

Pro*C/C++ プリコンパイラ

プログラマーズ・ガイド Vol.1

リリース 8.1

Pro*C/C++ プリコンパイラ・プログラマーズ・ガイド Vol.1 リリース 8.1

部品番号：A62730-1

第1版 1999年5月（第1刷）

原本名：Pro*C/C++ Precompiler Programmer's Guide, Volume 1, Release 8.1.5

原本部品番号：A68021-01

原本著者：Jack Melnick, Tom Portfolio, Tim Smith

グラフィック・デザイナー：Valarie Moore

原本協力者：Ruth Baylis, Paul Lane, Julie Basu, Brian Becker, Beethoven Cheng, Michael Chiocca, Pierre Dufour, Nancy Ikeda, Thomas Kurian, Shiao-Yen Lin, Jacqui Pons, Ajay Papat, Ekkehard Rohwedder, Pamela Rothman, Alan Thiesen, Gael Turk Stevens

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

Printed in Japan.

制限付権利の説明

プログラムの使用、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。

当ソフトウェア（プログラム）のリバース・エンジニアリング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 Legend が適用されます。

Restricted Rights Legend

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication and disclosure of the Programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-14, Rights in Data -- General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

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	xxix
このマニュアルの説明事項	xxx
このマニュアルの対象	xxx
Pro*C/C++ マニュアルの構成	xxx
このマニュアルの表記規則	xxxiii
BNF 表記法	xxxiv
ANSI/ISO 準拠	xxxiv
要件	xxxiv
工業規格への Oracle Pro*C/C++ プリコンパイラの準拠性	xxxv
準拠性	xxxvi
準拠の証示	xxxvi
FIPS フラガー	xxxvi
以前のリリースからのアプリケーションの移行	xxxvii
1 序説	
Oracle プリコンパイラとは何か	1-2
なぜ Oracle Pro*C/C++ プリコンパイラを使うのか	1-3
なぜ SQL を使うのか	1-3
なぜ PL/SQL を使うのか	1-4
Pro*C/C++ プリコンパイラの利点	1-4
よく聞かれる質問 (FAQ)	1-6
2 プリコンパイラの場合	
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	2-5
静的 SQL 文と動的 SQL 文の対比	2-6
埋込み PL/SQL ブロック	2-6
ホスト変数および標識変数	2-6
Oracle のデータ型	2-7
配列	2-7
データ型の同値化	2-8
プライベート SQL 領域およびカーソル、アクティブ・セット	2-8
トランザクション	2-8
エラーおよび警告	2-9
埋込み SQL アプリケーションの開発ステップ	2-10
プログラミングのガイドライン	2-11
コメント	2-11
定数	2-11
宣言節	2-11
デリミタ	2-12
ファイルの長さ	2-13
関数プロトタイプ	2-13
ホスト変数名	2-14
行の継続	2-14
行の長さ	2-14
MAXLITERAL デフォルト値	2-14
演算子	2-14
文終了記号	2-15
条件付きプリコンパイル	2-15
記号の定義	2-16
例	2-16
分割プリコンパイル	2-17
ガイドライン	2-17
コンパイルとリンク	2-18
サンプル表	2-18
サンプル・データ	2-19
サンプル・プログラム : 単純な問合せ	2-19

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
高度な接続オプション.....	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守る方法.....	3-15
トランザクションの開始・終了方法.....	3-15
COMMIT 文の使い方.....	3-16
SAVEPOINT 文の使い方.....	3-17
ROLLBACK 文の使い方.....	3-18
文レベルのロールバック.....	3-20
RELEASE オプションの使用法.....	3-20
SET TRANSACTION 文の使い方.....	3-21
デフォルト・ロックのオーバーライド.....	3-22
FOR UPDATE OF の使い方.....	3-22
LOCK TABLE の使い方.....	3-23
COMMIT をまたぐ FETCH.....	3-23
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.....	3-25
PL/SQL の使用.....	3-25

4 データ型とホスト変数

オラクルのデータ型.....	4-2
内部データ型.....	4-2
外部データ型.....	4-3
ホスト変数.....	4-10

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	4-16
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	4-18
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	4-19
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	4-20
カーソル変数	4-24
カーソル変数の宣言	4-24
カーソル変数の割当て	4-25
カーソル変数のオープン	4-25
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）	4-28
制限	4-29
サンプル・プログラム	4-30
CONTEXT 変数	4-33
ユニバーサル ROWID	4-35
関数 SQLRowidGet()	4-36
ホスト構造体	4-37
ホスト構造体と配列	4-38
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	4-39
サンプル・プログラム : カーソルとホスト構造体	4-40
ポインタ変数	4-43
ポインタ変数の宣言	4-43
ポインタ変数の参照	4-43

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	4-47
環境変数 NLS_NCHAR	4-48
VAR 内の CONVBUFFSZ 句	4-48
埋込み SQL 内の文字列	4-48
文字列の制限事項	4-49
標識変数	4-49

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	5-11
ユーザ定義型同値化	5-13
CHARF 外部データ型	5-14
EXEC SQL VAR と TYPE ディレクティブの利用	5-14
Sample4.pc: データ型の同値化	5-14
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・ディレクティブ	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 に使うことができない SQL 文	5-32
SQLCA および ORACA、SQLDA の組込み	5-32
EXEC SQL INCLUDE および #include の要約	5-33
定義済みマクロ	5-33
インクルード・ファイル	5-34

プリコンパイル済みのヘッダー・ファイル.....	5-34
プリコンパイル済みのヘッダー・ファイルの作成.....	5-34
プリコンパイル済みのヘッダー・ファイルの使用.....	5-35
例.....	5-35
オプションの効果.....	5-37
使用上の注意.....	5-39
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.....	5-40
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
実行時コンテキストと 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.....	5-43
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.....	5-47
リモートの複数接続.....	5-48
SQLLIB バブリック関数の新しい名前.....	5-48
X/Open アプリケーションの開発.....	5-51
Oracle 固有の項目.....	5-52

6 埋込み SQL

ホスト変数の使用方法.....	6-2
出力ホスト変数および入力ホスト変数.....	6-2
標識変数の使用方法.....	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 オプション.....	6-15
オプティマイザ・ヒント.....	6-15
ヒントの発行.....	6-15
CURRENT OF 句の使用法.....	6-16
制限.....	6-16
すべてのカーソル文の使用法.....	6-17
完全な例.....	6-18

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	7-11
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 のカーソル使い方	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.....	7-20
ストアド PL/SQL または Java サブプログラムのコール.....	7-22
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	7-28
外部プロシージャの制限.....	7-29
外部プロシージャの作成.....	7-29
SQLExtProcError 関数	7-30
動的 SQL の使用	7-31

8 ホスト配列

どうして配列を使うのか?	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 文での配列の使用方法	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
<i>sqlca.sqlerrd[2]</i> の使用方法.....	8-6
FETCH される行数	8-6
Sample Program 3: ホスト配列	8-7
制限.....	8-9
NULL 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
構造体の配列	8-16
構造体の配列の使用方法	8-16
構造体の配列の制限	8-16
構造体の配列の宣言	8-17
標識変数の使用方法	8-18
構造体の配列へのポインタの宣言	8-19
例	8-20
CURRENT OF の疑似実行	8-25

9 実行時エラーの処理

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 通信領域	9-2
SQLSTATE 状態変数	9-3
SQLSTATE の宣言	9-4
SQLSTATE の値	9-4
SQLSTATE の使用	9-13
SQLCODE の宣言	9-14
SQLCA を使用したエラー報告の主要コンポーネント	9-14
ステータス・コード	9-14
警告フラグ	9-15
処理済み行数	9-15
解析エラー・オフセット	9-15
エラー・メッセージ・テキスト	9-16
SQL 通信領域 (SQLCA) の使用	9-16

SQLCA の宣言	9-16
ORACA に含まれているもの	9-17
SQLCA の構造	9-19
PL/SQL の考慮事項	9-22
エラー・メッセージの全文の取得	9-22
WHENEVER 文の使用	9-24
条件	9-24
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	9-28
WHENEVER のガイドライン	9-29
SQL 文のテキスト取得	9-31
制限	9-33
サンプル・プログラム	9-34
ORACLE 通信領域 (ORACA) の使用	9-34
ORACA の宣言	9-34
ORACA を使用可能にする	9-34
ORACA に含まれているもの	9-35
ランタイム・オプションの選択	9-37
ORACA の構造	9-37
ORACA の使用例	9-40

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
オプション値の優先順位	10-3
マクロ・オプションおよびマイクロ・オプション	10-5
構成ファイル	10-5
プリコンパイル中になにが起きているか?	10-6
オプションの適用範囲	10-6
早見表	10-7
オプションの入力	10-9
コマンド行	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	10-13
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	10-29
NLS_CHAR	10-30
NLS_LOCAL	10-31
OBJECTS	10-31
ONAME	10-32
ORACA	10-33
PAGELEN	10-33
PARSE	10-33

PREFETCH.....	10-34
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

スレッドとは?.....	11-2
Pro*C/C++ の実行時コンテキスト	11-2
実行時コンテキストの使用モデル.....	11-5
単一の実行時コンテキストを共有する複数のスレッド.....	11-5
複数の実行時コンテキストを共有する複数のスレッド.....	11-6
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 文とディレクティブ	11-8
例.....	11-10
プログラミングで考慮すべき点.....	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	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.....	12-6
cppdemo1.pc.....	12-6

cppdemo2.pc	12-9
cppdemo3.pc	12-13

13 Oracle 動的 SQL

動的 SQL とは？.....	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.....	13-18
PREPARE.....	13-18
DECLARE.....	13-19
OPEN	13-19
FETCH	13-19
CLOSE.....	13-20
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.....	13-24
SQLDA の必要性.....	13-24
DESCRIBE 文	13-25
SQLDA とは？.....	13-25
Oracle 方法 4 の実行	13-26
制限.....	13-27
DECLARE STATEMENT 文の使用方法	13-27
ホスト配列の使用方法.....	13-27

PL/SQL の使用方法	13-28
方法 1 の場合	13-28
方法 2 の場合	13-28
方法 3 の場合	13-29
Oracle 方法 4 の場合	13-29
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・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
リファレンス・セマンティクス	14-7
配列を使った一括操作	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	14-19
DESCRIBE INPUT	14-20
DESCRIBE OUTPUT	14-21
EXECUTE	14-22
EXECUTE IMMEDIATE の使用	14-23
DYNAMIC DECLARE CURSOR の使用	14-23
OPEN カーソル	14-24
FETCH	14-25
動的カーソルの CLOSE	14-25
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	14-35

15 Oracle 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	15-8
F 変数	15-9
S 変数	15-9
M 変数	15-9
C 変数	15-10
X 変数	15-10
Y 変数	15-10
Z 変数	15-10
予備知識	15-10
データの変換	15-11
データ型の強制変換	15-13
NULL/NOT NULL データ型の処理	15-16
基本手順	15-17
各手順の詳細	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
問合せのテキストをホスト文字列に設定する	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
ブレースホルダの最大数の再設定	15-26
バインド変数の値の取得と記憶領域の割当て	15-26
カーソルの OPEN	15-28
選択リストの DESCRIBE	15-28
選択リスト項目の最大数の再設定	15-30
各選択リスト項目の長さとデータ型の再設定	15-30
アクティブ・セットからの行の FETCH	15-33
選択リストの値の取得と処理	15-33
記憶領域の割当て解除	15-35
カーソルの CLOSE	15-35
ホスト配列の使用	15-35
sample12.pc	15-38
サンプル・プログラム : 動的 SQL 方法 4	15-38

16 ラージ・オブジェクト (LOB)

LOB とは ?	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
一時 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	16-7
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	16-12
COPY	16-13
CREATE TEMPORARY	16-14
DISABLE BUFFERING	16-15
ENABLE BUFFERING	16-15
ERASE	16-16
FILE CLOSE ALL	16-16
FILE SET	16-17
FLUSH BUFFER	16-18
FREE TEMPORARY	16-18
LOAD FROM FILE	16-19
OPEN	16-20
READ	16-21
TRIM	16-22
WRITE	16-23
DESCRIBE	16-24
LOB およびナビゲーション・インタフェース	16-27
一時オブジェクト	16-27
永続オブジェクト	16-27
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	16-29
BLOB の READ およびファイル書込みの例	16-29
ファイルの読み込みおよび BLOB の WRITE の例	16-30
lobdemo1.pc	16-33

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
永続的オブジェクト対一時的コピー	17-4
アソシエイティブ・インタフェース	17-4
アソシエイティブ・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
ナビゲーションル・インタフェース	17-7
ナビゲーションル・インタフェースを使う場合.....	17-8
ナビゲーション文に使われるルール.....	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	17-21
結合アクセス.....	17-22

ナビゲーション・アクセス.....	17-23
ナビゲーション・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.....	17-32
OCIDate、OCISString、OCINumber と OCIRaw の使用.....	17-33
OCIDate、OCISString、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.....	17-37

18 コレクション

コレクション.....	18-2
ネストした表.....	18-2
VARRAY.....	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とは何か	19-2
データベースにおける型の作成	19-4
OTT の呼出し	19-5
OTT コマンド行	19-6
Intype ファイル	19-7
OTT データ型のマップ	19-9
NULL 標識構造体	19-15
Outtype ファイル	19-15
OCI アプリケーションでの OTT の使用方法	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 の使用方法	19-21
OTT 参照	19-23
OTT コマンド行構文	19-24
OTT パラメータ	19-25
OTT パラメータの位置	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・イグジットとは何か	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	20-7
IAP 定数	20-8
SQLEEM 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	20-9
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	20-13
ホスト変数の使用方法	20-13
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	20-16

A 新機能

構造体の配列	A-2
プリコンパイル済みヘッダー・ファイル	A-2
CALL 文	A-2
実行時のパスワードの変更	A-2
各国文字キャラクタ・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-5
PL/SQL からの Java のコール	A-5
DML 戻り句	A-5
ユニバーサル 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
以前のリリースからの移行	A-5
文字列	A-6
エラー・メッセージ・コード	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とは	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-6

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 文およびディレクティブ

プリコンパイラのディレクティブと埋込み SQL 文の概要	F-5
文の説明について	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-10
構文ループ	F-11
複数パーツの図	F-11

データベース・オブジェクト	F-11
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-15
CALL (実行可能埋込み SQL)	F-16
CLOSE (実行可能埋込み SQL)	F-17
CONNECTION APPEND (実行可能埋込み SQL 拡張要素)	F-18
CONNECTION DESCRIBE (実行可能埋込み SQL 拡張要素)	F-19
CONNECTION GET (実行可能埋込み SQL 拡張要素)	F-21
CONNECTION RESET (実行可能埋込み SQL 拡張要素)	F-21
COLLECTION SET (実行可能埋込み SQL 拡張要素)	F-22
COLLECTION TRIM (実行可能埋込み SQL 拡張要素)	F-23
COMMIT (実行可能埋込み SQL)	F-23
CONNECT (実行可能埋込み SQL 拡張要素)	F-25
CONTEXT ALLOCATE (実行可能埋込み SQL 拡張要素)	F-27
CONTEXT FREE (実行可能埋込み SQL 拡張要素)	F-28
CONTEXT OBJECT OPTION GET (実行可能埋込み SQL 拡張要素)	F-29
CONTEXT OBJECT OPTION SET (実行可能埋込み SQL 拡張要素)	F-30
CONTEXT USE (Oracle 埋込み SQL ディレクティブ)	F-31
DEALLOCATE DESCRIPTOR (埋込み SQL 文)	F-33
DECLARE CURSOR (埋込み SQL ディレクティブ)	F-34
DECLARE DATABASE (Oracle 埋込み SQL ディレクティブ)	F-36
DECLARE STATEMENT (埋込み SQL ディレクティブ)	F-37
DECLARE TABLE (Oracle 埋込み SQL ディレクティブ)	F-38
DECLARE TYPE (Oracle 埋込み SQL ディレクティブ)	F-40
DELETE (実行可能埋込み SQL)	F-41
DESCRIBE (実行可能埋込み SQL 拡張要素)	F-45
DESCRIBE DESCRIPTOR (実行可能埋込み SQL)	F-46
ENABLE THREADS (実行可能埋込み SQL 拡張要素)	F-48
EXECUTE ... END-EXEC (実行可能埋込み SQL 拡張要素)	F-49
EXECUTE (実行可能埋込み SQL)	F-50
EXECUTE DESCRIPTOR (実行可能埋込み SQL)	F-52
EXECUTE IMMEDIATE (実行可能埋込み SQL)	F-54
FETCH (実行可能埋込み SQL)	F-56
FETCH DESCRIPTOR (実行可能埋込み SQL)	F-58
FREE (実行可能埋込み SQL 拡張要素)	F-61
GET DESCRIPTOR (実行可能埋込み SQL)	F-62

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-68
LOB CLOSE (実行可能埋込み SQL 拡張要素)	F-69
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-70
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-72
LOB ENABLE BUFFERING (実行可能埋込み SQL 拡張要素)	F-72
LOB ERASE (実行可能埋込み SQL 拡張要素)	F-73
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-74
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-75
LOB LOAD (実行可能埋込み SQL 拡張要素)	F-76
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-77
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-78
OBJECT CREATE (実行可能埋込み SQL 拡張要素)	F-79
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-85
OBJECT SET (実行可能埋込み SQL 拡張要素)	F-86
OBJECT UPDATE (実行可能埋込み SQL 拡張要素)	F-87
OPEN (実行可能埋込み SQL)	F-88
OPEN DESCRIPTOR (実行可能埋込み SQL)	F-90
PREPARE (実行可能埋込み SQL)	F-92
REGISTER CONNECT (実行可能埋込み SQL 拡張要素)	F-94
ROLLBACK (実行可能埋込み SQL)	F-95
SAVEPOINT (実行可能埋込み SQL)	F-98
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 ディレクティブ)	F-106
UPDATE (実行可能埋込み SQL)	F-108
VAR (Oracle 埋込み SQL ディレクティブ)	F-112

WHENEVER (埋込み SQL ディレクティブ).....	F-115
---------------------------------	-------

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++ プリコンパイラのレファレンスとして作業中にお使いいただけます。ここでは、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
- このマニュアルの対象
- Pro*C/C++ マニュアルの構成
- このマニュアルの表記規則
- ANSI/ISO 準拠

このマニュアルの説明事項

このガイドでは、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紹介します。また、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のノウハウを説明します。さらに、できるだけ短時間で埋込み SQL プログラムの作成方法を習得できるように援助します。

このガイドの大きな特徴の 1 つは、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。これらのツールを最大限に活用するために、このマニュアルではプログラムのパフォーマンスを向上する方法を含めた作業のこつをすべて掲載しています。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。

このマニュアルの対象

このガイドは、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マ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。このガイド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

Volume 1 では、Pro*C/C++ プログラミングの基本について説明します。多くの Pro*C/C++ 開発者は、これらの章を読めば便利かつ強力な Pro*C/C++ アプリケーションが書けるようになります。

後半の章では、Pro*C/C++ についてより詳しく説明します。これらの章では、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、ラージ・オブジェクトを使用する Pro*C/C++ プログラムなど、この製品で利用できる、より複雑な機能について説明します。

第 1 章「序説」

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、重要なよく聞かれる質問 (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間い合わせとともに、使用する表のサンプルが紹介されます。

第3章「データベースの概念」

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、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 プログラムの基本事項を説明します。ホスト変数および標識変数、カーソル、カーソル変数の使用方法と、Oracle データの挿入および更新、選択、削除を行う基本的な SQL コマンドの使用方を説明します。

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、動的 SQL を使った 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を説明します。ここで紹介するのは、配列を使って Oracle データを操作する方法、単一の SQL 文を使っ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です。

第9章「実行時エラーの処理」

この章では、エラーの報告および回復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、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 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、単純なものから複雑なものまで、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 は、新しい方法4アプリケーションのすべてで使用する必要があります。この方法では、いくつかの変数を含む SQL 文を受け取ったり、作成したりできます。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 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め込み 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。アソシエイティブおよびナビゲーション・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、その使用例を用いて説明します。

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」

Pro*C/C++ の 8.0 および 8.1 リリースに導入された機能の改善点と新機能について説明します。

付録 B 「予約語、キーワードおよび名前領域」

Oracle にとって特別の意味を持つ予約語およびキーワード、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++ のシステム固有の面について説明します。

付録 F 「埋込み SQL 文およびディレクティブ」

この付録では、プリコンパイラ・ディレクティブの説明（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。

このマニュアルの表記規則

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す。
- < > 山カッコは構文記述の構文要素の名前を示す。山カッコのかわりにイタリック体が使われる場合もある。
- { } 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う。
- .. 2 つのドットは、ある範囲内の最低値と最高値を区切る。
- ... 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、以下の 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なければなりません。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。詳細情報については、xxxvi 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 を使って、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・プログラムを使って 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 フラガーが用意されています。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 拡張機能のリストについては『Oracle8i SQL リファレンス』を参照してください。

FIPS フラガーを使うと、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

プリコンパイラ・オプション FIPS 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行で指定します。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については A-5 ページの「以前のリリースからの移行」を参照してください。旧リリースの Pro*C および Pro*C/C++ からアプリケーションを移行するには、『Oracle8i 移行ガイド』を参照してください。

