



EDB Postgres™ Advanced Server

リリース **13**

ECPGPlus Guide

2020 年 11 月 29 日

目次

第 1 章	はじめに	1
第 2 章	ECPGPlus-概要	3
2.1	インストールと構成	7
2.2	Makefile の構築	8
2.3	ECPGPlus コマンドラインオプション	10
第 3 章	埋め込み SQL の使用	13
3.1	例-簡単なクエリ	13
3.1.1	インジケータ変数の使用	17
3.1.2	ホスト変数の宣言	18
3.2	例-カーソルを使用して結果セットを処理する	20
第 4 章	記述子の使用	25
4.1	例-記述子を使用してデータを返す	28
第 5 章	動的 SQL ステートメントの作成と実行	39
5.1	例-パラメータなしの非クエリステートメントの実行	40
5.2	例-指定された数のプレースホルダーを使用した非クエリステートメントの実行	43
5.3	例-既知の数のプレースホルダーでクエリを実行する	46
5.4	例-未知の数の変数を使用したクエリの実行	49
第 6 章	エラー処理	59
6.1	sqlca でのエラー処理	59
6.2	EXECSQLWHENEVER	66
第 7 章	参照先	69
7.1	C プリプロセッサディレクティブ	69
7.2	サポートされている C データ型	73
7.3	タイプコード	75
7.4	SQLDA 構造	76
7.5	ECPGPlus ステートメント	80
7.5.1	記述子の割り当て	80
7.5.2	電話する	81

7.5.3	閉じる	81
7.5.4	コミット	82
7.5.5	接続する	83
7.5.6	記述子の割り当て解除	85
7.5.7	宣言カーソル	85
7.5.8	データベースの宣言	86
7.5.9	宣言文	86
7.5.10	削除	87
7.5.11	DESCRIBE	88
7.5.12	DESCRIBEDESCRIPTOR	89
7.5.13	切断	90
7.5.14	実行	91
7.5.15	記述子の実行	92
7.5.16	EXECUTE...ENDEXEC	92
7.5.17	即時実行	93
7.5.18	FETCH	94
7.5.19	FETCHDESCRIPTOR	94
7.5.20	記述子を取得する	95
7.5.21	挿入	97
7.5.22	開く	98
7.5.23	記述子を開く	99
7.5.24	準備する	99
7.5.25	ロールバック	100
7.5.26	セーブポイント	101
7.5.27	選択	101
7.5.28	接続の設定	103
7.5.29	記述子の設定	104
7.5.30	更新	106
7.5.31	いつでも	107
第 8 章	Conclusion	109
索引		111

第 1 章

はじめに

EDB は ECPG (PostgreSQL プリコンパイラ) を強化して ECPGPlus を作成しました。ECPGPlus を使用すると、EDBPostgresAdvancedServer (AdvancedServer) データベースに接続したときに、C アプリケーションに Pro*C 互換の埋め込み SQL コマンドを含めることができます。ECPGPlus を使用してアプリケーションをコンパイルすると、SQL コードは構文チェックされ、C に変換されます。

ECPGPlus は以下をサポートします:

- OracleDynamicSQL-方法 4 (ODS-M4)。
- Pro*C 互換の匿名ブロック。
- Oracle データベースと互換性のある CALL ステートメント。

ECPGPlus の Pro*C 互換性の一部として、BEGIN DECLARE SECTION および END DECLARE SECTION ディレクティブを含める必要はありません。

PostgreSQL Compatibility

ほとんどの ECPGPlus ステートメントはコミュニティ PostgreSQL で機能しますが、CALL ステートメントおよび EXECUTE...END EXEC ステートメントは、クライアントアプリケーションが EDBPostgresAdvancedServer に接続されている場合にのみ機能します。

第 2 章

ECPGPlus-概要

EDB は ECPG (PostgreSQL プリコンパイラ) を強化して ECPGPlus を作成しました。ECPGPlus は、PostgreSQLC プリコンパイラの Pro*C 互換バージョンです。ECPGPlus は、C コードと埋め込み SQL ステートメントを組み合わせたプログラムを同等の C プログラムに変換します。変換を実行する際、ECPGPlus は各 SQL コンストラクトの構文が正しいことを確認します。

次の図は、実行可能ファイルにコンパイルされる埋め込み SQL ステートメントを含むプログラムのパスを示しています。

埋め込み SQL ステートメントを含む C プログラムから実行可能ファイルを生成するには、プログラム（上図の `my_program.pgc`）を ECPGPlus プリコンパイラに渡します。ECPGPlus は、`my_program.pgc` の各 SQL ステートメントを `ecpglib` API を呼び出す C コードに変換し、C プログラム（`my_program.c`）を生成します。次に、C プログラムを C コンパイラに渡します。C コンパイラはオブジェクトファイル（`my_program.o`）を生成します。最後に、オブジェクトファイル（`my_program.o`）、`ecpglib` ライブラリファイル、およびその他の必要なライブラリをリンカーに渡し、リンカーが実行可能ファイル（`my_program`）を生成します。

ECPGPlus プリプロセッサは各 SQL ステートメントの*構文*を検証しますが、セマンティクスを検証することはできません。たとえば、プリプロセッサは INSERT ステートメントが構文的に正しいことを確認しますが、INSERT ステートメントに記載されているテーブルが実際に存在することは確認できません。

Behind the Scenes

クライアントアプリケーションには、次の要素で構成される C コードと SQL コードが混在しています。

- C プリプロセッサディレクティブ
- C 宣言（変数、型、関数、...）
- C 定義（変数、型、関数、...）
- SQL プリプロセッサディレクティブ
- SQL 文

例:

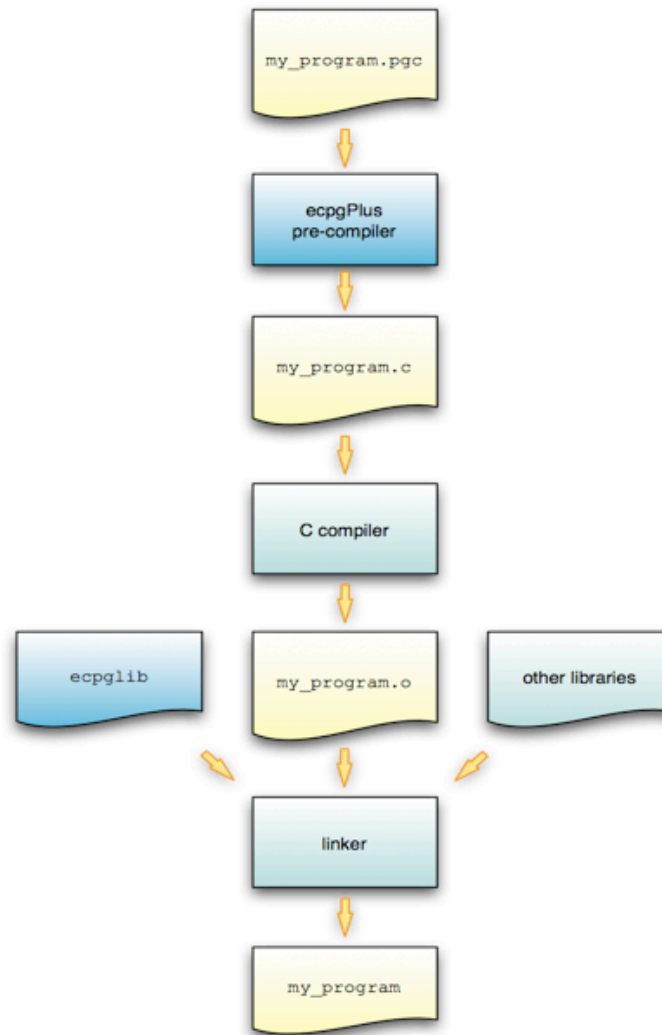


図 1 埋め込み SQL ステートメントを含むプログラムのコンパイル

```
1 #include <stdio.h>
2 EXEC SQL INCLUDE sqlca;
3
4 extern void printInt(char *label, int val);
5 extern void printStr(char *label, char *val);
6 extern void printFloat(char *label, float val);
7
8 void displayCustomer(int custNumber)
9 {
10  EXEC SQL BEGIN DECLARE SECTION;
11    VARCHAR custName[50];
12    float custBalance;
13    int custID = custNumber;
14  EXEC SQL END DECLARE SECTION;
```

(次のページに続く)

(前のページからの続き)

```

15
16 EXEC SQL SELECT name, balance
17     INTO :custName, :custBalance
18     FROM customer
19     WHERE id = :custID;
20
21 printInt("ID", custID);
22 printStr("Name", custName);
23 printFloat("Balance", custBalance);
24 }

```

上記のコードフラグメント:

- 1 行目は、C プリプロセッサへのディレクティブを指定しています。

C プリプロセッサディレクティブは解釈または無視されます。このオプションは、ECPGPlus を起動するときに入力されるコマンドラインオプション（-C PROC）によって制御されます。いずれの場合でも、ECPGPlus は各 C プリプロセッサディレクティブを変更せずに出力ファイル（4）にコピーします。ソースファイルで見つかった C プリプロセッサディレクティブは、出力ファイルに表示されます。

- 2 行目は、SQL プリプロセッサへのディレクティブを指定しています。

SQL プリプロセッサディレクティブは ECPGPlus プリプロセッサによって解釈され、出力ファイルにコピーされません。

- 行 4～6 には C 宣言が含まれています。

C 宣言は、各 VARCHAR 宣言が同等の struct 宣言に変換されることを除いて、変更せずに出力ファイルにコピーされます。

- 行 10～14 には、埋め込み SQL 宣言セクションが含まれています。

SQL コード内で参照する C 変数は host variables と呼ばれます。Pro*C モード（-C PROC）で ECPGPlus プリプロセッサを呼び出す場合、SQL ステートメント内で*any* C 変数を参照できます。それ以外の場合は、BEGIN/END DECLARATION SECTION ペア内の各ホスト変数を宣言する必要があります。

- 行 16～19 には SQL ステートメントが含まれています。

SQL ステートメントは、ECPGPlus ランタイムライブラリの呼び出しに変換されます。

- 行 21～23 には C コードが含まれています。

C コードは変更せずに出力ファイルにコピーされます。

SQL ステートメントには EXEC SQL をプレフィックスとして付け、次の（引用符なし）セミコロンまで拡張する必要があります。例:

```
printf( "Updating employee salaries\n" );

EXEC SQL UPDATE emp SET sal = sal * 1.25;
EXEC SQL COMMIT;

printf( "Employee salaries updated\n" );
```

プリプロセッサは上記のコードフラグメントを検出すると、C コード（最初の行と最後の行）を変換せずに出力ファイルに渡し、各 EXEC SQL ステートメントを `ecpglib` 関数の呼び出しに変換します。結果は次のようになります。

```
printf("Updating employee salaries\n");

{
    ECPGdo( __LINE__, 0, 1, NULL, 0, ECPGst_normal,
           "update emp set sal = sal * 1.25",
           ECPGt_EOIT, ECPGt_EORT);
}

{
    ECPGtrans(__LINE__, NULL, "commit");
}

printf( "Employee salaries updated\n" );
```

2.1 インストールと構成

Windows では、ECPGPlus は AdvancedServer インストールウィザードによって Database Server コンポーネントの一部としてインストールされます。Linux では、`xx` が AdvancedServer のバージョン番号である `edb-asxx-server-devel RPM` パッケージでインストールします。デフォルトでは、実行可能ファイルは次の場所にあります。

Windows の場合:

```
C:\Program Files\edb\as13\bin
```

Linux の場合:

```
/usr/edb/as13/bin
```

ECPGPlus コンパイラーを呼び出すとき、実行可能ファイルは検索パス（Windows では `%PATH%`、Linux では `$PATH`）になければなりません。たとえば、次のコマンドは、ECPGPlus 実行可能ファイル `ecpg` を保持するディレクトリを含むように検索パスを設定します。

Windows の場合:

```
set EDB_PATH=C:\Program Files\edb\as13\bin
set PATH=%EDB_PATH%;%PATH%
```

Linux の場合:

```
export EDB_PATH=/usr/edb/as13/bin
export PATH=$EDB_PATH:$PATH
```

2.2 Makefile の構築

makefile には、「埋め込み SQL を含む」C で記述されたプログラムを C プログラムに変換する方法を make ユーティリティに指示する一連の命令が含まれています。このガイドの例を試すには、次のものがが必要です。

- C コンパイラー（およびリンカー）
- make ユーティリティ
- ECPGPlus プリプロセッサとライブラリ
- ECPGPlus の指示を含む makefile

次のコードは、このガイドに含まれるサンプルの makefile の例です。サンプルコードを使用するには、ソースコードファイルを含むディレクトリの makefile という名前のファイルに保存します。

```
INCLUDES = -I$(shell pg_config --includedir)
LIBPATH = -L $(shell pg_config --libdir)
CFLAGS += $(INCLUDES) -g
LDFLAGS += -g
LDLIBS += $(LIBPATH) -lecpq -lpq

.SUFFIXES: .pgc, .pc

.pgc.c:
    ecpq -c $(INCLUDES) $?

.pc.c:
    ecpq -C PROC -c $(INCLUDES) $?
```

最初の 2 行は pg_config プログラムを使用して、必要なヘッダーファイルとライブラリディレクトリを見つけます。

```
INCLUDES = -I$(shell pg_config --includedir)
LIBPATH = -L $(shell pg_config --libdir)
```

pg_config プログラムは AdvancedServer に付属しています。

make は、C コンパイラを実行するときに CFLAGS 変数を使用し、リンカーを呼び出すときに LDFLAGS および LDLIBS 変数を使用する必要があることを知っています。ECPG プログラムは、ECPG ランタイムライブラリ（-lecpq）および libpq ライブラリ（-lpq）にリンクする必要があります。

```
CFLAGS += $(INCLUDES) -g
LDFLAGS += -g
LDLIBS += $(LIBPATH) -lecpq -lpq
```

サンプル makefile は、.pgc または .pc ファイルを C プログラムに変換する方法を make に指示します。makefile の 2 行は、ソースファイルがコンパイルされるモードを指定します。最初のコンパイルオプションは

次のとおりです。

最初のオプションは、コミュニティ ECPG (ECPGPlus 拡張なし) を使用して、.pgc で終わるファイル (おそらく ECPG ソースファイル) を .c (C プログラム) で終わるファイルに変換する方法を make に指示します。INCLUDES 変数の値と .pgc ファイルの名前を使用して、-c フラグ (SQL コードを C に変換するようコンパイラに指示します) で ECPG プリコンパイラを呼び出します。

```
.pc.c:
    ecpg -C PROC -c $(INCLUDES) $?
```

2 番目のオプションは、ECPGPlus 拡張機能を使用して、.pg で終わるファイル (ECPG ソースファイル) を .c で終わるファイル (C プログラム) に変換する方法を make に指示します。-c フラグ (SQL コードを C に変換するようにコンパイラに指示する) と -C PROC フラグ (Pro*C 互換モードで ECPGPlus を使用するようコンパイラに指示する) で ECPG プリコンパイラを呼び出します。INCLUDES 変数の値と .pgc ファイルの名前。

make を実行するとき、コンパイルしたい ECPG ソースコードファイルの名前を渡します。たとえば、customer_list.pgc という名前の ECPG ソースコードファイルをコンパイルするには、次のコマンドを使用します。

```
make customer_list
```

make ユーティリティは makefile (現在のディレクトリにある) を調べ、makefile に customer_list.pgc を C プログラムにコンパイルする規則 (customer_list.c) が含まれていることを検出し、次に組み込まれた規則を使用します customer_list.c を実行可能プログラムにコンパイルするには make。

2.3 ECPGPlus コマンドラインオプション

上記のサンプル makefile では、ECPGPlus を Pro*C 互換モードで起動するように指定するために ECPGPlus を起動するときに make に `-C` オプションが含まれています。

コマンドラインに ECPG 構文に加えて `-C PROC` キーワードを含める場合、Pro*C コマンドライン構文を使用できます。例えば:

```
$ ecpg -C PROC INCLUDE=/usr/edb/as13/include acct_update.c
```

利用可能な他の ECPGPlus オプションの完全なリストを表示するには、ECPGPlus インストールディレクトリに移動して、次のように入力します。

```
./ecpg --help
```

コマンドラインオプションは次のとおりです。

Option	Description
-c	埋め込み SQL コードから C コードを自動的に生成します。
-C*モード*	-C オプションを使用して、互換モードを指定します。 INFORMIX INFORMIX_SE PROC
-D*記号*	プリプロセッサ*シンボル* を定義します。 <i>PROC</i> モードでコンパイルする場合、-D*キーワードはサポートされません。代わりに、Oracle スタイルの* 'DEFINE=' * 句を使用してください。
-h	ヘッダーファイルを解析します。このオプションにはオプション*-c* が含まれます。
-i	システムを解析し、ファイルもインクルードします。
-I ディレクトリ	ディレクトリ でインクルードファイルを検索します。
-o*outfile*	結果を*outfile* に書き込みます。
-r*オプション*	実行時の動作を指定します。 <i>option</i> は次のいずれかです。 no_indicator -インジケータを使用せず、代わりに特別な値を使用して NULL 値を表します。 prepare -すべてのステートメントを使用する前に準備します。 questionmarks -疑問符をプレースホルダーとして使用できるようにします。 usebulk -ホスト変数配列を操作する INSERT、UPDATE、および DELETE ステートメントの一括処理を有効にします。
-回帰	回帰テストモードで実行します。
-t	トランザクションの autocommit をオンにします。
-l	#line ディレクティブを無効にします。
-ヘルプ	ヘルプオプションを表示します。
-バージョン	バージョン情報を出力します。

注釈: ECPGPlus を呼び出すときに出力ファイル名を指定しない場合、出力ファイル名は .pgc ファイル名拡張子を取り除き、.c をファイル名に追加することによって作成されます。

第 3 章

埋め込み SQL の使用

次の各セクションは、コードサンプルで始まり、コードサンプル内の各セクションの説明が続きます。

3.1 例-簡単なクエリ

最初のコードサンプルは、SELECT ステートメント（1 行を返す）を実行して、ホスト変数のグループに結果を保存する方法を示しています。ホスト変数を宣言した後、ハードコードされたロール名と関連付けられたパスワードを使用して edb サンプルデータベースに接続し、emp テーブルを照会します。クエリは、宣言されたホスト変数に値を返します。NULL インジケータ変数の値を確認した後、簡単な結果セットを画面に出力し、接続を閉じます。

```
/* *****  
 * print_emp.pgc  
 *  
 */  
#include <stdio.h>  
  
int main(void)  
{  
    EXEC SQL BEGIN DECLARE SECTION;  
        int v_empno;  
        char v_ename[40];  
        double v_sal;  
        double v_comm;  
        short v_comm_ind;  
    EXEC SQL END DECLARE SECTION;  
  
    EXEC SQL WHENEVER SQLERROR sqlprint;  
  
    EXEC SQL CONNECT TO edb  
        USER 'alice' IDENTIFIED BY 'lsafepwd';
```

(次のページに続く)

(前のページからの続き)

```

EXEC SQL
  SELECT
    empno, ename, sal, comm
  INTO
    :v_empno, :v_ename, :v_sal, :v_comm INDICATOR:v_comm_ind
  FROM
    emp
  WHERE
    empno = 7369;

if (v_comm_ind)
  printf("empno(%d), ename(%s), sal(%.2f) comm(NULL)\n",
        v_empno, v_ename, v_sal);
else
  printf("empno(%d), ename(%s), sal(%.2f) comm(%.2f)\n",
        v_empno, v_ename, v_sal, v_comm);
EXEC SQL DISCONNECT;
}
/*****

```

コードサンプルは、C の `stdio` ライブラリのプロトタイプと型定義を含めることから始まり、`main` 関数を宣言します。

```

#include <stdio.h>

int main(void)
{

```

次に、アプリケーションは、データベースサーバーとの対話に使用されるホスト変数のセットを宣言します。

```

EXEC SQL BEGIN DECLARE SECTION;
  int v_empno;
  char v_ename[40];
  double v_sal;
  double v_comm;
  short v_comm_ind;
EXEC SQL END DECLARE SECTION;

```

PROC モードでコードをプリコンパイルする予定がある場合は、`BEGIN DECLARE...END DECLARE` セクションを省略できます。ホスト変数の宣言の詳細については、[ホスト変数の宣言](#) を参照してください。

宣言セクション内の各変数に関連付けられているデータ型は、C データ型です。サーバーとクライアントアプリケーションの間で渡されるデータは、互換性のあるデータ型を共有する必要があります。データ型の詳細については、[サポートされている C データ型](#) を参照してください。

