



EDB

ECPGPlus Guide

リリース *1.0.0*

EnterpriseDB

2021 年 12 月 16 日

目次

第 1 章	ECPGPlus - Overview	3
1.1	インストールと構成	6
1.2	メイクファイルの構築	7
1.3	ECPGPlus コマンドラインオプション	8
第 2 章	Using Embedded SQL	11
2.1	例-簡単なクエリ	11
2.1.1	インジケータ変数の使用	14
2.1.2	ホスト変数の宣言	14
2.2	例-カーソルを使用して結果セットを処理する	15
第 3 章	Using Descriptors	21
3.1	例-記述子を使用してデータを返す	22
第 4 章	Building and Executing Dynamic SQL Statements	33
4.1	例-パラメーターなしの非クエリステートメントの実行	33
4.2	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	35
4.3	例-既知の数のプレースホルダーでクエリを実行する	38
4.4	例-未知の数の変数でクエリを実行する	40
第 5 章	Error Handling	51
5.1	sqlca によるエラー処理	51
5.2	EXEC SQL WHENEVER	55
第 6 章	Reference	59
6.1	C プリプロセッサディレクティブ	59
6.2	サポートされている C データ型	62
6.3	型コード	63
6.4	SQLDA 構造	64
6.5	ECPGPlus ステートメント	67
6.5.1	記述子の割り当て	68
6.5.2	電話	68
6.5.3	閉じる	69
6.5.4	コミット	69
6.5.5	接続	70
6.5.6	記述子の割り当て解除	72
6.5.7	宣言カーソル	72
6.5.8	データベースの宣言	73
6.5.9	宣言文	73
6.5.10	削除	74
6.5.11	記述	75

6.5.12	DESCRIBE DESCRIPTOR	76
6.5.13	切断	77
6.5.14	実行	78
6.5.15	記述子の実行	79
6.5.16	EXECUTE ... END EXEC	79
6.5.17	即時実行	80
6.5.18	FETCH	80
6.5.19	FETCH DESCRIPTOR	81
6.5.20	記述子を取得する	82
6.5.21	挿入	83
6.5.22	開く	84
6.5.23	記述子を開く	85
6.5.24	準備する	85
6.5.25	ロールバック	86
6.5.26	セーブポイント	87
6.5.27	選択	88
6.5.28	接続を設定	89
6.5.29	SET DESCRIPTOR	90
6.5.30	更新	92
6.5.31	いつでも	93

EDB は ECPG (PostgreSQL プリコンパイラ) を拡張して ECPGPlus を作成しました。ECPGPlus を使用すると、EDB Postgres Advanced Server (Advanced Server) 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：

オラクル Dynamic SQL -方法 4 (ODS-M4)。- Pro * C 互換の匿名ブロック。オラクル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 互換性のパートとして、BEGIN DECLARE SECTION および END DECLARE SECTION 指示子を含める必要はありません。

**** PostgreSQL の互換性****

ほとんどの ECPGPlus ステートメントはコミュニティ PostgreSQL で動作しますが、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 EDB Postgres Advanced Server に接続されている場合にのみ機能します。

第1章 ECPGPlus - Overview

</div>

EDB は ECPG (PostgreSQL プリコンパイラ) を拡張して ECPGPlus を作成しました。ECPGPlus は、PostgreSQL C 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。翻訳を実行する際、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プログラムのコンパイル

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`) と `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。例、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` ステートメントで言及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。

舞台裏

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 プリプロセッサ指示子
- C 宣言 (変数、型、関数、…)
- C 定義 (変数、型、関数、…) SQL プリプロセッサ指示子 SQL ステートメント

例：

```
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)
```

(次のページに続く)

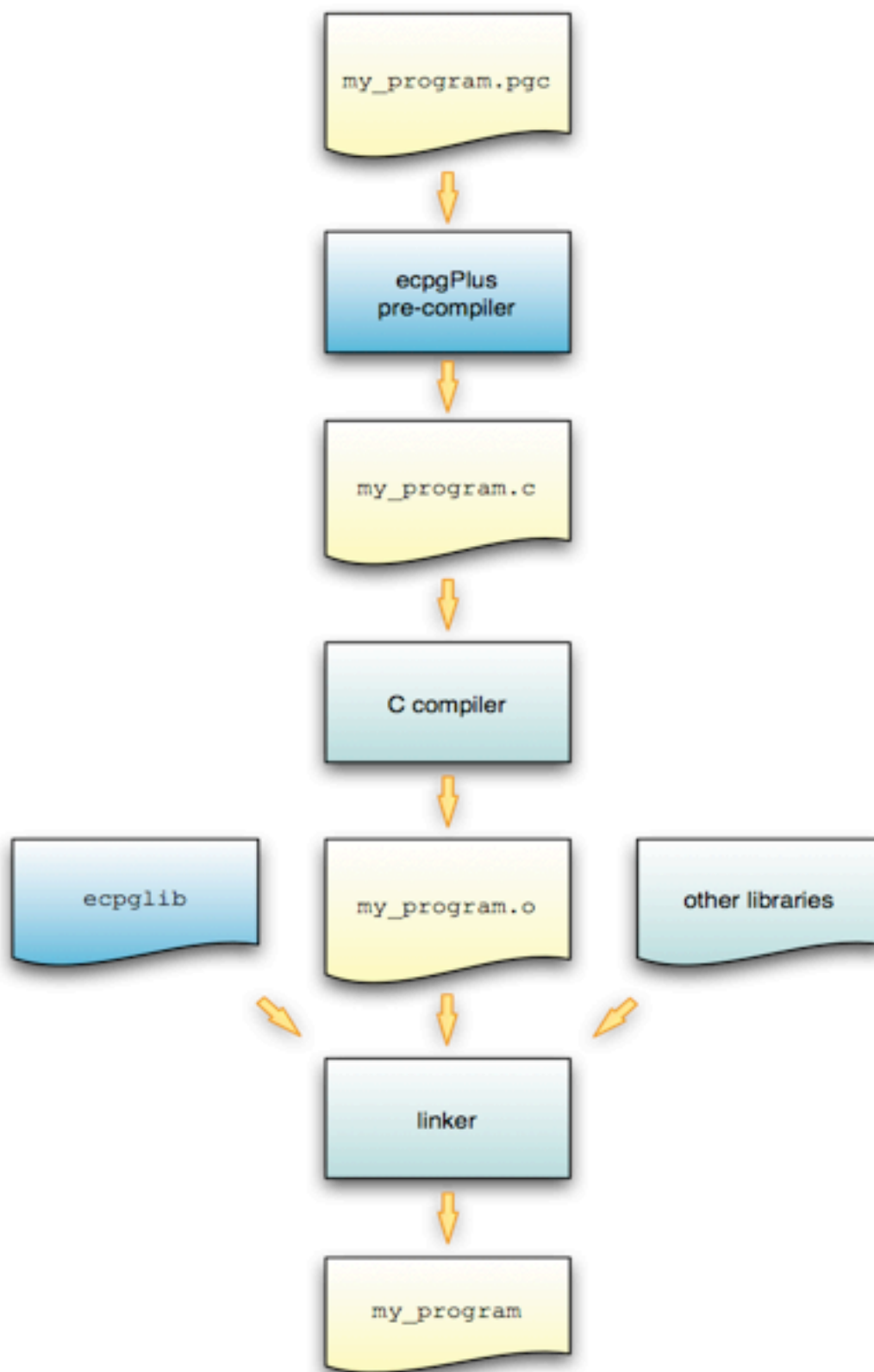


図 1: Compilation of a program containing embedded SQL statements

(前のページからの続き)

```

9 {
10 EXEC SQL BEGIN DECLARE SECTION;
11     VARCHAR custName[50];
12     float custBalance;
13     int custID = custNumber;
14 EXEC SQL END DECLARE SECTION;
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 }

```

上記のコードのフラグメント：

- 行 1 は、C プリプロセッサへのディレクティブを指定します。
 - C プリプロセッサ指示子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。
- 行 2 は、SQL プリプロセッサへのディレクティブを指定します。
 - SQL プリプロセッサ指示子は 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 ステートメント内の * any * C 変数を参照できます。それ以外の場合は、BEGIN/END DECLARATION SECTION ペア内で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 から 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" );
```

プリプロセッサが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）を翻訳せずに出力ファイルに渡し、各 EXEC SQL ステートメントを `ecpglib` ファンクション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" );
```