1

序説

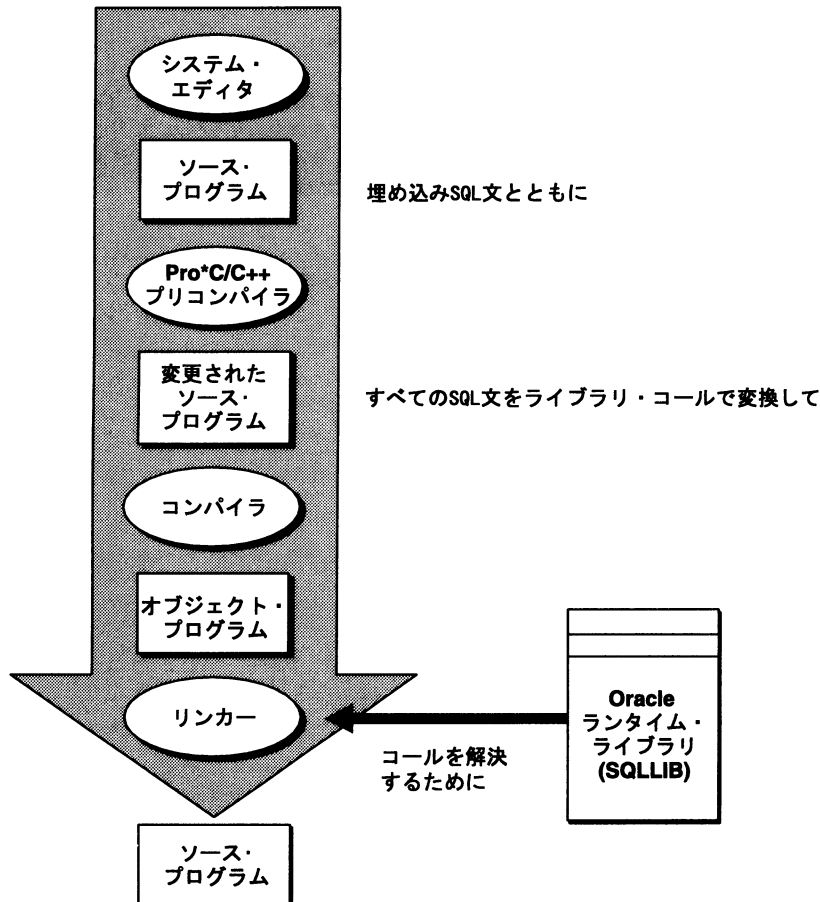
この章では、Pro*C/C++ プリコンパイラについて説明します。Oracle データを操作するアプリケーション・プログラムを開発する上での Pro*C/C++ プリコンパイラの役割と、Pro*C/C++ プリコンパイラによってアプリケーションが実行できる処理について説明します。この章で説明する内容は、次のとおりです。

- Oracle プリコンパイラとは何か
- なぜ Oracle Pro*C/C++ プリコンパイラを使うのか
- なぜ SQL を使うのか
- なぜ PL/SQL を使うのか
- Pro*C/C++ プリコンパイラの利点
- よく聞かれる質問 (FAQ)

Oracle プリコンパイラとは何か

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・ライブラリ呼出しに翻訳して、通常の方法でコンパイル、リンク、実行できるように変更されたソース・ファイルを生成します。

図 1-1 埋込み SQL プログラム開発



なぜ Oracle Pro*C/C++ プリコンパイラを使うのか

Oracle Pro*C/C++ プリコンパイラを使うと、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。便利で使い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 にアクセスできるようになります。

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、"1234" の CHAR 値を C の short 値に変換できます。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および SQL 文の実行状況、各種の実行時の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。

なぜ SQL を使うのか

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 を対話形式で使うか、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 を使ってください。

SQL は、その柔軟かつ強力な特性と習得のしやすさによって、最もすぐれたデータベース言語となりました。SQL は非手続き型言語であるため、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、

- 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

などです。

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 を使って 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。

なぜ PL/SQL を使うのか

SQL を拡張した PL/SQL は、手続き型構造および変数宣言、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。

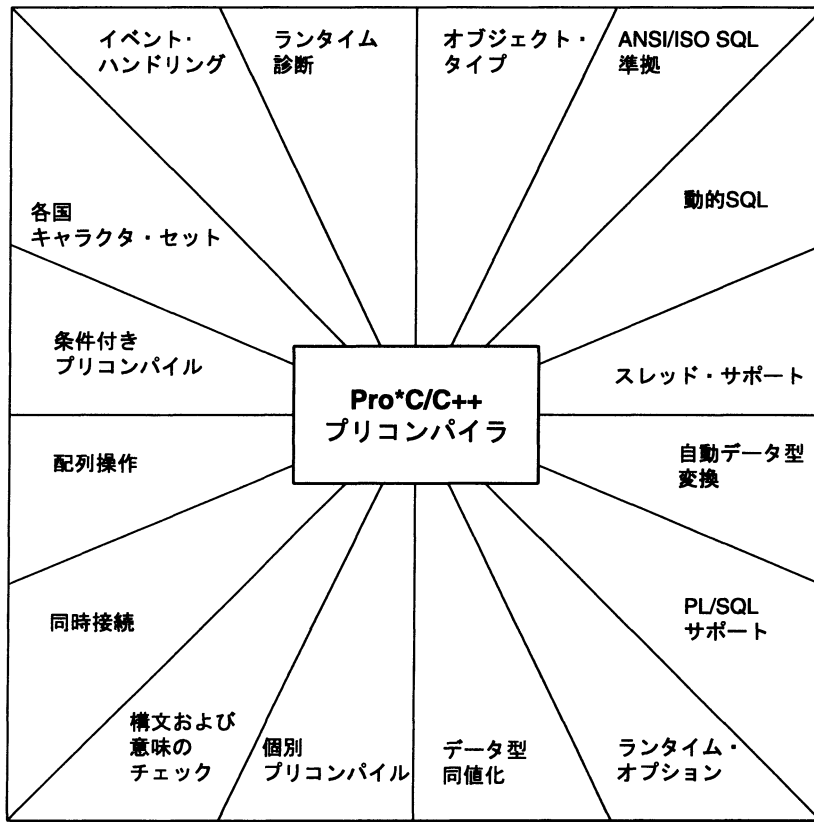
埋込み PL/SQL の主な利点はパフォーマンスの向上です。SQL と異なり、PL/SQL では、SQL 文を論理的にグループ化し、1 文ずつではなくブロック単位で Oracle に送ることができます。この結果、ネットワークの通信量および処理のオーバーヘッドが減少します。

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 章の「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 を使うことにより、プログラム実行時に適切な SQL 文を組み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行で指定し、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って、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 SQL データ操作文および PL/SQL ブロックの、構文および意味を完全にチェックします。
- Net8 を使って、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 およびネストした表) 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文字キャラクタ・セットを使用します。
- プログラム内で OCI (Oracle コール・インタフェース) 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++ について、および Pro*C/C++ との関連で 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、このガイドの他の部分に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。

質問：

VARCHAR がわかりません。VARCHAR とはなんですか。

回答：

以下は 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。4-4 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うときだけです。データ型同値化については、4-7 ページの「VARCHAR」、第 14 章の「ANSI 動的 SQL」、および第 15 章の「Oracle 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。

質問：

Pro*C/C++ は、Oracle コール・インタフェースのコールを生成するのですか。

回答：

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。

質問：

それでは、Pro*C/C++ を使用しないで、単に SQLLIB のコールを使ってコーディングしてもいいですか。

回答：

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 コール・インタフェースを使ってください。Oracle では、Oracle コール・インタフェースのサポートに力を入れています。

OCI と Pro*C/C++ を組み合わせて使うこともできます。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。

質問：

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。

回答：

もちろんできます。第7章の「埋込み PL/SQL」を参照してください。7-22 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。

質問：

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

回答：

はい。第12章の「C++ アプリケーション」を参照してください。

質問：

SQL 文の任意の場所でバインド変数を使えますか。たとえば、実行時に、SQL 文の表名を入力できるようにしたいのですが。ところが、ホスト変数を使うと、プリコンパイラのエラーが発生します。

回答：

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 を使う必要があります。第13章の「Oracle 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。

質問：

Pro*C/C++ の文字処理がよくわかりません。オプションが多すぎるように思えます。説明してください。

回答：

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] ホスト変数を使う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 も不適當であり、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* 文を使い、データ型の同値化を使っ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、4-40 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。

質問:

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。

回答:

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] を使うか、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い、malloc() を使っ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だけでなく、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。

質問:

Pro*C/C++ では、なぜ SPOOL が動作しないのでしょうか。

回答:

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。

質問:

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。

回答:

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。

質問：

アプリケーションをコンパイルしてリンクするには、どうしたらいいですか。

回答：

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、コマンド行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・オプションを使う必要がある場合は、Pro*C/C++ を別に実行してから *make* を実行できます。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、次のように入力できます。

```
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。

質問：

Pro*C/C++ では構造体をホスト変数として使えるようになったと聞きました。配列インタフェースではどのようなのか教えてください。

回答：

1 つの構造体の内部で複数の配列が使えます。また、構造体の配列が配列インタフェースとともに使えます。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。

質問：

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

回答：

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

質問：

すべてのリリースの Pro*C/C++ を、すべてのバージョンの Oracle Server で使えますか。

回答：

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えますが、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えません。

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。

質問：

アプリケーションを Oracle8i のもとで実行すると、いつも ORA-1405 エラー（フェッチされた列値が NULL である）になります。どうなっているのでしょうか。

回答：

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、標識変数を使ってプログラムを書き換え、今後の開発には標識変数を使いませう。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。

代りに MODE=ORACLE と DBMS=V7 または V8 でプリコンパイルしているのなら、ORA-01405 メッセージを無効にするためにコマンド行で UNSAFE_NULL=YES を指定（詳細は「UNSAFE_NULL」を参照）してください。

質問：

すべての SQLLIB 関数がプライベート関数ですか。

回答：

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 SQLLIB 関数がいくつかあります。パブリック SQLLIB 関数を次に示します。

SQLSQLDAAAlloc()

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います。15-3 ページの「SQLDA の参照方法」を参照してください。

SQLCDAFromResultSetCursor()

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います。5-48 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。

SQLSQLDAFree()

SQLSQLDAAAlloc() を使って割り当てた SQLDA を解放するために使います。5-48 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>SQLCDAToResultSetCursor()</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います。5-48 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います。9-20 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います。9-31 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オン・データ領域 を取得するのに使います。5-48 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オン・データ領域を 取得するのに使います。5-48 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 状態の表示を返します。 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>	数値の精度および位取りを戻します。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用い 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。

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。関数の名前は、リリース 8.0 で変更されましたが、従来の名前も Pro*C/C++ では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）5-49 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。

質問：

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。

回答：

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については第 17 章の「オブジェクト」および第 19 章の「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って要点を具体的に説明します。

この章のトピックは次のとおりです。

- 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 領域およびカーソル、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、問合せを行うには、INSERT および UPDATE、DELETE、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 文を使うと、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・プログラムの間でデータを転送できるその他の文があります。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います。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

実行可能 SQL 文

実行文	用途
ALLOCATE*	Oracle データの定義、制御
ALTER	
ANALYZE	
AUDIT	
CLOSE	
COMMENT	
CONNECT*	
CONTEXT	
CREATE	
DROP	
ENABLE THREADS	
FREE	
GRANT	
NOAUDIT	
RENAME	
REVOKE	
TRUNCATE	

表 2-1 埋込み SQL 文

DELETE	DML
EXPLAIN PLAN	
FETCH*	
INSERT	
LOCK TABLE	
OPEN*	
SELECT	
UPDATE	
COMMIT	トランザクションの処理
ROLLBACK	
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ことだけです。Pro*C/C++ はすべての EXEC SQL 文をランタイム・ライブラリ SQLLIB 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われていることだけです。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がわかっています。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わかっているわけです。

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 を使って PL/SQL ブロックを囲むだけで済みます。

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ので、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 章の「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 文内では、ホスト変数と Oracle オブジェクトを区別するために、ホスト変数の前にコロン (:) が必要です。

C 構造体を使って、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。

任意のホスト変数に任意指定の標識変数を対応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」とは欠けている値、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、標識変数を使いたいときは、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だけで済みます。また、INDICATOR キーワードを使って、ホスト構造体とそれに対応付けられた標識構造体と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しなければなりません。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どのようにデータを格納するかを指定します。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どのように格納されるかを指定します。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 文を使うと、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うこともできます。

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 領域およびカーソル、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うと、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 行だけを戻す SELECT 文（問合せ）を含めて、すべてのデータ定義文およびデータ操作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、ホスト配列を使う必要があります。

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（「現在行」と呼びます）を識別するには、明示的なカーソルを使います。

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、明示的なカーソルはアクティブ・セット内の現在行を「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。

データ定義文または COMMIT 文、ROLLBACK 文が最後に実行された時点以降に実行するデータ操作文すべてが、現行の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 文および ROLLBACK 文、SAVEPOINT 文を使ってトランザクションを定義できます。

COMMIT は現行のトランザクションに対する変更をすべて確定します。ROLLBACK は現行の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現在の点に印を付けま

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。

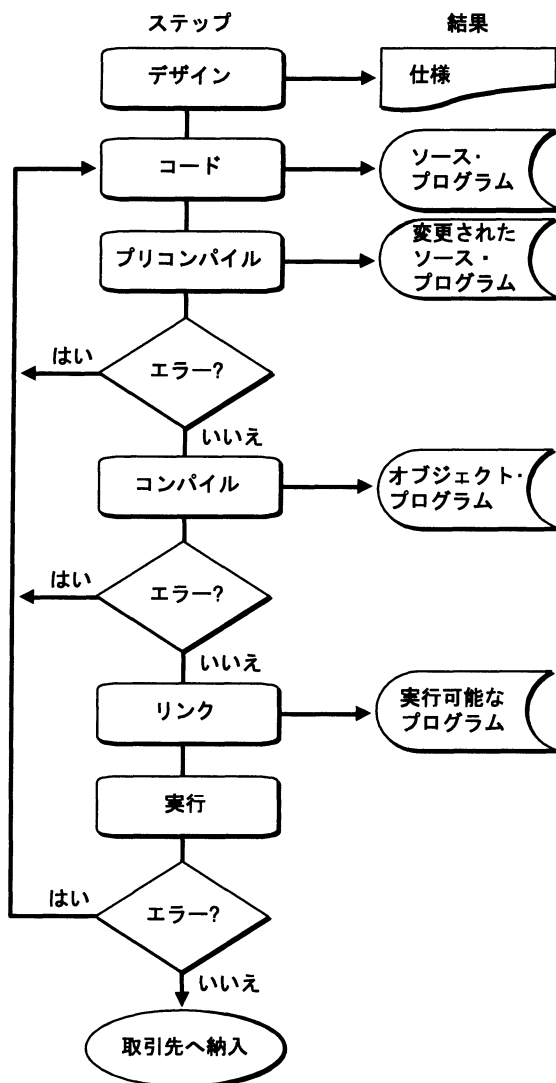
SQLCA は、ホスト・プログラム内に組み入れる（つまりハードコードする）データ構造体です。SQLCA は Oracle によって使用されるプログラム変数を定義して、実行時のステータス情報をプログラムに引き渡します。SQLCA を使うと、直前に実行を試みた作業に関する Oracle からのフィードバックに基づいて、別の処理を行うことができます。たとえば、PL/SQL を使用しないと、Oracle は一度に SQL 文を 1 つずつ処理しなければなりません。

WHENEVER 文を使うと、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、および関数のコール、ラベル付き文への分岐、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 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みのソース・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るようになります。

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 に固有の機能と制限を扱います。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。また、次の例に示すように、SQL 文内の行末に ANSI 形式のコメント（--...）を入れることもできます。

```
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・オプションを使っ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 を接尾辞としてつけると、**long** 型定数の指定になります。U もしくは u を接尾辞としてつけると符号なし **unsigned** 型定数の指定になります。0X もしくは 0x を接頭辞としてつけると、16 進整数定数の指定になります。F もしくは f を接頭辞につけると、浮動小数点定数の指定になります。これらの形式は SQL 文では許されません。

宣言節

ホスト変数宣言およびそのフォームを含む「宣言節」

```
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節

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

これらの2つの文の間で利用できるものは、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 または CODE=CPP (C++ アプリケーション内) または PARSE=NONE または PARTIAL の場合には、宣言節が必要です。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。

複数の宣言節を、異なるコード・モジュールで使用できます。

デリミタ

C は引用符を単一文字の区切りとして使います。たとえば

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

SQL では引用符は以下のように文字列の区切りに使います。

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

C では、二重引用符は次のように文字列を区切るときに使います。

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

SQL では二重引用符は特殊文字もしくは小文字を含んだ識別子を区切るのに用います。

```
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。内部で使用する変数によって、生成されるファイルのサイズが制限される場合があります。使用可能な行数には制限があります。ソース・ファイルのファイル・サイズは次の条件によって制限されます。

- 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 句を使っ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ってソース・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」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、コマンド行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・アウトされている場合を除き、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 コンパイラを使う場合は、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 章の「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 の「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。文字列リテラルの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、円記号 (¥) 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、プリコンパイラは円記号を継続文字として扱います。

行の長さ

ASCII 文字だけ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。必要ならより小さな値を指定してください。たとえばお使いの 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のならば、MAXLITERAL=512 を指定します。お使いの C コンパイラのユーザーズ・ガイドを調べてください。

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以下のとおりです。

型	C 演算子
アドレス	&
ビット単位	&, , ^, ~
複合代入	+=, -=, *= 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たいことがあります。条件付きプリコンパイルを使えば、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節が区切られます。これらの条件節には、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、次の 2 通りの方法があります。次の文を含めます。

```
EXEC ORACLE DEFINE symbol;
```

次の構文を使ってコマンド行で記号を定義します。

```
... 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 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み記号には、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、いくつかのよく発生する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 を使っても、MAXOPENCURSORS の値は無視されます。実行時は CONNECT に対する有効値だけが使用されます。

単一の SQLCA の使用

SQLCA を 1 つだけ使うときは、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 記憶域クラスを使い、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ほかのプログラム・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: ひとつのアプリケーションのすべてのソース・ファイルは、名前が一意でなければなりません。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てから、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しなければなりません。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ル参照を解決します。これらの参照で競合が発生すると、リンクは失敗します。他社製のソフトウェアをプリコンパイル済みのプログラムにリンクしようとする、このようにリンクが失敗することがあります。これはすべての他社製のソフトウェアが Oracle と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があります。

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。これらを使って Pro*C/C++ アプリケーションのプリコンパイル、コンパイル、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。次に示すのはそれらの定義です。

```
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 に精通する方法の 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どれかが「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` を使っ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, comm
                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、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され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。

この章のトピックは次のとおりです。

- データベースへの接続
- 高度な接続オプション
- 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守る方法
- トランザクションの開始・終了方法
- COMMIT 文の使い方
- SAVEPOINT 文の使い方
- ROLLBACK 文の使い方
- RELEASE オプションの使用方法
- SET TRANSACTION 文の使い方
- 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 文を使ってください。

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

ここで *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-25 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 文でなければなりません。つまり、プリコンパイル・ユニット内で他の SQL 文を物理的に CONNECT 文の前に置くことはできませんが、論理的に前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 つだけです。)

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しておかなければ 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、次の例にあるように自分で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るようになりました。

この項では、ALTER AUTHORIZATION 句をさまざまに変化させて使っ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何の問題もなく接続されます。
- 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、パスワードは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するようにしてください。そうしないと、アカウントがロックされてしまいます。
- 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 文を使ったとします。

```
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しようとしていることを示します。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何の問題もなく接続されます。
- 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 ドライバを使って接続するには、*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います。

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 に関する詳細は、『Oracle8i Net8 管理者ガイド』を参照してください。

自動接続

次のユーザー名で自動的に Oracle8 に接続することができます。

OPS\$username

この場合、*username* は現行のオペレーティング・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引き渡すだけです。

```
...
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、'/' を用いた接続で 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が入っていない限りはなりません。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ならば、アプリケーションは次のユーザー ID を使って接続を試みます。

OPS\$<username>

この場合、*username* は現行のオペレーティング・システムのユーザー名またはタスク名で、*OPSS\$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 文を使う必要があります。

SYSDBA または SYSOPER システム権限

Oracle8i がリリースされる前は、SYSDBA または SYSOPER システム権限を得るためには、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 以降では使用しなければなりません。

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。）

高度な接続オプション

予備知識

ネットワーク内の通信ポイントを「ノード」といいます。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 文およびデータ、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回復手順、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うだけで済みます。

