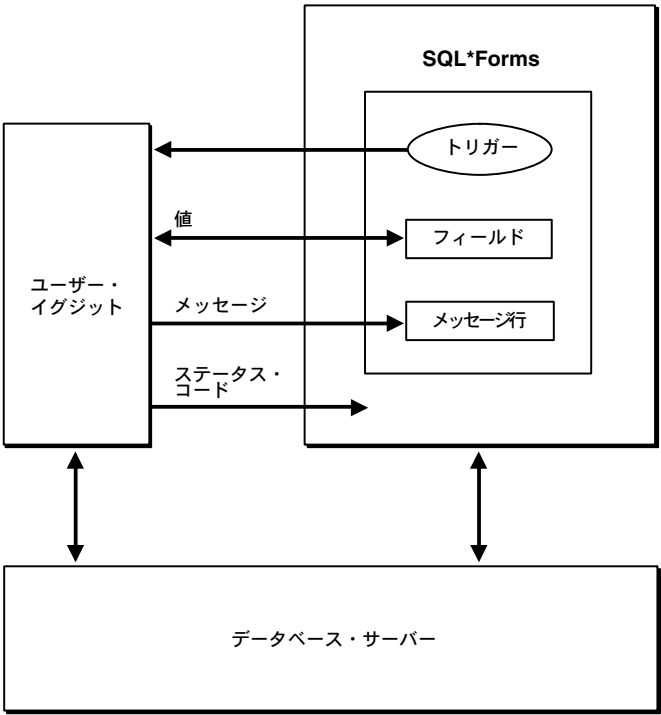
ユーザー・イグジット

ユーザー・イグジットとは、特別な目的の処理を実行するために作成する C のサブルーチンを指します。このサブルーチンは Oracle Forms によってコールされます。ユーザー・イグジットに SQL 文および PL/SQL ブロックを組み込み、その後ホスト・プログラムの場合と同様にこれをプリコンパイルできます。

Oracle Forms バージョン 3 のトリガーからユーザー・イグジットをコールすると、ユーザー・イグジットが実行され、その後ステータス・コードが Oracle Forms に戻されます。ユーザー・イグジットでは、Oracle Forms ステータス行へのメッセージ表示、フィールド値の取得と設定、高速計算と表参照および Oracle データの操作ができます。

図 20-1 では、Oracle Forms のアプリケーションとユーザー・イグジットの対話方法が示されています。

図 20-1 Oracle Forms とユーザー・イグジット



ユーザー・イグジットを作成する理由

SQL*Forms バージョン 3 では、トリガー内で PL/SQL ブロックを使用できます。したがってほとんどの場合は、ユーザー・イグジットをコールするかわりに PL/SQL のプロシージャ機能を使用できます。ユーザー・イグジットが必要になったときは、USER_EXIT 関数を使用すると PL/SQL ブロックからユーザー・イグジットをコールできます。SQL、PL/SQL あるいは SQL*Forms コマンドに比べて、ユーザー・イグジットは記述方法もインプリメントの方法も複雑です。したがって、ユーザー・イグジットの使用は、SQL、PL/SQL および SQL*Forms の範囲を超える処理の実行に限定するのが一般的です。通常は次のような処理に使用します。

- C などの第 3 世代の言語内で実行すると迅速かつ簡単になる演算（数値積分など）
- リアルタイム・デバイスや処理の制御（たとえば、プリンタまたはグラフィックス・デバイスへの命令の発行）
- 拡張プロシージャ機能が必要なデータ操作（たとえば再帰ソート）
- 特殊なファイル I/O 処理

ユーザー・イグジットの開発

この項では SQL*Forms 3.0 ユーザー・イグジットの開発方法の概要を示します。詳細はこの後の項で説明します。SQL*Forms 4 で使用できる EXEC TOOLS に関する詳細は、20-14 ページの「[EXEC TOOLS 文](#)」を参照してください。ユーザー・イグジットをフォームに取り込むには、次のステップに従います。

- Pro*C でユーザー・イグジットを記述します。
- ソース・コードをプリコンパイルします。
- ステップ 2 で生成された .c ファイルをコンパイルします。
- GENXTB ユーティリティを使用してデータベース表 IAPXTB を作成します。
- SQL*Forms の GENXTB フォームを使用して、ユーザー・イグジット情報を表に挿入します。
- GENXTB ユーティリティを使用して表から情報を読み込み、IAPXIT ソース・コード・モジュールを作成します。次に、ソース・コード・モジュールをコンパイルします。
- 標準 SQL*Forms モジュール、ユーザー・イグジットのオブジェクト、ステップ 6 で作成した IAPXIT オブジェクトをリンクして、新しい SQL*Forms 実行可能プログラムを作成します。
- このフォーム内に、ユーザー・イグジットをコールするためのトリガーを定義します。
- オペレータがフォームを実行する場合、新しい IAP を使用するよう通知してください。標準フォームをこの新しい IAP で置き換えるときは、この通知は必要ありません。

詳細は、各システム専用の Oracle インストレーション・ガイドまたはユーザーズ・ガイドを参照してください。

ユーザー・イグジットの作成

次の種類の文を使用すると、SQL*Forms ユーザー・イグジットを記述できます。

- C コード
- EXEC SQL
- EXEC ORACLE
- EXEC TOOLS

この項では、SQL*Forms とユーザー・イグジットの間での値の引渡しを可能にする EXEC TOOLS 文を中心に説明します。

変数の要件

EXEC TOOLS 文で使用される変数は、フォーム定義で 사용되는フィールド名に対応している必要があります。ブロック名を指定していないためにフィールド参照があいまいな場合は、EXEC IAF のデフォルトはコンテキスト・ブロック（ユーザー・イグジットをコールするブロック）になります。フォーム・フィールドへの参照が無効またはあいまいなときはエラーが発生します。EXEC IAF 文内では、ホスト変数の前にコロン (:) が必要です。

注意：EXEC IAF GET および PUT 文では、標識変数は使用できません。

IAF GET 文

この文により、ユーザー・イグジットがフォームのフィールドから値を取得して、ホスト変数に割り当てることが可能になります。その結果、ユーザー・イグジットでの計算、データ操作、更新などにこのデータを使用できます。GET 文の構文は次のとおりです。

```
EXEC IAF GET field_name1, field_name2, ...  
        INTO :host_variable1, :host_variable2, ...;
```

このとき *field_name* は、次の SQL*Forms 変数のいずれかとなります。

- フィールド
- ブロック・フィールド
- システム変数
- グローバル変数
- フィールド、block.field、システム変数、グローバル変数のいずれかの値を含むホスト変数（先頭コロン付き）

field_name が修飾されていないときは、このフィールドはコンテキスト・ブロック内にある必要があります。

IAF GET の使用方法

ユーザー・イグジットでフィールド値を取得（GET）して、その値をホスト変数に割り当てる方法を次の例に示します。

```
EXEC IAF GET employee.job INTO :new_job;
```

フィールド値はすべて文字列です。可能であれば、GET はフィールド値を対応するホスト変数のデータ型に変換します。不当な変換またはサポートされていないデータ型を変換すると、エラーが発生します。

最後の例では、定数を使用して *block.field* を指定しています。次に示すように、ホスト文字列を使用するとブロック名およびフィールド名も指定できます。

```
char blkfld[20] = "employee.job";
EXEC IAF GET :blkfld INTO :new_job;
```

このフィールドがコンテキスト・ブロック内にないときは、ホスト文字列中に *block.field* の参照全体が含まれる必要があります。このときブロックとフィールドをピリオドでつないでください。たとえば次の指定は無効です。

```
char blk[20] = "employee";
strcpy(fld, "job");
EXEC IAF GET :blk.:fld INTO :new_job;
```

GET 文のフィールド・リストには明示的なフィールド名と変数内に格納されているフィールド名をともに指定できます。ただし単一フィールドの参照では、これらを組み合わせて指定することはできません。たとえば次の指定は無効です。

```
strcpy(fld, "job");
EXEC IAF GET employee.:fld INTO :new_job;
```

IAF PUT 文

この文を使用することにより、ユーザー・イグジットは定数およびホスト変数の値をフォームのフィールドに入れることができます。つまり、SQL*Forms 画面上に任意の値およびメッセージをユーザー・イグジットで表示できます。PUT 文の構文は次のとおりです。

```
EXEC IAF PUT field_name1, field_name2, ...
      VALUES (:host_variable1, :host_variable2, ...);
```

このとき *field_name* は、次の SQL*Forms 変数のいずれかとなります。

- フィールド
- ブロック・フィールド

- システム変数
- グローバル変数
- フィールド、`block.field`、システム変数、グローバル変数のいずれかの値を含むホスト変数（先頭コロン付き）

IAF PUT の使用方法

ユーザー・イグジットで数値定数、文字列定数およびホスト変数をフォームのフィールドに書き出す（PUT）方法を次の例に示します。

```
EXEC IAF PUT employee.number, employee.name, employee.job  
VALUES (7934, 'MILLER', :new_job);
```

GET と同様に、PUT でもホスト文字列を使用してブロック名およびフィールド名を次のように指定できます。

```
char blkfld[20] = "employee.job";  
...  
EXEC IAF PUT :blkfld VALUES (:new_job);
```

文字モード端末のとき、このフィールドが現在表示されているページ内にある場合は、フィールドに PUT される値は割当てが行われたときではなく、ユーザー・イグジットが戻ったときに表示されます。ブロックモード端末のときは、次にデバイスからフィールドを読み込むときにこの値が表示されます。

ユーザー・イグジットでフィールドの値が何度か変更されても、最後に変更された値のみが有効となります。

ユーザー・イグジットのコール

SQL*Forms トリガーからユーザー・イグジットをコールするには、`USER_EXIT`（SQL*Forms が提供する）という名前のパッケージ・プロシージャを使用します。使用する構文は次のとおりです。

```
USER_EXIT(user_exit_string [, error_string]);
```

ここで、`user_exit_string` にはユーザー・イグジットの名前とオプションのパラメータを指定して、`error_string` にはユーザー・イグジットが異常終了したときに SQL*Forms により発行されるエラー・メッセージを指定します。たとえば次のトリガー・コマンドは、`LOOKUP` という名前のユーザー・イグジットをコールします。

```
USER_EXIT('LOOKUP');
```

ユーザー・イグジット文字列は引用符（二重引用符は不可）で囲んでください。

ユーザー・イグジットへのパラメータの引渡し

ユーザー・イグジットをコールすると、SQL*Forms は自動的に次のパラメータをユーザー・イグジットに渡します。

コマンドライン	ユーザー・イグジット文字列。
コマンドラインの長さ	ユーザー・イグジット文字列の長さ（文字数）。
エラー・メッセージ	定義済の場合、エラー文字列（障害メッセージ）。
エラー・メッセージの長さ	エラー文字列の長さ。
問合せモード	ブール値。ユーザー・イグジットのコールが通常モードと問合せモードのどちらで行われたかを示します。

しかしユーザー・イグジット文字列を使用すると、追加パラメータをユーザー・イグジットに渡せません。たとえば次のトリガー・コマンドを使用すると、2つのパラメータと1つのエラー・メッセージがユーザー・イグジット LOOKUP に渡されます。

ユーザー・イグジット文字列は引用符（二重引用符は不可）で囲んでください。

```
USER_EXIT('LOOKUP 2025 A', 'Lookup failed');
```

次の例に示すように、この機能を使用してフィールド名をユーザー・イグジットに渡せます。

```
USER_EXIT('CONCAT firstname, lastname, address');
```

ただしユーザー・イグジット文字列の解析は、SQL*Forms ではなくユーザー・イグジットによって実行されます。

フォームへの値のリターン

ユーザー・イグジットでは SQL*Forms に制御が戻るときに必ずコードが戻ります。このコードはユーザー・イグジットが成功したか、失敗したか、致命的エラーが発生したかどうかを示します。このリターン・コードは SQL*Forms によって定義される integer の整数定数です（次の項を参照）。この 3 種類の結果は次の意味を持ちます。

成功	ユーザー・イグジットでエラーが発生しませんでした。コール元のトリガー・ステップで逆戻りコード・スイッチが設定されていなければ、SQL*Forms は成功ラベルまたは次のステップに進みます。
失敗	ユーザー・イグジットで、フィールド内の無効値などのエラーが検出されました。このイグジットによって渡された任意指定のメッセージが、SQL*Forms 画面下部のメッセージ行およびエラー表示画面に表示されます。SQL*Forms は行に影響を与えない SQL 文に対するときと同様に応答します。
致命的エラー	ユーザー・イグジットで、SQL 文中の実行エラーなど、それ以上処理を続行できない条件が検出されました。このイグジットによって渡された任意指定のエラー・メッセージが SQL*Forms エラー表示画面に表示されます。SQL*Forms は SQL 文内の致命的エラーに対するときと同様に応答します。ユーザー・イグジットでフィールドの値が変更された後で失敗または致命的エラーコードが戻ったときは、SQL*Forms はこの変更を破棄しません。また、逆戻りコード・スイッチが設定されているときに成功コードが戻されたときにも、SQL*Forms は変更を破棄しません。

IAP 定数

リターン・コードとして使用する 3 つの記号定数が SQL*Forms によって定義されます。ホスト言語に応じて、これらの定数には文字 IAP または SQL という接頭辞が付きます。たとえば IAPSUC、IAPFAIL、IAPFTL などと定義されます。

SQLIEM 関数の使用方法

関数 SQLIEM をコールすることにより、SQL*Forms が表示するエラー・メッセージをユーザー・イグジットから指定できます。このエラー・メッセージは、トリガー・ステップでエラーが発生したときはメッセージ行に、このステップで致命的エラーが発生したときはエラー表示画面に表示されます。ここで指定したメッセージが、このステップに対して定義されていた任意のメッセージと置換されます。SQLIEM ファンクション・コールの構文は次のとおりです。

```
sqliem (char *error_message, int message_length);
```


error_message は文字変数、*message_length* は整変数です。Pro*C/C++ プリコンパイラによって適切な外部関数宣言が生成されます。この2つのパラメータは参照によって渡します（つまり、値ではなくアドレスを渡します）。SQLIEM は SQL*Forms の関数です。したがって SQL*ReportWriter など別の Oracle Tools の製品からはコールできません。

WHENEVER の使用方法

イグジット内で WHENEVER 文を指定すると、不当なデータ型の変換（SQLERROR）、フォーム・フィールドに PUT された値の切捨て（SQLWARNING）、および行を戻さない問合せ（NOT FOUND）を検出できます。

例

次の例では、*sqliem* 関数、EXEC IAF GET ルーチンおよび EXEC IAF PUT ルーチンを使用するユーザー・イグジットの記述方法を示します。

```
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 IAF GET :field1, :field2 INTO :value1, :value2;
    /* manipulate the values to obtain result_val */
    ...
    /* put result_val into form field result */
    EXEC IAF PUT result VALUES (:result_val);
    return IAPSUC; /* trigger step succeeded */

sql_error:
    strcpy(errmsg, CONCAT("MYEXIT", sqlca.sqlerrm.sqlerrmc));
    errlen = strlen(errmsg);
    sqliem(errmsg, &errlen); /* send error msg to Forms */
    return IAPFAIL;
```

ユーザー・イグジットのプリコンパイルおよびコンパイル

ユーザー・イグジットはスタンドアロン型のホスト・プログラムと同じ方法でプリコンパイルされます。第 10 章「プリコンパイラのオプション」を参照してください。ユーザー・イグジットのコンパイル方法については、各システム専用の Oracle インストレーション・ガイドまたはユーザーズ・ガイドを参照してください。

サンプル・プログラム：ユーザー・イグジット

次の例にユーザー・イグジットを示します。

```
/******  
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 */
int *query;     /* TRUE if invoked by post-query trigger,
                 FALSE otherwise */
{
    char *cp = cmd + 7;      /* pointer to field list in
                             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] = '\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 = '\0';
                field.len = strlen((char *) field.arr);
                /* Get field value from form. */
                EXEC IAF 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 = '\0';
}

```

```
        field.len = strlen((char *) field.arr);
        result.len = strlen((char *) result.arr);

/* Put result into form. */
    EXEC IAF PUT :field VALUES (:result);

/* Trigger step succeeded. */
    return(IAPSUC);

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. */
    sqliem(errmsg, &errlen);
    return(IAPFAIL); /* Trigger step failed. */
}
```

GENXTB ユーティリティの使用方法

IAPXIT モジュール内の IAP プログラム表 IAPXTB には、IAP 内にリンクされているそれぞれのユーザー・イグジット用のエントリが格納されています。IAPXTB は IAP に各ユーザー・イグジットの名前、位置およびホスト言語を指示します。新しいユーザー・イグジットを IAP に追加するときは、対応するエントリを IAPXTB に追加する必要があります。IAPXTB は、IAPXTB という同じ名前のデータベース表から導出されます。次に示すように、オペレーティング・システムのコマンドラインで GENXTB フォームを実行することによって、データベースの表を修正できます。

RUNFORM GENXTB username/password

定義するそれぞれのユーザー・イグジットについて次の情報を入力できるフォームが表示されます。

- イグジット名 (20-13 ページの「[ガイドライン](#)」を参照してください)
- C 言語コード
- 作成日
- 最終変更日
- コメント

IAPXTB データベース表を変更してから、GENXTB ユーティリティを使用してその表を読み込み、IAPXIT モジュールとそれに含まれる IAPXTB プログラム表を定義するアセンブラまたは C のソース・プログラムを作成します。使用するソース言語は、オペレーティング・システムによって異なります。GENXTB ユーティリティの構文は次のとおりです。

```
GENXTB username/password outfile
```

このとき *outfile* は、GENXTB が作成するアセンブラまたは C のソース・プログラムに指定する名前です。

ユーザー・イグジットの SQL*Forms へのリンク

ユーザー・イグジットをコールするフォームの実行前に、フォームを実行する SQL*Forms のコンポーネントである IAP にこのユーザー・イグジットをリンクする必要があります。ユーザー・イグジットは標準的なバージョンの IAP にリンクしたり、そのイグジットをコールするフォーム用の特別なバージョンの IAP にリンクできます。

IAP の実行可能コピーを新規に作成するには、Oracle ライブラリおよび C リンク・ライブラリからユーザー・イグジット・オブジェクト・モジュール、標準 IAP モジュール、IAPXIT モジュール、その他必要なモジュールをすべてリンクします。

リンク方法はシステムによって異なります。各システム専用の Oracle インストレーション・ガイドまたはユーザーズ・ガイドで確認してください。

ガイドライン

この項では、一般的な問題を回避するためのガイドラインを示します。

イグジットの命名

ユーザー・イグジットの名前を Oracle の予約語にしないでください。また、SQL*Forms コマンドの名前、関数コードの名前、SQL*Forms に使用される外部定義済の名前と競合を起こす名前は使用しないでください。ソース・コード内のユーザー・イグジットのエン트리・ポイントの名前はユーザー・イグジット自体の名前となります。このイグジット名は有効な C 関数名であり、同時に使用しているオペレーティング・システムの規則に従った有効なファイル名である必要があります。

SQL*Forms は検索前にそのユーザー・イグジットの名前を大文字に変換します。したがって、イグジット名はソース・コード内では大文字になっている必要があります。

Oracle への接続

ユーザー・イグジットでは SQL*Forms が確立した接続によって Oracle と通信します。ただし、ユーザー・イグジットで SQL*Net を使用して任意のデータベースに追加の接続を確立できます。詳細は、3-5 ページの「[アドバンスト・コネクション・オプション](#)」の項を参照してください。

I/O コールの発行

ファイル I/O はサポートされていますが、画面 I/O はサポートされていません。

ホスト変数の使用方法

スタンドアロン型のプログラムに適用されるホスト変数の制限事項はユーザー・イグジットにも適用されます。EXEC SQL および EXEC IAF 文内では、ホスト変数の前にコロン (:) が必要です。ただし EXEC IAF 文では、ホスト配列は使用できません。

表の更新

一般に、ユーザー・イグジットではフォームに関連付けられたデータベースの表を UPDATE しないでください。たとえば、SQL*Forms 作業領域でオペレータがレコードを更新した後で、関連付けられたデータベース表内の対応する行を UPDATE したとします。このときトランザクションが COMMIT されると、SQL*Forms 作業領域内のレコードがその表に適用され、ユーザー・イグジットの UPDATE は上書きされます。

コマンドの発行

Oracle は、ユーザー・イグジットで実行された処理に限らず、SQL*Forms のオペレータが開始した作業もコミットまたはロールバックするため、ユーザー・イグジットからは COMMIT または ROLLBACK コマンドを発行しないでください。かわりに SQL*Forms トリガーから COMMIT または ROLLBACK コマンドを発行してください。データ定義コマンド (ALTER、CREATE、GRANT など) についても同様です。それらのコマンドも実行の前後に暗黙的に COMMIT を発行するためです。

EXEC TOOLS 文

EXEC TOOLS 文は、ユーザー・イグジットからの読み込み、設定および例外コールバックを処理する包括的な方法を提供することによって、基本的な Oracle Toolset (Oracle Forms バージョン 4、Oracle Report バージョン 2 および Oracle Graphics バージョン 2) をサポートします。次の説明は Oracle Forms が中心になっていますが、Oracle Report および Oracle Graphics についても概念は同じです。

Toolset ユーザー・イグジットの作成

EXEC SQL、EXEC ORACLE およびホスト言語文の他にも、次の EXEC TOOLS 文を使用して Oracle Forms ユーザー・イグジットを記述できます。

- SET
- GET
- SET CONTEXT

- GET CONTEXT
- MESSAGE

EXEC TOOLS GET 文および EXEC TOOLS SET 文は、Oracle Forms の以前のバージョンで使用されていた EXEC IAF GET 文および EXEC IAF PUT 文に相当します。ただし IAF GET および IAF PUT とは異なり、TOOLS GET および TOOLS SET は標識変数を受け付けます。EXEC TOOLS MESSAGE 文は、メッセージ処理関数 *sqliem* に相当します。次に、すべての EXEC TOOLS 文について簡単に説明します。詳細は、『Oracle Forms リファレンス・マニュアル』を参照してください。

EXEC TOOLS SET

EXEC TOOLS SET 文はユーザー・イグジットから Oracle Forms に値を渡します。この文は、特にホスト変数および定数の値を Oracle Forms 変数および項目に割り当てます。フォーム項目に渡された値は、ユーザー・イグジットで制御がフォームに戻った後に表示されます。EXEC TOOLS SET 文を記述するには、次の構文を使用します。

```
EXEC TOOLS SET form_variable[, ...]  
VALUES ({:host_variable :indicator | constant}[, ...]);
```

このとき *form_variable* は、Oracle Forms フィールド、*block.field*、システム変数、グローバル変数、またはこれらの項目のうち 1 つの値を含むホスト変数（先頭コロン付き）です。次の例では、ユーザー・イグジットによって従業員名を Oracle Forms に渡します。

```
char ename[20];  
short ename_ind;  
  
...  
  
strcpy(ename, "MILLER");  
ename_ind = 0;  
EXEC TOOLS SET emp.ename VALUES (:ename :ename_ind);
```

ここで *emp.ename* は Oracle Forms の *block.field* の 1 つです。

EXEC TOOLS GET

EXEC TOOLS GET 文は Oracle Forms からユーザー・イグジットに値を渡します。この文は、特に Oracle Forms 変数および項目の値をホスト変数に割り当てます。値が渡されるとすぐに、ユーザー・イグジットでそれらの値を任意の目的に使用できます。EXEC TOOLS GET 文を記述するときは次の構文を使用してください。

```
EXEC TOOLS GET form_variable[, ...]  
INTO :host_variable:indicator[, ...];
```

このとき *form_variable* は、Oracle Forms フィールド、`block.field`、システム変数、グローバル変数、またはこれらの項目のうち 1 つの値を含むホスト変数（先頭コロン付き）です。次の例では、Oracle Forms はブロックからユーザー・イグジットに項目名を渡します。

```
...
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;
```

EXEC TOOLS SET CONTEXT

EXEC TOOLS SET CONTEXT 文は、後で別のユーザー・イグジットで使用するためにユーザー・イグジットからのコンテキスト情報を保存します。ポインタ変数はコンテキスト情報が格納されているメモリーのブロックを指します。SET CONTEXT 文と併用するときは、情報を保持するためのグローバル変数を宣言する必要はありません。EXEC TOOLS SET CONTEXT 文を記述するには、次の構文を使用します。

```
EXEC TOOLS SET CONTEXT :host_pointer_variable
    IDENTIFIED BY context_name;
```

このとき *context_name* は未宣言の識別子またはコンテキスト領域を命名する文字ホスト変数（先頭コロン付き）です。

```
...
char *context_ptr;
char context[20];

strcpy(context, "context1")
EXEC TOOLS SET CONTEXT :context IDENTIFIED BY application1;
```

EXEC TOOLS GET CONTEXT

EXEC TOOLS GET CONTEXT 文は、（SET CONTEXT によって保存されている）コンテキスト情報をユーザー・イグジットに取り出します。ホスト言語ポインタ変数は、コンテキスト情報が格納されているメモリーのブロックを指します。EXEC TOOLS GET CONTEXT 文を記述するには、次の構文を使用します。

```
EXEC TOOLS GET CONTEXT context_name
    INTO :host_pointer_variable;
```

このとき *context_name* は未宣言の識別子またはコンテキスト領域を命名する文字ホスト変数（先頭コロン付き）です。次の例では、ユーザー・イグジットは前に保管されたコンテキスト情報を取り出します。


```
...  
char *context_ptr;  
  
EXEC TOOLS GET CONTEXT application1 INTO :context_ptr;
```

EXEC TOOLS MESSAGE

EXEC TOOLS MESSAGE 文は、ユーザー・イグジットから Oracle Forms にメッセージを渡します。ユーザー・イグジットによりフォームに制御が戻った後にメッセージが Oracle Forms のメッセージ行に表示されます。EXEC TOOLS MESSAGE 文を記述するには、次の構文を使用します。

```
EXEC TOOLS MESSAGE message_text [severity_code];
```

ここで *message_text* は、引用文字列または文字ホスト変数（先頭コロン付き）で、オプションの *severity_code* は整数定数または整数ホスト変数（先頭コロン付き）です。MESSAGE 文に標識変数は指定できません。次の例では、ユーザー・イグジットからエラー・メッセージが Oracle Forms に渡されます。

```
EXEC TOOLS MESSAGE 'Bad field name! Please reenter.';
```


この付録では、Pro*C/C++ プリコンパイラのリリース 8.0 に備わっている新機能を簡単に説明しています。詳細は該当する章を参照してください。

この章では、次の事項について説明します。

- 構造体の配列
- プリコンパイル済ヘッダー・ファイル
- CALL 文
- 実行時のパスワードの変更
- 各国語キャラクタ・セットのサポート
- CHAR_MAP プリコンパイラ・オプション
- SQLLIB 関数の新規名
- WHENEVER 文の新規アクション
- オブジェクト型のサポート
- オブジェクト型トランスレータ
- LOB サポート
- ANSI 動的 SQL
- コレクション
- その他のトピック
- 以前のリリースからの移行

構造体の配列

Pro*C/C++ では、構造体の配列の使用をサポートしています。これにより、複数行、複数列を操作できます。この拡張要素は、ユーザー・データの処理をより簡単にするために、Pro*C/C++ がスカラーの構造体の単純な配列を埋込み SQL 文でバインド変数として処理できるようにしています。これで、プログラミングがさらに直観的になり、データ編成もはるかに自由にできます。

Pro*C/C++ では、構造体の配列をバインド変数として使用できるのみでなく、標識構造体の配列も構造体の配列の宣言で 사용할 ことができます。8-16 ページの「[構造体配列](#)」を参照してください。

プリコンパイル済ヘッダー・ファイル

プリコンパイラ・オプション HEADER は、プリコンパイル済ヘッダー・ファイルを作成し、大規模なプロジェクトの開発に要する時間およびコンピュータ・リソースを削減するのに使用できます。5-34 ページの「[プリコンパイル済のヘッダー・ファイル](#)」を参照してください。

CALL 文

CALL 埋込み SQL 文はストアド・プロシージャを呼び出します。新しいアプリケーションの埋込み PL/SQL ブロックのかわりに使用することもできます。F-17 ページの「[CALL \(実行可能埋込み SQL\)](#)」を参照してください。

実行時のパスワードの変更

Pro*C/C++ のクライアント・アプリケーションでは、EXEC SQL CONNECT 文を拡張し、実行時にユーザーのパスワードを変更できます。3-3 ページの「[ALTER AUTHORIZATION 句を使用したパスワードの変更](#)」を参照してください。

各国語キャラクタ・セットのサポート

Pro*C/C++ では、NLS_LOCAL=NO の場合にマルチバイト・キャラクタ・セット (NCHAR 列、NVARCHAR2 列、NCLOB 列) をサポートしています。NLS_LOCAL=NO の場合に新しい環境変数 NLS_NCHAR が有効な各国語キャラクタ・セットに設定されると、Oracle8i データベース・サーバーで NCHAR がサポートされます。4-48 ページの「[環境変数 NLS_NCHAR](#)」を参照してください。

CHARACTER SET [IS] NCHAR_CS 句は、文字変数宣言で指定できます。この結果は、NLS_CHAR プリコンパイラ・オプションで変数を命名した場合と同じです。4-48 ページの「[CHARACTER SET \[IS\] NCHAR_CS](#)」を参照してください。

新しい句 CONVBUFSZ を EXEC SQL VAR 文で使用する と、キャラクタ・セットを変換できます。5-14 ページの「[EXEC SQL VAR と TYPE 宣言文の利用](#)」を参照してください。

CHAR_MAP プリコンパイラ・オプション

このオプションで、C ホスト文字変数のデフォルトのマッピングを指定します。Oracle8i のデフォルト設定では、文字列は CHARZ（固定長の空白埋めおよび 0 終了）になっています。詳細は、5-2 ページの「[プリコンパイラ・オプション CHAR_MAP](#)」を参照してください。

SQLLIB 関数の新規名

今回の Pro*C/C++ リリースでは、各 SQLLIB 関数に新しい別名が付いていますが、その旧名もそのまま残っています。5-49 ページの「[SQLLIB パブリック関数の新しい名前](#)」を参照してください。

WHENEVER 文の新規アクション

埋込み SQL 宣言文 WHENEVER では、DO BREAK および DO CONTINUE アクションをサポートしています。9-24 ページの「[WHENEVER 宣言文の使用](#)」および F-114 ページの「[WHENEVER（埋込み SQL 宣言文）](#)」を参照してください。

オブジェクト型のサポート

Pro*C/C++ では、データベース・サーバーに定義したオブジェクト型に C の構造体をマップできます。

結合インタフェースとナビゲーション・インタフェース（実行可能埋込み SQL 拡張要素）を使用して Pro*C/C++ プログラムのオブジェクトにアクセスする方法の詳細は、[第 17 章「オブジェクト」](#)を参照してください。

オブジェクトへのアクセス方法を示すサンプル・プログラムは、17-24 ページの「[ナビゲーション・アクセスのサンプル・コード](#)」を参照してください。

オブジェクト型トランスレータ

オブジェクト型トランスレータ（OTT）のユーティリティについては、新しい章で詳しく説明しています。このユーティリティにより、データベースのオブジェクト型を C の構造体にマップして、OCI アプリケーションや Pro*C/C++ アプリケーションで使用できます。OTT は、プリコンパイラよりも先に実行します。[第 19 章「オブジェクト型トランスレータ」](#)を参照してください。

OCI 関数コールと埋込み SQL 文を組み合わせ、アプリケーションで使用することができます。OCIString および OCINumber データ型を操作するライブラリ・ルーチンのみでなく、新しい OCI 操作相互性機能を使用することができます。Pro*C/C++ のオブジェクト型サポートの詳細は、[第 17 章「オブジェクト」](#)を参照してください。

LOB サポート

埋込み SQL 文インタフェースを使用すると、LOB（ラージ・オブジェクト）をプリコンパイラ・アプリケーションで 사용할 수 있습니다。LOB の使用方法、内部 LOB および外部 LOB、LOB を処理する他の方法との比較が示されます。新しい SQL 文の一つ一つが紹介されます。LOB インタフェースの使用 방법은、サンプル・コードによって示されています。完全な詳細は、[第 16 章「ラージ・オブジェクト \(LOB\)」](#)を参照してください。

ANSI 動的 SQL

埋込み SQL 文を使用した動的 SQL 方法 4 の完全な ANSI インプリメンテーションについては、[第 14 章「ANSI 動的 SQL」](#)で説明しています。まず簡単な例を使用して概要が説明されます。その後新しい SQL 文の完全な説明が続きます。それから、demo ディレクトリのサンプル・プログラムが示されます。

コレクション

2 種類のコレクション（VARRAY および NESTED TABLE）が紹介され、他のデータ型と比較されます。それから、コレクションを操作する埋込み SQL コマンドについて説明します。[第 18 章「コレクション」](#)を参照してください。

その他のトピック

Unicode サポート

バインド変数および定義変数の Unicode (UCS2) キャラクタ・セットのサポートは、5-9 ページの「[Unicode 変数](#)」で説明しています。

PREFETCH オプション

このプリコンパイラ・オプションを使用すると、値を「プリフェッチ」するためデータベース・アクセスが速くなり、結果としてネットワークへの往復回数が少なくなります。6-15 ページの「[PREFETCH プリコンパイラ・オプション](#)」を参照してください。

外部プロシージャ

C で作成された外部プロシージャは、PL/SQL ブロックからコールできます。REGISTER CONNECT 埋込み SQL 文はプロシージャで使用します。7-27 ページの「[外部プロシージャ](#)」を参照してください。

PL/SQL からの Java のコール

Java で作成されたストアド・プロシージャはアプリケーションからコールできます。Java で作成されたプロシージャのコール方法は、7-19 ページの「[ストアド PL/SQL および Java サブプログラム](#)」を参照してください。

DML 戻り句

この句は INSERT、DELETE および UPDATE 文で使用できます。6-10 ページの「[DML 戻り句](#)」を参照してください。

汎用 ROWID

汎用 ROWID データ型のサポートが提供されています。索引構成表はこの概念に基づいて作成されています。4-36 ページの「[汎用 ROWID](#)」を参照してください。

CONNECT 文の SYSDBA/SYSOPER 権限

CONNECT 文の使用権限を設定する方法は、3-5 ページの「[データベースへの接続](#)」を参照してください。

CLOSE_ON_COMMIT プリコンパイラ・オプション

CLOSE_ON_COMMIT マイクロ・プリコンパイラ・オプションを使用すると、マクロ・オプション MODE=ANSI で COMMIT が実行されるときにすべてのカーソルをクローズするかどうか選択できます。6-14 ページの「[CLOSE_ON_COMMIT プリコンパイラ・オプション](#)」および 10-12 ページの「[CLOSE_ON_COMMIT](#)」を参照してください。

文字列

アプリケーションの多くは、文字列が可変長（たとえば VARCHAR2 など）であるという前提で作成されています。デフォルトの Oracle8i では固定長、空白埋め、NULL 終了文字列（CHARZ）を使用し、カレント SQL 標準に準拠しています。

アプリケーションの文字列の長さが変わることを見込んでいる場合（文字列の比較方法で特に重要）、オプション DBMS=V8 と CHAR_MAP=VARCHAR2 を指定してアプリケーションをプリコンパイルする必要があります。詳細は、5-2 ページの「[文字データの処理](#)」を参照してください。

DBMS オプションの効果の完全なリストは 10-15 ページの「[DBMS](#)」の DBMS オプションの説明を参照してください。

エラー・メッセージ・コード

以前の Pro*C/C++ リリースとカレント・リリースでは、エラーおよび警告コードが異なっています。コードとメッセージの完全なリストは、『Oracle8i エラー・メッセージ』を参照してください。

SQLLIB によって発行されるランタイム・メッセージは、以前の Pro*C/C++ リリースおよび Pro*C のリリースでは RTL- という接頭辞が付いていましたが、これが SQL- という接頭辞に変更されました。メッセージ・コードは以前のリリースと同じです。

SQLCHECK=SEMANTICS を指定してプリコンパイルする場合、PL/SQL コンパイラでは接頭辞として PLS が使用されます。このようなエラーは、Pro*C/C++ によるものではありません。

LINES オプション

LINES=YES の場合、C コンパイラによって発行されたエラー・メッセージは常に、修正済（プリコンパイル済）のソース・ファイルではなく、元のソース・ファイルを参照します。これにより、大半のデバッガを使用した元のソース・コードを参照できます。10-26 ページの「[LINES](#)」を参照してください。

以前のリリースからの移行

Pro*C/C++ で作成された既存のアプリケーションは、Oracle8i Server でも変わりなく動作します。

詳細は、『Oracle8i 移行ガイド』を参照してください。

予約語、キーワードおよび名前領域

この章では、次の事項について説明します。

- [予約語およびキーワード](#)
- [Oracle の予約名前領域](#)

予約語およびキーワード

一部の語は Oracle により予約されています。つまり、Oracle で特別な意味を持っている語なので、改めて定義することはできません。このため、列、表、索引などのデータベース・オブジェクトの名前には使用できません。SQL および PL/SQL の Oracle 予約語のリストは、『Oracle8i SQL リファレンス』および『Oracle8i PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

Pro*C/C++ キーワードは、C または C++ キーワードと同様、プログラムの中で変数として使用しないでください。使用すると、エラーが発生します。列などのデータベース・オブジェクトの名前に使用される場合、結果としてエラーが返されます。Pro*C/C++ で使用されるキーワードは次のとおりです。

all	allocate	alter	analyze	and
any	arraylen	as	asc	at
audit	authorization	avg	begin	between
bind	both	break	by	cache
call	cast	char	character	character
charf	charz	check	close	collection
comment	commit	connect	constraint	constraints
context	continue	convbufsz	count	create
current	currval	cursor	database	date
dateformat	datelang	day	deallocate	dec
decimal	declare	default	define	delete
deref	desc	describe	descriptor	display
distinct	do	double	drop	else
enable	end	endif	escape	exec
exec	execute	exists	explain	extract
fetch	float	flush	for	force
found	free	from	function	get
global	go	goto	grant	group
having	hour	iaf	identified	ifdef
ifndef	immediate	in	indicator	input
insert	integer	intersect	interval	into
is	is	leading	level	like

list	lob	local	lock	long
max	message	min	minus	minute
mode	month	multiset	nchar	nchar_cs
next	nextval	noaudit	not	notfound
nowait	null	number	numeric	nvarchar
nvarchar2	object	ocifilelocator	ocibloblocator	ocicloblocator
ocidate	ociextprocontext	ocinumber	ociraw	ocirowid
ocistring	of	only	open	option
option	or	oracle	order	output
overlaps	package	partition	precision	prepare
prior	procedure	put	raw	read
real	ref	reference	register	release
rename	replace	return	returning	revoke
role	rollback	rowid	rownum	savepoint
second	section	select	set	set
smallint	some	sql	sql_context	sql_cursor
sqlerror	sqlwarning	start	statement	stddev
stop	string	sum	sysdate	sysdba
sysoper	table	temporary	the	threads
time	timestamp	timezone_hour	timezone_minute	to
tools	trailing	transaction	trigger	trim
truncate	type	uid	ulong_varchar	union
unique	unsigned	update	use	user
using	uvarchar	validate	values	varchar
varchar	varchar2	variables	variance	varnum
varraw	view	whenever	where	with
work	year	zone		

Oracle の予約名前領域

Oracle によって予約されている名前領域のリストを次の表に示します。Oracle ライブラリにある関数名の先頭は、このリストの文字列に限られています。名前が競合する可能性があるため、名前がこれらの文字で始まる関数を使用しないでください。たとえば、Net8 透過ネットワーク・サービス関数はすべて NS で始まるため、関数に NS で始まる名前を付けないように注意する必要があります。

表 B-1 Oracle の予約名前領域

名前領域	ライブラリ
XA	XA アプリケーション専用の外部関数
SQ	Oracle プリコンパイラおよび SQL*Module アプリケーションによって使用される外部 SQLLIB 関数
O、OCI	外部 OCI 関数内部 OCI 関数
UPI、KP	Oracle UPI レイヤーからの関数名
NA	Net8 固有サービス製品
NC	Net8 RPC 製品
ND	Net8 ディレクトリ
NL	Net8 ネットワーク・ライブラリ・レイヤー
NM	Net8 ネット管理プロジェクト
NR	Net8 交換
NS	Net8 透過ネットワーク・サービス
NT	Net8 ドライバ
NZ	Net8 セキュリティ・サービス
OSN	Net8 V1
TTC	Net8 2 タスク
GEN、L、ORA	コア・ライブラリ関数
LI、LM、LX	Oracle NLS レイヤーからの関数名
S	システム依存ライブラリからの関数名

表のリストは Oracle 予約名前領域のすべての関数を包括的に示したものではありません。特定の名前領域での関数の完全なリストは、該当する Oracle ライブラリに対応するマニュアルを参照してください。

パフォーマンスの最適化

この付録では、アプリケーションのパフォーマンスを改善させる手軽な方法をいくつか紹介します。これらの方法を使用すると、多くの場合、処理時間を 25% 以上削減できます。

この章では、次の事項について説明します。

- パフォーマンスを低下させる原因
- パフォーマンスの改善方法
- ホスト配列の使用
- 埋込み PL/SQL の利用
- SQL 文の最適化
- 索引の使用
- 行レベル・ロックの利用
- 不要な解析の排除

パフォーマンスを低下させる原因

パフォーマンスを低下させる原因の 1 つは通信オーバーヘッドが高いことです。Oracle8i では、一度に 1 つの SQL 文を処理する必要があります。つまり、各文によって Oracle8i への別のコールが発生し、オーバーヘッドが増加します。ネットワーク化された環境下では、ネットワークを介して SQL 文を送信する必要があるため、ネットワーク通信量が増加することになります。ネットワーク通信量が多いとアプリケーションの処理速度は著しく低下します。

パフォーマンスを低下させるもう 1 つの原因は非効率的な SQL 文です。SQL はたいへん柔軟性に富むため、2 つの異なる文から同一の結果を得ることもできますが、効率に差がある場合もあります。たとえば、次の 2 つの SELECT 文は同じ行（従業員が最低 1 人いる部門ごとの名称および番号）を戻します。

```
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);
```

ただし、この最初の文は DEPT 表内のすべての部門番号を探して EMP 表全体をスキャンするため、処理に時間がかかります。EMP 表内の DEPTNO 列に索引を付けていても、この副問合せには DEPTNO を指定する WHERE 句がないので、索引は使用されません。

パフォーマンスを低下させる 3 番目の原因は、不要な解析とバインディングです。SQL 文を実行する前に、Oracle8i でこの SQL 文を解析しバインドする必要があることに注意してください。解析とは、SQL 文を調べて、これが構文規則に従って正しいデータベース・オブジェクトを参照していることを確認する作業です。バインディングとは、SQL 文内のホスト変数をそれぞれのアドレスに対応付け、Oracle8i でその値に対して読み込みまたは書き込みができるようにする作業です。