1.1 インストールと構成

Windows、ECPGPlus は Advanced Server インストールウィザードによって Database Server コンポーネントのパートとしてインストールされます。Linux では、`edb-asxx-server-devel` RPM パッケージを使用してインストールします。xx は Advanced Server のバージョン番号です。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：

```
C:\Program Files\edb\as14\bin
```

Linux の場合：

```
/usr/edb/as14/bin
```

ECPGPlus コンパイラを呼び出す場合、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。例、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 :

```
set EDB_PATH=C:\Program Files\edb\as14\bin
set PATH=%EDB_PATH%;%PATH%
```

Linux の場合 :

```
export EDB_PATH=/usr/edb/as14/bin
export PATH=$EDB_PATH:$PATH
```

1.2 メークファイルの構築

makefile には、C で記述されたプログラム（埋め込み SQL を含む）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（およびリンカ）
- make ユーティリティ
- ECPGPlus プリプロセッサとライブラリ
- ECPGPlus の手順を含む 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 名前付け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
LDLAGS += -g
LDLIBS += $(LIBPATH) -lecpq -lpq

.SUFFIXES: .pgc, .pc

.pgc.c:
*    ecpg -c $(INCLUDES) $?

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

最初の 2 行では、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 Advanced Server に付属しています。

`make` は、C コンパイラの実行時に `CFLAGS` 変数を使用し、リンカの呼び出し時に `LDFLAGS` および `LDLIBS` を使用する必要があることを認識しています。ECPG プログラムは、ECPG ランタイムライブラリ (`-lecpg`) 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) -lecpg -lpq
```

サンプル `makefile` は、`make` が `.pgc` または `.pc` ファイルを C プログラムに変換 `make` 方法を指示します。`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。

```
.pgc.c:
*      ecpg -c $(INCLUDES) $?
```

最初のオプションは、コミュニティ ECPG (ECPGPlus 拡張機能なし) 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` (ECPG ソースファイル) で終わるファイルを `.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。`-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` 変数と `.pgc` ファイルの名前。

`make` を実行するときに、コンパイル ECPG ソースコードファイルの名前を渡します。例、`customer_list.pgc` 名前付けの 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` (カレントディレクトリにある) を調べ、`makefile` に `customer_list.pgc` を C プログラム (`customer_list.c`) にコンパイルするルールが含まれていることを検出し、`make` に組み込まれたルールを使用して `customer_list.c` を実行可能プログラムにコンパイルします。

1.3 ECPGPlus コマンドラインオプション

上記のサンプル `makefile` では、ECPGPlus を呼び出して ECPGPlus を Pro * C 互換モードで呼び出すように指定する場合、`make` に `-C` オプションが含まれています。

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

```
$ ecpg -C PROC INCLUDE=/usr/edb/as14/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 インストレーションディレクトリに移動し、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 <i>mode</i>	-C オプションを使用して、互換性モードを指定します： INFORMIX INFORMIX_SE PROC
-D <i>symbol</i>	プリプロセッサ <i>symbol</i> を定義します。 PROC 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、インクルードファイル。
-I <i>directory</i>	*ディレクトリ*で 'include' ファイルを検索します。
-o <i>outfile</i>	結果を *outfile* に書き込みます。
-r <i>option</i>	実行時の動作を指定します。 *オプション* は次のいずれかになります。 no_indicator-インジケータを使用せず、代わりに NULL 値を表すために特別な値を使用します。 prepare-使用する前にすべてのステートメントを準備します。 questionmarks-プレースホルダとして質問マークの使用を許可します。 usebulk-ホスト変数配列で動作する '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。
--regression	リグレッション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' 指示子を無効にします。
--help	ヘルプオプションを表示します。
--version	バージョン情報を出力します。
	NA

!!! Note * ECPGPlus を呼び出すときに出力ファイル名前を指定しない場合、出力ファイル名前は、ファイル名拡張子 .pgc を取り除き、ファイル名前に .c を追加することによって作成されます。

第2章 Using Embedded SQL

</div>

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。

2.1 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（単一の行を返す）を実行し、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、ハードコードたロール名前と関連する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。クエリーは、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、画面にシンプル結果セットを出力し、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';
    * EXEC SQL
    *   SELECT
    *     empno, ename, sal, comm
    *   INTO
    *     :v_empno, :v_ename, :v_sal, :v_comm INDICATOR:v_comm_ind
    *   FROM
    *     emp

```

(次のページに続く)

(前のページからの続き)

```

*   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 `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、次に `main` ファンクションを宣言します。

```

# 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;
```

ノートモードでコードをプリコンパイルする場合は、`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、*Declaring Host Variables* 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タイプ型は、C データタイプです。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タイプを共有する必要があります。データ型の詳細については、[Supported C Data Types](#) を参照してください。

次のステートメントは、エラーのハンドル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、サーバーはプリントで提供される `sqlprint()` ファンクションを使用して、エラーメッセージを `stderr` (標準エラー) に出力します。次の `EXEC SQL` ステートメントは、Advanced Server 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、alice 名前付けのロールとパスワード 1safepwd を使用して、edb データベースに接続します。

次に、コードは emp テーブルに対してクエリーを実行します。

```
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;
```

クエリーは、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、INTO 節を使用して、取得した値 (empno、ename、sal、および comm 列から) を :v_empno、:v_ename、:v_sal、および edb_tran_ed のホスト変数 (および edb_tran_ed) に割り当て null。取得された最初の値は 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。

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、null 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);
```

null 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 ファンクション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。NULL をそのホスト変数にフェッチする場合、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ノートしてください。null インジケータを使用して、NULL の可能性がある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;
}
```

2.1.1 インジケータ変数の使用

前の例には、(サーバーから返されたときに) `comm` 列の値が `NULL` であった行を識別する `* indicator *` 変数が含まれていました。インジケータ変数は、前の変数のコンテンツが `NULL` であるか、切り捨てられている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。

Indicator Value	Denotes
If an indicator variable is less than 0.	The value returned by the server was <i>NULL</i> .
If an indicator variable is equal to 0.	The value returned by the server was not <i>NULL</i> , and was not truncated.
If an indicator variable is greater than 0.	The value returned by the server was truncated when stored in the host variable.

`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。

クエリーが対応するホスト変数に `NULL` 値を結果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、クエリーが `NULL` 値を返す場合、`ecpglib` は実行時エラーを発生させます。

</div>

2.1.2 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、**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アプリケーションにデータを結果ために使用できる `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
- `struct`
- 任意のスカラ `C` データタイプ

以下のコードフラグメントは、`PROC` モードおよび `non-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`、挿入、および `v_loc` によって返された値を、それぞれ `deptno` 列、`dname` 列、および `loc` 列に挿入します。

PROC モードでコンパイルする場合は、EXEC SQL BEGIN DECLARE SECTION および EXEC SQL END DECLARE SECTION 指示子を省略できます。PROC モードでは、C ファンクション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 指示子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);
}
```

SELECT 文に INTO 句を含めて、ホスト変数を使用して情報を取得することもできます。

```
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 句に型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シンプル例です。複数の行を取得するには、次の例で示すように、カーソルを定義する必要があります。

2.2 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。OPEN CURSOR ステートメントを使用して、カーソルを開きます。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、ユーザー指定のロール名前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。クエリーは、employees 名前付けのカーソルに値を返します。次に、コードサンプルはカーソルを開き、結果セットを一度に 1 行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
```

(次のページに続く)

(前のページからの続き)

```

*
*/
# 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;
    * 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 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、次に main ファンクションを宣言します。

```
# include <stdio.h>
```

(次のページに続く)

(前のページからの続き)

```
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ノートしてください。宣言は `username` および `password` の値を初期化し、ユーザ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が、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` はファンクションパラメータ（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、SQL ステートメント内の `* any * 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によって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 Advanced Server 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、接続時にサーバーに対してアプリケーションを認証する `edb` データベースへの接続を作成します。

次のステートメントは、`employees` 名前付けのカーソルを宣言します。

```
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。クエリーは、次の列から従業員情報を返します：`empno`、`ename`、`sal`、および `comm`。カーソルを宣言するとき、`INTO` 句を含め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` 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;
```

次に、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 `INTO` から以前に宣言されたホスト変数の行を取得します。