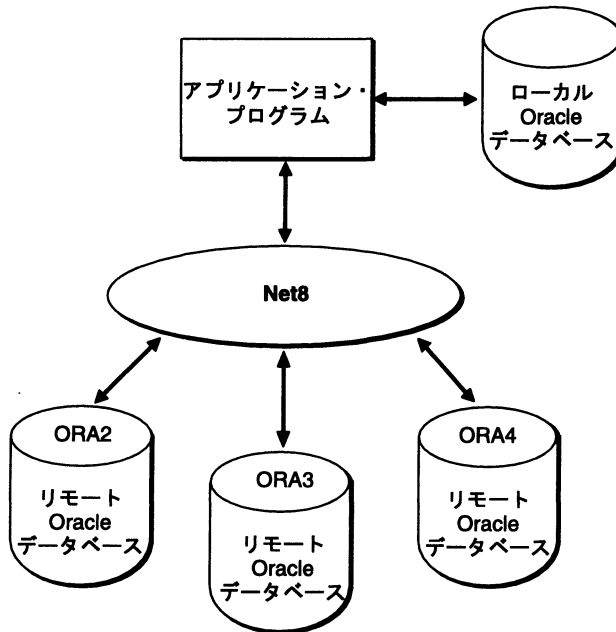
そのサービス名がデフォルト（ローカル）ドメイン内にない場合は、グローバル指定（すべてのドメインの指定）を使用する必要があります。たとえば、次のとおりです。

HR.US.ORACLE.COM

同時ログイン

Pro*C/C++ は、Net8 を介して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 および ORA3、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もたらします。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については『Oracle8i Net8 管理者ガイド』と各システムの Oracle8 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、データベース名だけ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 句をもつ 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。

データベース名は一意でなければなりません。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もつことができます。

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;
```

ここで、*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* は現行のオペレーティング・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、現行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したい場合、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およびマシン、データベースを指定します。その後、AT 句 (*DB_NAME* 付き) を使っている 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だけ、CONNECT ...AT *DB_NAME* 文を実行する前に DECLARE *DB_NAME* DATABASE 文を実行しなければなりません。

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 データ操作文を実行できます。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-38 ページの「DECLARE TABLE (Oracle 埋込み SQL ディレクティブ)」を参照してください。

PL/SQL ブロック

PL/SQL ブロックは、AT 句を使って実行できます。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 句を決して使いません。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たい場合は、次に示すとおり、DECLARE CURSOR 文で AT 句を使ってください。

```
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、宣言は DECLARE されたカーソルを参照するすべての SQL 文の有効範囲内になければなりません。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 句が使わ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* 句を使うと、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 文を実行しようとしているときに *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なく、*emp_cursor* は 1 つしかありません。*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だけで済みます。

```
...
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したい場合、AT 句を使う必要があります。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したい場合、DECLARE STATEMENT 文でだけ AT 句を使います。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

複数の明示的接続には、単一の明示的接続の場合と同様に、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を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。これにより、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、この方法を使っ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 文字列を使う場合も、個々に異なるデータベース名を使う必要があります。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しなければなりません。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しなければなりません。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。分散問合せを使うと、単一の SELECT 文によって1つ以上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、これは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、単一の非デフォルト・データベースに接続します。まず、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・リンクを使って非デフォルトの 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 句で使わ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 に伝えるだけです。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 Server 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。(ここでも通常、対話的に作成します。)

```
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 表の問合せができます。

```
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 表とを結合して問合せを実行します。

トランザクション用語の定義

トランザクションの本题に入る前に、この項に定義されている用語に慣れておく必要があります。

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 Oracle 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 は並行性、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しなければなりません。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）を使って、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しようとすると、「デッドロック」が発生することがあります。たとえば動的 SQL 方法 2 で入力ホスト配列を使うには、次の構文を使います。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されていて、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「読み取り一貫性」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み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守る方法

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っ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 文を使ったトランザクションの確定
- ROLLBACK TO 文とともに SAVEPOINT 文を使ってのトランザクションの部分的な取消し
- ROLLBACK 文を使っ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オフ
- SET TRANSACTION 文を使っ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 文を使ってのデフォルトのロックの変更（上書き）

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 文を記述する。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 または CREATE、GRANT など）を記述する。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）の状態に復元されます。

COMMIT 文の使い方

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が 1 つのトランザクションとみなされます。（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 文を使う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。

- 現在の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。
- これらの変更が他のユーザーにわかるようにします。
- 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。例については 3-23 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オフします。

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 文の使い方

SAVEPOINT 文を使うと、トランザクションの処理の現時点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るようになります。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復元し、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、表 MAIL_LIST 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を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だけが消去されます。

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。

ROLLBACK 文の使い方

ROLLBACK 文を使うと、保留状態になっているデータベースへの変更を取り消せます。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 文を使えば元のデータを復元できます。TO SAVEPOINT 句を使うと、現在の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、未完了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。

- 現在の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。3-20 ページの「RELEASE オプションの使用方法」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現在の文をロールバックします。

エラーとなった SQL 文が開始した作業だけが失われます。つまり、現在の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 オプションの使用方法

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 RELEASE;
```

または

```
EXEC SQL ROLLBACK RELEASE;
```

上記以外の場合は、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、ロックされたり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 文の使い方

SET TRANSACTION 文を使うと、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 文でなければなりません。また、1つのトランザクション内で一度しか使うことができません。READ ONLY パラメータは必須です。READ ONLY パラメータを使っ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えるのは SELECT 文および COMMIT 文、ROLLBACK 文だけです。したがって、INSERT 文、DELETE 文または SELECT FOR UPDATE OF 文などを使う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って1日および1週間前、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したいときは、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 文を使うと、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 文を使うと、表全体を明示的にロックできます。

FOR UPDATE OF の使い方

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 句を使う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 句を使って、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 句を使うと、複数の表を参照できなくなります。

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

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 の使い方

LOCK TABLE 文を使うと、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なくなります。

```
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ユーザーだけです。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行を INSERT または UPDATE、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、問合せで表ロックが取得されることはありません。したがって、問合せが他の問合せまたは更新の妨げになることもありません。また、更新が問合せの妨げになることもありません。2つの異なるトランザクションが同じ行を更新しようとしたときだけ、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。

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 句は使わないでください。そのかわりに、まず各行の ROWID を SELECT します。次にその値を使って、更新または削除中の現在行を確定します。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してしまうと、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 つのノードにだけ影響するときは、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だけをコミットまたはロールバックするため、現行の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 句を使うと、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なければなりません。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、いくつかのよく発生する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。余分なものも不足ありません。

表内で参照するデータは一貫したものでなければなりません。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、ふたつの銀行口座間の資金移動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) だけをロックできます。

PL/SQL の使用

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。

- オラクルのデータ型
- ホスト変数
- 標識変数
- VARCHAR 変数
- カーソル変数
- CONTEXT 変数
- ユニバーサル 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、コンパイルおよび実行、必要に応じた修正もできます。

オラクルのデータ型

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）を表すのに使う形式を指定します。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う形式を指定します。Oracle のデータ型の詳細は『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。

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います。

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	2 進化 10 進数形式で表した数値
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、<= 255 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、<= 4 G バイト
BLOB	バイナリ・データ、<= 4 G バイト
CLOB	文字データ、<= 4 G バイト
NCLOB	マルチバイト・キャラクタ・データ、<= 4 G 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、現行の 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、グラフィック・イメージ、ビデオ・クリップおよびサウンド波形）を格納するには、LOB データ型を使います。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 章の「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われます。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 に示したように、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	2 進浮動小数点形式で表した、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ります。
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、 $\leq 2^{31}-1$ バイト
UNSIGNED	符号なし整数
LONG VARCHAR	可変長文字列、 $\leq 2^{31}-5$ バイト
LONG VARRAW	可変長バイナリ・データ、 $\leq 2^{31}-5$ 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 データ型を使って、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 オプションを使っ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 終了記号は付けしないでください。

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。標識変数と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、科学表記法が使われます。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' を返します。

NUMBER

NUMBER データ型を使って、Oracle の固定小数点数または浮動小数点数を格納します。精度および位取りを指定できます。NUMBER 値の最大精度は 38 桁です。大きさの範囲は 1.0E-129 から 9.99E125 になります。位取りの範囲は -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 を底とする 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 21 バイトの長さにしなければなりません。数を表現するのに使われるバイトだけが戻されます。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 データ型を使います。

この外部データ型を使う必要はほとんどありません。

INTEGER

INTEGER データ型を使って、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 のデータ型を使って、小数部分をもつ数または 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って表示されます。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 オプションを使っ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って、NULL 終了記号の走査を制限します。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。空白だけからなる値は、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 は最初の文字位置に 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しなければなりません。出力では、ホスト変数には長さについて 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す。

通常、このデータ型はほとんど使用されません。

LONG

LONG データ型を使って、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 データ型を使って、可変長文字列を格納します。VARCHAR 変数では、長さ 2 バイトのフィールドに ≤ 65533 バイト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必ず 2 バイトを付加してください。もっと長い文字列には、LONG VARCHAR データ型を使ってください。

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っ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、ユニバーサル 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
```

ここで、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

```
0000000E.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。

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。使用例は 8-25 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、もしくはアプリケーションが Oracle Open Gateway テクノロジを使ってオラクル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 データ型を使って、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 データ型を使って、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 255 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 データ型を使って、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あとに長さが 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ものでなければなりません。

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

LONG RAW

LONG RAW データ型を使って、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 データ型を使って、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 データ型を使って、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 (2³¹-5) バイトです。VAR 文または TYPE 文で使う 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 データ型を使って、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 文もしくは TYPE 文で使う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 データ型を使って、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。入力値が空白だけからなる場合、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だけを果たし、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 文で使います。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 を使うと、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* でなければなりません。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なければなりません。

MODE=ORACLE の場合、特別な宣言節でホスト変数を宣言する必要はありません。ただし宣言節が ANSI SQL 標準の一部である場合に宣言節を使用しないと、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、宣言節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える疑似型を示します。これらのデータ型だけ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)	char	単一の文字

(注意 1)

表 4-5 C と Oracle のデータ型の互換性

内部型	C 型	説明
CHAR(X)	char[n]	n バイトの文字配列
(注意 1)	VARCHAR[n]	n バイトの可変長文字配列
	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
NUMBER	int	整数
NUMBER(P,S)	short	小さい整数
(注意 2)	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
	char	単一の文字
	char[n]	n バイトの文字配列
DATE	VARCHAR[n]	n バイトの可変長文字配列
	char[n]	n バイトの文字配列
LONG	VARCHAR[n]	n バイトの可変長文字配列
	char[n]	n バイトの文字配列
RAW(X)	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
LONG RAW	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
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

表 4-5 C と Oracle のデータ型の互換性

内部型	C 型	説明
注意：		
1. X の範囲は 1 から 255 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 2 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「オラクルのデータ型」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] の場合にだけ指定できます。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うことができます。ただし、プリコンパイラはホスト変数の前にアンバサンド（&）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っ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うことができるのは、ブロック内だけです。

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。上記の例で、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 文字以内でなければなりません。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たなければなりません。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 データ操作文で使います。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てはなりません。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;
```

混乱するかもしれませんが、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 / 小文字の使用が一致してはなりません。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 の「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されなければなりません。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。以下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しなければなりません。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しなければなりません。宣言節を使用する場合は、宣言節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。詳細は第 17 章の「オブジェクト」の説明を参照してください。

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います。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 文を使って、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ったと、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* を使って、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「標識変数の使用方法」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（宣言節があればそこに）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、次のようにしてはなりません。

- ホスト言語文の中で、コロンを前に付けてはなりません。
- ホスト言語文の中で、そのホスト変数の直後に指定してはなりません。

- Oracle の予約語であってはなりません。

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: fetched column value is NULL
```

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 ホスト変数を使っ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、大文字と小文字が混在してはなりません。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み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も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'）終了記号を加えるといったことに使えます。

```
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 の範囲になければなりません。たとえば、以下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** による定義済みマクロか、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」を参照してください。

次のような typedef 文は使わないでください。

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

上記のような文を使うと、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って、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って 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、DBMS=V6 または UNSAFE_NULL=YES プリコンパイラ・オプションを使います。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。埋め込み処理された配列が 14 バイト、長さが 2 バイトです。

`SQLVarcharGetLength()` (以前の `sqlvcp()`) 関数 (SQLLIB 実行時ライブラリの一部) 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) が返されます。

`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り当ての `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。

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、`SQLStmTGetText()` (以前の `sqlgls()`) 関数も使用しています。この関数の詳細は第 9 章の「実行時エラーの処理」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()` 関数を使って 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("%.*s\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うことができます。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 を使って定義しオープンしなければならない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、ストアド・プロシージャ 1 か所を変更するだけで済みます。各アプリケーションを変更する必要はありません。
- **セキュリティに好都合なこと**：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わ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許可が与えられている必要がありますが、問合せに使用される表に対する読み取り許可は必要ありません。このセキュリティ機能を使って、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 を使って、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だけ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ています。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です。

カーソル変数の割当て

カーソル変数をオープンするか、カーソル変数を使って 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 を使います。たとえば上の例で宣言されている SQL_CURSORemp_cursor を割り当てるには以下の文を書きます。

```
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だけ解放されます。

```
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 上でオープンしなければなりません。埋込み SQL OPEN コマンドを使っ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 ブロックを使っ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 PL/SQL 無名ブロックの 1 つを使っ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;
...
```

上記の各例では、PL/SQL を使っ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);

```

上記の文では、`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だけでなく、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

上記の例では、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しなくてもかまいません。

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 コマンドを使います。たとえば、上記の例で宣言した `SQL_CURSORemp_cursor` を割り当てるには、次の文を記述します。

```
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だけオープンおよび FETCH、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 埋め込み 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。これらの関数を使って、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、割当て済みのカーソル変数を 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);
```

ここで、各パラメータは以下のとおりです。

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 の行処理済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 5-48 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。

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

ここで、各パラメータは以下のとおりです。

<code>context</code>	SQLLIB 実行時コンテキストへのポインタ。
<code>cur</code>	変換先の <code>Pro*C/C++</code> カーソル変数へのポインタ。
<code>cda</code>	変換元の OCI カーソル・データ領域へのポインタ。
<code>retval</code>	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だけです。

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る必要があります。

SQLLIB パブリック関数の詳細は 5-48 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、接続後直ちに `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` コマンドだけで使用できます。

- 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、すでにクローズ済み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、カーソル変数の型だけです。
- 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

以下のサンプル・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実演しています。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

「実行時コンテキスト」（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）、およびほか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・ディレクティブを使用して、コンテキスト用のメモリーの割り当ておよび初期化を行います。

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

ここの 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・ディレクティブを使って、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 ディレクティブ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 が定義および割り当てられていない場合、エラーが返されます。

CONTEXT FREE 文を使うと、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');
...
}

```

ユニバーサル 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 を基数にする 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 リリースからユニバーサル ROWID が導入されています。ユニバーサル 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、ユニバーサル 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。

ROWID の詳細は『Oracle8i 概要』を参照してください。

ユニバーサル 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。
- ユニバーサル ROWID 変数用のメモリーを割り当てます。
- ユニバーサル 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、

```
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 ;
...
```

18 から 4000 までの幅を持つ文字列ホスト変数をユニバーサル 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。文字ベースのユニバーサル 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。ユニバーサル 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=4000 ;
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 ;
...
```

関数 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ユニバーサル 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)

は、実行時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)

は、ユニバーサル 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 が返されます。

注意: ユニバーサル 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かねばなりません。ユニバーサル ROWID では後で FREE を使用してください。

ホスト構造体

C 構造体を使うと、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 で説明するように、有効な Pro*C/C++ のホスト変数でなければなりません。

構造体がホスト変数として使用されると、構造体の名前だけが 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たりします。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していなければなりません。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に一致する必要があります。

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;
```

上記の例は、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章の「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って、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 RECORD 変数のホスト 変数として使用できません。

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

標識変数を使う必要があるのに、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しなければなりません。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: このガイドでは慣習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あります。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い、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("-----      -      -\n");

/* 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。これはほかの変数を「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。

ポインタを使うと、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。構造体メンバーのデータ型とポインタ変数は、同じでなければなりません。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。以下の例は

- 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問合せで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。実際には使う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、シングルの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。

さまざまな 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>
```

ここで、*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 シェルを使う 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、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・インタフェースが作成できます。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` を使って、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` を使って、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 と呼びます。) これらのデータ型は、リレーショナル列でだけ指定できます。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。

コマンド行オプション 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 はブランクの埋め込みと除去、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、空白の埋め込みと除去、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、次のようになります。

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

この例では、<Japanese_string> は 2 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、コマンド行に 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・セットが指定されていなければなりません。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 文を使うと、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上書きできます。これを「ホスト変数の同値化」といいます。

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-112 ページの「VAR (Oracle 埋込み SQL ディレクティブ)」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>;
```

上記の文には、マルチバイト文字列が入っています。（<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" のどちらを使ってもかまいません。

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 マルチバイトである（と、NLS_CHAR オプションで指定された）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。この章のトピックは次のとおりです。

- 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
- プリコンパイル済みのヘッダー・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进行处理する方法を説明します。
ホスト変数文字型には次の 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）と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・コマンド行オプションを使って、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。Oracle8i 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 NULL 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 固定文字型に準拠。
STRING	新しい形式	NULL 終了記号 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。非 NULL 埋込み。

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う 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 章の「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 を使います。これは CHAR_MAP=VARCHAR2、STRING、または CHARF を指定することで上書きできません。

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ようにしてください。DBMS=V7 または V8 のときには、文字配列は NULL 終了記号（'\0'）でなければなりません。

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なければなりません。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* のすべての値の取り出しに使います。

```
char str[5] = {'X', 'X', 'X', 'X', 'X'} ;
short str_ind;
...
EXEC SQL SELECT strval INTO :str:str_ind WHERE ... ;
```

CHAR_MAP が VARCHAR2 または CHARF、CHARZ、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

ここで 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 オプションを使っても、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。

プリコンパイラの実行時環境は *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') が入っていないことを確認しておいてください。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 または V8 と MODE=ANSI でプリコンパイルした Pro*C/C++ プログラムで使うことができます。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い、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;
```

上記の 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うと、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`）を設定することだけです。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しなければなりません。

```
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 ポインタを使います。

```
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うと、プログラム・インタフェースは長さ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、長さメンバーを使って、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しなければならない場合もあります。

VARCHAR ポインタ 出力ホスト変数として VARCHAR へのポインタを使うと、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 とも呼ばれます。）を使うことができます。UCS-2 では、1 文字に対して 2 バイトが使われます。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 dbase 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。

- `includeoratypes.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_varchar ulong_varchar ;
```

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 文に静的および動的 SQL に Unicode を含めることはできません。次のように記述することはできません。

```
#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。次のコードを使うことはできません。

```
typedef utext utext_5 ;
EXEC SQL TYPE utext_5 IS STRING ;
```

CONVBUSZ は、変換バッファ・サイズとして使うことはできません。CHAR_MAP オプションを使ってください。詳細は、4-48 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 13 章の「Oracle 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 章の「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 はそのコードを使って、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 `C short` 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うと、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上書きできます。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 にデフォルトの割り当てを示します。

表 5-2 デフォルトの型割り当て

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 文を使うと、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。使う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* はすでに宣言済み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だけで済みます。

```
...
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 を使います）のどれを使うこともできます。

ユーザ定義型同値化

また、ユーザ定義のデータ型を 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 文を使って新しいデータ型を外部データ型に同値化します。

TYPE 文を使うと、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、次の構文を使います。

```
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 struct に基づく新規のデータ型を、**typedef** を使って定義します。最後に、次の例のように、新しいユーザ定義のデータ型を 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）も戻し、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する可能性があるため、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、必ずその長さと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。）使用例については4-20 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 文で使用する、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 データ型を使えば、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 を使うと、DBMS オプションまたは CHAR_MAP オプションの設定に関係なく、固定長の文字型になるように同値化が行われます。

EXEC SQL VAR と TYPE ディレクティブの利用

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可能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、宣言節を使わなければなりません。この場合、TYPE 文または VAR 文は宣言節の中になければなりません。詳細は、F-106 ページの「TYPE (Oracle 埋込み SQL ディレクティブ)」および F-112 ページの「VAR (Oracle 埋込み SQL ディレクティブ)」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 コマンドが使用されます。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)elite 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)elite 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: hl

Retrieving executable stored under key 'hello' to file 'hl'...