大部分のアプリケーションは、十分にカーソルを管理しているわけではありません。このため不要な解析またはバインドが発生し、結果的に処理のオーバーヘッドが著しく増加します。

パフォーマンスの改善方法

プリコンパイルしたプログラムのパフォーマンスがよくない場合でも、オーバーヘッドを減少させる方法があります。

特にネットワーク化された環境下では、次の処理によって通信オーバーヘッドを大幅に削減できます。

- ホスト配列の使用
- 埋込み PL/SQL の使用

処理のオーバーヘッドは次の方法で大幅に削減できる場合があります。

- SQL 文の最適化
- 索引の使用
- 行レベル・ロックの利用
- 不要な解析の排除

以降の項では、オーバーヘッドを削減するための方法を検討します。

ホスト配列の使用

ホスト配列を使用すると、1 つの SQL 文でデータの集まり全体を操作できるため、パフォーマンスが向上します。たとえば、300 人の従業員の給料を EMP 表に INSERT する場合を考えます。配列がないと、プログラムは 300 の個々の INSERT（各従業員に 1 つ）を実行する必要があります。配列を使用すると、必要な INSERT は 1 回のみになります。次の文について考えてみましょう。

```
EXEC SQL INSERT INTO emp (sal) VALUES (:salary);
```

salary が通常の変数の場合は、Oracle8i でこの INSERT 文を 1 回実行すると、EMP 表には 1 行のみが挿入されます。この行の SAL 列には *salary* の値が格納されます。この方法で 300 行を挿入するには、この INSERT 文を 300 回実行する必要があります。

しかし、*salary* がサイズ 300 のホスト配列の場合は、Oracle8i では一度に 300 行すべてが EMP 表に挿入されます。各行の SAL 列には *salary* 配列の要素の値が格納されます。

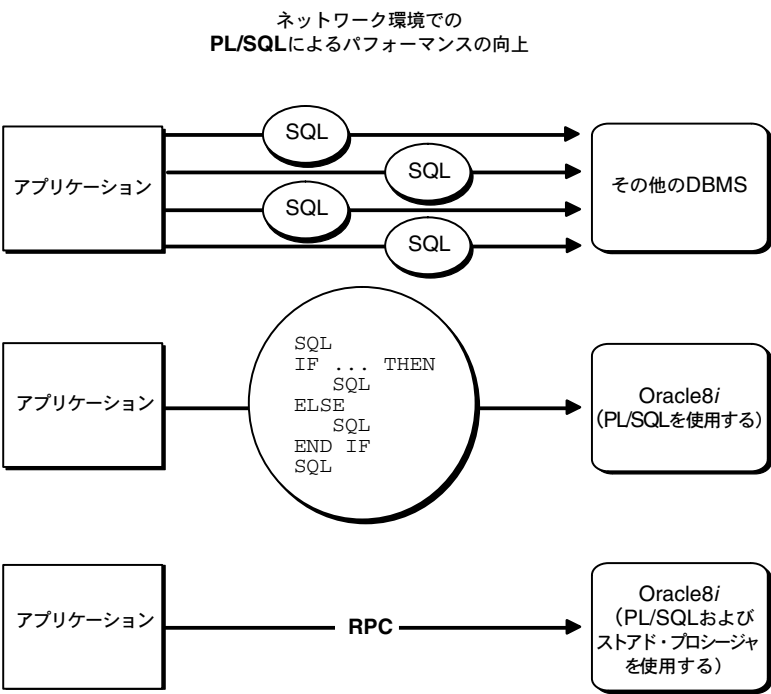
詳細は、第 8 章「ホスト配列」を参照してください。

埋込み PL/SQL の利用

図 C-1 に示されているように、アプリケーションがデータベース集約型であれば、制御構造体を使用して PL/SQL ブロック内で SQL 文をグループ化し、ブロック全体をデータベース・サーバーに送ることができます。これによってアプリケーションとデータベース・サーバーとの間の通信量は大幅に減少します。

また、PL/SQL サブプログラムを使用してアプリケーションから O へのコールを少なくすることもできます。たとえば、個別の SQL 文を実行するには 10 回のコールが必要ですが、10 個の SQL 文を含んでいるサブプログラムを実行するには、1 回のコールで済みます。

図 C-1 PL/SQL によるパフォーマンスの向上



PL/SQL は、SQL*Forms、SQL*Menu、SQL*ReportWriter などの Oracle アプリケーション開発ツールでも使用できます。PL/SQL によって Tools にプロシージャ型の処理能力が加えられるため、パフォーマンスが向上します。PL/SQL を使用すると、Tools ではデータベース・サーバーをコールすることなくすべての計算を迅速かつ効率的に処理できます。この結果、時間が節約され、ネットワーク通信量が減少します。

詳細は、[第 7 章「埋込み PL/SQL」](#) および『Oracle8i PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

SQL 文の最適化

Oracle8i オプティマイザにより、すべての SQL 文について実行計画が生成されます。実行計画とは、Oracle8i でその SQL 文を実行するための一連の手順です。これらの手順は、『Oracle8i アプリケーション開発者ガイド 基礎編』に記載されたルールによって決まります。これらのルールに従うと、最適な SQL 文を作成できます。

オプティマイザ・ヒント

場合によっては、Oracle8i に対して SQL 文を最適化する方法を示すことができます。このようにして示す内容はヒントと呼ばれ、これによりオプティマイザによる決定に運用側から影響を与えることができます。

ヒントは宣言文ではありません。オプティマイザがジョブを実行するのを助けるためのものにすぎません。ヒントの中には、SQL 文を最適化するのに使用される情報の有効範囲を制限するものもあり、また総体的な方針を提示するものもあります。

ヒントを使用して、次の事項を指定できます。

- SQL 文のための最適化アプローチ
- 参照されているそれぞれの表へのアクセス・パス
- 結合のための結合順序
- 表を結合するための方法

つまり、ヒントは次の 4 つのカテゴリに分けられます。

- 最適化アプローチ
- アクセス・パス
- 結合順序
- 結合操作

たとえば、2 つの最適化アプローチ・ヒントである COST と NOCOST は、コストベースのオプティマイザとルールベースのオプティマイザをそれぞれ起動します。

SELECT、UPDATE、INSERT あるいは DELETE 文の動詞の直後に C スタイルのコメントを記述して、オプティマイザにヒントを与えます。たとえば、オプティマイザは次の文でコストベースのアプローチを使用します。

```
SELECT /*+ COST */ ename, sal INTO ...
```

C++ コードでは、//+ という形式のオプティマイザ・ヒントも認識されます。

オプティマイザ・ヒントの詳細は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

トレース機能

SQL トレース機能と EXPLAIN PLAN 文を使用すると、アプリケーションの処理速度を低下させるおそれのある SQL 文を特定できます。

SQL トレース機能は、Oracle8i で実行された各 SQL 文についての統計情報を生成します。これらの統計情報から、処理に最も時間のかかる SQL 文を判断できます。このため、それらの文の処理効率のチューニングに専念できます。

EXPLAIN PLAN 文はアプリケーション内の各 SQL 文に対する実行計画を示します。実行計画には SQL 文を実行するために Oracle8i で実行する必要があるデータベース処理が記述されています。実行計画を使用すると、非効率的な SQL 文を特定できます。

これらのツールの使用方法および出力解析方法は、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

索引の使用

索引は、ROWID を使用して、表の列のそれぞれの値をその値が入っている行に対応付けます。索引は CREATE INDEX 文で作成します。詳細は『Oracle8i SQL リファレンス』を参照してください。

表の 15% 未満の行しか戻さない問合せでは、索引を使用することによりパフォーマンスが向上します。表の 15% 以上の行を戻す問合せは、フル・スキャンによる方法、つまり、すべての行を順番に読み込む方法の方が速く処理されます。

WHERE 句内で索引の付いた列を指定する問合せは、その索引を使用します。索引を付ける列を選択するためのガイドラインは、『Oracle8i アプリケーション開発者ガイド 基礎編』を参照してください。

行レベル・ロックの利用

デフォルトでは、Oracle8i では表レベルではなく行レベルでデータがロックされます。行レベルでロックすると、複数のユーザーが同一の表内の別の行に同時にアクセスできます。その結果、パフォーマンスが大幅に向上します。

表レベルでのロックも指定できますが、これはトランザクション処理オプションの効果を低下させます。表ロックの詳細は、3-23 ページの「[LOCK TABLE の使用方法](#)」を参照してください。

オンラインのトランザクション処理を実行するアプリケーションには、行レベル・ロックが最も有効です。アプリケーションを表レベル・ロックで運用している場合は、行レベル・ロックを利用できるように変更してください。通常、明示的な表レベル・ロックは使用しないようにします。

不要な解析の排除

不要な解析をなくすには、カーソルを正しく操作することと、次に示すカーソル管理オプションを選択して使用する必要があります。

- MAXOPENCURSORS
- HOLD_CURSOR
- RELEASE_CURSOR

これらのオプションは、暗黙的なカーソルおよび明示的なカーソル、カーソル・キャッシュおよびプライベート SQL 領域に影響します。

明示的なカーソルの操作

暗黙的なカーソルと明示的なカーソルの 2 種類のカーソルがあります。Oracle8i では、すべてのデータ定義および DML 文に対して暗黙的にカーソルが宣言されます。ただし、複数の行を戻す問合せについては、ユーザーが明示的にカーソルを宣言（つまりホスト配列を使用）する必要があります。DECLARE CURSOR 文を使用すると、明示的なカーソルを宣言できます。明示的なカーソルのオープンおよびクローズを処理する方法はパフォーマンスに影響します。

アクティブ・セットを再評価する必要がある場合は、そのカーソルを再度 OPEN するのみで済みます。OPEN では任意の新しいホスト変数値が使用されます。カーソルを再 OPEN する前に CLOSE しなければ、処理時間を節約できます。

注意：パフォーマンスの最適化を単純化するために、Oracle8i ではすでにオープンしているカーソルを再 OPEN できます。ただし、これは ANSI 拡張要素です。したがって、MODE=ANSI を指定している場合には、カーソルを再 OPEN する前に CLOSE する必要があります。

カーソルの OPEN によって取得したリソース（メモリーおよびロック）を解放するときのみそのカーソルを CLOSE します。たとえば、プログラムでは終了前にすべてのカーソルを CLOSE する必要があります。

カーソルの制御

通常、明示的に宣言したカーソルを制御する方法は次の 3 つです。

- DECLARE、OPEN および CLOSE を使用します。
- PREPARE、DECLARE、OPEN および CLOSE を使用します。
- MODE=ANSI の場合、COMMIT によってカーソルをクローズします。

最初の方法を使用する場合は、不要な解析に注意する必要があります。カーソルを CLOSE したか、まだ OPEN していないために、解析された文を使用できないときにかぎり、OPEN で解析を実行します。プログラムはカーソルを DECLARE し、ホスト変数の値が変わるたびにこれを再度 OPEN し、この SQL 文がなくなったときのみこれを CLOSE する必要があります。

2 番目の方法（動的 SQL 方法 3 および方法 4 のための）を使用する場合は、PREPARE で解析が実行され、解析された文は CLOSE を実行するまで使用できます。プログラムは SQL 文を PREPARE し、カーソルを DECLARE し、ホスト変数の値が変わるたびにこのカーソルを再度 OPEN し、SQL 文が変わった場合に SQL 文を再度 PREPARE してカーソルを再度 OPEN し、この SQL 文が不要となった場合にのみカーソルを CLOSE する必要があります。

OPEN 文および CLOSE 文をループの中に配置するのはできるだけ避けてください。SQL 文の不要な再解析の原因になります。次の例では、OPEN 文と CLOSE 文がどちらも外側の *while* ループの中にあります。MODE=ANSI の場合は、CLOSE 文は例に示す位置に配置する必要があります。ANSI では、カーソルを再度 OPEN する前に CLOSE する必要があります。

```
EXEC SQL DECLARE emp_cursor CURSOR FOR
    SELECT ename, sal from emp where sal > :salary and
                                           sal <= :salary + 1000;

salary = 0;
while (salary < 5000)
{
    EXEC SQL OPEN emp_cursor;
    while (SQLCODE==0)
    {
        EXEC SQL FETCH emp_cursor INTO ....
        ...
    }
    salary += 1000;
    EXEC SQL CLOSE emp_cursor;
}
```

一方、MODE=ORACLE のときは、カーソルを再 OPEN せずに CLOSE 文を実行できます。CLOSE 文を外側の *while* ループの外に配置することによって、OPEN 文が繰り返されるたびに再解析されるのを回避できます。

```
...
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;
```

カーソル管理オプションの使用

SQL 文は、その構成を変更しないかぎり、一度のみ解析すれば十分です。たとえば、その選択リストまたは WHERE 句に 1 列追加して、問合せの構成を変更します。HOLD_CURSOR、RELEASE_CURSOR および MAXOPENCURSORS オプションによって、SQL 文の解析および再解析を Oracle8i でどのように管理するかを制御できます。明示的なカーソルを宣言すると、解析を最大限に制御できます。

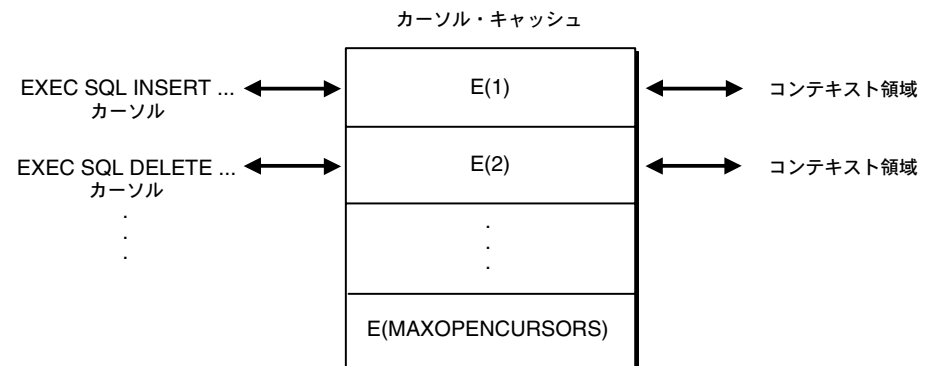
SQL 領域とカーソル・キャッシュ

DML 文を実行すると、その文に対応しているカーソルが Pro*C/C++ カーソル・キャッシュ内のエントリにリンクされます。カーソル・キャッシュとはカーソル管理のために使用されて連続的に更新されるメモリー領域です。カーソル・キャッシュ・エントリは、次々に1つのプライベート SQL 領域にリンクされます。

プライベート SQL 領域とは、Oracle8i で実行時に動的に作成される作業領域で、ホスト変数のアドレスおよびその文を処理するのに必要なその他の情報が保存されます。明示的なカーソルを使用すると、SQL 文に名前を付け、プライベート SQL 領域に保存されている情報にアクセスし、この情報の処理をある程度制御できます。

図 C-2 は、プログラムで INSERT および DELETE を実行した後のカーソル・キャッシュを表します。

図 C-2 カーソル・キャッシュでリンクされたカーソル



リソースの使用

ユーザー・セッションごとのオープン・カーソルの最大数は、初期化パラメータ OPEN_CURSORS によって設定します。

MAXOPENCURSORS は、カーソル・キャッシュの初期サイズを指定します。新しいカーソルが必要で、しかも空きのキャッシュ・エントリがない場合、Oracle8i ではエントリの再利用が試行されます。再利用の可能性は HOLD_CURSOR と RELEASE_CURSOR の値によって決まり、また明示的カーソルの場合には、カーソル自身の状態によって決まります。

MAXOPENCURSORS の値が実際に必要なキャッシュ・エントリの数よりも小さい場合、Oracle8i では最初のキャッシュ・エントリが再利用可能としてマークされます。たとえば、INSERT 文のキャッシュ・エントリ E (1) が再利用可能とマークされていると、キャッシュ・エントリの数は MAXOPENCURSORS と等しくなります。プログラムが新しい文を実行する場合、キャッシュ・エントリ E (1) とそのプライベート SQL 領域は新しい文に再度

割り当てられることがあります。INSERT 文を再実行するために、Oracle8i ではそれをもう一度解析しなおして、別のキャッシュ・エントリを再度割り当てる必要があります。

再利用できるキャッシュ・エントリが見つからない場合、Oracle8i では追加のキャッシュ・エントリが割り当てられます。たとえば、MAXOPENCURSORS=8 で、8 エントリすべてがアクティブな場合、9 番目のエントリが作成されます。必要に応じて Oracle8i では、空きメモリーがなくなるか OPEN_CURSORS で設定された限界に達するまで、キャッシュ・エントリの割り当てが続行されます。この動的割り当ては、処理オーバーヘッドを増大させます。

したがって、MAXOPENCURSORS の値を小さく設定すると、メモリーの節約にはなりますが、新しいキャッシュ・エントリの動的割り当ておよび割り当て解除に資源を消耗する場合があります。MAXOPENCURSORS の値を大きく設定すると、実行は確実に速くなりますが、より大きなメモリーを使用することになります。

実行回数の少ない場合

実行回数の少ない SQL 文とそのプライベート SQL 領域間のリンクは、一時的なものにした方がよい場合もあります。

HOLD_CURSOR=NO (デフォルト値) と指定した場合、Oracle8i によりその文が実行されカーソルがクローズされた後、このカーソルとカーソル・キャッシュとの間のリンクがプリコンパイラにより再利用可能としてマークされます。このリンクは、それが示すカーソル・キャッシュ・エントリが別の SQL 文に必要になると、すぐに再利用されます。これにより、プライベート SQL 領域に割り当てられたメモリーが解放され、解析ロックが解除されます。しかし、PREPARE したカーソルは実行状態のままにする必要があるため、HOLD_CURSOR=NO と指定した場合でもそのリンクは維持されます。

RELEASE_CURSOR=YES と指定した場合は、Oracle8i によりその SQL 文が実行されカーソルがクローズされた後、プライベート SQL 領域が自動的に解放され、解析した文は失われます。メモリーの節約のために MAXOPENCURSORS を低い値に設定しているような場合には、この指定が必要です。

DML 文がデータ定義文より前にあり、どちらの文も同じ表を参照する場合には、DML 文に RELEASE_CURSOR=YES を指定してください。これにより、DML 文が得る解析ロックと、データ定義文が得る排他ロックとの間の対立が回避されます。

RELEASE_CURSOR=YES を指定した場合は、プライベート SQL 領域とキャッシュ・エントリ間のリンクはただちに削除され、このプライベート SQL 領域は解放されます。HOLD_CURSOR=YES と指定している場合でも、HOLD_CURSOR=YES が RELEASE_CURSOR=YES によって上書きされるため、Oracle8i で SQL 文を実行する前にプライベート SQL 領域のメモリーを再び割り当て、この SQL 文を再解析する必要があります。

ただし、RELEASE_CURSOR=YES を指定した場合は、Oracle8i では SQL 文と PL/SQL ブロックの解析された表現が共有 SQL キャッシュに保持されるため、それ以上再解析を処理する必要がないこともあります。カーソルをクローズしても、解析された表現はキャッシュの期限切れまで使用できます。

実行回数の多い場合

プライベート SQL 領域には SQL 文の実行に必要なすべての情報が含まれるため、頻繁に実行される SQL 文とそのプライベート SQL 領域のリンクを維持する必要があります。この情報へのアクセスを上手に管理すれば、後続の文の実行速度をさらに向上させることができます。

HOLD_CURSOR=YES の場合、Oracle8i で SQL 文を実行した後にカーソルとカーソル・キャッシュのリンクが維持されます。したがって、解析された文と割り当てられたメモリーが、利用可能なまま維持されます。これは、不必要な再解析を避けるために、アクティブにしておく SQL 文で役に立ちます。

RELEASE_CURSOR=NO（デフォルト値）の場合、Oracle8i で SQL 文を実行した後にキャッシュ・エントリとプライベート SQL 領域間のリンクが維持され、オープンしたカーソルの数が MAXOPENCURSORS の値を超えないかぎり、そのリンクは再利用されません。これは、解析した文および割り当てたメモリーが使用可能な状態のままなので、頻繁に実行する SQL 文の場合に有効です。

注意 : Oracle の前のバージョンでは、SQL 文の実行後に RELEASE_CURSOR=NO となっていると、解析後の表現を引き続き使用することができました。しかし、Oracle8i では、RELEASE_CURSOR=NO および HOLD_CURSOR=YES が指定されている場合、解析後の表現を使用できるのは共有 SQL キャッシュの内容が期限切れになるまでの間です。通常、このことは問題にはなりませんが、SQL 文が再解析される前に参照されたオブジェクトの定義が変更されると、予期しない結果をもたらすことがあります。

埋込み PL/SQL の考慮事項

カーソルを管理する目的で、埋込み PL/SQL ブロックは SQL 文と同様に扱われます。埋込み PL/SQL ブロックが実行されると、親カーソルはブロック全体と関連付けられ、キャッシュ・エントリと埋込み PL/SQL ブロックに対する PGA のプライベート SQL 領域との間にリンクが作成されます。PL/SQL ブロック内の各 SQL 文にも、PGA のプライベート SQL 領域が必要だということに注意してください。それらの SQL 文は PL/SQL 自体で管理される子カーソルを使用します。子カーソルの性質は関連する親カーソルによって決定されます。つまり、子カーソルによって使用されるプライベート SQL 領域は、親カーソルのプライベート SQL 領域が解放されたときに解放されます。

パラメータの相互作用

表 C-1 に、HOLD_CURSOR と RELEASE_CURSOR の相互関係を示します。HOLD_CURSOR=NO を指定すると、RELEASE_CURSOR=NO は変更され、RELEASE_CURSOR=YES を指定すると、HOLD_CURSOR=YES が変更されることに注意してください。

表 C-1 HOLD_CURSOR と RELEASE_CURSOR の相互関係

HOLD_CURSOR	RELEASE_CURSOR	リンク
NO	NO	再利用可能とマークされる。
YES	NO	維持される。
NO	YES	ただちに削除される。
YES	YES	ただちに削除される。

構文検査と意味検査

埋込み SQL 文および PL/SQL ブロックの構文および意味を検査することによって、Pro*C/C++ プリコンパイラはコーディングの誤りをすみやかに発見し修正できるように支援します。この付録では、プリコンパイラ・オプションの SQLCHECK を使用して検査の種類および範囲を制御する方法を説明します。

この章では、次の事項について説明します。

- 構文検査と意味検査
- 検査の種類および範囲の制御
- SQLCHECK=SEMANTICS の指定
- SQLCHECK=SYNTAX の指定
- SQLCHECK オプションの入力

構文検査と意味検査

構文規則は、言語構造を並べて正しい文を作成する基準を示します。つまり、構文検査はキーワード、オブジェクト名、演算子、デリミタなどが SQL 文に正しく配置されていることを検証します。たとえば、次の埋込み SQL 文には構文上のエラーがあります。

```
EXEC SQL DELETE FROM EMP WHERE DEPTNO = 20;
-- misspelled keyword WHERE
EXEC SQL INSERT INTO EMP COMM, SAL VALUES (NULL, 1500);
-- missing parentheses around column names COMM and SAL
```

意味上の規則は、有効な外部参照を行う方法を示しています。つまり、意味検査はデータベース・オブジェクトおよびホスト変数への参照が正しいこと、さらにホスト変数のデータ型が正しいことを検証します。たとえば、次の埋込み SQL 文には意味上のエラーがあります。

```
EXEC SQL DELETE FROM emp WHERE deptno = 20;
-- nonexistent table, EMPP
EXEC SQL SELECT * FROM emp WHERE ename = :emp_name;
-- undeclared host variable, emp_name
```

SQL 構文と方法に関するルールは、『Oracle8i SQL リファレンス』に定義されています。

検査の種類および範囲の制御

コマンドラインでプリコンパイラ・オプションの SQLCHECK を指定することによって、検査の種類および範囲を制御します。SQLCHECK では、検査の種類は、構文、意味、その両方の 3 種類があります。検査の範囲には、次の要素を含めることができます。

- データ定義文 (CREATE および GRANT など)
- DML 文 (SELECT および INSERT など)
- PL/SQL ブロック

ただし、動的 SQL 文は実行時まで完全に定義されないため、SQLCHECK では動的 SQL 文を検査できません。

SQLCHECK について次の値を指定できます。

- SEMANTICS または FULL
- SYNTAX

デフォルト値は SYNTAX です。

SQLCHECK を使用しても、データ制御、カーソル制御、動的 SQL 文に対する通常の構文検査には影響しません。

SQLCHECK=SEMANTICS の指定

SQLCHECK=SEMANTICS の場合、プリコンパイラは次の内容について構文および意味の検査を行います。

- DML 文 (INSERT、UPDATE など)
- PL/SQL ブロック
- ホスト変数のデータ型

次の構文も同様です。

- データ定義文 (CREATE、ALTER など)

ただし、`ATdb_name` 句を使用する DML 文に対しては構文検査のみを行います。

SQLCHECK=SEMANTICS の場合、プリコンパイラは、埋め込まれた `DECLARE TABLE` 文を使用して、意味検査に必要な情報を入手します。ただし、コマンドラインで `USERID` オプションを指定した場合は、`Oracle8i` に接続し、そのデータ・ディクショナリにアクセスしてこの情報を入手します。DML 文または PL/SQL ブロックで参照される表がすべて `DECLARE TABLE` 文で定義されていれば、`Oracle8i` に接続する必要はありません。

`Oracle8i` に接続したとき、データ・ディクショナリに必要な情報の一部が見つからない場合、`DECLARE TABLE` トリガー文を使用して、欠けている情報を補充する必要があります。`DECLARE TABLE` 文とデータ・ディクショナリの定義が矛盾する場合は、`DECLARE TABLE` の定義が使用されます。

ホスト・プログラム内に PL/SQL ブロックを埋め込むときは、必ず `SQLCHECK=SEMANTICS` を指定してください。

DML 文を検査する際に、プリコンパイラは『`Oracle8i SQL リファレンス`』に記述されている `Oracle8i` の構文規則を使用しますが、意味検査にはより厳密な規則を使用します。特に、データ型のチェックは厳密に行います。その結果、`SQLCHECK=SEMANTICS` のときは、`Oracle` の以前のバージョン用に作成した既存のアプリケーションを正常にプリコンパイルできない場合があります。

新規のプログラムをプリコンパイルする場合、またはデータ型のチェックを厳密に行う場合は、`SQLCHECK=SEMANTICS` と指定してください。

意味検査の使用許可

`SQLCHECK=SEMANTICS` を指定すると、プリコンパイラは意味検査に必要な情報を、次の方法のどれかで入手できます。

- `Oracle8i` に接続してデータ・ディクショナリにアクセスします。
- 埋め込まれた `DECLARE TABLE` 文と `DECLARE TYPE` 文を使用します。

Oracle8i への接続

意味検査を行うには、表、データ型、ホスト・プログラムで参照されるビューをメンテナンスする Oracle データベースにプリコンパイラを接続します。

Oracle に接続した後、プリコンパイラはデータ・ディクショナリにアクセスして必要な情報を探します。データ・ディクショナリには、表および列の名前、表および列の制約、列の長さ、列のデータ型などが格納されています。

必要な情報の一部がデータ・ディクショナリ内で見つからない場合（たとえば、プログラムがまだ作成していない表を参照するためなど）は、DECLARE TABLE 文（この付録で後述されています）を使用して足りない情報を指定する必要があります。

Oracle8i に接続するには、次の構文を使用してコマンドラインで USERID オプションを指定します。

```
USERID=username/password
```

username および *password* により有効な Oracle8i ユーザー ID が構成されます。パスワードを省略すると、パスワードの入力が求められます。

仮に、ユーザー名とパスワードのかわりに、次のように指定したとします。

```
USERID=/  

```

プリコンパイラにより Oracle8i への接続が自動的に試行されます。この自動接続が成功するのは、既存の Oracle8i ユーザー名がオペレーティング・システムの ID の前に「OPS\$」を付けたもの、または INI.ORA ファイル内の OS_AUTHENT_PREFIX パラメータの設定値と一致する場合のみです。たとえば、使用しているオペレーティング・システムの ID が MBLAKE であるときに、自動接続が成功するのは OPS\$MBLAKE が有効な Oracle8i ユーザー名である場合のみです。

USERID オプションを省略した場合、プリコンパイラは埋込み DECLARE TABLE 文から必要な情報を取得する必要があります。

Oracle8i に接続しようとしてもできない場合（たとえば、データベースが使用できない場合）、エラー・メッセージが出されてプログラムのプリコンパイルは実行されません。

DECLARE TABLE の使用

プリコンパイラは Oracle8i に接続せずに意味検査を実行できます。この検査を行うには、プリコンパイラは表およびビューに関する情報を埋込み DECLARE TABLE 文から取得する必要があります。つまり、DML 文または PL/SQL ブロック内で参照する表をすべて DECLARE TABLE 文内で定義する必要があります。

DECLARE TABLE 文の構文は、次のとおりです。

```
EXEC SQL DECLARE table_name TABLE  
      (col_name col_datatype [DEFAULT expr] [NULL|NOT NULL], ...);
```

expr は、CREATE TABLE 文で列のデフォルト値として使用できる整数です。

ユーザー定義のオブジェクト・データ型の場合、サイズは使用されないのがオプションです。

DECLARE TABLE を使用して既存のデータベースの表を定義した場合、プリコンパイラはその定義に従います。このときデータ・ディクショナリの定義は無視されます。

DECLARE TYPE の使用

同様に、TYPE の場合は、次の構文による DECLARE TYPE 文があります。

```
EXEC SQL DECLARE type TYPE  
[AS OBJECT (col_name col_datatype, ...)] |  
[AS VARRAY(size) OF element_type ] |  
[AS TABLE OF object_type ] ;
```

この文を使用すると、プリコンパイル時に SQLCHECK=SEMANTICS と指定した場合に、ユーザー定義型のより適切なタイプ・チェックを実行できます。SQLCHECK=SYNTAX と指定すると、DECLARE TYPE 文はドキュメントとしてのみ動作し、コメントとして扱われ、無視されます。

SQLCHECK=SYNTAX の指定

SQLCHECK=SYNTAX を指定すると、プリコンパイラで SQL 文の構文がチェックされず。構文は『Oracle8i SQL リファレンス』に説明されています。

- DML 文
- ホスト変数表現

意味上のチェックが実行されないため、次の制限事項が適用されます。

- Oracle8i への接続は行われず、USERID は無効なオプションとなります。USERID を指定すると、警告メッセージが発行されます。
- DECLARE TABLE 文と DECLARE TYPE 文は無視され、ドキュメントとしてのみ機能します。
- PL/SQL ブロックの埋込みはできません。プリコンパイラが PL/SQL ブロックを検出すると、エラー・メッセージが発行されます。

DML 文をチェックする場合、プリコンパイラは Oracle8i の構文規則を使用します。これらの規則は下位互換性があるため、プリコンパイルしたプログラムを移行する際には SQLCHECK=SYNTAX を指定してください。

SQLCHECK オプションの入力

SQLCHECK オプションは、インラインまたはコマンドラインで入力できます。ただし、インラインで指定するチェックのレベルを、コマンドラインで指定する（またはデフォルトによって受け入れる）レベルよりも高くすることはできません。たとえば、SQLCHECK=SYNTAX をコマンドラインで指定した場合、インラインでは SQLCHECK=SEMANTICS を指定できません。

システム固有の参照

この付録では、このマニュアルで参照しているシステム固有の情報をすべてまとめて記載します。

システム固有の情報

システム固有の情報は、使用している Oracle システムのマニュアルで説明しています。

標準ヘッダー・ファイルの位置

標準 Pro*C/C++ ヘッダー・ファイル (*sqlca.h*, *oraca.h* および *sqlda.h*) の位置は、システムによって異なります。他のシステムについては、使用している Oracle システムのマニュアルを参照してください。

C コンパイラ用組み込みファイルの位置指定

Pro*C/C++ コマンドライン・オプション INCLUDE= を使用して、組み込まれる非標準ファイルの位置を指定する場合は、C コンパイラにも同じ位置を指定する必要があります。これを実行する方法はシステム固有です。5-34 ページの「[インクルード・ファイル](#)」を参照してください。

ANSI C サポート

CODE= オプションを使用して、Pro*C/C++ で生成される C コードと使用中のシステムの C コンパイラとの互換性を確保します。2-11 ページの「[関数プロトタイプ](#)」を参照してください。

構造体コンポーネントの位置合せ

通常は、システムのハードウェアによって、C コンパイラが構造体のコンポーネントの位置合せを行う方法が異なります。*sqlvcp()* 関数を使用して、VARCHAR 構造体の *.arr* コンポーネントに追加される埋込みを判別してください。詳細は 4-20 ページの「[VARCHAR 配列コンポーネントの長さを調べる方法](#)」を参照してください。

整数と ROWID のサイズ

整数データ型のバイト数と ROWID データ型のバイナリ外部サイズは、システムによって異なります。4-6 ページの「[INTEGER](#)」および 4-7 ページの「[ROWID](#)」を参照してください。

バイトの並び

1つのワード中のバイトの並びは、プラットフォームによって異なります。詳細は 4-9 ページの「[UNSIGNED](#)」を参照してください。

Oracle8i への接続

Net8 ドライバを使用して Oracle8i に接続するには、システム固有のネットワーク・プロトコルが必要です。詳細は、5-44 ページの「[OCI リリース 8 とのインタフェース](#)」を参照してください。

XA ライブラリでのリンク

XA ライブラリでのリンクは、システムによって異なります。詳細は、5-53 ページの「[リンク](#)」および Oracle のインストレーション・ガイドまたはユーザーズ・ガイドを参照してください。

Pro*C/C++ 実行モジュールの位置

Pro*C/C++ プリコンパイラの位置は、システム固有です。詳細は、10-2 ページの「[プリコンパイラのコマンド](#)」およびインストレーション・ガイドまたはユーザーズ・ガイドを参照してください。

システム構成ファイル

各プリコンパイラのインストールには、システム構成ファイルがあります。このファイルはプリコンパイラに付属しているものではないため、システム管理者が作成する必要があります。Pro*C/C++ がシステム構成ファイルを検索する位置（ディレクトリ・パス）は、システムによって異なります。詳細は、10-6 ページの「[プリコンパイル中の状況](#)」を参照してください。

INCLUDE オプションの構文

INCLUDE コマンドライン・オプションの値に対応する構文は、システム固有です。10-24 ページの「[INCLUDE](#)」を参照してください。

コンパイルとリンク

Pro*C/C++ 出力をコンパイルおよびリンクして実行可能なアプリケーションを得る方法は、常にシステムによって異なります。詳細は、2-16 ページの「[コンパイルとリンク](#)」およびこの後の各項を参照してください。

ユーザー・イグジット

Oracle Forms のユーザー・イグジットのコンパイルおよびリンクは、システム固有です。第 20 章「[ユーザー・イグジット](#)」を参照してください。

埋込み SQL 文および宣言文

この付録では、SQL92 の埋込み文と宣言文、および Oracle の埋込み SQL の拡張要素について説明します。

注意：この付録では、非埋込み SQL と構文が異なる文のみを説明します。非埋込み SQL 文の詳細は、『Oracle8i SQL リファレンス』を参照してください。

この付録の構成は、次のとおりです。

- プリコンパイラの宣言文と埋込み SQL 文の概要
- 文の説明
- 構文図の読み方
- ALLOCATE (実行可能埋込み SQL 拡張要素)
- ALLOCATE DESCRIPTOR (実行可能埋込み SQL)
- CACHE FREE ALL (実行可能埋込み SQL 拡張要素)
- CALL (実行可能埋込み SQL)
- CLOSE (実行可能埋込み SQL)
- COLLECTION APPEND (実行可能埋込み SQL 拡張要素)
- COLLECTION DESCRIBE (実行可能埋込み SQL 拡張要素)
- COLLECTION GET (実行可能埋込み SQL 拡張要素)
- COLLECTION RESET (実行可能埋込み SQL 拡張要素)
- COLLECTION SET (実行可能埋込み SQL 拡張要素)
- COLLECTION TRIM (実行可能埋込み SQL 拡張要素)
- COMMIT (実行可能埋込み SQL)
- CONNECT (実行可能埋込み SQL 拡張要素)

-
- CONTEXT ALLOCATE (実行可能埋込み SQL 拡張要素)
 - CONTEXT FREE (実行可能埋込み SQL 拡張要素)
 - CONTEXT OBJECT OPTION GET (実行可能埋込み SQL 拡張要素)
 - CONTEXT OBJECT OPTION SET (実行可能埋込み SQL 拡張要素)
 - CONTEXT USE (Oracle 埋込み SQL 宣言文)
 - DEALLOCATE DESCRIPTOR (埋込み SQL 文)
 - DECLARE CURSOR (埋込み SQL 宣言文)
 - DECLARE DATABASE (Oracle 埋込み SQL 宣言文)
 - DECLARE STATEMENT (埋込み SQL 宣言文)
 - DECLARE TABLE (Oracle 埋込み SQL 宣言文)
 - DECLARE TYPE (Oracle 埋込み SQL 宣言文)
 - DELETE (実行可能埋込み SQL)
 - DESCRIBE (実行可能埋込み SQL 拡張要素)
 - DESCRIBE DESCRIPTOR (実行可能埋込み SQL)
 - ENABLE THREADS (実行可能埋込み SQL 拡張要素)
 - EXECUTE... END-EXEC (実行可能埋込み SQL 拡張要素)
 - EXECUTE (実行可能埋込み SQL)
 - EXECUTE DESCRIPTOR (実行可能埋込み SQL)
 - EXECUTE IMMEDIATE (実行可能埋込み SQL)
 - FETCH (実行可能埋込み SQL)
 - FETCH DESCRIPTOR (実行可能埋込み SQL)
 - FREE (実行可能埋込み SQL 拡張要素)
 - GET DESCRIPTOR (実行可能埋込み SQL)
 - INSERT (実行可能埋込み SQL)
 - LOB APPEND (実行可能埋込み SQL 拡張要素)
 - LOB ASSIGN (実行可能埋込み SQL 拡張要素)
 - LOB CLOSE (実行可能埋込み SQL 拡張要素)
 - LOB COPY (実行可能埋込み SQL 拡張要素)
 - LOB CREATE TEMPORARY (実行可能埋込み SQL 拡張要素)

-
- LOB DESCRIBE (実行可能埋込み SQL 拡張要素)
 - LOB DISABLE BUFFERING (実行可能埋込み SQL 拡張要素)
 - LOB ENABLE BUFFERING (実行可能埋込み SQL 拡張要素)
 - LOB ERASE (実行可能埋込み SQL 拡張要素)
 - LOB FILE CLOSE ALL (実行可能埋込み SQL 拡張要素)
 - LOB FILE SET (実行可能埋込み SQL 拡張要素)
 - LOB FLUSH BUFFER (実行可能埋込み SQL 拡張要素)
 - LOB FREE TEMPORARY (実行可能埋込み SQL 拡張要素)
 - LOB LOAD (実行可能埋込み SQL 拡張要素)
 - LOB OPEN (実行可能埋込み SQL 拡張要素)
 - LOB READ (実行可能埋込み SQL 拡張要素)
 - LOB TRIM (実行可能埋込み SQL 拡張要素)
 - LOB WRITE (実行可能埋込み SQL 拡張要素)
 - OBJECT CREATE (実行可能埋込み SQL 拡張要素)
 - OBJECT DELETE (実行可能埋込み SQL 拡張要素)
 - OBJECT Deref (実行可能埋込み SQL 拡張要素)
 - OBJECT FLUSH (実行可能埋込み SQL 拡張要素)
 - OBJECT GET (実行可能埋込み SQL 拡張要素)
 - OBJECT RELEASE (実行可能埋込み SQL 拡張要素)
 - OBJECT SET (実行可能埋込み SQL 拡張要素)
 - OBJECT UPDATE (実行可能埋込み SQL 拡張要素)
 - OPEN (実行可能埋込み SQL)
 - OPEN DESCRIPTOR (実行可能埋込み SQL)
 - PREPARE (実行可能埋込み SQL)
 - REGISTER CONNECT (実行可能埋込み SQL 拡張要素)
 - ROLLBACK (実行可能埋込み SQL)
 - SAVEPOINT (実行可能埋込み SQL)
 - SELECT (実行可能埋込み SQL)
 - SET DESCRIPTOR (実行可能埋込み SQL)

-
- TYPE (Oracle 埋込み SQL 宣言文)
 - UPDATE (実行可能埋込み SQL)
 - VAR (Oracle 埋込み SQL 宣言文)
 - WHENEVER (埋込み SQL 宣言文)

埋込み SQL コマンドにより、DDL、DML およびトランザクション制御文を Pro*C/C++ プログラム内で使用できます。表 F-1 に埋込み SQL 文と宣言文の機能の概要を示します。

タイプ 実行 (E) 文または宣言文 (D)

EXEC SQL 文	ソース/タイプ	用途
ALLOCATE	O/E	カーソル変数またはオブジェクト型にメモリーを割り当てます。
ALLOCATE DESCRIPTOR	S/E	ANSI 動的 SQL の記述子を割り当てます。
CACHE FREE ALL	O/E	割り当てられたオブジェクト・キャッシュ・メモリーをすべて解放します。
CALL	S/E	ストアド・プロシージャをコールします。
CLOSE	S/E	保持されているリソースを解放し、カーソルを使用禁止にします。
COLLECTION APPEND	O/E	1つのコレクションの要素を別のコレクションの最後に追加します。
COLLECTION DESCRIBE	O/E	コレクションについての情報を取得します。
COLLECTION GET	O/E	コレクションの要素を取得します。
COLLECTION RESET	O/E	コレクションのスライス・エンドポイントをコレクションの最初にリセットします。
COLLECTION SET	O/E	コレクションの値を更新します。
COLLECTION TRIM	O/E	コレクションの最後から要素を削除します。
COMMIT	S/E	データベースへの変更内容をすべて確定して、カレント・トランザクションを終了します。(オプションでリソースを解放し、データベースから切断します。)
CONNECT	O/E	インスタンスにログインします。
CONTEXT ALLOCATE	O/E	SQLLIB ランタイム・コンテキストにメモリーを割り当てます。
CONTEXT FREE	O/E	SQLLIB ランタイム・コンテキストのメモリーを解放します。

