



EDB

ECPGPlus

リリース *15.2.0*

EnterpriseDB Corporation

2023 年 03 月 03 日

目次

第 1 章 PostgreSQL の互換性	3
1.1 ECPGPlus overview	3
1.1.1 舞台裏	5
1.1.2 インストールと構成	7
1.1.3 makefile の作成	7
1.1.4 ECPGPlus のコマンドラインオプション	9
1.2 Using embedded SQL	10
1.2.1 例：単純なクエリ	10
1.2.2 例：カーソルを使用して結果セットを処理する	14
1.3 Using descriptors	18
1.3.1 例：記述子を使用してデータを返す	20
1.4 Building and executing dynamic SQL statements	30
1.4.1 例：パラメーターなしで非クエリステートメントを実行する	31
1.4.2 例：指定された数のプレースホルダーを使用して非クエリステートメントを実行する	33
1.4.3 例：既知の数のプレースホルダーを使用してクエリを実行する	36
1.4.4 例：変数の数が不明なクエリを実行する	39
1.5 Error handling	48
1.5.1 sqlca でのエラー処理	48
1.5.2 EXEC SQL WHENEVER	52
1.5.3 例	53
1.6 Reference	54
1.6.1 C プリプロセッサ指示子	54
1.6.2 サポートされている C データ型	57
1.6.3 タイプコード	58
1.6.4 SQLDA 構造	59
1.6.5 ECPGPlus ステートメント	62

ECPGPlus を作成するための EDB 拡張 ECPG (PostgreSQL プリコンパイラ)。ECPGPlus を使用すると、EDB Postgres Advanced Server データベースに接続しているときに、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 コードの構文がチェックされ、C に変換されます。

ECPGPlus は以下をサポートしています。

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

ECPGPlus の Pro*C 互換性の一部として、`BEGIN DECLARE SECTION` および `END DECLARE SECTION` 指示子を含める必要はありません。

第1章 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 ステートメントと 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。

はじめに 概要 using_embedded_sql using_descriptors building_executing_dynamic_sql_statements error_handling リファレンス結論

1.1 ECPGPlus overview

ECPGPlus を作成するための EDB 拡張 ECPG (PostgreSQL プリコンパイラ)。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。変換を実行するときに、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ときに埋め込み 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 文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：

1. プログラム（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 を呼び出して C プログラム（`my_program.c`）を生成する C コードに変換します。

1. 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。

1. オブジェクトファイル（`my_program.o`）、`ecpglib` ライブラリファイル、およびその他の必要なライブラリをリンカーに渡します。リンカーは実行可能ファイル（`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、セマンティクスは検証できません。たとえば、プリプロセッサは INSERT ステートメントが構文的に正しいことを確認しますが、INSERT ステートメントで指定されたテーブルが存在することを確認できません。