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(varchar *, char *);
    void do_retrieve(varchar *, char *);
```

```
void do_delete(varchar *);
ub4 read_file(char *, OCIBlobLocator *);
void write_file(char *, OCIBlobLocator *);
#else
void do_connect(/*_ void _*/);
void create_table(/*_ void _*/);
void sql_error(/*_ char * _*/);
void list_executables(/*_ void _*/);
void print_menu(/*_ void _*/);
void do_insert(/*_ varchar *, char * _*/);
void do_retrieve(/*_ varchar *, char * _*/);
void do_delete(/*_ varchar * _*/);
ub4 read_file(/*_ char *, OCIBlobLocator * _*/);
void write_file(/*_ char *, OCIBlobLocator * _*/);
#endif

void main()
{
    char reply[20], filename[100];
    varchar key[20];
    short ok = 1;

    /* Connect to the database. */
    do_connect();

    printf("Do you want to create (or 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・ディレクティブ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** ディレクティブを使用した、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みエンティティの使用。
- **#include** ディレクティブを使用した、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およびマクロの定義、およびプリコンパイラでの **#include** ディレクティブ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、およびファイルの組込み、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い、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、以下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 */
...
```

現行の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 の定義済みの値（つまり 20）を使って、*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・ディレクティブ

Pro*C/C++ がサポートするプリプロセッサ・ディレクティブは次のとおりです。

- **#define**: プリコンパイラおよびCまたはC++のプリ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う他のソース・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だけ、ソース・テキストをプリコンパイルし、コンパイルします。
- **#ifdef**: 定義済みコンテキスト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。

無視されるディレクティブ

Pro*C/C++ プリプロセッサが使わないCプリプロセッサ・ディレクティブもあります。これらのディレクティブのほとんどは、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ディレクティブです。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プリプロセッサのディレクティブは、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ディレクティブをサポートしている場合でも、Pro*C/C++ はこれらのディレクティブを使いません。これらのディレクティブ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ディレクティブを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っ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。このマクロを使うと、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げます。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 マクロを使っ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 マクロを使って *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・ディレクティブに対してだけ使うことができます。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みの 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 文を使って 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が読むべきヘッダー・ファイル、たとえば *sqlca.h*、*oraca.h* や *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にない場合、コマンド行に、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 章の「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・オプションを使います。詳細は、第 10 章の「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ることができます。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うことができます。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];
    ...
}
```

引数をもつプリプロセッサ・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、次のようになります。

```
#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・ディレクティブを使って、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、次のようにします。

```
#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・ディレクティブの使用には制限があります。**#define** ディレクティブを使って実行可能な SQL 文の中で使う記号定数を生成することはできません。以下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える宣言 SQL 文は TYPE 文と VAR 文だけです。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、ディレクティブ **#** および **##** を無視して、プリコンパイラが認識しなければならない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れば）純粋な C コードの中では使えます。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 に使うことができない SQL 文

Pro*C/C++ プリコンパイラが **#include** ディレクティブを処理する方法については前の項で触れましたが、この方法のために、**#include** ディレクティブを使って埋込み SQL 文の入ったファイルを組み合わせることはできません。**#include** を使って、純粋に宣言文とディレクティブ、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 および ORACA、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 コマンドを使うと、*sqlca.h* および *oraca.h*、*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については、第 9 章の「実行時エラーの処理」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・ディレクティブを使用して 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** ディレクティブを使う場合は、必ずファイル拡張子 (*.h* など) を指定してください。

注意: **#include** ディレクティブを使って複数箇所に SQLCA を組み込む必要がある場合には、**#include** の前にディレクティブ **#undef SQLCA** を置かなければなりません。これは、*sqlca.h* が行の先頭になければならないためです。

```
#ifndef SQLCA
#define SQLCA 1
```

そして次に、SQLCA が定義されていないときにかぎり、SQLCA 構造体を宣言します。

#include ディレクティブまたは 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。コマンド行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= オプションを使うことができます。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、構成ファイルについての詳細は第 10 章の「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* や *sqlda.h* などの標準的プリプロセッサ・ヘッダー・ファイルのデフォルト位置はプリプロセッサ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って、`#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、次のコマンドを使って、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` に前もって設定しておきます。

EXEC SQL INCLUDE および #include の要約

プログラム中で `EXEC SQL INCLUDE` 文を使う場合、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` を使っ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` を使っ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だけです。

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` ディレクティブを使っ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。

定義済みマクロ

C コンパイラのコマンド行でマクロを定義しているなら、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行でも定義しなければならないかもしれません。たとえば UNIX のコマンド行で以下のようなコンパイルをします。

```
cc -DDEBUG ...
```

この場合は、`DEFINE=` オプションを使っ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、コマンド行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++ コマンド行および `cc` コマンド行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います。

```
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。

プリコンパイル済みのヘッダー・ファイル

プリコンパイル済みのヘッダー・ファイルを使うと、`#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、プリコンパイル済みのヘッダー・ファイルが作成されます。
- このプリコンパイル済みのヘッダーが、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。

- プリコンパイル済み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`。

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行だけです。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` でなければなりません。

プリコンパイル済みのヘッダー・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'` でなければなりません。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みのヘッダー・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 オプションを使っ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 とともに使っても無視されます。

プリコンパイル済みのヘッダー・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ス化されます。

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

プリコンパイル済みのヘッダー・ファイルは、#include ディレクティブを使ってインクルードされた回数にかかわらず、1 度しかインスタンス化できません。

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 ディレクティブ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、プリコンパイル済みのヘッダー・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みヘッダー・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みのヘッダー・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みヘッダー・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。

次に、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() {}
```

プリコンパイル済みヘッダー・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化されます。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 回インスタンス化されます。

Pro*C/C++ では、プリコンパイル済みヘッダー・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みヘッダー・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、プリコンパイル済みのヘッダー・ファイルを使う意味がなくなってしまいます。

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・オプションを使うことができます。

DEFINE および INCLUDE オプション

プリコンパイル済みヘッダーを使ってプリコンパイルするときは、DEFINE および INCLUDE の値に対して、プリコンパイル済みのヘッダー・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、プリコンパイル済みのヘッダー・ファイルを再作成しなければなりません。

開発環境を変更した場合も、プリコンパイル済みのヘッダー・ファイルを再作成しなければなりません。

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、プリコンパイル済みのヘッダー・ファイルのファイルは、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うことができなくなります。

DEFINE または INCLUDE オプションの値を変更したため、プリコンパイル済みのヘッダー・ファイルのファイルは、`#include` ディレクティブの対応する .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、プリコンパイル済みのヘッダー・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しなければなりません。

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。

複数ユーザーの場合

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、プリコンパイル済みのヘッダー・ファイル `common.hdrA` を作りました。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` を作りました。しかし、ユーザー A とユーザー B の環境が異なるため、2 人のユーザーが使った 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 が作った `common.hdr` の内容が同じになるとは限りません。

コードは次のようになります。

```
A> proc HEADER=hdr DEFINE=<A macros> INCLUDE=<A dirs> common.h
B> proc HEADER=hdr DEFINE=<B macros> INCLUDE=<B dirs> common.h
```

異なる環境で作られたため、生成されたプリコンパイル済みのヘッダー・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` を使っ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、プリコンパイル済みのヘッダー・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` を使わないでください。ソース・コードに `h` ヘッダー・ファイルだけ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。

プリコンパイル済みのヘッダー・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` 以外は使えません。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う `PARSE` の値は、プリコンパイル済みのヘッダー・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みのヘッダー・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、ホスト変数を宣言節の中で宣言しておく必要があります。C++ コードは、`PARSE=PARTIAL` を指定した場合にだけ読み込まれます。`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 10-33 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 マクロだけが処理されます。

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 ホスト変数を使えません。

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、明示的な宣言節の内部で宣言する必要があります。

使用上の注意

プリコンパイル済み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っていることがあります。Pro*C/C++ では、プリコンパイル済みのヘッダー・ファイルの出力ファイルの生成に使わ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、プリコンパイル済みのヘッダー・ファイルを使っ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強くお勧めします。

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うことはできません。

Oracle プリプロセッサ

コードの条件節は、環境および処理を定義する EXEC ORACLE ディレクティブによってマークされます。これらの条件節には、C の文、および埋込み SQL 文とディレクティブを記述できます。次の EXEC ORACLE ディレクティブで、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。以下の文をインクルードします。

```
EXEC ORACLE DEFINE symbol;
```

次の構文を使ってコマンド行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・ディレクティブは、EXEC ORACLE DEFINE コマンドと同じではありません。

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うことができます。

```
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++ コンパイラを使っ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う位置であれば、どこでも数値定数を宣言できます。

これが主に使われるのは、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、必ず次の規則を守ってください。

- 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なければなりません。

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うと、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++ 実行時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++ 実行時コンテキストは、複数の OCI 環境ハンドルと関連付けることはできません。実行時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、各実行時コンテキストは、その OCI 環境ハンドルに関連付けられます。

実行時コンテキストと OCI リリース 8 環境の確立と終了

EXEC SQL CONTEXT USE 文には、Pro*C/C++ プログラムで使う実行時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。もしソース・ファイルに EXEC SQL CONTEXT USE が出現しなければ、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、現行の実行時コンテキストと、それに関連付けられている現行の 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 文を使ってデータベースへのログインが実行されると、実行時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 文を使って Pro*C/C++ 実行時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++ 実行時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われます。

- メモリーの割当ておよびメモリーの解放、テキスト・ファイルへの書込み、出力バッファのフラッシュに対してその環境で使われるコールバック関数は、それぞれ `malloc()` および `free()`、`fprintf(stderr,...)`、`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領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、ユーザーはさまざまな 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 章の「オブジェクト」を参照してください。これらの SQLLIB 関数は以下で説明します。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 とほかの 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ことを学ばなければなりません。たとえば、ユーザーが OCI インタフェースを使っ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 実行時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えます。
- **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えます。
- シングルスレッド実行時コンテキストを使うときに、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 実行時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> を実行時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 実行時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・コンテキストを使って、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 *svc* を実行時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 *rctx* (IN): SQLLIB 実行時コンテキストへのポインタ。

dbname (IN) = この接続の「論理」名を含んだバッファ。

dbnamelen (IN): *dbname* バッファの長さ。

svc (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 句で使わ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. コールしたい OCI ファンクションが `ServiceContext` ハンドルを必要とするならば、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` を使って 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. 希望する実行時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って取得します。

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って 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 コールが必要とするなら、`OCIServiceContext` ハンドルを `SQLSvcCtxGet` コールを使って取得します。

シングルスレッド・アプリケーションの場合

```
retcode = 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、宣言節の外に宣言します。)
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、『Oracle 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 を使って 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 文で使う接続のための 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だけに使います。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・アプリケーションの場合は、コンテキストとして定義された定数 `SQL_`

SINGLE_RCTX を渡します。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、実行時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えます。新たに作成するアプリケーションでは、新しい関数名を使うことができます。

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方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
<code>sqlaltd()</code>	<code>struct SQLDA *SQLSQLDAAlloc(dvoid *context, unsigned int maximum_variables, unsigned int maximum_name_length, unsigned int maximum_ind_name_length);</code>	15-5 ページの <code>sqlaltd()</code> も参照してください。
<code>sqlcdat()</code>	<code>void SQLCDAFromResultSetCursor(dvoid *context, Cda_Def *cda, void *cursor, sword *return_value);</code>	4-28 ページの <code>sqlcdat()</code> も参照してください。
<code>sqlclut()</code>	<code>void SQLSQLDAFree(dvoid *context, struct SQLDA *descriptor_name);</code>	15-35 ページの <code>sqlcu()</code> も参照してください。
<code>sqlcurt()</code>	<code>void SQLCDAToResultSetCursor(dvoid *context, void *cursor, Cda_Def *cda, sword *return_value)</code>	4-28 ページの <code>sqlcur()</code> を参照してください。
<code>sqlglmt()</code>	<code>void SQLErrorGetText(dvoid *context, char *message_buffer, size_t *buffer_size, size_t *message_length);</code>	9-22 ページの「エラー・メッセージの全文の取得」の <code>sqlglm()</code> も参照してください。
<code>sqlglst()</code>	<code>void SQLStmtGetText(dvoid *context, char *statement_buffer, size_t *statement_length, size_t *sqlfc);</code>	9-31 ページの「SQL 文のテキスト取得」の <code>sqlglst()</code> 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
sqlld2t()	void SQLLDGetName(dvoid *context, Lda_Def *lda, text *cname, int *cname_length);	5-52 ページの sqlld2() も参照してください。
sqlldat()	void SQLCDAGetCurrent(dvoid *context, Lda_Def *lda);	5-48 ページの sqllda() を参照してください。
sqlnult()	void SQLColumnNullCheck(dvoid *context, unsigned short *value_type, unsigned short *type_code, int *null_status);	15-16 ページの sqlnul() 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 sqlprc() も参照してください。
sqlpr2t()	void SQLNumberPrecV7(dvoid *context, unsigned long *length, int *precision, int *scale);	15-15 ページの sqlpr2() 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」の sqlvcp() も参照してください。
N/A	sword SQLEnvGet(dvoid *context, OCIEnv **oeh);	5-44 ページの「SQLEnvGet()」も参照してください。
N/A	sword SQLSvcCtxGet(dvoid *context, text *dbname, int dbnamelen, OCISvcCtx **svc);	5-45 ページの「SQLSvcCtxGet()」も参照してください。
N/A	void SQLRowidGet(dvoid *context, OCIRowid **urid);	4-35 ページの「ユニバーサル ROWID」を参照してください。
N/A	void SQLExtProcError(dvoid *context, char *msg, size_t msglen);	外部プロシージャ内でのこの関数の使用方は、7-30 ページの「SQLExtProcError 関数」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われ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求めます。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

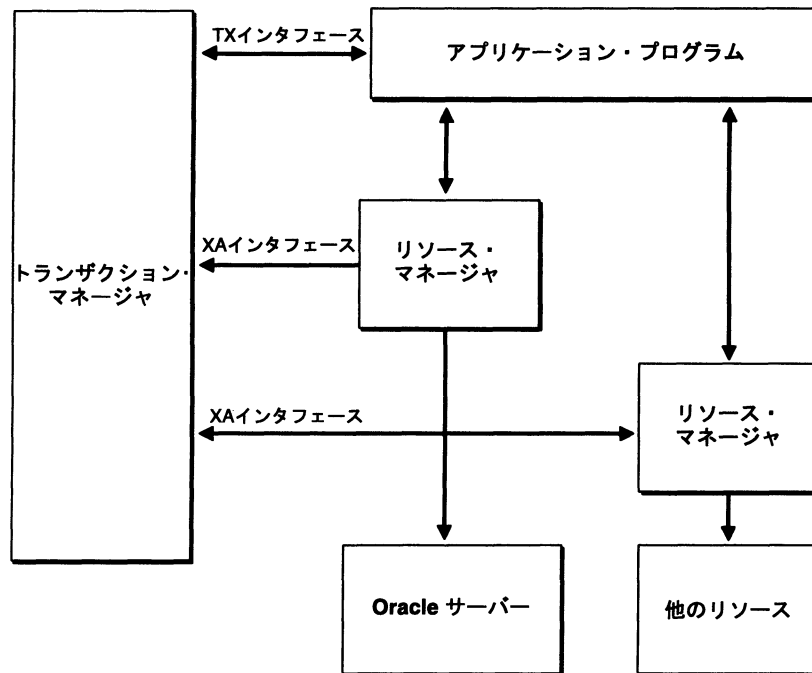


図 5-1 に DTP モデルのコンポーネントが Oracle8 データベースのデータに効率的なアクセスを行うよう相互作用できる方法のひとつを示します。この DTP モデルでは、リソース・マネージャとトランザクション・マネージャとの間に XA インタフェースが指定されています。Oracle は XA 準拠ライブラリを提供します。このライブラリは、X/Open アプリケーションにリンクする必要があります。また、アプリケーション・プログラムとリソース・マネージャ間で「固有のインタフェース」を指定する必要もあります。

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。これは以下の宛て先に書面を送ることで入手できます。

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って、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さなければなりません。

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も維持もしません。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 などの文を実行してはなりません。たとえば、アプリケーションで COMMIT 文を実行してはなりません。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てはなりません。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れば、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()` を使う必要があります。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 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。

以下の 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++ での使い方については、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・モデ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。

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。

- ホスト変数の使用方法
- 標識変数の使用方法
- 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

ホスト変数の使用方法

Oracle はホスト変数を使っ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ってデータを 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います。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、ホスト変数が使えますが、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 文があるとします。

```
EXEC SQL SELECT ename, empno INTO :name, :number FROM emp ORDER BY :ord;
```

この文には、見かけ上、: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 を使います。動的 SQL に関する情報は 13-1 ページの「Oracle 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。

標識変数の使用方法

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 文内で使うたびに、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って、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って、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なければなりません。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

標識変数を使うと、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: fetched column value is NULL
```

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って、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、関係演算子を使って 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 をもつ場合、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 文を使うと、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、DELETE などのデータ操作文を実行する場合、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だけです。これは、SQLCA を調べるだけでわかります。(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 章の「実行時エラーの処理」を参照してください。

SELECT 文（問合せ）を実行するときは、返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（有無を調べるだけ）
- 1 行だけを戻す問合せ
- 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）を使う必要があります。

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 章の「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 文を使うと、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 文を使うと、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 文およびディレクティブ」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 文を使います。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なければなりません。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 行だけを戻す場合で、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、カーソルを使ってこれらの行を FETCH するか、これらの行をホスト変数の配列内に SELECT しなければなりません。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 章の「ホスト配列」で説明します。

1 行だけ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 句をすべて使うことができます。

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 を使っ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 文を使うと、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ものでなければなりません。VALUES 句には挿入すべき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数に等しくなければなりません。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。

副問合せの使用法

「副問合せ」はネストされた SELECT 文です。副問合せを使うと、マルチパートの検索を実行できます。これを使うことができるのは以下の場合です。

- 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 文によって挿入すべき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って、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っていることに注意してください。

UPDATE 文の使用法

UPDATE 文を使うと、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 句を使うと、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、副問合せを使う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-108 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 文を使うと、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 句を使って、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 文でも使えます。オプションの WHERE 条件の後に指定します。上記の例では、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-41 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 句を使うと、表またはビュー内の検索条件を満たす行だけを SELECT、UPDATE、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ード・ファンクションを組み込みめます。

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 もしくは 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じでなければなりません。この句を使うと、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、または DELETE の前に行を選択する必要がありません。戻り句を使うと、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 章の「ANSI 動的 SQL」を参照してください。

カーソルの使用方法

検索で複数の行が返されると、カーソルを明示的に定義することで、以下のことができます。

- 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。

ホスト配列を使うこともできます。第8章の「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現在行を示します。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。次の文を使っ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 文を使っ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（ホスト配列を使っていない場合）。FETCH 文を使ってアクティブ・セット内の現在の行を取り出します。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 が終了したら、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 文の使用方法

次の例に示すように、DECLARE CURSOR 文を使っ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、宣言節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だけ 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 文は一意でなければなりません。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うときは、MAXOPENCURSORS オプションの指定が必要な場合があります。詳細は、第 10 章の「プリコンパイラのオプション」および付録 C の「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 文を使うと、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みの行カウントが 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ることはありません。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 しなければなりません。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 の「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 文を使うと、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。この行が現在行になります。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 (現在行を変更します)。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・セットを変更したい場合は、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って、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、同じデータ型をもっていなければなりません。

```
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、行がそれ以上なければ、FETCH は「データなし」エラー・コードを SQLCA の *sqlcode* もしくは SQLCODE または SQLSTATE 状態変数に返します。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うには、このカーソルを再度 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
- 「データなし」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 を終了したら、そのカーソルを CLOSE して、そのカーソルの 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 章の「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・オプションを使うと、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 オプション

プリコンパイラ・オプション PREFETCH を使うと、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行から PREFETCH オプション値で設定する行数は、標準的な慣例に従い、明示カーソルに関係するすべての問合せに使われます。PREFETCH オプションをインラインで使う場合、次のカーソル文のどれ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に取得する行数が 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。

オプティマイザ・ヒント

Pro*C/C++ プリコンパイラは、SQL 文中のオプティマイザ・ヒントをサポートしています。「オプティマイザ・ヒント」とは、Oracle SQL オプ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上書きできます。ヒントを使って、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う場合は、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ないでプラス記号 (+) 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 ヒントを使っています。

```
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・ヒントだけでなく、他のコメントも組み込みます。

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 の使い方」を参照してください。

次の例では、CURRENT OF 句を使って、*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うことはできません。

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 されるのではなく、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 8-25 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方法

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 文を使っ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、"no data found" のエラー・コードを *sqlca.sqlcode* に返します。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);
}
```

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 のカーソル使い方
- 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、『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 を使わないと、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 を使うと、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 を使えば、カーソルを定義し操作するのに DECLARE 文および OPEN 文、FETCH 文、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。これらを使うと、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、プロシージャを使って処理を行い、ファンクションを使って値を計算します。

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うことにより、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、新しい番号および名前、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・モードを使います。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 パラメータを使うと、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 パラメータを使うと、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 パラメータを使うと、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できなければなりません。表 7-1 はデータ型間の有効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、およびプログラム・オブジェクト、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、使用可能な型および定数、変数、例外、カーソル、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。以下の例では 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だけが参照可能で、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 という名前の複合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、主キーを使って、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 または DATE、NUMBER など) に属してもかまいませんが、主キーは BINARY_INTEGER 型に属していなければなりません。

ブロックまたはプロシージャ、ファンクション、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って 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 表の中の 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 属性を使って、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。複合データ型 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 型のオブジェクトを「レコード」といいます。PL/SQL 表と違って、レコードは固有の名前をも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、および給料、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。

ブロックまたはプロシージャ、ファンクション、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。

ドット表記法を使うと、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 ブロックを囲むだけで済みます。

```
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終わったら、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ので、必ず SQLCHECK=SEMANTICS コマンド行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 だけでなく 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、外国語の文字変数を宣言し、それを文字列関数（INSTRB および LENGTHB、SUBSTRB など）に渡すことができます。これにより、PL/SQL を使っ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・プログラムに戻せるようになります。

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 と使っ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* および *hire_date*、*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 データ型を使って、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に、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うには、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）を使って、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 演算子を使うと、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 (:=) を使って、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う場合は、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 ブロックに入る前の *ind_sal* の値が -1 であった場合には、*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っ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・コールが使えます。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 になります。

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っています。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って、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、13-24 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る場合にも、プロシージャ・コールを使用できます。例については、7-19 ページの「ストアード PL/SQL および Java サブプログラム」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要素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 型または LONG 型、RAW 型、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 有効なデータ型変換

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、索引の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います。しかし、その配列全体を処理したくない場合もあります。この場合には、ARRAYLEN 文を使って、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。文の構文は以下のとおりです。