```
for (;;)
{
    * 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`、および：`v_comm` ホスト変数（および：`v_comm_ind` null 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 null インジケータを保持するホスト変数も含まれています。取得したレコードの `comm` 列に `NULL` 値が含まれている場合、`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 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);
}
```

null 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` ファンクション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。`NULL` をそのホスト変数にフェッチする場合、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ノートしてください。`NULL` の可能性のある値には null インジケータを使用する必要があります。

サンプルコード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3章 Using Descriptors

</div>

動的 SQL を使用すると、クライアントアプリケーションは、ランタイムに構成される SQL ステートメントを実行します。これは、クライアントアプリケーションを作成しているときにコンテンツやフォームがわからない場合に便利です。ECPGPlus は、識別子（テーブル名前、列名前、インデックス名前など）の代わりにホスト変数を使用することを許可しません。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**ディスクリプタ**のフォームで渡されます。ディスクリプタは、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ディスクリプタ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）のディスクリプタを割り当てます。**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）のディスクリプタを割り当てます。SQL ステートメントのテキストの取得、アセンブル、または計算 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。**DESCRIBE** ステートメントを使用して、選択リストを選択リストディスクリプタに記述します。**DESCRIBE** ステートメントを使用して、バインド変数ディスクリプタ.7 への入力パラメーターを記述します。各入力パラメータの値をユーザに要求します（必要な場合）。**SET DESCRIPTOR** ステートメントを使用して、値をディスクリプタに割り当てます。**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。**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行をフェッチし、行を選択リストディスクリプタ.11. に保存します。**GET DESCRIPTOR** コマンドを使用して、選択リストディスクリプタを調べ、現在の行.12 の各列の値を見つけます。**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フリーしカーソル。

ディスクリプタには、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
<i>CARDINALITY</i>	<i>integer</i>	The number of rows in the result set.
<i>DATA</i>	N/A	The data value.
<i>DATE-TIME_INTERVAL_CODE</i>	<i>integer</i>	If <i>TYPE</i> is 9: 1 - <i>DATE</i> 2 - <i>TIME</i> 3 - <i>TIMESTAMP</i> 4 - <i>TIME WITH TIMEZONE</i> 5 - <i>TIMESTAMP WITH TIMEZONE</i>
<i>DATE-TIME_INTERVAL_PRECISION</i>	<i>integer</i>	Unused.

次のページに続く

表 1 – 前のページからの続き

Field	Type	Attribute Description
INDICATOR	integer	Indicates a <i>NULL</i> or truncated value.
KEY_MEMBER	integer	Unused (returns <i>FALSE</i>).
LENGTH	integer	The data length (as stored on server).
NAME	string	The name of the column in which the data resides.
NULLABLE	integer	Unused (returns <i>TRUE</i>).
OCTET_LENGTH	integer	The data length (in bytes) as stored on server.
PRECISION	integer	The data precision (if the data is of <i>numeric</i> type).
RETURNED_LENGTH	integer	Actual length of data item.
RE- TURNED_OCTET_LENGTH	integer	Actual length of data item.
SCALE	integer	The data scale (if the data is of <i>numeric</i> type).
TYPE	integer	A numeric code that represents the data type of the column: 1 - <i>SQL3_CHARACTER</i> 2 - <i>SQL3_NUMERIC</i> 3 - <i>SQL3_DECIMAL</i> 4 - <i>SQL3_INTEGER</i> 5 - <i>SQL3_SMALLINT</i> 6 - <i>SQL3_FLOAT</i> 7 - <i>SQL3_REAL</i> 8 - <i>SQL3_DOUBLE_PRECISION</i> 9 - <i>SQL3_DATE_TIME_TIMESTAMP</i> 10 - <i>SQL3_INTERVAL</i> 12 - <i>SQL3_CHARACTER_VARYING</i> 13 - <i>SQL3_ENUMERATED</i> 14 - <i>SQL3_BIT</i> 15 - <i>SQL3_BIT_VARYING</i> 16 - <i>SQL3_BOOLEAN</i>

3.1 例-記述子を使用してデータを返す

次のシンプルアプリケーションは、エンドユーザが入力した 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 ディスクリプタを使用して **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、エンドユーザから SQL ステートメントを受け取り、その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、クエリーのテキストを含む文字列を提供する必要があります。

例、ユーザ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"
```

(次のページに続く)

(前のページからの続き)

```

/*****
/* 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;
    * 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;

```

(次のページに続く)

```

*      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;
*          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      ",

```

(前のページからの続き)

```

* "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;

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 );
}

```

(次のページに続く)

(前のページからの続き)

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

/*****
```

コードサンプルは、C `stdio` および `stdlib` ライブラリのプロトタイプと型定義、SQL データタイプシンボル、および SQLCA (SQL 通信領域) 構造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ハンドリングを提供します。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スクリーンに出力されます。

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

アプリケーションには、ディスクリプタで見つかったメタデータをプリントする `print_meta_data()` という名前付けの関クション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 ステートメントのコンテンツ、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 ディスクリプタを割り当てて、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、ユーザから提供された文字列の構文チェックを行います。

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

クエリーのメタデータを SQL ディスクリプタに移動する 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ディスクリプタに問い合わせで結果セットの列数を検出し、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、エンドユーザは SELECT ステートメントを入力していません。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
{
```

ユーザが入力したステートメントが SELECT ステートメント（列カウント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` 名前付け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ディスクリプタを割り当てます。

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

アプリケーションは、プリペアド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;
```

結果セット内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ディスクリプタに取得します。

```
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 が失敗した場合、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、行ディスクリプタ (row_desc) に問い合わせ、列値 (:val)、null インジケータ (:ind) および列名前 (: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、ディスクリプタからマルチプル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、列の値が長すぎて val ホスト変数に収まりません (したがって、<truncated>) をプリントします。そうでない場合、null インジケータは 0 (NOT NULL を意味しプリント)。値 (またはプレフィックスまたは<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()` ファンクションは、ディスクリプタからメタデータを抽出し、各列の名前、データタイプ、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、データタイプ値 (`numeric`) をデータタイプ型名にマップする文字列の配列を定義します。ディスクリプタ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。例、特定の列のタイプが2であることがわかった場合、`types[2]` と記述すると、そのタイプの (`NUMERIC`) の名前を見つけることができます。

```

static char *types[] =
{
    * "unused ",
    * "CHARACTER ",
    * "NUMERIC ",
    * "DECIMAL ",
    * "INTEGER ",
    * "SMALLINT ",

```

(次のページに続く)

(前のページからの続き)

```

* "FLOAT ",
* "REAL ",
* "DOUBLE ",
* "DATE_TIME ",
* "INTERVAL ",
* "unused ",
* "CHARACTER_VARYING",
* "ENUMERATED ",
* "BIT ",
* "BIT_VARYING ",
* "BOOLEAN ",
* "abstract "
};

```

アプリケーションは、ディスクリプタから列カウントを取得します。プログラムは、ディスクリプタの名前を含むホスト変数 (desc) を使用してディスクリプタ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、識別子を使用してディスクリプタを参照しますが、この場合、呼び出し元がディスクリプタ名前を提供したため、ホスト変数を使用してディスクリプタを参照できます。

```
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、ディスクリプタにある各列をループし、各列の名前、タイプ、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、型コード、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、type_name

を対応する型の名前に設定します。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、タイプ名前、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	data	ret
num	name	type	len

01:	empno	NUMERIC	0004
02:	ename	CHARACTER_VARYING	0004
03:	job	CHARACTER_VARYING	0007
04:	mgr	NUMERIC	0004
05:	hiredate	DATE_TIME	0018
06:	sal	NUMERIC	0007
07:	comm	NUMERIC	0000
08:	deptno	NUMERIC	0002
[RECORD 1]			
	empno	: 7902	
	ename	: FORD	
	job	: ANALYST	
	mgr	: 7566	
	hiredate	: 03-DEC-81 00:00:00	
	sal	: 3000.00	
	comm	: <null>	
	deptno	: 20	
[RECORD 2]			
	empno	: 7934	
	ename	: MILLER	
	job	: CLERK	

(次のページに続く)

(前のページからの続き)

mgr	: 7782
hiredate	: 23-JAN-82 00:00:00
sal	: 1300.00
comm	: <null>
deptno	: 10

2 rows

第4章 Building and Executing Dynamic SQL Statements

</div>

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、オラクル Dynamic SQL Method 1 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、オラクル Dynamic SQL Method 2 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、オラクル Dynamic SQL Method 3 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、オラクル Dynamic SQL Method 4 で使用される手法に対応しています。

4.1 例-パラメーター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を含むステートメントを実行ことはできません。

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インパクト可能性があります。同じステートメントを繰り返し実行場合は、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);