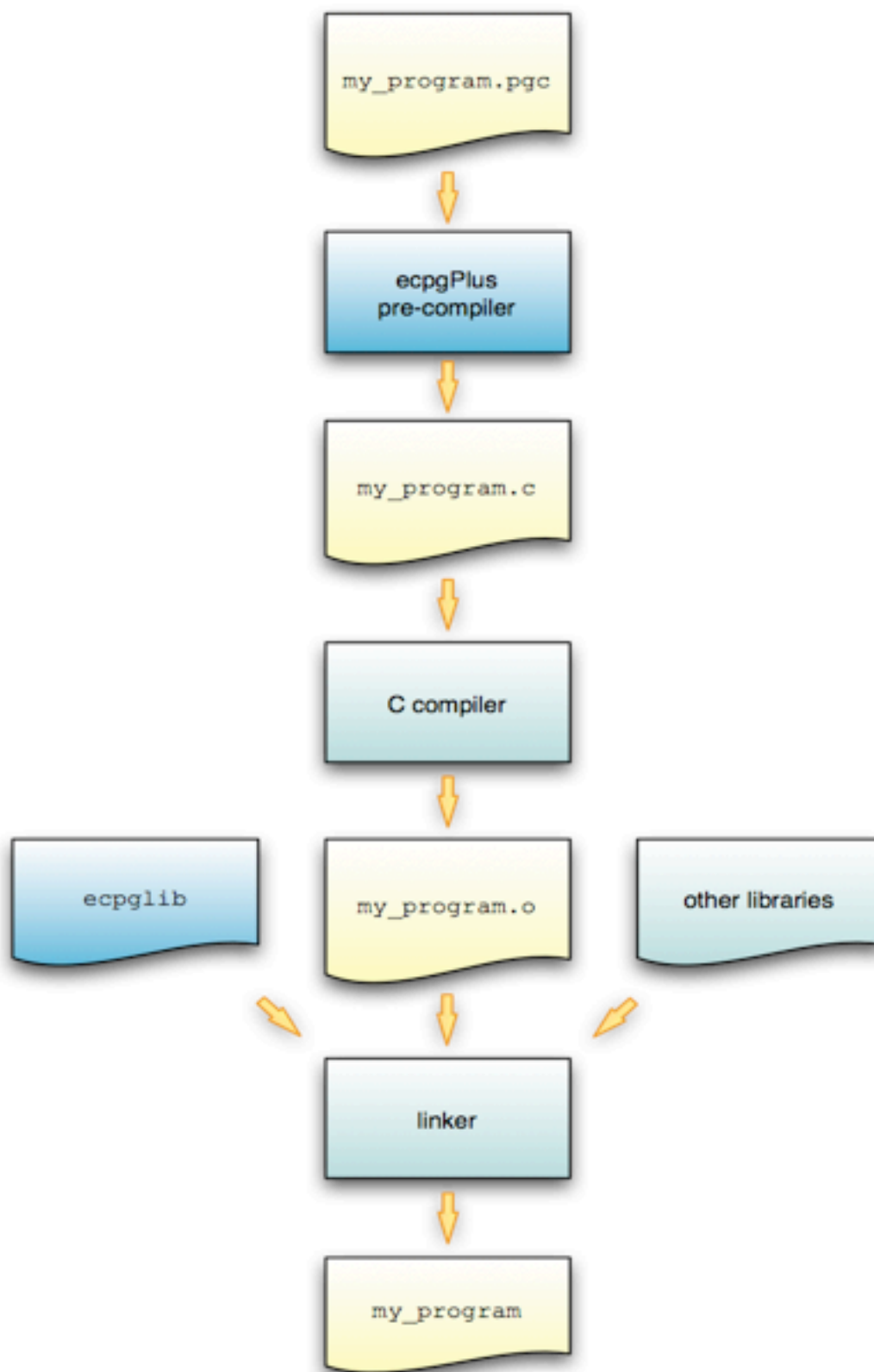


図 1: Compilation of a program containing embedded SQL statements

1.1.1 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る SQL。

- C プリプロセッサ指示子
- C 宣言（変数、型、関数など）
- C 定義（変数、型、関数など）
- 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)
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 }
```

このコードフラグメントでは：

- 1 行目は C プリプロセッサへの指示子を指定します。

C プリプロセッサ指示子は、解釈または無視できます。このオプションは、ECPGPlus を呼び出す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。いずれの場合も、ECPGPlus は各 C プリプロセッサ指示子を変更なしで出力ファイル（4）にコピーします。ソースファイルで見つかった C プリプロセッサ指示子は、出力ファイルに表示されます。

- 2 行目は、SQL プリプロセッサへの指示を指定します。

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。

- 4～6 行目は C 宣言を含みます。

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き、変更なしで出力ファイルにコピーされます。

- 10～14 行目は、embedded- SQL の宣言部です。

SQL コードで参照する C 変数は、**ホスト変数**と呼ばれます。ECPGPlus プリプロセッサを Pro*C モード (-C PROC) で呼び出すと、SQL 文で C 変数を参照できます。それ以外の場合は、各ホスト変数を BEGIN/END DECLARATION SECTION ペアで宣言する必要があります。

- 16～19 行目は SQL 文です。

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

- 21～23 行目は C コードです。

C コードは変更なしで出力ファイルにコピーされます。

任意の SQL ステートメントに EXEC SQL をプレフィックスします。SQL ステートメントは、次の (引用符で囲まない) セミicolon まで拡張されます。例：

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

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

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

プリプロセッサはこのコードフラグメントを検出すると、C コード (最初の行と最後の行) を変換せずに出力ファイルに渡し、各 EXEC SQL ステートメントを `ecpglib` ファンクションへ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" );
```