次のステートメントは、エラーの処理方法をサーバーに指示します。

```
EXEC SQL WHENEVER SQLERROR sqlprint;
```

クライアントアプリケーションが SQL コードでエラーを検出すると、サーバーは `ecpglib` で提供される `sqlprint()` 関数を使用して、`stderr`（標準エラー）にエラーメッセージを出力します。次の EXEC SQL ステートメントは、AdvancedServer との接続を確立します。

```
EXEC SQL CONNECT TO edb
      USER 'alice' IDENTIFIED BY 'lsafepwd';
```

この例では、クライアントアプリケーションは `edb` データベースに接続し、`alice` という名前のロールと `lsafepwd` というパスワードを使用します。

次に、コードは `emp` テーブルに対してクエリを実行します。

```
EXEC SQL
      SELECT
        empno, ename, sal, comm
      INTO
        :v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind
      FROM
        emp
      WHERE
        empno = 7369;
```

クエリは、従業員番号 7369 に関する情報を返します。

SELECT ステートメントは「11」句を使用して、取得した値（`empno`、`ename`、`sal`、`comm` 列から）を `:v_empno`、`:v_ename`、`:v_sal`、`:v_comm` ホスト変数（および「10」ヌルインジケータ）。取得された最初の値は「11」句にリストされている最初の変数に割り当てられ、2 番目の値は 2 番目の変数に割り当てられます。

`comm` 列には従業員が獲得したコミッション値が含まれ、潜在的に NULL 値が含まれる場合があります。このステートメントには、INDICATOR キーワードと、ヌルインジケータを保持するホスト変数が含まれています。

コードは `null` インジケータをチェックし、適切な画面上の結果を表示します。

```
if (v_comm_ind)
    printf("empno(%d), ename(%s), sal(%.2f) comm(NULL)\n",
           v_empno, v_ename, v_sal);
else
    printf("empno(%d), ename(%s), sal(%.2f) comm(%.2f)\n",
           v_empno, v_ename, v_sal, v_comm);
```

ヌルインジケータが 0（つまり `false`）の場合、`comm` 列には意味のある値が含まれ、`printf` 関数は手数料を表示します。`null` インジケータにゼロ以外の値が含まれる場合、`comm` は NULL であり、`printf` は NULL の値を表示します。そのホスト変数に NULL をフェッチすると、ホスト変数（`null` インジケータ以外）には意味のある値が含まれないことに注意してください。`null` インジケータを使用して、NULL の値を識別する必要があります。

コードサンプルの最後のステートメントは、サーバーへの接続を閉じます。

```
EXEC SQL DISCONNECT;  
}
```

3.1.1 インジケータ変数の使用

前の例には、comm 列の値（サーバーから返されたとき）が NULL である行を識別する*インジケータ**変数*が含まれていました。インジケータ変数は、前の変数の内容が NULL または切り捨てられているかどうかを示す追加のホスト変数です。インジケータ変数は、行の内容が保存されるときに入力されます。インジケータ変数には、次の値が含まれる場合があります。

指標値	意味する
標識変数が 0 より小さい場合。	サーバーから返された値は NULL でした。
標識変数が 0 に等しい場合。	サーバーから返された値は NULL ではなく、切り捨てられませんでした。
標識変数が 0 より大きい場合。	サーバーから返された値は、ホスト変数に格納されたときに切り捨てられました。

INTO 句にインジケータ変数を含める場合、オプションの INDICATOR キーワードを含める必要はありません。

クエリが NULL 値を対応するホスト変数に決して返さないことが確実な場合は、インジケータ変数を省略できます。インジケータ変数を省略し、クエリが NULL 値を返す場合、ecpglib は実行時エラーを発生させます。

3.1.2 ホスト変数の宣言

SQL ステートメント内で値が表示される可能性のある任意の時点で、そのステートメント内で*ホスト変数*を使用できます。ホスト変数は、クライアントアプリケーションからサーバーにデータ値を渡し、サーバーからクライアントアプリケーションにデータを返すために使用できる C 変数です。ホスト変数には次のものがあります。

- 配列
- typedef
- ポインター
- struct
- 任意のスカラー C データ型

以下のコードフラグメントは、PROC モードおよび非 PROC モードでコンパイルされたコードでホスト変数を使用する方法を示しています。SQL ステートメントは、dept テーブルに行を追加し、変数 v_deptno、v_dname、v_loc によって返される値を deptno 列、dname 列、loc 列に挿入します。それぞれ。

PROC モードでコンパイルする場合、EXEC SQL BEGIN DECLARE SECTION および EXEC SQL END DECLARE SECTION ディレクティブを省略できます。PROC モードでは、C 関数パラメーターをホスト変数として使用できます。

```
void addDept(int v_deptno, char v_dname, char v_loc)
{
    EXEC SQL INSERT INTO dept VALUES( :v_deptno, :v_dname, :v_loc);
}
```

PROC モードでコンパイルしていない場合は、以下に示すように、埋め込み変数宣言を EXEC SQL BEGIN DECLARE SECTION および EXEC SQL END DECLARE SECTION ディレクティブでラップする必要があります。

```
void addDept(int v_deptno, char v_dname, char v_loc)
{
    EXEC SQL BEGIN DECLARE SECTION;
        int v_deptno_copy = v_deptno;
        char v_dname_copy[14+1] = v_dname;
        char v_loc_copy[13+1] = v_loc;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO dept VALUES( :v_deptno, :v_dname, :v_loc);
}
```

ホスト変数を使用して情報を取得するために、SELECT ステートメントに INTO 句を含めることもできます。

```
EXEC SQL SELECT deptno, dname, loc  
        INTO :v_deptno, :v_dname, v_loc FROM dept;
```

SELECT ステートメントによって返される各列には、 INTO 句にタイプ互換のターゲット変数が必要です。これは、単一の行を取得する簡単な例です。複数の行を取得するには、次の例に示すように、カーソルを定義する必要があります。

3.2 例-カーソルを使用して結果セットを処理する

次のコードサンプルは、カーソルを使用して結果セットを処理する方法を示しています。カーソルの作成と使用には、4 つの基本的な手順が含まれます。

1. DECLARE CURSOR ステートメントを使用して、カーソルを定義します。
2. OPEN CURSOR ステートメントを使用して、カーソルを開きます。
3. FETCH ステートメントを使用して、カーソルからデータを取得します。
4. CLOSE CURSOR ステートメントを使用して、カーソルを閉じます。

ホスト変数を宣言した後、この例では、ユーザーが指定したロール名とパスワードを使用して edb データベースに接続し、emp テーブルを照会します。クエリは、値を employees という名前のカーソルに返します。次に、サンプルコードはカーソルを開き、結果セットを一度に 1 行ずつループして結果セットを出力します。サンプルが結果セットの終わりを検出すると、接続を閉じます。

```

/*****
 * print_emps.pgc
 *
 */
#include <stdio.h>

int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
        char *username = argv[1];
        char *password = argv[2];
        int v_empno;
        char v_ename[40];
        double v_sal;
        double v_comm;
        short v_comm_ind;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR sqlprint;

    EXEC SQL CONNECT TO edb USER :username IDENTIFIED BY :password;

    EXEC SQL DECLARE employees CURSOR FOR
        SELECT
            empno, ename, sal, comm
        FROM
            emp;

    EXEC SQL OPEN employees;

```

(次のページに続く)

(前のページからの続き)

```

EXEC SQL WHENEVER NOT FOUND DO break;

for (;;)
{
    EXEC SQL FETCH NEXT FROM employees
        INTO
            :v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind;

    if (v_comm_ind)
        printf("empno(%d), ename(%s), sal(%.2f) comm(NULL)\n",
            v_empno, v_ename, v_sal);
    else
        printf("empno(%d), ename(%s), sal(%.2f) comm(%.2f)\n",
            v_empno, v_ename, v_sal, v_comm);
}
EXEC SQL CLOSE employees;
EXEC SQL DISCONNECT;
}
/*****

```

コードサンプルは、C の `stdio` ライブラリのプロトタイプと型定義を含めることから始まり、`main` 関数を宣言します。

```

#include <stdio.h>

int main(int argc, char *argv[])
{

```

次に、アプリケーションは、データベースサーバーとの対話に使用されるホスト変数のセットを宣言します。

```

EXEC SQL BEGIN DECLARE SECTION;
    char *username = argv[1];
    char *password = argv[2];
    int v_empno;
    char v_ename[40];
    double v_sal;
    double v_comm;
    short v_comm_ind;
EXEC SQL END DECLARE SECTION;

```

`argv[]` は、ユーザーがクライアントアプリケーションを実行するときに入力したコマンドライン引数を含む配列です。`argv[1]` には最初のコマンドライン引数（この場合は `username`）が含まれ、`argv[2]` には 2 番目のコマンドライン引数（`password`）が含まれます。通常、実際のアプリケーションに含めるエラーチェックコードは省略していることに注意してください。宣言は、`username` と `password` の値を初期化し、ユーザーがクライアントアプリケーションを呼び出したときに入力した値に設定します。

（各変数の個別のコピーを作成する代わりに）SQL ステートメントで `argv[1]` と `argv[2]` を参照できると考え

ているかもしれません。それは機能しません。すべてのホスト変数は、BEGIN/END DECLARE SECTION 内で宣言する必要があります（PROC モードでコンパイルしている場合を除く）。argv は関数*パラメーター*（自動変数ではない）であるため、BEGIN/END DECLARE SECTION 内で宣言することはできません。PROC モードでコンパイルしている場合は、SQL ステートメント内で*任意*の C 変数を参照できます。

次のステートメントは、ECPGPlus またはデータベースサーバーから返されたエラーメッセージのテキストを出力することにより、SQL エラーに応答するようサーバーに指示します。

```
EXEC SQL WHENEVER SQLERROR sqlprint;
```

次に、クライアントアプリケーションは AdvancedServer との接続を確立します。

```
EXEC SQL CONNECT TO edb USER :username IDENTIFIED BY :password;
```

CONNECT ステートメントは、:username および :password ホスト変数にある値を使用して edb データベースへの接続を作成し、接続時にサーバーに対してアプリケーションを認証します。

次のステートメントは、employees という名前のカーソルを宣言しています。

```
EXEC SQL DECLARE employees CURSOR FOR
SELECT
    empno, ename, sal, comm
FROM
    emp;
```

employees には、emp テーブルの SELECT ステートメントの結果セットが含まれます。クエリは、empno、ename、sal、comm の列から従業員情報を返します。カーソルを宣言するとき、INTO 節が含まれないことに注意してください。代わりに、カーソルから FETCH を指定するときにターゲット変数（または記述子）を指定します。

カーソルから行をフェッチする前に、クライアントアプリケーションはカーソルを OPEN にする必要があります。

```
EXEC SQL OPEN employees;
```

後続の FETCH セクションでは、クライアントアプリケーションはカーソルの内容をループします。クライアントアプリケーションには、カーソルの最後に到達したときにサーバーに break（つまり、ループを終了する）を指示する WHENEVER ステートメントが含まれています。

```
EXEC SQL WHENEVER NOT FOUND DO break;
```

クライアントアプリケーションは、FETCH ステートメントを使用して、カーソルから INTO 以前に宣言されたホスト変数の各行を取得します。

```
for (;;)
{
```

(次のページに続く)

(前のページからの続き)

```
EXEC SQL FETCH NEXT FROM employees
INTO
:v_empno, :v_ename, :v_sal, :v_comm INDICATOR :v_comm_ind;
```

FETCH ステートメントは INTO 句を使用して、取得した値を :v_empno、:v_ename、:v_sal、:v_comm ホスト変数（および :v_comm_ind ヌルインジケータ）に割り当てます。カーソルの最初の値は INTO 句にリストされている最初の変数に割り当てられ、2 番目の値は 2 番目の変数に割り当てられます。

FETCH ステートメントには、INDICATOR キーワードとヌルインジケータを保持するホスト変数も含まれます。取得したレコードの comm 列に NULL 値が含まれる場合、v_comm_ind はゼロ以外の値に設定され、列が NULL であることを示します。

次に、コードはヌルインジケータをチェックし、適切な画面上の結果を表示します。

```
if (v_comm_ind)
    printf("empno(%d), ename(%s), sal(%.2f) comm(NULL)\n",
           v_empno, v_ename, v_sal);
else
    printf("empno(%d), ename(%s), sal(%.2f) comm(%.2f)\n",
           v_empno, v_ename, v_sal, v_comm);
}
```

ヌルインジケータが 0（つまり false）の場合、v_comm には意味のある値が含まれ、printf 関数は手数料を表示します。null インジケータにゼロ以外の値が含まれる場合、comm は NULL であり、printf は文字列 'NULL' を表示します。そのホスト変数に NULL をフェッチすると、ホスト変数（null インジケータ以外）には意味のある値が含まれないことに注意してください。NULL になる可能性のある値にはヌルインジケータを使用する必要があります。

コードサンプルの最後のステートメントはカーソル (employees) を閉じ、サーバーへの接続を閉じます。

```
EXEC SQL CLOSE employees;
EXEC SQL DISCONNECT;
```


第 4 章

記述子の使用

動的 SQL を使用すると、クライアントアプリケーションは、実行時に構成される SQL ステートメントを実行できます。これは、クライアントアプリケーションを記述しているときに内容や形式がわからない場合に便利です。ECPGPlus では、識別子（テーブル名、列名、インデックス名など）の代わりにホスト変数を使用することはできません。代わりに、動的 SQL ステートメントを使用して、情報を含む文字列を作成し、その文字列を実行する必要があります。文字列は、クライアントとサーバーの間で*descriptor* の形式で渡されます。記述子は、データとデータの形状に関する情報の両方を含むデータ構造です。

クライアントアプリケーションは、GET DESCRIPTOR ステートメントを使用して記述子から情報を取得する必要があります。次の手順では、動的 SQL を使用したクライアントアプリケーションの基本的なフローについて説明します。

1. ALLOCATE DESCRIPTOR ステートメントを使用して、結果セット（選択リスト）の記述子を割り当てます。
2. ALLOCATE DESCRIPTOR ステートメントを使用して、入力パラメーター（バインド変数）の記述子を割り当てます。
3. SQL ステートメントのテキストを取得、アセンブル、または計算します。
4. PREPARE ステートメントを使用して、SQL ステートメントを解析および構文チェックします。
5. DESCRIBE ステートメントを使用して、選択リストを選択リスト記述子に記述します。
6. DESCRIBE ステートメントを使用して、バインド変数記述子への入力パラメーターを記述します。
7. 各入力パラメーターの値をユーザーに要求します（必要な場合）。SET DESCRIPTOR ステートメントを使用して、値を記述子に割り当てます。
8. DECLARE CURSOR ステートメントを使用して、ステートメントのカーソルを定義します。
9. OPEN CURSOR ステートメントを使用して、ステートメントのカーソルを開きます。
10. FETCH ステートメントを使用して、カーソルから各行をフェッチし、選択リスト記述子に各行を保存します。

11. GET DESCRIPTOR コマンドを使用して、選択リスト記述子に問い合わせ、現在の行の各列の値を見つけます。
12. CLOSE CURSOR ステートメントを使用して、カーソルを閉じ、カーソルリソースを解放します。

記述子には、以下の表にリストされている属性が含まれる場合があります。

Field	Type	Attribute Description
カーディナリティ	整数	結果セットの行数。
データ	なし	データ値。
DATETIME_INTERVAL_CODE	整数	TYPE が 9 の場合: 1-日付 2-時間 3-タイムスタンプ 4-タイムゾーンでの時間 5-タイムゾーン付きタイムスタンプ
DATETIME_INTERVAL_PRECISION	整数	未使用。
指標	整数	NULL または切り捨てられた値を示します。
KEY_MEMBER	整数	未使用 (FALSE を返します)。
長さ	整数	データの長さ (サーバーに保存されている)。
NAME	ひも	データが存在する列の名前。
NULLABLE	整数	未使用 (TRUE を返します)。
OCTET_LENGTH	整数	サーバーに保存されているデータの長さ (バイト単位)。
精度	整数	データ精度 (データが数値型の場合)。
RETURNED_LENGTH	整数	データ項目の実際の長さ。
RETURNED_OCTET_LENGTH	整数	データ項目の実際の長さ。
スケール	整数	データスケール (データが数値型の場合)。
TYPE	整数	列のデータ型を表す数値コード: 1-SQL3_CHARACTER 2-SQL3_NUMERIC 3-SQL3_DECIMAL 4-SQL3_INTEGER 5-SQL3_SMALLINT 6-SQL3_FLOAT 7-SQL3_REAL 8-SQL3_DOUBLE_PRECISION 9-SQL3_DATE_TIME_TIMESTAMP 10-SQL3_INTERVAL 12-SQL3_CHARACTER_VARYING 13-SQL3_ENUMERATED 14-SQL3_BIT 15-SQL3_BIT_VARYING 16-SQL3_BOOLEAN

4.1 例-記述子を使用してデータを返す

次の簡単なアプリケーションは、エンドユーザーが入力した SQL ステートメントを実行します。コードサンプルは以下を示します。

- SQL 記述子を使用して SELECT ステートメントを実行する方法。
- ステートメントによって返されたデータとメタデータを見つける方法。

アプリケーションは、エンドユーザーからの SQL ステートメントを受け入れ、ステートメントをテストして SELECT キーワードが含まれているかどうかを確認し、ステートメントを実行します。

アプリケーションを呼び出すとき、エンドユーザーは、SQL ステートメントが実行されるデータベースの名前と、クエリのテキストを含む文字列を提供する必要があります。

たとえば、ユーザーは次のコマンドでサンプルを呼び出すことができます。

```
./exec_stmt edb "SELECT * FROM emp"

/*****
/* exec_stmt.pgc
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <sql3types.h>
#include <sqlca.h>

EXEC SQL WHENEVER SQLERROR SQLPRINT;
static void print_meta_data( char * desc_name );

char *md1 = "col field          data          ret";
char *md2 = "num name          type          len";
char *md3 = "--- -----";

int main( int argc, char *argv[] )
{

EXEC SQL BEGIN DECLARE SECTION;
    char *db    = argv[1];
    char *stmt = argv[2];
    int    col_count;
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO :db;
```

(次のページに続く)

(前のページからの続き)

```

EXEC SQL ALLOCATE DESCRIPTOR parse_desc;
EXEC SQL PREPARE query FROM :stmt;
EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
EXEC SQL GET DESCRIPTOR 'parse_desc' :col_count = COUNT;

if( col_count == 0 )
{
    EXEC SQL EXECUTE IMMEDIATE :stmt;

    if( sqlca.sqlcode >= 0 )
        EXEC SQL COMMIT;
}
else
{
    int row;

    EXEC SQL ALLOCATE DESCRIPTOR row_desc;
    EXEC SQL DECLARE my_cursor CURSOR FOR query;
    EXEC SQL OPEN my_cursor;

    for( row = 0; ; row++ )
    {
        EXEC SQL BEGIN DECLARE SECTION;
            int    col;
        EXEC SQL END DECLARE SECTION;
        EXEC SQL FETCH IN my_cursor
            INTO SQL DESCRIPTOR row_desc;

        if( sqlca.sqlcode != 0 )
            break;

        if( row == 0 )
            print_meta_data( "row_desc" );

        printf("[RECORD %d]\n", row+1);

        for( col = 1; col <= col_count; col++ )
        {
            EXEC SQL BEGIN DECLARE SECTION;
                short    ind;
                varchar val[40+1];
                varchar name[20+1];
            EXEC SQL END DECLARE SECTION;

            EXEC SQL GET DESCRIPTOR 'row_desc'
                VALUE :col
                :val = DATA, :ind = INDICATOR, :name = NAME;

```

(次のページに続く)

(前のページからの続き)

```

        if( ind == -1 )
            printf( "  %-20s : <null>\n", name.arr );
        else if( ind > 0 )
            printf( "  %-20s : <truncated>\n", name.arr );
        else
            printf( "  %-20s : %s\n", name.arr, val.arr );
    }

    printf( "\n" );

}

printf( "%d rows\n", row );
}

exit( 0 );
}

static void print_meta_data( char *desc_name )
{
    EXEC SQL BEGIN DECLARE SECTION;
    char  *desc = desc_name;
    int    col_count;
    int    col;
    EXEC SQL END DECLARE SECTION;

    static char *types[] =
    {
        "unused          ",
        "CHARACTER        ",
        "NUMERIC           ",
        "DECIMAL            ",
        "INTEGER            ",
        "SMALLINT           ",
        "FLOAT              ",
        "REAL               ",
        "DOUBLE             ",
        "DATE_TIME          ",
        "INTERVAL           ",
        "unused             ",
        "CHARACTER_VARYING",
        "ENUMERATED         ",
        "BIT                 ",
        "BIT_VARYING         ",
        "BOOLEAN            ",
        "abstract           "
    };

};

EXEC SQL GET DESCRIPTOR :desc :col_count = count;