```

(次のページに続く)

```

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);
}

/*****/

```

コードサンプルは、C 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、プログラムの基本的な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;

```

例、エラーハンドラを設定します。ECPGPlus は、SQL エラーが発生するたびに handle_error() フังก์ションを呼び出します。

```
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例。

```
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;
```

何らかの理由で EXECUTE IMMEDIATE コマンドが失敗した場合、ECPGPlus は `handle_error()` ファンクションを呼び出します (エラーメッセージをユーザに表示した後、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メッセージ (ok) をユーザに表示し、変更をコミットし、サーバーから切断し、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 エラーが発生するたびに `handle_error()` ファンクションを呼び出します。`handle_error()` ファンクションは、エラーメッセージのコンテンツを出力し、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);
}
```

4.2 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プレースホルダーを含むクエリ以外のコマンドを実行には、最初に PREPARE ステートメント (ステートメントハンドルを提供)、次に EXECUTE ステートメントハンドルを使用したステートメントを実行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 回解析および計画されますが、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 はエラーをレポートします。

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

(次のページに続く)

```

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;
    * EXEC SQL ROLLBACK RELEASE;
    * exit(EXIT_FAILURE);
}
/*****/

```

コードサンプルは、C `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、プログラムの基本的な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;

```

例、エラーハンドラを設定します。ECPGPlus は、SQL エラーが発生するたびに `handle_error()` フังก์ションを呼び出します。

```
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例。

```
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つのパラメータマーカ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付けは **stmtHandle**）。同じステートメントハンドルを使用して、指定したステートメントをマルチプル回実行できます。

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

```
EXEC SQL PREPARE stmtHandle FROM :stmtText;
```

ステートメントをパースおよび計画した後、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ているステートメントを実行し、パラメータマーカ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];
```

何らかの理由で **EXECUTE** コマンドが失敗した場合、ECPGPlus は **handle_error()** ファンクションを呼び出します（エラーメッセージをユーザに表示した後、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メッセージ (**ok**) をユーザに表示し、変更をコミットし、サーバーから切断し、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 エラーが発生するたびに **handle_error()** ファンクションを呼び出します。**handle_error()** ファンクションは、エラーメッセージのコンテンツを出力し、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);
}
```

4.3 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、結果セット内の既知の数の列を使用してクエリ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);
}

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 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、プログラムの基本的な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];
```

例、エラーハンドラを設定します。ECPGPlus は、SQL エラーが発生するたびに `handle_error()` ファンクションを呼び出します。

```
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例。

```
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、単一のパラメータマーカを含むクエリー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は名前付け）。同じステートメントハンドルを使用して、指定したステートメントをマルチプル回実行できます。

```
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マーカ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、クエリーで見つかったプレースホルダごとに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 エラーが発生するたびに `handle_error()` ファンクションを呼び出します。`handle_error()` ファンクションは、エラーメッセージのコンテンツを出力し、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);
}

```

</ div>

4.4 例-未知の数の変数でクエリを実行する

次の例は、結果セット内の未知の数の入力パラメーターや列を使用して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プロンプトをユーザに要求したとき、またはユーザがいくつかの条件（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ときに発生します。

```

/*****
# 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);

```

(次のページに続く)

(前のページからの続き)

```

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);
}

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);
    *       }
    *   }
}

```

(次のページに続く)

```

}

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;
    *         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;
    *     }
    * }

```

(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;
*      while (1)
*      {
*          const char *delimiter = "";
*          EXEC SQL FETCH dynCursor USING DESCRIPTOR results;
*          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");
*      }
*  }
}
/*****/

```

コードサンプルは、C `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 `sqllda.h` および `sqlcpr.h` ヘッダーファイルが含まれています。`sqllda.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 構造体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 <sqllda.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マーカ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、動的クエリーの実行によって取得されたメタデータと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への接続に使用されます。3番目のホスト変数 (`stmtTxt`) は、実行クエリーのテキストを含む 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、エラーハンドラが設定され、データベー

スサーバに接続されます。

```
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()` ファンクションを呼び出して、各ディスクリプタに必要なメモリを割り当てます。各ディスクリプタには（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。

SQLDA ディスクリプタを割り当てる場合、結果セットで見つけると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見つけると予想するパラメーターの最大数（バインド変数記述子の場合）を指定します-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）名前と各インジケータ変数名前の最大長さを指定する必要があります。この場合、名前の長さは 64 バイト以下であると予想されます。

SQLDA 構造の完全な説明については、[SQLDA Structure](#) 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()` ファンクションを呼び出します。`bindParams()` ファンクションはバインドディスクリプタプロンプトを調べ、動的クエリーで見つかった各パラメータマーカの代わりに置き換える値をユーザに求めます。

```
bindParams();
```

最後に、プログラムはカーソルを開き（ユーザが指定したパラメータ値があればそれを使用して）、`displayResultSet()` ファンクションを呼び出してクエリーによって生成された結果セットをプリントします。

```
EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

displayResultSet();
}
```

`bindParams()` ファンクションは、動的クエリーにパラメータ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、各パラメータの値をユーザに求め、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マーカーを説明する情報を設定します。

```
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マーカーが含まれない場合、`params->F` には 0 が含まれます。ステートメントにディスクリプタに収まるよりも多くのパラメータが含まれる場合、`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）。`params->F` に正の数値が含まれている場合、その数値はステートメントで見つかったパラメータ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);
```

指定されたパラメータの値の入力をユーザに求めた後、プログラムは、値のデータタイプを示すように `params->T[i]` を設定することにより、その値をパラメータにバインドし（型コードのリストについては `Type Codes` を参照）、`params->L[i]` の長さを値 (`fgets()` によって追加された末尾の改行文字を削除するために 1 を減算します)、およびユーザによって提供された NULL 終了文字列のコピーを指す `params->V[i]`。

```

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

```

`displayResultSet()` ファンクションは、結果セットの各行をループし、各行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 ディスクリプタ (`results`) に結果セット内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。ステートメントがディスクリプタに収まるよりも多くの列を返す場合、`results->F` には負の数が含まれます（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、その数はクエリー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、結果セットの列ごとに 1 回反復します。

```

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

```

`results->T` にあるタイプコードをデコードために、プログラムは `sqlnul()` ファンクションを呼び出します（SQLDA 構造体の `T` メンバの説明を参照してください）。この `sqlnul()` の呼び出しは、型コードのみを含むように変更します（nullability フラグはにコピーされます）。`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);

```

列の実際のデータタイプをデコードした後、プログラムは結果ディスクリプタを変更して、ECPGPlus に NULL で終わる文字列のフォームで各値を結果ように指示します。ディスクリプタを変更前に、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 `make` に、プログラムは各列 (`results->V[col]`) の最大長さと各列 (`results->T[col]`) のデータタイプを調べます。

数値の場合 (`results->T[col] = 2`)、プログラムは `sqlprc()` ファンクションを呼び出して、列の長さから精度とスケールを抽出します。文字列フォーム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、10 進数、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` によって報告された最大列幅で始まり、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;
}
```

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の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、列 (`results->T[col]`) の型コードを 1 に設定して今後の指示を出します NULL で終わる文字列のフォームで値を結果 `FETCH` ステートメント。

```
* results->V[col] = malloc(length);
* results->L[col] = length;
```

(次のページに続く)

(前のページからの続き)