表 F-1 プリコンパイラ宣言文および埋込み SQL 文および句

EXEC SQL 文	ソース/タイプ	用途
CONTEXT OBJECT OPTION GET	O/E	オプションの設定方法を判断します。
CONTEXT OBJECT OPTION SET	O/E	オプションを設定します。
CONTEXT USE	O/D	後続の実行 SQL 文で使用する SQLLIB ランタイム・コンテキストを指定します。
DEALLOCATE DESCRIPTOR	S/E	メモリーを解放するために記述子領域の割当てを解除します。
DECLARE CURSOR	S/D	問合せに対応付けてカーソルを宣言します。
DECLARE DATABASE	O/D	後続の埋込み SQL 文でアクセスされるデフォルト以外のデータベースの識別子を宣言します。
DECLARE STATEMENT	S/D	SQL 文に SQL 変数名を割り当てます。
DECLARE TABLE	O/D	Pro*C/C++ によって埋込み SQL 文の意味検査に使用される表の構造を宣言します。
DECLARE TYPE	O/D	Pro*C/C++ による埋込み SQL 文の意味検査に使用される型の構造体を宣言します。
DELETE	S/E	表またはビューのベース表から行を削除します。
DESCRIBE	S/E	記述子（ホスト変数の説明を保持している構造体）を初期化します。
DESCRIBE DESCRIPTOR	S/E	ANSI SQL 文の変数についての情報を取得します。
ENABLE THREADS	O/E	複数のスレッドをサポートするプロセスを初期化します。
EXECUTE...END-EXEC	O/E	無名 PL/SQL ブロックを実行します。
EXECUTE	S/E	準備済の動的 SQL 文を実行します。
EXECUTE DESCRIPTOR	S/E	ANSI 方法 4 動的 SQL 文を実行します。
EXECUTE IMMEDIATE	S/E	ホスト変数をもたない SQL 文を準備して実行します。
FETCH	S/E	問合せで選択した行を取り出します。
FETCH DESCRIPTOR	S/E	ANSI 方法 4 動的 SQL を使用して選択された行を取得します。
FREE	O/E	オブジェクト・キャッシュまたはカーソルに割り当てられているメモリーを解放します。
GET DESCRIPTOR	S/E	ANSI SQL 記述子領域の情報をホスト変数に移動します。
INSERT	S/E	表またはビューのベース表に行を追加します。

表 F-1 プリコンパイラ宣言文および埋込み SQL 文および句

EXEC SQL 文	ソース/タイプ	用途
LOB APPEND	O/E	LOB の最後に別の LOB を追加します。
LOB ASSIGN	O/E	LOB または BFILE ロケータを別のロケータに割り当てます。
LOB CLOSE	O/E	LOB または BFILE をクローズします。
LOB COPY	O/E	LOB 値の全部または一部を別の LOB にコピーします。
LOB CREATE TEMPORARY	O/E	テンポラリ LOB を作成します。
LOB DESCRIBE	O/E	LOB から属性を取得します。
LOB DISABLE BUFFERING	O/E	LOB バッファリングを使用禁止にします。
LOB ENABLE BUFFERING	O/E	LOB バッファリングを使用可能にします。
LOB ERASE	O/E	LOB データの任意の値の消去を任意のオフセットから開始します。
LOB FILE CLOSE ALL	O/E	オープンしているすべての BFILE をクローズします。
LOB FILE SET	O/E	BFILE ロケータに DIRECTORY および FILENAME を設定します。
LOB FLUSH BUFFER	O/E	データベース・サーバーに LOB バッファを書き込みます。
LOB FREE TEMPORARY	O/E	LOB ロケータのテンポラリ領域を解放します。
LOB LOAD	O/E	BFILE の全部または一部を内部 LOB にコピーします。
LOB OPEN	O/E	読み込みまたは読書きアクセスで使用する LOB または BFILE をオープンします。
LOB READ	O/E	LOB または BFILE の全部または一部をバッファに読み込みます。
LOB TRIM	O/E	LOB 値を切り捨てます。
LOB WRITE	O/E	バッファの内容を LOB に書き込みます。
OBJECT CREATE	O/E	キャッシュ内で参照可能オブジェクトを作成します。
OBJECT DELETE	O/E	オブジェクトに削除マークを設定します。
OBJECT Deref	O/E	オブジェクトを間接参照します。
OBJECT FLUSH	O/E	永続オブジェクトをサーバーに送信します。
OBJECT GET	O/E	オブジェクト属性を C の型に変換します。
OBJECT RELEASE	O/E	キャッシュ内のオブジェクトを確保解除します。

表 F-1 プリコンパイラ宣言文および埋込み SQL 文および句

EXEC SQL 文	ソース/タイプ	用途
OBJECT SET	O/E	キャッシュ内のオブジェクト属性を更新します。
OBJECT UPDATE	O/E	キャッシュ内のオブジェクトに更新マークを設定します。
OPEN	S/E	カーソルに対応付けられた問合せを実行します。
OPEN DESCRIPTOR	S/E	カーソルに対応付けられた問合せを実行します (ANSI 動的 SQL 方法 4)。
PREPARE	S/E	動的 SQL 文を解析します。
REGISTER CONNECT	O/E	外部プロシージャへのコールを使用可能にします。
ROLLBACK	S/E	カレント・トランザクションを終了し、カレント・トランザクションの変更内容をすべて破棄し、ロックをすべて解除します (オプションでリソースを解放し、データベースから切断します)。
SAVEPOINT	S/E	後でロールバックする位置をトランザクション内に指定します。
SELECT	S/E	選択した値をホスト変数に割り当てて、1 つまたは複数の表、ビューまたはスナップショットからデータを取り出します。
SET DESCRIPTOR	S/E	ホスト変数からの情報を記述子領域に設定します。
TYPE	O/D	外部のデータ型をユーザー定義のデータ型と同値化して、外部データ型をホスト変数のクラス全体に割り当てます。
UPDATE	S/E	表またはビューのベース表内の既存値を変更します。
VAR	O/D	デフォルトのデータ型を無効にして、特定の外部データ型をホスト変数に割り当てます。
WHENEVER	S/D	エラー状態および警告状態の処置を指定します。

文の説明

宣言文および文はアルファベット順に並べてあります。各文の説明には、次の項目があります。

用途	文の基本的な用途を示します。
前提条件	必要な権限と、文を使用する前に実行する必要がある手順を示します。特記していない限り、ほとんどの文ではユーザーのインスタンスでデータベースがオープンされている必要があります。
構文	文のキーワードとパラメータを示します。
キーワードおよびパラメータ	各キーワードとパラメータの用途を示します。
使用上の注意	文の使用方法与条件を示します。
例	文の例文を示します。
関連項目	関連する文、句およびこのマニュアルの関連項目を示します。

構文図の読み方

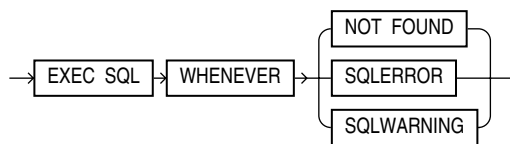
埋込み SQL の構文は、分かりやすいように構文図を使用して説明します。構文図は、正しい構文のパスを示す図です。

構文図は、左から右に矢印が指す方向にたどってください。

文と他のキーワードは、四角形の中に大文字で表記されています。これらの文字は、四角形の中に表示されているとおり正確に入力してください。パラメータは、楕円形の中に小文字で表記されています。記述する文のパラメータを変数に置き換えてください。演算子、デリミタおよび終了記号は、円の中に表記されています。「はじめに」で定義した規則に従って、文の終わりにはセミコロンを付けます。

構文図に複数のパスがある場合は、任意のパスを選択できます。

キーワード、演算子またはパラメータの選択肢が複数ある場合は、オプションを縦に並べて示します。次の例では、縦方向に自由に進み、それから横線に戻ることができます。

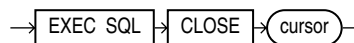


この図は、次の文がすべて有効であることを示しています。

```
EXEC SQL WHENEVER NOT FOUND ...
EXEC SQL WHENEVER SQLERROR ...
EXEC SQL WHENEVER SQLWARNING ...
```

必須のキーワードとパラメータ

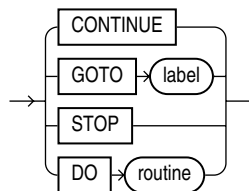
必須のキーワードとパラメータは単一で、あるいは代替の選択肢を縦に並べて示します。必須のキーワードまたはパラメータが1つしかない場合は、メイン・パス、つまり現在たっている横線上に示します。次の例では、*cursor* は必須パラメータです。



emp_cursor というカーソルが存在する場合、この図では次の文が有効です。

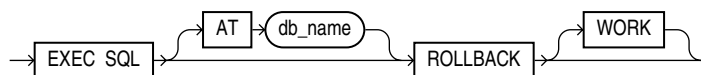
```
EXEC SQL CLOSE emp_cursor;
```

複数のキーワードまたはパラメータがメイン・パス上に縦に並んでいる場合は、その1つが必須です。つまり、キーワードやパラメータを1つ選択する必要がありますが、メイン・パスのちょうど上にあるものとは限りません。次の例では、4つのアクションのうち1つを選択する必要があります。



オプションのキーワードとパラメータ

キーワードとパラメータがメイン・パスの上に縦に並べられている場合、それらはオプションです。次の例では、「AT:db_name」および「WORK」がオプション設定になります。

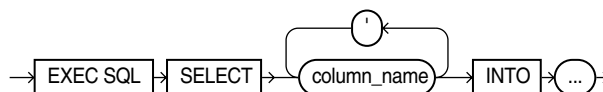


この図では、*oracle2* という名前のデータベースが存在する場合、次の文はすべて有効です。

```
EXEC SQL ROLLBACK;
EXEC SQL ROLLBACK WORK;
EXEC SQL AT oracle2 ROLLBACK;
```

構文ループ

ループは、その中の構文を何回でも繰り返せることを示します。次の例では、*column_name* がループの中にあります。したがって、列名を1つ選択した後に、他の列名をカンマで区切って繰り返し選択できます。

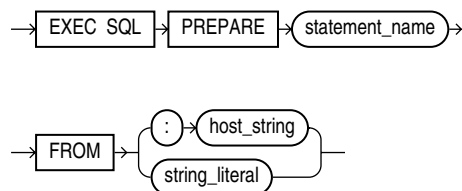


DEBIT、CREDIT および BALANCE が列名の場合、この図では次の文がすべて有効です。

```
EXEC SQL SELECT DEBIT INTO ...
EXEC SQL SELECT CREDIT, BALANCE INTO ...
EXEC SQL SELECT DEBIT, CREDIT, BALANCE INTO ...
```

複数パーツの図

複数パーツの図では、メイン・パスがすべて端から端まで結合されていると考えます。次の例は2パーツの図です。



この図は、次の文が有効であることを示しています。

```
EXEC SQL PREPARE statement_name FROM string_literal;
```

Oracle の名前

表や列などの Oracle データベース・オブジェクトの名前の長さは、30 文字を超えないでください。先頭文字は英文字である必要がありますが、残りの文字には、英文字、数字、ドル記号（\$）、ポンド記号（#）、アンダースコア（_）を任意に組み合わせて使用できます。

ただし、Oracle 識別子を引用符（"）で囲むと、有効な文字を任意に組み合わせて使用することができ、この場合空白は有効な文字ですが、引用符は無効です。

Oracle の名前は、引用符で囲んだ場合を除いて大 / 小文字の区別がありません。

文終了記号

どの埋込み SQL 図の場合も、各文は文終了記号 ";" で終わるものと見なされます。

ALLOCATE（実行可能埋込み SQL 拡張要素）

用途

カーソル変数が PL/SQL ブロックで参照されるように割り当てるか、オブジェクト・キャッシュにスペースを割り当てます。

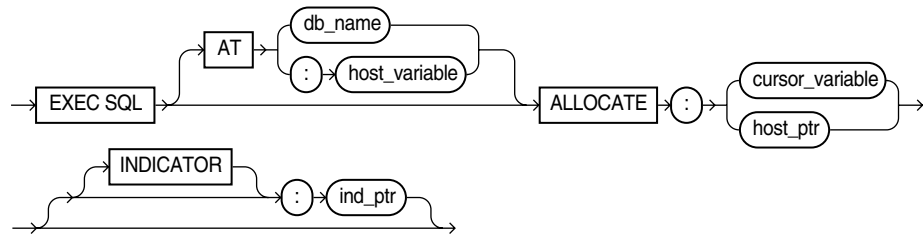
前提条件

カーソル変数にメモリーを割り当てるには、その前に `sql_cursor` 型のカーソル変数（[第 4 章「データ型とホスト変数」](#)を参照）を宣言する必要があります。

オブジェクト・キャッシュにメモリーを割り当てるには、その前にホスト構造体を指すポインタおよびオプションの標識構造体を指すポインタを宣言する必要があります。

データベースへの接続が必ずアクティブである必要があります。

構文



キーワードおよびパラメータ

<i>db_name</i>	事前に CONNECT 文で確立されたデータベース接続の名前を含む NULL 終了記号付き文字列。これが省略されたり、空の文字列であったときは、デフォルトのデータベース接続とみなされます。
<i>host_variable</i>	データベース接続の名前を含むホスト変数。
<i>cursor_variable</i>	割り当てるカーソル変数。
<i>host_ptr</i>	オブジェクト型に対して OTT により生成されるホスト構造体へのポインタ、 <code>sql_context</code> 型のコンテキスト変数、 <code>OCIRowid</code> へのタイプ・ポインタの ROWID 変数、または LOB の型に対応する LOB ロケータ変数。
<i>ind_ptr</i>	標識構造体へのオプションのポインタ。

使用上の注意

カーソルは静的ですが、カーソル変数は特定の問合せに結び付けられていないため動的です。カーソル変数は、型の互換性のある任意の問合せに対してオープンできます。

この文の詳細は、『Oracle8i PL/SQL ユーザーズ・ガイドおよびリファレンス』および『Oracle8i SQL リファレンス』を参照してください。

例

この部分的な例では、Pro*C/C++ プログラムで ALLOCATE 文を使用する方法を示します。

```

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;  
  ...  
}
```

関連項目

F-16 ページの [CACHE FREE ALL](#)（実行可能埋込み SQL 拡張要素）

F-18 ページの [CLOSE](#)（実行可能埋込み SQL）

F-51 ページの [EXECUTE](#)（実行可能埋込み SQL）

F-57 ページの [FETCH](#)（実行可能埋込み SQL）

F-62 ページの [FETCH DESCRIPTOR](#)（実行可能埋込み SQL）

F-62 ページの [FREE](#)（実行可能埋込み SQL 拡張要素）

ALLOCATE DESCRIPTOR（実行可能埋込み SQL）

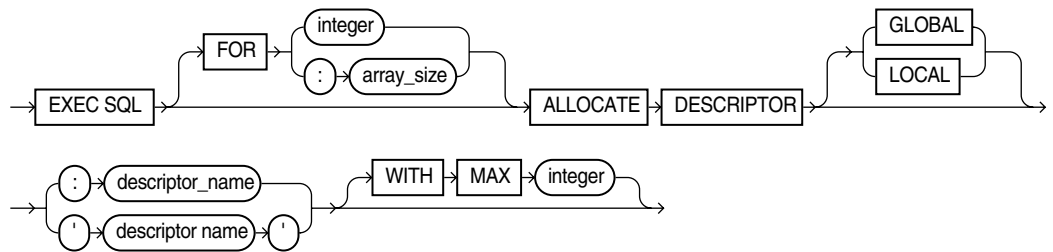
用途

記述子を割り当てる ANSI 動的 SQL 文。

前提条件

なし。

構文



キーワードおよびパラメータ

<i>array_size</i> <i>integer</i>	処理する行数を含むホスト変数。処理する行数。
<i>descriptor_name</i> <i>descriptor name</i>	ANSI 記述子の名前を含むホスト変数。ANSI 記述子の名前。
GLOBAL LOCAL	GLOBAL はアプリケーションの有効範囲を示し、LOCAL (デフォルト) は ファイルの有効範囲を示します。
WITH MAX <i>integer</i>	ホスト変数の最大数。デフォルトは 100 です。

使用上の注意

DYNAMIC=ANSI プリコンパイラ・オプションを使用します。この文の使用方法は、14-12 ページの「[ALLOCATE DESCRIPTOR](#)」を参照してください。

例

```
EXEC SQL FOR :batch ALLOCATE DESCRIPTOR GLOBAL :binddes WITH MAX 25 ;
```

関連項目

F-47 ページの [DESCRIBE DESCRIPTOR](#) (実行可能埋込み SQL)

F-34 ページの [DEALLOCATE DESCRIPTOR](#) (埋込み SQL 文)

F-63 ページの [GET DESCRIPTOR](#) (実行可能埋込み SQL)

F-103 ページの [SET DESCRIPTOR](#) (実行可能埋込み SQL)

CACHE FREE ALL（実行可能埋込み SQL 拡張要素）

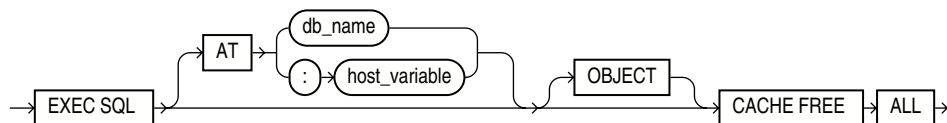
用途

オブジェクト・キャッシュ内のすべてのメモリーを解放します。

前提条件

データベースへの接続がアクティブである必要があります。

構文



キーワードおよびパラメータ

db_name 事前に CONNECT 文で確立されたデータベース接続の名前を含む NULL 終了記号付き文字列。これが省略されたり、空の文字列であったときは、デフォルトのデータベース接続とみなされます。

host_variable データベース接続の名前を含むホスト変数。

使用上の注意

接続カウントが 0（ゼロ）になると、SQLLIB により自動的にすべてのオブジェクト・キャッシュ・メモリーが解放されます。詳細は、17-6 ページの「[CACHE FREE ALL](#)」を参照してください。

例

```
EXEC SQL AT mydb CACHE FREE ALL ;
```

関連項目

F-12 ページの [ALLOCATE](#)（実行可能埋込み SQL 拡張要素）

F-62 ページの [FREE](#)（実行可能埋込み SQL 拡張要素）

CALL（実行可能埋込み SQL）

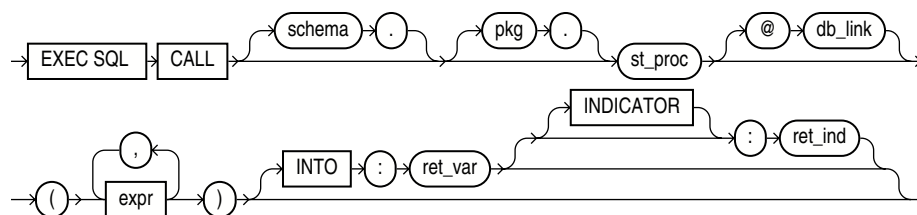
用途

ストアド・プロシージャをコールします。

前提条件

データベースへの接続がアクティブである必要があります。

構文



キーワードおよびパラメータ

<i>schema</i>	プロシージャを含むスキーマ。スキーマを省略すると、Oracle8iではプロシージャが自分のスキーマ内にあるとみなされます。
<i>pkg</i>	プロシージャが格納されているパッケージ。
<i>st_proc</i>	コールされるストアド・プロシージャ。
<i>db_link</i>	プロシージャがあるリモート・データベースへのデータベース・リンクの完全名または部分名。データベース・リンクの参照の詳細は、『Oracle8i SQL リファレンス』を参照してください。
<i>expr</i>	プロシージャのパラメータになる式のリスト。
<i>ret_var</i>	ファンクションの戻り値を受け取るホスト変数。
<i>ret_ind</i>	<i>ret_var</i> の標識変数。

使用上の注意

この文の詳細は、7-21 ページの「[ストアド PL/SQL または Java サブプログラムのコール](#)」を参照してください。

ストアド・プロシージャの完全な説明は、『Oracle8i アプリケーション開発者ガイド 基礎編』の「外部ルーチン」の章を参照してください。

例

```
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 */
...
```

関連項目

なし

CLOSE（実行可能埋込み SQL）

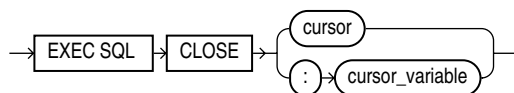
用途

カーソルのオープン時に取得したリソースを解放し、解析ロックを解除して、カーソルを使用禁止にします。

前提条件

MODE=ANSI の場合は、カーソルまたはカーソル変数はオープンである必要があります。

構文



キーワードおよびパラメータ

<i>cursor</i>	クローズするカーソル。
<i>cursor_variable</i>	クローズするカーソル変数。

使用上の注意

クローズしたカーソルからは行をフェッチできません。カーソルをリオープンするには、そのカーソルがクローズされている必要はありません。HOLD_CURSOR および RELEASE_CURSOR のプリコンパイラ・オプションにより CLOSE 文の機能が変更されます。これらのオプションの詳細は、[第 10 章「プリコンパイラのオプション」](#)を参照してください。

例

この例では、CLOSE 文の使用方法を示します。

```
EXEC SQL CLOSE emp_cursor;
```

関連項目

F-93 ページの [PREPARE（実行可能埋込み SQL）](#)

F-35 ページの [DECLARE CURSOR（埋込み SQL 宣言文）](#)

F-89 ページの [OPEN（実行可能埋込み SQL）](#)

COLLECTION APPEND（実行可能埋込み SQL 拡張要素）

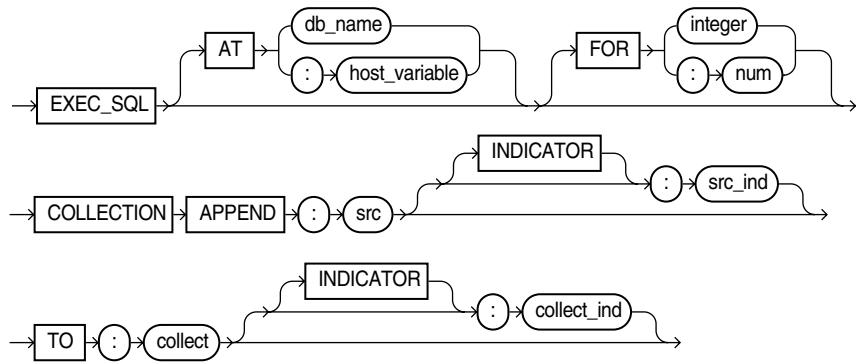
用途

1 つのコレクションの要素を別のコレクションの最後に追加します。

前提条件

NULL コレクションに追加すること、またはコレクションの上限を超えて追加することはできません。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、18-11 ページの「[COLLECTION APPEND](#)」を参照してください。

関連項目

他の COLLECTION 文を参照してください。

COLLECTION DESCRIBE（実行可能埋込み SQL 拡張要素）

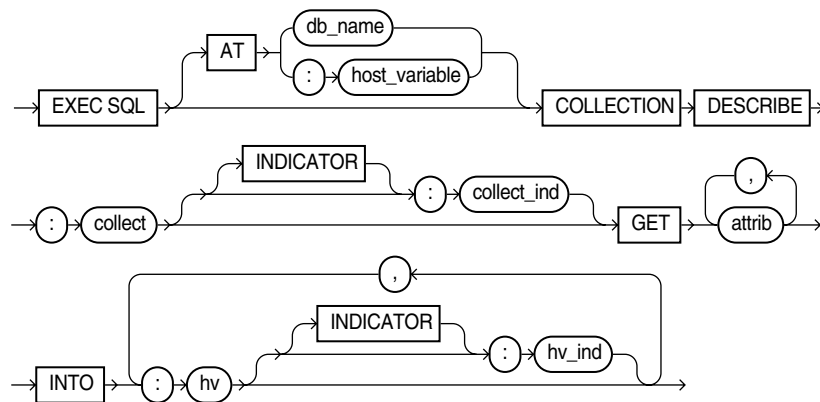
用途

コレクションについての情報を取得します。

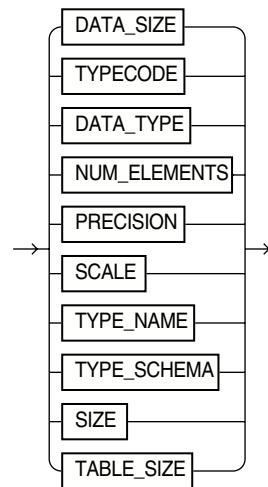
前提条件

ALLOCATE および OBJECT GET 文を使用して記述子を割り当て、記述子にコレクション属性を格納します。

構文



attrib は次のとおりです。



使用上の注意

使用上の注意、キーワード、パラメータ、例は、18-13 ページの「[COLLECTION DESCRIBE](#)」を参照してください。

関連項目

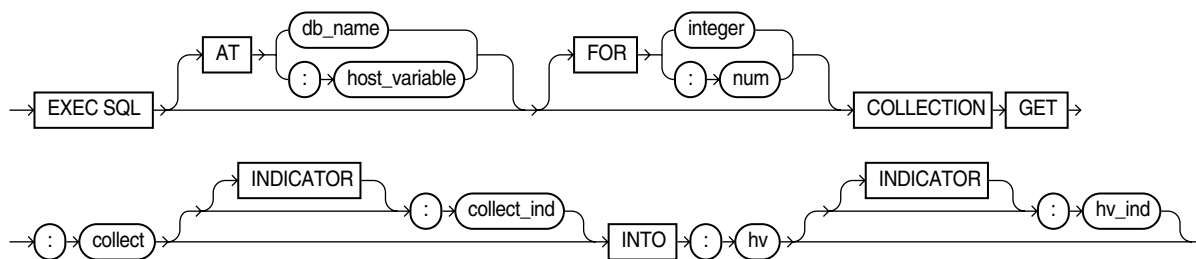
他の COLLECTION 文を参照してください。

COLLECTION GET（実行可能埋込み SQL 拡張要素）

用途

コレクションの要素を取得します。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、18-7 ページの「[COLLECTION GET](#)」を参照してください。

関連項目

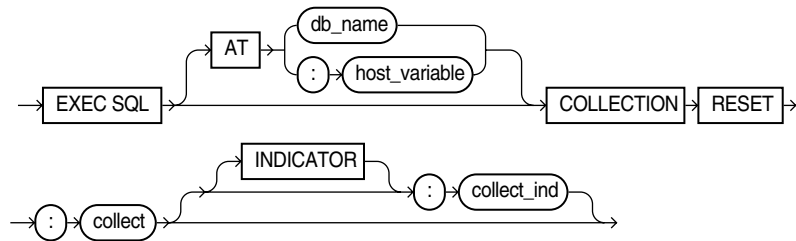
他の COLLECTION 文を参照してください。

COLLECTION RESET（実行可能埋込み SQL 拡張要素）

用途

コレクションのスライス・エンドポイントをコレクションの最初にリセットします。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、18-11 ページの「[COLLECTION RESET](#)」を参照してください。

関連項目

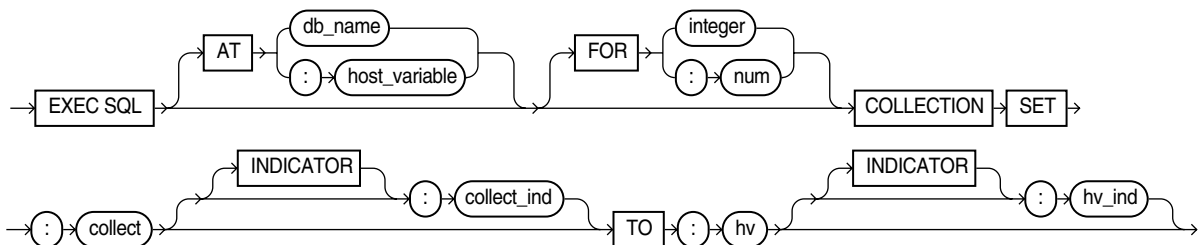
他の `COLLECTION` 文を参照してください。

COLLECTION SET（実行可能埋込み SQL 拡張要素）

用途

コレクションのカレント・スライスの要素値を更新します。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、18-9 ページの「[COLLECTION SET](#)」を参照してください。

関連項目

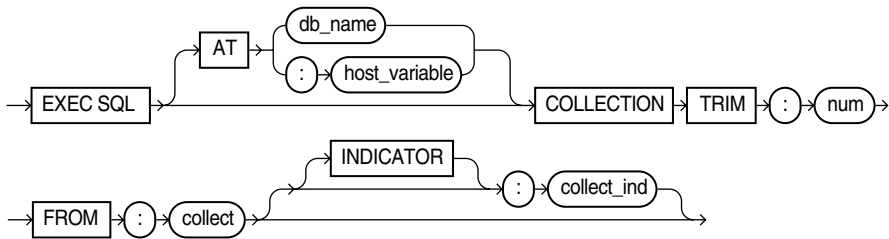
他の COLLECTION 文を参照してください。

COLLECTION TRIM（実行可能埋込み SQL 拡張要素）

用途

コレクションの最後から要素を削除します。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、18-12 ページの「COLLECTION TRIM」を参照してください。

関連項目

他の COLLECTION 文を参照してください。

COMMIT（実行可能埋込み SQL）

用途

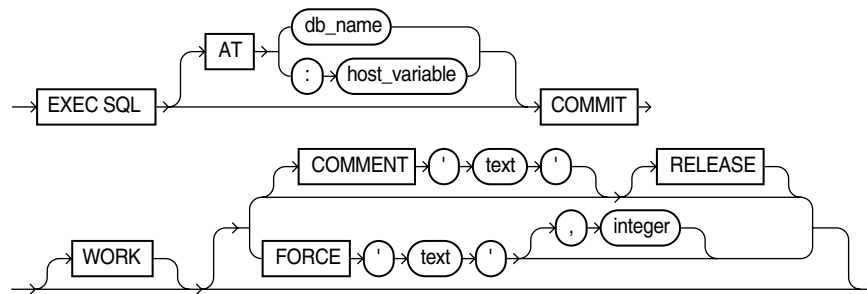
データベースの変更内容をすべて確定し、またオプションですべてのリソースを解放して切
断し、カレント・トランザクションを終了します。

前提条件

カレント・トランザクションをコミットするために必要な権限はありません。

自分でコミットしたインダウトの分散トランザクションを手動でコミットするには、FORCE TRANSACTION のシステム権限が必要です。他のユーザーがコミットしたインダウトの分散トランザクションを手動でコミットするには、FORCE ANY TRANSACTION のシステム権限が必要です。

構文



キーワードおよびパラメータ

AT	COMMIT 文の発行先のデータベースを指定します。次のいずれかを使用してデータベースを指定します。 <i>db_name</i> DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 <i>host_variable</i> 値が <i>db_name</i> のホスト変数。 この句を省略した場合、Oracle8i ではデフォルトのデータベースに対して文が発行されます。
WORK	標準 SQL に準拠してサポートされます。COMMIT 文および COMMIT WORK 文と同等です。
COMMENT	カレント・トランザクションに関連付けるコメントを指定します。'text' は最大 50 文字の引用符付きのリテラルで、トランザクションがインダウトになった場合に、Oracle8i によりデータ・ディクショナリ・ビュー DBA_2PC_PENDING にトランザクション ID とともに格納されます。
RELEASE	リソースをすべて解放し、アプリケーションをサーバーから切断します。

FORCE

インダウトの分散トランザクションを手動でコミットします。トランザクションは、ローカル・トランザクション ID またはグローバル・トランザクション ID を含む 'text' により指定します。これらのトランザクションの ID を検索するには、データ・ディクショナリ・ビュー DBA_2PC_PENDING に問い合わせます。また、オプションの整数を使用してトランザクションにシステム変更番号（SCN）を明示的に割り当てることができます。整数を省略した場合、トランザクションはカレント SCN を使用してコミットされます。

使用上の注意

プログラムの最後のトランザクションは、COMMIT 文または ROLLBACK 文と RELEASE オプションを使用して、必ず明示的にコミットまたはロールバックしてください。プログラムが異常終了すると、Oracle8i では自動的に変更内容がロールバックされます。

COMMIT 文は、ホスト変数やプログラムの制御の流れには影響しません。この文の詳細は、[第3章「データベースの概念」](#)を参照してください。

例

この例では、埋込み SQL COMMIT 文の使用方を示します。

```
EXEC SQL AT sales_db COMMIT RELEASE;
```

関連項目

F-96 ページの [ROLLBACK（実行可能埋込み SQL）](#)

F-99 ページの [SAVEPOINT（実行可能埋込み SQL）](#)

CONNECT（実行可能埋込み SQL 拡張要素）

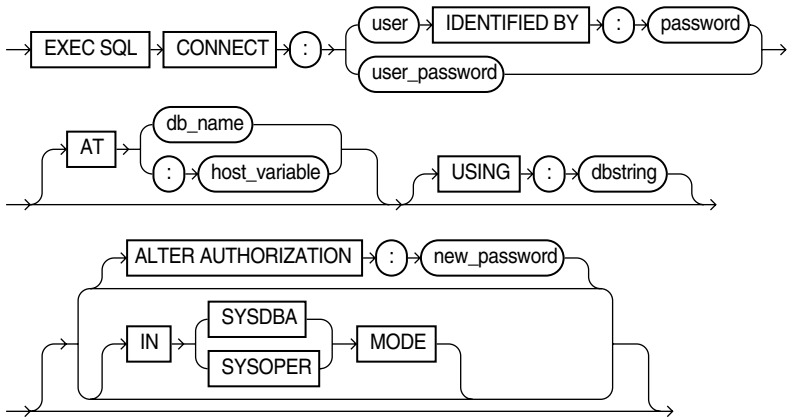
用途

データベースにログインします。

前提条件

指定するデータベースに対して CREATE SESSION のシステム権限が必要です。

構文



キーワードおよびパラメータ

<i>user</i> <i>password</i>	ユーザー名とパスワードを個別に指定します。
<i>user_password</i>	ユーザー名とパスワードをスラッシュ (/) で区切って格納した 1 つのホスト変数。 使用中のオペレーティング・システム経由の接続を Oracle8i で確認する場合は、 <i>:user_password</i> の値に「/」を指定します。
AT	接続先のデータベースを指定します。次のいずれかを使用してデータベースを指定します。 <i>db_name</i> DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 <i>:host_variable</i> 事前に宣言した <i>db_name</i> の値を持つホスト変数。
USING	デフォルト以外のデータベースへの接続に使用される Net8 データベースの指定文字列を指定します。この句を省略した場合は、デフォルトのデータベースに接続します。
ALTER AUTHORIZATION <i>new_password</i>	パスワードを次の文字列に変更します。 新しいパスワード。

IN SYSDBA MODE
SYSOPER MODE

SYSDBA または SYSOPER システム権限で接続します。
ALTER AUTHORIZATION が使用されている場合またはプリ
コンパイラ・オプション AUTO_CONNECT に YES が設定さ
れている場合は許可されません。

使用上の注意

プログラムは複数の接続を持つことができますが、デフォルトのデータベースには一回しか
接続できません。この文の詳細は、5-47 ページの「[埋込み（OCI リリース 7） Oracle コール](#)」
を参照してください。

例

次の例では、CONNECT の使用方法を示します。

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

この文では、*userid* の値に、「SCOTT/TIGER」のように、*username* の値と *password* の値を
スラッシュ (/) で区切って使用することもできます。

```
EXEC SQL CONNECT :userid ;
```

関連項目

F-24 ページの [COMMIT（実行可能埋込み SQL）](#)

F-37 ページの [DECLARE DATABASE（Oracle 埋込み SQL 宣言文）](#)

F-96 ページの [ROLLBACK（実行可能埋込み SQL）](#)

CONTEXT ALLOCATE（実行可能埋込み SQL 拡張要素）

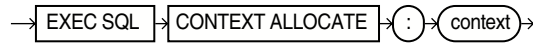
用途

EXEC SQL CONTEXT USE 文で参照する SQLLIB ランタイム・コンテキストを初期化しま
す。

前提条件

ランタイム・コンテキストは `sql_context` 型で宣言する必要があります。

構文



キーワードおよびパラメータ

context メモリーを割り当てる SQLLIB ランタイム・コンテキスト。

使用上の注意

マルチスレッド・アプリケーションでは、ランタイム・コンテキストごとにこの関数を実行します。

この文の詳細は、5-43 ページの「[OCI リリース 8 の SQLLIB 拡張相互運用性](#)」を参照してください。

例

この例では、Pro*C/C++ プログラムで CONTEXT ALLOCATE 文を使用する方法を示します。

```
EXEC SQL CONTEXT ALLOCATE :ctx1;
```

関連項目

F-29 ページの [CONTEXT FREE（実行可能埋込み SQL 拡張要素）](#)

F-30 ページの [CONTEXT USE（Oracle 埋込み SQL 宣言文）](#)

F-49 ページの [ENABLE THREADS（実行可能埋込み SQL 拡張要素）](#)

CONTEXT FREE（実行可能埋込み SQL 拡張要素）

用途

ランタイム・コンテキストに関連付けられているすべてのメモリーを解放し、ホスト・プログラム変数に NULL ポインタを代入します。

前提条件

CONTEXT FREE 文を使用して割り当てられているメモリーを解放する前に、指定のランタイム・コンテキストに CONTEXT ALLOCATE 文を使用してメモリーが割り当てられている必要があります。

構文

→ EXEC SQL → CONTEXT FREE → (:) → context →

キーワードおよびパラメータ

`:context` メモリーの割当てを解除する、割当て済のランタイム・コンテキスト。

使用上の注意

この文の詳細は、5-43 ページの「[OCI リリース 8 の SQLLIB 拡張相互運用性](#)」を参照してください。

例

この例では、Pro*C/C++ プログラムで CONTEXT FREE 文を使用する方法を示しています。

```
EXEC SQL CONTEXT FREE :ctx1;
```

関連項目

F-28 ページの [CONTEXT ALLOCATE](#)（実行可能埋込み SQL 拡張要素）

F-30 ページの [CONTEXT USE](#)（Oracle 埋込み SQL 宣言文）

F-49 ページの [ENABLE THREADS](#)（実行可能埋込み SQL 拡張要素）

CONTEXT OBJECT OPTION GET（実行可能埋込み SQL 拡張要素）

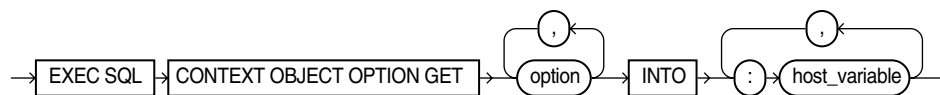
用途

CONTEXT OBJECT OPTION SET によって設定される使用中のコンテキストのオプションの値を決定します。

前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。

構文



キーワードおよびパラメータ

option オプション値は、表 17-1「CONTEXT OBJECT OPTION 値の有効な選択肢」に説明しています。

host_variable *option* リストと同じ順序で表された STRING、VARCHAR、CHARZ 型の出力変数

使用上の注意

17-17 ページの「CONTEXT OBJECT OPTION SET」を参照してください。

例

```

char EuroFormat[50];
...
EXEC SQL CONTEXT OBJECT OPTION GET DATEFORMAT INTO :EuroFormat ;
printf("Date format is %s\n", EuroFormat);

```

関連項目

F-12 ページの [CONTEXT ALLOCATE](#)（実行可能埋込み SQL 拡張要素）

F-29 ページの [CONTEXT FREE](#)（実行可能埋込み SQL 拡張要素）

F-31 ページの [CONTEXT OBJECT OPTION SET](#)（実行可能埋込み SQL 拡張要素）

F-30 ページの [CONTEXT USE](#)（Oracle 埋込み SQL 宣言文）

CONTEXT OBJECT OPTION SET（実行可能埋込み SQL 拡張要素）

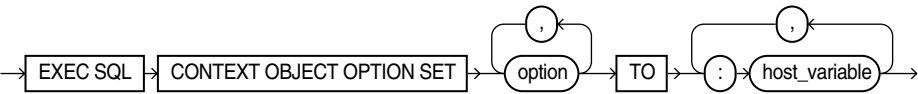
用途

Date 属性、すなわち使用中のコンテキストの DATEFORMAT, DATELANG、の指定値にオプションを設定します。

前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。

構文



キーワードおよびパラメータ

option オプション値は、表 17-1「CONTEXT OBJECT OPTION 値の有効な選択肢」に説明しています。

host_variable STRING、VARCHAR、CHARZ 型の入力変数。*option* リストと同じ順序。

使用上の注意

17-18 ページの「CONTEXT OBJECT OPTION GET」を参照してください。

例

```
char *new_format = "DD-MM-YYY";
char *new_lang = "French";
...
EXEC SQL CONTEXT OBJECT OPTION SET DATEFORMAT, DATELANG TO :new_format, :new_lang;
```

関連項目

- F-12 ページの [CONTEXT ALLOCATE](#) (実行可能埋込み SQL 拡張要素)
- F-29 ページの [CONTEXT FREE](#) (実行可能埋込み SQL 拡張要素)
- F-30 ページの [CONTEXT USE](#) (Oracle 埋込み SQL 宣言文)
- F-30 ページの [CONTEXT OBJECT OPTION SET](#) (実行可能埋込み SQL 拡張要素)

CONTEXT USE (Oracle 埋込み SQL 宣言文)

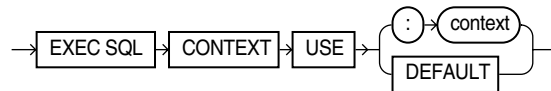
用途

後続の実行 SQL 文で指定の SQLLIB ランタイム・コンテキストを使用するようにプリコンパイラに指示します。

前提条件

CONTEXT USE 宣言文によって指定されたランタイム・コンテキストが事前に宣言されている必要があります。

構文



キーワードおよびパラメータ

context 後続の実行 SQL 文用を使用する割当て済のランタイム・コンテキスト。たとえば、使用するコンテキストを（複数のコンテキストを割り当てることができる）ソース・コードで指定すると、Oracle Server に接続してそのコンテキストの範囲内でデータベースを操作できます。DEFAULT は作業したグローバル・コンテキストが使用されることを示します。

DEFAULT グローバル・コンテキストを使用することを示します。

使用上の注意

この文は EXEC SQL INCLUDE または EXEC ORACLE OPTION などの宣言文では無効です。この文は、標準的な C の適用範囲の規則に従わず、特定のソース・ファイル内で後続するすべての実行 SQL 文に影響する点では、EXEC SQL WHENEVER 宣言文に似ています。

この文の詳細は、5-43 ページの「[OCI リリース 8 の SQLLIB 拡張相互運用性](#)」を参照してください。

例

この例では、Pro*C/C++ の埋込み SQL プログラムで CONTEXT USE 宣言文を使用する方法を示します。

```
EXEC SQL CONTEXT USE :ctx1;
```

関連項目

F-28 ページの [CONTEXT ALLOCATE](#) (実行可能埋込み SQL 拡張要素)

F-29 ページの [CONTEXT FREE](#) (実行可能埋込み SQL 拡張要素)

F-49 ページの [ENABLE THREADS](#) (実行可能埋込み SQL 拡張要素)