1.1.2 インストールと構成

Windows、ECPGPlus は EDB Postgres Advanced Server インストールウィザードによって Database Server コンポーネントの一部としてインストールされます。Linux では、`edb-asxx-server-devel` RPM パッケージでインストールします。ここで、`xx` は EDB Postgres 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 Windows 上の次の場所にあります。

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

Linux では、次の場所にあります。

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

ECPGPlus コンパイラを呼び出すとき、実行可能ファイルは検索パス（Windows では `%PATH%`、Linux では `$PATH`）にある必要があります。たとえば、次のコマンドは、ECPGPlus 実行ファイル `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
```

1.1.3 makefile の作成

makefile には、埋め込み SQL を含む C で記述されたプログラムを C プログラムに変換する方法を make ユーティリティに指示する一連の命令が含まれています。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 の指示を含む make ファイル

次のコードは、このドキュメントに含まれるサンプルのメイクファイルの例です。サンプルコードを使用するには、ソースコードファイルが含まれるディレクトリにある `makefile` という名前のファイルに保存します。

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

(次のページに続く)

(前のページからの続き)

```
LDLIBS += $(LIBPATH) -lecpg -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 EDB Postgres Advanced Server に同梱されています。

make は、C コンパイラの実行時に CFLAGS 変数を使用し、リンカーを呼び出すときに LDFLAGS および LDLIBS を使用することを知っています。ECPG プログラムは、ECPG ランタイムライブラリ (-lecpg) および libpq ライブラリ (-lpq) に対して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 make ファイルは、.pgc または .pc ファイルを C プログラムに変換する方法を make に示しています。makefile の 2 行は、ソースファイルのコンパイル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、ECPGPlus の拡張機能なしで、コミュニティ ECPG を使用して、.pgc (おそらく 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指示します。これは、INCLUDES 変数の値と .pgc ファイルの名前を使用して、SQL コードを C に変換するようにコンパイラーに指示します。

```
.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指示します。-c フラグを使用して ECPG プリコンパイラーを呼び出し、SQL コードを C に変換するようにコンパイラーに指示します。また、-C PROC フラグを使用して、INCLUDES 変数と .pgc ファイルの名前。

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

を使用します。

```
make customer_list
```

make ユーティリティ：

1. 現在のディレクトリにある makefile を調べます。
2. 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発見します。
3. `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。

1.1.4 ECPGPlus のコマンドラインオプション

サンプル make ファイルでは、ECPGPlus を呼び出して Pro*C 互換モードで ECPGPlus を呼び出すときに、`make` に `-C` オプションが含まれます。

ECPG 構文に加えて、コマンドラインで `-C PROC` キーワードを含めると、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
```

使用可能な他の ECPGPlus オプションの完全なリストを表示するには、ECPGPlus インストールディレクトリで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
<code>-c</code>	Embedded SQL コードから C コードを自動生成します。
<code>-C mode</code>	<code>-C</code> オプションを使用して、互換性モードを指定します。 <code>

 'INFORMIX'

 'INFORMIX_SE'

 'PROC</code>
<code>-D symbol</code>	プリプロセッサ*シンボル*を定義します。 <code>

 *PROC モード*でコンパイルする場合、<code>-D*</code>キーワードはサポートされていません。代わりに、Oracle スタイルの <code>'DEFINE=*</code>句を使用します。</code>
<code>-h</code>	ヘッダーファイルを解析します。このオプションにはオプション <code>'-c'</code> が含まれます。
<code>-i</code>	パースシステム、インクルードファイルも。
<code>-I directory</code>	<code>'include'</code> ファイルを*ディレクトリ*で検索します。
<code>-o outfile</code>	結果を*outfile*に書き込みます。
<code>-r option</code>	実行時の動作を指定します。 <code>option:</code> <code>br />
 'questionmarks' - プレースホルダーとしての疑問符の使用を許可します。 <code>

 'usebulk' - ホスト変数配列で動作する <code>'INSERT'</code>、<code>UPDATE</code>、および <code>'DELETE'</code> ステートメントの一括処理を有効にします。</code></code>

次のページに続く

表 1 – 前のページからの続き

Option	Description
--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。

注釈: ECPGPlus を呼び出すときに出力ファイル名を指定しない場合、出力ファイル名はファイル名から`.pgc` 拡張子を削除し、`.c` を追加することによって作成されます。