```
* results->T[col] = 1;
}
```

この時点で、FETCH ステートメントが各値を NULL で終わる文字列のフォームで適切なサイズのバッファにコピーできるように、結果ディスクリプタが構成されます。

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` ディスクリプタにフェッチします。FETCH ステートメントが失敗した場合（カーソルが使い果たされたため）、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、制御はループの末尾に移動しループ。

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

FETCH ステートメントは、結果ディスクリプタの次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 NULL ではないが、指定されたスペースに収まるにはラージすぎる場合、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、指定された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", delimiter, "<null>");
    * else
    *     printf("%s", delimiter, results->V[col]);
    *     delimiter = ", ";
}

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

(次のページに続く)

(前のページからの続き)

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


第5章 Error Handling

</div>

ECPGPlus は、埋め込み SQL コードのエラーを検出してハンドルする 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、`sqlca` データ構造のエラーメッセージを調べ、クライアントクライアントアプリケーション用にカスタマイズされたエラーハンドリングを提供できます。
- クライアントアプリケーションには、エラー処理コードを追加するように ECPGPlus コンパイラに指示する `EXEC SQL WHENEVER` 指示子を含めることができます。

5.1 `sqlca` によるエラー処理

`sqlca` (SQL 通信領域) は、サーバーからクライアントアプリケーションに情報を通信するために `ecpglib` によって使用されるグローバル変数です。SQL ステートメント (例、`INSERT` または `SELECT` ステートメント) 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 sqlerrrd[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で`#include sqlca.h` ファイルを使用する必要はありません。

Advanced Server `sqlca` 構造には、次のメンバーが含まれています。

`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` メンバは SQL 92 で非推奨になりました。Advanced Server 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、負の値はワーニング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` には値 0 が含まれます。クライアントアプリケーションでステートメントの実行中にエラー（またはワーニング）が発生した場合、`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 他のデータベース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` 内に埋め込みれた構造体であり、2 つのメンバーで構成されています。

`sqlerrml`

`sqlerrml` には、現在 `sqlerrmc` に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ています。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` には文字列"NOT SET"が含まれます。

`sqlerrd` は、6 つの要素を含む配列です。

- `sqlerrd[1]` には、処理された行の OID が含まれます（該当する場合）。
- `sqlerrd[2]` には、処理または返された行の数が含まれます。
- `sqlerrd[0]`, `sqlerrd[3]`, `sqlerrd[4]` および `sqlerrd[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で致命的でないワーニングが発生した場合、`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 null で終わる文字列ではないことにノートしてください。

`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。

SQL アプリケーションの実行中にクライアントアプリケーションでマルチプルのエラー（または警告）が発生した場合、`sqlstate` には最後に返された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前とエラーの説明を示します。

sqlstate	sqlcode (Depre- cated)	Symbolic Name	Description
YE001	-12	ECPG_OUT_OF_MEMORY	仮想メモリが使い果たされています。
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, or 07002	-201	ECPG_TOO_MANY_ARGUMENTS	プログラムは、コマンドが期待するよりも多くの変数を指定します。
07001, or 07002	-202	ECPG_TOO_FEW_ARGUMENTS	プログラムは、コマンドが期待するよりも少ない変数を指定しました。
21000	-203	ECPG_TOO_MANY_MATCHES	SQL コマンドはマルチプル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ように準備されています。

次のページに続く

表 1 – 前のページからの続き

sqlstate	sqlcode (Depre- cated)	Symbolic Name	Description
42804	-204	<i>ECPG_INT_FORMAT</i>	ホスト変数 (C コードで定義) 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()` ファンクションを使用して文字列値を数値フォームに変換します。
42804	-205	<i>ECPG_UINT_FORMAT</i>	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、`strtoul()` ファンクションを使用して文字列値を数値フォームに変換します。
42804	-206	<i>ECPG_FLOAT_FORMAT</i>	ホスト変数 (C コードで定義) は FLOAT タイプであり、選択されたデータは FLOAT に変換できないタイプです。`ecpglib` は、`strtod()` ファンクションを使用して文字列値を数値フォームに変換します。
42804	-211	<i>ECPG_CONVERT_BOOL</i>	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 に格納することはできません。
YE002	-2-1	<i>ECPG_EMPTY</i>	サーバーに送信されたステートメントは空でした。
22002	-213	<i>ECPG_MISSING_INDICATOR</i>	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 値を受け取りました)。
42804	-214	<i>ECPG_NO_ARRAY</i>	サーバーは配列を返しましたが、対応するホスト変数は配列を格納できません。
42804	-215	<i>ECPG_DATA_NOT_ARRAY</i>	サーバーは、配列ではない値を、配列値を期待するホスト変数に返しました。
08003	-220	<i>ECPG_NO_CONN</i>	クライアントアプリケーションが存在しない接続を使用しようとした。
YE002	-221	<i>ECPG_NOT_CONN</i>	クライアントアプリケーションは、割り当てられているが閉じられている接続を使用しようとした。
26000	-230	<i>ECPG_INVALID_STMT</i>	声明は作成されていません。

次のページに続く

表 1 – 前のページからの続き

sqlstate	sqlcode (Depre- cated)	Symbolic Name	Description
33000	-240	<i>ECPG_UNKNOWN_DESCRIPTOR</i>	指定されたディスクリプタが見つかりません。
07009	-241	<i>ECPG_INVALID_DESCRIPTOR_INDEX</i>	ディスクリプタインデックスが範囲外です。
YE002	-242	<i>ECPG_UNKNOWN_DESCRIPTOR_ITEM</i>	クライアントアプリケーションが無効なディスクリプタアイテムを要求しました (内部エラー)。
07006	-243	<i>ECPG_VAR_NOT_NUMERIC</i>	動的ステートメントが、数値以外のホスト変数の数値を返しました。
07006	-244	<i>ECPG_VAR_NOT_CHAR</i>	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	<i>ECPG_PGSQL</i>	サーバーは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	<i>ECPG_TRANS</i>	サーバーは、指定されたトランザクションをスタート、コミット、またはロールバックできません。
08001	-402	<i>ECPG_CONNECT</i>	クライアントアプリケーションのデータベースへの接続に失敗しました。
02000	100	<i>ECPG_NOT_FOUND</i>	最後のコマンドで行が取得または処理されていないか、カーソルの最後に到達しました。
			NA

5.2 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シンプルエラーハンドリングを実装します。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 がゼロより小さいの場合、SQLERROR 条件が存在します。

SQLWARNING

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

NOT FOUND

`sqlca.sqlcode` が `ECPG_NOT_FOUND` の場合（クエリー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

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

DO CONTINUE

DO CONTINUE のアクションは、出力された C コードで 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 goto ステートメントを使用して、指定された label にジャンプします。

SQLPRINT

`sqlprint()` ファンクションを使用して、エラーメッセージを `stderr`（標準エラー）に出力します。`sqlprint()` ファンクションは `sql error` を出力し、続いて `sqlca.sqlerrm.sqlerrmc` の内容を出力します。

STOP

`exit(1)` を呼び出してエラーをシグナルし、プログラムを終了します。

DO BREAK

C 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 ファンクションを呼び出します。

例：

次のコードは、クライアントアプリケーションでフラグメント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;
```

!!! Note * 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。コンパイラディレクティブは各行にオーダーに適用され、コンパイラが別のディレクティブ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があるプログラムの任意の部分にエラー処理指示子を追加することを検討する必要があります。

第6章 Reference

</div>

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

6.1 C プリプロセッサディレクティブ

ECPGPlus C 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
- non-PROC モード

**** PROC モードでのコンパイル****

PROC モードでは、ECPGPlus で次のことができます。

- EXEC SQL BEGIN/END DECLARE SECTION の外部でホスト変数を宣言します。
- ECPG と互換性のあるデータタイプである限り、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により) ECPGPlus を呼び出すと、ECPG コンパイラは次の C プリプロセッサ指示子を受け入れます。

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

(次のページに続く)

(前のページからの続き)

```
# undef symbolName
# defined(symbolName)
```

プリプロセッサ指示子は、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。例、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指示子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 パーサは処理されたトークンストリームのみを参照しながら、完全なコードはオリジナル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指示子を実行するときに、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
```

例、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 パーサに送信します。