```

(次のページに続く)

(前のページからの続き)

```

printf( "%s\n", md1 );
printf( "%s\n", md2 );
printf( "%s\n", md3 );

for( col = 1; col <= col_count; col++ )
{

    EXEC SQL BEGIN DECLARE SECTION;
        int      type;
        int      ret_len;
        varchar name[21];
    EXEC SQL END DECLARE SECTION;
    char *type_name;

    EXEC SQL GET DESCRIPTOR :desc
        VALUE :col
        :name = NAME,
        :type = TYPE,
        :ret_len = RETURNED_OCTET_LENGTH;

    if( type > 0 && type < SQL3_abstract )
        type_name = types[type];
    else
        type_name = "unknown";

    printf( "%02d: %-20s %-17s %04d\n",
        col, name.arr, type_name, ret_len );
}
printf( "\n" );
}

/*****

```

コードサンプルは、C の `stdio` および `stdlib` ライブラリ、SQL データ型シンボル、および SQLCA（SQL コミュニケーションエリア）構造のプロトタイプと型定義を含めることから始まります。

```

#include <stdio.h>
#include <stdlib.h>
#include <sql3types.h>
#include <sqlca.h>

```

サンプルは最小限のエラー処理を提供します。アプリケーションで SQL エラーが発生すると、エラーメッセージが画面に出力されます。

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;
```

アプリケーションには、記述子で見つかったメタデータを出力する `print_meta_data()` という名前の関数の前方宣言が含まれています。

```
static void print_meta_data( char * desc_name );
```

次のコードは、アプリケーションがメタデータを印刷するときに使用する列ヘッダー情報を指定します。

```
char *md1 = "col field data ret";
char *md2 = "num name type len";
char *md3 = "--- -----";

int main( int argc, char *argv[] )
{
```

次の宣言セクションでは、アプリケーションが接続するデータベースの名前、SQL ステートメントの内容、および結果セット（ある場合）の列数を保持するホスト変数を含むホスト変数を識別します。

```
EXEC SQL BEGIN DECLARE SECTION;
    char *db = argv[1];
    char *stmt = argv[2];
    int col_count;
EXEC SQL END DECLARE SECTION;
```

アプリケーションはデータベースに接続します（デフォルトの資格情報を使用）：

```
EXEC SQL CONNECT TO :db;
```

次に、アプリケーションは SQL 記述子を割り当てて、ステートメントのメタデータを保持します。

```
EXEC SQL ALLOCATE DESCRIPTOR parse_desc;
```

アプリケーションは PREPARE ステートメントを使用して、ユーザーから提供された文字列を構文チェックします。

```
EXEC SQL PREPARE query FROM :stmt;
```

クエリのメタデータを SQL 記述子に移動するための DESCRIBE ステートメント。

```
EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;
```

次に、アプリケーションは記述子に問い合わせて結果セット内の列数を検出し、それをホスト変数 `col_count` に格納します。

```
EXEC SQL GET DESCRIPTOR parse_desc :col_count = COUNT;
```

列数がゼロの場合、エンドユーザーは SELECT ステートメントを入力しませんでした。アプリケーションは EXECUTE IMMEDIATE ステートメントを使用して、ステートメントの内容を処理します。

```
if( col_count == 0 )
{
    EXEC SQL EXECUTE IMMEDIATE :stmt;
```

ステートメントが正常に実行されると、アプリケーションは COMMIT を実行します。

```
    if( sqlca.sqlcode >= 0 )
        EXEC SQL COMMIT;
}
else
{
```

ユーザーが入力したステートメントが SELECT ステートメント（列数がゼロ以外であることがわかっている）である場合、アプリケーションは row という名前の変数を宣言します。

```
int row;
```

次に、アプリケーションは、結果セット内の特定の行の説明と値を保持する別の記述子を割り当てます。

```
EXEC SQL ALLOCATE DESCRIPTOR row_desc;
```

アプリケーションは、準備されたステートメントのカーソルを宣言して開きます。

```
EXEC SQL DECLARE my_cursor CURSOR FOR query;
EXEC SQL OPEN my_cursor;
```

結果セットの行をループします。

```
for( row = 0; ; row++ )
{
    EXEC SQL BEGIN DECLARE SECTION;
        int col;
    EXEC SQL END DECLARE SECTION;
```

次に、FETCH を使用してカーソルから次の行を記述子に取得します。

```
EXEC SQL FETCH IN my_cursor INTO SQL DESCRIPTOR row_desc;
```

アプリケーションは、FETCH が失敗しなかったことを確認します。FETCH が失敗すると、アプリケーションは結果セットの最後に到達し、ループを中断します。

```
if( sqlca.sqlcode != 0 )
    break;
```

アプリケーションは、これがカーソルの最初の行であるかどうかを確認します。そうである場合、アプリケーションは行のメタデータを出力します。

```
if( row == 0 )
    print_meta_data( "row_desc" );
```

次に、行番号を含むレコードヘッダーを出力します。

```
printf( "[RECORD %d]\n", row+1 );
```

次に、行の各列をループします。

```
for( col = 1; col <= col_count; col++ )
{
    EXEC SQL BEGIN DECLARE SECTION;
        short ind;
        varchar val[40+1];
        varchar name[20+1];
    EXEC SQL END DECLARE SECTION;
```

アプリケーションは、行記述子 (row_desc) に問い合わせて、列値 (:val)、ヌルインジケータ (:ind)、および列名 (:name) を上記で宣言したホスト変数にコピーします。コンマ区切りリストを使用して、記述子から複数のアイテムを取得できることに注意してください。

```
EXEC SQL GET DESCRIPTOR row_desc
    VALUE :col
    :val = DATA, :ind = INDICATOR, :name = NAME;
```

null インジケータ (ind) が負の場合、列の値は NULL です。null インジケータが 0 より大きい場合、列の値が長すぎて val ホスト変数に収まらないため (<truncated>) を出力します。それ以外の場合、null インジケータは 0 (NOT NULL を意味する) であるため、出力しますいずれの場合も、値 (または <null> または <truncated>) の前に列の名前を付けます。

```
    if( ind == -1 )
        printf( " %-20s : <null>\n", name.arr );
    else if( ind > 0 )
        printf( " %-20s : <truncated>\n", name.arr );
    else
        printf( " %-20s : %s\n", name.arr, val.arr );
}

printf( "\n" );
}
```

ループが終了すると、アプリケーションはフェッチされた行数を出力して終了します。

```
    printf( "%d rows\n", row );
}

exit( 0 );
}
```

print_meta_data() 関数は、記述子からメタデータを抽出し、各列の名前、データ型、および長さを出力します。

```
static void print_meta_data( char *desc_name )
{
```

アプリケーションはホスト変数を宣言します:

```
EXEC SQL BEGIN DECLARE SECTION;
    char *desc = desc_name;
    int col_count;
    int col;
EXEC SQL END DECLARE SECTION;
```

次に、アプリケーションは、データ型値 (numeric) をデータ型名にマッピングする文字列の配列を定義します。記述子で見つかった数値を使用して、この配列にインデックスを付けます。たとえば、特定の列のタイプが 2 であることが判明した場合、types[2] と記述することでそのタイプの名前 (NUMERIC) を見つけることができます。

```
static char *types[] =
{
    "unused ",
    "CHARACTER ",
    "NUMERIC ",
    "DECIMAL ",
    "INTEGER ",
    "SMALLINT ",
    "FLOAT ",
    "REAL ",
    "DOUBLE ",
    "DATE_TIME ",
    "INTERVAL ",
    "unused ",
    "CHARACTER_VARYING",
    "ENUMERATED ",
    "BIT ",
    "BIT_VARYING ",
    "BOOLEAN ",
    "abstract "
```

(次のページに続く)

(前のページからの続き)

};

アプリケーションは、記述子から列カウントを取得します。プログラムは、記述子の名前を含むホスト変数 (desc) を使用して記述子を参照することに注意してください。ほとんどのシナリオでは、識別子を使用して記述子を参照しますが、この場合、呼び出し元が記述子名を提供したため、ホスト変数を使用して記述子を参照できます。

```
EXEC SQL GET DESCRIPTOR :desc :col_count = count;
```

アプリケーションは列ヘッダーを印刷します（このアプリケーションの冒頭で定義されています）：

```
printf( "%s\n", md1 );
printf( "%s\n", md2 );
printf( "%s\n", md3 );
```

次に、記述子にある各列をループし、各列の名前、タイプ、および長さを出力します。

```
for( col = 1; col <= col_count; col++ )
{
    EXEC SQL BEGIN DECLARE SECTION;
        int type;
        int ret_len;
        varchar name[21];
    EXEC SQL END DECLARE SECTION;
    char *type_name;
```

現在の列の名前、タイプコード、および長さを取得します。

```
EXEC SQL GET DESCRIPTOR :desc
VALUE :col
:name = NAME,
:type = TYPE,
:ret_len = RETURNED_OCTET_LENGTH;
```

数値型コードが「既知の」型コード（つまり、types[] 配列にある型コード）と一致する場合、対応する型の名前に type_name を設定します。それ以外の場合は、type_name を /"unknown/" に設定します。

```
if( type > 0 && type < SQL3_abstract )
    type_name = types[type];
else
    type_name = "unknown";
```

列番号、名前、タイプ名、および長さを出力します。

```

printf( "%02d: %-20s %-17s %04d\n",
        col, name.arr, type_name, ret_len );
}
printf( "\n" );
}

```

次のコマンドでサンプルアプリケーションを呼び出す場合:

```
./exec_stmt test "SELECT * FROM emp WHERE empno IN(7902, 7934)"
```

アプリケーションは以下を返します。

```

col field ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0data_
↳ ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0ret
num name ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 type_
↳ ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0len
-----
01: empno ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0NUMERIC ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 0004
02: ename ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0CHARACTER_VARYING 0004
03: job ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0CHARACTER_VARYING 0007
04: mgr ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0NUMERIC ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 0004
05: hiredate ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 DATE_TIME ^c2^a0_
↳ ^c2^a0 ^c2^a0 ^c2^a0 0018
06: sal ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0NUMERIC ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 0007
07: comm ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ NUMERIC ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 0000
08: deptno ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 NUMERIC_
↳ ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 0002

[RECORD 1]
^c2^a0 empno ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: 7902
^c2^a0 ename ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: FORD
^c2^a0 job ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: ANALYST
^c2^a0 mgr ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: 7566
^c2^a0 hiredate ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 : 03-DEC-81_
↳ 00:00:00
^c2^a0 sal ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: 3000.00
^c2^a0 comm ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ : <null>

```

(次のページに続く)

(前のページからの続き)

```

^c2^a0 deptno ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 : 20

[RECORD 2]
^c2^a0 empno ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: 7934
^c2^a0 ename ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: MILLER
^c2^a0 job ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: CLERK
^c2^a0 mgr ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: 7782
^c2^a0 hiredate ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 : 23-JAN-82_
↳ 00:00:00
^c2^a0 sal ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ ^c2^a0: 1300.00
^c2^a0 comm ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0_
↳ : <null>
^c2^a0 deptno ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 ^c2^a0 : 10

2 rows

```

第 5 章

動的 SQL ステートメントの作成と実行

次の例は、動的 SQL ステートメントを作成および実行するための 4 つの手法を示しています。各例は、ステートメントと入力タイプの異なる組み合わせの処理を示しています。

- 最初の例は、SELECT ステートメントを含まず、入力変数を必要としない SQL ステートメントの処理と実行を示しています。この例は、OracleDynamicSQLMethod1 で使用される手法に対応しています。
- 2 番目の例は、SELECT ステートメントを含まず、既知の数の入力変数を含む SQL ステートメントの処理と実行を示しています。この例は、OracleDynamicSQLMethod2 で使用される手法に対応しています。
- 3 番目の例は、SELECT ステートメントを含むことがあり、既知の数の入力変数を含む SQL ステートメントの処理と実行を示しています。この例は、OracleDynamicSQLMethod3 で使用される手法に対応しています。
- 4 番目の例は、SELECT ステートメントを含む可能性があり、入力変数の数が不明な SQL ステートメントの処理と実行を示しています。この例は、OracleDynamicSQLMethod4 で使用される手法に対応しています。

5.1 例-パラメータなしの非クエリステートメントの実行

次の例は、EXECUTE IMMEDIATE コマンドを使用して、アプリケーションを実行するまでステートメントのテキストがわからない SQL ステートメントを実行する方法を示しています。EXECUTE IMMEDIATE を使用して、結果セットを返すステートメントを実行することはできません。EXECUTE IMMEDIATE を使用して、パラメータプレースホルダーを含むステートメントを実行することはできません。

EXECUTE IMMEDIATE ステートメントは、実行するたびに SQL ステートメントを解析および計画します。これは、アプリケーションのパフォーマンスに悪影響を及ぼす可能性があります。同じ文を繰り返し実行する予定がある場合は、次の例で説明する PREPARE/EXECUTE テクニックの使用を検討してください。

```

/*****/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
    char    *insertStmt;

    EXEC SQL WHENEVER SQLERROR DO handle_error();

    EXEC SQL CONNECT :argv[1];

    insertStmt = "INSERT INTO dept VALUES(50, 'ACCTG', 'SEATTLE')";

    EXEC SQL EXECUTE IMMEDIATE :insertStmt;

    fprintf(stderr, "ok\n");

    EXEC SQL COMMIT RELEASE;

    exit(EXIT_SUCCESS);
}

static void handle_error(void)
{
    fprintf(stderr, "%s\n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;

    exit(EXIT_FAILURE);
}

```

(次のページに続く)

(前のページからの続き)

```

/*****

```

コードサンプルは、C の `stdio`、`string` および `stdlib` ライブラリのプロトタイプと型定義を含めることから始まり、プログラムの基本インフラストラクチャを提供します。

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

static void handle_error(void);
int main(int argc, char *argv[])
{
    char *insertStmt;

```

次に、エラーハンドラーを設定します。ECPGPlus は、SQL エラーが発生するたびに `handle_error()` 関数を呼び出します。

```

EXEC SQL WHENEVER SQLERROR DO handle_error();

```

次に、コマンドラインで指定された資格情報を使用してデータベースに接続します。

```

EXEC SQL CONNECT :argv[1];

```

次に、プログラムは `EXECUTE IMMEDIATE` ステートメントを使用して SQL ステートメントを実行し、`dept` テーブルに行を追加します。

```

insertStmt = "INSERT INTO dept VALUES (50, 'ACCTG', 'SEATTLE')";

EXEC SQL EXECUTE IMMEDIATE :insertStmt;

```

`EXECUTE IMMEDIATE` コマンドが何らかの理由で失敗した場合、ECPGPlus は `handle_error()` 機能呼び出します (エラーメッセージをユーザーに表示した後、アプリケーションを終了します)。`EXECUTE IMMEDIATE` コマンドが成功すると、アプリケーションは (ok) というメッセージをユーザーに表示し、変更をコミットし、サーバーから切断して、アプリケーションを終了します。

```

fprintf(stderr, "ok\n");

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS);
}

```

ECPGPlus は、SQL エラーが発生すると `handle_error()` 関数を呼び出します。`handle_error()` 関数は、エラーメッセージの内容を出力し、エラーハンドラをリセットし、変更をロールバックし、データベースから切断

し、アプリケーションを終了します。

```
static void handle_error(void)
{
    fprintf(stderr, "%s\n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;

    exit(EXIT_FAILURE);
}
```

5.2 例-指定された数のプレースホルダーを使用した非クエリステートメントの実行

既知の数のパラメータープレースホルダーを含むクエリ以外のコマンドを実行するには、最初にステートメントを PREPARE にし（ステートメントハンドル*を提供）、次にステートメントハンドルを使用してステートメントを “EXECUTE” にする必要があります。アプリケーションがステートメントを実行するとき、ステートメントで見つかった各プレースホルダーに*値*を提供する必要があります。

アプリケーションが PREPARE/EXECUTE メカニズムを使用する場合、各 SQL ステートメントは 1 回解析されて計画されますが、何度も実行される可能性があります（毎回異なる*値*を提供）。

ECPGPlus は、可能であれば、各パラメーター値を SQL ステートメントに必要なタイプに変換します。不可能な場合、ECPGPlus はエラーを報告します。

```

/*****
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
    char *stmtText;

    EXEC SQL WHENEVER SQLERROR DO handle_error();

    EXEC SQL CONNECT :argv[1];

    stmtText = "INSERT INTO dept VALUES (?, ?, ?)";

    EXEC SQL PREPARE stmtHandle FROM :stmtText;

    EXEC SQL EXECUTE stmtHandle USING :argv[2], :argv[3], :argv[4];

    fprintf(stderr, "ok\n");

    EXEC SQL COMMIT RELEASE;

    exit(EXIT_SUCCESS);
}

static void handle_error(void)
{
    printf("%s\n", sqlca.sqlerrm.sqlerrmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
}

```

(次のページに続く)

(前のページからの続き)

```
EXEC SQL ROLLBACK RELEASE;

    exit(EXIT_FAILURE);
}

/*****
```

コードサンプルは、C の `stdio`、`string`、`stdlib` および `sqlca` ライブラリのプロトタイプと型定義を含めることから始まり、プログラムの基本インフラストラクチャを提供します。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
    char *stmtText;
```

次に、エラーハンドラーを設定します。ECPGPlus は、SQL エラーが発生するたびに `handle_error()` 関数を呼び出します。

```
EXEC SQL WHENEVER SQLERROR DO handle_error();
```

次に、コマンドラインで指定された資格情報を使用してデータベースに接続します。

```
EXEC SQL CONNECT :argv[1];
```

次に、プログラムは PREPARE ステートメントを使用して、3 つのパラメーターマーカーを含むステートメントを解析および計画します。PREPARE ステートメントが成功すると、ステートメントの実行に使用できるステートメントハンドルを作成します（この例では、ステートメントハンドルの名前は `stmtHandle` です）。同じステートメントハンドルを使用して、特定のステートメントを複数回実行できます。

```
stmtText = "INSERT INTO dept VALUES(?, ?, ?)";

EXEC SQL PREPARE stmtHandle FROM :stmtText;
```

ステートメントを解析および計画した後、アプリケーションは EXECUTE ステートメントを使用して、ステートメントハンドルに関連付けられたステートメントを実行し、パラメーターマーカーをユーザーが指定した値に置き換えます。

```
EXEC SQL EXECUTE stmtHandle USING :argv[2], :argv[3], :argv[4];
```

EXECUTE コマンドが何らかの理由で失敗した場合、ECPGPlus は `handle_error()` 機能呼び出します（エ

ラーメッセージをユーザーに表示した後、アプリケーションを終了します)。EXECUTE コマンドが成功すると、アプリケーションは (ok) というメッセージをユーザーに表示し、変更をコミットし、サーバーから切断して、アプリケーションを終了します。

```
fprintf(stderr, "ok\n");

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS);
}
```

ECPGPlus は、SQL エラーが発生すると `handle_error()` 関数を呼び出します。 `handle_error()` 関数は、エラーメッセージの内容を出力し、エラーハンドラをリセットし、変更をロールバックし、データベースから切断し、アプリケーションを終了します。

```
static void handle_error(void)
{
    printf("%s\n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(EXIT_FAILURE);
}
```

5.3 例-既知の数のプレースホルダーでクエリを実行する

この例は、既知の数の入力パラメーターと、結果セットの既知の数の列を使用して*query* を実行する方法を示しています。このメソッドは、カーソルを開いて結果セットを反復処理する前に、PREPARE ステートメントを使用してクエリを解析および計画します。

```

/*****
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
    VARCHAR empno[10];
    VARCHAR ename[20];

    EXEC SQL WHENEVER SQLERROR DO handle_error();

    EXEC SQL CONNECT :argv[1];

    EXEC SQL PREPARE queryHandle
        FROM "SELECT empno, ename FROM emp WHERE deptno = ?";

    EXEC SQL DECLARE empCursor CURSOR FOR queryHandle;

    EXEC SQL OPEN empCursor USING :argv[2];

    EXEC SQL WHENEVER NOT FOUND DO break;

    while(true)
    {
        EXEC SQL FETCH empCursor INTO :empno, :ename;

        printf("%-10s %s\n", empno.arr, ename.arr);
    }

    EXEC SQL CLOSE empCursor;

    EXEC SQL COMMIT RELEASE;

    exit(EXIT_SUCCESS);
}

```

(次のページに続く)

(前のページからの続き)

```
static void handle_error(void)
{
    printf("%s\n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;

    exit(EXIT_FAILURE);
}

/*****
```

コードサンプルは、C の `stdio`, `string`, `stdlib`, `stdbool` および `sqlca` ライブラリのプロトタイプと型定義を含めることから始まり、プログラムの基本インフラストラクチャを提供します。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sqlca.h>

static void handle_error(void);

int main(int argc, char *argv[])
{
    VARCHAR empno[10];
    VARCHAR ename[20];
```