```
EXEC SQL ARRAYLEN host_array (dimension) [EXECUTE];
```

ここで *dimension* は 4 バイト整数型のホスト変数です。リテラルないしは式ではありません。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）表示されなければなりません。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 の機能が使える場合もあります。次の例では、ARRAYLEN を使って、*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 要素に減らされるので、25 の配列要素だけが 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が一緒に送られます。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 文で使われるホスト配列には、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 索引表に結合されてはなりません。

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ていなければなりません。
- PL/SQL ブロックは一回だけ実行されます。
- EXEC SQL EXECUTE 文中のすべてのホスト変数は
 - ARRAYLEN EXECUTE で指定するか、
 - スカラーとします。

たとえば、以下の 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;
```

以下の 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う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;
}
```

以下の Pro*C/C++ ファンクションは動的な方法 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;
}
```

以下の 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; ";

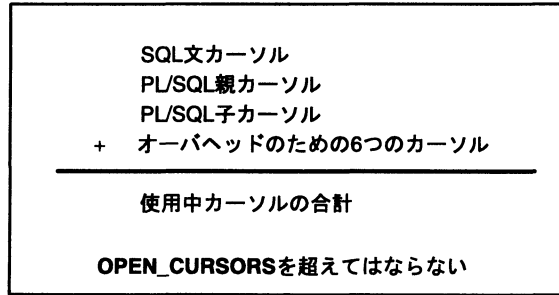
    EXEC SQL PREPARE s3 FROM :stmt3;
    EXEC SQL EXECUTE s3 USING :int_tab, :int_arr, :int_tab;
}
```

注意: ホスト配列についての完全な説明は、第 8 章の「ホスト配列」を参照してください。

埋込み PL/SQL のカーソル使い方

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。OPEN_CURSORS 制限に対して親カーソルと子カーソルの両方がカウントされます。図 7-1 の「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 ツールを使って明示的に作成したサブプログラムをストアド・サブプログラムといいます。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けば、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 および CREATE PROCEDURE、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 終了子 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 CharArrayTyp IS TABLE OF VARCHAR2(10)
        INDEX BY BINARY_INTEGER;
    TYPE NumArrayTyp 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   CharArrayTyp,
        job_title   OUT   CharArrayTyp,
        salary      OUT   NumArrayTyp);
END emp_actions;
```



```

END-EXEC;
EXEC SQL CREATE OR REPLACE PACKAGE BODY emp_actions AS

    CURSOR get_emp (dept_number IN INTEGER) IS
        SELECT ename, job, sal FROM emp
           WHERE deptno = dept_number;

    PROCEDURE get_employees(
        dept_number IN     INTEGER,
        batch_size  IN     INTEGER,
        found       IN OUT INTEGER,
        done_fetch  OUT    INTEGER,
        emp_name    OUT    CharArrayTyp,
        job_title   OUT    CharArrayTyp,
        salary      OUT    NumArrayTyp) 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 句を指定すれば、パッケージの削除および再作成、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しなければなりません。

```
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 定数もしくは 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、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、リテラルまたはプロシージャ・コール、式にしてはなりません。

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* (7-20 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。CALLEDemo 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なければなりません。また、ストアド・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なければなりません。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 ブロックを使う場合、SQLCHECK=SEMANTICS は必須です。

リモート・アクセス

PL/SQL を使うと、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* を使って、*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

ここで説明した埋込み 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返します。

```
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 文を使っている 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-16 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についての詳細な説明は、『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 章の「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* を使って、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さなければなりません。パッケージ化されたサブプログラムの場合は、パッケージの名前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）を返します。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うことができます。

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 文を使う必要があります。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。PL/SQL により、epctx がプロシージャに渡されます。

host_context は、外部プロシージャに返される実行時コンテキストです。現行の設定は、デフォルト（グローバル）コンテキストです。

REGISTER CONNECT 文により、現行 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）が返されます。これらのハンドルは、グローバル SQLLIB 実行時コンテキストの Pro*C/C++ デフォルト名前なし接続の定義に使われます。このため、REGISTER CONNECT が、CONNECT 文のかわりに使われます。

後続の埋込み SQL 文では、この OCI ハンドル・セットを使います。後続の埋込み SQL 文は、グローバル SQLLIB 実行時コンテキスト、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）実行時コンテキストはオプションの RETURNING 句に返されるようになります。

グローバル実行時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 を使った場合、実行時エラーが返されます。

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、接続を追加することはできません。実行時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 文は使うことができません。実行時エラーが返されます。Pro*C/C++ では、ここで説明している外部プロシージャを使わない場合は、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 文を使うことはできません。実行時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うことはできません。
- EXEC SQL OBJECT などのオブジェクト・ナビゲーション文を使うことはできません。
- EXEC SQL LOB 文のポーリングを行うことはできません。
- EXEC TOOLS 文を使うことはできません。実行時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。

次に、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す。ただし、CREATE 文のキーワード CONTEXT は、ブレース・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については、『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。

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 関数

SQLLIB 関数 SQLExtProcError() を使うと、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 実行時コンテキストです。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、グローバル実行時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、9-19 ページの「SQLCA の構造」を参照してください。

SQLExtProcError() の使い方の例です。

```
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\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 を使って、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 を使って 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 を使う必要があります。

詳細は、第 13 章の「Oracle 動的 SQL」、第 14 章の「ANSI 動的 SQL」および第 15 章の「Oracle 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、型 "table" のパラメータが指定された PL/SQL プロシージャに、ホスト配列をバインドすることはできません。詳細は、13-24 ページの「方法 4 の使用方法」を参照してください。

この章では、配列を使っ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。ここで紹介するのは、配列を使って Oracle データを操作する方法、単一の SQL 文を使っ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です。次の事項を扱います。

- どうして配列を使うのか？
- ホスト配列の宣言
- SQL 文での配列の使用方法
- 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
- 構造体の配列
- CURRENT OF の疑似実行
- sqlca.sqlerrd[2] の使用方法

どうして配列を使うのか？

配列を使うと、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。配列を使うと、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）を実行しなければなりません。配列を使えば、INSERT の実行は 1 回ですみます。

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 回のフェッチの配列によりアクセスできる最大バイト数は使用するリソースに依存します。最大サイズを超過するホスト配列を宣言すると、「パラメータ範囲外」実行時エラーが発生します。

SQL 文での配列の使用方法

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えます。

ホスト配列に使われる埋込み SQL 構文は、単純ホスト変数に使われる埋込み 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、データ操作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う場合、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します。この警告を無視すると、プリコンパイラは SQL 操作で要素の最小数を使用します。

次の例では、INSERT されるのは 25 行だけです。

```
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って、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てはいけません。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。

標識配列の使用方法

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 または SET、INTO、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なければなりません。

ホスト配列は UPDATE 文もしくは DELETE 文で CURRENT OF 句とともに使うことはできません。

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・オプションを使ってフラグできます。

配列を SELECT および FETCH するときは、必ず標識配列を使います。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]* を処理された行数に設定して、エラーを返します。

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えます。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 行だけを取り出したいときはこの方法を使います。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るにもかかわらず最初の 50 行だけがまた戻されます。この場合は大きな配列を宣言するか、FETCH 文で使う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 が返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出来ます。

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現行のアクティブ・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返された行数をチェックして処理することを忘れずに行ってください。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、UPDATE、DELETE および SELECT INTO 文の場合は、*sqlca.sqlerrd[2]* に処理済み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 でホスト配列を使っているときに、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ておいた）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。「データなし」Oracle エラー・コードは SQLCA 内で SQLCODE に返されます。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) */
```

Sample Program 3: ホスト配列

この項のデモ・プログラムでは、Pro*C/C++ で問合わせを作成するときにどのように配列を使うことができるかを示しています。特に、SQLCA (*sqlca.sqlerrd[2]*) の「処理済み行数」に注目してください。SQLCA の詳細は、第 9 章の「実行時エラーの処理」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\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-----      -");
    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

副問い合わせ文中を除き、SELECT 文の WHERE 句にホスト配列を使うことはできません。例については、8-15 ページの「WHERE 句の使用方法」の項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なければなりません。

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 のフェッチ

配列を SELECT および FETCH するときは、必ず標識配列を使います。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]* が処理済み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います。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えます。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 句を使って 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」の項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み行数 *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しなくてはならないため、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えません。つまり配列要素はすべてデータ項目でなければなりません。

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なければなりません。

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えます。

```
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 句を使っ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) を使っ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なければなりません。さらに、SET 句でホスト配列を使うときには、WHERE 句の要素数と同じ数のものを使ってください。

UPDATE 文の CURRENT OF 句ではホスト配列は使えません。別の方法については 8-25 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えます。これは、WHERE 句内のホスト配列の連続した要素を使って 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 行または 1 行、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) を使っ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なければなりません。

DELETE 文の CURRENT OF 句ではホスト配列は使えません。別の方法については 8-25 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 句を使うと、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 文および INSERT 文、DELETE 文内で FOR 句を使うと便利です。これらの文に配列全体を使う必要がないときもあります。次の例に示すように、FOR 句を使うと、使用する要素数を任意の数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えるほか、整数リテラルも使えます。整数値をとる C の複合式を使うことはできません。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符号なし数値として扱われます。符号付きホスト変数を使っ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 文では使えません。また CURRENT OF 句とともに使うことはできません。

SELECT 文中

SELECT 文中で FOR 句を使うと、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 文中では使えません。「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 句を使った方がよいでしょう。

```
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 句を使うと、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。（つまり現在行を 1 回だけ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には、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える出力変数が 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 できるようになっています。したがってエラー・メッセージは発行されません。

構造体の配列

1つの列で複数行を処理するには、スカラーの配列を使用できるようになりました。また1つの行を複数の列で操作するには、スカラーの構造体を使用できるようになりました。

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する方が便利です。

Pro*C/C++で「構造体の配列」がサポートされ、アプリケーション・プログラマはC構造体の配列を使って複数行および複数列の操作を実行できるようになりました。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。これにより、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だけでなく、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たな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 ブロック内で使うこともできません。これ以外の制限については 8-16 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、通常次のように使わ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 ブロック内で使ってはならない
- 構造体の配列を WHERE 句や FROM 句で使ってはならない

- 構造体の配列は Oracle 動的 SQL 方法 4 では使えません。ANSI 動的 SQL では使えます。詳細は、第 14 章の「ANSI 動的 SQL」を参照してください。
- 構造体の配列を UPDATE 文の SET 句で使ってはならない

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、構造体の配列を使う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うための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なければなりません。

- 構造体には必ず構造体タグを付ける。たとえば、次のコード・セグメントでは、

```
struct person {
    char name[15];
    int  VARCHARage;
} people[10];
```

person 変数は、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ってストライドのサイズを計算できます。

- 構造体のメンバーが配列であってはならない。ただし、**CHAR** や **VARCHAR** などの文字型の場合には、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われるためです。
- **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が構造体の配列のメンバーであってはなりません。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てはなりません。通常、この最大値は 2GB です。

構造体の配列の使用方法に上記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` が上記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きをします。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、さらに以下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、またはそれ以下でなければなりません。
- 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しなければなりません。
- 標識構造体に含まれるすべての要素のデータ型は、**SHORT** でなければなりません。

- 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、またはそれ以上でなければなりません。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。配列制限も以前使わ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時にはわかりません。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うときには、明示的な FOR 句を使わなければなりません。

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、さまざま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];
```

データの選択に使われる問合せは、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・データの一括処理機能に複数の配列を使う場合は、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だけで済みます。

```
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つだけ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;
```

は無効です。(また望ましくもありません。) 上の文には 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 および comm、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います。

```
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 句を使った場合も、取り出す行数について FETCH を指示できます。FOR 句は、SELECT 文が使わ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行で指定しなければならないため、1行だけ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なければなりません。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。上記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-1405: fetched column value is NULL
```

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、上記と同じ構造体の標識配列がまだ使われています。

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-1405 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えば複数の列を 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、上記の例はこれまで 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 を指定すると、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れば、さまざまな処理を実行できます。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。

```
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だけで関数を起動できます。

```
exec sql for :n insert into dept values (:dptr);
```

FOR 句の使い方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って現在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][18];
...
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 を使うことができません。)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。

実行時エラーの処理

アプリケーション・プログラムでは、ランタイム・エラーを見込んで、そのエラーを回復するように対処しなければなりません。この章では、エラーの報告および回復について詳しく説明します。SQL 通信領域 (SQLCA)、WHENEVER 文および SQLSTATE 状態変数を使っ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 通信領域 (SQLCA) の使用
- エラー・メッセージの全文の取得
- WHENEVER 文の使用
- SQL 文のテキスト取得
- ORACLE 通信領域 (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なければなりません。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、さまざま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ていくことはできます。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回復を意味します。

「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、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 データ操作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、特定の手段は推奨しません。最終的にどの手段を使うかは、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、エラーが回復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 9-14 ページの「SQLCODE の宣言」を参照してください。

SQL 通信領域

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: このガイドで使われる *sqlca* 構造体は、一般に「SQL 通信領域 (SQL Communications Area)」の頭字語 SQLCA を指して使用されています。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って 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 文のあとで SQLCA を更新します。(SQLCA の値は宣言文のあと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 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 文を使うか、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えます。

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 9-16 ページの「SQL 通信領域 (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* が使われます。

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 を使えます。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、現在の 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 34 ページの「ORACLE 通信領域 (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行オプション 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 とともに使っても別々に使ってもかまいません。

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・コードだけ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、状態変数としては SQLSTATE を使うのが望ましいといえます。SQL92 では SQLCODE は SQL89 との互換性を保つためだけに維持されている「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す。

SQLCODE は符号付き整数を格納するためのもので、宣言節の外で宣言できるのに対し、SQLSTATE は NULL 終了記号を付けた 5 文字の文字列を格納するためのもので、宣言節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なければなりません。

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 (「成功終了」) 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、事前定義済みの条件 (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。事前定義済みのクラスのうち、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、事前定義済みの副条件用に確保されています。他のサブクラス・コードはすべて処理系定義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

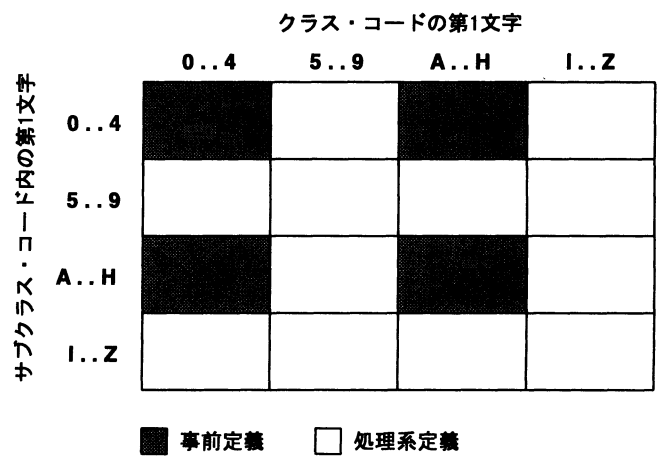


表 9-1 に SQL92 で事前定義済みのクラスを示します。

表 9-1 事前定義済みクラス

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

表 9-1 事前定義済みクラス

CLASS	条件
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
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・コードは導入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-1095 ORA-1403
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-2126
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-2121
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-1427 SQL-2112
22000	データ例外	
22001	文字列データの右側切捨て	ORA-1406
22002	NULL 値 - 標識パラメータなし	SQL-2124
22003	数値が範囲外	ORA-1426
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-1476
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	レパートリに文字がない	
22022	標識のオーバーフロー	ORA-1411
22023	パラメータ値が無効	ORA-1025 ORA-04000 ～ 04019
22024	C 文字列が未終了	ORA-1479 ORA-1480

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-1424 ORA-1425
22026	文字列データの長さが不一致	ORA-1401
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-001002 ORA-001003 SQL-2114 SQL-2117
25000	トランザクション状態が無効	SQL-2118
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-2091 ORA-2092
40001	直列化の障害	

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-1031 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-1402
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-2128
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 ディア回復のエラー	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-2100
82101	カーソル・キャッシュが矛盾 (UCE/CUC が不一致)	SQL-2101
82102	カーソル・キャッシュが矛盾 (UCE の CUC エントリがない)	SQL-2102
82103	カーソル・キャッシュが矛盾 (CUC 参照の範囲外)	SQL-2103
82104	カーソル・キャッシュが矛盾 (使用可能な CUC がない)	SQL-2104
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-2105
82106	カーソル・キャッシュが矛盾 (カーソル番号が無効)	SQL-2106
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 ; 再プリコンパイルが必要	SQL-2107
82108	ランタイム・ライブラリに無効な記述子が渡された	SQL-2108
82109	ホスト・キャッシュが矛盾 (SIT 参照が範囲外)	SQL-2109
82110	ホスト・キャッシュが矛盾 (SQL の型が無効)	SQL-2110
82111	ヒープ一貫性のエラー	SQL-2111
82113	コード生成の内部整合性の障害	SQL-2115
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-2116
82117	この接続での OPEN または PREPARE が無効	SQL-2122
82118	アプリケーション・コンテキストが見つからない	SQL-2123
82119	エラー・メッセージのテキストを取り出せない	SQL-2125
82120	プリコンパイラと SQLLIB のバージョンが不一致	SQL-2127
82121	NCHAR エラー ; フェッチされたバイト数が奇数	SQL-2129
82122	EXEC TOOLS インタフェースが使用できない	SQL-2130

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-2131
82124	ランタイム・コンテキストが割り当てられない	SQL-2132
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-2133
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 は宣言節内で宣言する必要があります。宣言節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。宣言節の内部で 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、宣言節の外部で SQLCODE を宣言すると、Oracle は SQLSTATE だけにステータス・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 を宣言してはいけません。

SQLSTATE を宣言しない場合

- 宣言節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 文を使って 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 文の後にだけ 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、宣言節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 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。標識変数が一つもなければ、Oracle はエラー・メッセージを発行します。

処理済み行数

最後に実行した SQL 文で処理された行の数は、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]` をチェックしてみると、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 文中の文字位置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、先頭の文字位置はゼロです。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。たとえば、次の動的 SQL 文は

```
"UPDATE emp SET jib = :job_title WHERE empno = :emp_number"
```

解析エラーとなります。

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（文字位置はゼロ）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` が負のときだけ、`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()` 関数を使います。詳細は 9-22 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 通信領域 (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、エラーおよび警告と、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 だけに情報を戻します。

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 を使っているとき、すべてのデータベースは 1 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「高度な接続オプション」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** 文を使って 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>
```

宣言節を使うときは、SQLCA は宣言節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るようになります。

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 文は SQLCA がなければ使えません。したがって、WHENEVER SQLWARNING 文を使用する場合は、SQLCA を宣言する必要があります。

注意: このガイドで使う SQLCODE は SQLCODE 状態変数のことを指し、*sqlca.sqlcode* は SQLCA 構造体のコンポーネントのことを特に指しています。

ORACA 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
- 処理済み行数
- 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.