```
long long customerBalance;
```

!!! Note * EXEC ORACLE プリプロセッサ指示子は、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。

** SELECT_ERROR プリコンパイラオプションの使用**

ECPGPlus を互換モードで使用する場合、SELECT_ERROR プリプロセッサ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をハンドルする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と、SELECT ステートメントはエラーメッセージを結果ます。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、エラーメッセージを抑制するようにプログラムに指示するには、NO を指定します。

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

非 PROC モードでのコンパイル

-C PROC コマンドラインオプションを含めない場合：

- C プリプロセッサ指示子は、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 PostgreSQL Core 文書で説明されている動作を実装します。

</div>

6.2 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 データタイプ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。例、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 データタイプを必要な SQL 型に変換しようとしします。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。SQL タイプを char[n] または varchar[n] との間で変換することは安全ですが、多くの場合、int、double、または 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 でサポートされる数値および文字型に加えて、pgtypeslib ランタイムライブラリは、日付/時刻および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 が提供するデータタイプを使用するには、適切なヘッダファイル #include を使用する必要があります。

6.3 型コード

次の表には、* external *データ型の型コードが含まれています。外部データタイプは、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にバインドするか、バッファを SELECT-リストアイテムにバインドする場合、対応する SQLDA ディスクリプタ (descriptor->T[column]) の型コードは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	<i>char[]</i>
3	<i>int</i>
4, 7, 21	<i>float</i>
5, 6	<i>null-terminated string (char[length+1])</i>
9	<i>varchar</i>
22	<i>double</i>
68	<i>unsigned int</i>

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タイプ配列 (descriptor->T[column]) を設定します。

Internal Type Code	Server Type
1	<i>VARCHAR2</i>
2	<i>NUMBER</i>
8	<i>LONG</i>
11	<i>ROWID</i>
12	<i>DATE</i>
23	<i>RAW</i>
24	<i>LONG RAW</i>
96	<i>CHAR</i>
100	<i>BINARY FLOAT</i>
101	<i>BINARY DOUBLE</i>
104	<i>UROWID</i>
187	<i>TIMESTAMP</i>
188	<i>TIMESTAMP W/TIMEZONE</i>

次のページに続く

表 2 – 前のページからの続き

Internal Type Code	Server Type
189	<i>INTERVAL YEAR TO MONTH</i>
190	<i>INTERVAL DAY TO SECOND</i>
232	<i>TIMESTAMP LOCAL_TZ</i>

</div>

6.4 SQLDA 構造

オラクル Dynamic SQL 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 */
};
```

パラメータ

N - maximum number of entries

N 構造体メンバには、SQLDA が記述できるエントリの最大数が含まれます。このメンバは、SQLDA 構造を割り当てるときに `sqlald()` ファンクションによって設定されます。OPEN または FETCH ステートメントでディスクリプタを使用する前に、N を説明されている値の**実際**の数に設定する必要があります。

V - data values

V 構造体メンバは、データ値の配列へのポインタです。

- SELECT-リストディスクリプタの場合、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（配列の各メンバは結果セットの列に対応します）。

- バインドディスクリプタの場合、**V** はパラメータ値の配列を指します（ディスクリプタ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](#) ファンクションを参照してください。

L - length of each data value

L 構造体メンバは、長さの配列へのポインタです。この配列の各メンバは、**V** 配列の対応するメンバで利用可能なメモリの量を示す必要があります。例、**V[5]** が 20 バイトの NULL 終了文字列を保持するのに十分なラジスのバッファを指す場合、**L[5]** 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）を含む必要があります。アプリケーションは、**L** 配列の各メンバを設定する必要があります。

T - data types

T 構造体のメンバは、ディスクリプタされた各列（またはパラメータ）に 1 つずつ、データ型の配列を指します。

- バインドディスクリプタの場合、**T** 配列の各メンバを設定して、ECPGPlus に各パラメータのデータタイプを通知する必要があります。
- SELECT-リストディスクリプタの場合、**DESCRIBE SELECT LIST** ステートメントは、**T** 配列の各メンバを、対応する列にあるデータのタイプを反映するように設定します。

FETCH ステートメントを実行する前に **T** 配列のメンバを変更して、ECPGPlus が対応する値を特定のデータタイプに変換するように強制できます。例、特定の列が **DATE** 型であることを **DESCRIBE SELECT LIST** ステートメントが示す場合、対応する **T** メンバを変更して、次の **FETCH** ステートメントが NULL で終わる文字列のフォームでその値を結果ように要求できます。**T** 配列の各メンバは、数値型コード（型コードのリストを参照）。**DESCRIBE SELECT LIST** ステートメントによって返されるタイプコードは、**FETCH** ステートメントによって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、**T** の各メンバは、対応する列が NULL 可能かどうかを示すデータタイプおよびフラグをエンコードします。このファンクションを使用して、**T** 配列のメンバから型コードと NULL 可能フラグを抽出できます。ファンクションのシグネチャは次のとおりです。

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

例、**results** 名前付けのディスクリプタの 3 列目の型コードと 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 構造体メンバは、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 **sqlald()** ファンクションを呼び出してディスクリプタを割り当てるときに割り当てられます。

- SELECT-リストディスクリプタの場合、**I** 配列の各メンバは、対応する列に NULL（非ゼロ）値または非 NULL（ゼロ）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の場合、アプリケーションは **I** 配列の各メンバを設定して、対応するパラメータ値が NULL かどうかを示す必要があります。

F - number of entries

F 構造体のメンバは、記述子によってディスクリプタされる値の数を示します (N 構造体のメンバは、記述子によってディスクリプタできる値の**最大数**を示し、F は値の**実際の数**を示します)。F メンバの値は、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-リストディスクリプタの場合、結果セットの列の数がディスクリプタによって許可される値の最大数 (N 構造体メンバによって決定される) より小さいである場合、F には正の値が含まれます。ステートメントが SELECT ステートメントではない場合は 0、クエリーが N 構造体メンバで許可されているよりも多くの列を返す場合は負の値。
- バインドディスクリプタの場合、ステートメントで見つかったパラメーターの数がディスクリプタで許可されている値の最大数 (N 構造体メンバによって決定される) 以下である場合、F には正の値が含まれます。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 に正の値が含まれている場合 (DESCRIBE ステートメントの実行後)、その数は結果セットの列数 (SELECT-リストディスクリプタの場合) 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 (バインドディスクリプタの場合) を反映します。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。例、SELECT リストを記述した後、F に -24 が含まれている場合、クエリーが 24 列を返すことがわかります。

S - column/parameter names

S 構造体メンバは、NULL で終了する文字列の配列を指します。

- SELECT-リストディスクリプタの場合、DESCRIBE SELECT LIST ステートメントは、この配列の各メンバを結果セット内の対応する列の名前に設定します。
- バインドディスクリプタの場合、DESCRIBE BIND VARIABLES ステートメントは、この配列の各メンバ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、クエリー内のパラメーターマーカーの左から右のオーダーによって決定されます。例、最初のパラメータの名前は常に?0、2 番目のパラメータの名前は常に?1 です、など。

M - maximum column/parameter name length

M 構造体のメンバは、長さの配列を指します。この配列の各メンバは、S 配列の対応するメンバの**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/パラメータ名前の最大長さを指定します)。この配列は、sqlald() ファンクションによって設定されます。

C - actual column/parameter name length

C 構造体のメンバは、長さの配列を指します。この配列の各メンバは、S 配列の対応するメンバの**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/パラメータ名前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 構造体メンバは、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 構造体のメンバは、長さの配列を指します。この配列の各メンバは、X 配列の対応するメンバの**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前の最大長さを指定します）。

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。

Z - actual indicator name length

Z 構造体のメンバは、長さの配列を指します。この配列の各メンバは、X 配列の対応するメンバの**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前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。

6.5 ECPGPlus ステートメント

埋め込み SQL ステートメントにより、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。

Advanced Server SQL 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フォームを取ります。

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

ここで、sql_command_body は標準 SQL ステートメントを表します。ホスト変数は、SQL ステートメントが評価式を予期する任意の場所で使用できます。値式のホスト変数の置換の詳細については、[Declaring Host Variables](#) を参照してください。

ECPGPlus は、一部のステートメントの PostgreSQL サーバサイドの構文を拡張します。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 コマンドの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/> 静的/sql-commands.html >

6.5.1 記述子の割り当て

ALLOCATE DESCRIPTOR ステートメントを使用して、SQL ディスクリプタ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、ディスクリプタ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。

descriptor_name は、ディスクリプタの名前またはディスクリプタ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フォームをとることがあります。

variable_count は、ディスクリプタ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。

次のコードフラグメントは、配列 (**emp_array**) として処理できる **emp_query** という名前付けのディスクリプタを割り当てます。

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

6.5.2 電話

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ファンクションを呼び出します。CALL ステートメントは、Advanced Server 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。最初のフォームは、**ファンクション**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 番目のフォームは、**プロシージャ**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ファンクションの名前です。プログラム名前は、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、プログラムが必要とする引数のコンマ区切りリストを指定します。各 **actual_argument** は、プログラムが予期する正式な引数に対応することに注意してください。各仮引数は、IN パラメータ、OUT パラメータ、または INOUT パラメータです。