次に、エラーハンドラーを設定します。ECPGPlus は、SQL エラーが発生するたびに `handle_error()` 関数を呼び出します。

```
EXEC SQL WHENEVER SQLERROR DO handle_error();
```

次に、コマンドラインで指定された資格情報を使用してデータベースに接続します。

```
EXEC SQL CONNECT :argv[1];
```

次に、プログラムは PREPARE ステートメントを使用して、単一のパラメーターマーカーを含むクエリを解析および計画します。PREPARE ステートメントが成功すると、ステートメントの実行に使用できるステートメントハンドルを作成します（この例では、ステートメントハンドルの名前は `stmtHandle` です）。同じステートメントハンドルを使用して、特定のステートメントを複数回実行できます。

```
EXEC SQL PREPARE stmtHandle
FROM "SELECT empno, ename FROM emp WHERE deptno = ?";
```

次に、プログラムはカーソル `empCursor` を宣言して開き、準備された SELECT ステートメントのパラメーター

マーカーをユーザーが指定した値に置き換えます。OPEN ステートメントには USING 句が含まれていることに注意してください。USING 句は、クエリで見つかった各プレースホルダーの*値*を提供する必要があります。

```
EXEC SQL DECLARE empCursor CURSOR FOR stmtHandle;

EXEC SQL OPEN empCursor USING :argv[2];

EXEC SQL WHENEVER NOT FOUND DO break;

while(true)
{
```

プログラムはカーソルを反復処理し、選択した部門の各従業員の従業員番号と名前を出力します。

```
EXEC SQL FETCH empCursor INTO :empno, :ename;

printf("%-10s %s\n", empno.arr, ename.arr);
}
```

その後、プログラムはカーソルを閉じ、変更をコミットし、サーバーから切断して、アプリケーションを終了します。

```
EXEC SQL CLOSE empCursor;

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS);
}
```

アプリケーションは、SQL エラーが発生すると handle_error() 関数を呼び出します。handle_error() 関数は、エラーメッセージの内容を出力し、エラーハンドラをリセットし、変更をロールバックし、データベースから切断し、アプリケーションを終了します。

```
static void handle_error(void)
{
    printf("%s\n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;

    exit(EXIT_FAILURE);
}
```

5.4 例-未知の数の変数を使用したクエリの実行

次の例は、結果セット内の入力パラメーターや列の数が不明なクエリを実行する方法を示しています。このタイプのクエリは、クエリのテキストの入力をユーザーに要求した場合、またはユーザーがさまざまな条件（フィルターなど）から選択したフォームからクエリが組み立てられた場合に発生する可能性があります。

```

/*****
#include <stdio.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>

SQLDA *params;
SQLDA *results;

static void allocateDescriptors(int count,
                                int varNameLength,
                                int indNameLenth);
static void bindParams(void);
static void displayResultSet(void);

int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
        char    *username = argv[1];
        char    *password = argv[2];
        char    *stmtText = argv[3];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR sqlprint;

    EXEC SQL CONNECT TO test
        USER :username
        IDENTIFIED BY :password;

    params = sqlald(20, 64, 64);
    results = sqlald(20, 64, 64);

    EXEC SQL PREPARE stmt FROM :stmtText;

    EXEC SQL DECLARE dynCursor CURSOR FOR stmt;

    bindParams();

    EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

    displayResultSet(20);
}

```

(次のページに続く)

(前のページからの続き)

```
static void bindParams(void)
{
    EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO params;

    if (params->F < 0)
        fprintf(stderr, "Too many parameters required\n");
    else
    {
        int i;

        params->N = params->F;

        for (i = 0; i < params->F; i++)
        {
            char *paramName = params->S[i];
            int   nameLen   = params->C[i];
            char  paramValue[255];

            printf("Enter value for parameter %.*s: ",
                  nameLen, paramName);

            fgets(paramValue, sizeof(paramValue), stdin);

            params->T[i] = 1;          /* Data type = Character (1) */
            params->L[i] = strlen(paramValue) - 1;
            params->V[i] = strdup(paramValue);
        }
    }
}

static void displayResultSet(void)
{
    EXEC SQL DESCRIBE SELECT LIST FOR stmt INTO results;

    if (results->F < 0)
        fprintf(stderr, "Too many columns returned by query\n");
    else if (results->F == 0)
        return;
    else
    {
        int col;

        results->N = results->F;

        for (col = 0; col < results->F; col++)
        {
            int null_permitted, length;
```

(次のページに続く)

(前のページからの続き)

```

    sqlnul(&results->T[col],
           &results->T[col],
           &null_permitted);

    switch (results->T[col])
    {
        case 2:          /* NUMERIC */
        {
            int precision, scale;

            sqlprc(&results->L[col], &precision, &scale);

            if (precision == 0)
                precision = 38;

            length = precision + 3;
            break;
        }

        case 12:         /* DATE */
        {
            length = 30;
            break;
        }

        default:         /* Others */
        {
            length = results->L[col] + 1;
            break;
        }
    }

    results->V[col] = realloc(results->V[col], length);
    results->L[col] = length;
    results->T[col] = 1;
}

EXEC SQL WHENEVER NOT FOUND DO break;

while (1)
{
    const char *delimiter = "";

    EXEC SQL FETCH dynCursor USING DESCRIPTOR results;

    for (col = 0; col < results->F; col++)
    {

```

(次のページに続く)

(前のページからの続き)

```

        if (*results->I[col] == -1)
            printf("%s%s", delimiter, "<null>");
        else
            printf("%s%s", delimiter, results->V[col]);
        delimiter = ", ";
    }

    printf("\n");
}
}
}
/*****

```

コードサンプルは、C の `stdio` および `stdlib` ライブラリのプロトタイプと型定義を含めることから始まります。さらに、プログラムには `sqlda.h` および `sqlcpr.h` ヘッダーファイルが含まれています。 `sqlda.h` は、この例全体で使用される **SQLDA** 構造を定義します。 `sqlcpr.h` は、**SQLDA** 構造で見つかったメタデータを調べるために使用される関数の小さなセットを定義します。

```

#include <stdio.h>
#include <stdlib.h>
#include <sqlda.h>
#include <sqlcpr.h>

```

次に、プログラムは 2 つの **SQLDA** 構造体へのポインターを宣言します。最初の **SQLDA** 構造体 (`params`) は、動的クエリテキストで見つかったパラメーターマーカーのメタデータを記述するために使用されます。2 番目の **SQLDA** 構造体 (`results`) には、メタデータと、動的クエリの実行によって取得された結果セットの両方が含まれます。

```

SQLDA *params;
SQLDA *results;

```

次に、プログラムは 2 つのヘルパー関数を宣言します（コードサンプルの終わり近くで定義されます）。

```

static void bindParams(void);
static void displayResultSet(void);

```

次に、プログラムは 3 つのホスト変数を宣言します。最初の 2 つ (`username` と `password`) は、データベースサーバーへの接続に使用されます。3 番目のホスト変数 (`stmtTxt`) は、実行するクエリのテキストを含む **NULL** で終了する C 文字列です。これら 3 つのホスト変数の値は、コマンドライン引数から派生していることに注意してください。プログラムの実行が開始されると、エラーハンドラーが設定され、データベースサーバーに接続されます。

```
int main(int argc, char *argv[])
{
    EXEC SQL BEGIN DECLARE SECTION;
        char *username = argv[1];
        char *password = argv[2];
        char *stmtText = argv[3];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR sqlprint;
    EXEC SQL CONNECT TO test
        USER :username
        IDENTIFIED BY :password;
```

次に、プログラムは `sqlald()` 関数を呼び出して、各記述子に必要なメモリを割り当てます。各記述子には（とりわけ）が含まれます。

- 列名の配列へのポインタ
- インジケータ名の配列へのポインタ
- データ型の配列へのポインタ
- 長さの配列へのポインタ
- データ値の配列へのポインター。

SQLDA 記述子を割り当てる場合、結果セットで検索することが予想される列の最大数（SELECT リスト記述子の場合）または動的クエリテキストを検索することが予想されるパラメーターの最大数（バインドの場合）を指定します - `variabledescriptors` - この場合、20 列と 20 パラメーターを超えないことを指定します。また、各列（またはパラメーター）名と各インジケータ変数名の最大長も指定する必要があります。この場合、名前の長さは 64 バイト以下である必要があります。

SQLDA 構造の完全な説明については、[SQLDA 構造](#) セクションを参照してください。

```
params = sqlald(20, 64, 64);
results = sqlald(20, 64, 64);
```

SELECT リストとバインド記述子を割り当てた後、プログラムは動的ステートメントを準備し、結果セット上にカーソルを宣言します。

```
EXEC SQL PREPARE stmt FROM :stmtText;

EXEC SQL DECLARE dynCursor CURSOR FOR stmt;
```

次に、プログラムは `bindParams()` 関数を呼び出します。 `bindParams()` 関数はバインド記述子 (`params`) を調べ、動的クエリで見つかった各パラメーターマーカーの代わりに置き換える値をユーザーに求めます。

```
bindParams();
```

最後に、プログラムはカーソルを開き(ユーザーが提供したパラメーター値を使用して)、`displayResultSet()` 関数を呼び出して、クエリによって生成された結果セットを印刷します。

```
EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

displayResultSet();
}
```

`bindParams()` 関数は、動的クエリにパラメーターマーカーが含まれているかどうかを判断し、含まれている場合は、各パラメーターの値をユーザーに求め、その値を対応するマーカーにバインドします。 `DESCRIBE BIND VARIABLE` ステートメントは、`params SQLDA` 構造体に各パラメーターマーカーを説明する情報を入力します。

```
static void bindParams(void)
{
    EXEC SQL DESCRIBE BIND VARIABLES FOR stmt INTO params;
```

ステートメントにパラメーターマーカーが含まれていない場合、`params->F` には 0 が含まれます。ステートメントに記述子に収まるよりも多くのパラメーターが含まれている場合、`params->F` には負の数が含まれます(この場合、`params->F` の絶対値はステートメントで見つかったパラメーターマーカーの数)。 `params->F` に正の数が含まれる場合、その数はステートメントで見つかったパラメーターマーカーの数を示します。

```
if (params->F < 0)
    fprintf(stderr, "Too many parameters required\n");
else
{
    int i;

    params->N = params->F;
```

次に、プログラムは、ユーザーに値の入力を求めるループを実行し、ステートメントで見つかったパラメーターマーカーごとに 1 回繰り返します。

```
for (i = 0; i < params->F; i++)
{
    char *paramName = params->S[i];
    int nameLen = params->C[i];
    char paramValue[255];

    printf("Enter value for parameter %.*s: ",
           nameLen, paramName);

    fgets(paramValue, sizeof(paramValue), stdin);
```

ユーザーに特定のパラメーターの値の入力を求めた後、プログラムは `params->T[i]` を設定して値のデータ型

を示すことにより、その値をパラメーターに*バインド*します（タイプコードのリストについては Type Codes を参照）、params->L[i] を値の長さ（1 を差し引いて fgets() で追加された末尾の改行文字を削除します）、params->V[i] をユーザーが指定した NULL 終了文字列のコピーを指します。

```

    params->T[i] = 1;          /* Data type = Character (1) */
    params->L[i] = strlen(paramValue) + 1;
    params->V[i] = strdup(paramValue);
}
}
}

```

displayResultSet() 関数は結果セットの各行をループし、各列で見つかった値を出力します。displayResultSet() は DESCRIBE SELECT LIST ステートメントを実行することから始まります-このステートメントは、SQLDA 記述子 (results) に結果セットの各列の説明を取り込みます。

```

static void displayResultSet(void)
{
    EXEC SQL DESCRIBE SELECT LIST FOR stmt INTO results;
}

```

動的ステートメントが列を返さない場合（つまり、動的ステートメントが SELECT ステートメントではない場合）、results->F には 0 が含まれます。ステートメントが記述子に収まるより多くの列を返す場合、results->F には負の値が含まれます数値（この場合、results->F の絶対値はステートメントによって返される列の数を示します）。results->F に正の数が含まれる場合、その数はクエリによって返される列の数を示します。

```

if (results->F < 0)
    fprintf(stderr, "Too many columns returned by query\n");
else if (results->F == 0)
    return;
else
{
    int col;

    results->N = results->F;
}

```

次に、プログラムはループに入り、結果セットの各列に対して 1 回反復します。

```

for (col = 0; col < results->F; col++)
{
    int null_permitted, length;
}

```

results->T にある型コードをデコードするために、プログラムは sqlnul() 関数を呼び出します（TheSQL-DAStructure<sqlda_structure>の SQLDA 構造体の ‘T’ メンバーの説明を参照）。この sqlnul() の呼び出しにより、results->T[col] が変更され、タイプコードのみが含まれるようになります（ヌル可能フラグは null_permitted にコピーされます）。DESCRIBE SELECT LIST ステートメントは各列のタイプと各列の NULL 可能性を T 配列にエンコードするため、このステップが必要です。

```
sqlnul(&results->T[col]
      &results->T[col]
      &null_permitted);
```

列の実際のデータ型をデコードした後、プログラムは結果記述子を変更して、ECPGPlus に NULL 終了文字列の形式で各値を返すように指示します。記述子を変更する前に、プログラムは各値を保持するために必要なスペースの量を計算する必要があります。この計算を行うために、プログラムは各列の最大長 (results->V[col]) と各列のデータ型 (results->T[col]) を調べます。

数値の場合 (results->T[col] = 2), プログラムは sqlprc() 関数を呼び出して、列の長さから精度とスケールを抽出します。文字列形式の数値を保持するために必要なバイト数を計算するには、displayResultSet() は精度 (つまり、最大桁数) で始まり、符号文字、小数点、および NULL ターミネータに 3 バイトを追加します。

```
switch (results->T[col])
{
    case 2:          /* NUMERIC */
    {
        int precision, scale;

        sqlprc(&results->L[col], &precision, &scale);

        if (precision == 0)
            precision = 38;
        length = precision + 3;
        break;
    }
}
```

日付値の場合、プログラムは 30 のやや任意のハードコードされた長さを使用します。実際のアプリケーションでは、必要なスペースの量をより慎重に計算することができます。

```
case 12:          /* DATE */
{
    length = 30;
    break;
}
```

日付または数値以外のタイプの値の場合、displayResultSet() は DESCRIBE SELECT LIST によって報告される最大列幅で始まり、NULL ターミネータに 1 バイト追加されます。繰り返しますが、実際のアプリケーションでは、他のデータ型に対してより慎重な計算を含めることができます。

```
default:          /* Others */
{
    length = results->L[col] + 1;
    break;
}
```

(次のページに続く)

(前のページからの続き)

```

    }
}

```

特定の列を保持するために必要なスペースの量を計算した後、プログラムは値を保持するために十分なメモリを割り当て、`results->L[col]` を設定して `results->V[col]` で見つかったバイト数を示し、(`results->T[col]`) 列のタイプコードを設定します NULL で終わる文字列の形式で値を返すように、今後の FETCH ステートメントに指示する 1 に。

```

results->V[col] = malloc(length);
results->L[col] = length;
results->T[col] = 1;
}

```

この時点で、結果記述子は、FETCH ステートメントが各値を NULL で終わる文字列の形式で適切なサイズのバッファにコピーできるように構成されます。

次に、プログラムは、カーソルが使い果たされたときに、次のループから抜け出す新しいエラーハンドラを定義します。

```

EXEC SQL WHENEVER NOT FOUND DO break;

while (1)
{
    const char *delimiter = "";

```

プログラムは FETCH ステートメントを実行して、カーソルの次の行を `results` 記述子にフェッチします。「カーソル」が使い果たされたために FETCH ステートメントが失敗した場合、ループの先頭の前に `EXEC SQL WHENEVER` ディレクティブが見つかったため、制御はループの最後に移動します。

```
EXEC SQL FETCH dynCursor USING DESCRIPTOR results;
```

FETCH ステートメントは、結果記述子の以下のメンバーを取り込みます。

- `*results->I[col]` は、列に NULL 値 (-1) または非 NULL 値 (0) が含まれているかどうかを示します。値が非 NULL であるが、指定されたスペースに収まるには大きすぎる場合、値は切り捨てられ、`*results->I[col]` には正の値が含まれます。
- `results->V[col]` には、特定の列に対してフェッチされた値が含まれます。(unless `*results->I[col]` は、列の値が NULL であることを示します)。
- `results->L[col]` には、指定された列に対してフェッチされた値の長さが含まれます

最後に、`displayResultSet()` は結果セットの各列を反復処理し、対応する NULL インジケータを調べて、値を出力します。結果セットは整列されません-代わりに、各値はコンマによって前の値から分離されます。

```
    for (col = 0; col < results->F; col++)
    {
        if (*results->I[col] == -1)
            printf("%s%s", delimiter, "<null>");
        else
            printf("%s%s", delimiter, results->V[col]);
        delimiter = ", ";
    }

    printf("\n");
}
}
/*****/
```

第 6 章

エラー処理

ECPGPlus は、埋め込み SQL コードのエラーを検出して処理するための 2 つの方法を提供します。

- クライアントアプリケーションは、エラーメッセージの `sqlca` データ構造を調べ、クライアントアプリケーションにカスタマイズされたエラー処理を提供できます。
- クライアントアプリケーションには、ECPGPlus コンパイラにエラー処理コードを追加するよう指示する `EXEC SQL WHENEVER` ディレクティブを含めることができます。

6.1 `sqlca` でのエラー処理

`sqlca`（SQL 通信領域）は、`ecpglib` がサーバーからクライアントアプリケーションに情報を伝達するために使用するグローバル変数です。SQL ステートメント（`INSERT` または `SELECT` ステートメントなど）を実行した後、`sqlca` の内容を調べて、ステートメントが正常に完了したか、ステートメントが失敗したかを判断できます。

`sqlca` の構造は次のとおりです。

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

次のディレクティブを使用して、`sqlca` 機能を実装します。

```
EXEC SQL INCLUDE sqlca;
```

`ecpg` ディレクティブを含める場合、クライアントアプリケーションのヘッダー宣言に `sqlca.h` ファイルを `#include` する必要はありません。

`AdvancedServer` の `sqlca` 構造には、次のメンバーが含まれます。

`sqlcaid`

`sqlcaid` には文字列 `/"SQLCA/"` が含まれます。

`sqlabc`

`sqlabc` には `sqlca` 構造のサイズが含まれます。

`sqlcode`

`sqlcode` メンバーは **SQL92** で非推奨になりました。`AdvancedServer` は下位互換性のために `sqlcode` をサポートしますが、新しいコードを記述するときは `sqlstate` メンバーを使用する必要があります。

`sqlcode` は整数値です。正の `sqlcode` 値は、クライアントアプリケーションが無害な処理条件に遭遇したことを示し、負の値は警告またはエラーを示します。

ステートメントがエラーなしで処理される場合、`sqlcode` には 0 の値が含まれます。クライアントアプリケーションがステートメントの実行中にエラー（または警告）に遭遇した場合、`sqlcode` には最後に返されたコードが含まれます。

SQL 標準では、正の値 100 のみが定義されています。これは、処理された最新の **SQL** ステートメントが返された/影響を受けた行がないことを示します。標準 **SQL** では他の `sqlcode` 値を定義していないため、各条件に割り当てられる値はデータベースごとに異なる可能性があることに注意してください。

`sqlerrm` は `sqlca` 内に埋め込まれた構造で、2 つのメンバーで構成されます。

`sqlerrml`

`sqlerrml` には、`sqlerrmc` に現在保存されているエラーメッセージの長さが含まれます。

`sqlerrmc`

`sqlerrmc` には、`sqlstate` に保存されているコードに関連付けられたヌル終了メッセージテキストが含まれます。メッセージの長さが 149 文字を超える場合、`ecpglib` はエラーメッセージを切り捨てます。