1.2 Using embedded SQL

1.2.1 例：単純なクエリ

最初のコードサンプルは、単一の行を返し、結果をホスト変数のグループに保存する `SELECT` ステートメントを実行する方法を示しています。ホスト変数を宣言した後、ハードコードされたロール名と関連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、単純な結果セットを画面に出力し、接続を閉じます。

```

/ ****  ****  ****  ****  ****  ****  ****  ****  ****  ****  ****  ****  ****  ****  ****
→ ****
* print_emp.pgc
*
*/
# include <stdio.h>

int main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
        int v_empno;
        char v_ename[40];
        double v_sal;
        double v_comm;
        short v_comm_ind;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL WHENEVER SQLERROR sqlprint;

    EXEC SQL CONNECT TO edb
        USER alice IDENTIFIED BY 1safepwd;

```

(次のページに続く)

(前のページからの続き)

```

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

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

このコードサンプルでは、C stdio ライブラリのプロトタイプと型定義を含めて、main 関数を宣言します。

```

# include <stdio.h>

int main(void)
{
```

次に、アプリケーションは、データベースサーバとの対話に使用される一連のホスト変数を宣言します。

```

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

PROC モードでコードをプリコンパイルする場合は、BEGIN DECLARE…END DECLARE セクションを省略できます。ホスト変数の宣言の詳細については、

[ホスト変数の宣言](#) を参照してください。

宣言セクションの各変数に関連付けられたデータ型は C データ型です。サーバーとクライアントアプリケーション間で渡されるデータは、互換性のあるデータ型を共有する必要があります。データ型の詳細については、

サポートされている C データ型 を参照してください。

次のステートメントは、サーバーにエラーを処理する方法を指示します。

```
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した場合、サーバーは `ecpglib` 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、EDB Postgres Advanced Server との接続を確立します。

```
EXEC SQL CONNECT TO edb
USER alice IDENTIFIED BY 1safepwd;
```

この例では、クライアントアプリケーションは `alice` という名前のロールと `1safepwd` のパスワードを使用して `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を実行します。

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

クエリは、従業員番号 7369 に関する情報を返します。

`SELECT` ステートメントは、`INTO` 句を使用して、取得した値（`empno`、`ename`、`sal`、および `comm` 列）を `:v_empno`、`:v_ename`、`:v_sal`、および `:v_comm` スル変数（および `:v_comm`）に割り当てます。取得された最初の値は `INTO` 句にリストされた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、従業員が稼いだコミッション値が含まれ、潜在的に `NULL` 値が含まれます。このステートメントには、`INDICATOR` キーワードと、スル標識を保持するホスト変数が含まれています。

このコードは `null` インジケータをチェックし、適切な結果を表示します。

```
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);
```

スル標識が `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。スル標識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した場合、ホスト変数（ヌル標識を除く）には意味のある値が含まれません。NULL の可能性がある値を識別するには、ヌル標識を使用する必要があります。

コードサンプルの最後のステートメントは、サーバーへの接続を閉じます。

```
EXEC SQL DISCONNECT;
}
```

インジケータ変数の使用

前の例には、comm 列（サーバーから返された場合）の値が NULL であった行を識別する**インディケータ変数**が含まれています。標識変数は、先行する変数の内容が 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設定されます。標識変数には、次の値を含めることができます。

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 句に標識変数を含める場合、オプションの INDICATOR キーワードを含める必要はありません。

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

ホスト変数の宣言

SQL ステートメント内で値を指定できる任意の時点で、**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にデータを返すために使用できる C 変数です。ホスト変数は次のとおりです。

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

以下のコード片は、PROC モードおよび非 PROC モードでコンパイルされたコードでのホスト変数の使用を示しています。SQL ステートメントは、dept テーブルに行を追加し、変数 v_deptno、v_dname、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している場合、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 関数パラメーターをホスト変数として使用できます。

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

PROC モードでコンパイルしていない場合は、EXEC SQL BEGIN DECLARE SECTION および EXEC SQL END DECLARE SECTION 指示子で埋め込み変数宣言を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 ステートメントに 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 句に型互換性のあるターゲット変数が必要です。これは、単一の行を取得する簡単な例です。複数の行を取得するには、次の例に示すように、カーソルを定義する必要があります。

1.2.2 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。
2. OPEN CURSOR ステートメントを使用して、カーソルを開きます。
3. FETCH ステートメントを使用して、カーソルからデータを取得します。
4. CLOSE CURSOR ステートメントを使用して、カーソルを閉じます。

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