:ret_variable は、プログラムがファンクション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、プログラムがファンクションである場合、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。

例、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 ファンクションを呼び出し、そのファンクションによって返された値を:job ホスト変数にキャプチャしホスト変数。

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

6.5.3 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リリースカーソル。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。カーソル名前は、識別子またはホスト変数のフォーム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 カーソルを使用する必要はありません。

emp_cursor 名前付け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。

6.5.4 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、他のユーザーに可視にします。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フォームをとることがあります。

互換性のために、ECPGPlus は COMMENT 句をエラーなしで受け入れますが、COMMENT 句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。

例、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、ステートメントは、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ときのみコミットされます。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** 機能呼び出すように指定します。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
```

6.5.5 接続

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つのフォームがあります。1つはオラクル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。

最初のフォームは、オラクル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名前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。値を変数のフォームで、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、ワーニングを報告します。

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フォームを使用して、次のコマンドを使用して、接続するロールの ID を含む `user` 名前付けホスト変数と、関連するパスワードを含む `password` 名前付けホスト変数変数との接続を確立します。

```
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フォームを使用して、`:connection_id` 名前付け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、ユーザのスラッシュ区切りのロール名前と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 番目のフォーム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 番目のフォーム：

`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前を使用してデフォルト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 を指定する場合は、`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、識別子のフォームをとる必要があります（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、マルチプル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。任意の EXEC SQL ステートメントで接続を名前（DEFAULT）で参照できます。

CURRENT は、最近開かれた SET CONNECTION TO ステートメントで言及された、最後に開かれた接続または接続です。EXEC SQL ステートメントで名前で接続を参照しない場合、ECPG は接続の名前が CURRENT であると想定します。

user_name は、Advanced Server 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。

次のコードは、フラグメントステートメントの 2 番目のフォームを使用して、alice というロールと 1safepwd に関連付けられたパスワードを使用して、edb 名前付け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 '1safepwd';
```

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前を変更するときに、接続名前を使用できます。

6.5.6 記述子の割り当て解除

DEALLOCATE DESCRIPTOR ステートメントを使用して、割り当てられたディスクリプタで使用中のメモリをフリーします。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、ディスクリプタの名前です。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フォームをとることがあります。

次の例では、emp_query 名前付けのディスクリプタの割り当てを解除します。

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

6.5.7 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フォームをとることがあります。データベース名前を指定しない場合、database_name のデフォルト値はデフォルトデータベースです。

`cursor_name` はカーソルの名前です。

`select_statement` は、カーソルの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` 名前付け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;
```

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、従業員名前、給与、および歩合を含む結果セットを生成します。

6.5.8 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（例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;
```

`acctg` をデータベース識別子として宣言するコマンドを呼び出した後、接続の確立時または `AT` 句で `acctg` データベースを名前で被参照できます。

このステートメントは効果がなく、`Pro * C` との互換性のためにのみ提供されています。

6.5.9 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。 `Advanced Server` 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フォームをとることがあります。

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;
```

このステートメントは効果がなく、Pro *C との互換性のためにのみ提供されています。

6.5.10 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：

ステートメントが実行され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。ONLY 句を含めて、指定したテーブルに処理を制限します。ONLY 句を含めない場合、名前付け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前です。

`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 条件に表示できます。

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、BOOLEAN 型の値を返す式、ホスト変数、またはパラメータマーカーです。`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ー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、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、行が削除された後に DELETE コマンドによって計算され、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、すべての列が結果れます。
- 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コ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。

例、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 PostgreSQL Core 文書を参照してください。

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

6.5.11 記述

DESCRIBE ステートメントを使用して、プリペアドステートメントたステートメントに必要な入力値の数、または準備されたプリペアド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 ディスクリプタを取り込みます。SQL ディスクリプタを移入するには、ALLOCATE DESCRIPTOR および DESCRIBE...DESCRIPTOR 文を使用します。

```
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 ステートメントまたは 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（およびディスクリプタディスクリプタにスペースを割り当てする必要があります）。

DESCRIBE ステートメントの最初のフォームを実行と、ECPG は、指定された記述子に、ステートメントに必要な各入力変数のディスクリプタを追加します。例、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ーを準備できます。

```
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、ディスクリプタを調べて、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 番目のフォームを実行と、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ディスクリプタを取り込みます。例、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、ディスクリプタを調べて、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 ステートメントの代替の Pro * C 互換構文です。

6.5.12 DESCRIBE DESCRIPTOR

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 ディスクリプタに格納します。DESCRIBE DESCRIPTOR を使用する前に、ALLOCATE DESCRIPTOR ステートメントでディスクリプタ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 はディスクリプタの名前です。descriptor_name は、引用符で囲まれた文字列値、またはディスクリプタの名前を含むホスト変数です。

INPUT 句を含めると、ECPGPlus は指定されたディスクリプタ子に、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。

例、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ーを準備できます。

```
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、ディスクリプタを調べて、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 は指定されたディスクリプタにステートメントによって返された値を取り込みます。

OUTPUT 句を含めると、ECPGPlus は、指定されたディスクリプタ子に、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。

例、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、ディスクリプタを調べて、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。

6.5.13 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前です。接続名前を指定しない場合、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 名前付け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 は接続に名前 DEFAULT を割り当てます。

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 名前付け）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);
}

```

6.5.14 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)。

プリペアド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。

- DESCRIPTOR `SQLDA_descriptor` 句を含めて、パラメータに `SQLDA` ディスクリプタ値を提供します。
- `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、`emp` テーブルにレコードを挿入するプリペアド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);

```

プリペアド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;

```

6.5.15 記述子の実行

EXECUTE ステートメントを使用して、SQL ディスクリプタを使用して、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ごとに 1 回実行されます。

statement_identifier は、EXEC SQL PREPARE ステートメントで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、プリペアドステートメントに必要な入力パラメータの値を指定します。

INTO 句を含めて、プリペアドステートメント文によって返された結果を EXECUTE 文が書き込むディスクリプタを指定します。

descriptor_name は、ディスクリプタの名前（単一引用符で囲まれた文字列リテラルとして）、またはディスクリプタの名前を含むホスト変数を指定します。

次の例では、ディスクリプタ **stmtText:** に含まれる値を使用して、プリペアドステートメント **give_raise** を実行します。

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

6.5.16 EXECUTE ... END EXEC

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;
```

!!! Note * EXECUTE...END EXEC ステートメントは、Advanced Server でのみサポートされています。

6.5.17 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 ステートメントによって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（クライアントアプリケーションを作成する場合など）に便利です。例、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）ユーザにプロンプト。ユーザが文のテキストを文字列値として提供した後、文は 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。

次の例では、`: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;
```

6.5.18 FETCH

FETCH ステートメントを使用して、カーソルから SQLDA ディスクリプタまたはホスト変数のターゲットリストに行の結果を返します。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 句を含める場合、FETCH ステートメントは、指定された SQLDA ディスクリプタにサーバーから返された値を取り込みます。

INTO 句を含める場合、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、フラグメントというテーブルから `employee number`、`name`、および `salary` を取得する `employees` 名前付けのカーソルを宣言しています。

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

6.5.19 FETCH DESCRIPTOR

FETCH DESCRIPTOR ステートメントを使用して、カーソルから行を SQL ディスクリプタに取得します。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 および OPEN する必要があります。

INTO 句を含めて、プリペアドステートメント文によって返された結果を EXECUTE 文が書き込む SQL ディスクリプタを指定します。`descriptor_name` は、ディスクリプタの名前（単一引用符で囲まれた文字列リテラルとして）、またはディスクリプタ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ディスクリプタ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` という名前付けのディスクリプタを割り当てます。次に、結果セットの行をループする前に、FETCH を使用してカーソルから次の行をディスクリプタに取得する前に、プリペアドステートメント (`my_cursor`) のカーソルを宣言して開きます。

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

(次のページに続く)

```
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';
```

6.5.20 記述子を取得する

GET DESCRIPTOR ステートメントを使用して、ディスクリプタ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。最初のフォームは、ディスクリプタの値（または列）の数を返します。