`sqlerrp`

`sqlerrp` には `/"NOT SET/"` という文字列が含まれます。

`sqlerrd` は、6 つの要素を含む配列です。

sqlerrrd[1] には、処理された行の OID が含まれます（該当する場合）。

sqlerrrd[2] には、処理または返された行の数が含まれます。

sqlerrrd[0], sqlerrrd[3], sqlerrrd[4] と sqlerrrd[5] は未使用です。

sqlwarn は 8 文字を含む配列です。

sqlwarn 内の他の要素が 'W' に設定されている場合、sqlwarn[0] には 'W' の値が含まれます。

データ値がホスト変数に格納されたときに切り捨てられた場合、sqlwarn[1] には 'W' の値が含まれます。

クライアントアプリケーションが致命的でない警告を検出した場合、sqlwarn[2] には 'W' の値が含まれます。

sqlwarn[3], sqlwarn[4], sqlwarn[5], sqlwarn[6] および sqlwarn[7] は未使用です。

sqlstate

sqlstate は、クライアントアプリケーションからステートメントを実行した後の SQL 準拠のステータスコードを含む 5 文字の配列です。ステートメントがエラーなしで処理される場合、sqlstate には 00000 の値が含まれます。sqlstate はヌル終端文字列ではないことに注意してください。

sqlstate コードは階層スキームで割り当てられます：

- sqlstate の最初の 2 文字は、条件の一般的なクラスを示します。
- sqlstate の最後の 3 文字は、クラス内の特定のステータスを示します。

クライアントアプリケーションが SQL ステートメントの実行中に複数のエラー（または警告）に遭遇した場合、sqlstate には返された最後のコードが含まれます。

次の表に、sqlstate と sqlcode の値、および関連する条件のシンボル名とエラーの説明を示します。

sqlstate	sqlcode (非推奨)	記号名	説明
YE001	-12	ECPG_OUT_OF_MEMORY	仮想メモリが枯渇しています。

次のページに続く

表 1 – 前のページからの続き

sqlstate	sqlcode (非推奨)	記号名	説明
YE002	-200	ECPG_UNSUPPORTED	プリプロセッサが認識されないアイテムを生成しました。プリプロセッサとライブラリ間の非互換性を示している可能性があります。
07001、または 07002	-201	ECPG_TOO_MANY_ARGUMENTS	プログラムは、コマンドが予想するよりも多くの変数を指定します。
07001、または 07002	-202	ECPG_TOO_FEW_ARGUMENTS	プログラムが指定した変数がコマンドの予想よりも少ない。
21000	-203	ECPG_TOO_MANY_MATCHES	SQL コマンドは複数の行を返しましたが、ステートメントは単一の行を受け取る準備ができていました。
42804	-204	ECPG_INT_FORMAT	ホスト変数 (C コードで定義されている) は INT 型であり、選択されたデータは INT に変換できない型です。ecpglib は、strtol () 関数を使用して文字列値を数値形式に変換します。

次のページに続く

表 1 – 前のページからの続き

sqlstate	sqlcode (非推奨)	記号名	説明
42804	-205	ECPG_UINT_FORMAT	ホスト変数 (C コードで定義) は符号なし INT であり、選択されたデータは符号なし INT に変換できないタイプです。ecpglib は、strtol() 関数を使用して文字列値を数値形式に変換します。
42804	-206	ECPG_FLOAT_FORMAT	ホスト変数 (C コードで定義) は FLOAT 型であり、選択されたデータは FLOAT に変換できない型です。ecpglib は strtod() 関数を使用して、文字列値を数値形式に変換します。
42804	-211	ECPG_CONVERT_BOOL	ホスト変数 (C コードで定義) は BOOL 型であり、選択したデータを BOOL に保存することはできません。
YE002	-2-1	ECPG_EMPTY	サーバーに送信されたステートメントは空でした。

次のページに続く

表 1 – 前のページからの続き

sqlstate	sqlcode (非推奨)	記号名	説明
22002	-213	ECPG_MISSING_INDICATOR	サーバーから返された NULL 値に NULL インジケータ変数が指定されていません (クライアントアプリケーションが予期しない NULL 値を受信しました)。
42804	-214	ECPG_NO_ARRAY	サーバーは配列を返しました。対応するホスト変数は配列を格納できません。
42804	-215	ECPG_DATA_NOT_ARRAY	サーバーは、配列値を予期するホスト変数に配列ではない値を返しました。
08003	-220	ECPG_NO_CONN	クライアントアプリケーションは、存在しない接続を使用しようとしてしました。
YE002	-221	ECPG_NOT_CONN	クライアントアプリケーションは、割り当てられたが閉じられた接続を使用しようとしてしました。
26000	-230	ECPG_INVALID_STMT	ステートメントは準備されていません。
33000	-240	ECPG_UNKNOWN_DESCRIPTOR	指定された記述子が見つかりません。
07009	-241	ECPG_INVALID_DESCRIPTOR_INDEX	記述子のインデックスが範囲外です。

次のページに続く

表 1 – 前のページからの続き

sqlstate	sqlcode (非推奨)	記号名	説明
YE002	-242	ECPG_UNKNOWN_DESCRIPTOR_ITEM	クライアントアプリケーションが無効な記述子項目を要求しました (内部エラー)。
07006	-243	ECPG_VAR_NOT_NUMERIC	動的ステートメントが、非数値ホスト変数の数値を返しました。
07006	-244	ECPG_VAR_NOT_CHAR	動的 SQL ステートメントが CHAR 値を返しましたが、ホスト変数は CHAR ではありません。
/	-400	ECPG_PGSQL	サーバーがエラーメッセージを返しました。結果のメッセージにはエラーテキストが含まれます。
08007	-401	ECPG_TRANS	サーバーは、指定されたトランザクションを開始、コミット、またはロールバックできません。
08001	-402	ECPG_CONNECT	クライアントアプリケーションがデータベースに接続しようとして失敗しました。
02000	100	ECPG_NOT_FOUND	最後のコマンドが行を取得または処理しなかったか、カーソルの終わりに達しました。

6.2 EXEC SQL WHENEVER

EXEC SQL WHENEVER ディレクティブを使用して、ECPGPlus でコンパイルされたクライアントアプリケーションに簡単なエラー処理を実装します。ディレクティブの構文は次のとおりです。

```
EXEC SQL WHENEVER <condition> <action>;
```

このディレクティブは、エラー処理コードをプログラムに挿入するように ECPG コンパイラーに指示します。

このコードは、クライアントアプリケーションが特定の条件を検出した場合に、指定されたアクションを実行するようにクライアントアプリケーションに指示します。条件 は次のいずれかです。

SQLERROR

sqlca.sqlcode がゼロ未満の場合、SQLERROR 条件が存在します。

SQLWARNING

sqlca.sqlwarn[0] に 'W' が含まれる場合、SQLWARNING 条件が存在します。

NOT FOUND

sqlca.sqlcode が ECPG_NOT_FOUND の場合（クエリがデータを返さない場合）、NOT FOUND 条件が存在します。

前の条件のいずれかに遭遇した場合、クライアントアプリケーションが次の*アクション*のいずれかを実行するように指定できます。

CONTINUE

CONTINUE を指定して、現在の condition を無視して、クライアントアプリケーションに処理を続行するよう指示します。CONTINUE がデフォルトのアクションです。

DO CONTINUE

DO CONTINUE のアクションは、放出された C コードに CONTINUE ステートメントを生成します。条件に遭遇すると、ループ内の残りのコードをスキップし、次の反復を続行します。ループ内でのみ使用できます。

GOTO label

または

GO TO label

C の goto ステートメントを使用して、指定した label にジャンプします。

SQLPRINT

`sqlprint()` 関数を使用して、エラーメッセージを `stderr` (標準エラー) に出力します。 `sqlprint()` 関数は `sql_error` を出力し、その後に `sqlca.sqlerrm.sqlerrmc` の内容が続きます。

STOP

`exit(1)` を呼び出してエラーを通知し、プログラムを終了します。

DO BREAK

C の `break` ステートメントを実行します。このアクションは、ループまたは `switch` ステートメントで使
用します。

CALL name(args)

または

DO name(args)

`args` パラメーターで指定されたパラメーターを使用して、 `parameter` という名前で指定された C 関数
を呼び出します。

Example:

次のコードは、クライアントアプリケーションで警告が発生した場合にメッセージを出力し、エラーが発生した場
合にアプリケーションを中止します。

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;  
EXEC SQL WHENEVER SQLERROR STOP;
```

注釈: クライアントアプリケーションが上から下に*実行*できない場合でも、ECPGPlus コンパイラはプログラ
ムを上から下に処理します。コンパイラ指令は各行に順番に適用され、コンパイラが別の指令に遭遇するまで有効
です。プログラム内のフローの制御が上から下でない場合は、コンパイル中に誤って見落とされる可能性のあるプ
ログラムの部分にエラー処理ディレクティブを追加することを検討する必要があります。

第 7 章

参照先

以下のセクションでは、**ecpgPlus** 言語要素について説明します。

- C プリプロセッサディレクティブ
- サポートされている C データ型
- タイプコード
- SQLDA 構造
- ECPGPlus ステートメント

7.1 C プリプロセッサディレクティブ

ECPGPlusC プリプロセッサは、ECPGPlus を呼び出すモードに依存する 2 つの動作を強制します。

- PROC モード
- 非 PROC モード

Compiling in PROC Mode

PROC モードでは、ECPGPlus で次のことができます。

- EXEC SQL BEGIN/END DECLARE SECTION 以外のホスト変数を宣言します。
- ECPG と互換性のあるデータ型である限り、任意の C 変数をホスト変数として使用します。

PROC モードで (`-C PROC` キーワードを含めることで) ECPGPlus を呼び出すと、ECPG コンパイラは次の C プリプロセッサディレクティブを受け入れます。

```
#include  
#if expression  
#ifdef symbolName
```

(次のページに続く)

(前のページからの続き)

```
#ifndef symbolName
#else
#elif expression
#endif
#define symbolName expansion
#define symbolName([macro arguments]) expansion
#undef symbolName
#define(symbolName)
```

プリプロセッサディレクティブは、コンパイラが受け取るコードを有効化または指示するために使用されます。たとえば、次のコードサンプルを使用します。

```
#if HAVE_LONG_LONG == 1
#define BALANCE_TYPE long long
#else
#define BALANCE_TYPE double
#endif
...
BALANCE_TYPE customerBalance;
```

次のコマンドライン引数で ECPGPlus を起動する場合:

```
ecpg -C PROC -DHAVE_LONG_LONG=1
```

ECPGPlus は、フラグメント全体を（変更なしで）出力ファイルにコピーしますが、次のトークンのみを ECPG パーサーに送信します。

```
long long customerBalance;
```

一方、次のコマンドライン引数で ECPGPlus を呼び出した場合:

```
ecpg -C PROC -DHAVE_LONG_LONG=0
```

ECPG パーサーは次のトークンを受け取ります。

```
double customerBalance;
```

コードがプリプロセッサディレクティブを使用してコンパイラに送信されるコードをフィルタリングする場合、ECPG パーサーは処理されたトークンストリームのみを参照している間、完全なコードは元のコードに保持されます。

EXEC ディレクティブを使用して次のプリプロセッサディレクティブを実行するときに、互換性のある構文を使用することもできます。

```
EXEC ORACLE DEFINE
EXEC ORACLE UNDEF
EXEC ORACLE INCLUDE
EXEC ORACLE IFDEF
EXEC ORACLE IFNDEF
EXEC ORACLE ELIF
EXEC ORACLE ELSE
EXEC ORACLE ENDIF
EXEC ORACLE OPTION
```

たとえば、コードに次が含まれている場合:

```
EXEC ORACLE IFDEF HAVE_LONG_LONG;
#define BALANCE_TYPE long long
EXEC ORACLE ENDIF;
BALANCE_TYPE customerBalance;
```

次のコマンドライン引数で ECPGPlus を起動する場合:

```
ecpg -C PROC DEFINE=HAVE_LONG_LONG=1
```

ECPGPlus は、次のトークンを出力ファイルと ECPG パーサーに送信します。

```
long long customerBalance;
```

注釈: EXEC ORACLE プリプロセッサディレクティブは、ECPG コマンドラインで -C PROC を指定した場合にのみ機能します。

Using the SELECT_ERROR Precompiler Option

ECPGPlus を互換モードで使用する場合、SELECT_ERROR プリコンパイラオプションを使用して、ホスト変数が収容できるよりも多くの行を含む結果セットの処理方法をプログラムに指示できます。構文は次のとおりです。

```
SELECT_ERROR={YES|NO}
```

デフォルト値は YES です。結果セットがホスト変数の容量を超える場合、SELECT ステートメントはエラーメッセージを返します。NO を指定すると、SELECT ステートメントがホスト変数が対応できる数を超える行を返す場合にエラーメッセージを抑制するようにプログラムに指示します。

EXEC ORACLE OPTION ディレクティブとともに SELECT_ERROR を使用します。

Compiling in non-PROC Mode

-C PROC コマンドラインオプションを含めない場合:

- C プリプロセッサディレクティブは、変更せずに出力ファイルにコピーされます。
- EXEC SQL BEGIN/END DECLARE セクション内でホスト変数として使用する各 C 変数のタイプと名前を宣言する必要があります。

非 PROC モードで呼び出されると、ECPG は PostgreSQLCore ドキュメントで説明されている動作を実装します。

7.2 サポートされている C データ型

ECPGPlus アプリケーションは、SQL データ型 (SMALLINT、DOUBLE PRECISION、CHARACTER VARYING など) と C データ型 (short、double、varchar[n] など) の 2 つのデータ型セットを処理する必要があります。アプリケーションがサーバーからデータをフェッチすると、ECPGPlus は各 SQL データ型を、データが返される C 変数の型にマップします。

一般に、ECPGPlus はほとんどの SQL サーバータイプを同様の C タイプに変換できますが、すべての組み合わせが有効であるとは限りません。たとえば、ECPGPlus は SQL 文字値を C 整数値に変換しようとしませんが、SQL 文字値に数字以外の文字が含まれている場合、変換は（実行時に）失敗する場合があります。逆も同様です。アプリケーションがサーバーに値を送信すると、ECPGPlus は C データ型を必要な SQL 型に変換しようとしています。繰り返しますが、C 値を必要な SQL 型に変換できない場合、変換は（実行時に）失敗する可能性があります。

ECPGPlus は、任意の SQL タイプを C 文字値 (char[n] または varchar[n]) に変換できます。任意の SQL タイプを char[n] または varchar[n] に/から変換することは安全ですが、多くの場合、int、double、float などのより自然な C タイプを使用すると便利です。

サポートされている C データ型は次のとおりです。

- short
- int
- unsigned int
- long long int
- float
- double
- char[n+1]
- varchar[n+1]
- bool
- および typedef によって作成された同等のもの

C でサポートされている数値型と文字型に加えて、pgtypeslib ランタイムライブラリは、日付/時刻と正確な数値を処理するためのカスタムデータ型（およびこれらの型を操作する関数）を提供します。

- timestamp
- interval
- date
- decimal

- `numeric`

`pgtypeslib` が提供するデータ型を使用するには、適切なヘッダーファイルを `#include` する必要があります。

7.3 タイプコード

次の表には、*external* データ型の型コードが含まれています。外部データ型は、C ホスト変数の型を示すために使用されます。アプリケーションが値をパラメーターにバインドするか、バッファを SELECT リスト項目にバインドする場合、対応する SQLDA 記述子 (descriptor->T[column]) のタイプコードを次の値のいずれかに設定する必要があります。

Type Code	Host Variable Type (C Data Type)
1、2、8、11、12、15、23、24、91、94、95、96、97	char[]
3	int
4、7、21	浮く
5、6	ヌル終了文字列 (char[length+1])
9	varchar
22	ダブル
68	符号なし整数

次の表には、*internal* データ型の型コードが含まれています。内部型コードは、データベースに存在する値の型を示すために使用されます。DESCRIBE SELECT LIST ステートメントは、次の値を使用してデータ型配列 (descriptor->T[column]) に入力します。

Internal Type Code	Server Type
1	VARCHAR2
2	NUMBER
8	長い
11	ROWID
12	DATE
23	RAW
24	ロングロウ
96	CHAR
100	バイナリフロート
101	バイナリダブル
104	ウルイド
187	タイムスタンプ
188	タイムスタンプ W/タイムゾーン
189	間隔月
190	2 日目から 2 日目
232	TIMESTAMPLOCAL_TZ

7.4 SQLDA 構造

OracleDynamicSQL メソッド 4 は、SQLDA データ構造を使用して、動的 SQL ステートメントのデータとメタデータを保持します。SQLDA 構造体は、動的ステートメントのテキストまたは動的ステートメントの結果セットにあるパラメーターマーカーに対応する入力パラメーターのセットを記述できます。SQLDA 構造のレイアウトは次のとおりです。

```
struct SQLDA
{
    int      N; /* Number of entries          */
    char  **V; /* Variables                          */
    int     *L; /* Variable lengths                     */
    short  *T; /* Variable types                       */
    short **I; /* Indicators                           */
    int     F; /* Count of variables discovered by DESCRIBE */
    char  **S; /* Variable names                       */
    short *M; /* Variable name maximum lengths        */
    short *C; /* Variable name actual lengths         */
    char  **X; /* Indicator names                      */
    short *Y; /* Indicator name maximum lengths       */
    short *Z; /* Indicator name actual lengths         */
};
```

Parameters

N – maximum number of entries

N 構造体のメンバーには、SQLDA で記述できるエントリの最大数が含まれます。このメンバーは、SQLDA 構造を割り当てるときに `sqlald()` 関数によって設定されます。OPEN または FETCH ステートメントで記述子を使用する前に、「実際の」数の値に N を設定する必要があります。

V – data values

V 構造体のメンバーは、データ値の配列へのポインターです。

- SELECT リスト記述子の場合、V は FETCH ステートメントによって返される値の配列を指します（配列の各メンバーは結果セットの列に対応します）。
- バインド記述子の場合、V はパラメーター値の配列を指します（記述子を使用するカーソルを開く前に、この配列に値を入力する必要があります）。

アプリケーションは、各値を保持するために必要なスペースを割り当てる必要があります。SELECT リスト値にスペースを割り当てる方法の例については、[displayResultSet](#) 関数を参照してください。

L – length of each data value

L 構造体のメンバーは、長さの配列へのポインターです。この配列の各メンバーは、V 配列の対応するメンバーで使用可能なメモリの量を示す必要があります。たとえば、V[5] が 20 バイトの NULL 終了文字列を

保持するのに十分な大きさのバッファを指す場合、`L[5]` には値 21（文字列内の文字に 20 バイトと NULL 終了文字に 1 バイト）が含まれている必要があります。アプリケーションは、`L` 配列の各メンバーを設定する必要があります。

T - data types

`T` 構造体のメンバーは、記述子で記述された各列（またはパラメーター）に対応するデータ型の配列を指します。

- バインド記述子の場合、`T` 配列の各メンバーを設定して、`ECPGPlus` に各パラメーターのデータ型を通知する必要があります。
- `SELECT` リスト記述子の場合、`DESCRIBE SELECT LIST` ステートメントは `T` 配列の各メンバーを設定して、対応する列にあるデータのタイプを反映します。

`FETCH` ステートメントを実行する前に「10」配列のメンバーを変更して、`ECPGPlus` に対応する値を特定のデータ型に変換させることができます。たとえば、`DESCRIBE SELECT LIST` ステートメントが特定の列のタイプが `DATE` であることを示している場合、対応する「10」メンバーを変更して、次の `FETCH` ステートメントが `NULL` で終わる形式でその値を返すように要求できます文字列。「10」配列の各メンバーは、数値タイプコードです（タイプコードのリストについては、「タイプコード<# type-codes>‘_」を参照してください）。`DESCRIBE SELECT LIST` ステートメントで返されるタイプコードは、`FETCH` ステートメントで予期されるタイプコードとは異なります。`DESCRIBE SELECT LIST` ステートメントの実行後、「10」の各メンバーは、対応する列が `NULL` 可能かどうかを示すデータ型*および* フラグをエンコードします。「12」関数を使用して、`T` 配列のメンバーから型コードと `NULL` 可能フラグを抽出できます。「12」関数のシグネチャは次のとおりです。

```
void sqlnul(unsigned short *valType,
            unsigned short *typeCode,
            int             *isNull)
```

たとえば、`results` という名前の記述子の 3 番目の列の型コードと `NULL` 可能フラグを見つけるには、次のように `sqlnul()` を呼び出します。