```
*/
#ifndef 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];
};
#ifndef SQLCA_NONE
#ifdef SQLCA_STORAGE_CLASS
SQLCA_STORAGE_CLASS struct sqlca sqlca
#else
    struct sqlca sqlca
#endif
#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 の構造体およびそのコンポーネント、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・コードは以下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	文は実行されたものの、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 んどの場合は現行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。

<i>sqlerrml</i>	この整数コンポーネントには、 <i>sqlerrmc</i> 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。
<i>sqlerrmc</i>	この文字列コンポーネントには、 <i>sqlcode</i> 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 <i>sqlerrml</i> コンポーネントを使います。

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*** 関数を使います。

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。

<i>sqlerrd</i> [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。
<i>sqlerrd</i> [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。
<i>sqlerrd</i> [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</i> [2] 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</i> [2] は正常に処理された行数を示します。
<i>sqlerrd</i> [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。
<i>sqlerrd</i> [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位置を指定するオフセットを保持します。先頭の文字位置は 0（ゼロ）です。
<i>sqlerrd</i> [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。

処理済み行数は OPEN 文の後に 0 (ゼロ) に設定され、FETCH 文の後に増分されます。処理済み行数は、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の数を反映します。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み行数は、25 ではなく 20 となります。

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」(ワーニング (警告) の意) 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。

<code>sqlwarn[0]</code>	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。
<code>sqlwarn[1]</code>	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だけ適用されます。つまり Oracle は、警告を設定することも負の <code>sqlcode</code> を戻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、またどれだけ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。

sqlext

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、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、処理済み行数 (*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* コンポーネントだけを使ってください。

エラー・メッセージの全文の取得

SQLCA には 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。*sqlglm()* の構文は

```
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>	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 2 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だけコールできます。`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 文と関連したメッセージ・テキストが得られます。

WHENEVER 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 文が必要です。

WHENEVER 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 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。）を返したか、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 を使うには、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 文を使わない場合と同じです。このアクションを使う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 ... 文に呼び出されるエラー・ハンドラにパラメータを渡し、関数によって値を返すことができます。

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り返しに移ります。

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 文字だけです。異なる最大長を必要とする C コンパイラもあります。お使いの 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()* コールを生成するだけです。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、

- 「データが見つからない」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。

という処理を実行するときは、最初の実行 SQL 文の前に次の WHENEVER 文を指定するだけで済みます。

```
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 文を使っ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だけ、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 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 文は、同じ条件をチェックする別の WHENEVER 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 文は 2 番目のものに置き換えられます。したがって、この文の制御は CONNECT 文だけに適用されます。2 番目の WHENEVER SQLERROR 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 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 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検出できます。

データの終わり条件の処理

カーソルを使って行をフェッチするときは、プログラムはデータの終わり条件に対処できるようになっていなければなりません。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;
...
```

無限ループの回避

WHENEVER SQLERROR GOTO 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、注意して使わ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 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 文の範囲内にないため、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 文の分岐先のラベルは、この文と同じプリコンパイル・ファイル内になければなりません。

エラー後の戻り

エラーの処理後にプログラムに戻らなければならないときは、DO *routine_call* アクションを使います。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 文または WHENEVER STOP 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 を使う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) — QLLIB ランタイム・ライブラリの一部 — は次の情報を返します。

- 最後に解析された SQL 文のテキスト
- 文の有効な長さ
- 文で使われている 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 されたらすぐに *SQLStmtGetText()* をコールできます。

QLLIB 関数の新しい名前は 5-48 ページの「QLLIB 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 *stmlen, size_t *sqlfc);
```

コンテキスト・パラメータは実行時コンテキストです。コンテキストの定義と使い方は 4-33 ページの「CONTEXT 変数」を参照してください。

sqlstmt 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。

stmllen 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 *sqlstmt* 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、*sqlstmt* 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。*stmllen* パラメータは、戻された文テキストの実際のバイト数を戻します。これは埋め込まれた空白のバイト数を含みません。*stmllen* 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 機能コードを戻す *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

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
17	CREATE SCHEMA	42	CREATE EXTERNAL DATABASE	67	ALTER PROFILE
18	CREATE CLUSTER	43	DROP EXTERNAL DATABASE	68	DROP PROCEDURE
19	CREATE USER	44	CREATE DATABASE	69	(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 (*stmlen*) 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 機能コードはありません。

サンプル・プログラム

第4章の「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 通信領域 (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 が提供する実行時の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 を使います。ORACA には豊富な診断ツールが用意されています。ただし ORACA の使用は実行時の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。

問題を診断する手助けのほかに、ORACA はプログラムの Oracle 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、プログラム内での適用範囲によって制限されます。Oracle は適用範囲内の ORACA だけに情報を戻します。

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・ディレクティブを使って 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 **extern** 記憶クラスでなければならないときは、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
```

プログラムで宣言節を使うときは、ORACA を宣言節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、コマンド行に次のように 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 orasfrml;
        char  orasfrmc[70];
        } orasfrm;      /* 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                     */
    };

#ifdef 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、以下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 (*oradbfg*) 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』に一覧にしています。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。

oradbfg

このマスター・フラグを使うと、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 (*oradbfg*) 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

このフラグを使うと、現行の 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 文のテキストだけを保存します。
- SQLERROR または SQLWARNING の SQL 文のテキストだけを保存します。
- 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います。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現行の SQL 文の長さが格納される。
orastxtc	この文字列コンポーネントには現行の SQL 文のテキストが格納される。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、 orastxl 長さコンポーネントを使います。

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、現行の 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る。
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る。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現行の 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 (**oradbfg**) およびカーソル・キャッシュ・フラグ (**oracchf**) が設定されているときは、次の変数を使うとカーソル・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 内の現在の設定値は、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* よりも大きくなることがあります。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 カーソルの現在の数を記録します。

oranor

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、できる限り高く保たなければなりません。つまり、不要な再解析は回避しなければなりません。ヘルプについては付録 C の「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 (strcmp(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.oraahoc);
    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、次のトピック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
- プリコンパイル中になにが起きているか？
- オプションの適用範囲
- 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

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>...
```

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うことによって、Pro*C/C++ 実行ファイルをアクセス可能にします。

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
```

現行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行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、コマンド

```
proc myfile
```

は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付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。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、大文字小文字の区別に関してはオペレーティング・システムの表記法に従ってください。

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

各オプションを使うと、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。たとえばオプション

```
... SELECT_ERROR=yes
```

は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。

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
- コマンド行で設定される値
- 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。このオプションをコマンド行で設定すると、その値がプリコンパイラのデフォルト値、システム構成ファイルの設定値およびユーザー構成ファイルの設定値より優先されます。

最終的には、インライン指定が上記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 10-5 ページの「構成ファイル」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。

現在の設定値を調べる

コマンド行で疑問符 (?) を使うと、1 つ以上のオプションの現在の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、それらの現在の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ビルトインされているもので、システム構成ファイルの値によって上書きされています。しかしコマンド

```
proc config=my_config_file.h ?
```

を入力したときに、現行ディレクトリに *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 つずつ指定しなければならない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 =? をつけるだけでどれか 1 つのオプションの現在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 オプションの現行の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 の「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

DBMS および MODE オプションは複数のオプションを同時に制御します。それらのオプションはマクロ・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、10-3 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
MODE=ORACLE	CLOSE_ON_COMMIT=NO END_OF_FETCH=1403 DYNAMIC=ORACLE TYPE_CODE=ORACLE
DBMS=NATIVE V7 V8	UNSAFE_NULL=NO

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、コマンド行で CLOSE_ON_COMMIT=YES を指定すると、カーソルはクローズします。

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。構成ファイルには、たとえば次のような行が含まれています。

```
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・オプションを使っ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: 構成ファイルをネストすることはできません。つまり、構成ファイル内では、CONFIG= は有効ではありません。

プリコンパイル中になにが起きているか？

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 QLLIB 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 QLLIB ルーチンのコールも入っています。

注意: プリコンパイラにより Oracle コール・インタフェース (OCI) ルーチンの呼び出しは生成されません。

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。ここで 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・ユニットにだけ効力を持ちます。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 値を使って実行されますが、ユニット B の SQL 文はデフォルト値を使って実行されます。

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	自動 OPS\$ アカウント
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する
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する
DEFINE= <i>name</i> *	なし	Pro*C/C++ プリコンパイラで使う名前を定義する
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する
DYNAMIC={ANSI ORACLE}	ORACLE	Oracle または ANSI SQL の意味を指定する
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・ファイルだけに送られ、端末には送られない)
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

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
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=]filename	なし	入力ファイルの名前
INCLUDE=pathname*	なし	EXEC SQL INCLUDE 文または #include 文のディレクトリ・パス
INTYPE=filename	なし	型情報の入力ファイル名
LINES={YES NO}	NO	#line ディレクティブを生成するかどうか
LNAME=filename	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・カーソルの数
MODE={ANSI ISO ORACLE}	ORACLE	ANSI/ISO または Oracle の動作
NLS_CHAR=(var1,..., varn)	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=]filename	iname.c	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
PAGELLEN=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する
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=pathname	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

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
THREADS={YES NO}	NO	マルチスレッド・アプリケーションを指定する
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る
USERID=username/password[@dbname]	なし	username/password[@dbname] 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返すか

オプションの入力

どのプリコンパイラ・オプションも、コマンド行に入力できます。また、その多くは、EXEC ORACLE OPTION 文を使ってプリコンパイラ・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。

コマンド行

プリコンパイラ・オプションをコマンド行に入力するには、次の構文を使います。

```
... [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、sel_desc1、sel_desc2、sel_desc3 という 3 種類の選択 SQLDA を宣言すると、同時に OPEN している 3 つのカーソルから FETCH できます。

```
... CODE=ANSI_C MODE=ANSI
```

インライン

次の構文を使って 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、次の問合せが 2 つの列値を戻すことがわかっているとします。

```
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、文ごとに HOLD_CURSOR および RELEASE_CURSOR を変更する場合があります。付録 C の「パフォーマンスの最適化」にインライン・オプションを使って実行時性能を最適化する方法を示します。

また、コマンド行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、コマンド行または構成ファイルからだけです。

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ければ、アプリケーションは次のユーザー ID を使って接続を試みます。

OPS\$<username>

このとき *username* は現行のオペレーティング・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 文を使う必要があります。

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 文を使って 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、コマンド行または構成ファイルからだけです。

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行で 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様式を指定します。(関数プロトタイプを使っ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

コマンド行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って他の 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うすべての結果は 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、コマンド行だけです。

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。上記で生成された 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 コンパイラを用いてシフト 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、コマンド行だけです。

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 オプションを使うと、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 オプションを使うと、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）のど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、コマンド行または構成ファイルからだけです。

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	DBMS=V7 V8
	MODE=ANSI	MODE=ORACLE
「データなし」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
標識変数を使わない切捨て値のフェッチ	エラーはなし。 SQLWARN (2) が設定されます。	エラーはなし。SQLWARN (2) 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、コマンド行または構成ファイルからだけです。

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 を使って実行 SQL 文の結果をチェックできるようになります。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。

また次の行のうち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う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・ディレクティブで用いることの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 文でも用いることができます。

構文

DEFINE=*name*

デフォルト値

なし

使用上の注意

コマンド行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 文を正しく使えば、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るようになります。

```
#ifdef XYZZY
...
#else
...
#endif
```

または、単純にコーディングすることもできます。

```
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**）ディレクティブ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、C コンパイラの名前を *-D* オプションを使用して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 文に使わ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 文を使っ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 文を使っ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・ファイルだけに送信（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、コマンド行または構成ファイルからだけです。

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つだけ生成されます。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

NO

使用上の注意

インラインまたはコマンド行で入力できます。

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の方法で ANSI SQL 機能を使うと、警告メッセージ（エラーではない）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
- 実行時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 文の `DO function_name()` および「do break」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

プリコンパイル済みヘッダー・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われます。

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えるようになります。

どちらにしろ、このオプションで `#include` ディレクティブを処理するときに使うファイル拡張子も指定できます。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` ディレクティブ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、コマンド行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う場合、ファイル拡張子だけを指定します。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「プリコンパイル済みのヘッダー・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行で入力できます。

HOLD_CURSOR を使うと、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 の「パフォーマンスの最適化」を参照してください。

SQL データ操作文を実行すると、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に再利用しません。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うときは、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うときは、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 は HOLD_CURSOR=YES を上書きし、HOLD_CURSOR=NO は RELEASE_CURSOR=NO を上書きすることに注意してください。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、コマンド行だけです。

すべての入力ファイル名はプリコンパイル時に固有でなければなりません。

.pc のファイルの拡張子は省略できます。入力ファイル名がコマンド行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 モードを使ってファイル *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 ディレクティブによって取り込まれ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++ に組み込まれているカレント・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行で入力できます。

INCLUDE は組み込ま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います。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. 現行ディレクトリ
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・ファイル用のビルトイン・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: 組み込みファイルに Oracle 固有のファイル名を拡張子なしで指定する場合、Pro*C/C++ では *.h* の拡張子が付いているものと見なされます。このため、組み込みファイルには、*.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、現行ディレクトリに格納されていない限り、INCLUDE を使ってディレクトリ・パスを指定する必要があります。次に示すように、コマンド行に複数のパスを指定できます。

```
... INCLUDE=path_1 INCLUDE=path_2 ...
```

警告: 組み込むファイルが別のディレクトリに存在している場合、同じ名前のファイルが現行ディレクトリに存在しないようにしてください。

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われる場合にだけです。)

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・ディレクティブ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、コマンド行だけです。

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・ディレクティブを追加します。

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** ディレクティブ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** ディレクティブ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っ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、修正済みのソース・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。

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** ディレクティブを追加しません。

注意：5-28 ページの「無視されるディレクティブ」では、Pro*C/C++ プリコンパイラは **#line** ディレクティブ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** ディレクティブ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= オプションを使えば、プリコンパイラに **#line** ディレクティブ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、コマンド行だけです。

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

コマンド行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だけが出力され、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がメッセージ条件を生成したコードを突き止める手助けになります。

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・ファイルは生成されません。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。そのようにしない場合、Pro*COBOL 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行で入力できます。

MAXOPENCURSORS を使うと、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 の「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 2-11 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。新しいカーソルが必要で、空きのキャッシュ・エントリがない場合、Oracle はエントリを再利用しようとします。これが成功するかどうかは HOLD_CURSOR と RELEASE_CURSOR の設定値によって決まります。また明示カーソルの場合は、さらにカーソル自体のステータスにも左右されます。再利用できるキャッシュ・エントリが見つからない場合、Oracle は追加のキャッシュ・エントリを割り当てます。

Oracle 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、必要に応じてキャッシュ・エントリの割当てを続行します。MAXOPENCURSORS は「最大オープン・カーソル数超過」Oracle エラーを避けるために、6 以上で、OPEN_CURSORS より小さくなければなりません。

プログラムが同時オープン・カーソル数を増やさなければならない場合には MAXOPENCURSORS を必要な数まで増やして指定し直したくなることがあるでしょう。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、または現行の 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、コマンド行または構成ファイルからだけです。

ISO は ANSI の同義語です。

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、宣言節はオプションで、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 文の範囲内に 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）を宣言しなければなりません。すべての場合に同一の SQLCODE または SQLSTATE 変数を使う必要はありません。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 に返される「データなし」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。
- ホスト変数に宣言節が必要です。

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、コマンド行または構成ファイルからだけです。

このオプションを使うと、プリコンパイラで各国語文字変数として扱うべき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 変数だけを指定できます。

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す。

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={NO | YES}

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 を使って、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います。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、コマンド行または構成ファイルからだけです。

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

コマンド行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、コマンド行でだけです。このオプションを使うと、入力 (.pc) ファイルとは異なるパス名を使って、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
```

デフォルト出力ファイル名は *my_test.c* です。出力ファイル名を *my_test_1.c* にしたい場合は、以下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わず、ONAME を使って明示的に名前をつ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行で入力できます。

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節の内側で宣言する必要があります。2-11 ページの「宣言節」を参照してください。

変数 SQLCODE を宣言節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・ディレクティブ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。2-15 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い合わせをスピード・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行で入力できます。優先順位の規則に従い、明示カーソルを使うすべての問合せに整数の値が実行に使われます。

インラインで使う場合、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い方法を指定します。

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行で入力できます。

RELEASE_CURSOR を使うと、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 の「パフォーマンスの最適化」を参照してください。

SQL データ操作文を実行すると、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 時に関連リソースが確実に解放されるようにするには、RELEASE_CURSOR=YES を指定する必要があります。

RELEASE_CURSOR=NO と HOLD_CURSOR=YES を指定すると、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うときは、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うときは、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` を上書きし、`HOLD_CURSOR=NO` は `RELEASE_CURSOR=NO` を上書き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 の「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行で入力できます。

`SELECT_ERROR=YES` のときは、行単位の `SELECT` 文が戻した行の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` 文が戻した行の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

構文および意味チェック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行で入力できます。

完全な詳細は付録 D の「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 コマンド行オプションを使うと、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 を使っ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せん。

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 ディレクティブを付けて、挿入しなければなりません。)

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. 現在のディレクトリ
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 *sqlda.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 になっていると、プリコンパイラは最初のコンテキストが見えて、実行可能 SQL 文が見つかるまでの間に EXEC SQL CONTEXT USE ディレクティブが現れなければエラーを生成します。詳細は第 11 章の「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 データ型を使う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、標識変数を使わないで 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 でしかも DBMS=V7 または V6_CHAR 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、必ず標識変数を使ってください。

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、コマンド行だけです。

自動修正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。自動修正機能では、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、コマンド行だけです。

VARCHAR=YES のとき、次のように C の構造体を記述すると、


```
struct {  
    short <len>;  
    char  <arr>[n];  
} name;
```

は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 文を使ってインライン入力できます。

RECENT は、現行のトランザクション内でオブジェクトが選択され、オブジェクト・キャッシュに入れられていれば、そのオブジェクトが戻されることを意味します。直列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ていなければ、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・キャッシュに存在している場合は、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ので、慎重に使ってください。LATEST を使うのは、オブジェクト・キャッシュとサーバー側のバッファ・キャッシュと可能な限り一致させておかなければならない場合だけです。

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れば、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。この値を使うのは、アプリケーションが読み込み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。

数字

- 2 タスク
- リンク, 2-18

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, xxxv
- ANSI C サポート, E-2
- ANSI, 住所, xxxv
- ANSI 動的 SQL, A-4
 - 「動的 SQL (ANSI)」を参照, 14-1
 - リファレンス・セマンティクス, 14-7
- ARRAYLEN 文, 7-16
- ARRAYLEN 文のオプション・キーワード EXECUTE, 7-17
- AT 句
 - COMMIT 文の, F-24
 - CONNECT 文中, 3-8
 - DECLARE CURSOR 文, 3-9
 - DECLARE STATEMENT 文, 3-10
 - DECLARE CURSOR ディレクティブの, F-34
 - DECLARE STATEMENT ディレクティブの, F-37
 - EXECUTE IMMEDIATE 文, 3-10
 - EXECUTE IMMEDIATE 文の, F-55
 - EXECUTE 文の, F-49
 - INSERT 文, F-66
 - SAVEPOINT 文の, F-99
 - SELECT 文の, F-102
 - UPDATE 文の, F-109

- 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, xxxiv
- BREAK アクション
 - WHENEVER の, F-116

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-15
- CALL SQL 文, F-16
- CALL 文, 7-26, A-2
 - 例, 7-27
- CASE OTT パラメータ, 19-28
- CHARF データ型, 4-10, 5-14
- CHAR_MAP プリコンパイラ・オプション, 5-2, 10-11, A-3
- CHARZ データ型, 4-10
- CHAR データ型, 4-9
- CLOSE CURSOR 文, 14-25
- CLOSE_ON_COMMIT
 - プリコンパイラ・オプション, 10-12, A-5
- CLOSE SQL 文, F-17
- CLOSE 文, 6-14

- 動的 SQL 方法 4 での使用, 15-35
- 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-18
- 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
- OBJECT GET 文
 - 例, 18-18
- COLLECTION APPEND, F-18
- COLLECTION APPEND 文, 18-11
 - SQL 文
 - COLLECTION APPEND, F-18
- COLLECTION DESCRIBE
 - 例, 18-19
- COLLECTION DESCRIBE 文, 18-13
 - SQL 文
 - COLLECTION DESCRIBE, F-19
- COLLECTION GET 文, 18-7
 - SQL 文
 - COLLECTION GET, F-21
- COLLECTION RESET 文, 18-11
 - SQL 文
 - COLLECTION RESET, F-21
 - 例, 18-20
- COLLECTION SET 文, 18-9
 - SQL 文
 - COLLECTION SET, F-22
 - 例, 18-18
- COLLECTION TRIM 文, 18-12
 - SQL 文
 - COLLECTION TRIM, F-23
- COMMENT 句
 - COMMIT 文の, F-24
- COMMIT SQL 文, F-23
- COMMIT 文, 3-16
 - PL/SQL ブロック内での使用, 3-25
 - RELEASE オプションを含む, 3-16
 - 影響, 3-16
 - トランザクションを終了, F-97
 - 配置する場所, 3-16
 - 用途, 3-16
 - 例, 3-16, F-25
- COMP_CHARSET プリコンパイラ・オプション, 10-13
- 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-25
 - 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-27
- const
 - 定数の宣言, 5-42
- CONTEXT ALLOCATE SQL 文, F-27
- CONTEXT ALLOCATE 文, 11-9
- CONTEXT FREE 文, 11-10, F-28
- CONTEXT OBJECT OPTION GET SQL 文, 17-18
- CONTEXT OBJECT OPTION SET SQL 文, 17-17
- CONTEXT USE SQL ディレクティブ, F-31
- CONTEXT USE SQL 文, 11-9
- CONTEXT USE ディレクティブ, 11-9
- CONTINUE アクション
 - WHENEVER ディレクティブの, F-115, F-116
 - WHENEVER 文, 9-25
 - 結果, 9-25
- CONVBUFSZ 句, 4-48
- 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, 7-20
- CURRENT OF 句, 8-4
 - 用途, 6-16
 - ROWID で代用, 3-23, 8-25
 - 制限, 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-26
- DDL トランザクションでの, 3-15
- DEALLOCATE DESCRIPTOR 文, 14-13, F-33
- DECLARE CURSOR ディレクティブ
 - 例, F-35
- DECLARE CURSOR 文, 14-23
 - AT 句, 3-9
 - 動的 SQL 方法 4 での使用, 15-23
- DECLARE DATABASE SQL ディレクティブ, F-36
- DECLARE STATEMENT ディレクティブ, F-37
- DECLARE STATEMENT 文
 - AT 句, 3-10
 - 使用例, 13-27
 - 動的 SQL での使用, 13-27
 - 必要な場合, 13-27
- DECLARE TABLE SQL ディレクティブ, F-38
- DECLARE TABLE 指示行
 - SQLCHECK オプションで使用, D-4
- DECLARE TABLE ディレクティブ
 - 例, F-40
- DECLARE TABLE 文
 - AT 句と一緒に必要, 3-9
- DECLARE TYPE ディレクティブ, F-40
- DECLARE 文, 6-12
 - 適用範囲, F-38
 - 動的 SQL 方法 3 での使用, 13-19
 - 配置が必要な, 6-12
 - 用途, 6-11
 - 例, 6-11, F-38
- 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, 5-33
- DEF_SQLCODE プリコンパイラ・オプション, 10-17
- DELETE CASCADE, 9-21
- DELETE SQL 文, F-41
- DELETE 文
 - WHERE 句を含む, 6-10
 - 埋込み SQL の例, F-44
 - ホスト配列の使用, 8-12
 - 用途, 6-10
 - 例, 6-10
- DEPT 表, 2-18
- DESCRIBE BIND VARIABLES 文
 - 動的 SQL 方法 4 での使用, 15-23
- DESCRIBE DESCRIPTOR 文, F-46
- DESCRIBE INPUT 文, 14-20
- DESCRIBE OUTPUT 文, 14-21
- DESCRIBE SELECT LIST 文