```

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

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

    EXEC SQL WHENEVER SQLERROR sqlprint;

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

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

    EXEC SQL OPEN employees;

    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);
    }
}

```

(次のページに続く)

(前のページからの続き)

```

}
EXEC SQL CLOSE employees;
EXEC SQL DISCONNECT;
}
/ *****
→*****

```

このコードサンプルでは、C stdio ライブラリのプロトタイプと型定義を含めて、main 関数を宣言します。

```

# include <stdio.h>

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

```

次に、アプリケーションは、データベースサーバとの対話に使用される一連のホスト変数を宣言します。

```

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

```

argv[] は、ユーザーがクライアントアプリケーションを実行するときに入力され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、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。この例では、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を省略しています。この宣言は、username および password の値を初期化し、ユーザーがクライアントアプリケーションを呼び出したときに入力された値に設定します。

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] を参照できると言うかもしれませんが、それはうまくいきません。PROC モードでコンパイルする場合を除き、すべてのホスト変数は BEGIN/END DECLARE SECTION で宣言する必要があります。argv はファンクションパラメーター（自動変数ではありません）であるため、BEGIN/END DECLARE SECTION で宣言できません。PROC モードでコンパイルしている場合、SQL ステートメントで C 変数を参照できます。

次のステートメントは、ECPGPlus またはデータベースサーバーから返されたエラーメッセージのテキストを出力して、SQL エラーに応答するように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 EDB Postgres 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 ホスト変数で見つかった値を使用して、接続時にサーバーに対してアプリケーションを認証します。

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

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

employees には、emp テーブルに対する SELECT ステートメントの結果セットが含まれています。クエリは、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に、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 を実行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、クライアントアプリケーションはカーソルの内容をループします。クライアントアプリケーションには、カーソルの最後に到達したときにサーバーに break（つまり、ループを終了）するように指示する 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 ヌル標識）に割り当てます。カーソルの最初の値は INTO 句にリストされている最初の変数に割り当てられ、2 番目の値は 2 番目の変数に割り当てられます。

FETCH ステートメントには、INDICATOR キーワードと、ヌル標識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 値が含まれる場合、v_comm_ind はゼロ以外の値に設定され、列が NULL であることを示します。

次に、コードはヌルインジケータをチェックし、適切な結果を表示します。

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

(次のページに続く)

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

ヌル標識が 0（つまり、false）の場合、v_comm には意味のある値が含まれ、printf 関数は手数料を表示します。ヌル標識にゼロ以外の値が含まれる場合、comm は NULL になり、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した場合、ホスト変数（ヌル標識を除く）には意味のある値が含まれません。NULL になる可能性のある値にはヌル標識を使用する必要があります。

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

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

1.3 Using descriptors

ダイナミック SQL を使用すると、クライアントアプリケーションは実行時に構成される SQL ステートメントを実行できます。この機能は、クライアントアプリケーションを作成するときにステートメントの内容または形式がわからない場合に役立ちます。ECPGPlus では、識別子（テーブル名、列名、インデックス名など）の代わりにホスト変数を使用できません。代わりに、動的 SQL ステートメントを使用して情報を含む文字列を作成し、その文字列を実行します。文字列は、クライアントとサーバーの間で記述子の形式で渡されます。記述子は、データとデータの形状に関する情報の両方を含むデータ構造です。

クライアントアプリケーションは、GET DESCRIPTOR ステートメントを使用して記述子から情報を取得する必要があります。動的 SQL を使用するクライアントアプリケーションの基本的なフローは次のとおりです。

1. ALLOCATE DESCRIPTOR ステートメントを使用して、結果セット（選択リスト）の記述子を割り当てます。
2. ALLOCATE DESCRIPTOR ステートメントを使用して、入力パラメーター（バインド変数）の記述子を割り当てます。
3. SQL ステートメントのテキストを取得、アセンブル、または計算します。
4. PREPARE ステートメントを使用して、SQL ステートメントの構文を解析および確認します。
5. DESCRIBE ステートメントを使用して、選択リストを選択リスト記述子に記述します。
6. DESCRIBE ステートメントを使用して、入力パラメーターをバインド変数記述子に記述します。
7. 各入力パラメーターの値の入力を求めます（必要な場合）。SET DESCRIPTOR ステートメントを使用して、値を記述子に割り当てます。
8. DECLARE CURSOR ステートメントを使用して、ステートメントのカーソルを定義します。
9. OPEN CURSOR ステートメントを使用して、ステートメントのカーソルを開きます。10. FETCH ステートメントを使用して、カーソルから各行をフェッチし、各行を選択リスト記述子に保存します。11.

GET DESCRIPTOR コマンドを使用して選択リスト記述子を調べ、現在の行の各列の値を見つけます。

12. CLOSE CURSOR ステートメントを使用して、カーソルを閉じ、カーソルリソースを解放します。

記述子には、これらの属性を含めることができます。

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.
<i>INDICATOR</i>	<i>integer</i>	Indicates a <i>NULL</i> or truncated value.
<i>KEY_MEMBER</i>	<i>integer</i>	Unused (returns <i>FALSE</i>).
<i>LENGTH</i>	<i>integer</i>	The data length (as stored on server).
<i>NAME</i>	<i>string</i>	The name of the column in which the data resides.
<i>NULLABLE</i>	<i>integer</i>	Unused (returns <i>TRUE</i>).
<i>OCTET_LENGTH</i>	<i>integer</i>	The data length (in bytes) as stored on server.
<i>PRECISION</i>	<i>integer</i>	The data precision (if the data is of <i>numeric</i> type).
<i>RE- TURNED_LENGTH</i>	<i>integer</i>	Actual length of data item.
<i>RE- TURNED_OCTET_LENGTH</i>	<i>integer</i>	Actual length of data item.
<i>SCALE</i>	<i>integer</i>	The data scale (if the data is of <i>numeric</i> type).

次のページに続く

表 3 – 前のページからの続き

Field	Type	Attribute description
<i>TYPE</i>	<i>integer</i>	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>

1.3.1 例：記述子を使用してデータを返す

次の単純なアプリケーションは、エンドユーザーが入力した SQL ステートメントを実行します。コードサンプルは次のことを示しています。

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

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

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

たとえば、ユーザーは次のコマンドでサンプルを呼び出すことができます。