```
sqlnul(&results->T[2], &typeCode, &isNull);
```

I - indicator variables

`I` 構造体のメンバーは、インジケータ変数の配列を指します。この配列は、アプリケーションが `sqlalld()` 関数を呼び出して記述子を割り当てるときに割り当てられます。

- `SELECT` リスト記述子の場合、`I` 配列の各メンバーは、対応する列に `NULL`（ゼロ以外）または `NULL` 以外（ゼロ）の値が含まれているかどうかを示します。
- バインドパラメーターの場合、アプリケーションは `I` 配列の各メンバーを設定して、対応するパラメーター値が `NULL` かどうかを示す必要があります。

F - number of entries

F 構造体のメンバーは、記述子によって記述される値の数を示します（N 構造体のメンバーは、記述子によって記述できる値の*最大*数を示し、F は実際の値の数を示します）。F メンバーの値は、DESCRIBE ステートメントの実行時に ECPGPlus によって設定されます。F は正、負、またはゼロです。

- SELECT リスト記述子の場合、結果セットの列数が記述子で許可される値の最大数（N 構造で決定される）以下である場合、F には正の値が含まれます（メンバー）；ステートメントが SELECT ステートメントではない場合は 0、クエリが N 構造メンバーで許可されているよりも多くの列を返す場合は負の値。
- バインド記述子の場合、ステートメントで見つかったパラメーターの数が記述子で許可されている値の最大数（N 構造体メンバーによって決定される）以下の場合、F には正の数が含まれます。ステートメントにパラメーターマーカーが含まれていない場合は 0、ステートメントに N 構造体メンバーで許可されているよりも多くのパラメーターマーカーが含まれている場合は負の値。

F に正数（DESCRIBE ステートメントの実行後）が含まれる場合、その数は結果セットの列数（SELECT リスト記述子の場合）またはステートメントで見つかったパラメーターマーカーの数（バインド記述子の場合）。F に負の値が含まれる場合、F の絶対値を計算して、必要な値（またはパラメーターマーカー）の数を検出できます。たとえば、SELECT リストを記述した後に F に -24 が含まれている場合、クエリが 24 列を返すことがわかります。

S - column/parameter names

S 構造体のメンバーは、NULL で終了する文字列の配列を指します。

- SELECT リスト記述子の場合、DESCRIBE SELECT LIST ステートメントは、この配列の各メンバーに、結果セット内の対応する列の名前を設定します。
- バインド記述子の場合、DESCRIBE BIND VARIABLES ステートメントは、この配列の各メンバーを対応するバインド変数の名前に設定します。

このリリースでは、各バインド変数の名前はクエリ内のパラメーターマーカーの左から右の順序によって決定されます。たとえば、最初のパラメーターの名前は常に ?0 で、2 番目のパラメーターの名前は常に ?1 など。

M - maximum column/parameter name length

M 構造体のメンバーは、長さの配列を指します。この配列の各メンバーは、S 配列の対応するメンバーの*最大*長さを指定します（つまり、M[0] は S[0] で見つかった列/パラメーター名の最大長を指定します）。この配列には、sqlald() 関数が入力されます。

C - actual column/parameter name length

C 構造体のメンバーは、長さの配列を指します。この配列の各メンバーは、S 配列の対応するメンバーの*実際の*長さを指定します（つまり、C[0] は S[0] で見つかった列/パラメーター名の実際の長さを指定します）。

この配列には、DESCRIBE ステートメントが入力されます。

X - indicator variable names

X 構造体のメンバーは、NULL で終了する文字列の配列を指します-各文字列は、対応する値の NULL インジケータの名前を表します。

この配列は ECPGPlus では使用されませんが、Pro*C アプリケーションとの互換性のために提供されています。

Y - maximum indicator name length

Y 構造体のメンバーは、長さの配列を指します。この配列の各メンバーは、X 配列の対応するメンバーの* 最大* 長さを指定します（つまり、Y[0] は X[0] で見つかったインジケータ名の最大長を指定します）。

この配列は ECPGPlus では使用されませんが、Pro*C アプリケーションとの互換性のために提供されています。

Z - actual indicator name length

Z 構造体のメンバーは、長さの配列を指します。この配列の各メンバーは、X 配列の対応するメンバーの* 実際の* 長さを指定します（つまり、Z[0] は X[0] で見つかったインジケータ名の実際の長さを指定します）。

この配列は ECPGPlus では使用されませんが、Pro*C アプリケーションとの互換性のために提供されています。

7.5 ECPGPlus ステートメント

埋め込み SQL ステートメントを使用すると、クライアントアプリケーションはサーバーと対話できますが、埋め込みディレクティブは ECPGPlus コンパイラへの命令です。

AdvancedServerSQL ステートメントを C プログラムに埋め込むことができます。各ステートメントはキーワード EXEC SQL で始まり、セミコロン (;) で終わる必要があります。C プログラム内では、SQL ステートメントは次の形式を取ります。

```
EXEC SQL <sql_command_body>;
```

sql_command_body は標準の SQL ステートメントを表します。ホスト変数は、SQL ステートメントが値式を予期する任意の場所で使用できます。値式をホスト変数に置き換える方法の詳細については、[ホスト変数の宣言](#)を参照してください。

ECPGPlus は、一部のステートメントの PostgreSQL サーバー側構文を拡張します。これらのステートメントの構文の違いについては、次のリファレンスセクションで説明します。他の SQL コマンドでサポートされている構文の完全なリファレンスについては、次の場所にある **PostgreSQL*Core*Documentation** を参照してください。

<https://www.postgresql.org/docs/current/static/sql-commands.html>

7.5.1 記述子の割り当て

ALLOCATE DESCRIPTOR ステートメントを使用して、SQL 記述子領域を割り当てます。

```
EXEC SQL [FOR <array_size>] ALLOCATE DESCRIPTOR <descriptor_name>
    [WITH MAX <variable_count>];
```

どこで:

array_size は、記述子に割り当てる配列要素の数を指定する変数です。array_size は INTEGER 値またはホスト変数です。

descriptor_name は、記述子の名前または記述子の名前を含むホスト変数です。この値は、識別子、引用符付き文字列リテラル、またはホスト変数の形式をとることがあります。

variable_count は、記述子内のホスト変数の最大数を指定します。variable_count のデフォルト値は 100 です。

次のコードは、配列 (emp_array) として処理される emp_query という名前の記述子を割り当てます。

```
EXEC SQL FOR :emp_array ALLOCATE DESCRIPTOR emp_query;
```

7.5.2 電話する

CALL ステートメントを使用して、サーバー上のプロシージャまたは関数を呼び出します。CALL ステートメントは、AdvancedServer でのみ機能します。CALL ステートメントには 2 つの形式があります。最初の形式は*関数*を呼び出すために使用されます:

```
EXEC SQL CALL <program_name> '('[<actual_arguments>]')'
      INTO [[:<ret_variable>][: <ret_indicator>]];
```

2 番目の形式は、*procedure* を呼び出すために使用されます。

```
EXEC SQL CALL <program_name> '('[<actual_arguments>]')';
```

どこで:

program_name は、CALL ステートメントが呼び出すストアドプロシージャまたは関数の名前です。プログラム名は、スキーマ修飾またはパッケージ修飾（またはその両方）にすることができます。プログラムが存在するスキーマまたはパッケージを指定しない場合、ECPGPlus は search_path の値を使用してプログラムを見つけます。

actual_arguments は、プログラムに必要な引数のコンマ区切りリストを指定します。各 actual_argument は、プログラムが预期する正式な引数に対応することに注意してください。各引数は、IN パラメーター、OUT パラメーター、または INOUT パラメーターです。

:ret_variable は、プログラムが関数である場合に返される値を受け取るホスト変数を指定します。

:ret_indicator は、プログラムが関数の場合、返されるインジケータ値を受け取るホスト変数を指定します。

たとえば、次のステートメントは、:ename ホスト変数に含まれる値で get_job_desc 関数を呼び出し、:job ホスト変数にその関数によって返される値をキャプチャします。

```
EXEC SQL CALL get_job_desc(:ename)
      INTO :job;
```

7.5.3 閉じる

CLOSE ステートメントを使用してカーソルを閉じ、カーソルが現在使用しているリソースをすべて解放します。クライアントアプリケーションは、閉じたカーソルから行をフェッチできません。CLOSE ステートメントの構文は次のとおりです。

```
EXEC SQL CLOSE [<cursor_name>;
```

どこで:

cursor_name は、ステートメントによって閉じられたカーソルの名前です。カーソル名は、識別子またはホスト変数の形式をとることがあります。

OPEN ステートメントはカーソルを初期化します。初期化されると、カーソルが再度開かない限り、カーソル結果セットは変更されません。カーソルを再度開く前に CLOSE にする必要はありません。

emp_cursor という名前のカーソルを手動で閉じるには、次のコマンドを使用します。

```
EXEC SQL CLOSE emp_cursor;
```

アプリケーションが終了すると、カーソルは自動的に閉じられます。

7.5.4 コミット

COMMIT ステートメントを使用して現在のトランザクションを完了し、すべての変更を永続化し、他のユーザーに見えるようにします。構文は次のとおりです。

```
EXEC SQL [AT <database_name>] COMMIT [WORK]
        [COMMENT '<text'>] [COMMENT '<text'> RELEASE];
```

どこで:

database_name は、作業が存在するデータベース（またはデータベースの名前を含むホスト変数）の名前です。この値は、引用符で囲まれていない文字列リテラルまたはホスト変数の形式をとることがあります。

互換性のため、ECPGPlus は COMMENT 節をエラーなしで受け入れますが、COMMENT 節に含まれるテキストを*保存*しません。

コミットの実行後に現在の接続を閉じるには、RELEASE 句を含めます。

たとえば、次のコマンドは、dept データベースで実行されたすべての作業をコミットし、現在の接続を閉じます。

```
EXEC SQL AT dept COMMIT RELEASE;
```

デフォルトでは、クライアントアプリケーションが COMMIT ステートメントを実行する場合にのみステートメントがコミットされます。ECPGPlus を呼び出すときに -t オプションを含めて、クライアントアプリケーションが AUTOCOMMIT 機能呼び出しするように指定します。次のステートメントを使用して、クライアントアプリケーションの AUTOCOMMIT 機能を制御することもできます。

```
EXEC SQL SET AUTOCOMMIT TO ON
```

そして

```
EXEC SQL SET AUTOCOMMIT TO OFF
```

7.5.5 接続する

CONNECT ステートメントを使用して、データベースへの接続を確立します。CONNECT ステートメントには 2 つの形式があります。1 つは Oracle データベースと互換性があり、もう 1 つは互換性がありません。

最初の形式は、Oracle データベースと互換性があります。

```
EXEC SQL CONNECT
  {{:<user_name> IDENTIFIED BY :<password>} | :<connection_id>}
  [AT <database_name>]
  [USING :database_string]
  [ALTER AUTHORIZATION :new_password];
```

どこで:

user_name は、クライアントアプリケーションがサーバーへの接続に使用するロールを含むホスト変数です。

password は、そのロールに関連付けられたパスワードを含むホスト変数です。

connection_id は、データベースへの接続に使用されるスラッシュ区切りのユーザー名とパスワードを含むホスト変数です。

AT 句を含めて、接続が確立されるデータベースを指定します。database_name は、クライアントが接続しているデータベースの名前です。値を変数の形式で、または文字列リテラルとして指定します。

USING 句を含めて、接続が確立されるデータベースを識別するヌル終了文字列を含むホスト変数を指定します。

ALTER AUTHORIZATION 句は、構文の互換性のためにのみサポートされています。ECPGPlus は ALTER AUTHORIZATION 句を解析し、警告を報告します。

CONNECT ステートメントの最初の形式を使用して、クライアントアプリケーションは、接続ロールの ID を含む user というホスト変数と、関連付けられたパスワードを含む password というホスト変数との接続を確立します。次のコマンド:

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

クライアントアプリケーションは、CONNECT ステートメントの最初の形式を使用して、:connection_id という名前の単一のホスト変数を使用して接続を確立することもできます。次の例では、connection_id にはスラッシュで区切られたロール名とユーザーの関連パスワードが含まれています。

```
EXEC SQL CONNECT :connection_id;
```

CONNECT ステートメントの 2 番目の形式の構文は次のとおりです。

```
EXEC SQL CONNECT TO <database_name>
  [AS <connection_name>] [<credentials>];
```

credentials は次のいずれかです。

```
USER user_name password
USER user_name IDENTIFIED BY password
USER user_name USING password
```

2 番目の形式:

database_name は、クライアントが接続しているデータベースの名前または ID です。次のいずれかの形式で、変数または文字列リテラルとして database_name を指定します。

```
<database_name>[@<hostname>][:<port>]

tcp:postgresql://<hostname>[:<port>][/<database_name>][options]

unix:postgresql://<hostname>[:<port>][/<database_name>][options]
```

どこで:

hostname は、データベースが存在するサーバーの名前または IP アドレスです。

port は、サーバーがリッスンするポートです。

DEFAULT の値を指定して、デフォルトのロール名を使用してデフォルトのデータベースとの接続を確立することもできます。ターゲットデータベースとして DEFAULT を指定する場合、connection_name または credentials を含めないでください。

connection_name はデータベースへの接続の名前です。connection_name は、識別子の形式を取る必要があります（つまり、文字列リテラルまたは変数ではありません）。接続ごとに一意の connection_name を提供することにより、複数の接続を開くことができます。

接続の名前を指定しない場合、ecpglib は DEFAULT の名前を接続に割り当てます。接続は、EXEC SQL ステートメントで名前（DEFAULT）で参照できます。

CURRENT は、最近開かれた、または最新の SET CONNECTION TO ステートメントで言及された接続です。EXEC SQL ステートメントで名前が接続を参照しない場合、ECPG は接続の名前を CURRENT と見なします。

user_name は、AdvancedServer データベースとの接続を確立するために使用されるロールです。指定されたロールの特権は、接続を介して実行されるすべてのコマンドに適用されます。

password は、指定された user_name に関連付けられたパスワードです。

次のコードは、CONNECT ステートメントの 2 番目の形式を使用して、alice ロールと lsafepwd ロールに関連付けられたパスワードを使用して、edb という名前のデータベースへの接続を確立します。

```
EXEC SQL CONNECT TO edb AS acctg_conn
    USER 'alice' IDENTIFIED BY 'lsafepwd';
```

接続の名前は `acctg_conn` です。SET CONNECTION ステートメントを使用して接続名を変更するときに、接続名を使用できます。

7.5.6 記述子の割り当て解除

DEALLOCATE DESCRIPTOR ステートメントを使用して、割り当てられた記述子で使用中のメモリを解放します。ステートメントの構文は次のとおりです。

```
EXEC SQL DEALLOCATE DESCRIPTOR <descriptor_name>
```

どこで:

`descriptor_name` は記述子の名前です。この値は、引用符付き文字列リテラルまたはホスト変数の形式をとることがあります。

次の例では、`emp_query` という名前の記述子の割り当てを解除します。

```
EXEC SQL DEALLOCATE DESCRIPTOR emp_query;
```

7.5.7 宣言カーソル

DECLARE CURSOR ステートメントを使用して、カーソルを定義します。ステートメントの構文は次のとおりです。

```
EXEC SQL [AT <database_name>] DECLARE <cursor_name> CURSOR FOR
(<select_statement> | <statement_name>);
```

どこで:

`database_name` は、カーソルが動作するデータベースの名前です。この値は、識別子またはホスト変数の形式を取る場合があります。データベース名を指定しない場合、`database_name` のデフォルト値はデフォルトのデータベースです。

`cursor_name` はカーソルの名前です。

`select_statement` はカーソル結果セットを定義する SELECT ステートメントのテキストです。SELECT ステートメントに INTO 句を含めることはできません。

`statement_name` は、カーソル結果セットを定義する SQL ステートメントまたはブロックの名前です。

次の例では、`employees` という名前のカーソルを宣言しています。

```
EXEC SQL DECLARE employees CURSOR FOR
SELECT
```

(次のページに続く)

(前のページからの続き)

```
empno, ename, sal, comm
FROM
emp;
```

カーソルは、emp テーブルに格納されている各従業員レコードの従業員番号、従業員名、給与、および歩合を含む結果セットを生成します。

7.5.8 データベースの宣言

DECLARE DATABASE ステートメントを使用して、後続の SQL ステートメント（CONNECT ステートメントなど）で使用するデータベース識別子を宣言します。構文は次のとおりです。

```
EXEC SQL DECLARE <database_name> DATABASE;
```

どこで:

database_name はデータベースの名前を指定します。

次の例は、acctg データベースの識別子の宣言を示しています。

```
EXEC SQL DECLARE acctg DATABASE;
```

データベース識別子として acctg を宣言するコマンドを呼び出した後、acctg データベースは、接続の確立時または AT 句で名前参照できます。

この文は効果がなく、Pro*C との互換性のためにのみ提供されています。

7.5.9 宣言文

DECLARE STATEMENT ディレクティブを使用して、SQL ステートメントの識別子を宣言します。AdvancedServer は、DECLARE STATEMENT ディレクティブの 2 つのバージョンをサポートしています。

```
EXEC SQL [<database_name>] DECLARE <statement_name> STATEMENT;
```

そして

```
EXEC SQL DECLARE STATEMENT <statement_name>;
```

どこで:

statement_name は、ステートメントに関連付けられた識別子を指定します。

database_name はデータベースの名前を指定します。この値は、識別子または識別子を含むホスト変数の形式をとることができます。

DECLARE STATEMENT ディレクティブを含む一般的な使用シーケンスは次のとおりです。

```
EXEC SQL DECLARE give_raise STATEMENT;          // give_raise is now a statement
handle (not prepared)
EXEC SQL PREPARE give_raise FROM :stmtText; // give_raise is now associated
with a statement
EXEC SQL EXECUTE give_raise;
```

この文は効果がなく、Pro*C との互換性のためにのみ提供されています。

7.5.10 削除

DELETE ステートメントを使用して、テーブルから 1 つ以上の行を削除します。ECPGPlus DELETE ステートメントの構文は SQL ステートメントの構文と同じですが、パラメーターマーカーとホスト変数は、式が許可されている任意の場所で使用できます。構文は次のとおりです。

```
[FOR <exec_count>] DELETE FROM [ONLY] <table> [[AS] <alias>]
[USING <using_list>]
[WHERE <condition> | WHERE CURRENT OF <cursor_name>]
[{{RETURNING|RETURN} * | <output_expression> [[ AS] <output_name>]}
[, ...] INTO <host_variable_list> ]
```

どこで:

ステートメントを実行する回数を指定するには、FOR exec_count 句を含めます。この句は、VALUES 句が配列または配列へのポインタを参照する場合にのみ有効です。

table は既存のテーブルの名前（オプションでスキーマ修飾）です。処理を指定されたテーブルに制限するには、ONLY 句を含めます。ONLY 句を含めない場合、名前付きテーブルから継承するテーブルも処理されます。

alias はターゲットテーブルの代替名です。

using_list はテーブル式のリストであり、他のテーブルの列を WHERE 条件で表示できます。

WHERE 句を含めて、削除する行を指定します。ステートメントに WHERE 句を含めない場合、DELETE はテーブルからすべての行を削除し、テーブル定義をそのまま残します。

condition は、タイプ BOOLEAN の値を返す式、ホスト変数、またはパラメーターマーカーです。condition が true を返す行は削除されます。

cursor_name は WHERE CURRENT OF 句で使用するカーソルの名前です。削除される行は、このカーソルから最後にフェッチされた行になります。カーソルは、DELETE ステートメントのターゲットテーブルに対する非

グループ化クエリでなければなりません。ブール条件を含む DELETE ステートメントで WHERE CURRENT OF を指定することはできません。

RETURN/RETURNING 句は、各行が削除された後に DELETE コマンドによって返される output_expression または host_variable_list を指定します。

output_expression は計算され、各行が削除された後に DELETE コマンドによって返される式です。output_name は返される列の名前です。*****を含めると、すべての列が返されます。

host_variable_list は、ホスト変数とオプションのインジケータ変数のカンマ区切りリストです。各ホスト変数は、RETURNING 句から対応する値を受け取ります。

たとえば、次のステートメントは、sal 列にホスト変数 :max_sal: で指定された値より大きい値が含まれる emp テーブルからすべての行を削除します

```
DELETE FROM emp WHERE sal > :max_sal;
```

DELETE ステートメントの使用の詳細については、次の場所にある PostgreSQLCore ドキュメントを参照してください。

<https://www.postgresql.org/docs/current/static/sql-delete.html>

7.5.11 DESCRIBE

DESCRIBE ステートメントを使用して、準備されたステートメントに必要な入力値の数、または準備されたステートメントによって返される出力値の数を見つけます。DESCRIBE ステートメントは、アプリケーションの作成時に形状が不明な SQL ステートメントを分析するために使用されます。

DESCRIBE ステートメントは、SQLDA 記述子を取り込みます。SQL 記述子を作成するには、ALLOCATE DESCRIPTOR および DESCRIBE...DESCRIPTOR ステートメントを使用します。

```
EXEC SQL DESCRIBE BIND VARIABLES FOR <statement_name> INTO  
<descriptor>;
```

または

```
EXEC SQL DESCRIBE SELECT LIST FOR <statement_name> INTO <descriptor>;
```

どこで:

statement_name は、準備された SQL 文または PL/SQL ブロックに関連付けられた識別子です。

descriptor は、SQLDA* 型の C 変数の名前です。DESCRIBE ステートメントを実行する前に sqlald() を呼び出して（および記述子を初期化して）記述子のスペースを割り当てる必要があります。

DESCRIBE ステートメントの最初の形式を実行すると、ECPG は、指定された記述子に、ステートメントで*必要* になっている各入力変数の説明を入力します。たとえば、2 つの記述子がある場合:

```
SQLDA *query_values_in;
SQLDA *query_values_out;
```

emp テーブルから情報を返すクエリを準備できます。

```
EXEC SQL PREPARE get_emp FROM
    "SELECT ename, empno, sal FROM emp WHERE empno = ?";
```

このコマンドには、1 つの入力変数が必要です（パラメーターマーカー（?）用）。

```
EXEC SQL DESCRIBE BIND VARIABLES
    FOR get_emp INTO query_values_in;
```

このステートメントのバインド変数を記述した後、記述子を調べて、必要な変数の数と各変数のタイプを見つけることができます。

2 番目の形式を実行すると、ECPG は、指定された記述子に、ステートメントによって*返される* 各値の説明を取り込みます。たとえば、次のステートメントは 3 つの値を返します。

```
EXEC SQL DESCRIBE SELECT LIST
    FOR get_emp INTO query_values_out;
```

このステートメントの選択リストを記述した後、記述子を調べて、返された値の数と各値の名前とタイプを見つけることができます。

ステートメントを*実行* する前に、各入力値の変数と各出力値の変数をバインドする必要があります。入力値にバインドする変数は、ステートメントで使用される実際の値を指定します。出力値用にバインドする変数は、ステートメントの実行時に値を配置する場所を ECPGPlus に指示します。

これは、DESCRIBE DESCRIPTOR ステートメントに対する Pro*C 互換の代替構文です。

7.5.12 DESCRIBEDESCRIPTOR

DESCRIBE DESCRIPTOR ステートメントを使用して、SQL ステートメントに関する情報を取得し、その情報を SQL 記述子に保存します。DESCRIBE DESCRIPTOR を使用する前に、ALLOCATE DESCRIPTOR ステートメントで記述子を割り当てる必要があります。構文は次のとおりです。

```
EXEC SQL DESCRIBE [INPUT | OUTPUT] <statement_identifier>
    USING [SQL] DESCRIPTOR <descriptor_name>;
```

どこで:

statement_name は準備された SQL ステートメントの名前です。

descriptor_name は記述子の名前です。descriptor_name は、引用符で囲まれた文字列値または記述子の名前を含むホスト変数です。

INPUT 句を含めると、ECPGPlus は、指定された記述子に、ステートメントで*必須* になっている各入力変数の説明を入力します。

たとえば、2 つの記述子がある場合:

```
EXEC SQL ALLOCATE DESCRIPTOR query_values_in;  
EXEC SQL ALLOCATE DESCRIPTOR query_values_out;
```

emp テーブルから情報を返すクエリを準備できます。

```
EXEC SQL PREPARE get_emp FROM  
    "SELECT ename, empno, sal FROM emp WHERE empno = ?";
```

このコマンドには、1 つの入力変数が必要です (パラメーターマーカー (?) 用)。

```
EXEC SQL DESCRIBE INPUT get_emp USING 'query_values_in';
```

このステートメントのバインド変数を記述した後、記述子を調べて、必要な変数の数と各変数のタイプを見つけることができます。

INPUT 句を指定しない場合、DESCRIBE DESCRIPTOR は指定された記述子にステートメントによって返された値を入力します。

OUTPUT 句を含めると、ECPGPlus は指定された記述子に、ステートメントによって*返される* 各値の説明を入力します。

たとえば、次のステートメントは 3 つの値を返します。

```
EXEC SQL DESCRIBE OUTPUT FOR get_emp USING 'query_values_out';
```

このステートメントの選択リストを記述した後、記述子を調べて、返された値の数と各値の名前とタイプを見つけることができます。

7.5.13 切断

DISCONNECT ステートメントを使用して、サーバーへの接続を閉じます。構文は次のとおりです。

```
EXEC SQL DISCONNECT [<connection_name>] [CURRENT] [DEFAULT] [ALL];
```

どこで:

connection_name は、接続の確立に使用される CONNECT ステートメントで指定された接続名です。接続名を指定しない場合、現在の接続は閉じられます。

CURRENT キーワードを含めて、ECPGPlus が最近使用した接続を閉じるように指定します。

DEFAULT キーワードを含めて、ECPGPlus が DEFAULT という名前の接続を閉じるように指定します。接続を開くときに名前を指定しない場合、ECPGPlus は名前 DEFAULT を接続に割り当てます。

ALL キーワードを含めて、すべてのアクティブな接続を閉じるように ECPGPlus に指示します。

次の例では、hr データベースに接続する接続（hr_connection という名前）を作成し、接続から切断します。

```
/* client.pgc*/
int main()
{
    EXEC SQL CONNECT TO hr AS connection_name;
    EXEC SQL DISCONNECT connection_name;
    return(0);
}
```

7.5.14 実行

EXECUTE ステートメントを使用して、EXEC SQL PREPARE ステートメントを使用して事前に準備されたステートメントを実行します。構文は次のとおりです。

```
EXEC SQL [FOR <array_size>] EXECUTE <statement_name>
[USING {DESCRIPTOR <SQLDA_descriptor>
|: <host_variable> [[INDICATOR] :<indicator_variable>]}];
```

どこで:

array_size は整数値、または処理する行数を指定する整数値を含むホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

statement_name は、ステートメントの作成時にステートメントに割り当てられた名前を指定します（EXEC SQL PREPARE ステートメントを使用）。

準備されたステートメント内のパラメーターの値を提供するには、USING 句を含めます。

パラメーターに SQLDA 記述子値を提供するには、DESCRIPTOR SQLDA_descriptor 句を含めます。

host_variable（およびオプションの indicator_variable）を使用して、パラメータにユーザー指定の値を指定します。

次の例では、レコードを emp テーブルに挿入する準備済みステートメントを作成します。

```
EXEC SQL PREPARE add_emp (numeric, text, text, numeric) AS
    INSERT INTO emp VALUES($1, $2, $3, $4);
```

準備済みステートメントを呼び出すたびに、ステートメントの新しいパラメーター値を指定します。

```
EXEC SQL EXECUTE add_emp USING 8000, 'DAWSON', 'CLERK', 7788;
EXEC SQL EXECUTE add_emp USING 8001, 'EDWARDS', 'ANALYST', 7698;
```

7.5.15 記述子の実行

EXECUTE ステートメントを使用して、SQL 記述子を使用して、EXEC SQL PREPARE ステートメントによって事前に準備されたステートメントを実行します。構文は次のとおりです。

```
EXEC SQL [FOR <array_size>] EXECUTE <statement_identifier>
    [USING [SQL] DESCRIPTOR <descriptor_name>]
    [INTO [SQL] DESCRIPTOR <descriptor_name>];
```

どこで:

array_size は整数値、または処理する行数を指定する整数値を含むホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

statement_identifier は、EXEC SQL PREPARE ステートメントとともにステートメントに割り当てられた識別子を指定します。

USING 句を含めて、準備されたステートメントに必要な入力パラメーターの値を指定します。

INTO 句を含めて、EXECUTE ステートメントが準備済みステートメントによって返された結果を書き込む記述子を指定します。

descriptor_name は、記述子の名前（単一引用符で囲まれた文字列リテラルとして）、または記述子の名前を含むホスト変数を指定します。

次の例では、記述子 stmtText: に含まれる値を使用して、準備済みステートメント give_raise を実行します

```
EXEC SQL PREPARE give_raise FROM :stmtText;
EXEC SQL EXECUTE give_raise USING DESCRIPTOR :stmtText;
```

7.5.16 EXECUTE...ENDEXEC

EXECUTE...END-EXEC ステートメントを使用して、匿名ブロックをクライアントアプリケーションに埋め込みます。構文は次のとおりです。

```
EXEC SQL [AT <database_name>] EXECUTE <anonymous_block> END-EXEC;
```

どこで:

database_name は、データベース識別子またはデータベース識別子を含むホスト変数です。AT 句を省略すると、ステートメントは現在のデフォルトデータベースで実行されます。

anonymous_block は、PL/pgSQL または SPL ステートメントと宣言のインラインシーケンスです。ブロック内にホスト変数とオプションのインジケータ変数を含めることができます。そのような各変数は IN/OUT 値として扱われます。

次の例では、匿名ブロックを実行します。

```
EXEC SQL EXECUTE

BEGIN
    IF (current_user = :admin_user_name) THEN
        DBMS_OUTPUT.PUT_LINE('You are an administrator');
    END IF;
END-EXEC;
```

注釈: EXECUTE...END EXEC ステートメントは AdvancedServer でのみサポートされています。

7.5.17 即時実行

EXECUTE IMMEDIATE ステートメントを使用して、SQL コマンドを含む文字列を実行します。構文は次のとおりです。

```
EXEC SQL [AT <database_name>] EXECUTE IMMEDIATE <command_text>;
```

どこで:

database_name は、データベース識別子またはデータベース識別子を含むホスト変数です。AT 句を省略すると、ステートメントは現在のデフォルトデータベースで実行されます。

command_text は EXECUTE IMMEDIATE ステートメントで実行されるコマンドです。

この動的 SQL ステートメントは、SQL ステートメントのテキストがわからない場合（つまり、クライアントアプリケーションを作成する場合）に役立ちます。たとえば、クライアントアプリケーションは、実行するステートメントを（信頼できる）ユーザーに要求する場合があります。ユーザーがステートメントのテキストを文字列値として提供した後、ステートメントは EXECUTE IMMEDIATE コマンドで実行されます。

ステートメントテキストには、ホスト変数への参照を含めることはできません。ステートメントにパラメーター

マーカーが含まれているか、1 つ以上の値を返す場合は、PREPARE および DESCRIBE ステートメントを使用する必要があります。

次の例は、:command_text ホスト変数に含まれるコマンドを実行します。

```
EXEC SQL EXECUTE IMMEDIATE :command_text;
```

7.5.18 FETCH

FETCH ステートメントを使用して、カーソルから **SQLDA** 記述子またはホスト変数のターゲットリストに行を返します。FETCH ステートメントを使用してカーソルから情報を取得する前に、DECLARE および OPEN ステートメントを使用してカーソルを準備する必要があります。ステートメントの構文は次のとおりです。

```
EXEC SQL [FOR <array_size>] FETCH <cursor>
    { USING DESCRIPTOR <SQLDA_descriptor> } || { INTO <target_list> };
```

どこで:

array_size は整数値またはフェッチする行数を指定する整数値を含むホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

cursor は、行のフェッチ元のカーソルの名前、またはカーソルの名前を含むホスト変数です。

USING 句を含めると、FETCH ステートメントは、指定された **SQLDA** 記述子にサーバーから返された値を取り込みます。

INTO 句を含めると、FETCH ステートメントは target_list で指定されたホスト変数（およびオプションのインジケータ変数）を設定します。

次のコードは、emp テーブルから employee number、name、salary を取得する employees という名前のカーソルを宣言しています。

```
EXEC SQL DECLARE employees CURSOR
    SELECT empno, ename, esal FROM emp
EXEC SQL OPEN emp_cursor
EXEC SQL FETCH emp_cursor INTO :emp_no, :emp_name, :emp_sal;
```

7.5.19 FETCHDESCRIPTOR

FETCH DESCRIPTOR ステートメントを使用して、カーソルから **SQL** 記述子に行を取得します。構文は次のとおりです。

```
EXEC SQL [FOR <array_size>] FETCH <cursor>
    INTO [SQL] DESCRIPTOR <descriptor_name>;
```

どこで:

`array_size` は整数値またはフェッチする行数を指定する整数値を含むホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

`cursor` は、行のフェッチ元のカーソルの名前、またはカーソルの名前を含むホスト変数です。クライアントは、FETCH DESCRIPTOR ステートメントを呼び出す前に、カーソルを DECLARE と OPEN にする必要があります。

INTO 句を含めて、EXECUTE ステートメントが準備されたステートメントによって返された結果を書き込む SQL 記述子を指定します。descriptor_name は、記述子の名前（単一引用符で囲まれた文字列リテラルとして）、または記述子の名前を含むホスト変数を指定します。使用する前に、ALLOCATE DESCRIPTOR ステートメントを使用して記述子を割り当てる必要があります。

次の例では、結果セットの特定の行の説明と値を保持する row_desc という名前の記述子を割り当てます。次に、結果セットの行をループする前に、準備されたステートメント（my_cursor）のカーソルを宣言して開き、FETCH を使用してカーソルから次の行を記述子に取得します。

```
EXEC SQL ALLOCATE DESCRIPTOR 'row_desc';
EXEC SQL DECLARE my_cursor CURSOR FOR query;
EXEC SQL OPEN my_cursor;

for( row = 0; ; row++ )
{
    EXEC SQL BEGIN DECLARE SECTION;
        int col;
EXEC SQL END DECLARE SECTION;
EXEC SQL FETCH my_cursor INTO SQL DESCRIPTOR 'row_desc';
```

7.5.20 記述子を取得する

GET DESCRIPTOR ステートメントを使用して、記述子から情報を取得します。GET DESCRIPTOR ステートメントには 2 つの形式があります。最初の形式は、記述子内の値（または列）の数を返します。

```
EXEC SQL GET DESCRIPTOR <descriptor_name>
    :<host_variable> = COUNT;
```

2 番目の形式は、特定の値（VALUE column_number 句で指定された）に関する情報を返します。

```
EXEC SQL [FOR <array_size>] GET DESCRIPTOR <descriptor_name>
    VALUE <column_number> {:<host_variable> = <descriptor_item> {','}};
```

どこで:

`array_size` は整数値、または処理する行数を指定する整数値を含むホスト変数です。`array_size` を指定する場合、`host_variable` はそのサイズの配列でなければなりません。たとえば、`array_size` が 10 の場合、

:host_variable は host_variables の 10 メンバー配列でなければなりません。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

descriptor_name は、記述子の名前（単一引用符で囲まれた文字列リテラルとして）、または記述子の名前を含むホスト変数を指定します。

VALUE 句を含めて、記述子から取得した情報を指定します。

column_number は、記述子内の変数の位置を示します。

host_variable は、アイテムの値を受け取るホスト変数の名前を指定します。

descriptor_item は、取得した記述子項目のタイプを指定します。

ECPGPlus は、次の descriptor_item タイプを実装します。

- TYPE
- LENGTH
- OCTET_LENGTH
- RETURNED_LENGTH
- RETURNED_OCTET_LENGTH
- PRECISION
- SCALE
- NULLABLE
- INDICATOR
- DATA
- NAME

次のコードは、GET DESCRIPTOR ステートメントを使用して、ユーザーが指定した文字列に入力された列の数を取得する方法を示しています。

```
EXEC SQL ALLOCATE DESCRIPTOR parse_desc;  
EXEC SQL PREPARE query FROM :stmt;  
EXEC SQL DESCRIBE query INTO SQL DESCRIPTOR parse_desc;  
EXEC SQL GET DESCRIPTOR parse_desc :col_count = COUNT;
```

この例では、PREPARE ステートメントを使用してユーザー (:stmt) から提供された文字列を構文チェックする前に、SQL 記述子 (parse_desc という名前) を割り当てます。DESCRIBE ステートメントは、ユーザーが指定した文字列を記述子 parse_desc に移動します。EXEC SQL GET DESCRIPTOR を呼び出すと、記述子に問い合わせ、結果セット内の (:col_count) 列の数を検出します。

7.5.21 挿入

INSERT ステートメントを使用して、1 つ以上の行をテーブルに追加します。ECPGPlus INSERT ステートメントの構文は SQL ステートメントの構文と同じですが、パラメーターマーカーとホスト変数は、値が許可されている任意の場所で使用できます。構文は次のとおりです。

```
[FOR <exec_count>] INSERT INTO <table> [(<column> [, ...])]
{DEFAULT VALUES |
  VALUES ({<expression> | DEFAULT} [, ...])[, ...] | <query>}
[RETURNING * | <output_expression> [[ AS ] <output_name>] [, ...]]
```

どこで:

ステートメントを実行する回数を指定するには、FOR exec_count 句を含めます。この句は、VALUES 句が配列または配列へのポインタを参照する場合にのみ有効です。

table は、既存のテーブルの（オプションでスキーマ修飾された）名前を指定します。

column はテーブル内の列の名前です。列名は、サブフィールド名または配列添え字で修飾できます。すべての列にデフォルト値を使用するには、DEFAULT VALUES 句を指定します。

expression は、対応する列に割り当てられる式、値、ホスト変数、またはパラメーターマーカーです。DEFAULT を指定して、対応する列にデフォルト値を入力します。

query は、挿入する行を提供する SELECT ステートメントを指定します。

output_expression は、各行が挿入された後に INSERT コマンドによって計算され、返される式です。式は、テーブル内の任意の列を参照できます。*****を指定すると、挿入された行のすべての列が返されます。

output_name は、返される列に使用する名前を指定します。

次の例では、employees テーブルに行を追加します。

```
INSERT INTO emp (empno, ename, job, hiredate)
  VALUES ('8400', :ename, 'CLERK', '2011-10-31');
```

注釈: INSERT ステートメントは、ホスト変数 (:ename) を使用して ename 列の値を指定します。

INSERT ステートメントの使用の詳細については、次の場所にある PostgreSQLCore ドキュメントを参照してください。

<https://www.postgresql.org/docs/current/static/sql-insert.html>

7.5.22 開く

OPEN ステートメントを使用して、カーソルを開きます。構文は次のとおりです。

```
EXEC SQL [FOR <array_size>] OPEN <cursor> [USING <parameters>];
```

parameters は次のいずれかです。

```
DESCRIPTOR <SQLDA_descriptor>
```

または

```
<host_variable> [ [ INDICATOR ] <indicator_variable>, ... ]
```

どこで:

array_*size は整数値またはフェッチする行数を指定する整数値を含むホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

cursor は、開かれているカーソルの名前です。

parameters は DESCRIPTOR SQLDA_descriptor、またはカーソルを初期化する host variables（およびオプションの indicator variables）のコンマ区切りリストです。SQLDA_descriptor を指定する場合、記述子は DESCRIBE ステートメントで初期化する必要があります。

OPEN ステートメントは、parameters で指定された値を使用してカーソルを初期化します。初期化されると、カーソルを閉じて再度開かない限り、カーソル結果セットは変更されません。アプリケーションが終了すると、カーソルは自動的に閉じられます。

次の例では、employees という名前のカーソルを宣言し、emp テーブルを照会して、ユーザーが指定した値と一致する名前を持つ従業員の employee number、name、salary、commission を返しますホスト変数 :emp_name)。

```
EXEC SQL DECLARE employees CURSOR FOR
  SELECT
    empno, ename, sal, comm
  FROM
    emp
  WHERE ename = :emp_name;
EXEC SQL OPEN employees;
...
```

カーソルを宣言した後、この例では OPEN ステートメントを使用して、カーソルの内容をクライアントアプリケーションで使えるようにします。

7.5.23 記述子を開く

OPEN DESCRIPTOR ステートメントを使用して、SQL 記述子でカーソルを開きます。構文は次のとおりです。

```
EXEC SQL [FOR <array_size>] OPEN <cursor>
    [USING [SQL] DESCRIPTOR <descriptor_name>]
    [INTO [SQL] DESCRIPTOR <descriptor_name>;
```

どこで:

array_size は整数値またはフェッチする行数を指定する整数値を含むホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

cursor は、開かれているカーソルの名前です。

descriptor_name は、SQL 記述子の名前（単一引用符で囲まれた文字列リテラルの形式）またはカーソルを初期化するクエリを含む SQL 記述子の名前を含むホスト変数を指定します。

たとえば、次のステートメントは、ホスト変数 :employees を使用してカーソル（emp_cursor という名前）を開きます。

```
EXEC SQL OPEN emp_cursor USING DESCRIPTOR :employees;
```

7.5.24 準備する

準備済みステートメントは、クライアントアプリケーションがタスクを複数回実行する必要がある場合に役立ちます。ステートメントは、ステートメントが実行されるたびにではなく、1 回だけ解析、作成、計画され、繰り返しの処理時間が節約されます。

PREPARE ステートメントを使用して、実行する SQL ステートメントまたは PL/pgSQL ブロックを準備します。このステートメントには 2 つの形式があります。最初の形式は次のとおりです。

```
EXEC SQL [AT <database_name>] PREPARE <statement_name>
    FROM <sql_statement>;
```

2 番目の形式は次のとおりです。