- 動的 SQL 方法 4 での使用, 15-28
- DESCRIBE SQL 文, F-45
- DESCRIBE コマンド
 - PREPARE コマンドとともに使用, F-45
- DESCRIBE 文
 - 動的 SQL 方法 4 での使用, 13-24
 - 例, F-46
- DML 戻り句, 6-10, A-5
- DO アクション
 - WHENEVER ディレクティブの, F-116
 - WHENEVER 文, 9-25
 - 結果, 9-25
- DTP モデル, 5-51
- DURATION プリコンパイラ・オプション, 10-19, 17-19

E

- EMP 表, 2-18
- ENABLE THREADS SQL 文, F-48
- 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 文
 - 構文, 10-9
 - 適用範囲, 10-10
 - 用途, 10-10
- EXEC ORACLE DEFINE 文, 5-40
- EXEC ORACLE ELSE 文, 2-15, 5-40
- EXEC ORACLE ENDIF 文, 2-15, 5-40
- EXEC ORACLE IFDEF 文, 2-15, 5-40
- EXEC ORACLE IFNDEF 文, 2-15, 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, 2-15
- EXEC SQL CACHE FREE 文, 17-6
- EXEC SQL INCLUDE
 - #include との対比, 5-33
- EXEC SQL VAR 文
 - CONVBUSZ 句, 4-48
- EXEC SQL 句
 - SQL 埋込みのための使用, 2-5
- EXEC TOOLS
 - GET CONTEXT 文, 20-16

GET 文, 20-15
MESSAGE 文, 20-16
SET CONTEXT 文, 20-16
SET 文, 20-15
EXEC TOOLS 文, 20-14
EXECUTE DESCRIPTOR 文
SQL 文
EXECUTE DESCRIPTOR, F-52
EXECUTE ... END-EXEC SQL 文, F-49
EXECUTE IMMEDIATE SQL 文, F-54
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-50
EXECUTE 文, 14-22
動的 SQL 方法 2 での使用, 13-12
例, F-50, F-52
EXPLAIN PLAN 文
機能, C-6
効率改善のために使用, C-5

F

FETCH DESCRIPTOR SQL 文, F-58
FETCH SQL 文, F-56
FETCH 文, 14-25
INTO 句を含む, 6-13
OPEN コマンドの後に使用, F-92
OPEN 文の後に使用する, F-89
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, 15-33
用途, 6-11, 6-13
例, 6-13, F-58
FIPS フラガー
宣言節の欠如を警告, 4-11
配列の使用方法に関する警告, 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 文の, F-25
ROLLBACK 文の, F-96
FOR UPDATE OF 句
使用する場合, 3-22

用途, 3-22
を用いた行のロック, 3-22
FOR 句
埋込み SQL EXECUTE DESCRIPTOR 文, F-53
埋込み SQL EXECUTE 文の, F-51
埋込み SQL INSERT 文の, F-66
使用例, 8-13
制限, 8-14
動的 SQL 方法 4 で使用, 15-35
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, 8-13
用途, 8-13
free() 関数, 15-35
使用例, 15-35
FREE SQL 文, 17-5, F-61

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, 20-12
ユーザー・イグジットでの使用方法, 20-12
GET DESCRIPTOR 文, 14-13
GOTO アクション
WHENEVER ディレクティブの, F-115
WHENEVER 文, 9-25
結果, 9-25

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 オプション
ORACLE プリコンパイラ, F-18
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 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, 9-16
使用, プリコンパイラ・オプション, 5-32
#include
ファイルの組込み, Pro*C の C との比較, 5-27
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, 19-27
IN OUT パラメータ・モード, 7-3
INSERT SQL 文, F-65
例, F-67
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
INTO 句
FETCH DESCRIPTOR 文の, F-59
FETCH 文中の, 6-13
FETCH 文の, F-57
INSERT 文中の, 6-8
SELECT のかわりに FETCH を伴った, 6-12
SELECT 文中の, 6-7
SELECT 文の, 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, 19-7
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, xxxv

L

LDA, 5-47
OCI rel 8 の設定, 5-47
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, 1-10
LOB
BFILES, 16-2
C のロケータ, 16-7
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-68
LOB ASSIGN 文, 16-12
LOB CLOSE SQL 文, F-69
LOB CLOSE 文, 16-12
LOB COPY SQL 文, F-70
LOB COPY 文, 16-13
LOB CREATE TEMPORARY SQL 文, F-70
LOB CREATE 一時文, 16-14
LOB DESCRIBE SQL 文, F-71
LOB DISABLE BUFFERING SQL 文, F-72
LOB DISABLE BUFFERING 文, 16-15
LOB ENABLE BUFFERING SQL 文, F-72
LOB ENABLE BUFFERING 文, 16-15
LOB ERASE SQL 文, F-73
LOB ERASE 文, 16-16
LOB FILE CLOSE ALL SQL 文, F-74
LOB FILE CLOSE ALL 文, 16-16
LOB FILE SET SQL 文, F-74
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-75
LOB FREE TEMPORARY 文, 16-18
LOB LOAD FROM FILE 文, 16-19

LOB LOAD SQL 文, F-76
LOB OPEN SQL 文, F-77
LOB OPEN 文, 16-20
LOB READ SQL 文, F-77
LOB READ 文, 16-21
LOB TRIM SQL 文, F-78
LOB WRITE SQL 文, F-78
LOB すべてのファイルを終了文, 16-16
LOCK TABLE 文
 NOWAIT パラメータを含む, 3-23
 用途, 3-23
 例, 3-23
 を用いた表のロック, 3-23
LONG RAW データ型, 4-9
LONG VARCHAR
 データ型, 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, 15-30
 用途, 15-30
MAXLITERAL
 デフォルト値, 2-14
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, 2-17
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, 10-29

N

NATIVE
 DBMS オプションの値, 10-15
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, xxxv
NIST, 住所, xxxvi
NLS_CHAR プリコンパイラ・オプション, 10-30
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 ディレクティブの, F-115
 WHENEVER 文, 9-24
 意味, 9-24
NOWAIT パラメータ
 LOCK TABLE 文中の, 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, 2-7
 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, 4-6
INTEGER データ型, 4-6
NUMBER データ型, 4-5
 sqlprc() 関数を使用, 15-15

O

OBJECT CREATE SQL 文, 17-9, F-79
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 RELEASE SQL 文, F-85
 OBJECT SET SQL 文, 17-14, F-86
 OBJECTS プリコンパイラ・オプション, 10-31, 17-20
 OBJECT UPDATE SQL 文, 17-11, F-87
 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
 OCIStrng, 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
 OCIStrng, 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
 環境ハンドルのパラメータ, 5-43
 へのインタフェース, 5-44
 OCI リリース 8
 オブジェクトのアクセスおよび操作, 19-18
 ONAME プリコンパイラ・オプション, 10-32
 使用上の注意, 10-32
 OPEN CURSOR 文, 14-24
 OPEN DESCRIPTOR SQL 文, F-90
 OPEN SQL 文, F-88
 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, 15-28
 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-90
 ORACA, 9-3
 カーソル・キャッシュ統計情報の収集, 9-39
 使用例, 9-40
 ORACAID コンポーネント, 9-37
 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, 2-7
 Oracle コール・インタフェース Rel 7, 5-47
 ORACLE 識別子
 形成の方法, F-11
 Oracle 通信領域, 9-34
 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, 19-28
 CODE, 19-26
 CONFIG, 19-27
 ERRTYPE, 19-27
 HFILE, 19-27
 INITFILE, 19-26
 INITFUNC, 19-27
 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, 10-33, 12-4
- PL/SQL, 1-4
 - AT 句を使ったブロックの実行, 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, 9-22
 - 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, 4-26
 - 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-49
- PREFETCH プリコンパイラ・オプション, 10-34, A-4
- PREPARE SQL 文, F-92
- PREPARE 文, 14-19
 - 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-94
- Pro*C/C++ プリコンパイラ
 - OTT の使用, 19-21
 - 新しいデータベース型, 17-34
- ProCC プリコンパイラ
 - NLS 用のサポート, 4-46
- 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
 - PL/SQL の使用, 7-6
 - 一般的な使用方法, 1-3
 - オブジェクト・サポート, 17-1
 - 実行時コンテキスト, 5-43
 - 新機能, A-1 ~ A-6

R

- RAW データ型, 4-8
- READ ONLY パラメータ

- SET TRANSACTION 文中の, 3-21
- REF
 - 埋込み SQL での使用, 17-32
 - 構造体, 17-32
 - 使用, 17-32
 - 宣言, 17-32
- REFERENCE 句
 - TYPE 文, 5-13
- REF (オブジェクトへの参照), 17-2
- REGISTER CONNECT SQL 文, F-94
- RELEASE_CURSOR オプション
 - ORACLE プリコンパイラ, F-18
- 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
 - 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, 3-20
 - COMMIT 文中の, 3-16
 - ROLLBACK 文中の, 3-19
 - 省略された場合, 3-21
 - 制限, 3-19
 - 用途, 3-16
- ROLLBACK SQL 文, F-95
- ROLLBACK 文, 3-20
 - PL/SQL ブロック内での使用, 3-25
 - RELEASE オプションを含む, 3-19
 - TO SAVEPOINT 句を含む, 3-18
 - 影響, 3-18
 - エラー処理ルーチン内の, 3-19
 - トランザクションを終了, F-97
 - 配置する場所, 3-19
 - 用途, 3-18
 - 例, 3-19, F-97
- ROWID
 - 擬似列, 3-23, 4-35
 - CURRENT OF のかわりとして, 3-23, 8-25
 - ユニバーサル, 4-7, 4-35
 - 論理値, 4-7, 4-35
- ROWID データ型, 4-7

S

- SAVEPOINT SQL 文, F-98
- SAVEPOINT 文

- 用途, 3-17
- 例, 3-17, F-99
- 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
- SET DESCRIPTOR SQL 文, 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 文中の, 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
- SQL92, xxxiv
- sqlald() 関数
 - 構文, 15-5
 - 使用例, 15-20
 - 用途, 15-5
- sqlalddt() 関数
 - 「SQLSQLDAAlloc」を参照, 5-49
- SQLCA, 9-2, 9-14
 - PL/SQL ブロック用コンポーネント・セット, 9-22
 - SQLCABC コンポーネント, 9-19
 - SQLCAID コンポーネント, 9-19
 - sqlcode コンポーネント, 9-19
 - sqlerrd, 9-20
 - sqlerrmc コンポーネント, 9-20
 - sqlerrml コンポーネント, 9-20
 - SQL*Net を使用している場合, 9-16
 - sqlwarn, 9-22
 - 概要, 2-9
 - コンポーネント, 9-19
 - 説明, 9-16
 - 宣言, 9-16
 - 複数回の組込み, 5-32
 - 複数使用, 9-16
 - 分割プリコンパイルでの使用, 2-17
 - 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, 2-17
 - リスト, 9-17
- SQLCAID コンポーネント, 9-19
- SQLCDAFromResultSetCursor(), 5-49
- SQLCDAGetCurrent, 5-50
- sqlcdat()
 - 「SQLCDAFromResultSetCursor()」を参照, 5-49
- 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, 15-35
 - 使用例, 15-35
 - 用途, 15-35
- sqlclut() 関数
 - 「SQLSQLDAFree()」を参照, 5-49
- SQLCODE
 - MODE=ANSI を設定, 10-30
- sqlcode
 - SQLCA の構成要素, 9-14
 - SQLCA のコンポーネント, 9-3
 - 値の解釈, 9-19
- SQLCODE 状態変数
 - SQLCA とともに宣言, 9-14
 - 使用する場合, 9-14

- 宣言, 9-14
- sqlcpr.h, 9-23
- SQL_CURSOR, F-12
- sqlcurt() 関数
 - 「SQLDAToResultSetCursor()」を参照, 5-49
- SQLDA
 - C 変数, 15-10
 - F 変数, 15-9
 - I 変数, 15-8
 - L 変数, 15-7
 - M 変数, 15-9
 - 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
 - 構造体, 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(), 5-49
- 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, 15-8
 - 用途, 15-8
- 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, 15-9
 - 用途, 15-9
- 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(), 5-50
- sqlerrd
 - コンポーネント, 9-15, 9-20
- sqlerrd[2] コンポーネント, 9-20
 - 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, 9-20
- sqlerrml コンポーネント, 9-20
- SQLERROR
 - WHENEVER ディレクティブ条件, F-115
- SQLErrorGetText(), 5-49
- SQLERROR 条件
 - WHENEVER 文, 9-24
 - 意味, 9-24
- SQLExtProcError(), 5-50, 7-30
- sqlglm(), 9-23
- sqlglm() 関数, 9-22
 - 使用例, 9-23
 - パラメータ, 9-22
- sqlglmt()
 - 「SQLErrorGetText」を参照, 5-49
- sqlgls() 関数, 9-31
 - 「SQLLIB」を参照
 - SQLStmGetText 関数, 4-20
 - サンプル・プログラム, 9-34
 - 使用例, 4-20
- sqlglst() 関数

- 「SQLStmtGetText」を参照, 5-49
- 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, 5-49
 - SQLDAToResultSetCursor 関数, 5-49
 - SQLEnvGet 関数, 5-44, 5-50
 - SQLExceptionGetText 関数, 5-49
 - SQLExtProcError 関数, 5-50, 7-30
 - SQLLDAGetName 関数, 5-50
 - SQLNumberPrecV6 関数, 5-50
 - SQLNumberPrecV7 関数, 5-50
 - SQLRowidGet 関数, 5-50
 - SQLStmtGetText() 関数, 5-49
 - SQLSvcCtxGet 関数, 5-45, 5-50
 - SQLVarcharGetLength 関数, 4-20
 - 埋込み SQL, 2-5
 - 関数
 - SQLCDAFromResultSetCursor, 5-49
 - 関数の新規名, A-3
 - パブリック関数の新規ファイル名, 5-48
- SQLLIB 関数
 - SQLSQLDAAlloc, 5-49
 - SQLVarcharGetLength, 5-50
- SQLLIB での SQLEnvGet 関数, 5-44
- SQLLIB での SQLSvcCtxGet 関数, 5-45
- SQL*Net
 - バージョン 2 を使用して接続, 3-4
- sqlnul() 関数
 - T 変数を使った使用方法, 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, 5-50
- SQL*Plus, 1-3
 - SELECT 文テストのための使用, 6-8
 - 対埋込み SQL, 1-3
- sqlpr2() 関数, 15-16
- sqlpr2t() 関数
 - 「SQLNumberPrecV7」を参照, 5-50
- sqlprc() 関数, 15-15
- sqlprct() 関数
 - 「SQLNumberPrecV6」を参照, 5-50
- SQLRowidGet(), 5-50
- SQL_SINGLE_RCTX
 - 定義, 5-44
 - 定義済みの定数, 5-49
- SQLSQLDAAlloc, 5-49
- SQLSQLDAFree(), 5-49
- SQLSTATE
 - MODE=ANSI を設定, 10-30
 - Oracle エラーへのマッピング, 9-13
 - 値, 9-4
 - クラス・コード, 9-4
 - 事前定義済みのクラス, 9-6
 - 使用, 9-13
 - 状態変数, 9-2, 9-3
 - ステータス・コード, 9-13
 - 宣言, 9-4
- SQLStmtGetText, 5-49
- SQLSvcCtxGet(), 5-50
- SQLVarcharGetLength, 5-50
- sqlvcp() 関数, 「SQLLIB」を参照。
 - SQLVarcharGetLength 関数, 4-20
- sqlvcpt() 関数
 - 「SQLVarcharGetLength」を参照, 5-50
- sqlwarn
 - フラグ, 9-22
- SQLWARNING
 - WHENEVER ディレクティブ条件, F-115
- SQLWARNING 条件
 - WHENEVER 文, 9-24
 - 意味, 9-24
- SQL 記述子領域
 - SQLDA, 13-24, 15-4
- SQL 通信領域, 9-2
 - SQLCA, 9-16
- SQL ディレクティブ
 - CONTEXT USE, 11-9, F-31
 - DECLARE DATABASE, F-36

DECLARE STATEMENT, F-37
DECLARE TABLE, F-38
DECLARE TYPE, F-40
TYPE, F-106
VAR, F-112
WHENEVER, F-115
SQL, 動的, 2-6
SQL 文
 ALLOCATE, F-12
 ALLOCATE DESCRIPTOR TYPE, F-14
 CACHE FREE ALL, F-15
 CALL, 7-26, F-16
 CLOSE, F-17
 COMMIT, F-23
 CONNECT, F-25
 CONTEXT ALLOCATE, F-27
 CONTEXT FREE, F-28
 CONTEXT OBJECT OPTION GET, F-29
 CONTEXT OBJECT OPTION SET, F-30
 DEALLOCATE DESCRIPTOR, F-33
 DELETE, F-41
 DESCRIBE, F-45
 DESCRIBE DESCRIPTOR, F-46
 ENABLE THREADS, F-48
 EXECUTE, F-50
 EXECUTE ...END-EXEC, F-49
 EXECUTE IMMEDIATE, F-54
 FETCH, F-56
 FETCH DESCRIPTOR, F-58
 FREE, F-61
 INSERT, F-65
 LOB APPEND, F-68
 LOB ASSIGN, F-68
 LOB CLOSE, F-69
 LOB COPY, F-70
 LOB CREATE, F-70
 LOB DESCRIBE, F-71
 LOB DISABLE BUFFERING, F-72
 LOB ENABLE BUFFERING, F-72
 LOB ERASE, F-73
 LOB FILE CLOSE, F-74
 LOB FILE SET, F-74
 LOB FLUSH BUFFER, F-75
 LOB FREE TEMPORARY, F-75
 LOB LOAD, F-76
 LOB OPEN, F-77
 LOB READ, F-77

LOB TRIM, F-78
LOB WRITE, F-78
OBJECT CREATE, F-79
OBJECT DELETE, F-81
OBJECT DEREf, F-82
OBJECT FLUSH, F-83
OBJECT GET, F-84
OBJECT RELEASE, F-85
OBJECT SET, F-86
OBJECT UPDATE, F-87
OPEN, F-88
OPEN DESCRIPTOR, F-90
ORACLE データ操作, 6-7
ORACLE データ問合せ用, 6-7
PREPARE, F-92
REGISTER CONNECT, F-94
ROLLBACK, F-95
SAVEPOINT, F-98
SELECT, F-100
SET DESCRIPTOR, F-103
UPDATE, F-108
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 ディレクティブの, F-115
 WHENEVER 文, 9-25
 結果, 9-25
STRING データ型, 4-6
struct
 REF の C 構造体の生成, 17-32
 ホスト変数, 4-37
 ホスト変数としてのポインタ, 4-44
SYS_INCLUDE
 C++ 内のシステム・ヘッダー・ファイル, 12-5
SYSDBA/SYSOPER 権限, A-5
SYS_INCLUDE プリコンパイラ・オプション, 10-37

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 文中の, 3-18

制限, 3-19

用途, 3-18

TO 句

ROLLBACK 文の, F-96

TYPE_CODE

プリコンパイラ・オプション, 10-39

TYPE SQL ディレクティブ, F-106

TYPE ディレクティブ

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, 1-10

UNSAFE_NULL プリコンパイラ・オプション, 10-39

UNSIGNED データ型, 4-9

UPDATE CASCADE, 9-21

UPDATE SQL 文, F-108

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 文中, 3-8

EXECUTE 文, 13-13

FETCH 文の, F-57

OPEN 文の, F-89

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 文中の, 6-8

INSERT 文の, F-67

埋込み SQL INSERT 文の, 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, 4-4, 5-14

VARCHAR 擬似型

PL/SQL で使用するための要件, 7-10

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

構造体, 4-17

参照による関数への引渡しが必要, 4-19

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, 4-17

VARNUM データ型, 4-6

VARRAW データ型, 4-8

VARRAY

作成, 18-3

VAR SQL ディレクティブ, F-112

VAR ディレクティブ

例, F-114

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, 1-10

W

WHENEVER SQL ディレクティブ, F-115

WHENEVER ディレクティブ

例, F-116

WHENEVER 文

CONTINUE アクション, 9-25

DO BREAK アクション, 9-25

DO CONTINUE アクション, 9-25

- DO アクション, 9-25
- GOTO アクション, 9-25
- NOT FOUND 条件, 9-24
- SQLCA の自動チェック, 9-24
- SQLERROR 条件, 9-24
- SQLWARNING 条件, 9-24
- STOP アクション, 9-25
- アドレス指定可能度の維持, 9-30
- ガイドライン, 9-29
- 概要, 2-9
- 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, 9-28
- 配置する場所, 9-29
- 無限ループの回避, 9-29
- 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 文中の, 6-10
 - DELETE 文の, F-43
 - SELECT 文中の, 6-7
 - UPDATE 文, 6-9
 - UPDATE 文の, 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 文の, F-24
 - ROLLBACK 文の, F-96