DEALLOCATE DESCRIPTOR（埋込み SQL 文）

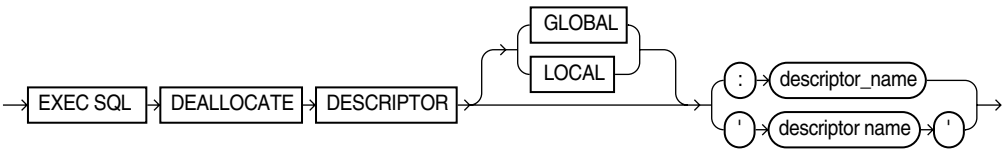
用途

ANSI 動的 SQL 文で記述子領域の割当てを解除してメモリーを解放します。

前提条件

DEALLOCATE DESCRIPTOR 文で指定された記述子は、事前に ALLOCATE DESCRIPTOR 文を使用して割り当てする必要があります。

構文



キーワードおよびパラメータ

GLOBAL | LOCAL GLOBAL はアプリケーションの有効範囲を示し、LOCAL（デフォルト）はファイルの有効範囲を示します。

descriptor_name 割り当てられた ANSI 記述子の名前が含まれるホスト変数。割り当てられた ANSI 記述子の名前。
'descriptor name'

使用上の注意

DYNAMIC=ANSI プリコンパイラ・オプションを使用します。

この文の詳細は、14-13 ページの「[DEALLOCATE DESCRIPTOR](#)」を参照してください。

例

```
EXEC SQL DEALLOCATE DESCRIPTOR GLOBAL 'SELDES' ;
```

関連項目

F-14 ページの [ALLOCATE DESCRIPTOR](#)（実行可能埋込み SQL）

F-46 ページの [DESCRIBE](#)（実行可能埋込み SQL 拡張要素）

F-63 ページの [GET DESCRIPTOR](#) (実行可能埋込み SQL)

F-93 ページの [PREPARE](#) (実行可能埋込み SQL)

F-103 ページの [SET DESCRIPTOR](#) (実行可能埋込み SQL)

DECLARE CURSOR (埋込み SQL 宣言文)

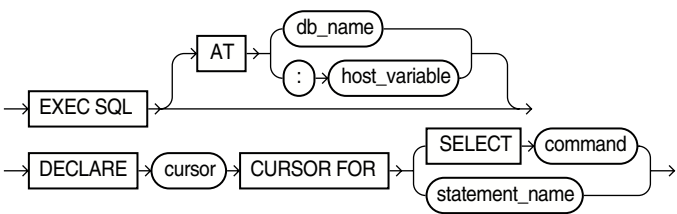
用途

カーソルに名前を付け、それを SQL 文または PL/SQL ブロックに対応付けて宣言します。

前提条件

SQL 文または PL/SQL ブロックの識別子を使用してカーソルに対応付けるには、DECLARE STATEMENT 文を使用してこの識別子を事前に宣言しておく必要があります。

構文



キーワードおよびパラメータ

AT	カーソルが宣言されるデータベースを指定します。次のいずれかを使用してデータベースを指定します。
<i>db_name</i>	DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。
<i>:host_variable</i>	事前に宣言した <i>db_name</i> の値を持つホスト変数。 この句を省略した場合、Oracle8i ではデフォルトのデータベースに対してこのカーソルが宣言されます。
<i>cursor</i>	宣言するカーソルの名前。

<code>SELECT statement</code>	カーソルに対応付ける SELECT 文。直後の文に INTO 句を含めないでください。
<code>statement_name</code>	カーソルに対応付ける SQL 文または PL/SQL ブロックを指定します。 <code>statement_name</code> または <code>block_name</code> は、DECLARE STATEMENT 文を使用して事前に宣言しておく必要があります。

使用上の注意

カーソルは、他の埋込み SQL 文で参照する前に、宣言する必要があります。カーソル宣言の適用範囲はプリコンパイル・ユニット内全体になるため、各カーソルの名前は適用範囲内で一意になるようにしてください。1つのプリコンパイル・ユニット内で同じ名前のカーソルを複数宣言することはできません。

カーソルは、UPDATE 文または DELETE 文の WHERE 句内で CURRENT OF 構文を使用して参照できます。このとき、カーソルは OPEN 文を使用してオープンし、FETCH 文を使用して行に位置付けられている必要があります。この文の詳細は、7-18 ページの「[埋込み PL/SQL のカーソルの使用方法](#)」を参照してください。

例

この例では、DECLARE CURSOR 文の使用方法を示します。

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

関連項目

- F-18 ページの [CLOSE \(実行可能埋込み SQL\)](#)
- F-37 ページの [DECLARE DATABASE \(Oracle 埋込み SQL 宣言文\)](#)
- F-38 ページの [DECLARE STATEMENT \(埋込み SQL 宣言文\)](#)
- F-42 ページの [DELETE \(実行可能埋込み SQL\)](#)
- F-57 ページの [FETCH \(実行可能埋込み SQL\)](#)
- F-89 ページの [OPEN DESCRIPTOR \(実行可能埋込み SQL\)](#)
- F-93 ページの [PREPARE \(実行可能埋込み SQL\)](#)
- F-100 ページの [SELECT \(実行可能埋込み SQL\)](#)
- F-107 ページの [UPDATE \(実行可能埋込み SQL\)](#)

DECLARE DATABASE (Oracle 埋込み SQL 宣言文)

用途

後続の埋込み SQL 文でアクセスされるデフォルト以外のデータベースの識別子を宣言します。

前提条件

デフォルト以外のデータベースのユーザー名にアクセスできる必要があります。

構文

```
→ EXEC SQL → DECLARE → db_name → DATABASE →
```

キーワードおよびパラメータ

db_name デフォルト以外のデータベースに対して設定する識別子。

使用上の注意

デフォルト以外のデータベースに対して *db_name* を宣言するのは、他の埋込み SQL 文が AT 句を使用してそのデータベースを参照できるようにするためです。AT 句を指定して CONNECT 文を発行する前に、DECLARE DATABASE 文を使用してデフォルト以外のデータベースに対して *db_name* を宣言する必要があります。

この文の詳細は、3-8 ページの「[単一の明示的接続](#)」を参照してください。

例

この例では、DECLARE DATABASE 宣言文の使用方法を示します。

```
EXEC SQL DECLARE oracle3 DATABASE ;
```

関連項目

F-24 ページの [COMMIT](#) (実行可能埋込み SQL)

F-26 ページの [CONNECT](#) (実行可能埋込み SQL 拡張要素)

F-35 ページの [DECLARE CURSOR](#) (埋込み SQL 宣言文)

F-38 ページの [DECLARE STATEMENT](#) (埋込み SQL 宣言文)

F-42 ページの [DELETE](#) (実行可能埋込み SQL)

- F-51 ページの EXECUTE... END-EXEC (実行可能埋込み SQL 拡張要素)
- F-55 ページの EXECUTE IMMEDIATE (実行可能埋込み SQL)
- F-65 ページの INSERT (実行可能埋込み SQL)
- F-100 ページの SELECT (実行可能埋込み SQL)
- F-107 ページの UPDATE (実行可能埋込み SQL)

DECLARE STATEMENT (埋込み SQL 宣言文)

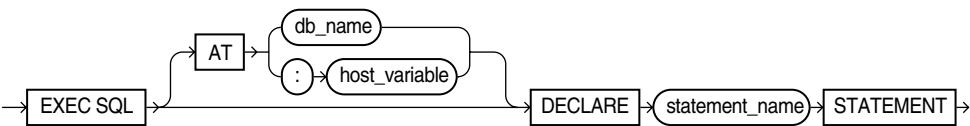
用途

SQL 文または PL/SQL ブロックの識別子を宣言し、他の埋込み SQL 文で使えるようにします。

前提条件

なし。

構文



キーワードおよびパラメータ

AT	SQL 文または PL/SQL ブロックが宣言されるデータベースを識別します。次のいずれかを使用してデータベースを指定します。
db_name	DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。
host_variable	値が db_name のホスト変数。 この句を省略した場合、Oracle8i ではデフォルトのデータベースに対して SQL 文または PL/SQL ブロックが宣言されます。
statement_name	文に対して宣言する識別子。

使用上の注意

DECLARE STATEMENT 文を使用して SQL 文または PL/SQL ブロックの識別子を宣言する必要があるのは、その識別子を参照する DECLARE CURSOR 文の埋込み SQL プログラム内での位置が、文またはブロックを解析して識別子と対応付ける PREPARE 文よりも物理的に（論理的ではなく）前になっているときのみです。

文の宣言の適用範囲は、カーソルの宣言と同様に、プリコンパイル・ユニット内全体です。この文の詳細は、[第 4 章「データ型とホスト変数」](#) および [第 13 章「Oracle の動的 SQL」](#) を参照してください。

例 I

この例では、DECLARE STATEMENT 文の使用方を示します。

```
EXEC SQL AT remote_db DECLARE my_statement STATEMENT;  
EXEC SQL PREPARE my_statement FROM :my_string;  
EXEC SQL EXECUTE my_statement;
```

例 II

この Pro*C/C++ の埋込み SQL プログラムからの例では、DECLARE CURSOR 文が PREPARE 文の前にあるので、DECLARE STATEMENT 文が必要です。

```
EXEC SQL DECLARE my_statement STATEMENT;  
EXEC SQL DECLARE emp_cursor CURSOR FOR my_statement;  
EXEC SQL PREPARE my_statement FROM :my_string;  
...
```

関連項目

F-18 ページの [CLOSE \(実行可能埋込み SQL\)](#)

F-37 ページの [DECLARE DATABASE \(Oracle 埋込み SQL 宣言文\)](#)

F-57 ページの [FETCH \(実行可能埋込み SQL\)](#)

F-89 ページの [OPEN DESCRIPTOR \(実行可能埋込み SQL\)](#)

F-93 ページの [PREPARE \(実行可能埋込み SQL\)](#)

DECLARE TABLE (Oracle 埋込み SQL 宣言文)

用途

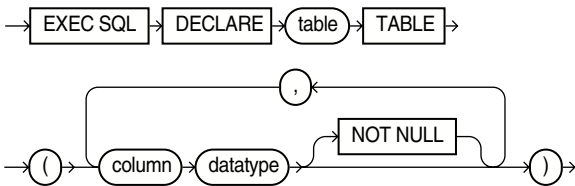
それぞれの列のデータ型、デフォルト値、Oracle プリコンパイラによる意味検査のための NULL または NOT NULL 仕様部など、表またはビューの構造を定義します。

前提条件

なし。

構文

リレーショナル表に使用する構文は、次のとおりです。



オブジェクト表に使用する構文は、次のとおりです。



キーワードおよびパラメータ

<i>table</i>	宣言した表の名前。
<i>column</i>	表の列。
<i>datatype</i>	列のデータ型。データ型の詳細は、4-2 ページの「 Oracle のデータ型 」を参照してください。 データ型がユーザー定義オブジェクトの場合は、 <i>size</i> をカッコで囲んで入力できます。 <i>size</i> はマクロまたは複合 C 表現にはできません。 <i>size</i> は省略できます。例を参照してください。
NOT NULL	NULL を含めることのできない列を指定します。
<i>obj_type</i>	オブジェクト型を表します。

使用上の注意

この文の使用方法は、D-4 ページの「[DECLARE TABLE の使用](#)」を参照してください。

例

次の文により、PARTNO、BIN、QTY の列を持つ PARTS 表が宣言されます。

```
EXEC SQL DECLARE parts TABLE
      (partno NUMBER NOT NULL,
       bin    NUMBER,
       qty    NUMBER);
```

次のようにオブジェクト型を使用します。

```
EXEC SQL DECLARE person TYPE AS OBJECT (name VARCHAR2(20), age INT);
EXEC SQL DECLARE odjtab1 TABLE OF person;
```

関連項目

F-41 ページの [DECLARE TYPE \(Oracle 埋込み SQL 宣言文\)](#)

DECLARE TYPE (Oracle 埋込み SQL 宣言文)

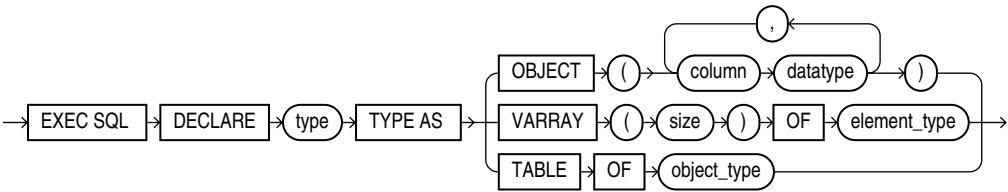
用途

プリコンパイラによる意味検査に使用される型の属性を定義します。

前提条件

なし。

構文



キーワードおよびパラメータ

<i>column</i>	列の名前。
<i>datatype</i>	列のデータ型。

<i>size</i>	VARRAY の要素数。
<i>element_type</i>	VARRAY の要素型。オブジェクトにすることもできます。
<i>object_type</i>	以前に宣言されていたオブジェクト型。

使用上の注意

この文の使用方法は、D-5 ページの「[DECLARE TYPE の使用](#)」を参照してください。

例

```
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 ;
```

関連項目

F-39 ページの [DECLARE TABLE（Oracle 埋込み SQL 宣言文）](#)

DELETE（実行可能埋込み SQL）

用途

表またはビューのベース表から行を削除します。

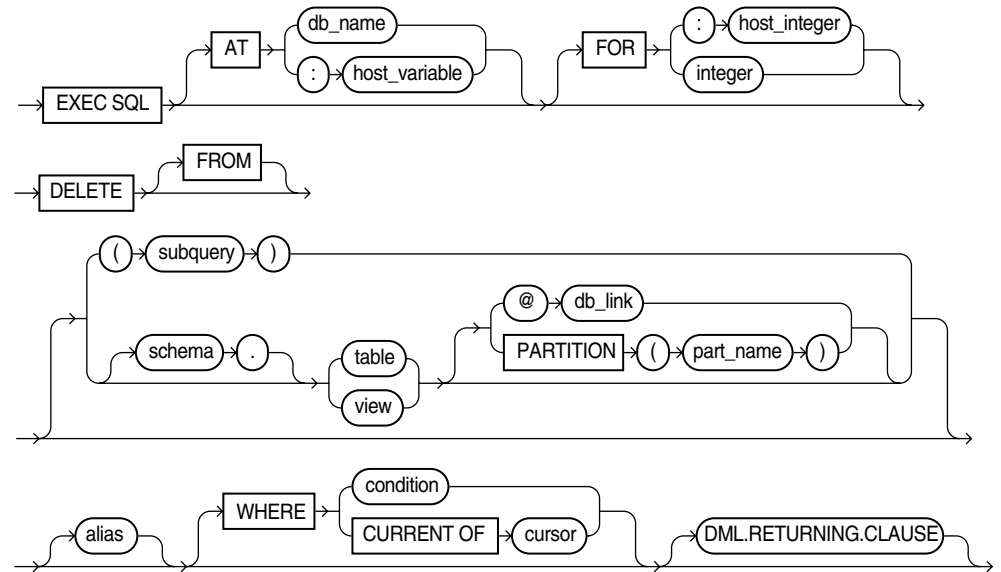
前提条件

表から行を削除するには、表が自分のスキーマ内にあるか、表に対して DELETE の権限を持つ必要があります。

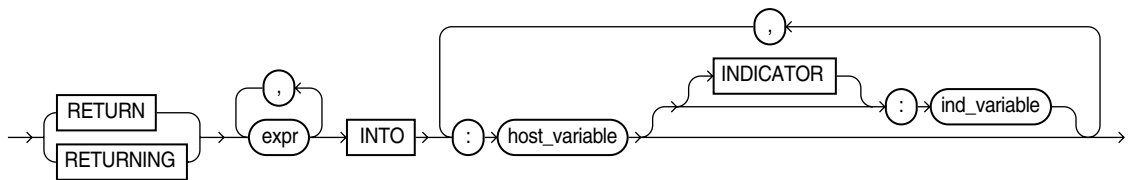
ビューのベース表から行を削除するには、ビューが属するスキーマの所有者が、ベース表に対して DELETE の権限を持つ必要があります。また、ビューが自分のスキーマ以外のスキーマ内にある場合は、ビューに対する DELETE の権限を付与されている必要があります。

DELETE ANY TABLE のシステム権限では、どの表またはビューのベース表からでも行を削除できます。

構文



DML 戻り句は次のとおりです。



キーワードおよびパラメータ

AT

SELECT 文の発行先のデータベースを識別します。次のいずれかを使用してデータベースを指定します。

db_name

DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。

host_variable

事前に宣言した *db_name* の値を持つホスト変数。

	この句を省略した場合、DELETE 文はデフォルトのデータベースに対して発行されます。
FOR <i>:host_integer</i>	WHERE 句に配列ホスト変数が含まれる場合に、文を実行する回数を制限します。この句を省略すると、Oracle8i では最小の配列の各コンポーネントについて 1 回ずつ文が実行されます。
<i>subquery</i>	対応する列に割り当てられた新しい値を戻す副問合せ。副問合せの構文については、『Oracle8i SQL リファレンス』の「SELECT」を参照してください。
<i>schema</i>	表またはビューを含むスキーマ。schema を省略した場合、Oracle8i では表またはビューが独自のスキーマにあるとみなされます。
<i>table</i>	行を削除する表の名前。
<i>view</i>	ビューの名前。Oracle8i ではビューのベース表から行が削除されます。
<i>db_link</i>	表またはビューがあるリモート・データベースへのデータベース・リンクの完全名または部分名。データベース・リンクの参照の詳細は、『Oracle8i SQL リファレンス』を参照してください。リモートの表またはビューから行を削除できるのは、Oracle8i を分散オプションで使用している場合に限られます。 dblink を省略した場合、Oracle8i では表またはビューがローカル・データベース内にあるとみなされます。
<i>part_name</i>	表のパーティション。
<i>alias</i>	表に割り当てられている別名。別名は一般に、DELETE 文で相関問合せとともに使用します。
WHERE	削除される行を指定します。 <i>condition</i> 条件を満たす行のみを削除します。条件には、ホスト変数およびオプションの標識変数を使用できます。『Oracle8i SQL リファレンス』の条件の構文の説明を参照してください。 CURRENT OF <i>cursor</i> <i>cursor</i> によって最後にフェッチされた行のみを削除します。FOR UPDATE 句が明確に 1 つの表のみをロックしていない限り、 <i>cursor</i> は結合を実行する SELECT 文に対応付けることができません。 この句を完全に省略した場合、Oracle8i では表またはビューからすべての行が削除されます。
DML 戻り句	6-10 ページの「 DML 戻り句 」を参照してください。

使用上の注意

WHERE 句のホスト変数は、すべてスカラーか、すべて配列である必要があります。変数がスカラーの場合、Oracle8i では DELETE 文が 1 回のみ実行されます。配列の場合、Oracle8i では配列のコンポーネント・セットごとに 1 回ずつこの文が実行されます。1 回の実行で 0 行または 1 行、複数行を削除できます。

WHERE 句の配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracle8i で文が実行される回数は、次の値のうち小さい方によって決まります。

- 最小の配列のサイズ
- オプションの FOR 句の *:host_integer* の値

この条件を満たす行が存在しない場合、行は削除されず、SQLCODE は NOT_FOUND 条件を戻します。

削除された行の累積数は SQLCA を介して戻されます。WHERE 句に配列ホスト変数が指定されていると、DELETE 文によって処理された配列のすべてのコンポーネントにおよぶ削除行数の合計がこの値に設定されます。

条件を満たす行がない場合、Oracle8i により SQLCA の SQLCODE を介してエラーが戻されます。WHERE 句を省略した場合、Oracle8i により SQLCA の SQLWARN の第 5 コンポーネントに警告フラグが設定されます。この文と SQLCA の詳細は、[第 9 章「ランタイム・エラーの処理」](#)を参照してください。

DELETE 文においてコメントを使用して、指示やヒントをオプティマイザに引き渡すことができます。オプティマイザはヒントを使用して文の実行計画を選択します。ヒントの詳細は、『Oracle8i パフォーマンスのための設計およびチューニング』を参照してください。

例

この例では、Pro*C/C++ の埋込み SQL プログラムにおける DELETE 文の使用法を示します。

```
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;
```

関連項目

F-37 ページの [DECLARE DATABASE](#)（Oracle 埋込み SQL 宣言文）

F-38 ページの [DECLARE STATEMENT](#)（埋込み SQL 宣言文）

DESCRIBE（実行可能埋込み SQL 拡張要素）

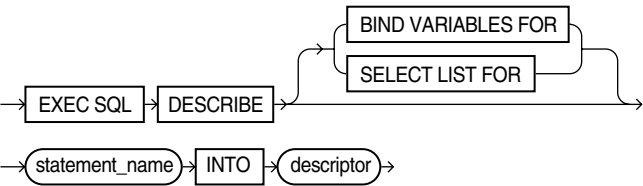
用途

Oracle 記述子に動的 SQL 文または PL/SQL ブロックについての情報を入力します。

前提条件

埋込み SQL の PREPARE 文を使用して、SQL 文または PL/SQL ブロックを事前に準備しておく必要があります。

構文



キーワードおよびパラメータ

BIND VARIABLES FOR	SQL 文または PL/SQL ブロックの入力変数に関する情報を保持する記述子を初期化します。
SELECT LIST FOR	SELECT 文の選択リストに関する情報が含まれる記述子を初期化します。 デフォルトは SELECT LIST FOR です。
<i>statement_name</i>	PREPARE 文を使用して事前に準備した SQL 文または PL/SQL ブロックを指定します。
<i>descriptor</i>	入力する記述子の名前。

使用上の注意

埋込み SQL プログラム内のバインド記述子または選択記述子进行操作するには、その前に DESCRIBE 文を発行する必要があります。

入力変数と出力変数の両方を同じ記述子に記述することはできません。

DESCRIBE 文で検出される変数の数は、一意に名前が指定されたプレースホルダの合計数ではなく、準備する SQL 文または PL/SQL ブロックのプレースホルダの合計数です。この文の詳細は、[第 13 章「Oracle の動的 SQL」](#)を参照してください。

例

この例では、Pro*C/C++ の埋込み SQL プログラムにおける DESCRIBE 文の使用法を示します。

```
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;
```

関連項目

F-93 ページの [PREPARE \(実行可能埋込み SQL\)](#)

DESCRIBE DESCRIPTOR (実行可能埋込み SQL)

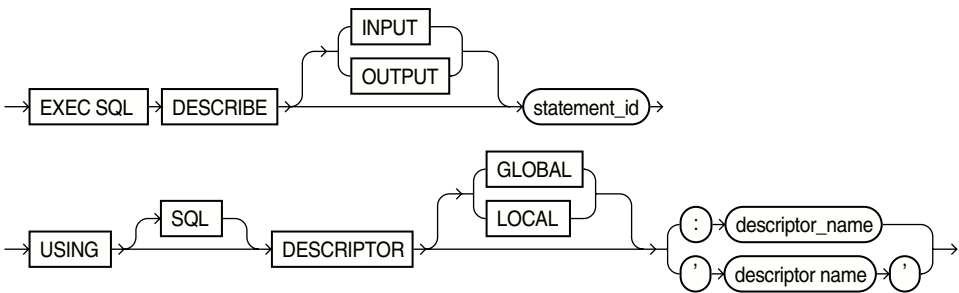
用途

SQL 文についての情報を取得するために使用する ANSI 動的 SQL 文で、情報は記述子に保管されます。

前提条件

埋込み SQL の PREPARE 文を使用して、SQL 文を事前に準備しておく必要があります。

構文



キーワードおよびパラメータ

<i>statement_id</i>	事前に準備された SQL 文または PL/SQL ブロックの名前。デフォルトは OUTPUT です。
<i>desc_nam</i>	SQL 文についての情報が含まれる記述子の名前が入るホスト変数。
<i>'descriptor name'</i>	記述子の名前。
GLOBAL LOCAL	GLOBAL はアプリケーションの有効範囲を示し、LOCAL（デフォルト）はファイルの有効範囲を示します。

使用上の注意

DYNAMIC=ANSI プリコンパイラ・オプションを使用します。

INPUT 記述子に使用できるのは、COUNT と NAME に限られます。

DESCRIBE 文で検出される変数の数は、一意に名前が指定されたプレースホルダの合計数ではなく、準備する SQL 文または PL/SQL ブロックのプレースホルダの合計数です。この文の詳細は、14-21 ページの「[DESCRIBE INPUT](#)」および14-21 ページの「[DESCRIBE OUTPUT](#)」を参照してください。

例

```
EXEC SQL PREPARE s FROM :my_stament;
EXEC SQL DESCRIBE INPUT s USING DESCRIPTOR 'in' ;
```

関連項目

- F-14 ページの [ALLOCATE DESCRIPTOR（実行可能埋込み SQL）](#)
- F-34 ページの [DEALLOCATE DESCRIPTOR（埋込み SQL 文）](#)

F-63 ページの [GET DESCRIPTOR](#)（実行可能埋込み SQL）

F-93 ページの [PREPARE](#)（実行可能埋込み SQL）

F-103 ページの [SET DESCRIPTOR](#)（実行可能埋込み SQL）

ENABLE THREADS（実行可能埋込み SQL 拡張要素）

用途

複数のスレッドをサポートするプロセスを初期化します。

前提条件

マルチスレッド・アプリケーションをサポートするプラットフォームでプリコンパイラ・アプリケーションを開発し、コンパイルしている必要があります。また、コマンドラインで `THREADS=YES` と指定する必要があります。

構文

→ EXEC SQL → ENABLE THREADS →

キーワードおよびパラメータ

なし。

使用上の注意

ENABLE THREADS 文は、他の実行 SQL 文の前、かつスレッドを作成する前に実行する必要があります。この文では、ホスト変数を指定する必要はありません。

この文の詳細は、5-43 ページの「[OCI リリース 8 の SQLLIB 拡張相互運用性](#)」を参照してください。

例

この例では、Pro*C/C++ プログラムで ENABLE THREADS 文を使用する方法を示しています。

```
EXEC SQL ENABLE THREADS;
```

関連項目

F-28 ページの [CONTEXT ALLOCATE](#)（実行可能埋込み SQL 拡張要素）

F-29 ページの [CONTEXT FREE（実行可能埋込み SQL 拡張要素）](#)

F-30 ページの [CONTEXT USE（Oracle 埋込み SQL 宣言文）](#)

EXECUTE... END-EXEC（実行可能埋込み SQL 拡張要素）

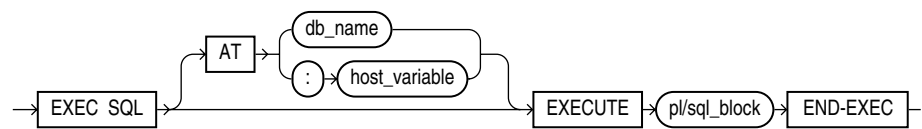
用途

Pro*C/C++ プログラムに無名 PL/SQL ブロックを埋め込みます。

前提条件

なし。

構文



キーワードおよびパラメータ

AT	PL/SQL ブロックが実行されるデータベースを指定します。次のいずれかを使用してデータベースを指定します。 <i>db_name</i> DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 <i>host_variable</i> 事前に宣言した <i>db_name</i> の値を持つホスト変数。 この句を省略した場合、PL/SQL ブロックはデフォルトのデータベースに対して実行されます。
<i>pl/sql_block</i>	PL/SQL ブロックの作成方法を含む PL/SQL の詳細は、『Oracle8i PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。
END-EXEC	このキーワードは、Oracle プリコンパイラ・プログラムが使用するプログラミング言語にかかわらず、埋込み PL/SQL ブロックの後に配置する必要があります。キーワード END-EXEC の後には、C/C++ の文終了記号「;」を付ける必要があります。

使用上の注意

Pro*C/C++ では埋込み PL/SQL ブロックが 1 つの埋込み SQL 文のように扱われるため、PL/SQL ブロックはプログラムで SQL 文を埋め込める場所であればどこでも埋込みが可能です。Oracle プリコンパイラ・プログラムでの PL/SQL ブロックの埋込みの詳細は、[第 7 章「埋込み PL/SQL」](#) を参照してください。

例

この EXECUTE 文を Pro*C/C++ プログラムに使用すると、PL/SQL ブロックがプログラムに埋め込まれます。

```
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;
```

関連項目

F-55 ページの [EXECUTE IMMEDIATE（実行可能埋込み SQL）](#)

EXECUTE（実行可能埋込み SQL）

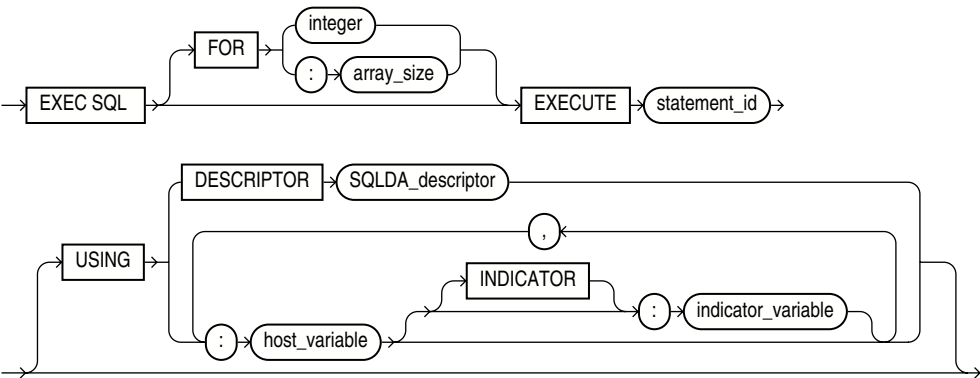
用途

Oracle 動的 SQL では、埋込み SQL の PREPARE 文によって準備済の DELETE 文、INSERT 文または UPDATE 文、あるいは PL/SQL ブロックを実行します。ANSI 動的 SQL メソッド 4 については、F-53 ページの「[EXECUTE DESCRIPTOR（実行可能埋込み SQL）](#)」を参照してください。

前提条件

埋込み SQL の PREPARE 文を使用して、SQL 文または PL/SQL ブロックを先に準備しておく必要があります。

構文



キーワードおよびパラメータ

FOR :array_size	処理する行を含むホスト変数。
FOR integer	処理する行数。
	USING 句に配列ホスト変数が含まれる場合にこの文が実行される回数を制限します。この句を省略する場合は、Oracle8i より最小の配列の各コンポーネントに対し 1 回この文が実行されます。
statement_id	実行する SQL 文または PL/SQL ブロックに対応付けられているプリコンパイラ識別子。プリコンパイラ識別子を文または PL/SQL ブロックに対応付けるには、埋込み SQL の PREPARE 文を使用します。
USING DESCRIPTOR SQLDA_descriptor	Oracle 識別子を使用します。ANSI 識別子（INTO 句）とともに使用することはできません。
USING	オプションの標識変数を使用してホスト変数のリストを指定します。Oracle8i では実行する文にこれらの変数が入力変数として代入されます。ホスト変数および標識変数は、すべてスカラーか、すべて配列である必要があります。
host_variable	ホスト変数。
indicator_variable	標識変数。

使用上の注意

この文の詳細は、Oracle バージョンの第 13 章「Oracle の動的 SQL」を参照してください。

例

この例では、Pro*C/C++ プログラムで EXECUTE 文を使用する方法を示しています。

```
EXEC SQL PREPARE my_statement
      FROM :my_string;
EXEC SQL EXECUTE my_statement
      USING :my_var;
```

関連項目

F-37 ページの [DECLARE DATABASE](#) (Oracle 埋込み SQL 宣言文)

F-93 ページの [PREPARE](#) (実行可能埋込み SQL)

EXECUTE DESCRIPTOR (実行可能埋込み SQL)

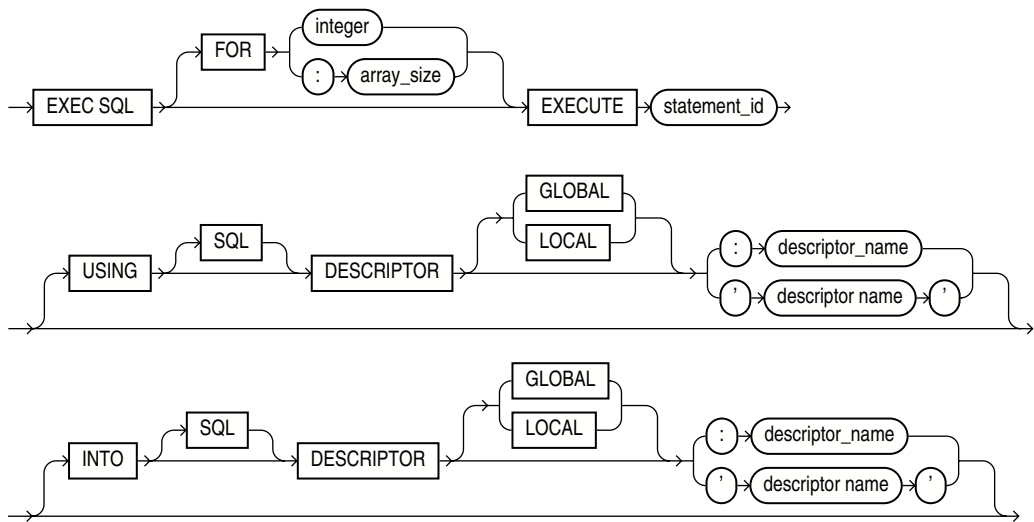
用途

ANSI SQL 方法 4 では、埋込み SQL の PREPARE 文によって準備済の DELETE 文、INSERT 文または UPDATE 文、あるいは PL/SQL ブロックを実行します。

前提条件

埋込み SQL の PREPARE 文を使用して、SQL 文または PL/SQL ブロックを先に準備しておく必要があります。

構文



キーワードおよびパラメータ

FOR :array_size	処理する行を含むホスト変数。
FOR=integer	処理する行数。 文が実行される回数を制限します。Oracle8i では、最小の配列の各コンポーネントに対して一回ずつ文が実行されます。
statement_id	実行する SQL 文または PL/SQL ブロックに対応付けられているプリコンパイラ識別子。プリコンパイラ識別子を文または PL/SQL ブロックに対応付けるには、埋込み SQL の PREPARE 文を使用します。
GLOBAL LOCAL	GLOBAL はアプリケーションの有効範囲を示し、LOCAL (デフォルト) は ファイルの有効範囲を示します。
USING	ANSI 記述子。
descriptor_name	入力記述子の名前が含まれるホスト変数。
'descriptor name'	入力記述子の名前。
INTO	ANSI 記述子。
descriptor_name	出力記述子の名前が含まれるホスト変数。

'descriptor name' 出力記述子の名前。

GLOBAL | LOCAL GLOBAL はアプリケーションの有効範囲を示し、LOCAL（デフォルト）はファイルの有効範囲を示します。

使用上の注意

この文の詳細は、[第 14 章「ANSI 動的 SQL」](#)を参照してください。

例

ANSI 動的 SQL 方法 4 では、EXECUTE の INTO 句を使用して次のように SELECT の DML 戻り句がサポートされます。

```
EXEC SQL EXECUTE S2 USING DESCRIPTOR :bv1 INTO DESCRIPTOR 'SELDES' ;
```

関連項目

F-37 ページの [DECLARE DATABASE（Oracle 埋込み SQL 宣言文）](#)

F-93 ページの [PREPARE（実行可能埋込み SQL）](#)

EXECUTE IMMEDIATE（実行可能埋込み SQL）

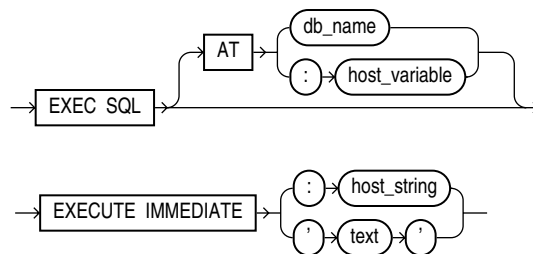
用途

ホスト変数を含まない DELETE 文、INSERT 文または UPDATE 文、あるいは PL/SQL ブロックを準備し、実行します。

前提条件

なし。

構文



キーワードおよびパラメータ

AT	SQL 文または PL/SQL ブロックが実行されるデータベースを指定します。次のいずれかを使用してデータベースを指定します。 <i>db_name</i> DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 <i>host_variable</i> 事前に宣言した <i>db_name</i> の値を持つホスト変数。 この句を省略した場合、文またはブロックはデフォルトのデータベースに対して実行されます。
<i>text</i>	実行する SQL 文または PL/SQL ブロックが含まれる引用符付きのテキスト・リテラル（または引用符なしのテキスト・リテラル）。 SQL 文は、DELETE 文、INSERT 文または UPDATE 文のいずれかである必要があります。
<i>host_string</i>	SQL 文を含むホスト変数。

使用上の注意

EXECUTE IMMEDIATE 文を発行すると、Oracle8i では指定した SQL 文または PL/SQL ブロックが解析されてエラー・チェックが行われ、実行されます。見つかったエラーは、SQLCA の SQLCODE コンポーネントに戻されます。

この文の詳細は、[第 13 章「Oracle の動的 SQL」](#) および [第 14 章「ANSI 動的 SQL」](#) を参照してください。

例

この例では、EXECUTE IMMEDIATE 文の使用方を示します。

```
EXEC SQL EXECUTE IMMEDIATE 'DELETE FROM emp WHERE empno = 9460' ;
```

関連項目

F-51 ページの [EXECUTE（実行可能埋込み SQL）](#)

F-93 ページの [PREPARE（実行可能埋込み SQL）](#)

FETCH (実行可能埋込み SQL)

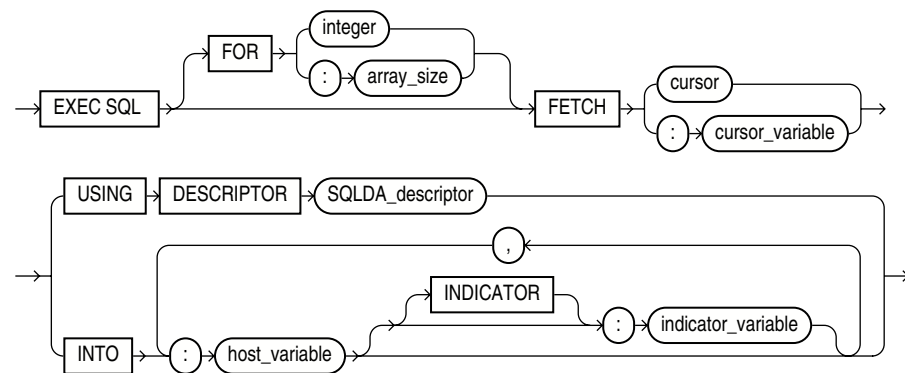
用途

Oracle 動的 SQL では、選択リストの値がホスト変数に割り当てられ、問合せが戻した 1 つまたは複数の行が取り出されます。ANSI 動的 SQL メソッド 4 については、F-59 ページの「[FETCH DESCRIPTOR \(実行可能埋込み SQL\)](#)」を参照してください。

前提条件

まず、OPEN 文を使用してカーソルを先にオープンしておく必要があります。

構文



キーワードおよびパラメータ

<i>FOR :array_size</i>	処理する行を含むホスト変数。
<i>FOR integer</i>	処理する行数。 配列ホスト変数を使用する場合にフェッチする行数を制限します。この句を省略した場合、Oracle8i では最小の配列を満たすのに十分な数の行がフェッチされます。
<i>cursor</i>	DECLARE CURSOR 文を使用して宣言したカーソル。FETCH 文は、カーソルに対応付けられた問合せが選択した行のうちの 1 行を戻します。
<i>cursor_variable</i>	カーソル変数は ALLOCATE 文で割り当てられます。FETCH 文は、カーソル変数に対応付けられた問合せが選択した行のうちの 1 行を戻します。

INTO	データのフェッチ先のホスト変数のリストとオプションの標識変数を指定します。これらのホスト変数および標識変数は、プログラム内で宣言されている必要があります。
<i>host_variable</i>	データを受け取るホスト変数。
<i>indicator_variables</i>	ホスト標識変数。
USING <i>SQLDA_variable</i>	前の DESCRIBE 文において参照された Oracle 記述子を指定します。この句は、動的埋込み SQL メソッド 4 のみで使います。カーソル変数を使用している場合は USING 句は適用されません。

使用上の注意

FETCH 文はアクティブ・セットの行を読み込み、結果が含まれる出力変数の名前を示します。対応付けられたホスト変数が NULL の場合、標識変数の値は -1 に設定されます。また、カーソルに対する最初の FETCH 文は、必要に応じてアクティブ・セットの行をソートします。

出力ホスト変数のサイズは取り出された行数を示し、FOR 句は値を示します。データを受け取るホスト変数は、すべてスカラーか、すべて配列である必要があります。スカラーの場合、Oracle8i では 1 行のみフェッチされます。配列の場合、Oracle8i では配列を満たすのに十分な数の行がフェッチされます。

配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracle8i でフェッチされる行数は、次の値のうち小さい方です。

- 最小の配列のサイズ
- オプションの FOR 句の *:array_size* の値

フェッチする行数は、実際に問合せを満たす行の数によってさらに限定できます。

FETCH 文が、問合せで戻された行をすべて取り出さない場合、カーソルは戻された次の行に配置されます。問合せで戻された最後の行を取り出すと、その次の FETCH ではエラー・コードが発生します。このエラー・コードは SQLCA の SQLCODE 要素に戻されます。

FETCH 文には AT 句が含まれないことに注意してください。カーソルによってアクセスされるデータベースは、DECLARE CURSOR 文で指定する必要があります。

FETCH 文では、アクティブ・セット内を前方向にのみ進めます。すでにフェッチした行に戻す場合は、カーソルを再オープンして各行を順番に取り出す必要があります。アクティブ・セットを変更する場合は、カーソルの問合せにおいて入力ホスト変数に新しい値を割り当てて、カーソルを再オープンする必要があります。

Oracle の記述子の詳細は、6-13 ページの「[FETCH 文の使用方法](#)」を参照してください。

例

この例では、FETCH 文を示します。

```
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;
    ...
}
```

関連項目

F-18 ページの [CLOSE](#) (実行可能埋込み SQL)

F-35 ページの [DECLARE CURSOR](#) (埋込み SQL 宣言文)

F-89 ページの [OPEN](#) (実行可能埋込み SQL)

F-93 ページの [PREPARE](#) (実行可能埋込み SQL)

FETCH DESCRIPTOR (実行可能埋込み SQL)