```
EXEC SQL [AT <database_name>] PREPARE <statement_name>
    AS <sql_statement>;
```

どこで:

database_name は、データベース識別子またはステートメントが実行されるデータベース識別子を含むホスト変数です。AT 句を省略すると、ステートメントは現在のデフォルトデータベースに対して実行されます。

statement_name は、準備された SQL 文または PL/SQL ブロックに関連付けられた識別子です。

sql_statement は、SELECT ステートメント、SQL ステートメントのテキストを含む単一引用符で囲まれた文字列リテラルまたはホスト変数の形式をとることがあります。

準備されたステートメントに変数を含めるには、ステートメントを PREPARE にすると変化する可能性のあるステートメント値をプレースホルダー（\$1, \$2, \$3 など）に置き換えます。ステートメントを EXECUTE にするとき、各パラメーターの値を指定します。値は、プレースホルダーを置き換える順序で提供する必要があります。

次の例では、レコードを emp テーブルに挿入する準備済みステートメント（add_emp という名前）を作成します。

```
EXEC SQL PREPARE add_emp (int, text, text, numeric) AS
    INSERT INTO emp VALUES($1, $2, $3, $4);
```

ステートメントを呼び出すたびに、ステートメントの新しいパラメーター値を指定します。

```
EXEC SQL EXECUTE add_emp(8003, 'Davis', 'CLERK', 2000.00);
EXEC SQL EXECUTE add_emp(8004, 'Myer', 'CLERK', 2000.00);
```

注釈: クライアントアプリケーションは、ステートメントが実行される各セッション内で PREPARE ステートメントを発行する必要があります。準備済みステートメントは、現在のセッションの間のみ持続します。

7.5.25 ロールバック

ROLLBACK ステートメントを使用して、現在のトランザクションを中止し、トランザクションによって行われた更新を破棄します。構文は次のとおりです。

```
EXEC SQL [AT <database_name>] ROLLBACK [WORK]
[ { TO [SAVEPOINT] <savepoint> } | RELEASE ]
```

どこで:

database_name は、データベース識別子またはステートメントが実行されるデータベース識別子を含むホスト変数です。AT 句を省略すると、ステートメントは現在のデフォルトデータベースに対して実行されます。

指定された savepoint の後に実行されたコマンドを中止するには、TO 句を含めます。SAVEPOINT ステートメントを使用して savepoint を定義します。TO 句を省略すると、ROLLBACK ステートメントはトランザクションを中止し、すべての更新を破棄します。

アプリケーションに EXEC SQL COMMIT RELEASE を実行させ、接続を閉じるには、RELEASE 句を含めます。

次のステートメントを使用して、完全なトランザクションをロールバックします。

```
EXEC SQL ROLLBACK;
```

このステートメントを呼び出すと、トランザクションが中止され、すべての変更が取り消され、セーブポイントが消去され、すべてのトランザクションロックが解除されます。セーブポイントを含める場合（次の例では `my_savepoint`）:

```
EXEC SQL ROLLBACK TO SAVEPOINT my_savepoint;
```

`my_savepoint` の後に発生したトランザクションの部分のみがロールバックされます。`my_savepoint` は保持されますが、`my_savepoint` の後に作成されたセーブポイントはすべて消去されます。

指定されたセーブポイントにロールバックすると、セーブポイントの後に取得されたすべてのロックが解放されます。

7.5.26 セーブポイント

SAVEPOINT ステートメントを使用して `savepoint` を定義します。セーブポイントは、トランザクション内のマーカーです。ROLLBACK ステートメントを使用して、現在のトランザクションを中止し、サーバーの状態を指定したセーブポイントの前の状態に戻すことができます。SAVEPOINT ステートメントの構文は次のとおりです。

```
EXEC SQL [AT <database_name>] SAVEPOINT <savepoint_name>
```

どこで:

`database_name` は、セーブポイントが存在するデータベース識別子を含むデータベース識別子またはホスト変数です。AT 句を省略すると、ステートメントは現在のデフォルトデータベースに対して実行されます。

`savepoint_name` はセーブポイントの名前です。`savepoint_name` を再利用すると、元のセーブポイントは破棄されます。

セーブポイントは、トランザクションブロック内でのみ確立できます。トランザクションブロックには、複数のセーブポイントが含まれる場合があります。

`my_savepoint` という名前のセーブポイントを作成するには、次のステートメントを含めます。

```
EXEC SQL SAVEPOINT my_savepoint;
```

7.5.27 選択

ECPGPlus は、INTO `host_variables` 節を提供することで SQL SELECT ステートメントのサポートを拡張します。この句を使用すると、指定された情報を AdvancedServer データベースからホスト変数に選択できます。SELECT ステートメントの構文は次のとおりです。

```
EXEC SQL [AT <database_name>]
SELECT
  [ <hint> ]
  [ ALL | DISTINCT [ ON( <expression>, ...) ] ]
  select_list INTO <host_variables>
  [ FROM from_item [, from_item ]...]
  [ WHERE condition ]
  [ hierarchical_query_clause ]
  [ GROUP BY expression [, ...]]
  [ HAVING condition ]
  [ { UNION [ ALL ] | INTERSECT | MINUS } (subquery) ]
  [ ORDER BY expression> [order_by_options]]
  [ LIMIT { count | ALL } ]
  [ OFFSET start [ ROW | ROWS ] ]
  [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
  [ FOR { UPDATE | SHARE } [OF table_name [, ...]] [NOWAIT ] [...]]
```

どこで:

database_name は、テーブルが存在するデータベースの名前（またはデータベースの名前を含むホスト変数）です。この値は、引用符で囲まれていない文字列リテラルまたはホスト変数の形式をとることがあります。

host_variables は SELECT ステートメントによって入力されるホスト変数のリストです。SELECT ステートメントが複数の行を返す場合、host_variables は配列でなければなりません。

ECPGPlus は、次の場所にある PostgreSQLCore ドキュメントに記載されている SQL SELECT ステートメントの追加句のサポートを提供します。

<https://www.postgresql.org/docs/current/static/sql-select.html>

INTO host_variables 句を使用するには、SELECT ステートメントを指定するときに定義済みのホスト変数の名前を含めます。たとえば、次の SELECT ステートメントは、:emp_name および :emp_sal ホスト変数に employee names および salaries のリストを取り込みます。

```
EXEC SQL SELECT ename, sal
  INTO :emp_name, :emp_sal
  FROM emp
  WHERE empno = 7988;
```

拡張された SELECT ステートメントでは、値が許可される句にパラメーターマーカー（疑問符）を含めることもできます。たとえば、次のクエリには WHERE 句にパラメーターマーカーが含まれています。

```
SELECT * FROM emp WHERE dept_no = ?;
```

この SELECT ステートメントを使用すると、実行時に dept_no パラメーターマーカーに値を指定できます。

7.5.28 接続の設定

特定のクライアントアプリケーションで複数の接続が必要になる可能性がある（少なくとも）3つの理由があります。

- ステートメントごとに異なる特権が必要な場合があります。
- 同じクライアント内の複数のデータベースと対話する必要がある場合があります。
- （クライアントアプリケーション内で）複数の実行スレッドが同時に接続を共有することはできません。

SET CONNECTION ステートメントの構文は次のとおりです。

```
EXEC SQL SET CONNECTION <connection_name>;
```

どこで:

connection_name はデータベースへの接続の名前です。

SET CONNECTION ステートメントを使用するには、CONNECT ステートメントの2番目の形式を使用してデータベースへの接続を開く必要があります。AS 句を含めて connection_name を指定します。

デフォルトでは、現在のスレッドは現在の接続を使用します。SET CONNECTION ステートメントを使用して、使用する現在のスレッドのデフォルト接続を指定します。デフォルトの接続は、接続名を明示的に指定しない EXEC SQL ステートメントを実行する場合にのみ使用されます。たとえば、次のステートメントでは AT connection_name 句が含まれていないため、デフォルトの接続が使用されます。:

```
EXEC SQL DELETE FROM emp;
```

このステートメントは、AT connection_name 句を使用して接続名を指定するため、デフォルトの接続を使用しません。

```
EXEC SQL AT acctg_conn DELETE FROM emp;
```

たとえば、複数の接続を作成および維持するクライアントアプリケーション（など）:

```
EXEC SQL CONNECT TO edb AS acctg_conn
  USER 'alice' IDENTIFIED BY 'acctpwd';
```

そして

```
EXEC SQL CONNECT TO edb AS hr_conn
  USER 'bob' IDENTIFIED BY 'hrpwd';
```

SET CONNECTION ステートメントを使用して接続間で変更できます。

```
SET CONNECTION acctg_conn;
```

または

```
SET CONNECTION hr_conn;
```

サーバーは、接続クライアントが使用できる特権を決定するときに、接続に関連付けられた特権を使用します。acctg_conn 接続を使用する場合、クライアントは alice というロールに関連付けられた特権を持ちます。hr_conn を使用して接続すると、クライアントは bob に関連付けられた特権を持ちます。

7.5.29 記述子の設定

SET DESCRIPTOR ステートメントを使用して、ホスト変数または整数値の形式でクライアントアプリケーションから提供される情報を使用して、記述子領域に値を割り当てます。ステートメントには 2 つの形式があります。最初の形式は次のとおりです。

```
EXEC SQL [FOR <array_size>] SET DESCRIPTOR <descriptor_name>  
    VALUE <column_number> <descriptor_item> = <host_variable>;
```

2 番目の形式は次のとおりです。

```
EXEC SQL [FOR <array_size>] SET DESCRIPTOR <descriptor_name>  
    COUNT = integer;
```

どこで:

array_size は整数値またはフェッチする行数を指定する整数値を含むホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

descriptor_name は、記述子の名前（単一引用符で囲まれた文字列リテラルとして）、または記述子の名前を含むホスト変数を指定します。

VALUE 句を含めて、記述子に格納されている情報を記述します。

column_number は、記述子内の変数の位置を示します。

descriptor_item は、記述子項目のタイプを指定します。

host_variable は、アイテムの値を含むホスト変数の名前を指定します。

ECPGPlus は、次の descriptor_item タイプを実装します。

- TYPE
- LENGTH

- [REF] INDICATOR
- [REF] DATA
- [REF] RETURNED LENGTH

たとえば、クライアントアプリケーションは、動的に作成されたクエリの入力をユーザーに求める場合があります。

```
query_text = promptUser("Enter a query");
```

動的に作成されたクエリを実行するには、最初にクエリを `prepare` にし（クエリの構文を解析および検証）、次に `EXEC SQL DESCRIBE INPUT` ステートメントを使用してクエリで見つかった `input` パラメータを `describe` にする必要があります。

```
EXEC SQL ALLOCATE DESCRIPTOR query_params;
EXEC SQL PREPARE emp_query FROM :query_text;

EXEC SQL DESCRIBE INPUT emp_query
  USING SQL DESCRIPTOR 'query_params';
```

クエリの説明後、`query_params` 記述子には、クエリに必要な各パラメーターに関する情報が含まれます。

この例では、ユーザーが入力したと仮定します。

```
SELECT ename FROM emp WHERE sal > ? AND job = ?;
```

この場合、記述子には `sal > ?` と `job = ?` の2つのパラメーターが記述されます。

クエリ内のパラメーターマーカー（疑問符）の数（したがって、クエリを実行する前に指定する必要がある値の数）を検出するには、次を使用します。

```
EXEC SQL GET DESCRIPTOR ... :host_variable = COUNT;
```

次に、`EXEC SQL GET DESCRIPTOR` を使用して各パラメーターの名前を取得できます。`EXEC SQL GET DESCRIPTOR` を使用して記述子から各パラメーターのタイプを（パラメーターの数とともに）取得することも、文字ストリングの形式で各 `value` を提供することもでき、ECPG はそのストリングを必要な文字列に変換しますデータ型。

最初のパラメーターのデータ型は `numeric` です。2 番目のパラメーターのタイプは `varchar` です。最初のパラメーターの名前は `sal` です。2 番目のパラメーターの名前は `job` です。

次に、各パラメーターをループして、ユーザーに値の入力を求め、それらの値をホスト変数に保存します。`GET DESCRIPTOR ... COUNT` を使用して、クエリ内のパラメータの数を見つけることができます。

```
EXEC SQL GET DESCRIPTOR 'query_params'
  :param_count = COUNT;
```

(次のページに続く)

(前のページからの続き)

```
for(param_number = 1;
    param_number <= param_count;
    param_number++)
{
```

GET DESCRIPTOR を使用して、パラメーターの名前を param_name ホスト変数にコピーします。

```
EXEC SQL GET DESCRIPTOR 'query_params'
    VALUE :param_number :param_name = NAME;

reply = promptUser(param_name);
if (reply == NULL)
    reply_ind = 1; /* NULL */
else
    reply_ind = 0; /* NOT NULL */
```

value を各パラメーターに関連付けるには、EXEC SQL SET DESCRIPTOR ステートメントを使用します。例:

```
EXEC SQL SET DESCRIPTOR 'query_params'
    VALUE :param_number DATA = :reply;
EXEC SQL SET DESCRIPTOR 'query_params'
    VALUE :param_number INDICATOR = :reply_ind;
}
```

これで、EXEC SQL EXECUTE DESCRIPTOR ステートメントを使用して、サーバー上で準備済みステートメントを実行できます。

7.5.30 更新

UPDATE ステートメントを使用して、テーブルに保存されているデータを変更します。構文は次のとおりです。

```
EXEC SQL [AT <database_name>] [FOR <exec_count>]
    UPDATE [ ONLY ] table [ [ AS ] alias ]
    SET {column = { expression | DEFAULT } |
        (column [, ...]) = ({ expression|DEFAULT } [, ...])} [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [[ AS ] output_name] [, ...] ]
```

どこで:

database_name は、テーブルが存在するデータベースの名前（またはデータベースの名前を含むホスト変数）です。この値は、引用符で囲まれていない文字列リテラルまたはホスト変数の形式をとることがあります。

ステートメントを実行する回数を指定するには、FOR exec_count 句を含めます。この句は、SET または

WHERE 句に配列が含まれる場合にのみ有効です。

ECPGPlus は、次の場所にある PostgreSQLCore ドキュメントに記載されている SQL UPDATE ステートメントの追加句のサポートを提供します。

<https://www.postgresql.org/docs/current/static/sql-update.html>

ホスト変数は、値を指定する任意の句で使用できます。ホスト変数を使用するには、文書化された UPDATE 句のいずれかに関連付けられた値を、定義済みの変数に置き換えるだけです。

次の UPDATE ステートメントは、従業員のジョブの説明（:ename ホスト変数で識別される）を :new_job ホスト変数に含まれる値に変更し、現在の給与に :increase ホスト変数:

```
EXEC SQL UPDATE emp
  SET job = :new_job, sal = sal * :increase
 WHERE ename = :ename;
```

拡張された UPDATE ステートメントにより、入力値が許可される句にパラメーターマーカー（疑問符）を含めることもできます。たとえば、WHERE 句にパラメーターマーカーを使用して同じ更新ステートメントを記述できます。

```
EXEC SQL UPDATE emp
  SET job = ?, sal = sal * ?
 WHERE ename = :ename;
```

この UPDATE ステートメントを使用すると、job 列の新しい値の入力をユーザーに求め、:ename で指定された従業員の sal 列の増分を指定できます。

7.5.31 いつでも

WHENEVER ステートメントを使用して、SQL エラーまたは警告が発生したときにクライアントアプリケーションが実行するアクションを指定します。構文は次のとおりです。

```
EXEC SQL WHENEVER <condition> <action>;
```

次の表に、action をトリガーする可能性のあるさまざまな条件を示します。

Condition	Description
見つかりません	サーバーは、行を返さない SELECT を検出したとき、または FETCH が結果セットの最後に到達したときに、NOTFOUND 条件を返します。
SQLERROR	サーバーは、SQL ステートメントによって返される重大なエラーを検出すると、SQLERROR 条件を返します。
SQL 警告	サーバーは、SQL ステートメントによって返される致命的ではない警告を検出すると、SQLWARNING 条件を返します。

次の表は、クライアントが `condition` に遭遇した結果として生じるアクションを説明しています。

Action	Description
CALL 関数 [([args])]	指定された関数を呼び出すようにクライアントアプリケーションに指示します。
続ける	次のステートメントに進むようにクライアントアプリケーションに指示します。
休憩する	クライアントアプリケーションに Cbreak ステートメントを指示します。 break ステートメントは、ループまたは switch ステートメントに現れる場合があります。実行されると、 break ステートメントはループまたは switch ステートメントを終了します。
続ける	クライアントアプリケーションに Ccontinue ステートメントを発行するように指示します。 continue ステートメントはループ内にのみ存在でき、実行された場合、制御フローがループの先頭に戻ります。
DO 関数 ([args])	指定された関数を呼び出すようにクライアントアプリケーションに指示します。
GOTO ラベルまたは GOTO ラベル	ラベルを含むステートメントに進むようにクライアントアプリケーションに指示します。
SQLPRINT	クライアントアプリケーションにメッセージを標準エラーに出力するように指示します。
止まる	実行を停止するようにクライアントアプリケーションに指示します。

次のコードは、クライアントアプリケーションで警告が発生した場合にメッセージを出力し、エラーが発生した場合にアプリケーションを中止します。

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

次のコードを含めて、ユーザーに問題を警告した後、クライアントが処理を続行するように指定します。

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
```

クエリが行を返さない場合、またはカーソルが結果セットの最後に達した場合に関数を呼び出すには、次のコードを含めます。

```
EXEC SQL WHENEVER NOT FOUND CALL error_handler(__LINE__);
```

第 8 章

Conclusion

EDBPostgres™AdvancedServerECPGPlus ガイド

Copyright©2012-2020EnterpriseDBCorporation。

無断複写・転載を禁じます。

EnterpriseDB®Corporation

34CrosbyDrive、Suite201、Bedford、MA01730、アメリカ

T+17813573390F+19784671307E

info@enterprisedb.com

www.enterprisedb.com

- EnterpriseDB および PostgresEnterpriseManager は、EnterpriseDBCorporation の登録商標です。EDB および EDBPostgres は、EnterpriseDBCorporation の商標です。Oracle は、Oracle、Inc の登録商標です。その他の商標は、それぞれの所有者の商標です。
- EDB は、存在する場合、EDB 製品のログオン UI の設計、コーディングのベストプラクティスの確立、レビュー、および入力検証の検証を行います。EDB は追加の入力コンポーネントに対して同じアプローチに従いますが、製品の性質上、限定された検証が可能な信頼できるユーザーによって入力および送信されるフリーフォーム SQL、WMI またはその他の文字列を受け入れる必要がある場合があります。このような場合、ユーザーが誤った入力や危険な入力を禁止することはできません。
- EDB は、将来、認証された信頼できるユーザーからのフリーフォーム SQL、WMI またはその他の潜在的に危険な入力を受け入れる製品に機能を追加する権利を留保しますが、そのような機能はすべて、可能な限り最小のリスクを提供するように設計およびテストされます可能であれば、スーパーユーザーまたは同等の特権が必要です。
- EDB は、すべての潜在的な脅威を予測できることを保証しないため、すべての潜在的な脆弱性が対処または検討されたことを完全に保証することはできません。

索引

- ALLOCATE DESCRIPTOR, 80
- Building and Executing Dynamic SQL Statements, 39
- C-preprocessor Directives, 69
- CALL, 81
- CLOSE, 81
- COMMIT, 82
- Conclusion, 109
- CONNECT, 83
- Constructing a Makefile, 8
- DEALLOCATE DESCRIPTOR, 85
- DECLARE CURSOR, 85
- DECLARE DATABASE, 86
- DECLARE STATEMENT, 86
- Declaring Host Variables, 18
- DELETE, 87
- DESCRIBE, 88
- DESCRIBE DESCRIPTOR, 89
- DISCONNECT, 90
- ECPGPlus Command Line Options, 10
- ECPGPlus Statements, 80
- Error Handling, 59
- Error Handling with sqlca, 59
- EXEC SQL WHENEVER, 66
- EXECUTE, 91
- EXECUTE DESCRIPTOR, 92
- EXECUTE IMMEDIATE, 93
- EXECUTE...END EXEC, 92
- Executing a Non-query Statement with a Specified Number of Placeholders, 43
- Executing a Non-query Statement Without Parameters, 40
- Executing a Query With a Known Number of Placeholders, 46
- Executing a Query With an Unknown Number of Variables, 49
- FETCH, 94
- FETCH DESCRIPTOR, 94
- GET DESCRIPTOR, 95
- INSERT, 97
- Installation and Configuration, 7
- OPEN, 98
- OPEN DESCRIPTOR, 99
- Overview, 3
- PREPARE, 99
- Reference, 69
- ROLLBACK, 100
- SAVEPOINT, 101
- SET CONNECTION, 101, 103
- SET DESCRIPTOR, 104
- SQLDA Structure, 76
- Supported C Data Types, 73
- Type Codes, 75
- UPDATE, 106
- Using a Cursor to Process a Result Set, 20
- Using a Descriptor to Return Data, 28
- Using Descriptors, 25
- Using Embedded SQL, 13
- Using Indicator Variables, 17
- WHENEVER, 107