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

2 番目のフォームは、特定の値（VALUE column_number 句で指定）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。例、array_size が 10 の場合、:host_variable は host_variables の 10 メンバ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ごとに 1 回実行されます。

descriptor_name は、ディスクリプタの名前（単一引用符で囲まれた文字列リテラルとして）、またはディスクリプタの名前を含むホスト変数を指定します。

VALUE 句を含めて、ディスクリプタから取得した情報を指定します。

- column_number は、ディスクリプタ内の変数の位置を識別します。
- 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、取得したディスクリプタアイテム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、フラグメントステートメントを使用して、ユーザーが指定した文字列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 (:stmt) によって提供された文字列を構文チェックする前に、SQL ディスクリプタ (parse_desc 名前付け) 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ディスクリプタ parse_desc に移動します。EXEC SQL GET DESCRIPTOR の呼び出しは、ディスクリプタに問い合わせ、結果セット内の列 (:col_count) の数を検出します。

6.5.21 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：

ステートメントが実行され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。列名前は、サブフィールド名前または配列添字で修飾できます。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、またはパラメータマーカーです。対応する列にデフォルト値を設定するには、DEFAULT を指定します。

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、行が挿入された後に `INSERT` コマンドによって計算されて返される式です。式は、テーブル内の任意の列を参照できます。挿入され行のすべての列を結果には、`*`を指定します。

`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');
```

!!! Note * `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 PostgreSQL Core 文書を参照してください。

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

6.5.22 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、カーソルを初期化する `DESCRIPTOR SQLDA_descriptor` または `host variables` (およびオプションの `indicator variables`) のコンマ区切りリストのいずれかです。`SQLDA_descriptor` を指定する場合、ディスクリプタは `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` で提供される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` 名前付け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 (ホスト変数 `emp_name` に格納されている) と一致する従業員の名前、`name`、`salary` および `commission` を返します。

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

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

6.5.23 記述子を開く

OPEN DESCRIPTOR ステートメントを使用して、SQL ディスクリプタ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 ディスクリプタの名前（単一引用符で囲まれた文字列リテラルのフォーム）またはカーソルを初期化するクエリーを含む SQL ディスクリプタの名前を含むホスト変数を指定します。

例、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 名前付け）を開きます。

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

6.5.24 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マルチプル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を節約します。

PREPARE ステートメントを使用して、実行のために 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。最初のフォームは次のとおりです。

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

2 番目のフォームは次のとおりです。

```
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 ステートメントまたは 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フォームをとることがあります。

プリペアド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** するときに変更され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** するときに、各パラメータの値を指定します。値は、プレースホルダーを置き換えるオーダーで提供する必要があります。

次の例では、`emp` テーブルにレコードを挿入するプリペアドステートメント（`add_emp` 名前付け）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);
```

!!! Note * 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。準備されたステートメントは現在のセッションの間だけ持続します。

6.5.25 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アボートし、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アボートするには、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アボートし、すべての更新を破棄します。

RELEASE 句を含めて、アプリケーションで EXEC SQL COMMIT RELEASE を実行し、接続を閉じます。

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アボートされ、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されます。セーブポイント（次の例の `my_savepoint`）を含める場合：

```
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消去されます。

指定し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。

6.5.26 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アボートし、サーバーの状態を指定され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、オリジナル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、マルチプルのセーブポイントが含まれる場合があります。

`my_savepoint` 名前付け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;
```

6.5.27 選択

ECPGPlus は、INTO host_variables 句を提供することにより、SQL SELECT ステートメントのサポートを拡張します。この句により、指定された情報を Advanced Server 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（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フォームをとることができます。

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 URL で入手可能な PostgreSQL Core 文書に記載されている 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/ 静的/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。例、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
```

(次のページに続く)

(前のページからの続き)

```
* 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マーカー（疑問符）を含めることもできます。例、次のクエリーには、WHERE 句にパラメータ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 パラメータマーカーに値を指定できます。

6.5.28 接続を設定

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マルチプル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番目のフォーム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、接続名前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。例、次のステートメントは AT connection_name 句が含まれていないため、デフォルトの接続を使用します。：

```
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前を指定するため、デフォルトの接続を使用しません。

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

例、マルチプル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、ロール alice 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。

6.5.29 SET DESCRIPTOR

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フォーム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、ディスクリプタ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。最初のフォームは次のとおりです。

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

2 番目のフォームは次のとおりです。

```
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ごとに 1 回実行されます。

descriptor_name は、ディスクリプタの名前（単一引用符で囲まれた文字列リテラルとして）、またはディスクリプタ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ディスクリプタします。

- column_number は、ディスクリプタ内の変数の位置を識別します。
- descriptor_item は、ディスクリプタアイテム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

例、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ーのプロンプトをユーザ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ーを実行するには、最初にクエリーの **prepare**（クエリーの構文のパーズと検証）を行い、次に **EXEC SQL DESCRIBE INPUT** ステートメントを使用してクエリーで見つかった **describe input** パラメータを使用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';
```

クエリーを記述した後、**query_params** ディスクリプタには、クエリーに必要な各パラメータに関する情報が含まれます。

この例では、ユーザが入力したと仮定します。

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

この場合、ディスクリプタは、**sal > ?**用と **job = ?**用の2つのパラメーターを記述します。

クエリ内のパラメータマーカー（疑問符）の数（したがって、クエリー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、次を使用しクエリー。

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

次に、**EXEC SQL GET DESCRIPTOR** を使用して各パラメータの名前を取得できます。また、**EXEC SQL GET DESCRIPTOR** を使用して、ディスクリプタから各パラメータのタイプ（パラメーターの数とともに）を取得することも、各 **value** を文字の並びのフォームで指定することもできます。ECPG はその文字列を必要なデータタイプに変換します。

最初のパラメータのデータタイプは **numeric** です。2 番目のパラメータのタイプは **varchar** です。最初のパラメータの名前は **sal** です。2 番目のパラメータの名前は **job** です。

次に、各パラメータをループして、ユーザに値を入力するように求め、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、クエリー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、サーバー上でプリペアドステートメントたステートメントを実行します。

6.5.30 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（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フォームをとることができます。

ステートメントが実行され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 URL で入手可能な PostgreSQL Core 文書に記載されている 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/ 静的/sql-update.html> >

ホスト変数は、値を指定する句で使用できます。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、従業員のジョブ記述 (:`ename` ホスト変数に含まれる値に変更し、現在の給与に:`increase` ホスト変数の値を掛けることで従業員の給与を増やします。

```
EXEC SQL UPDATE emp
* SET job = :new_job, sal = sal * :increase
* WHERE ename = :ename;
```

拡張された UPDATE ステートメントでは、入力値が許可される句にパラメータマーカー（疑問符）を含めることもできます。例、WHERE 句にパラメータ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、プロンプト列の新しい値をユーザに求め、:`ename` で指定された従業員の sal 列の増分量を提供できます。

6.5.31 いつでも

WHENEVER ステートメントを使用して、SQL エラーまたはワーニング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
<i>NOT FOUND</i>	サーバーは、行を返さない `SELECT` に遭遇したとき、または `FETCH` が結果セットの最後に到達したときに、`NOT FOUND` 条件を返します。
<i>SQLERROR</i>	サーバーは、SQL ステートメントによって返された重大なエラーを検出すると、`SQLERROR` 条件を返します。
<i>SQLWARNING</i>	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ないワーニングを検出すると、`SQLWARNING` 条件を返します。
	NA

次の表では、クライアントが `condition` に遭遇した結果のアクションについて説明します。

Action	Description
<i>CALL function</i> <i>[(args)]</i>	名前付けた `function` 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。
<i>CONTINUE</i>	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。
<i>DO BREAK</i>	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、`loop` または `switch` ステートメントに表示される場合があります。実行されると、break ステートメントは `loop` または `switch` ステートメントを終了します。
<i>DO CONTINUE</i>	C `continue` ステートメントを発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。 `continue` ステートメントはループ内にのみ存在でき、実行されると、制御フローがループの先頭に結果ます。
<i>DO function</i> <i>[(args)]</i>	名前付けた `function` 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。
<i>GOTO label</i> or <i>GO TO label</i>	`label` 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。
<i>SQLPRINT</i>	標準エラーにメッセージをプリント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。
<i>STOP</i>	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。
	NA

次のコードは、クライアントアプリケーションでフラグメント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、ユーザに問題をワーニングした後、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;
```

クエリー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ファンクションを呼び出すには、次のコードを含めます。

```
EXEC SQL WHENEVER NOT FOUND CALL error_handler(__LINE__);
```