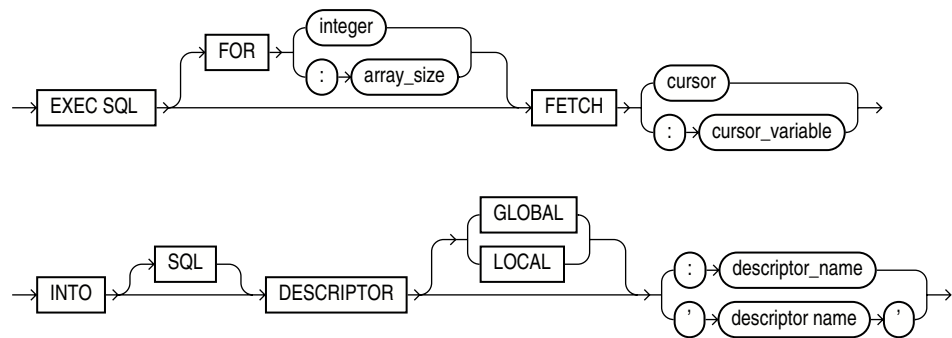
用途

選択リストの値をホスト変数に割り当てて、問合せが戻した 1 つまたは複数の行を取り出します。ANSI 動的 SQL メソッド 4 で使用されます。

前提条件

まず、OPEN 文を使用してカーソルを先にオープンしておく必要があります。

構文



キーワードおよびパラメータ

<i>array_sizeinteger</i>	処理する行を含むホスト変数。処理する行数。配列ホスト変数を使用する場合にフェッチする行数を制限します。この句を省略した場合、Oracle8i では最小の配列を満たすのに十分な数の行がフェッチされます。
<i>cursor</i>	DECLARE CURSOR 文を使用して宣言したカーソル。FETCH 文は、カーソルに対応付けられた問合せが選択した行のうちの 1 行を戻します。
<i>cursor_variable</i>	カーソル変数は ALLOCATE 文で割り当てられます。FETCH 文は、カーソル変数に対応付けられた問合せが選択した行のうちの 1 行を戻します。
GLOBAL LOCAL	GLOBAL はアプリケーションの有効範囲を示し、LOCAL（デフォルト）は ファイルの有効範囲を示します。
INTO	データのフェッチ先のホスト変数のリストとオプションの標識変数を指定します。これらのホスト変数および標識変数は、プログラム内で宣言されている必要があります。
<i>'descriptor name'</i> <i>descriptor_name</i>	出力 ANSI 記述子の名前。出力記述子の名前が含まれるホスト変数。

使用上の注意

出力ホスト変数のサイズは取り出された行数を示し、FOR 句は値を示します。データを受け取るホスト変数は、すべてスカラーか、すべて配列である必要があります。スカラーの場合、Oracle8i では 1 行のみフェッチされます。配列の場合、Oracle8i では配列を満たすのに十分な数の行がフェッチされます。

配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracle8i でフェッチされる行数は、次の値のうち小さい方です。

- 最小の配列のサイズ
- オプションの FOR 句の `:array_size` の値

フェッチする行数は、実際に問合せを満たす行の数によってさらに限定できます。

FETCH 文が、問合せで戻された行をすべて取り出したのではない場合、カーソルは戻された次の行に配置されます。問合せで戻された最後の行を取り出すと、その次の FETCH ではエラー・コードが発生します。このエラー・コードは SQLCA の SQLCODE 要素に戻されます。

FETCH 文には AT 句が含まれないことに注意してください。カーソルによってアクセスされるデータベースは、DECLARE CURSOR 文で指定する必要があります。

FETCH 文では、アクティブ・セット内を前方向にのみ進めます。すでにフェッチした行に戻りたい場合は、カーソルを再オープンして各行を順番に取り出す必要があります。アクティブ・セットを変更する場合は、カーソルの問合せにおいて入力ホスト変数に新しい値を割り当て、カーソルを再オープンする必要があります。

ANSI SQL メソッド 4 アプリケーションには DYNAMIC=ANSI プリコンパイラ・オプションを指定します。ANSI SQL メソッド 4 アプリケーションの詳細は、14-25 ページの「[FETCH](#)」を参照してください。

例

```
...
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' ;
...
```

関連項目

F-18 ページの [CLOSE](#) (実行可能埋込み SQL)

F-35 ページの [DECLARE CURSOR](#) (埋込み SQL 宣言文)

F-91 ページの [OPEN DESCRIPTOR](#) (実行可能埋込み SQL)

F-93 ページの [PREPARE](#) (実行可能埋込み SQL)

FREE（実行可能埋込み SQL 拡張要素）

用途

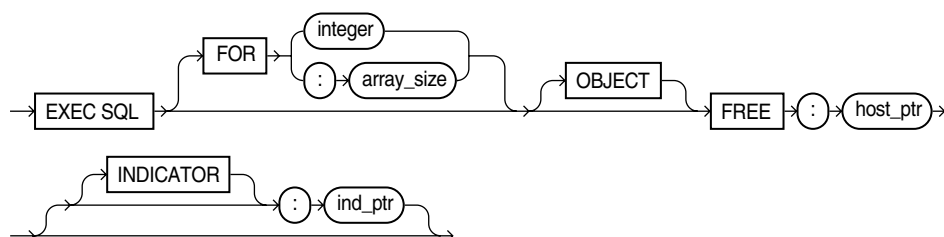
オブジェクト・キャッシュ内のメモリーを解放します。

前提条件

メモリーが割当て済である必要があります。

データベースへの接続がアクティブである必要があります。

構文



キーワードおよびパラメータ

<i>dbname</i>	事前に CONNECT 文で確立されたデータベース接続の名前を含む NULL 終了記号付き文字列。これが省略されたり、空の文字列であったときは、デフォルトのデータベース接続とみなされます。
<i>host_ptr</i>	事前に ALLOCATE で割り当てられていたホスト変数ポインタ。
<i>ind_ptr</i>	標識ポインタ。

使用上の注意

接続が切り離されると、オブジェクト・キャッシュに割り当てられているメモリーはすべて自動的に解放されます。詳細は、17-5 ページの「[FREE](#)」を参照してください。

例

```
EXEC SQL FREE :ptr ;
```


関連項目

F-12 ページの [ALLOCATE](#) (実行可能埋込み SQL 拡張要素)

F-16 ページの [CACHE FREE ALL](#) (実行可能埋込み SQL 拡張要素)

GET DESCRIPTOR (実行可能埋込み SQL)

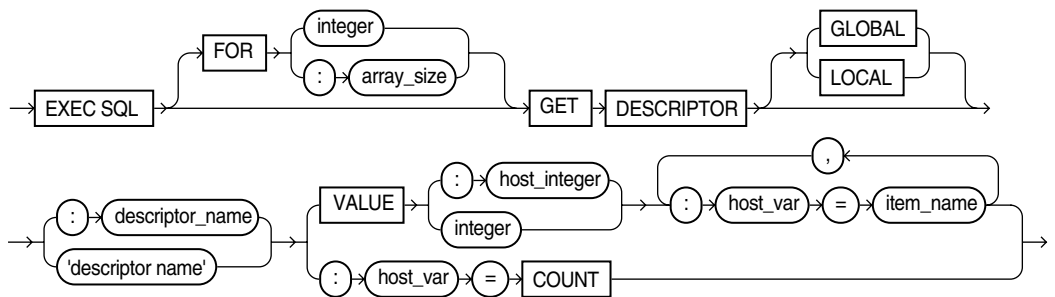
用途

SQL 記述子領域からホスト変数についての情報を取得します。

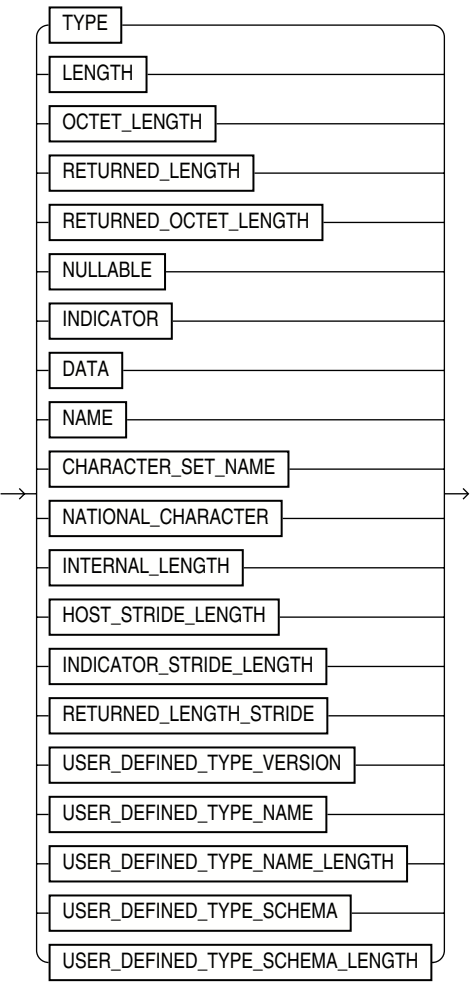
前提条件

値構文でのみ使用します。

構文



item_name は次のうちの 1 つになります。



キーワードおよびパラメータ

<i>array_sizeinteger</i>	処理する行を含むホスト変数。処理する行数。
<i>descriptor_name</i>	割り当てられた ANSI 記述子の名前が含まれるホスト変数。割り当てられた
<i>'descriptor name'</i>	ANSI 記述子の名前。

GLOBAL LOCAL	GLOBAL はアプリケーションの有効範囲を示し、LOCAL（デフォルト）はファイルの有効範囲を示します。
<i>host_var</i> = COUNT <i>integer</i>	入力変数または出力変数の合計数を含むホスト変数。入力変数または出力変数の合計数。
VALUE : <i>host_integer</i>	参照される入力または出力変数の位置を含むホスト変数。
VALUE <i>integer</i>	参照される入力または出力変数の位置。
<i>host_var</i>	項目の値を受け取るホスト変数。
<i>item_name</i>	<i>item_name</i> の例は 14-15 ページの表 14-4「GET DESCRIPTOR の記述子項目名の定義」および 14-15 ページの表 14-5「Oracle 拡張機能により追加された GET DESCRIPTOR の記述子項目名の定義」を参照してください。

使用上の注意

DYNAMIC=ANSI プリコンパイラ・オプションを使用します。配列サイズ句は、DATA、RETURNED_LENGTH および INDICATOR 項目名で使用できます。14-13 ページの「GET DESCRIPTOR」を参照してください。

例

```
EXEC SQL GET DESCRIPTOR GLOBAL 'mydesc' :mydesc_num_vars = COUNT ;
```

関連項目

F-14 ページの [ALLOCATE DESCRIPTOR（実行可能埋込み SQL）](#)

F-34 ページの [DEALLOCATE DESCRIPTOR（埋込み SQL 文）](#)

F-103 ページの [SET DESCRIPTOR（実行可能埋込み SQL）](#)

INSERT（実行可能埋込み SQL）

用途

表またはビューのベース表に行を追加します。

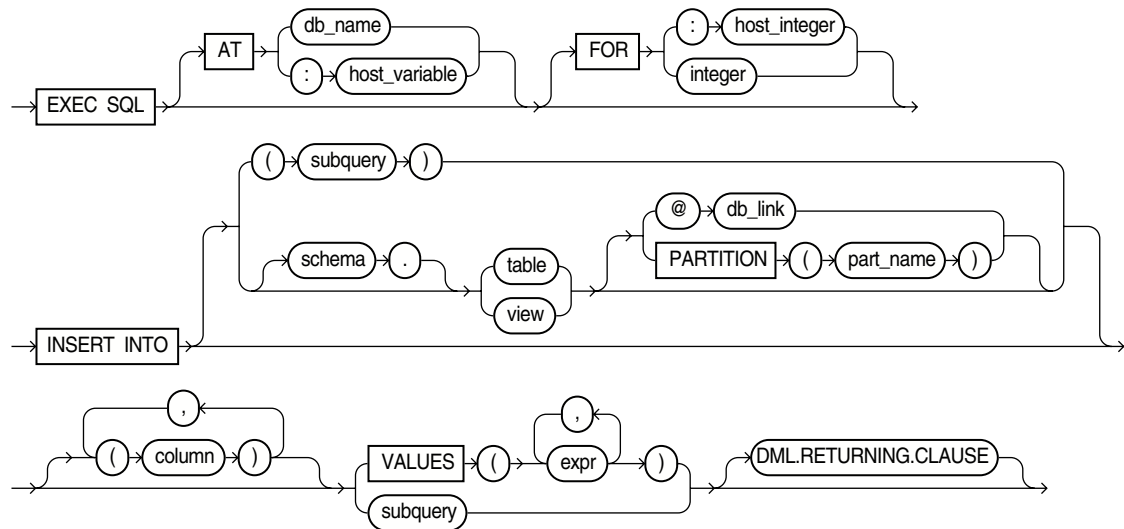
前提条件

表に行を挿入するには、その表が自分のスキーマ内にあるか、またはその表に対して INSERT の権限を持つ必要があります。

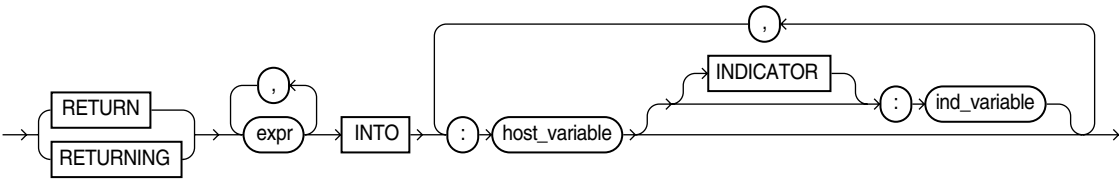
ビューのベース表に行を挿入するには、ビューが属するスキーマの所有者が、そのベース表に対して INSERT の権限を持つ必要があります。また、ビューが自分のスキーマ以外のスキーマ内にある場合は、ビューに対して INSERT の権限を持つ必要があります。

INSERT ANY TABLE システム権限を使用すると、どの表またはビューのベース表にも行を挿入できます。

構文



DML 戻り句は次のとおりです。



キーワードおよびパラメータ

AT	INSERT 文を実行するデータベースを指定します。次のいずれかを使用してデータベースを指定します。
db_name	DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。
host_variable	値が db_name のホスト変数。

	この句を省略した場合、INSERT 文はデフォルトのデータベースについて実行されます。
FOR :host_integer integer	VALUES 句に配列ホスト変数が含まれる場合に、文を実行する回数を制限します。この句を省略すると、Oracle8i では最小の配列の各コンポーネントについて 1 回ずつ文が実行されます。
schema	表またはビューが含まれるスキーマ。schema を省略した場合、Oracle8i では表またはビューが独自のスキーマにあるとみなされます。
table view	行を挿入する表の名前。ビューを指定すると、Oracle8i によりそのビューのベース表が挿入されます。
db_link	表またはビューがあるリモート・データベースへのデータベース・リンクの完全名または部分名。データベース・リンクの参照の詳細は、『Oracle8i SQL リファレンス』を参照してください。 リモートの表またはビューに行を挿入できるのは、Oracle8i を分散オプションで使用している場合に限られます。 dblink を省略した場合、Oracle8i では表またはビューがローカル・データベース内にあるとみなされます。
part_name	表のパーティションの名前。
column	このリストから表の 1 列を省略すると、挿入行のその列値には、表の作成時に指定される列のデフォルト値が使用されます。列のリストを完全に省略した場合は、VALUES 句または問合せによって、表のすべての列の値を指定する必要があります。
VALUES	表またはビューに挿入される値の行を指定します。『Oracle8i SQL リファレンス』における構文記述を参照してください。式には、ホスト変数とオプションの標識変数を使用できるので注意してください。VALUES 句では、列のリストの各列に式を指定する必要があります。
subquery	表に挿入される行を戻す副問合せ。この副問合せの選択リストの列数は、INSERT 文の列のリストの列数と同じである必要があります。副問合せの構文の説明は、『Oracle8i SQL リファレンス』の「SELECT」を参照してください。
DML 戻り句	6-10 ページの「 DML 戻り句 」を参照してください。

使用上の注意

WHERE 句内のホスト変数は、すべてスカラーか、すべて配列である必要があります。スカラーの場合、Oracle8i では INSERT 文が 1 回のみ実行されます。変数が配列の場合、Oracle8i では INSERT 文を各配列コンポーネント・セットについて 1 回ずつ実行して、1 行ずつ挿入します。

WHERE 句の配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracle8i で文が実行される回数は、次の値のうち小さい方によって決まります。

- 最小の配列のサイズ
- オプションの FOR 句の `:host_integer` の値

この文の詳細は、6-8 ページの「[INSERT 文の使用法](#)」を参照してください。

例 I

この例では、埋込み SQL INSERT 文の使用法を示しています。

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

例 II

この例では、副問合せを使用した埋込み SQL の INSERT 文を示します。

```
EXEC SQL
    INSERT INTO new_emp (ename, empno, sal)
    SELECT ename, empno, sal FROM emp
    WHERE deptno = :deptno ;
```

関連項目

F-37 ページの [DECLARE DATABASE](#)（Oracle 埋込み SQL 宣言文）

LOB APPEND（実行可能埋込み SQL 拡張要素）

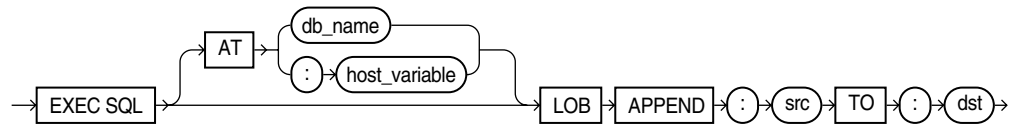
用途

LOB の最後に別の LOB を追加します。

前提条件

LOB バッファリングは使用可能である必要があります。宛先 LOB は初期化されている必要があります。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-11 ページの「[APPEND](#)」を参照してください。

関連項目

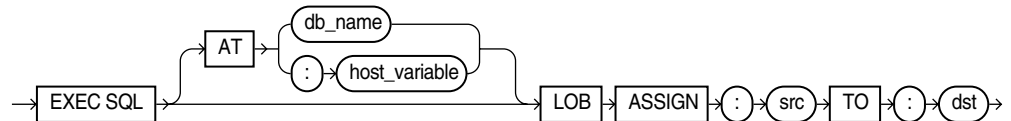
他の LOB 文を参照してください。

LOB ASSIGN（実行可能埋込み SQL 拡張要素）

用途

LOB または BFILE ロケータを別のロケータに割り当てます。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-12 ページの「[ASSIGN](#)」を参照してください。

関連項目

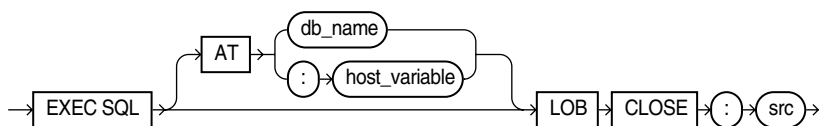
他の LOB 文を参照してください。

LOB CLOSE (実行可能埋込み SQL 拡張要素)

用途

LOB または BFILE をクローズします。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-13 ページの「[CLOSE](#)」を参照してください。

関連項目

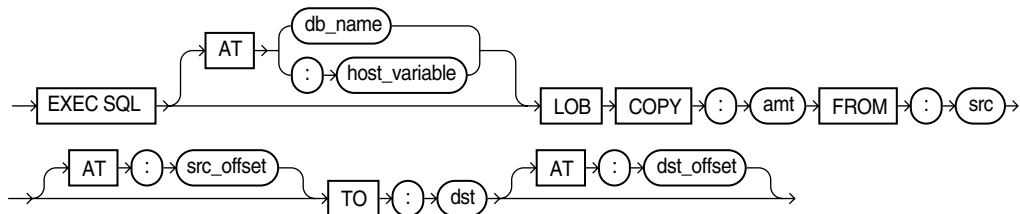
他の LOB 文を参照してください。

LOB COPY (実行可能埋込み SQL 拡張要素)

用途

LOB 値の全部または一部を別の LOB にコピーします。

構文



使用上の注意

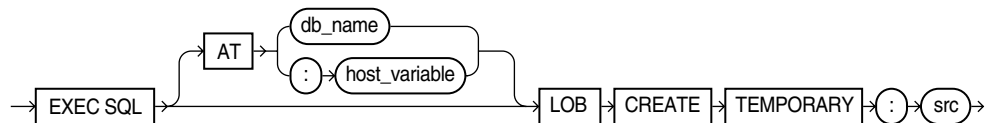
使用上の注意、キーワード、パラメータ、例は、16-13 ページの「[COPY](#)」を参照してください。

関連項目

他の LOB 文を参照してください。

LOB CREATE TEMPORARY（実行可能埋込み SQL 拡張要素）**用途**

テンポラリ LOB を作成します。

構文**使用上の注意**

使用上の注意、キーワード、パラメータ、例は、16-14 ページの「[CREATE TEMPORARY](#)」を参照してください。

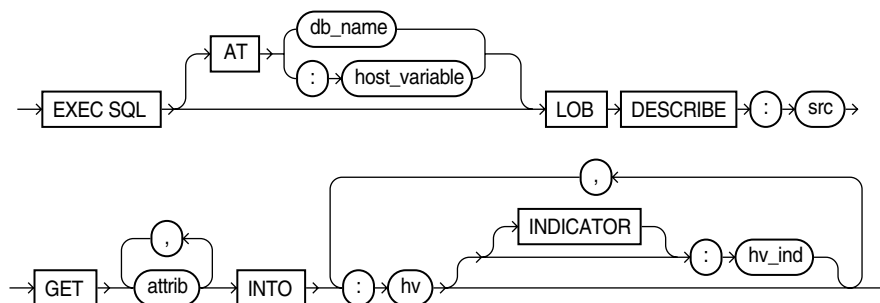
関連項目

他の LOB 文を参照してください。

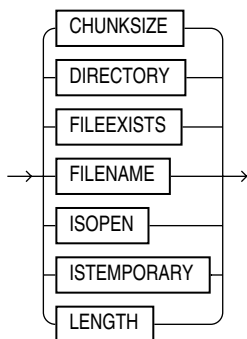
LOB DESCRIBE（実行可能埋込み SQL 拡張要素）**用途**

LOB から属性を取得します。

構文



attrib は次のとおりです。



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-25 ページの「[DESCRIBE](#)」を参照してください。

関連項目

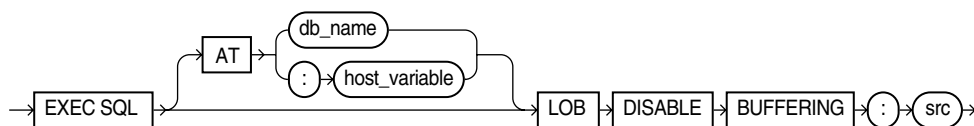
他の LOB 文を参照してください。

LOB DISABLE BUFFERING（実行可能埋込み SQL 拡張要素）

用途

LOB バッファリングを使用禁止にします。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-15 ページの「[DISABLE BUFFERING](#)」を参照してください。

関連項目

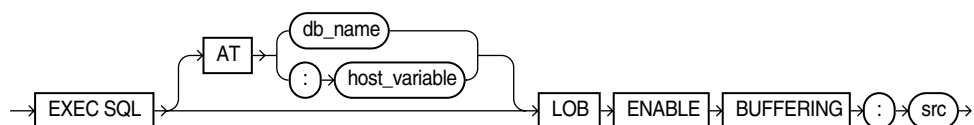
他の LOB 文を参照してください。

LOB ENABLE BUFFERING（実行可能埋込み SQL 拡張要素）

用途

LOB バッファリングを使用可能にします。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-15 ページの「[ENABLE BUFFERING](#)」を参照してください。

関連項目

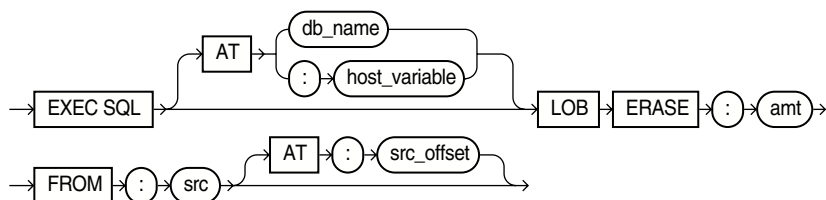
他の LOB 文を参照してください。

LOB ERASE（実行可能埋込み SQL 拡張要素）

用途

LOB データの任意の値の消去を任意のオフセットから開始します。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-16 ページの「[ERASE](#)」を参照してください。

関連項目

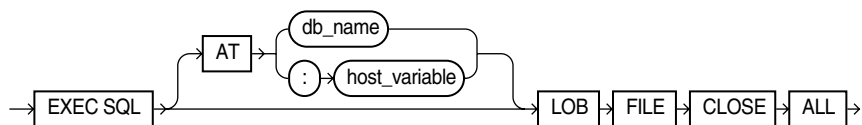
他の LOB 文を参照してください。

LOB FILE CLOSE ALL（実行可能埋込み SQL 拡張要素）

用途

カレント・セッションでオープンしているすべての BFILE をクローズします。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-17 ページの「[FILE CLOSE ALL](#)」を参照してください。

関連項目

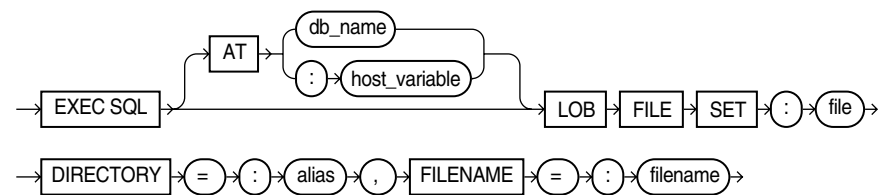
他の LOB 文を参照してください。

LOB FILE SET（実行可能埋込み SQL 拡張要素）

用途

BFILE ロケータに DIRECTORY および FILENAME を設定します。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-17 ページの「[FILE SET](#)」を参照してください。

関連項目

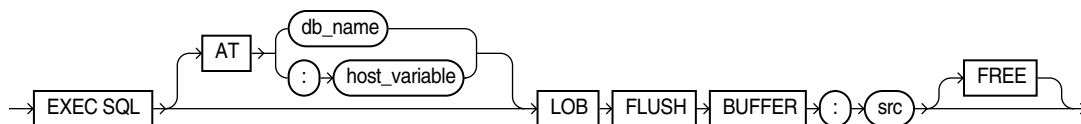
他の LOB 文を参照してください。

LOB FLUSH BUFFER（実行可能埋込み SQL 拡張要素）

用途

データベース・サーバーに LOB バッファを書き込みます。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-18 ページの「[FLUSH BUFFER](#)」を参照してください。

関連項目

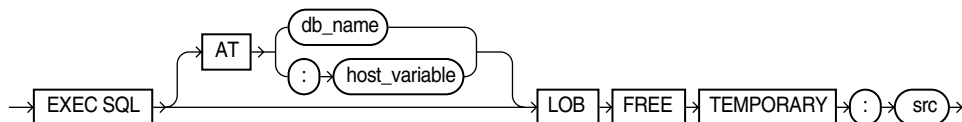
他の LOB 文を参照してください。

LOB FREE TEMPORARY（実行可能埋込み SQL 拡張要素）

用途

LOB ロケータのテンポラリ領域を解放します。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-18 ページの「[FREE TEMPORARY](#)」を参照してください。

関連項目

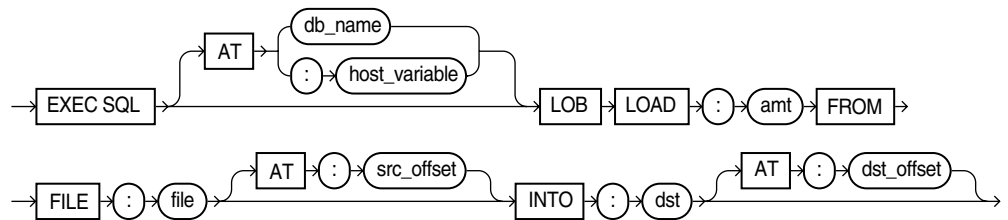
他の LOB 文を参照してください。

LOB LOAD（実行可能埋込み SQL 拡張要素）

用途

BFILE の全部または一部を内部 LOB にコピーします。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-19 ページの「[LOAD FROM FILE](#)」を参照してください。

関連項目

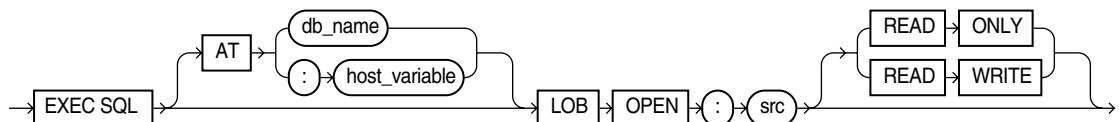
他の LOB 文を参照してください。

LOB OPEN（実行可能埋込み SQL 拡張要素）

用途

読み込みまたは読み込み / 書き込みアクセスで使用する LOB または BFILE をオープンします。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-20 ページの「[OPEN](#)」を参照してください。

関連項目

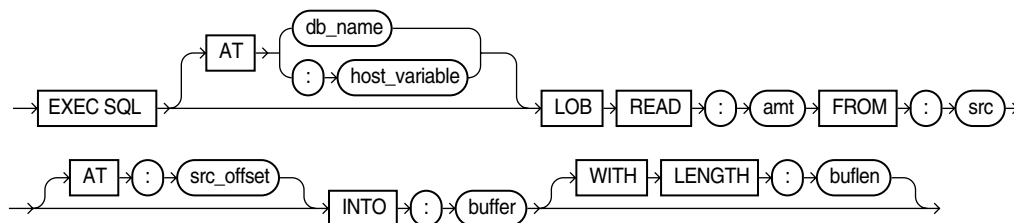
他の LOB 文を参照してください。

LOB READ (実行可能埋込み SQL 拡張要素)

用途

LOB または BFILE の全部または一部をバッファに読み込みます。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-21 ページの「[READ](#)」を参照してください。

関連項目

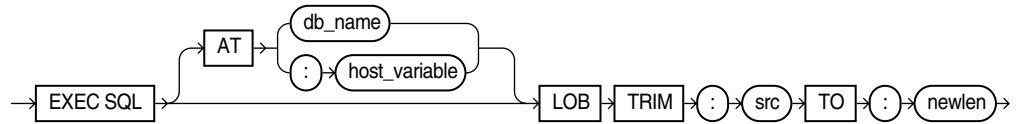
他の LOB 文を参照してください。

LOB TRIM (実行可能埋込み SQL 拡張要素)

用途

LOB 値を切り捨てます。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-23 ページの「[TRIM](#)」を参照してください。

関連項目

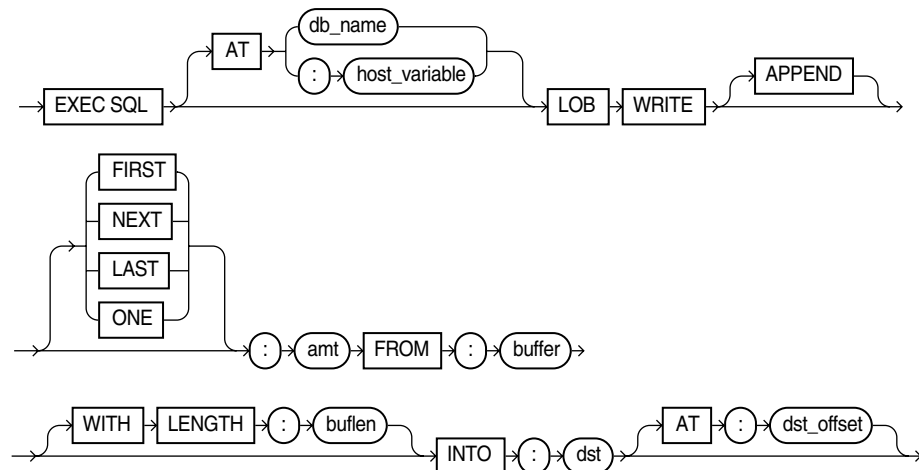
他の LOB 文を参照してください。

LOB WRITE（実行可能埋込み SQL 拡張要素）

用途

バッファの内容を LOB に書き込みます。

構文



使用上の注意

使用上の注意、キーワード、パラメータ、例は、16-23 ページの「[WRITE](#)」を参照してください。

関連項目

他の LOB 文を参照してください。

OBJECT CREATE（実行可能埋込み SQL 拡張要素）

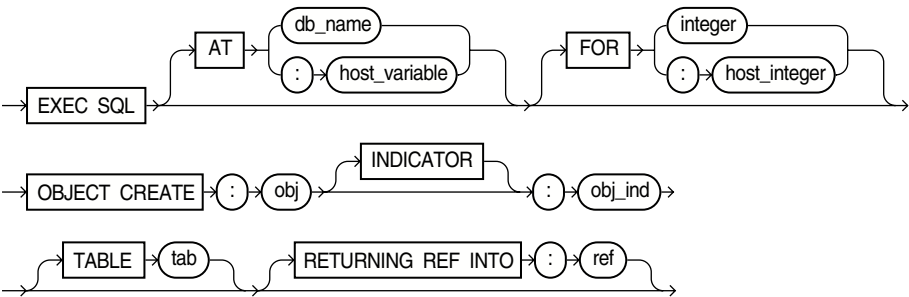
用途

オブジェクト・キャッシュ内に参照可能なオブジェクトを作成します。

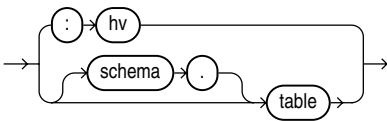
前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。INTYPE オプションでは、OTT によって生成される型ファイルを指定する必要があります。OTT によって生成されるヘッダー・ファイルをプログラムに組み込んでください。

構文



tab は次のとおりです。



使用上の注意

使用上の注意およびキーワードとパラメータは、17-9 ページの「[OBJECT CREATE](#)」を参照してください。

例

```
person *pers_p;
person_ind *pers_ind;
person_ref *pers_ref;
...
EXEC SQL OBJECT CREATE :pers_p:pers_ind TABLE PERSON_TAB
      RETURNING REF INTO :pers_ref ;
```

関連項目

この付録における他のすべての OBJECT 文を参照してください。

OBJECT DELETE（実行可能埋込み SQL 拡張要素）

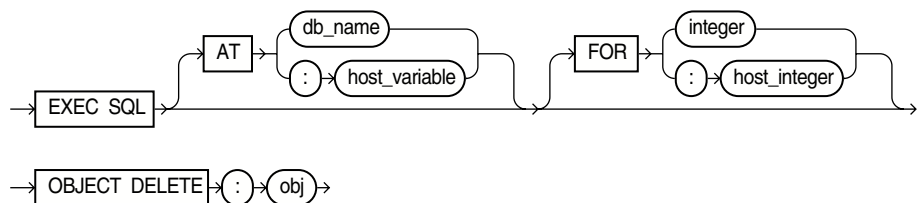
用途

オブジェクト・キャッシュ内の永続オブジェクトまたはオブジェクトの配列を削除済としてマーク設定します。

前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。INTYPE オプションでは、OTT によって生成される型ファイルを指定する必要があります。OTT によって生成されるヘッダー・ファイルをプログラムに組み込んでください。

構文



使用上の注意

使用上の注意およびキーワードとパラメータは、17-11 ページの「OBJECT DELETE」を参照してください。

例

```
customer *cust_p;  
...  
EXEC SQL OBJECT DELETE :cust_p;
```

関連項目

この付録にある他のすべての OBJECT 文を参照してください。永続オブジェクトの場合、この文はオブジェクト・キャッシュ内のオブジェクトまたはオブジェクトの配列を削除済としてマーク設定します。

OBJECT Deref (実行可能埋込み SQL 拡張要素)

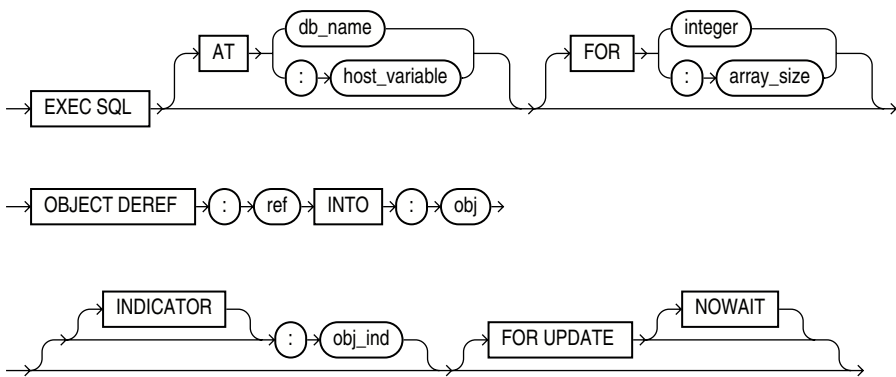
用途

オブジェクト・キャッシュ内にオブジェクトまたはオブジェクトの配列を保持します。

前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。INTYPE オプションでは、OTT によって生成される型ファイルを指定する必要があります。OTT によって生成されるヘッダー・ファイルをプログラムに組み込んでください。

構文



使用上の注意

使用上の注意およびキーワードとパラメータは、17-10 ページの「[OBJECT DEREf](#)」を参照してください。

例

```
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;
```

関連項目

この付録にある他のすべての OBJECT 文を参照してください。F-12 ページの「[ALLOCATE（実行可能埋込み SQL 拡張要素）](#)」を参照してください。

OBJECT FLUSH（実行可能埋込み SQL 拡張要素）

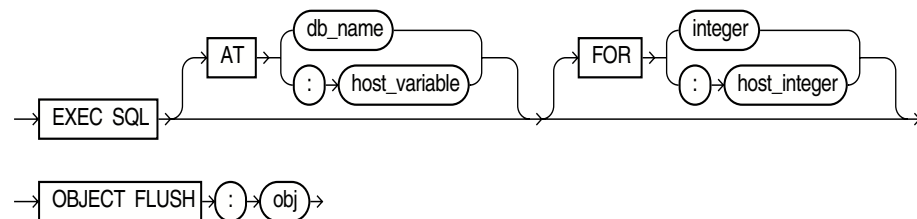
用途

更新済、削除済または作成済としてマーク設定された永続オブジェクトを、サーバーに反映します。この処理をフラッシュと呼びます。

前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。INTYPE オプションでは、OTT によって生成される型ファイルを指定する必要があります。OTT によって生成されるヘッダー・ファイルをプログラムに組み込んでください。

構文



使用上の注意

使用上の注意およびキーワードとパラメータは、17-12 ページの「[OBJECT FLUSH](#)」を参照してください。

例

```
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;
```

関連項目

この付録にある他のすべての OBJECT 文を参照してください。

OBJECT GET（実行可能埋込み SQL 拡張要素）

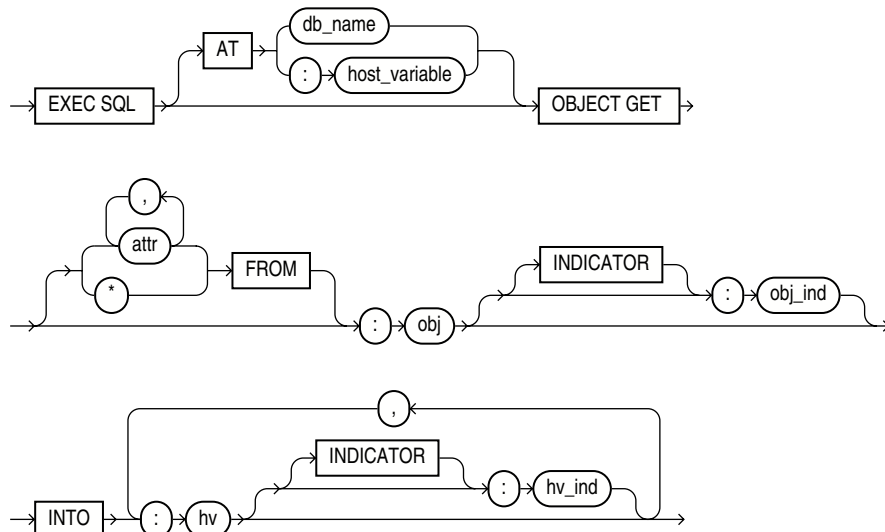
用途

オブジェクト型の属性を固有の C の型に変換します。

前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。INTYPE オプションでは、OTT によって生成される型ファイルを指定する必要があります。OTT によって生成されるヘッダー・ファイルをプログラムに組み込んでください。

構文



使用上の注意

使用上の注意およびキーワードとパラメータは、17-16 ページの「[OBJECT GET](#)」を参照してください。

例

```

person *pers_p;
struct { char lname[21], fname[21]; int age; } pers;
...
/* Convert object types to native C types */
EXEC SQL OBJECT GET lastname, firstname, age FROM :pers_p INTO :pers;
printf("Last Name: %s\nFirstName: %s\nAge: %d\n",
       pers.lname, pers.fname, pers.age );
  
```

関連項目

この付録にある他のすべての OBJECT 文を参照してください。

OBJECT RELEASE（実行可能埋込み SQL 拡張要素）

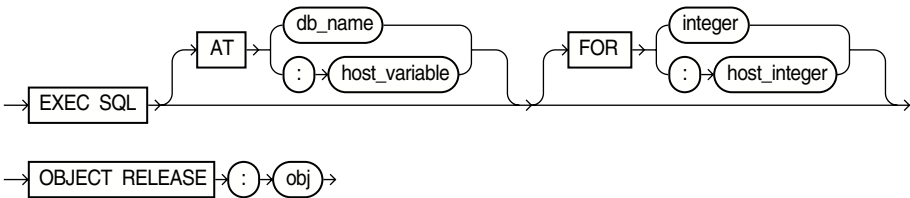
用途

オブジェクト・キャッシュ内のオブジェクトを解放します。オブジェクトが確保されず、更新されなければ、暗黙的な解放の対象になります。

前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。INTYPE オプションでは、OTT によって生成される型ファイルを指定する必要があります。OTT によって生成されるヘッダー・ファイルをプログラムに組み込んでください。

構文



使用上の注意

使用上の注意およびキーワードとパラメータは、17-11 ページの「[OBJECT RELEASE](#)」を参照してください。

例

```
person *pers_p;  
...  
EXEC SQL OBJECT RELEASE :pers_p;
```

関連項目

この付録にある他のすべての OBJECT 文を参照してください。

例

```
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;
```

関連項目

この付録にある他のすべての OBJECT 文を参照してください。

OBJECT UPDATE（実行可能埋込み SQL 拡張要素）

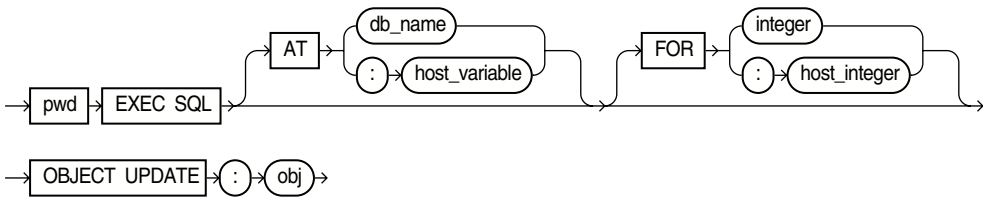
用途

オブジェクト・キャッシュ内の永続オブジェクトまたはオブジェクトの配列を更新されたとしてマーク設定します。

前提条件

プリコンパイラ・オプション OBJECTS を YES に設定する必要があります。INTYPE オプションでは、OTT によって生成される型ファイルを指定する必要があります。OTT によって生成されるヘッダー・ファイルをプログラムに組み込んでください。

構文



使用上の注意

使用上の注意およびキーワードとパラメータは、17-11 ページの「[OBJECT UPDATE](#)」を参照してください。

例

```
person *pers_p;  
...  
/* Mark as updated */  
EXEC SQL OBJECT UPDATE :pers_p;
```