X

- XA インタフェース, 5-51
- 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, 5-51

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, 2-8

- フェッチ, 6-13
- 変更, 6-12, 6-13
- 未定義になった場合, 6-11
- アソシエイティブ・インタフェース, 17-4
 - 使用する場合, 17-4
- アプリケーション開発過程, 2-10
- 暗黙的な接続, 3-12
 - 単一, 3-12
 - 複数の, 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-25
- 以前のリリースからの移行, A-5
- 一時 LOB の作成, 16-14
- 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 オプションを使った制御, D-2
 - 使用可能, D-3
 - 定義, D-2
- インダウト・トランザクション, 3-24
- インタフェース
 - XA, 5-51
 - ネイティブ, 5-51

う

- 埋込み
 - プリコンパイラ・プログラム中の PL/SQL ブロック, F-49
- 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, 2-6
 - VARCHAR 擬似型, 7-10
 - カーソル FOR ループ, 7-2
 - 概要, 2-6
 - 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-17

CONTEXT ALLOCATE 文, 11-9, F-27

CONTEXT FREE 文, 11-10

ENABLE THREADS 文, 11-8

EXEC SQL CACHE FREE ALL, 17-6

EXECUTE 文, F-49

OCI 型の使用, 17-33

OPEN 文, F-88

PREPARE 文, F-92

REF の使用, 17-32

SAVEPOINT 文, F-98

SELECT 文, F-100

SQL*Plus を使ったテスト, 1-3

TYPE ディレクティブ, F-106

UPDATE 文, F-108

VAR ディレクティブ, F-112

WHENEVER ディレクティブ, F-115

概要, 2-2

構文, 2-5

主要概念, 2-2

使用する場合, 1-3

対話型 SQL との差異, 2-5

定義, 2-2

ホスト言語との混在, 2-5

要件, 2-5

埋込み SQL での REF の使用, 17-32

埋込み SQL 文

アポストロフィの使用, 2-12

引用符の使用, 2-12

終了記号, 2-15

接尾辞および接頭辞は許されない, 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, 9-25

え

永続オブジェクト, 17-4

エラー検出

エラー報告, F-116

エラー処理, 2-9

ROLLBACK 文の用途, 3-19

SQLCA 文と WHENEVER 文の対比, 9-3

概要, 2-9

代替手段, 9-2

必要, 9-2

エラー報告

WHENEVER ディレクティブ, F-116

エラー・メッセージの使用, 9-16

解析エラー・オフセットの使用, 9-15

警告フラグの使用, 9-15

主要コンポーネント, 9-14

処理済み行数の使用, 9-15

エラー・メッセージ

SQLCA 内の格納場所, 9-16

エラー報告での使用, 9-16

最大長, 9-23

取得用の sqlglm() 関数使用, 9-22

エンキュー

ロック, 3-14

演算子

C と SQL の対比, 2-14

制限, 2-14

お

オーバーヘッド

削減, C-2

オープン

カーソル, F-88, F-90

カーソル変数, 4-25

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, 19-7
- Outtype ファイル, 19-15
- Pro*C/C++ での使用, 19-21
- コマンド行, 19-6
- コマンド行構文, 19-24
- 参照, 19-23
- 使用, 19-1, 19-2
- 制限, 19-36
- データ型マップ, 19-9
- 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, 17-32
 - 使用, 17-32
 - 宣言, 17-32
- オプションの入力, 5-29, 10-9
- オブティマイザ・ヒント, C-5
 - C, 6-15
 - C++, 6-15
 - C++ での, 12-4

か

- カーソル, 2-17, 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-56, F-58
 - オープン, F-88, F-90
 - カーソル変数の割当て, 4-25
 - 型, 2-8
 - クローズ, F-17
 - 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, 2-8
 - 適用範囲, 6-12
 - 問合せとの関連付け, 6-11
 - 複数使用, 6-12
 - 明示的と暗黙的の対比, 2-8

- 命名規則, 6-12
- 用途, 6-11
- 類似性, 2-8
- 割当て, F-12
- カーソル・キャッシュ
 - 定義, 9-37
 - 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, 4-29
 - 宣言, 4-24
 - 割当て, 4-25
- 解除
 - スレッド・コンテキスト, 11-10
- 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, 9-15
 - 解釈方法, 9-15
- 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, 2-17
 - 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, 2-7
- 外部プロシージャ, A-4
 - PL/SQL からのコール, 7-28
 - エラー処理, 7-30
 - コールバック, 7-28
 - 制限, 7-29
 - 生成, 7-29
- 解除
 - スレッド・コンテキスト, F-28
- 拡張子
 - デフォルト・ファイル名, 19-35
- 各国語サポート (NLS), 4-45
- 可変長配列, 18-3
- 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
カーソル
 複数行問合せ用, 6-11

き

記号
 定義, 2-16
記号の定義, 2-16
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, 15-35
キャッシュ, 17-3
行
 カーソル以降のフェッチ, F-56, F-58
 継続, 2-14
 更新, F-108
 最大長, 2-14
 表およびビューに挿入, F-65
行のロック
 FOR UPDATE OF を用いた, 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, 9-19
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-17

け

警告フラグ
 エラー報告での使用, 9-15
結合
 制限, 6-17
現在の行
 検索用の FETCH 使用, 6-11
 定義, 2-8
検索条件
 WHERE 句中の, 6-10
 定義, 6-10

こ

更新
 表とビューの行, F-108
構成ファイル, 10-5
 位置, 10-6
 およびオブジェクト型トランスレータ, 19-5
 システム, 10-3
 ユーザー, 10-3
構造体
 コレクション型, 18-3
 C, 使用, 17-31
 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
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, 2-5
構文検査
 SQLCHECK オプションを使った制御, 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
 読みかた, 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, 2-11
- コード体系 (文字セットまたはコード・ページ), 4-46
- コード・ページ, 4-46
- コミット
 - 機能, 3-15
 - 自動, 3-15
 - トランザクション, F-23
 - 明示的と暗黙的の対比, 3-15
- コメント
 - ANSI, 2-11
 - PL/SQL ブロックの制限, 13-29
 - 許可, 2-11
- コレクション, A-4
 - 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
 - ネストした表, 18-2
 - 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, 2-18
 - 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
- 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-98

- 左辺値, 4-10
- 参照
 - ホスト配列, 8-2, 8-3
- 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, 2-18
 - EMP 表, 2-18
- サンプル・プログラム
 - ansidyn1.pc, 14-27
 - ansidyn2.pc, 14-35
 - calldemo.sql と sample9.pc, 7-22
 - coldemo1.pc, 18-22
 - cppdemo1.pc, 12-6
 - cppdemo2.pc, 12-9
 - cppdemo3.pc, 12-13
 - cv_demo.pc, 4-31
 - cv_demo.sql, 4-30
 - extp1.pc, 7-30
 - lobdemo1.pc, 16-33
 - navdemo1.pc, 17-24
 - oraca.pc, 9-40
 - sample10.pc, 15-38
 - sample11.pc, 4-31
 - sample12.pc, 15-38
 - sample1.pc, 2-19
 - sample2.pc, 4-40
 - sample3.pc, 8-7
 - sample4.pc, 5-14
 - sample5.pc, 20-10
 - sample6.pc, 13-9
 - sample7.pc, 13-14
 - sample8.pc, 13-20
 - sample9.pc, 7-22
 - sqlvcp.pc, 4-20
 - カーソル変数デモ, 4-30
 - プリコンパイルする方法, 2-19

し

- 識別子, ORACLE
 - 形成の方法, F-11
- システム・グローバル領域 (SGA), 7-19
- システム構成ファイル, 10-3, E-3
- システム固有の Oracle ドキュメント, xxx, 1-10,
2-18, 3-6, 5-29, 5-53
- システム固有の Oracle マニュアル, 20-1
- システム固有の参照, 4-6, 10-2, 10-6, 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
- 事前定義済み記号, 2-16
- 実行可能な SQL 文
 - グループ化, 2-5
 - 許される場所, 2-3
 - 用途, 2-3, 6-6
- 実行計画, C-4, C-6
- 実行時コンテキスト
 - 確立, 5-43
 - 終了, 5-43
- 実行時タイプ・チェック, 17-21
- 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, 3-20
- 出力ホスト変数
 - 値の割当て, 6-2
 - 定義, 6-2
- 準拠
 - ANSI, xxxv
 - ISO, xxxv
 - NIST, xxxv
- 条件付きプリコンパイル, 2-15
 - 記号の定義, 5-40
 - 例, 2-16, 5-40
- 状態変数, 9-2
- 使用方法
 - スレッド・コンテキスト, 11-9, F-31
- 省略記号, xxxiv
- 初期化関数
 - コール, 19-19
 - 作業, 19-21
- 処理済み行数
 - SQLCA, 9-21
 - エラー報告での使用, 9-15

す

- 垂直バー, xxxiv
- 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-113

- 定義, 15-15, F-113
- 負の場合, 15-15, F-113
- ステータス・コード
 - 意味, 9-14
- ストアド・サブプログラム
 - コール, 7-22
 - ストアドとインラインの対比, 7-19
 - 生成, 7-20
 - 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, 3-14
- スレッド, F-27
 - 解除コンテキスト, 11-10
 - コンテキストの解放, F-28
 - コンテキストの使用, 11-9
 - コンテキストの割当て, 11-9, F-27
 - 使用可能, F-48
 - 有効化, 11-8

せ

- 制限
 - AT 句, 3-10
 - CURRENT OF 句の使用, 8-4
 - CURRENT OF 句に対する, 6-16
 - 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, 2-17
 - ホスト配列, 8-4, 8-9, 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-98
 - 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-25
 - 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, 9-16
- カーソルの, 6-11
- ポインタ変数, 4-43
- ホスト配列, 8-2

宣言 SQL 文

- トランザクション中の, 3-15
- 許される場所, 2-3
- 用途, 2-3

宣言節

- MODE=ANSI の場合, 5-14
- MODE=ANSI の場合に必須, 10-30
- 使用可能な文, 2-12
- 定義規則, 2-12
- 必要な場合, 2-11, 4-11
- フォーム, 2-11
- 要件, 2-11
- 用途, 2-11

宣言文, 2-3

全体走査

- 説明, C-6

選択記述子, 13-24, 15-4

- 情報, 13-25
- 定義, 13-24

選択リスト

- free() 関数の使用, 15-35
- malloc() 関数を使用, 15-30
- 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, xxxiv

ダミー・ホスト変数

ブレースホルダ, 13-3

端末

コード体系, 4-46

ち

中カッコ, xxxiv

調整, 効率, C-2

て

データ型

- NUMBER を VARCHAR2 に強制変換, 15-14
- Oracle, 2-7
- ORACLE 内部データ型の取扱い, 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, 17-37
- 内部, 4-2
- 内部データ型と外部データ型の対比, 2-7
- 内部データ型のリスト, 15-11
- ユーザー定義タイプ同値化, F-106

データ型コード

- 記述子内で使用, 15-14

データ型同値化, 5-11, 2-8

- 用途, 2-8

データ型の同値化

- データ型同値化, 2-8

データ型変換, 5-11

データ型マップ, 19-9

データ定義文

- トランザクション中の, 3-15

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 文の使用法, F-66
- 格納場所, 3-13
- シノニムの作成, 3-13
- 使用する例, 3-13
- 定義, 3-12

データ・ロック, 3-14

適用範囲

- DECLARE STATEMENT ディレクティブの, F-38

- EXEC ORACLE 文, 10-10
- WHENEVER 文, 9-28
- カーソル変数, 4-24
- プリコンパイラ・オプション, 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, 2-12

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

同時接続, 3-6

同値化

- ホスト変数の同値化, F-112
- ユーザー定義タイプ同値化, F-106

動的 PL/SQL

- 規則, 13-28
- 動的 SQL との対比, 13-28

動的 SQL

- PL/SQL の使用, 7-31
- カーソル変数の同時使用不可能, 4-29
- 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, 2-6
- データ型使用の制限, 17-37
- 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

- 一括操作, 14-8

- 概要, 14-3

- 基本, 14-2

- サンプル・プログラム, 14-27, 14-35

- プリコンパイラ・オプション, 14-2, 14-10

- リファレンス・セマンティクス, 14-7

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

- プ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, 13-4

- 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, 13-27

- 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, 13-27

- DECLARE の使用, 13-19

- FETCH の使用, 13-19

- OPEN の使用, 13-19

- PL/SQL の使用, 13-29

- 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, 13-18

- 要件, 13-5

動的 SQL 方法 4

CLOSE 文の使用, 15-35
DECLARE CURSOR 文の使用, 15-23
DECLARE STATEMENT の使用, 13-27
DESCRIBE の使用, 13-24
DESCRIBE 文の使用, 15-23, 15-28
FETCH 文の使用, 15-33
FOR 句の使用, 13-28, 15-35
OPEN 文の使用, 15-28
PL/SQL の使用, 13-29
PREPARE 文の使用, 15-23
SQLDA の使用, 13-24, 15-4
概要, 13-24
記述子の使用, 13-24
記述子の必要性, 15-4
使用する場合, 13-24
使用の前提条件, 15-10
使用文の順序, 13-26, 15-18
手順, 15-17
ホスト配列の使用, 15-35
要件, 13-5, 15-2
動的 SQL 方法 4 サンプル・プログラム, 15-38
動的文の解析
PREPARE 文, F-92
ドット, xxxiv
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-23
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, 3-16
終了方法, 3-15
セーブポイントによる副分割, 3-17
説明, 3-15
定義, 2-8
取消し, 3-18
内容, 2-8, 3-15
分散された, F-98
変更の確定, 3-16
読取り専用, 3-21
ロール・バック, F-95
を用いたデータベースの保護, 3-15
トランザクション処理
概要, 2-8
使用される文, 2-8
トランザクション処理モニター, 5-51

トランザクションの取消し, F-95
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, 2-7
ナビゲーション・アクセス・サンプル・プログラム,
17-24

に

入力ホスト変数
値の割当て, 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, 5-51
ネストした表, 18-2
作成, 18-2
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
現行, 3-7
定義, 3-5

は

バイトの並び, E-2
配列
一括操作 (ANSI 動的 SQL), 14-8
可変長, 18-3
使用方法を説明する章, 8-1
操作, 2-7
定義, 4-38

- バッチ・フェッチ, 8-5
- ホスト配列, 2-7
- バインド
 - 定義, 13-4
- バインド SQLDA
 - 用途, 15-3
- バインド記述子, 13-24, 15-4
 - 情報, 13-25
 - 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

ひ

- ヒープ
 - 定義, 9-37
- ビュー
 - 行の更新, F-108
 - 行の挿入, F-65
- 表
 - 行の更新, F-108
 - 行の挿入, F-65
 - ネストした, 18-2
- 表から行の取出し
 - 埋込み SQL, F-100
- 表記規則
 - 説明, xxxiii
 - 表記法, xxxiii
- 表記法
 - 規則, xxxiv
 - 表記規則, xxxiv
- 表記法, BNF, xxxiv
- 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, 7-11
- 値の割当て, 6-3
- ガイドライン, 4-16
- 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, 4-39
- 参照, 4-15
- 宣言, 4-15, 18-4
- 定義, 2-7
- の値の解釈, 6-3
- 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, 4-49
- 命名, 4-40
- 要件, 6-4
- 標準ヘッダー・ファイル, E-2
- 表のロック
 - LOCK TABLE を用いた, 3-23
 - 影響, 3-23
 - 解除される場合, 3-23
 - 行の共有, 3-23
- ヒント
 - COST, C-5
 - DELETE 文中の, F-44
 - ORACLE SQL 文オプティマイザ用, 6-15
 - SELECT 文中の, F-103
 - UPDATE 文, F-111

ふ

- フェッチ
 - カーソル以降の行, F-56, F-58
- 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, 2-8
 - カーソルとの関連付け, 2-8
 - 用途, C-9
 - 定義, 2-8
- フラグ
 - 警告フラグ, 9-15

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, 10-13
CONFIG, 10-14
CPP_SUFFIX, 10-15
DBMS, 10-15
DEFINE, 10-17
DEF_SQLCODE, 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, 2-14, 10-28
MAXOPENCURSORS, 10-28
MODE, 10-29, 14-10
NLS_CHAR, 10-30
NLS_LOCAL, 10-31
OBJECTS, 10-31
ONAME, 10-32
ORACA, 10-33
PAGELEN, 10-33
PARSE, 10-33
PREFETCH, 10-34
RELEASE_CURSOR, 10-35
SELECT_ERROR, 10-36
SQLCHECK, 10-37, 17-21
SYS_INCLUDE, 10-37
THREADS, 10-38, 11-8
TYPE_CODE, 14-10, 10-39
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, 10-7, 10-11
大文字と小文字の区別, 10-2

現在の設定値を調べる, 10-4

構成ファイル, 10-5

構文, 10-9

コマンド行へ入力, 10-9

指定, 10-9

使用, 10-11 ~ 10-41

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, 10-3

リスト, 10-11

プリコンパイラ・オプションの現在の設定値を調べる,
10-4

プリコンパイラ・オプションの優先順位, 10-3

プリコンパイル, 10-6

条件付き, 2-15

分割, 2-17

プリコンパイル済みのヘッダー・ファイル, 5-34

CODE オプション, 5-38

C++ 制限, 5-38

PARSE オプション, 5-38

プリコンパイル済みヘッダー・ファイル, A-2

プリコンパイル・ユニット, 3-2, 10-9

C プリプロセッサ

Pro*C でサポートされるディレクティブ, 5-27

プリプロセッサ

EXEC ORACLE ディレクティブ, 5-40

例, 5-40

プリプロセッサ, サポート, 4-2

プリプロセッサ・ディレクティブ

Pro*C でサポートされないディレクティブ, 5-28

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, 2-11

プログラムの終了

通常と異常の対比, 3-20

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, 2-17

カーソルの参照, 2-17

ガイドライン, 2-17

制限, 2-17

単一 SQLCA の使用, 2-17

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-98
- 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

- 平行性
 - 定義, 3-14
- 変数, 2-6
 - カーソル, 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, 4-43
 - 参照, 4-43
 - 参照値のサイズ決定, 4-43
 - 宣言, 4-43
- ホスト言語
 - 定義, 2-2, 2-3
- ホスト構造体
 - 宣言, 4-37
 - 配列, 4-38
- 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 文中の, 8-11
 - WHERE 句中の, 8-15
 - 効率改善のために使用, C-3
 - サイズの一致, 8-3
 - 最大サイズ, 8-2
 - 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, 8-9, 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, 15-35
- 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-51
 - OPEN 文中の, F-89
 - PL/SQL での使用, 7-6
 - アドレスへ設定されなければならない, 4-15
 - 概要, 2-6
 - 制限, 4-14
 - 宣言, 2-11, 18-4
 - ダミー, 13-3
 - 定義, 2-6
 - に値を割当てる, 2-6
 - 入力と出力の対比, 6-2
 - ホスト変数の同値化, F-112
 - 命名規則, 2-14
 - ユーザー・イグジット, 20-4
 - 許される場所, 2-6
 - 要件, 2-6
 - 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
- 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 文とディレクティブ, 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
 - 複数の, 3-11

命名

- SQL*Forms ユーザー・イグジット, 20-13
- カーソルの, 6-12
- 選択リスト項目, 15-4
- データベース・オブジェクト, F-11
- 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, 4-48
- 文字列ホスト変数
 - 宣言, 5-7
- 戻り句, 6-10
 - DELETE, 6-10
 - INSERT 文中の, 6-10
 - UPDATE 文中の, 6-9

や

- 山カッコ, xxxiv

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, 20-12
 - 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・オプションを設定する, 10-3

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
- ユニバーサル ROWID, A-5

よ

- 読取り専用トランザクション
 - 終了方法, 3-21
 - 説明, 3-21
 - 例, 3-21
- 読取りの一貫性
 - 定義, 3-14
- 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, 9-25

り

- リソース・マネージャ, 5-51
- リターン・コード
 - ユーザー・イグジット, 20-8
- リファレンス・セマンティクス (ANSI 動的 SQL), 14-7
- リモート・データベース
 - 宣言, F-36
- リンク, 2-18
 - 2 タスク, 2-18
 - UNIX, 1-10
 - VMS, 1-10
 - データベース・リンク, 3-12

れ

- 例外, PL/SQL
 - 定義, 7-12
- レコード, 7-5

列リスト

INSERT 文中の, 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-97

セーブポイントへ, F-98

ロールバック・セグメント

機能, 3-14

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 文を用いた, 3-23

ROLLBACK 文により解除, F-97

取得用の権限, 3-25

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-27