関連項目

この付録にある他のすべての OBJECT 文を参照してください。

OPEN（実行可能埋込み SQL）

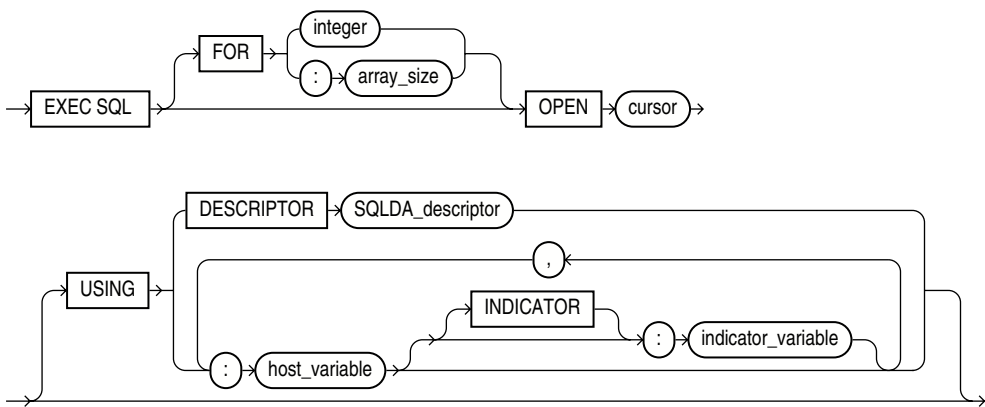
用途

対応付けられた問合せを評価し、USING 句が示すホスト変数名を問合せの WHERE 句に代入して、カーソルをオープンします。ANSI 動的 SQL メソッド 4 バージョンは、F-91 ページの「[OPEN DESCRIPTOR（実行可能埋込み SQL）](#)」を参照してください。

前提条件

カーソルは、オープンする前に埋込み SQL の DECLARE CURSOR 文を使用して宣言しておく必要があります。

構文



キーワードおよびパラメータ

<i>array_sizeinteger</i>	処理する行を含むホスト変数。処理する行数。
<i>cursor</i>	（事前に宣言された）オープンするカーソル。
<i>host_variable</i>	カーソルに対応付けられている文に代入されるオプションの標識変数を使用して、ホスト変数を指定します。ANSI 識別子（INTO 句）とともに使用することはできません。
DESCRIPTOR <i>SQLDA_descriptor</i>	対応付けられた問合せの WHERE 句に代入するホスト変数を表す Oracle 記述子を指定します。記述子は、DESCRIBE 文を使用して事前に初期化されている必要があります。代入は、位置に基づきます。この文で指定するホスト変数名は、対応付けられた問合せの変数名と異なってもかまいません。ANSI 識別子（INTO 句）とともに使用することはできません。

使用上の注意

OPEN 文は、行のアクティブ・セットを定義し、アクティブ・セットの最初の行の直前でカーソルを初期化します。OPEN 時のホスト変数の値が文に代入されます。この文は、実際には行を取り出しません。行は FETCH 文を使用して取り出されます。

カーソルを一度オープンすると、入力ホスト変数はカーソルを再オープンするまで再テストされません。入力ホスト変数およびアクティブ・セットを変更するには、カーソルを再オープンする必要があります。

プログラムを開始するときまたは CLOSE 文を使用してカーソルを明示的にクローズした後には、プログラム内のすべてのカーソルがクローズ状態になります。

カーソルはクローズせずに再オープンできます。この文の詳細は、6-8 ページの「[INSERT 文の使用方法](#)」を参照してください。

例

この例では、Pro*C/C++ プログラムで OPEN 文を使用する方法を示しています。

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

関連項目

F-18 ページの [CLOSE（実行可能埋込み SQL）](#)

F-35 ページの [DECLARE STATEMENT（埋込み SQL 宣言文）](#)

F-57 ページの [FETCH（実行可能埋込み SQL）](#)

F-93 ページの [PREPARE（実行可能埋込み SQL）](#)

OPEN DESCRIPTOR（実行可能埋込み SQL）

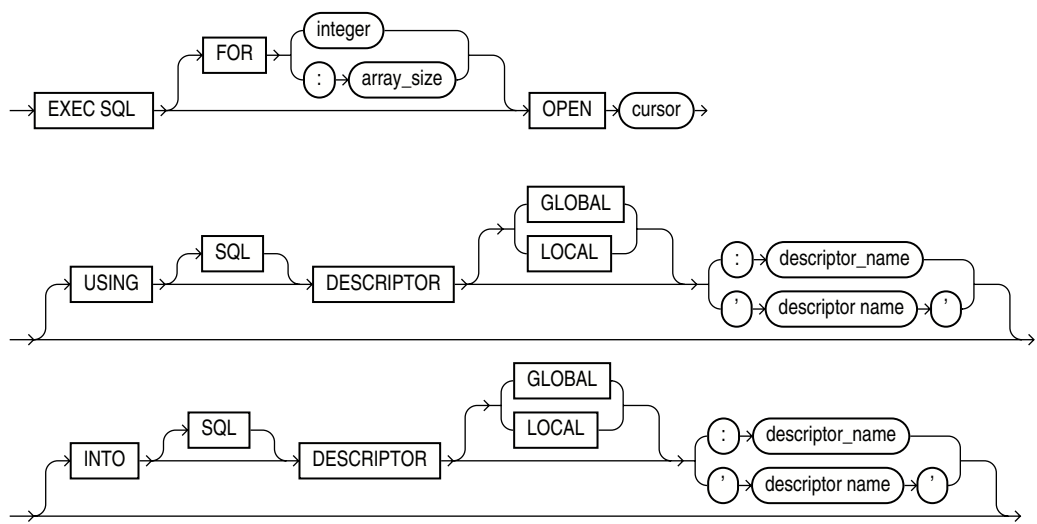
用途

対応付けられた問合せを評価し、USING 句が示すホスト変数名を問合せの WHERE 句に代入して、カーソルをオープンします（ANSI 動的 SQL メソッド 4 の場合）。INTO 句は出力記述子を示します。

前提条件

カーソルは、オープンする前に埋込み SQL の DECLARE CURSOR 文を使用して宣言しておく必要があります。

構文



キーワードおよびパラメータ

<i>array_sizeinteger</i>	処理する行を含むホスト変数。処理する行数。
<i>cursor</i>	（事前に宣言された）オープンするカーソル。
SET DESCRIPTOR	ANSI 入力記述子を指定します。
<i>descriptor_name</i>	ANSI 記述子の名前が含まれるホスト変数。
<i>'descriptor name'</i>	ANSI 記述子の名前。
INTO DESCRIPTOR	ANSI 出力記述子を指定します。
<i>descriptor_name</i>	ANSI 記述子の名前が含まれるホスト変数。
<i>'descriptor name'</i>	ANSI 記述子の名前。
GLOBAL LOCAL	GLOBAL はアプリケーションの有効範囲を示し、LOCAL（デフォルト）はファイルの有効範囲を示します。

使用上の注意

プリコンパイラ・オプション DYNAMIC を ANSI に設定します。

OPEN 文は、行のアクティブ・セットを定義し、アクティブ・セットの最初の行の直前でカーソルを初期化します。OPEN 時のホスト変数の値が SQL 文に代入されます。この文は、実際には行を取り出しません。行は FETCH 文を使用して取り出されます。

カーソルを一度オープンすると、入力ホスト変数はカーソルを再オープンするまで再テストされません。入力ホスト変数およびアクティブ・セットを変更するには、カーソルを再オープンする必要があります。

プログラムを開始するときまたは CLOSE 文を使用してカーソルを明示的にクローズした後は、プログラム内のすべてのカーソルがクローズ状態になります。

カーソルはクローズせずに再オープンできます。この文の詳細は、6-8 ページの「[INSERT 文の使用法](#)」を参照してください。

例

```
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' ;
...
```

関連項目

F-18 ページの [CLOSE（実行可能埋込み SQL）](#)

F-35 ページの [DECLARE CURSOR（埋込み SQL 宣言文）](#)

F-59 ページの [FETCH DESCRIPTOR（実行可能埋込み SQL）](#)

F-93 ページの [PREPARE（実行可能埋込み SQL）](#)

PREPARE（実行可能埋込み SQL）

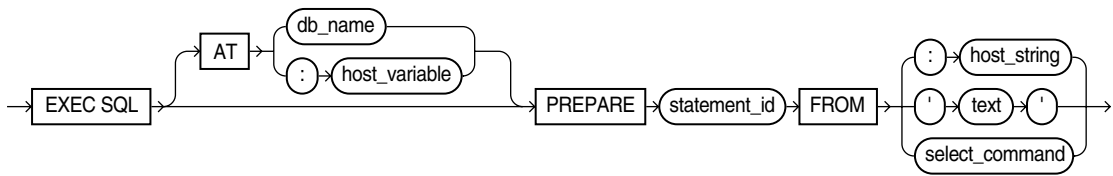
用途

ホスト変数により指定される SQL 文または PL/SQL ブロックを解析し、それを識別子に対応付けます。

前提条件

なし。

構文



キーワードおよびパラメータ

<i>statement_id</i>	準備済の SQL 文または PL/SQL ブロックに対応付ける識別子。この識別子がすでに別の文またはブロックに割り当てられている場合は、以前の割当てが置き換えられます。
<i>host_string</i>	準備する SQL 文または PL/SQL ブロックのテキストを値とするホスト変数。 この句を省略した場合、Oracle8i ではデフォルトのデータベースに対して文が発行されます。
<i>text</i>	準備する SQL 文または PL/SQL ブロックを含む文字列リテラル。
<i>select_command</i>	選択文。
<i>statement_id</i>	準備済の SQL 文または PL/SQL ブロックに対応付ける識別子。この識別子がすでに別の文またはブロックに割り当てられている場合は、以前の割当てが置き換えられます。
<i>host_string</i>	準備する SQL 文または PL/SQL ブロックのテキストを値とするホスト変数。
<i>text</i>	準備する SQL 文または PL/SQL ブロックを含む文字列リテラル。引用符は省略できます。
<i>select_command</i>	選択文。

使用上の注意

host_string または *text* の変数はすべてプレースホルダです。実際のホスト変数名は、OPEN 文の USING 句（入力ホスト変数）、または FETCH 文の INTO 句（出力ホスト変数）で割り当てます。

SQL 文は一度準備すれば、何回でも実行できます。詳細は、13-18 ページの「[PREPARE](#)」を参照してください。

例

この例では、Pro*C/C++ の埋込み SQL プログラムにおける PREPARE 文の使用方を示します。

```
EXEC SQL PREPARE my_statement FROM :my_string;
EXEC SQL EXECUTE my_statement;
```

関連項目

F-18 ページの [CLOSE](#)（実行可能埋込み SQL）

F-35 ページの [DECLARE CURSOR](#)（埋込み SQL 宣言文）

F-57 ページの [FETCH](#)（実行可能埋込み SQL）

F-89 ページの [OPEN](#)（実行可能埋込み SQL）

REGISTER CONNECT（実行可能埋込み SQL 拡張要素）

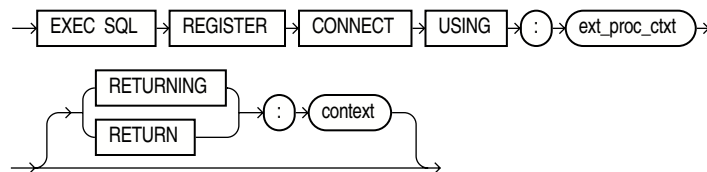
用途

外部 C プロシージャを Pro*C/C++ アプリケーションからコールできるようにします。

前提条件

なし。

構文



キーワードおよびパラメータ

ext_proc_ctxt

PL/SQL によってプロシージャに渡される外部プロシージャ・コンテキスト。OCIExtProcContext へのタイプ・ポインタです。

context

戻されるランタイム・コンテキスト。型 `sql_context` になります。現在の設定は、デフォルト（グローバル）コンテキストです。

使用上の注意

外部プロシージャを作成する方法および効果の制限についての完全な説明は、7-27 ページの「[外部プロシージャ](#)」を参照してください。

例

```
void myfunction(epctx)
OCIExtProcContext *epctx;
sql_context context;
...
{
EXEC SQL REGISTER CONNECT USING :epctx ;
EXEC SQL USE :context;
...
}
```

関連項目

なし

ROLLBACK（実行可能埋込み SQL）

用途

カレント・トランザクションで実行した作業を取り消します。

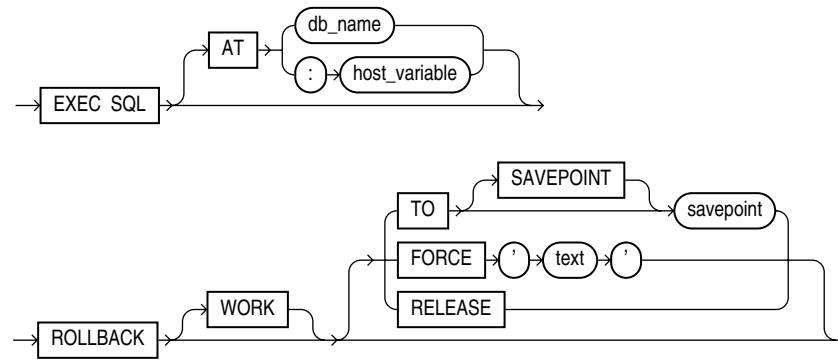
この文は、インダウトの分散トランザクションの処理を手動で取り消すときにも使用できます。

前提条件

カレント・トランザクションをロールバックするには、権限は必要ありません。

自分でコミットしたインダウトの分散トランザクションを手動でロールバックするには、FORCE TRANSACTION のシステム権限が必要です。他のユーザーがコミットしたインダウトの分散トランザクションを手動でロールバックするには、FORCE ANY TRANSACTION のシステム権限が必要です。

構文



キーワードおよびパラメータ

AT	セーブポイントが作成されるデータベースを指定します。次のいずれかを使用してデータベースを指定します。
<i>db_name</i>	事前に <code>CONNECT</code> 文で確立されたデータベース接続の名前を含む <code>NULL</code> 終了記号付き文字列。これが省略されたり、空の文字列であったときは、デフォルトのデータベース接続とみなされます。
<i>host_variable</i>	データベース接続の名前を含むホスト変数。 この句を省略した場合、セーブポイントはデフォルトのデータベースに対して作成されます。
WORK	オプションで、ANSI の互換性のために提供されます。
TO	指定したセーブポイントまでカレント・トランザクションをロールバックします。この句を省略した場合、 <code>ROLLBACK</code> 文はトランザクション全体をロールバックします。
FORCE	インダウトの分散トランザクションを手動でロールします。トランザクションは、ローカル・トランザクション ID またはグローバル・トランザクション ID が設定されたテキストにより指定します。 <code>ROLLBACK</code> 文での <code>FORCE</code> 句の使用は PL/SQL ではサポートされていません。
RELEASE	リソースをすべて解放し、アプリケーションをデータベースから切断します。 <code>RELEASE</code> 句は、 <code>SAVEPOINT</code> 句および <code>FORCE</code> 句とは併用できません。

savepoint

ロールバックするセーブポイント

使用上の注意

トランザクション（論理的な作業単位）は、Oracle8i で 1 つの単位として扱われる一連の SQL 文です。トランザクションは、COMMIT 文、ROLLBACK 文またはデータベースへの接続の後の、最初の実行 SQL 文から始まります。トランザクションは、COMMIT 文、ROLLBACK 文、またはデータベースからの切離し（意図的または不本意な切離し）により終了します。Oracle8i では、データ定義言語文の処理前および処理後に暗黙の COMMIT 文を発行することに注意してください。

TO SAVEPOINT 句を指定せずに ROLLBACK 文を使用すると、次の処理が実行されます。

- トランザクションを終了します。
- カレント・トランザクションの変更内容がすべて取り消されます。
- トランザクションのセーブポイントがすべて消去されます。
- トランザクションのロックが解除されます。

TO SAVEPOINT 句を指定して ROLLBACK 文を使用すると、次の処理が実行されます。

- トランザクションのセーブポイント後の部分のみがロールバックされます。
- 指定したセーブポイントの後に作成したセーブポイントがすべて消去されます。名前付きのセーブポイントが保持されるため、何度でも同じセーブポイントにロールバックすることができます。それ以前のセーブポイントも保持されます。
- 指定したセーブポイント後に取得した表と行のロックがすべて解除されます。セーブポイント後にロックされた行へのアクセスを要求した他のトランザクションは、コミットまたはロールバックされるまで待機する必要があります。行をまだ要求していない他のトランザクションは、ただちに行を要求し、アクセスできます。

アプリケーション・プログラムでは、COMMIT 文または ROLLBACK 文を使用してトランザクションを明示的に終了することをお勧めします。トランザクションを明示的にコミットしなかった場合にプログラムが異常終了すると、Oracle8i は最後のコミットされていないトランザクションをロールバックします。

例 I

次の文により、カレント・トランザクション全体がロールバックされます。

```
EXEC SQL ROLLBACK;
```

例 II

次の文はカレント・トランザクションをセーブポイント SP5 までロールバックします。

```
EXEC SQL ROLLBACK TO SAVEPOINT sp5;
```

分散トランザクション Oracle8i で分散オプションを使用すると、分散トランザクション、または複数データベース上のデータを変更するトランザクションが可能になります。分散トランザクションをコミットまたはロールバックするには、他のトランザクションと同じように COMMIT 文または ROLLBACK 文を発行するのみで済みます。

分散トランザクションのコミット・プロセス中にネットワーク障害が発生すると、トランザクションの状態が不明、つまりインダウトになる可能性があります。そのトランザクションに関連する他のデータベースの管理者に問い合せて、ローカル・データベースのトランザクションを手動でコミットするか、ロールバックするかを決定できます。ローカル・データベースのトランザクションを手動でロールバックするには、FORCE 句を指定して ROLLBACK 文を発行します。

インダウトのトランザクションのロールバックは、『Oracle8i 分散システム』を参照してください。

インダウトのトランザクションを手動でセーブポイントまでロールバックすることはできません。

FORCE 句を指定した ROLLBACK 文は、指定したトランザクションだけをロールバックします。このような文は、カレント・トランザクションには影響しません。

例 III

次の文はインダウトの分散トランザクションを手動でロールバックします。

```
EXEC SQL
      ROLLBACK WORK
      FORCE '25.32.87' ;
```

関連項目

F-24 ページの [COMMIT（実行可能埋込み SQL）](#)

F-99 ページの [SAVEPOINT（実行可能埋込み SQL）](#)

SAVEPOINT（実行可能埋込み SQL）

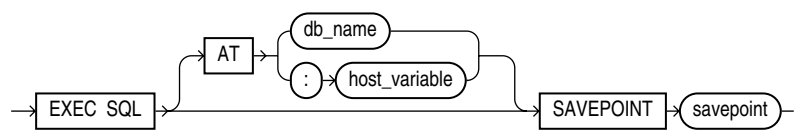
用途

後でロールバックする位置をトランザクション内に指定します。

前提条件

なし。

構文



キーワードおよびパラメータ

AT	セーブポイントが作成されるデータベースを指定します。次のいずれかを使用してデータベースを指定します。 <i>db_name</i> DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。 <i>host_variable</i> 事前に宣言した <i>db_name</i> の値を持つホスト変数。 この句を省略した場合、セーブポイントはデフォルトのデータベースに対して作成されます。
<i>savepoint</i>	作成するセーブポイントの名前。

使用上の注意

この文の詳細は、3-17 ページの「[SAVEPOINT 文の使用方法](#)」を参照してください。

例

この例では、埋込み SQL SAVEPOINT 文の使用法を示しています。

```
EXEC SQL SAVEPOINT save3;
```

関連項目

- F-24 ページの [COMMIT（実行可能埋込み SQL）](#)
- F-96 ページの [ROLLBACK（実行可能埋込み SQL）](#)

SELECT（実行可能埋込み SQL）

用途

選択した値をホスト変数に割り当てて、1 つまたは複数の表、ビューまたはスナップショットからデータを取り出します。

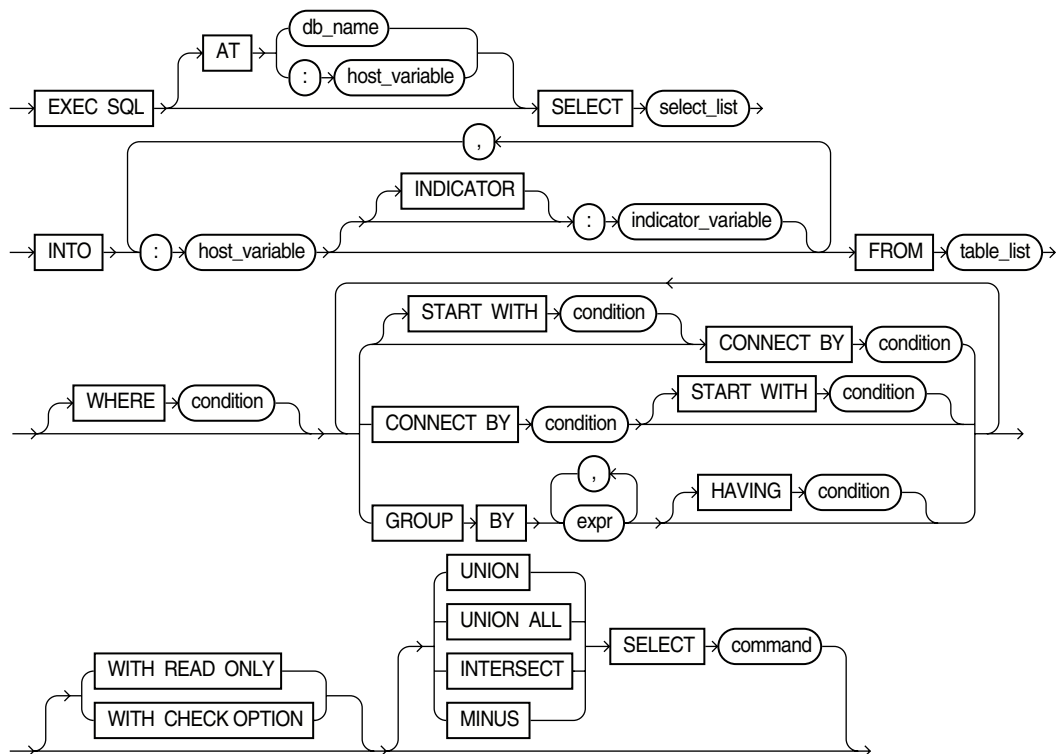
前提条件

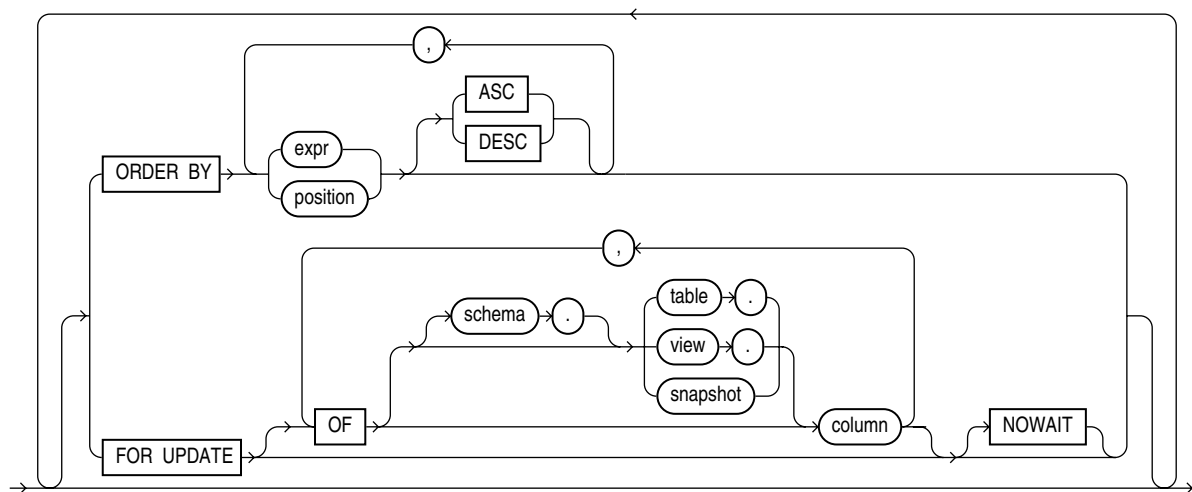
表またはスナップショットからデータを選択するには、表またはスナップショットが自分のスキーマ内にあるか、あるいは表またはスナップショットに対して SELECT の権限を持つ必要があります。

ビューのベース表から行を選択するには、ビューが属するスキーマの所有者が、ベース表に対して SELECT の権限を持つ必要があります。また、ビューが自分のスキーマ以外のスキーマ内にある場合は、ビューに対して SELECT の権限を持つ必要があります。

SELECT ANY TABLE のシステム権限を使用すると、どんな表、スナップショット、ビューのベース表からでもデータを選択できます。

構文





キーワードおよびパラメータ

AT	SELECT 文の発行先のデータベースを識別します。次のいずれかを使用してデータベースを指定します。
db_name	DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。
host_variable	事前に宣言した db_name の値を持つホスト変数。
	この句を省略した場合、SELECT 文はデフォルトのデータベースに対して発行されます。
select_list	非埋込み SELECT 文と同じです。ただし、リテラルのかわりにホスト変数を使用することができます。
INTO	SELECT 文が戻すデータを受け取る出力ホスト変数とオプションの標識変数を指定します。これらの変数は、すべてスカラーか、すべて配列である必要があります。ただし、配列は同じサイズでなくてもかまいません。
WHERE	条件が TRUE の行のみが戻されるように制限します。 『Oracle8i SQL リファレンス』の条件の構文の説明を参照してください。条件には、ホスト変数は使用できませんが、標識変数は使用できません。これらのホスト変数は、スカラーと配列のどちらでもかまいません。

その他のキーワードとパラメータは、非埋込み SQL の SELECT 文と同じです。ORDER BY 句のデフォルトは、昇順を示す ASC です。

使用上の注意

WHERE 句の条件を満たす行が存在しない場合、行は取り出されず、Oracle8i では SQLCA の SQLCODE コンポーネントを使用してエラー・コードが戻されます。

SELECT 文においてコメントを使用して、指示やヒントをオプティマイザに引き渡すことができます。オプティマイザはヒントを使用して文の実行計画を選択します。ヒントの詳細は、『Oracle8i パフォーマンスのための設計およびチューニング』を参照してください。

例

この例では、埋込み SQL SELECT 文の使用方を示しています。

```
EXEC SQL SELECT' ename, sal + 100, job
        INTO :ename, :sal, :job
        FROM emp
        WHERE empno = :empno;
```

関連項目

F-35 ページの [DECLARE CURSOR](#) (埋込み SQL 宣言文)

F-37 ページの [DECLARE DATABASE](#) (Oracle 埋込み SQL 宣言文)

F-51 ページの [EXECUTE](#) (実行可能埋込み SQL)

F-57 ページの [FETCH](#) (実行可能埋込み SQL)

F-93 ページの [PREPARE](#) (実行可能埋込み SQL)

SET DESCRIPTOR (実行可能埋込み SQL)

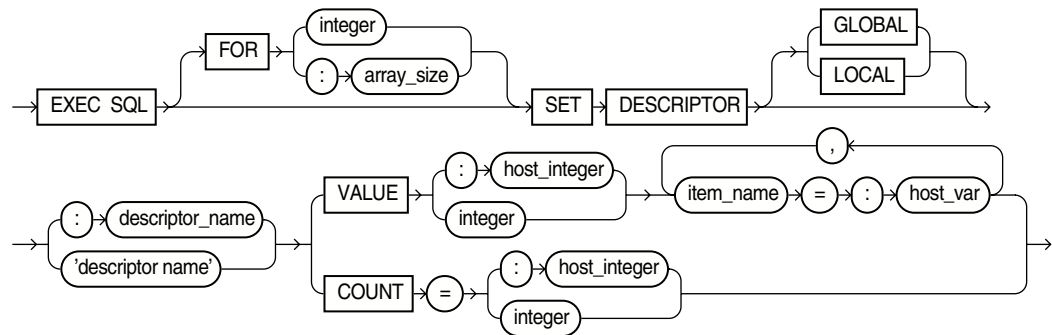
用途

この ANSI 動的 SQL 文は、ホスト変数からの情報を記述子領域に設定するために使用します。

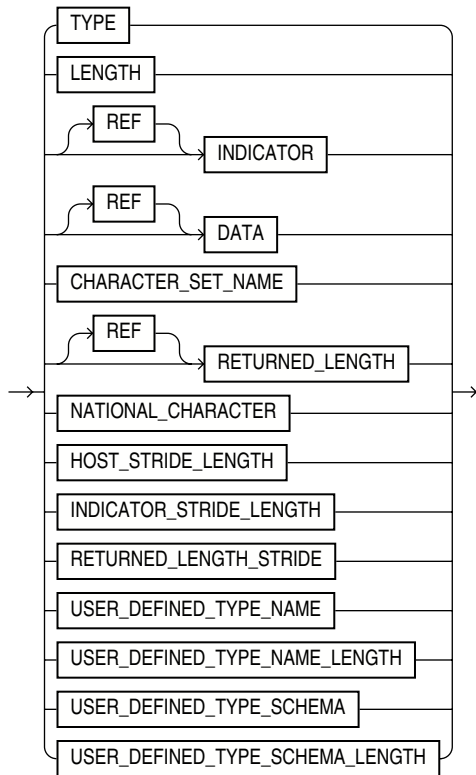
前提条件

DESCRIBE DESCRIPTOR 文の後に使用します。

構文



item_name は次のうちの 1 つになります。



キーワードおよびパラメータ

<i>array_sizeinteger</i>	処理する行を含むホスト変数。処理する行数。配列サイズ句は、DATA、RETURNED_LENGTH および INDICATOR 項目名でのみ使用できます。
GLOBAL LOCAL	GLOBAL はアプリケーションの有効範囲を示し、LOCAL (デフォルト) はファイルの有効範囲を示します。
<i>descriptor_name</i> 'descriptor name'	割り当てられた ANSI 記述子の名前が含まれるホスト変数。割り当てられた ANSI 記述子の名前。
COUNT	入力変数または出力変数の数。
VALUE	参照されるホスト変数の位置。
<i>item_name</i>	item_names のリストおよびその説明は、14-18 ページの表 14-6「SET DESCRIPTOR の記述子項目名」 および 14-19 ページの表 14-7「Oracle 拡張機能により追加された SET DESCRIPTOR の記述子項目名の定義」を参照してください。
<i>host_integerinteger</i>	項目または COUNT または VALUE を設定するのに使用するホスト変数。COUNT または VALUE を設定するのに使用する整数。
<i>host_var</i>	記述子項目を設定するのに使用するホスト変数。
REF	使用する参照セマンティクス。RETURNED_LENGTH、DATA および INDICATOR 項目名に限り使用できます。RETURNED_LENGTH を設定するのに使用する必要があります。

使用上の注意

DYNAMIC=ANSI プリコンパイラ・オプションを使用します。

クライアント側 Unicode サポートには、CHARACTER_SET_NAME を UTF16 に設定します。

記述子項目名の表を含む完全な詳細は、F-103 ページの「[SET DESCRIPTOR](#)」を参照してください。

例

```
EXEC SQL SET DESCRIPTOR GLOBAL :mydescr COUNT = 3 ;
```

関連項目

F-14 ページの [ALLOCATE DESCRIPTOR](#) (実行可能埋込み SQL)

F-34 ページの [DEALLOCATE DESCRIPTOR](#) (埋込み SQL 文)

F-46 ページの [DESCRIBE](#) (実行可能埋込み SQL 拡張要素)

F-63 ページの [GET DESCRIPTOR](#) (実行可能埋込み SQL)

F-93 ページの [PREPARE](#) (実行可能埋込み SQL)

TYPE (Oracle 埋込み SQL 宣言文)

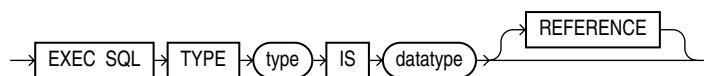
用途

ユーザー定義型の同値化を行うか、外部データ型をユーザー定義のデータ型に同値化することで、外部データ型をホスト変数のクラス全体に割り当てます。

前提条件

ユーザー定義のデータ型は、埋込み SQL プログラムで事前に宣言しておく必要があります。

構文



キーワードおよびパラメータ

type 外部データ型と同値化するユーザー定義のデータ型。

datatype プリコンパイラによって認識される内部データ型ではない外部データ型。データ型には、長さ、精度またはスケールを含めることができます。この外部データ型は、ユーザー定義の型と同値化された後、型が割り当てられているホスト変数すべてに割り当てられます。外部データ型のリストは、4-2 ページの「[Oracle のデータ型](#)」を参照してください。

REFERENCE 同値化した型をポインタ型にします。

使用上の注意

ユーザー定義のデータ型の同値化は、データ型の同値化の一種です。Pro*C/C++ プログラムでは、ユーザー定義のデータ型を同値化するには、埋込み SQL の TYPE 文を使用する必要があります。データ型の同値化は、次のいずれかの目的で使用します。

- 文字ホスト変数を自動的に NULL で終了します。
- プログラム・データをバイナリ・データとしてデータベースに格納します。

- デフォルトのデータ型のかわりに使用します。

Pro*C/C++ では、VARCHAR および VARRAW 配列がワード整列されているものとみなされます。配列型を VARCHAR または VARRAW データ型に同値化する場合、長さ +2 が 4 で割り切れる数になっていることを確認してください。

Pro*C/C++ では、ホスト変数の同値化のための埋込み SQL VAR 文もサポートされています。詳細は、5-13 ページの「[ユーザー定義型同値化](#)」を参照してください。

例 I

この例では、Pro*C/C++ のプリコンパイラ・プログラムにおける埋込み SQL TYPE 文を示します。

```
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
...
```

関連項目

F-111 ページの [VAR（Oracle 埋込み SQL 宣言文）](#)

UPDATE（実行可能埋込み SQL）

用途

表またはビューのベース表内の既存値を変更します。

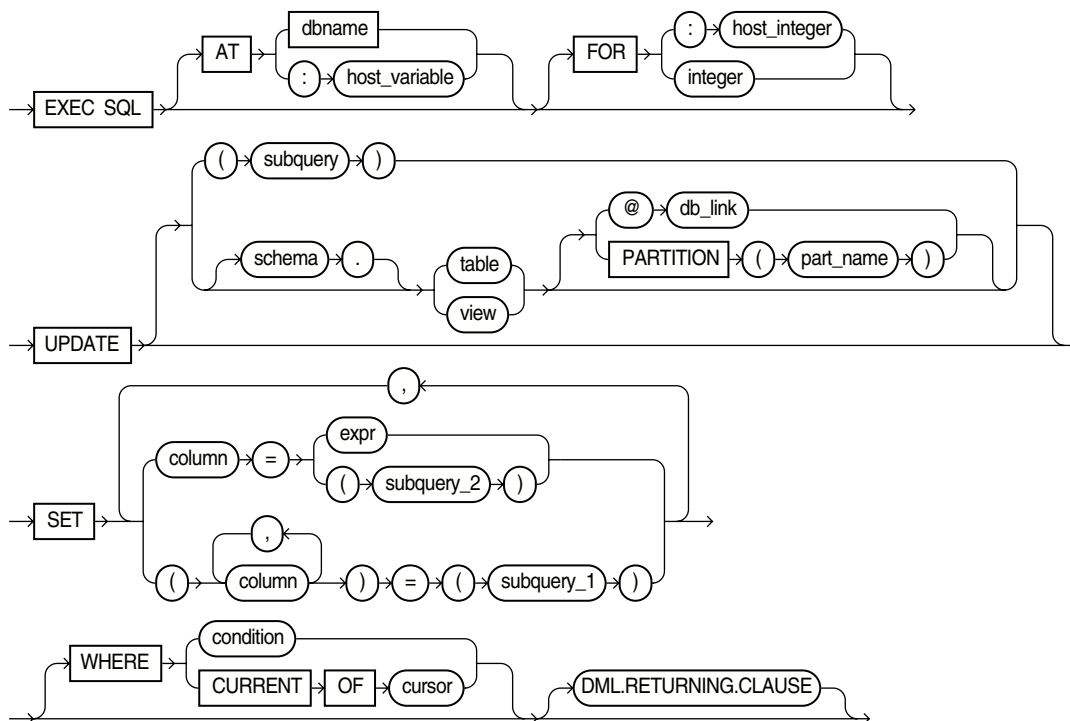
前提条件

表またはスナップショットの値を更新するには、表が自分のスキーマ内にあるか、または表に対して UPDATE の権限を持つ必要があります。

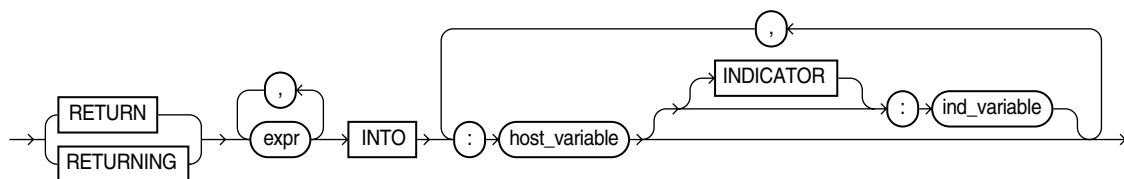
ビューのベース表の値を更新するには、ビューが属するスキーマの所有者が、ベース表に対して UPDATE の権限を持つ必要があります。また、ビューが自分のスキーマ以外のスキーマ内にある場合は、ビューに対して UPDATE の権限を持つ必要があります。

UPDATE ANY TABLE のシステム権限により、すべての表またはビューのベース表の値も更新できます。

構文



DML 戻り句は次のとおりです。



キーワードおよびパラメータ

AT

UPDATE 文の発行先のデータベースを識別します。次のいずれかを使用してデータベースを指定します。

<i>dbname</i>	DECLARE DATABASE 文を使用して事前に宣言したデータベース識別子。
<i>:host_variable</i>	事前に宣言した <i>db_name</i> の値を持つホスト変数。
	この句を省略した場合、UPDATE 文はデフォルトのデータベースに対して発行されます。
<i>FOR :host_integer integer</i>	SET 句および WHERE 句が配列ホスト変数を含む場合に、UPDATE 文を実行する回数を制限します。この句を省略すると、Oracle8i では最小の配列の各コンポーネントについて 1 回ずつ文が実行されます。
<i>schema</i>	表またはビューを含むスキーマ。 <i>schema</i> を省略した場合、Oracle8i では表またはビューが独自のスキーマにあるとみなされます。
<i>table,view</i>	更新する表の名前。ビューを指定すると、Oracle8i によりそのビューのベース表が更新されます。
<i>dblink</i>	表またはビューがあるリモート・データベースへのデータベース・リンクの完全名または部分名。データベース・リンクの参照の詳細は、『Oracle8i リファレンス・マニュアル』を参照してください。データベース・リンクを使用してリモートの表またはビューを更新できるのは、Oracle8i で分散オプションを使用している場合に限られます。
<i>part_name</i>	表のパーティションの名前。
<i>column</i>	表またはビューで更新する列の名前。SET 句の表の列を省略すると、その列の値は変更されません。
<i>expr</i>	対応する列に割り当てる新しい値。この式には、ホスト変数およびオプションの標識変数を含めることができます。『Oracle8i SQL リファレンス』の <i>expr</i> の構文を参照してください。
<i>subquery_1</i>	対応する列に割り当てられた新しい値を戻す副問合せ。副問合せの構文については、『Oracle8i SQL リファレンス』の「SELECT」を参照してください。
<i>subquery_2</i>	対応する列に割り当てられた新しい値を戻す副問合せ。副問合せの構文については、『Oracle8i SQL リファレンス』の「SELECT」を参照してください。

WHERE	更新される表またはビューの行を指定します。
<i>condition</i>	この条件が真の行のみを更新します。条件には、ホスト変数およびオプションの標識変数を使用できます。『Oracle8i SQL リファレンス』の条件の構文を参照してください。
CURRENT OF	カーソルによって最後にフェッチされた行のみを更新します。結合を実行する SELECT 文にカーソルを対応付けるには、FOR UPDATE 句 で明示的に 1 つの表のみをロックする他に方法はありません。
	この句を完全に省略した場合、表またはビューのすべての行が更新されます。
DML 戻り句	詳細は、6-10 ページの「 DML 戻り句 」を参照してください。

使用上の注意

SET 句および WHERE 句のホスト変数は、すべてスカラーか、すべて配列である必要があります。スカラーの場合、Oracle8i では UPDATE 文が 1 回のみ実行されます。配列の場合、Oracle8i では配列のコンポーネント・セットごとに 1 回ずつこの文が実行されます。1 回の実行で、0 行、1 行または複数行を更新できます。

配列ホスト変数は、サイズが異なってもかまいません。この場合、Oracle8i で文が実行される回数は、次の値のうち小さい方によって決まります。

- 最小の配列のサイズ
- オプションの FOR 句の *:host_integer* の値

更新された行の累積数は、SQLCA の SQLERRD コンポーネントの第 3 要素に設定されて戻されます。入力ホスト変数として配列を使用した場合、この数値は UPDATE 文で処理された配列のすべてのコンポーネントにおよぶ更新数の合計を示します。条件を満たす行が存在しない場合、行は更新されず、Oracle8i では SQLCA の SQLCODE 要素にエラー・メッセージが設定されて戻されます。WHERE 句を省略した場合は、すべての行が更新され、Oracle8i では SQLCA の SQLWARN 要素の第 5 コンポーネントに警告フラグを設定します。

UPDATE 文においてコメントを使用して、指示やヒントをオプティマイザに引き渡すことができます。オプティマイザはヒントを使用して文の実行計画を選択します。ヒントの詳細は、『Oracle8i パフォーマンスのための設計およびチューニング』を参照してください。

この文の詳細は、[第 6 章「埋込み SQL」](#) および [第 3 章「データベースの概念」](#) を参照してください。

例

次の例では、埋込み SQL UPDATE 文の使用方を示します。

```
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';
```

関連項目

F-37 ページの [DECLARE DATABASE \(Oracle 埋込み SQL 宣言文\)](#)

VAR (Oracle 埋込み SQL 宣言文)

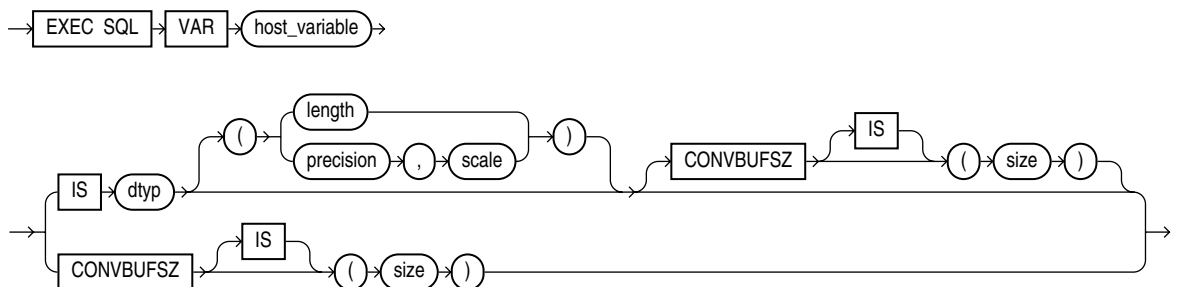
用途

ホスト変数の同値化を行うか、特定の外部データ型を個々のホスト変数に割り当て、デフォルトのデータ型割当てをオーバーライドします。また、オプションの CONVBUFSZ 句を使用して、キャラクタ・セットを変換するためのバッファ・サイズを指定します。

前提条件

ホスト変数が Pro*C/C++ プログラムで宣言済である必要があります。

構文



キーワードおよびパラメータ

<i>host_variable</i>	前に宣言された入力または出力ホスト変数（あるいはホスト表）。VARCHAR および VARRAW 外部データ型が 2 バイト長のフィールドで <i>n</i> バイトのデータ・フィールドが続く場合、 <i>n</i> の値の範囲は 1 ～ 65533 になります。そのため、 <i>type_name</i> が VARCHAR または VARRAW の場合、 <i>host_variable</i> には少なくとも 3 バイトの長さが必要です。LONG VARCHAR および LONG VARRAW 外部データ型が 4 バイト長のフィールドで <i>n</i> バイトのデータ・フィールドが続く場合、 <i>n</i> の値の範囲は 1 ～ 2147483643 になります。そのため、 <i>type_name</i> が LONG VARCHAR または LONG VARRAW の場合、 <i>host_variable</i> には少なくとも 5 バイトの長さが必要です。
<i>dtyp</i>	Pro*C/C++ によって認識される内部データ型ではない外部データ型。データ型には、長さ、精度またはスケールを含めることができます。この外部データ型が <i>host_variable</i> に割り当てられます。外部データ型のリストは、4-4 ページの「 外部データ型 」を参照してください。
<i>length</i>	データ型の長さ。有効な長さをバイト数で指定する定数式または整定数です。長さの値には、外部データ型を収容できるサイズを指定する必要があります。 <i>type_name</i> が ROWID または DATE の場合、 <i>length</i> は事前定義されているため指定できません。他の外部データ型では、 <i>length</i> はオプションです。デフォルトは <i>host_variable</i> の長さです。 <i>length</i> を指定するとき、 <i>type_name</i> が VARCHAR、VARRAW、LONG VARCHAR または LONG VARRAW の場合には、データ・フィールドの最大長を指定してください。この長さフィールドは、pro*C/C++ が指定します。 <i>type_name</i> が LONG VARCHAR または LONG VARRAW で、データ・フィールドが 65533 バイトを超える場合は、 <i>length</i> フィールドに "-1" を入れてください。
<i>precision</i> および <i>scale</i>	それぞれ有効桁数と四捨五入が実行される点を表す定数式または定数。たとえばスケールが 2 のときは、1/100 の倍数の近似値に値が四捨五入される（3.456 は 3.46 になる）ことを意味します。またスケールが -3 のときは、1000 の倍数の近似値に値が四捨五入される（3456 が 3000 になる）ことを意味します。精度は 1 ～ 99 の範囲で、スケールは -84 ～ 99 の範囲で指定できます。ただし、最大精度は 38、データベース列のスケールは 127 になります。したがって、 <i>precision</i> が 38 を超えていると、 <i>host_variable</i> の値はデータベース列に挿入できません。一方、列値のスケールが 99 を超えていると、 <i>host_variable</i> に入れる値の選択もフェッチもできません。
<i>size</i>	指定した <i>host_variable</i> から他のキャラクタ・セットへの変換に使用されるバッファのバイト単位のサイズ。定数または定数式です。ランタイム・ライブラリ内のバッファのバイト単位のサイズ。これを使用して、 <i>host_variable</i> のキャラクタ・セットを変換します。

使用上の注意

length、*precision*、*scale* および *size* は定数式になる場合があります。

ホスト変数の同値化は、データ型の同値化の一つです。次の目的には、データ型の同値化が有効です。

- 文字ホスト変数を自動的に NULL で終了します。
- プログラム・データをバイナリ・データとしてデータベースに格納します。
- デフォルトのデータ型のかわりに使用します。

size、*length*、*precision*、*scale* には、プリコンパイラの実行時に値が認識される複雑な C 定数式を任意に使用することができることに注意してください。

たとえば、次のとおりです。

```
#define LENGTH 10
...
char character set is nchar_cs ename[LENGTH+1];
exec sql var ename is string(LENGTH+1) convbufsz is (LENGTH*2);
```

また、この文ではマクロも使用することができるので注意してください。

CONVBUSZ 句を指定していないと、Oracle8 ランタイム・ライブラリが、ホスト変数のキャラクタ・サイズ (NLS_LANG で判別) とデータベース・キャラクタ・セットのキャラクタ・サイズとの割合に基づいてバッファ・サイズを自動的に決定します。これによって、LONG サイズのバッファが生成されることが時々あります。データベース表では、LONG 列は 1 列しか格納できません。複数の LONG 値が指定されると、エラーとなります。

このようなエラーを避けるには、LONG サイズよりも短い長さを使用してください。キャラクタ・セットの変換によって値が CONVBUSZ で指定した長さを超える場合は、実行時にエラーが戻されます。Pro*C/C++ プリコンパイラは、ユーザー定義のデータ型の同値化に使用できるプリコンパイラ宣言文 TYPE もサポートしています。5-12 ページの「[ホスト変数の同値化](#)」も参照してください。

例

この例では、ホスト変数 DEPT_NAME がデータ型 STRING に同値化され、ホスト変数 BUFFER がデータ型 RAW (200) に同値化されます。

```
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;
```

関連項目

F-106 ページの [TYPE（Oracle 埋込み SQL 宣言文）](#)

WHENEVER（埋込み SQL 宣言文）

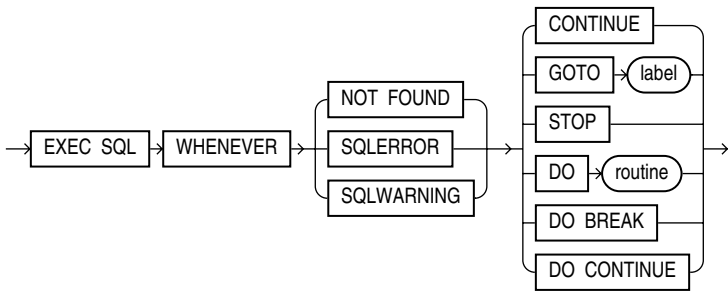
用途

埋込み SQL プログラムの実行時にエラーまたは警告が発生した場合の処置を指定します。

前提条件

なし。

構文



キーワードおよびパラメータ

NOT FOUND	SQLCODE に +1403 のエラー・コード（または MODE=ANSI の場合は +100 コード）を戻す例外条件をすべて識別します。
SQLERROR	負のリターン・コードに終わる条件を識別します。
SQLWARNING	致命的ではない警告条件を識別します。
CONTINUE	プログラムに次の文に進むように指示します。
GOTO <i>label</i>	プログラムが、ラベルによって名前が付けられている文に分岐するように指示します。
STOP	プログラムの実行を停止します。
DO <i>routine</i>	プログラムが <i>routine</i> という名前のファンクションをコールすることを示します。

DO BREAK	条件が満たされると、ループから <i>break</i> 文が実行されます。
DO CONTINUE	条件が満たされると、ループから <i>continue</i> 文が実行されます。

使用上の注意

WHENEVER 宣言文により、埋込み SQL 文でエラーまたは警告が生じた場合、プログラムの制御のエラー処理ルーチンへの移行が可能になります。

WHENEVER 宣言文の適用範囲は論理的にはなく、位置的に適用されます。WHENEVER 文は、プログラム論理の流れではなく、ソース・ファイル内で物理的に後続するすべての埋込み SQL 文に適用されます。WHENEVER 宣言文は、同じ条件をチェックする別の WHENEVER 宣言文に置換されるまで有効です。

この宣言文の詳細は、9-24 ページの「[WHENEVER 宣言文の使用](#)」を参照してください。

埋込み SQL の WHENEVER 宣言文と SQL*Plus の WHENEVER コマンドを混同しないでください。

例

次の 2 つの例では、埋込み SQL プログラムにおける WHENEVER 文の使用方法を示しています。

例 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;
...
```

例 2:

```
EXEC SQL WHENEVER SQLERROR GOTO connect_error;
...
connect_error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    printf("\nInvalid username/password\n");
    exit(1);
```

関連項目

なし

数字

2 タスク
 リンク, 2-16

A

ALLOCATE
 カーソル変数の割当て, 4-25
ALLOCATE DESCRIPTOR 文, 14-12, F-14
ALLOCATE SQL 文, 17-5, F-12
ANSI
 準拠, xxxiii
ANSI C サポート, E-2
ANSI, 住所, xxxiii
ANSI 動的 SQL, A-4
 参照セマンティクス, 14-7
 「動的 SQL (ANSI)」も参照, 14-1
ARRAYLEN 文, 7-16
ARRAYLEN 文のオプション・キーワード EXECUTE,
 7-17
AT 句
 COMMIT 文, F-25
 CONNECT 文, 3-8
 DECLARE CURSOR 宣言文, F-35
 DECLARE CURSOR 文, 3-9
 DECLARE STATEMENT 宣言文, F-38
 DECLARE STATEMENT 文, 3-10
 EXECUTE IMMEDIATE 文, 3-10, F-56
 EXECUTE 文, F-50
 INSERT 文, F-66
 SAVEPOINT 文, F-100
 SELECT 文, F-102
 SROLLBACK 文, F-97
 UPDATE 文, F-108

 使用, 3-9
 制限, 3-10
AUTO_CONNECT, 10-11
 プリコンパイラ・オプション, 3-4
AUTO_CONNECT プリコンパイラ・オプション,
 10-11

B

BFILES, 16-2
 セキュリティ, 16-2
BNF 表記法, xxxii
BREAK アクション
 WHENEVER, F-115

C

C++, 1-8
C++ アプリケーション, 12-1
CACHE FREE ALL SQL 文, 17-6
CACHE FREE ALL 文, F-16
CALL SQL 文, F-17
CALL 文, 7-26, A-2
 例, 7-26
CASE OTT パラメータ, 19-27
CHAR_MAP プリコンパイラ・オプション, 5-2,
 10-11, A-3
CHARF データ型, 4-10, 5-14
CHARZ データ型, 4-10
CHAR データ型, 4-10
CLOSE_ON_COMMIT
 プリコンパイラ・オプション, 10-12, A-5
CLOSE CURSOR 文, 14-26
CLOSE SQL 文, F-18

CLOSE 文, 6-14
動的 SQL 方法 4 での使用, 15-33
プリコンパイラ・オプションへの依存, 6-14
用途, 6-11, 6-14
例, 6-14, F-19

CODE
プリコンパイラ・オプション, 12-3

CODE OTT パラメータ, 19-26

CODE プリコンパイラ・オプション, 10-13

COLLECTION APPEND, F-19

COLLECTION APPEND 文, 18-11
SQL 文
COLLECTION APPEND, F-19

COLLECTION DESCRIBE
例, 18-19

COLLECTION DESCRIBE 文, 18-13
SQL 文
COLLECTION DESCRIBE, F-20

COLLECTION GET 文, 18-7
SQL 文
COLLECTION GET, F-22

COLLECTION RESET 文, 18-11
SQL 文
COLLECTION RESET, F-22
例, 18-20

COLLECTION SET 文, 18-9
SQL 文
COLLECTION SET, F-23
例, 18-18

COLLECTION TRIM 文, 18-12
SQL 文
COLLECTION TRIM, F-24

COMMENT 句
COMMIT 文, F-25

COMMIT SQL 文, F-24

COMMIT 文, 3-17
PL/SQL ブロック内での使用, 3-26
RELEASE オプションを含む, 3-17
影響, 3-16
トランザクションを終了, F-98
配置する場所, 3-17
用途, 3-16
例, 3-17, F-26

COMP_CHARSET プリコンパイラ・オプション,
10-14

CONFIG OTT パラメータ, 19-27

CONFIG プリコンパイラ・オプション, 10-14

CONNECT 文, F-26
AT 句, 3-8
Oracle への接続, 3-2
USING 句, 3-8
意味検査の有効化に使用, D-4
要件, 3-2
例, F-28

const
定数の宣言, 5-42

CONTEXT ALLOCATE SQL 文, F-28

CONTEXT ALLOCATE 文, 11-9

CONTEXT FREE 文, 11-10, F-29

CONTEXT OBJECT OPTION GET SQL 文, 17-18

CONTEXT OBJECT OPTION SET SQL 文, 17-17

CONTEXT USE SQL 宣言文, F-32

CONTEXT USE SQL 文, 11-9

CONTEXT USE 宣言文, 11-9

CONTINUE アクション
WHENEVER 宣言文, F-114, F-115
WHENEVER 文, 9-25
結果, 9-25

CONVBUFSZ 句, 4-49

CPP_SUFFIX
プリコンパイラ・オプション, 12-5

CPP_SUFFIX プリコンパイラ・オプション, 10-15

CREATE PROCEDURE 文
埋込み, 7-19

CURRENT OF 句, 8-4
ROWID で代用, 3-24, 8-26
制限, 6-17
用途, 6-16
例, 6-16

C 構造体
REF に対して生成, 17-32
使用, 17-31

C 構造体の使用, 17-31

C プリプロセッサ
Pro*C でサポートされる宣言文, 5-27
Pro*C での使用方法, 5-27

D

DATE データ型, 4-8

DBMS オプション, 5-14

DBMS と MODE の相互作用, 10-16

DBMS プリコンパイラ・オプション, 10-15

- dbstring の使用
 - Net8 データベース ID 指定, F-27
- DEALLOCATE DESCRIPTOR 文, 14-13, F-34
- DECLARE CURSOR 宣言文
 - 例, F-36
- DECLARE CURSOR 文, 14-24
 - AT 句, 3-9
 - 動的 SQL 方法 4 での使用, 15-23
- DECLARE DATABASE SQL 宣言文, F-37
- DECLARE STATEMENT 宣言文, F-38
- DECLARE STATEMENT 文
 - AT 句, 3-10
 - 使用例, 13-27
 - 動的 SQL での使用, 13-26
 - 必要な場合, 13-27
- DECLARE TABLE SQL 宣言文, F-39
- DECLARE TABLE 宣言文
 - 例, F-41
- DECLARE TABLE ディレクティブ
 - SQLCHECK オプションで使用, D-4
- DECLARE TABLE 文
 - AT 句と一緒に必要, 3-9
- DECLARE TYPE 宣言文, F-41
- DECLARE 文, 6-12
 - 適用範囲, F-39
 - 動的 SQL 方法 3 での使用, 13-18
 - 配置が必要な, 6-12
 - 用途, 6-11
 - 例, 6-11, F-39
- DEF_SQLCODE プリコンパイラ・オプション, 10-17
- DEFINE プリコンパイラ・オプション, 10-17
 - アプリケーションの移行に使用, 5-34
- DELETE SQL 文, F-42
- DELETE 文
 - WHERE 句を含む, 6-10
 - 埋込み SQL の例, F-45
 - ホスト配列の使用, 8-12
 - 用途, 6-10
 - 例, 6-10
- DEPT 表, 2-16
- DESCRIBE BIND VARIABLES 文
 - 動的 SQL 方法 4 での使用, 15-23
- DESCRIBE DESCRIPTOR 文, F-47
- DESCRIBE INPUT 文, 14-21
- DESCRIBE OUTPUT 文, 14-21
- DESCRIBE SELECT LIST 文
 - 動的 SQL 方法 4 での使用, 15-27

- DESCRIBE SQL 文, F-46
- DESCRIBE コマンド
 - PREPARE コマンドとともに使用, F-46
- DESCRIBE 文
 - 動的 SQL 方法 4 での使用, 13-24
 - 例, F-47
- DML 戻り句, 6-10, A-5
- DO アクション
 - WHENEVER 宣言文, F-114
 - WHENEVER 文, 9-26
 - 結果, 9-26
- DTP モデル, 5-52
- DURATION プリコンパイラ・オプション, 10-19, 17-19

E

- EMP 表, 2-16
- ENABLE THREADS SQL 文, F-49
- ENABLE THREADS 文, 11-8
- ERRORS プリコンパイラ・オプション, 10-20
- ERRTYPE
 - プリコンパイラ・オプション, 10-20
- ERRTYPE OTT パラメータ, 19-27
- ERRTYPE プリコンパイラ・オプション, 17-21
- EXEC ORACLE DEFINE 文, 5-40
- EXEC ORACLE ELSE 文, 2-14, 5-40
- EXEC ORACLE ENDIF 文, 2-14, 5-40
- EXEC ORACLE IFDEF 文, 2-14, 5-40
- EXEC ORACLE IFNDEF 文, 2-14, 5-40
- EXEC ORACLE OPTION 文
 - インラインでのオプション値の設定, 10-9
- EXEC ORACLE 文, 2-14
 - 構文, 10-9
 - 適用範囲, 10-10
 - 用途, 10-10
- EXEC SQL CACHE FREE 文, 17-6
- EXEC SQL INCLUDE
 - #include との対比, 5-33
- EXEC SQL VAR 文
 - CONVBUFSZ 句, 4-49
- EXEC SQL 句
 - SQL 埋込みのための使用, 2-4
- EXEC TOOLS
 - GET CONTEXT 文, 20-16
 - GET 文, 20-15
 - MESSAGE 文, 20-17

- SET CONTEXT 文, 20-16
- SET 文, 20-15
- EXEC TOOLS 文, 20-14
- EXECUTE DESCRIPTOR 文
 - SQL 文
 - EXECUTE DESCRIPTOR, F-53
- EXECUTE ... END-EXEC SQL 文, F-50
- EXECUTE IMMEDIATE SQL 文, F-55
- EXECUTE IMMEDIATE 文, 14-23
 - AT 句, 3-10
 - 動的 SQL 方法 1 での使用, 13-8
 - 例, F-56
- EXECUTE SQL 文, F-51
- EXECUTE 文, 14-22
 - 動的 SQL 方法 2 での使用, 13-12
 - 例, F-51, F-53
- EXPLAIN PLAN 文
 - 機能, C-6
 - 効率改善のために使用, C-5

F

- FAQ, 1-7
- FETCH DESCRIPTOR SQL 文, F-59
- FETCH SQL 文, F-57
- FETCH 文, 14-25
 - INTO 句を含む, 6-13
 - OPEN コマンドの後に使用, F-93
 - OPEN 文の後に使用する, F-90
 - 結果, 6-13
 - 動的 SQL 方法 3 での使用, 13-19
 - 動的 SQL 方法 4 での使用, 15-31
 - 用途, 6-11, 6-13
 - 例, 6-13, F-59
- FIPS フラガー
 - 宣言文の欠如を警告, 4-11
 - 配列の使用法に関する警告, 8-4
 - ポインタをホスト変数として使用した場合の警告, 5-7
- FIPS プリコンパイラ・オプション, 10-21
- FLOAT データ型, 4-6
- FORCE 句
 - COMMIT 文, F-26
 - ROLLBACK 文, F-97
- FOR UPDATE OF 句
 - 使用した行のロック, 3-22

- 使用する場合, 3-22
- 用途, 3-22
- FOR 句
 - 埋込み SQL EXECUTE DESCRIPTOR 文, F-54
 - 埋込み SQL EXECUTE 文, F-52
 - 埋込み SQL INSERT 文, F-67
 - 使用例, 8-13
 - 制限, 8-14
 - 動的 SQL 方法 4 で使用, 15-33
 - 変数が負または 0 (ゼロ) の場合, 8-14
 - ホスト配列と併用, 8-13
 - 要件, 8-14
 - 用途, 8-13
- FREE SQL 文, 17-5, F-62
- free() 関数, 15-33
 - 使用例, 15-33

G

- GENXTB フォーム
 - 実行方法, 20-12
 - ユーザー・イグジットでの使用方法, 20-12
- GENXTB ユーティリティ
 - 実行方法, 20-13
 - ユーザー・イグジットでの使用方法, 20-13
- GET DESCRIPTOR 文, 14-13
- GOTO アクション
 - WHENEVER 宣言文, F-114
 - WHENEVER 文, 9-26
 - 結果, 9-26

H

- HEADER プリコンパイラ・オプション, 5-34, 10-22
- HFILE OTT パラメータ, 19-27
- HOLD_CURSOR
 - プリコンパイラ・オプション
 - 影響される事項, C-7
 - 効率改善のために使用, C-11
- HOLD_CURSOR オプション
 - ORACLE プリコンパイラ, F-19
- HOLD_CURSOR プリコンパイラ・オプション, 10-22

I

IAF GET 文

- 構文, 20-4
- 使用例, 20-5
- ブロック名およびフィールド名の指定, 20-5
- ユーザー・イグジット, 20-4
- 用途, 20-4

IAF PUT 文

- 構文, 20-5
- 使用例, 20-6
- ブロック名およびフィールド名の指定, 20-6
- ユーザー・イグジット, 20-5
- 用途, 20-5

INAME プリコンパイラ・オプション, 10-24

INCLUDE

- SQLCA を組み込むために使用, 9-17
- 使用、プリコンパイラ・オプション, 5-33

#include

- ファイルの組込み、Pro*C の C との比較, 5-28

INCLUDE プリコンパイラ・オプション, E-3

INDICATOR キーワード, 4-15

INITFILE OTT パラメータ, 19-26

INITFUNC OTT パラメータ, 19-26

IN OUT パラメータ・モード, 7-3

INSERT SQL 文, F-65

- 例, F-68

INSERT 文

- INTO 句を含む, 6-8
- VALUES 句を含む, 6-8
- ホスト配列の使用, 8-10
- 要件, 6-8
- 用途, 6-8
- 例, 6-8
- 列リストを含む, 6-8

INTEGER データ型, 4-6

INTO 句

- FETCH DESCRIPTOR 文, F-60
- FETCH 文, 6-13, F-58
- INSERT 文, 6-8
- SELECT のかわりに FETCH を伴った, 6-12
- SELECT 文, 6-7, F-102
- 出力ホスト変数用, 6-2

INTYPE OTT パラメータ, 19-25

Intype ファイル, 19-29

- OTT 実行時の供給, 19-8
- 構造, 19-29

INTYPE プリコンパイラ・オプション, 10-25

IN パラメータ・モード, 7-3

ISO

- 準拠, xxxiii

L

LDA, 5-47

- OCI rel 8 の設定, 5-48
- リモートの複数接続, 5-48

LINES プリコンパイラ・オプション, 10-26

LNAME プリコンパイラ・オプション, 10-27

LNPROC

- VMS リンク・スクリプト, 1-9

LOB

- BFILES, 16-2
- C のロケータ, 16-6
- アクセス方法, 16-5
- 一時, 16-3
- 外部, 16-2
- 初期化, 16-7
- 内部, 16-2
- バッファリング・システム, 16-9
- ロケータ, 16-3

LOB APPEND SQL 文, F-68

LOB APPEND 文, 16-11

LOB ASSIGN SQL 文, F-69

LOB ASSIGN 文, 16-12

LOB CLOSE SQL 文, F-70

LOB CLOSE 文, 16-13

LOB COPY SQL 文, F-70

LOB COPY 文, 16-13

LOB CREATE TEMPORARY SQL 文, F-71

LOB CREATE 一時文, 16-14

LOB DESCRIBE SQL 文, F-71

LOB DISABLE BUFFERING SQL 文, F-73

LOB DISABLE BUFFERING 文, 16-15

LOB ENABLE BUFFERING SQL 文, F-73

LOB ENABLE BUFFERING 文, 16-15

LOB ERASE SQL 文, F-74

LOB ERASE 文, 16-16

LOB FILE CLOSE ALL SQL 文, F-74

LOB FILE CLOSE ALL 文, 16-17

LOB FILE SET SQL 文, F-75

LOB FILE SET 文, 16-17

LOB FLUSH BUFFER SQL 文, F-75

LOB FLUSH BUFFER 文, 16-18

LOB FREE TEMPORARY SQL 文, F-76
LOB FREE TEMPORARY 文, 16-18
LOB LOAD FROM FILE 文, 16-19
LOB LOAD SQL 文, F-77
LOB OPEN SQL 文, F-77
LOB OPEN 文, 16-20
LOB READ SQL 文, F-78
LOB READ 文, 16-21
LOB TRIM SQL 文, F-78
LOB WRITE SQL 文, F-79
LOB すべてのファイルを終了文, 16-17
LOCK TABLE 文
 NOWAIT パラメータを含む, 3-23
 使用した表のロック, 3-23
 すべてのカーソルを閉じる, 3-24
 用途, 3-23
 例, 3-23
LONG RAW データ型, 4-9
LONG VARCHAR
 datatype, 4-9
LONG VARRAW データ型, 4-9
LONG データ型, 4-7
LTYPE プリコンパイラ・オプション, 10-27

M

malloc()
 使用例, 15-29
 用途, 15-29
MAXLITERAL
 デフォルト値, 2-12
MAXLITERAL プリコンパイラ・オプション, 10-28
MAXOPENCURSORS
 プリコンパイラ・オプション
 影響される事項, C-7
 効率への影響, C-10
 複数カーソル用の, 6-12
 分割プリコンパイルのために指定, 2-15
MAXOPENCURSORS プリコンパイラ・オプション,
 10-28
MODE
 プリコンパイラ・オプション
 OPEN への影響, 6-13
MODE と DBMS の相互作用, 10-16
MODE プリコンパイラ・オプション, 10-30

N

NATIVE
 DBMS オプションの値, 10-15
NESTED TABLE, 18-2
 作成, 18-2
Net8
 Oracle への接続, 3-6
 機能, 3-5
 接続構文, 3-5
 同時接続, 3-6
NIST
 準拠, xxxiii
NIST、住所, xxxiv
NLS_CHAR プリコンパイラ・オプション, 10-31
NLS_LOCAL プリコンパイラ・オプション, 10-31
NLS (各国語サポート), 4-45, A-2
NLS パラメータ
 NLS_CURRENCY, 4-45
 NLS_DATE_FORMAT, 4-45
 NLS_DATE_LANGUAGE, 4-45
 NLS_ISO_CURRENCY, 4-45
 NLS_LANG, 4-46
 NLS_LANGUAGE, 4-45
 NLS_NUMERIC_CHARACTERS, 4-45
 NLS_SORT, 4-45
 NLS_TERRITORY, 4-45
NOT FOUND 条件
 WHENEVER 宣言文, F-114
 WHENEVER 文, 9-25
 意味, 9-25
NOWAIT パラメータ
 LOCK TABLE 文, 3-23
 影響, 3-23
 省略, 3-23
NULL
 検知, 6-4
 制限, 6-5
 挿入, 6-4
 定義, 2-5
 テスト, 6-5
 テストに sqlnul() 関数を使用, 15-16
 動的 SQL 方法 4 での取扱い, 15-16
 ハードコード, 6-4
 戻す, 6-5

NULL 終了文字列, 4-7
NUMBER データ型, 4-5
 sqlprc() 関数を使用, 15-15

O

OBJECT CREATE SQL 文, 17-9, F-80
OBJECT DELETE SQL 文, 17-11, F-81
OBJECT DEREf SQL 文, 17-10, F-82
OBJECT FLUSH SQL 文, 17-12, F-83
OBJECT GET SQL 文, 17-16, F-84
OBJECT GET 文
 例, 18-18
OBJECT RELEASE SQL 文, F-86
OBJECT SET SQL 文, 17-14, F-87
OBJECT UPDATE SQL 文, 17-11, F-88
OBJECTS プリコンパイラ・オプション, 10-32, 17-20
OCIDate, 17-33
 宣言, 17-33
ocidfn.h, 5-47
OCINumber, 17-33
 宣言, 17-33
OCI onblon() コール
 接続には使用されない, 5-47
OCI orlon() コール
 接続には使用されない, 5-47
OCIRaw, 17-33
 宣言, 17-33
OCIStrIng, 17-33
 宣言, 17-33
OCI アプリケーション
 OTT の使用, 19-17
OCI 型
 OCIDate, 17-33
 OCINumber, 17-33
 OCIRaw, 17-33
 OCIStrIng, 17-33
 埋込み SQL での使用, 17-33
 宣言, 17-33
 操作, 17-34
OCI コール, 1-7
 X/A 環境で, 5-53
 埋込み, 5-47
OCI バージョン 8, 5-43
 Pro*C/C++ への埋込み, 5-46
 SQLLIB 拡張機能, 5-43
 オブジェクトのアクセスおよび操作, 19-18

 環境ハンドルのパラメータ, 5-44
 へのインタフェース, 5-44
ONAME プリコンパイラ・オプション, 10-32
 使用上の注意, 10-33
OPEN CURSOR 文, 14-24
OPEN DESCRIPTOR SQL 文, F-91
OPEN SQL 文, F-89
OPEN 文, 6-13
 影響, 6-12
 動的 SQL 方法 3 での使用, 13-19
 動的 SQL 方法 4 での使用, 15-27
 プリコンパイラ・オプションへの依存, 6-13
 用途, 6-11, 6-12
 例, 6-12, F-91
ORACA, 9-3
 カーソル・キャッシュ統計情報の収集, 9-40
 使用例, 9-41
ORACAID コンポーネント, 9-38
ORACA プリコンパイラ・オプション, 10-33
Oracle
 Forms バージョン 4, 20-14
 Open Gateway
 ROWID データ型の使用, 4-8
 Toolset, 20-14
 データ型, 2-6
Oracle コール・インタフェース Rel 7, 5-47
Oracle 通信領域, 9-35
Oracle の名前
 形成の方法, F-12
Oracle への接続, 3-2
 Net8 を使用, 3-6
 自動接続, 3-4
 同時に, 3-6
 例, 3-2
OTT, 「オブジェクト型トランスレータ」を参照
OTT パラメータ
 CASE, 19-27
 CODE, 19-26
 CONFIG, 19-27
 ERRTYPE, 19-27
 HFILE, 19-27
 INITFILE, 19-26
 INITFUNC, 19-26
 INTYPE, 19-25
 OUTTYPE, 19-26
 SCHEMA_NAMES, 19-28

USERID, 19-25
使用場所, 19-28
OUTTYPE OTT パラメータ, 19-26
Outtype ファイル, 19-29
OTT の実行時, 19-15
OUT パラメータ・モード, 7-3

P

PAGELEN
プリコンパイラ・オプション, 10-33
PARSE
プリコンパイラ・オプション, 10-34, 12-4
PL/SQL, 1-4
AT 句を使用したブロックの実行, 3-9
PL/SQL 表, 7-4
RECORD タイプ
C 構造体に結合できない, 4-39
SQLCA の設定, 9-23
SQL との関係, 1-4
SQL との違い, 1-4
主な利点, 1-4
カーソル FOR ループ, 7-2
説明, 1-4
データベース・サーバーとの統合, 7-2
パッケージ, 7-4
プロシージャとファンクション, 7-3
無名ブロック
カーソル変数のオープンに使用, 4-27
ユーザー定義のレコード, 7-5
PL/SQL からの Java のコール, A-5
PL/SQL ブロック
プリコンパイラ・プログラムに埋め込まれる, F-50
PREFETCH プリコンパイラ・オプション, 6-15,
10-35, A-4
PREPARE SQL 文, F-93
PREPARE 文, 14-20
データ定義文への影響, 13-5
動的 SQL での使用, 13-12, 13-18
動的 SQL 方法 4 での使用, 15-23
例, F-95
Pro*C/C++ 実行可能プログラムの位置, E-3
Pro*C/C++ プリコンパイラ
NLS 用のサポート, 4-47
OTT の使用, 19-21
PL/SQL の使用, 7-6
新しいデータベース型, 17-34

一般的な使用方法, 1-3
オブジェクト・サポート, 17-1
新機能, A-1 ~ A-6
ランタイム・コンテキスト, 5-43

R

RAW データ型, 4-9
READ ONLY パラメータ
SET TRANSACTION 文, 3-21
REF
埋込み SQL での使用, 17-33
構造体, 17-32
使用, 17-32
宣言, 17-32
REFERENCE 句
TYPE 文, 5-14
REF (オブジェクトへの参照), 17-2
REGISTER CONNECT SQL 文, F-95
RELEASE_CURSOR
プリコンパイラ・オプション
影響される事項, C-7
RELEASE_CURSOR オプション
ORACLE プリコンパイラ, F-19
効率改善のために使用, C-11
RELEASE_CURSOR プリコンパイラ・オプション,
10-35
RELEASE オプション, 3-21
COMMIT 文, 3-17
ROLLBACK 文, 3-19
省略された場合, 3-21
制限, 3-19
用途, 3-17
ROLLBACK SQL 文, F-96
ROLLBACK 文, 3-20
PL/SQL ブロック内での使用, 3-26
RELEASE オプションを含む, 3-19
TO SAVEPOINT 句を含む, 3-19
影響, 3-19
エラー処理ルーチン内, 3-20
トランザクションを終了, F-98
配置する場所, 3-19
用途, 3-19
例, 3-19, F-98
ROWID
擬似列, 3-24, 4-36
CURRENT OF のかわりとして, 3-24, 8-26

汎用, 4-7, 4-36, A-5
論理値, 4-7, 4-36
ROWID データ型, 4-7

S

SAVEPOINT SQL 文, F-99
SAVEPOINT 文, F-99
 用途, 3-17
 例, 3-17, F-100
SCHEMA_NAMES OTT パラメータ, 19-28
 使用方法, 19-33
SELECT_ERROR
 プリコンパイラ・オプション, 6-8, 10-36
SELECT SQL 文, F-100
SELECT 文, 6-7
 INTO 句を含む, 6-7
 WHERE 句を含む, 6-7
 埋込み SQL の例, F-103
 使用可能な句, 6-8
 テスト, 6-8
 ホスト配列の使用, 8-4
 用途, 6-7
 例, 6-7
SET DESCRIPTOR 文, 14-17, F-103
SET TRANSACTION 文
 READ ONLY パラメータを含む, 3-21
 制限, 3-21
 要件, 3-21
 用途, 3-21
 例, 3-21
SET 句
 UPDATE 文, 6-9
 副問合せを含む, 6-9
 用途, 6-9
SQL
 埋込み SQL, 1-3
 性質, 1-3
 必要, 1-3
 利点, 1-3
SQL*Forms
 IAP 定数, 20-8
 値を戻す, 20-8
 エラー表示画面, 20-8
 逆戻りリターン・コード・スイッチ, 20-8
SQL*Forms の IAP
 用途, 20-13

SQL*Net
 バージョン 2 を使用して接続, 3-4
SQL*Plus, 1-3
 SELECT 文テストのための使用, 6-8
 対埋込み SQL, 1-3
SQL_CURSOR, F-12
SQL_SINGLE_RCTX
 定義, 5-44
 定義済の定数, 5-49
SQL92, xxxii
sqlald() 関数
 構文, 15-5
 使用例, 15-20
 用途, 15-5
sqlaldd() 関数
 「SQLSQLDAAlloc」を参照, 5-50
SQLCA, 9-2, 9-15
 PL/SQL ブロック用コンポーネント・セット, 9-23
 SQL*Net を使用している場合, 9-17
 SQLCABC コンポーネント, 9-20
 SQLCAID コンポーネント, 9-20
 sqlcode コンポーネント, 9-20
 sqlerrd, 9-22
 sqlerrmc コンポーネント, 9-21
 sqlerrml コンポーネント, 9-21
 sqlwarn, 9-23
 概要, 2-7
 コンポーネント, 9-20
 説明, 9-17
 宣言, 9-17
 複数回の組込み, 5-33
 複数使用, 9-17
 分割プリコンパイルでの使用, 2-15
 明示的チェックと暗黙的チェックの対比, 9-3
sqlca.h
 SQLCA_STORAGE_CLASS の使用, 2-15
 リスト, 9-18
SQLCAID コンポーネント, 9-20
SQLCDAFromResultSetCursor(), 5-50
SQLCDAGetCurrent, 5-50
sqlcdat()
 「SQLCDAFromResultSetCursor()」を参照, 5-50
SQLCHECK オプション
 影響される事項, D-2
 使用上の注意, 10-37
 制限, D-2

SQLCHECK プリコンパイラ・オプション, 10-37,
17-21, D-4

sqlclu() 関数

構文, 15-33

使用例, 15-33

用途, 15-33

sqlclut() 関数

「SQLSQLDAFree()」を参照, 5-50

SQLCODE

MODE=ANSI を設定, 10-30

sqlcode

SQLCA の構成要素, 9-3, 9-15

値の解釈, 9-20

SQLCODE 状態変数

SQLCA とともに宣言, 9-15

使用する場合, 9-15

宣言, 9-15

sqlcpr.h, 9-24

sqlcurt() 関数

「SQLDAToResultSetCursor()」を参照, 5-50

SQLDA

C 変数, 15-10

F 変数, 15-9

I 変数, 15-9

L 変数, 15-7

M 変数, 15-10

N 変数, 15-6

S 変数, 15-9

T 変数, 15-8

V 変数, 15-7

X 変数, 15-10

Y 変数, 15-10

Z 変数, 15-10

構造, 15-6

構造体、内容, 15-5

定義, 13-25

動的 SQL 方法 4 での使用, 15-4

内の格納情報, 13-25

バインドと選択の対比, 13-25

用途, 13-24

sqlda.h, 15-3

SQLDAToResultSetCursor(), 5-50

SQLDA の C 変数

値設定の方法, 15-10

用途, 15-10

SQLDA の F 変数

値設定の方法, 15-9

用途, 15-9

SQLDA の I 変数

値設定の方法, 15-9

用途, 15-9

SQLDA の L 変数

値設定の方法, 15-7

用途, 15-7

SQLDA の M 変数

値設定の方法, 15-10

用途, 15-10

SQLDA の N 変数

値設定の方法, 15-6

用途, 15-6

SQLDA の S 変数

値設定の方法, 15-9

用途, 15-9

SQLDA の T 変数

値設定の方法, 15-8

用途, 15-8

SQLDA の V 変数

値設定の方法, 15-7

用途, 15-7

SQLDA の X 変数

値設定の方法, 15-10

用途, 15-10

SQLDA の Y 変数

値設定の方法, 15-10

用途, 15-10

SQLDA の Z 変数

値設定の方法, 15-10

用途, 15-10

SQLDA の選択

用途, 15-3

SQLEnvGet(), 5-51

sqlerrd

コンポーネント, 9-16, 9-22

sqlerrd[2] コンポーネント, 9-21

N 行またはフェッチされた行を戻す, 8-7

データ処理文との併用, 8-6

sqlerrm

SQLCA のコンポーネント, 9-3

sqlerrmc コンポーネント, 9-21

sqlerrml コンポーネント, 9-21

SQLERROR

WHENEVER 宣言文条件, F-114

- SQLErrorGetText(), 5-50
- SQLERROR 条件
 - WHENEVER 文, 9-25
 - 意味, 9-25
- SQLExtProcError(), 5-51, 7-30
- sqlglm(), 9-24
- sqlglm() 関数, 9-23
 - 使用例, 9-24
 - パラメータ, 9-23
- sqlglmt()
 - 「SQLErrorGetText」を参照, 5-50
- sqlgls() 関数, 9-32
 - 「SQLLIB」を参照
 - 関数 SQLStmGetText, 4-21
 - サンプル・プログラム, 9-34
 - 使用例, 4-21
- sqlglst() 関数
 - 「SQLStmGetText」を参照, 5-50
- SQLIEM 関数
 - 構文, 20-8
 - ユーザー・イグジット, 20-8
 - 用途, 20-8
- sqlld2() 関数, 5-53
- sqlld2t() 関数
 - 「SQLLDAGetName」を参照, 5-50
- SQLLDAGetName, 5-50
- sqlldat() 関数
 - 「SQLCDAGetCurrent」を参照, 5-50
- SQLLIB
 - OCI の拡張相互運用性, 5-43
 - SQLCDAGetCurrent 関数, 5-50
 - SQLColumnNullCheck 関数, 5-50
 - SQLDAFree 関数, 5-50
 - SQLDAToResultSetCursor 関数, 5-50
 - SQLEnvGet 関数, 5-44, 5-51
 - SQLErrorGetText 関数, 5-50
 - SQLExtProcError 関数, 5-51
 - SQLLDAGetName 関数, 5-50
 - SQLNumberPrecV6 関数, 5-50
 - SQLNumberPrecV7 関数, 5-51
 - SQLRowidGet 関数, 5-51
 - SQLStmGetText() 関数, 5-50
 - SQLSvcCtxGet 関数, 5-45, 5-51
 - SQLVarcharGetLength 関数, 4-20
 - 埋込み SQL, 2-4
 - 関数
 - SQLCDAFromResultSetCursor, 5-50
 - 関数 SQLExtProcError, 7-30
 - 関数の新規名, A-3
 - パブリック関数の新規ファイル名, 5-49
- SQLLIB 関数
 - SQLSQLDAAlloc, 5-50
 - SQLVarcharGetLength, 5-51
- SQLLIB での SQLEnvGet 関数, 5-44
- SQLLIB での SQLSvcCtxGet 関数, 5-45
- sqlnul() 関数
 - T 変数を使用した使用方法, 15-8
 - 構文, 15-16
 - 使用例, 15-17
 - 用途, 15-16
- sqlnult() 関数
 - 「SQLColumnNullCheck()」を参照, 5-50
- SQLNumberPrecV6, 5-50
- SQLNumberPrecV7, 5-51
- sqlpr2() 関数, 15-16
- sqlpr2t() 関数
 - 「SQLNumberPrecV7」を参照, 5-51
- sqlprc() 関数, 15-15
- sqlprct() 関数
 - 「SQLNumberPrecV6」を参照, 5-50
- SQLRowidGet(), 5-51
- SQLSQLDAAlloc, 5-50
- SQLSQLDAFree(), 5-50
- SQLSTATE
 - MODE=ANSI を設定, 10-30
 - Oracle エラーへのマッピング, 9-14
 - 値, 9-4
 - クラス・コード, 9-4
 - 事前定義済のクラス, 9-7
 - 使用, 9-14
 - 状態変数, 9-2, 9-3
 - ステータス・コード, 9-14
 - 宣言, 9-4
- SQLStmGetText, 5-50
- SQLSvcCtxGet(), 5-51
- SQLVarcharGetLength, 5-51
- sqlvcpt() 関数, 「SQLLIB」を参照
 - SQLVarcharGetLength 関数, 4-20
- sqlvcpt() 関数
 - 「SQLVarcharGetLength」を参照, 5-51
- sqlwarn
 - フラグ, 9-23

- SQLWARNING
 - WHENEVER 宣言文条件, F-114
- SQLWARNING 条件
 - WHENEVER 文, 9-25
 - 意味, 9-25
- SQL 記述子領域
 - SQLDA, 13-24, 15-4
- SQL 宣言文
 - CONTEXT USE, 11-9, F-32
 - DECLARE DATABASE, F-37
 - DECLARE STATEMENT, F-38
 - DECLARE TABLE, F-39
 - DECLARE TYPE, F-41
 - TYPE, F-106
 - VAR, F-111
 - WHENEVER, F-114
- SQL 通信領域, 9-2
 - SQLCA, 9-17
- SQL、動的, 2-4
- SQL 文
 - ALLOCATE, F-12
 - ALLOCATE DESCRIPTOR TYPE, F-14
 - CACHE FREE ALL, F-16
 - CALL, 7-26, F-17
 - CLOSE, F-18
 - COMMIT, F-24
 - CONNECT, F-26
 - CONTEXT ALLOCATE, F-28
 - CONTEXT FREE, F-29
 - CONTEXT OBJECT OPTION GET, F-30
 - CONTEXT OBJECT OPTION SET, F-31
 - DEALLOCATE DESCRIPTOR, F-34
 - DELETE, F-42
 - DESCRIBE, F-46
 - DESCRIBE DESCRIPTOR, F-47
 - ENABLE THREADS, F-49
 - EXECUTE, F-51
 - EXECUTE ... END-EXEC, F-50
 - EXECUTE IMMEDIATE, F-55
 - FETCH, F-57
 - FETCH DESCRIPTOR, F-59
 - FREE, F-62
 - INSERT, F-65
 - LOB APPEND, F-68
 - LOB ASSIGN, F-69
 - LOB CLOSE, F-70
 - LOB COPY, F-70
 - LOB CREATE, F-71
 - LOB DESCRIBE, F-71
 - LOB DISABLE BUFFERING, F-73
 - LOB ENABLE BUFFERING, F-73
 - LOB ERASE, F-74
 - LOB FILE CLOSE, F-74
 - LOB FILE SET, F-75
 - LOB FLUSH BUFFER, F-75
 - LOB FREE TEMPORARY, F-76
 - LOB LOAD, F-77
 - LOB OPEN, F-77
 - LOB READ, F-78
 - LOB TRIM, F-78
 - LOB WRITE, F-79
 - OBJECT CREATE, F-80
 - OBJECT DELETE, F-81
 - OBJECT DEREf, F-82
 - OBJECT FLUSH, F-83
 - OBJECT GET, F-84
 - OBJECT RELEASE, F-86
 - OBJECT SET, F-87
 - OBJECT UPDATE, F-88
 - OPEN, F-89
 - OPEN DESCRIPTOR, F-91
 - ORACLE データ操作, 6-7
 - ORACLE データ問合せ用, 6-7
 - PREPARE, F-93
 - REGISTER CONNECT, F-95
 - ROLLBACK, F-96
 - SAVEPOINT, F-99
 - SELECT, F-100
 - SET DESCRIPTOR, F-103
 - UPDATE, F-107
 - カーソル操作, 6-7, 6-11
 - 概要, F-5
 - 型, 2-3
 - 効率改善のために最適化, C-4
 - 実行時の注意点, 6-6
 - 実行のルール, C-4
 - 実行文と宣言文, 2-3
 - トランザクションの定義および制御用, 3-15
- STOP アクション
 - WHENEVER 宣言文, F-114
 - WHENEVER 文, 9-26
 - 結果, 9-26
- STRING データ型, 4-6

SYS_INCLUDE

C++ 内のシステム・ヘッダー・ファイル, 12-5

SYS_INCLUDE プリコンパイラ・オプション, 10-37

SYSDBA/SYSOPER 権限, A-5

T

THREADS

プリコンパイラ・オプション, 10-38, 11-8

Toolset

Oracle, 20-14

TO SAVEPOINT 句

ROLLBACK 文, 3-19

制限, 3-19

用途, 3-19

TO 句

ROLLBACK 文, F-97

TYPE_CODE

プリコンパイラ・オプション, 10-39

TYPE SQL 宣言文, F-106

TYPE 宣言文

例, F-107

U

Unicode 変数, A-4

Unicode 文字セット, 5-9

UNIX

ProC アプリケーションのリンク, 1-9

UNSAFE_NULL プリコンパイラ・オプション, 10-39

UNSIGNED データ型, 4-9

UPDATE SQL 文, F-107

UPDATE 文

SET 句を含む, 6-9

WHERE 句を含む, 6-9

埋込み SQL の例, F-111

ホスト配列の使用, 8-11

用途, 6-9

例, 6-9

USERID OTT パラメータ, 19-25

USERID オプション

必要な場合, 10-40

USERID プリコンパイラ・オプション, 10-40

SQLCHECK オプションで使用, D-4

USING 句

CONNECT 文, 3-8

EXECUTE 文, 13-13

FETCH 文, F-58

OPEN 文, F-90

標識変数の使用, 13-13

用途, 13-13

V

V7

DBMS オプションの値, 10-15

VALUES 句

INSERT 文, 6-8, F-67

埋込み SQL INSERT 文, F-67

副問合せを含む, 6-9

許される値の種類, 6-8

要件, 6-8

用途, 6-8

VARCHAR

配列, 8-2

VARCHAR2 データ型, 4-5, 5-14

VARCHAR 擬似型

PL/SQL で使用するための要件, 7-11

VARCHAR データ型, 4-7

VARCHAR プリコンパイラ・オプション, 10-40

VARCHAR 変数

構造, 4-17

参照による関数への引渡しが必要, 4-20

宣言, 4-17

長さの指定, 4-18

長さの定義にマクロを使用, 5-27

長さメンバー, 4-18

文字配列との比較, 5-8

利点, 4-18

VARNUM データ型, 4-7

VARRAW データ型, 4-9

VARRAY (可変長配列)

作成, 18-3

VAR SQL 宣言文, F-111

VAR 宣言文

例, F-113

VAR 文

構文, 5-12, 5-13

VERSION プリコンパイラ・オプション, 10-41, 17-19

VMS

プリコンパイラ・アプリケーションのリンク, 1-9

W

WHENEVER SQL 宣言文, F-114

WHENEVER 宣言文

例, F-115

WHENEVER 文

CONTINUE アクション, 9-25

DO BREAK アクション, 9-26

DO CONTINUE アクション, 9-26

DO アクション, 9-26

GOTO アクション, 9-26

NOT FOUND 条件, 9-25

SQLCA の自動チェック, 9-25

SQLERROR 条件, 9-25

SQLWARNING 条件, 9-25

STOP アクション, 9-26

アドレス指定可能度の維持, 9-31

ガイドライン, 9-29

概要, 2-8

新規アクション, A-3

データの終わり条件に対処, 9-29

適用範囲, 9-29

配置する場所, 9-29

無限ループの回避, 9-30

ユーザー・イグジットでの使用方法, 20-9

例, 9-26

WHERE CURRENT OF 句

CURRENT OF 句, 6-16

WHERE 句

DELETE 文, 6-10, F-44

SELECT 文, 6-7

UPDATE 文, 6-9, F-110

検索条件, 6-10

省略された場合, 6-10

ホスト配列, 8-15

用途, 6-10

WORK オプション

COMMIT 文, F-25

ROLLBACK 文, F-97

X

XA インタフェース, 5-52

XA ライブラリでのリンク, E-3

X/Open, 5-52

アプリケーション開発, 5-52

あ

アクティブ・セット

カーソル移動, 6-13

空の場合, 6-14

識別方法, 6-11

定義, 2-7

フェッチ, 6-13

変更, 6-12, 6-13

未定義になった場合, 6-11

アプリケーション開発過程, 2-8

暗黙的な接続, 3-12

単一, 3-13

複数, 3-13

い

移行

インクルード・ファイル, 5-34

エラー・メッセージ・コード, A-6

異常終了

自動ロールバック, F-26

以前のリリースからの移行, A-6

一時オブジェクト, 17-4

位置の透過性

提供方法, 3-13

意味検査

SQLCHECK オプション, D-2

SQLCHECK オプションを使用した制御, D-2

定義, D-2

有効化, D-3

インダウト・トランザクション, 3-25

インタフェース

XA, 5-52

ネイティブ, 5-52

う

埋込み

プリコンパイラ・プログラム中の PL/SQL プ

ロック, F-50

埋込み PL/SQL

%TYPE の使用, 7-2

PL/SQL 表, 7-4

SQLCHECK オプション, 7-6

SQL へのサポート, 2-5

VARCHAR 擬似型との使用, 7-11

- カーソル FOR ループ, 7-2
- 概要, 2-5
- 効率改善のために使用, C-3
- パッケージ, 7-4
- プロシージャとファンクション, 7-3
- ユーザー定義のレコード, 7-5
- 許される場所, 7-6
- 要件, 7-6
- 利点, 7-2
- 例, 7-7, 7-8
- 埋込み SQL
 - ALLOCATE 文, F-12
 - CLOSE 文, F-18
 - CONTEXT ALLOCATE 文, 11-9, F-28
 - CONTEXT FREE 文, 11-10
 - ENABLE THREADS 文, 11-8
 - EXEC SQL CACHE FREE ALL, 17-6
 - EXECUTE 文, F-50
 - OCI 型の使用, 17-33
 - OPEN 文, F-89
 - PREPARE 文, F-93
 - REF の使用, 17-33
 - SAVEPOINT 文, F-99
 - SELECT 文, F-100
 - SQL*Plus を使ったテスト, 1-3
 - TYPE 宣言文, F-106
 - UPDATE 文, F-107
 - VAR 宣言文, F-111
 - WHENEVER 宣言文, F-114
 - 概要, 2-2
 - 構文, 2-4
 - 主要概念, 2-2
 - 使用する場合, 1-3
 - 対話型 SQL との差異, 2-4
 - 定義, 2-2
 - ホスト言語との混在, 2-4
 - 要件, 2-4
- 埋込み SQL での REF の使用, 17-33
- 埋込み SQL 文
 - 引用符の使用, 2-10
 - 終了記号, 2-13
 - 接尾辞および接頭辞は許されない, 2-9
 - 二重引用符の使用, 2-11
 - ホスト配列の参照, 8-3
 - ホスト変数の参照, 4-14
 - ラベル, 9-26

え

- 永続オブジェクト, 17-4
- エラー処理, 2-7
 - ROLLBACK 文の用途, 3-20
 - SQLCA 文と WHENEVER 文の対比, 9-3
 - 概要, 2-7
 - 代替手段, 9-2
 - 必要, 9-2
- エラー報告
 - エラー・メッセージの使用, 9-17
 - 解析エラー・オフセットの使用, 9-16
 - 警告フラグの使用, 9-16
 - 主要コンポーネント, 9-15
 - 処理済の行数の使用, 9-16
- エラー・メッセージ
 - SQLCA 内の格納場所, 9-17
 - エラー報告での使用, 9-17
 - 最大長, 9-24
 - 取得用の sqlglm() 関数使用, 9-23
- エンキュー
 - ロック, 3-14
- 演算子
 - C と SQL の対比, 2-13
 - 制限, 2-13

お

- オーバヘッド
 - 削減, C-2
- オープン
 - カーソル, F-89, F-91
 - カーソル変数, 4-26
- 大文字と小文字の区別
 - プリコンパイラ・オプション, 10-2
- オブジェクト
 - OCI を使用したアクセス, 19-18
 - OCI を使用した操作, 19-18
 - Pro*C/C++ でのオブジェクト型の使用, 17-3
 - 一時, 17-4
 - 永続, 17-4
 - 永続コピーと一時コピーの対比, 17-4
 - 概要, 17-2
 - 型, 17-2
 - サポート, 17-1
 - 参照, 17-2
- オブジェクト型, A-3

- オブジェクト型トランスレータ (OTT), A-3
 - Intype ファイルの供給, 19-8
 - Outtype ファイル, 19-15
 - Pro*C/C++ での使用, 19-21
 - コマンドライン, 19-6
 - コマンドライン構文, 19-24
 - 参照, 19-23
 - 使用, 19-1, 19-2
 - 制限, 19-36
 - データベースに型を作成, 19-4
 - デフォルト名マップ, 19-35
 - パラメータ, 19-25 ~ 19-28
- オブジェクト・キャッシュ, 17-3
- オブジェクトに対する SQLCHECK のサポート, 17-21
- オブジェクトの一時コピー, 17-4
- オブジェクトの永続コピー, 17-4
- オブジェクトへの参照 (REF)
 - 埋込み SQL での使用, 17-33
 - 使用, 17-32
 - 宣言, 17-32
- オプションの入力, 5-30, 10-9
- オプティマイザ・ヒント, C-5
 - C, 6-15
 - C++, 6-16, 12-4

か

- カーソル, 2-15, 4-24
 - アクティブ・セット内での移動, 6-13
 - 以降の行をフェッチ, F-57, F-59
 - オープン, F-89, F-91
 - カーソル変数の割当て, 4-25
 - 型, 2-7
 - クローズ, F-18
 - 再オープン, 6-12, 6-14
 - 再オープン前のクローズ, 6-13
 - 自動的にクローズされた場合, 6-14
 - 処理が効率に及ぼす影響, C-7
 - 宣言, 6-11
 - 宣言時の制限, 6-12
 - 操作用の文, 6-11
 - 定義, 2-6
 - 適用範囲, 6-12
 - 問合せとの関連付け, 6-11
 - ネーミング規則, 6-12
 - 複数行問合せ用, 6-11
 - 複数使用, 6-12

- 明示的と暗黙的の対比, 2-7
- 用途, 6-11
- 類似性, 2-7
- 割当て, F-12
- カーソル・キャッシュ
 - 定義, 9-38
 - 用途, C-9
- カーソル制御文
 - 一般的な順序の例, 6-17
- カーソル操作
 - 概要, 6-11
- カーソル変数, 4-24, F-12
 - 再帰関数での使用方法, 1-10
 - 制限, 4-30
 - 宣言, 4-25
 - 割当て, 4-25
- 解除
 - スレッド・コンテキスト, 11-10, F-29
- 解析
 - 定義, 13-3
- 解析エラー・オフセット
 - エラー報告での使用, 9-16
- 解釈方法, 9-16
- ガイドライン
 - WHENEVER 文, 9-29
 - 動的 SQL, 13-6
 - トランザクション用, 3-25
 - 分割プリコンパイル, 2-15
 - ユーザー・イグジット, 20-13
- 外部データ型
 - FLOAT, 4-6
 - INTEGER, 4-6
 - STRING, 4-6
 - 定義, 2-6
- 外部プロシージャ, A-4
 - PL/SQL からのコール, 7-27
 - エラー処理, 7-30
 - コールバック, 7-27
 - 作成, 7-29
 - 制限, 7-28
- 各国語サポート (NLS), 4-45
- 拡張子
 - デフォルト・ファイル名, 19-35
- 可変長配列, 18-3
- カレント行
 - 検索用の FETCH 使用, 6-11
 - 定義, 2-7

関数

ホスト変数に指定できない, 4-15

関数プロトタイプ

定義, 10-13

き

記号

定義, 2-14

記号の定義, 2-14

記述子, 15-4

選択記述子, 13-24

定義, 13-24

バインド記述子, 13-24

必要, 15-4

割当てに sqlald() 関数を使用, 15-5

割当ての解除に sqlclu() 関数を使用, 15-33

キャッシュ, 17-3

行

カーソル以降のフェッチ, F-57, F-59

継続, 2-12

更新, F-107

最大長, 2-12

表およびビューに挿入, F-65

行のロック

FOR UPDATE OF を使用した, 3-22

解除される場合, 3-23

効率改善のために使用, C-6

取得される場合, 3-23

利点, C-6

共用体

ホスト構造体として許されない, 4-39

ホスト構造体内にネストできない, 4-39

行を挿入できない

原因, 9-20

切捨てエラー

生成時の, 6-6

切り捨てられた値

検知, 6-4, 7-13

く

組込みファイルの位置, E-2

クローズ

カーソル, F-18

け

警告フラグ

エラー報告での使用, 9-16

結合

制限, 6-17

結合インタフェース, 17-4

使用する場合, 17-4

検索条件

WHERE 句, 6-10

定義, 6-10

こ

更新

表とビューの行, F-107

構成ファイル, 10-3

位置, 10-3

およびオブジェクト型トランスレータ, 19-5

システム, 10-4

ユーザー, 10-4

構造型

コレクション型, 18-3

構造体

C、使用, 17-31

REF の C 構造体の生成, 17-32

ネスト不可能, 4-39

配列, 8-16, A-2

ホストにネスト不可, 4-39

ホスト変数, 4-37

ホスト変数としてのポインタ, 4-44

構造体コンポーネントの位置合せ, E-2

構造体の配列, 8-16, A-2

構文、埋込み SQL, 2-4

構文検査

SQLCHECK オプションを使用した制御, D-2

定義, D-2

構文図

使用, F-9

使用される符号, F-9

説明, F-9

読み方, F-9

効率

改善のために HOLD_CURSOR を使用, C-11

改善のために RELEASE_CURSOR を使用, C-11

改善のために SQL 文を最適化, C-4

改善のために埋込み PL/SQL を使用, C-3

- 改善のために過剰な解析を排除, C-6
- 改善のために行レベル・ロックを使用, C-6
- 改善のために索引を使用, C-6
- 改善のためにホスト配列を使用, C-3
- 低下の原因, C-2
- コーディング規則, 2-9
- コード体系 (文字セットまたはコード・ページ), 4-47
- コード・ページ, 4-47
- コミット
 - 機能, 3-15
 - 自動, 3-16
 - トランザクション, F-24
 - 明示的と暗黙的の対比, 3-16
- コメント
 - ANSI, 2-9
 - PL/SQL ブロックの制限, 13-29
 - 許可, 2-9
- コレクション, A-4
 - NESTED TABLE, 18-2
 - OBJECT GET 文, 18-6
 - OBJECT SET 文, 18-6
 - VARRAY, 18-3
 - および C, 18-3
 - 記述子, 18-3
 - 自立型アクセス, 18-4
 - 操作, 18-4
 - 要素アクセス, 18-4
- コレクション・オブジェクト型
 - 処理, 18-4
- コレクション型
 - 構造体, 18-3
- コレクション型の使用, 17-32
- コレクション属性の C 型, 18-14
- コレクション属性の説明, 18-14
- コレクションの属性
 - 説明, 18-14
- コンテキスト・ブロック
 - 定義, 20-4
- コンパイル, 2-16
 - インクルード・ファイルの位置指定, 5-33

さ

サーバー

- PL/SQL との統合, 7-2
- 再構成
 - CURRENT OF 句の使用, 8-4

- 最適化のアプローチ, C-5
- 索引

- 効率改善のために使用, C-6

作成

- セーブポイント, F-99

左辺値, 4-11

参照

- ホスト配列, 8-2, 8-3

- 参照セマンティクス (ANSI 動的 SQL), 14-7

- サンプル・オブジェクト型コード, 17-24

- サンプル・データベース表

- DEPT 表, 2-16

- EMP 表, 2-16

- サンプル・プログラム

- ansidyn1.pc, 14-27

- ansidyn2.pc, 14-36

- calldemo.sql と sample9.pc, 7-22

- coldemo1.pc, 18-22

- cppdemo1.pc, 12-5

- cppdemo2.pc, 12-9

- cppdemo3.pc, 12-13

- cv_demo.pc, 4-31

- cv_demo.sql, 4-31

- extp1.pc, 7-29

- lobdemo1.pc, 16-34

- navdemo1.pc, 17-24

- oraca.pc, 9-41

- sample1.pc, 2-17

- sample2.pc, 4-41

- sample3.pc, 8-7

- sample4.pc, 5-15

- sample5.pc, 20-10

- sample6.pc, 13-9

- sample7.pc, 13-14

- sample8.pc, 13-20

- sample9.pc, 7-22

- sample10.pc, 15-36

- sample11.pc, 4-31

- sample12.pc, 15-36

- sqlvcp.pc, 4-21

- カーソル変数デモ, 4-31

- プリコンパイルする方法, 2-18

し

識別子, ORACLE

- 形成の方法, F-12

- システム・グローバル領域 (SGA), 7-19
- システム構成ファイル, 10-4, E-3
- システム固有の Oracle ドキュメント, 2-16, 3-6, 5-30, 5-53, 20-1
- システム固有の Oracle マニュアル, xxviii, 1-9
- システム固有の参照, 4-6, 10-2, 10-3, 10-25, 10-38
- システム障害
 - トランザクションへの影響, 3-16
- システム・ヘッダー・ファイル
 - 位置の指定, 12-5
- 事前定義済記号, 2-14
- 実行 SQL 文
 - グループ化, 2-3
 - 許される場所, 2-3
 - 用途, 2-3, 6-6
- 実行計画, C-4, C-6
- 実行時のタイプ・チェック, 17-21
- 自動接続, 3-4, 3-7
- 終了、プログラム
 - 通常と異常の対比, 3-21
- 出力ホスト変数
 - 定義, 6-2
 - に値を割り当てる, 6-2
- 準拠
 - ANSI, xxxiii
 - ISO, xxxiii
 - NIST, xxxiii
- 条件付きプリコンパイル, 2-14
 - 記号の定義, 5-40
 - 例, 2-14, 5-41
- 状態変数, 9-2
- 使用方法
 - スレッド・コンテキスト, 11-9, F-32
- 省略記号, xxxii
- 初期化関数
 - コール, 19-19
 - 作業, 19-21
- 処理済の行数
 - エラー報告での使用, 9-16

す

- 垂直バー, xxxii
- 数式
 - ホスト変数に指定できない, 4-15
- スケール
 - 抽出に sqlprc() 関数を使用, 15-15

- 抽出のための SQLPRC の使用, F-112
- 定義, 15-15, F-112
- 負の場合, 15-15, F-112
- ステータス・コード
 - 意味, 9-15
- ストアド・サブプログラム
 - コール, 7-21
 - 作成, 7-19
 - ストアドとインラインの対比, 7-19
 - パッケージとスタンドアロンの対比, 7-19
- ストアド・プロシージャ
 - プログラム例, 7-22
- スナップショット, 3-15
- スレッド, F-28
 - コンテキストの解放, 11-10, F-29
 - コンテキストの使用, 11-9
 - コンテキストの割当て, 11-9, F-28
 - 有効化, F-49, 11-8

せ

- 制限
 - AT 句, 3-10
 - CURRENT OF 句に対する, 6-17
 - CURRENT OF 句の使用, 8-4
 - FOR 句, 8-14
 - NULL に対する, 6-5
 - SET TRANSACTION 文に対する, 3-21
 - カーソル宣言時の, 6-12
 - コメント, 13-29
 - 入力ホスト変数に対する, 6-2
 - 分割プリコンパイル, 2-15
 - ホスト配列, 8-4, 8-10, 8-11, 8-12, 8-13
- 整数と ROWID のサイズ, E-2
- 精度
 - 指定されていない場合, 15-15
 - 抽出に sqlprc() 関数を使用, 15-15
 - 定義, 15-15
- セーブポイント
 - 作成, F-99
 - 消去される場合, 3-18
 - 定義, 3-17
 - 用途, 3-17
- セッション
 - 開始, F-26
 - 定義, 3-14

接続

- 暗黙的, 3-12
- デフォルトと非デフォルト, 3-7
- 同時, 3-11
- 明示的な接続, 3-7
- 命名, 3-7

宣言

- SQLCA, 9-17
- カーソル, 6-11
- ポインタ変数, 4-44
- ホスト配列, 8-2

宣言 SQL 文

- トランザクション, 3-15
- 許される場所, 2-3
- 用途, 2-3

宣言文, 2-3

- MODE=ANSI の場合, 5-14
- MODE=ANSI の場合に必須, 10-30
- 使用可能な文, 2-10
- 定義規則, 2-10
- 必要な場合, 2-9, 4-11
- フォーム, 2-9
- 要件, 2-10
- 用途, 2-10

選択記述子, 13-24, 15-4

- 情報, 13-24
- 定義, 13-24

選択リスト

- free() 関数の使用, 15-33
- malloc() 関数を使用, 15-29
- 定義, 6-7
- 含まれる項目数, 6-7

前方参照

- 許可されない理由, 6-12

そ

挿入

- 行を表およびビューに, F-65

た

大カッコ, xxxii

ダミー・ホスト変数

- プレースホルダ, 13-3

端末

- コード体系, 4-47

ち

中カッコ, xxxii

チューニング、効率, C-2

て

データ型

- NUMBER を VARCHAR2 に強制変換, 15-14

Oracle, 2-6

ORACLE 内部データ型の取扱い, 15-14

OTT パラメータ, 19-9

記述子内で使用するコード, 15-14

強制変換の必要性, 15-14

再設定が必要な場合, 15-14

使用の制限, 17-36

同値化, 5-11

同値化、用途, 2-6

内部, 4-2

内部データ型と外部データ型の対比, 2-6

内部データ型のリスト, 15-12

変換, 5-11

ユーザー定義タイプ同値化, F-106

データ型の同値化, 2-6

データ型同値化, 2-6

データ定義言語

トランザクション, 3-16

データの整合性, 3-12

定義, 3-14

データベース

命名, 3-7

データベース型

新しい, 17-34

データベース・リンク

INSERT 文の使用方法, F-67

格納場所, 3-13

シノニムの作成, 3-13

使用する例, 3-13

定義, 3-13

データ・ロック, 3-14

適用範囲

DECLARE STATEMENT 宣言文, F-39

EXEC ORACLE 文, 10-10

WHENEVER 文, 9-29

カーソル変数, 4-25

プリコンパイラ・オブション, 10-9

- デッドロック
 - 解除方法, 3-20
 - 定義, 3-14
 - トランザクションへの影響, 3-20
- デフォルトの接続, 3-7
- デフォルトのデータベース, 3-7
- デフォルトのファイルの拡張子, 19-35
- デリミタ
 - C と SQL の対比, 2-10
- テンポラリ LOB の作成, 16-14

と

- 問合せ
 - カーソルとの関連付け, 6-11
 - 単一行と複数行の対比, 6-7
 - 転送, 3-13
 - 複数行を戻す, 6-6
 - 不正にコードされた, 6-8
 - 分類, 6-6
 - 要件, 6-6
- 同時実行性
 - 定義, 3-14
- 同時接続, 3-6
- 同値化
 - ホスト変数の同値化, F-111
 - ユーザー定義タイプ同値化, F-106
- 動的 PL/SQL
 - 規則, 13-27
 - 動的 SQL との対比, 13-27
- 動的 SQL
 - PL/SQL の使用, 7-31
 - カーソル変数の同時使用不可能, 4-30
 - ガイドライン, 13-6
 - 概要, 13-2
 - 使用する場合, 13-2
 - 制限, 6-17
 - 長所と短所, 13-2
 - 定義, 2-4
 - データ型使用の制限, 17-36
 - 適した方法の選択, 13-6
 - 内での AT 句使用, 3-10
 - 用途, 13-2
- 動的 SQL (ANSI)
 - Oracle 拡張機能, 14-7
 - Oracle 動的との違い, 14-26
 - 概要, 14-3

- 基本, 14-2
- 参照セマンティクス, 14-7
- サンプル・プログラム, 14-27, 14-36
- バルク操作, 14-8
- プリコンパイラ・オプション, 14-10
- プリコンパイラのオプション, 14-2
- 動的 SQL 文
 - 解析, 13-3
 - 処理, 13-3
 - 静的 SQL 文との対比, 13-2
 - 定義, 13-2
 - ブレースホルダの使用, 13-3
 - ホスト配列の使用, 13-27
 - ホスト変数のバインド, 13-3
 - 要件, 13-3
- 動的 SQL 方法
 - 概要, 13-4
- 動的 SQL 方法 1
 - EXECUTE IMMEDIATE の使用, 13-8
 - PL/SQL の使用, 13-28
 - 使用, 13-8
 - 使用コマンド, 13-4
 - 説明, 13-8
 - 要件, 13-4
 - 例, 13-9
- 動的 SQL 方法 2
 - DECLARE STATEMENT の使用, 13-26
 - EXECUTE の使用, 13-12
 - PL/SQL の使用, 13-28
 - PREPARE の使用, 13-12
 - 使用コマンド, 13-5
 - 説明, 13-12
 - 要件, 13-5
 - 例, 13-14
- 動的 SQL 方法 3
 - DECLARE STATEMENT の使用, 13-26
 - DECLARE の使用, 13-18
 - FETCH の使用, 13-19
 - OPEN の使用, 13-19
 - PL/SQL の使用, 13-28
 - PREPARE の使用, 13-18
 - サンプル・プログラム, 13-20
 - 使用コマンド, 13-5
 - 使用文の順序, 13-18
 - 方法 2 との比較, 13-17
 - 要件, 13-5

動的 SQL 方法 4

- CLOSE 文の使用, 15-33
- DECLARE CURSOR 文の使用, 15-23
- DECLARE STATEMENT の使用, 13-26
- DESCRIBE の使用, 13-24
- DESCRIBE 文の使用, 15-23, 15-27
- FETCH 文の使用, 15-31
- FOR 句の使用, 13-27, 15-33
- OPEN 文の使用, 15-27
- PL/SQL の使用, 13-28
- PREPARE 文の使用, 15-23
- SQLDA の使用, 13-24, 15-4
- 概要, 13-23, 13-24
- 記述子の使用, 13-24
- 記述子の必要性, 15-4
- サンプル・プログラム, 15-36
- 使用する場合, 13-24
- 使用の前提条件, 15-11
- 使用文の順序, 13-26, 15-18
- 手順, 15-17
- ホスト配列の使用, 15-33
- 要件, 13-5, 15-2

動的文の解析

- PREPARE 文, F-93

ドット, xxxii

トランザクション

- 一部取消し, 3-17
- 開始方法, 3-15
- ガイドライン, 3-25
- コミット, F-24
- 実行中の障害, 3-16
- 自動的にロールバックされる場合, 3-16, 3-20
- 終了, 3-17
- 終了方法, 3-16
- セーブポイントによる副分割, 3-17
- 説明, 3-15
- 定義, 2-7
- 取消し, 3-19
- 内容, 2-7, 3-15
- 分散された, F-99
- 変更の確定, 3-16
- 読取り専用, 3-21
- ロール・バック, F-96
- を使用したデータベースの保護, 3-15

トランザクション処理

- 概要, 2-7
- 使用される文, 2-7

トランザクション処理モニター, 5-52

トランザクションの取消し, F-96

トレース機能

- 機能, C-5
- 効率改善のために使用, C-5

な

内部データ型

- 定義, 2-6

ナビゲーションル・アクセス・サンプル・プログラム, 17-24

に

入力ホスト変数

- 値を割り当てる, 6-2
- 制限, 6-2
- 定義, 6-2
- 許される場所, 6-2
- 用途, 6-2

ね

ネイティブ・インタフェース, 5-52

ネットワーク

- 上で通信, 3-5
- 通信量の低減, C-4
- プロトコル, 3-5

ネットワーク上で通信, 3-5

の

ノード

- カレント, 3-7
- 定義, 3-5

は

バイトの並び, E-2

配列

- 可変長, 18-3
- 使用方法を説明する章, 8-1
- 操作, 2-6
- 定義, 4-39
- バッチ・フェッチ, 8-5

- バルク操作 (ANSI 動的 SQL), 14-8
- ホスト配列, 2-6
- バインド
 - 定義, 13-3
- バインド SQLDA
 - 用途, 15-3
- バインド記述子, 13-24, 15-4
 - 情報, 13-24
 - 定義, 13-24
- バインド変数
 - 入力ホスト変数, 13-24
- パスワード
 - 実行時に変更, A-2
 - 定義, 3-2
- バッチでフェッチする
 - バッチ・フェッチ, 8-5
- バッチ・フェッチ
 - 戻される行の数, 8-6
 - 利点, 8-5
 - 例, 8-5
- パラメータ・モード, 7-3
- 汎用 ROWID, 4-7, 4-36, A-5

ひ

- ヒープ
 - 定義, 9-38
- ビュー
 - 行の更新, F-107
 - 行の挿入, F-65
- 表
 - NESTED, 18-2
 - 行の更新, F-107
 - 行の挿入, F-65
- 表から行の取出し
 - 埋込み SQL, F-100
- 表記規則
 - 説明, xxxi
 - 表記法, xxxi
- 表記法
 - 規則, xxxii
 - 表記規則, xxxii
- 表記法、BNF, xxxii
- 標識配列, 8-3
 - 使用例, 8-3
 - 用途, 8-3

- 標識変数
 - NULL 検知のための使用, 6-4
 - NULL 挿入のための使用, 6-4
 - NULL テストのための使用, 6-5
 - NULL を戻すための使用, 6-5
 - PL/SQL での使用, 7-12
 - 値の解釈, 6-3
 - 値を割り当てる, 6-3
 - ガイドライン, 4-17
 - 機能, 6-3
 - 切り捨てられた値検知のための使用, 6-4
 - 構造体, 4-40
 - 参照, 4-15
 - 宣言, 4-15, 18-4
 - 定義, 2-5
 - ホスト変数との関連付け, 6-3
 - マルチバイト文字列との使用, 4-50
 - 命名, 4-41
 - 要件, 6-4

- 標準ヘッダー・ファイル, E-2

- 表のロック
 - LOCK TABLE を使用した, 3-23
 - 影響, 3-23
 - 解除される場合, 3-24
 - 行の共有, 3-23

- ヒント
 - COST, C-5
 - DELETE 文, F-45
 - ORACLE SQL 文オプションマイザ用, 6-15
 - SELECT 文, F-103
 - UPDATE 文, F-110

ふ

- フェッチ
 - カーソル以降の行, F-57, F-59
- 副問合せ
 - SET 句中での使用, 6-9
 - VALUES 句中での使用, 6-9
 - 定義, 6-9
 - 用途, 6-9
 - 例, 6-9
- プライベート SQL 領域
 - オープニング, 2-6
 - カーソルとの関連付け, 2-6
 - 定義, 2-6
 - 用途, C-9

フラグ

警告フラグ, 9-16

プリコンパイラ・オプション

AUTO_CONNECT, 10-11

CHAR_MAP, 5-2, 10-11, A-3

CLOSE_ON_COMMIT, 6-14, 10-12

CODE, 10-13

COMP_CHARSET, 10-14

CONFIG, 10-14

CPP_SUFFIX, 10-15

DBMS, 10-15

DEF_SQLCODE, 10-17

DEFINE, 10-17

DURATION, 10-19

DYNAMIC, 14-10

ERRORS, 10-20

ERRTYPE, 10-20

FIPS, 10-21

HEADER, 10-22

HOLD_CURSOR, 10-22, 10-23

INAME, 10-24

INCLUDE, 10-24

INTYPE, 10-25

LINES, 10-26

LNAME, 10-27

LTYPE, 10-27

MAXLITERAL, 2-12, 10-28

MAXOPENCURSORS, 10-28

MODE, 10-30, 14-10

NLS_CHAR, 10-31

NLS_LOCAL, 10-31

OBJECTS, 10-32

ONAME, 10-32

ORACA, 10-33

PAGELN, 10-33

PARSE, 10-34

PREFETCH, 10-35

RELEASE_CURSOR, 10-35

SELECT_ERROR, 10-36

SQLCHECK, 10-37, 17-21

SYS_INCLUDE, 10-37

THREADS, 10-38, 11-8

TYPE_CODE, 10-39, 14-10

UNSAFE_NULL, 10-39

USERID, 10-40

VARCHAR, 10-40

VERSION, 10-41

アルファベット順のリスト, 10-6, 10-11

大文字と小文字の区別, 10-2

カレント設定値を調べる, 10-4

構成ファイル, 10-3

構文, 10-9

コマンドラインへ入力, 10-9

指定, 10-9

使用, 10-11 ~ 10-42

適用範囲, 10-6, 10-9

入力, 10-9

マイクロおよびマクロ, 10-5

優先順位, 10-4

リスト, 10-11

プリコンパイラ・オプションのカレント設定値を
調べる, 10-4

プリコンパイラ・オプションの優先順位, 10-4

プリコンパイル

条件付き, 2-14

分割, 2-15

プリコンパイル済みヘッダー・ファイル, 5-34, A-2

C++ 制限, 5-38

CODE オプション, 5-38

PARSE オプション, 5-38

プリコンパイル・ユニット, 3-2, 10-9

プリプロセッサ

EXEC ORACLE 宣言文, 5-40

例, 5-41

プリプロセッサ、サポート, 4-2

プリプロセッサ宣言文

Pro*C でサポートされない宣言文, 5-28

フル・スキャン

説明, C-6

プレースホルダ

適切な順序, 13-13

動的 SQL 文での使用, 13-3

複製, 13-13, 13-28

命名, 13-13

プログラム作成ガイドライン, 2-9

プログラムの終了

通常と異常の対比, 3-21

プロシージャ・データベース拡張要素, 7-4

分割プリコンパイル

MAXOPENCURSORS の指定, 2-15

カーソルの参照, 2-15

ガイドライン, 2-15

制限, 2-15

単一 SQLCA の使用, 2-15

分散処理

- Net8 を使用, 3-6
- サポート, 3-6

分散トランザクション, F-99

文の実行, 13-4

文レベルのロールバック

- 説明, 3-20
- デッドロックの解除, 3-20

へ

変数, 2-5

- cursor, 4-24
- 標識, 18-4
- ホスト, 18-4

ほ

ポインタ

- カーソル変数へ
- 制限, 4-25
- 定義, 4-43

ポインタ変数

- 構造体メンバーの参照, 4-44
- 参照, 4-44
- 参照値のサイズ決定, 4-44
- 宣言, 4-44

ホスト言語

- 定義, 2-2, 2-3

ホスト構造体

- 宣言, 4-37
- 配列, 4-39

ホスト配列

- DELETE 文, 8-12
- FOR 句の使用, 8-13
- INSERT 文, 8-10
- SELECT 文, 8-4
- UPDATE 文, 8-11
- WHERE 句, 8-15
- 効率改善のために使用, C-3
- サイズの一致, 8-3
- 参照, 8-2, 8-3
- 次元, 8-2
- 出力ホスト変数としての使用, 8-3
- 制限, 8-4, 8-10, 8-11, 8-12, 8-13
- 宣言, 8-2
- 動的 SQL 文での使用, 13-27

動的 SQL 方法 4 で使用, 15-33

入力ホスト変数としての使用, 8-3

有効でない場合, 8-2

利点, 8-2

ホスト・プログラム

定義, 2-2

ホスト変数, 6-2

EXECUTE 文, F-52

PL/SQL での使用, 7-7

アドレスへ設定する必要がある, 4-15

概要, 2-5

制限, 4-15

宣言, 2-10, 18-4

ダミー, 13-3

定義, 2-5

に値を割り当てる, 2-5

入力と出力の対比, 6-2

ネーミング規則, 2-12

ホスト変数の同値化, F-111

ユーザー・イグジット, 20-4

許される場所, 2-5

要件, 2-5

用途, 6-2

ま

マイクロ・プリコンパイラ・オプション, 10-5

マクロ・プリコンパイラ・オプション, 10-5

丸カッコ, xxxii

マルチスレッド・アプリケーション

サンプル・プログラム, 11-12

ユーザー・インタフェースの特徴

埋込み SQL 文と宣言文, 11-8

む

無効な使用

プリコンパイラ・プリプロセッサ, 5-31

め

明示的な接続, 3-7

説明, 3-7

単一, 3-8

複数, 3-11

命名

- SQL*Forms ユーザー・イグジット, 20-13
- カーソル, 6-12
- 選択リスト項目, 15-4
- データベース・オブジェクト, F-12
- メタデータ, 18-15

も

- モード、パラメータ, 7-3
- 文字データ, 5-2
- 文字列
 - マルチバイト, 4-49
- 文字列ホスト変数
 - 宣言, 5-7
- 戻り句, 6-10
 - DELETE, 6-10
 - INSERT 文, 6-10
 - UPDATE 文, 6-9

や

- 山カッコ, xxxii

ゆ

- 有効化
 - スレッド, 11-8
- ユーザー・イグジット, E-3
 - GENXTB フォームの実行, 20-12
 - GENXTB ユーティリティの実行, 20-13
 - IAF GET 文の使用法, 20-5
 - IAF PUT 文の使用法, 20-6
 - IAP ヘリンク, 20-13
 - SQL*Forms トリガーからコール, 20-6
 - WHENEVER 文の使用法, 20-9
 - 一般的な使用法, 20-3
 - ガイドライン, 20-13
 - 開発手順, 20-3
 - 使用可能な文の種類, 20-4
 - パラメータを渡す, 20-7
 - 変数の要件, 20-4
 - 命名, 20-13
 - リターン・コードの意味, 20-8
 - 例, 20-9
- ユーザー構成ファイル
 - プリコンパイラ・オブションを設定する, 10-4

ユーザー・セッション

- 定義, 3-14
- ユーザー定義タイプ同値化, F-106
- ユーザー定義のストアド・ファンクション
 - WHERE 句中での使用, 6-10
- ユーザー定義のレコード, 7-5
- ユーザー名
 - 定義, 3-2

よ

- よく聞かれる質問 (FAQ), 1-7
- 読取り専用トランザクション
 - 終了方法, 3-21
 - 説明, 3-21
 - 例, 3-22
- 読取りの一貫性
 - 定義, 3-15
- 予約語およびキーワード, B-2
- 予約名前領域, B-4

ら

- ラージ・オブジェクト (LOB), A-4
- ラベル名
 - 最大長, 9-26
- ランタイム・コンテキスト
 - 確立, 5-43
 - 終了, 5-43

り

- リソース・マネージャ, 5-52
- リターン・コード
 - ユーザー・イグジット, 20-8
- リモート・データベース
 - 宣言, F-37
- リンク, 2-16
 - 2 タスク, 2-16
 - UNIX, 1-9
 - VMS, 1-9
 - データベース・リンク, 3-13

れ

- 例外、PL/SQL
 - 定義, 7-13
- レコード, 7-5
- 列リスト
 - INSERT 文, 6-8
 - 省略が可能な場合, 6-8

ろ

- ロールバック
 - 機能, 3-15
 - 自動, 3-20
 - 文レベル, 3-20
- ロール・バック
 - 同じセーブポイントへ複数回ロール・バック, F-98
 - セーブポイントへ, F-99
- ロールバック・セグメント
 - 機能, 3-15
- ログイン, 3-2
- ログイン・データ領域, 5-47
- ロック, 3-22
 - FOR UPDATE OF を伴った, 3-22
 - LOCK TABLE 文を使用した, 3-23
 - ROLLBACK 文により解除, F-98
 - 取得用の権限, 3-26
 - 定義, 3-14
 - デフォルトの無効化, 3-22
 - 表と行の対比, 3-22
 - 明示的と暗黙的の対比, 3-22
 - モード, 3-14
 - 用途, 3-22

わ

- 割当て
 - カーソル, F-12
 - カーソル変数, 4-25
 - スレッド・コンテキスト, 11-9, F-28