```

./exec_stmt edb "SELECT * FROM emp"

/ *****
→ *****
/* exec_stmt.pgc

*
*/

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

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

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

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

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

    EXEC SQL CONNECT TO :db;

    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

```

(次のページに続く)

```
{  
    int row;  
  
    EXEC SQL ALLOCATE DESCRIPTOR row_desc;  
    EXEC SQL DECLARE my_cursor CURSOR FOR query;  
    EXEC SQL OPEN my_cursor;  
  
    for( row = 0; ; row++ )  
    {  
        EXEC SQL BEGIN DECLARE SECTION;  
        int col;  
        EXEC SQL END DECLARE SECTION;  
        EXEC SQL FETCH IN my_cursor  
            INTO SQL DESCRIPTOR row_desc;  
  
        if( sqlca.sqlcode != 0 )  
            break;  
  
        if( row == 0 )  
            print_meta_data( "row_desc" );  
  
        printf("[RECORD %d]\n", row+1);  
  
        for( col = 1; col <= col_count; col++ )  
        {  
            EXEC SQL BEGIN DECLARE SECTION;  
            short ind;  
            varchar val[40+1];  
            varchar name[20+1];  
            EXEC SQL END DECLARE SECTION;  
  
            EXEC SQL GET DESCRIPTOR row_desc  
                VALUE :col  
                :val = DATA, :ind = INDICATOR, :name = NAME;  
  
            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" );  
    }  
}
```

(前のページからの続き)

```

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

exit( 0 );
}

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

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

EXEC SQL GET DESCRIPTOR :desc :col_count = count;

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

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

```

(次のページに続く)

```

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

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

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

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

/ *****
→*****

```

コードサンプルは、C の `stdio` および `stdlib` ライブラリ、SQL データ型シンボル、および SQLCA (SQL 通信領域) 構造体のプロトタイプと型定義を含めることから始まります。

```

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

```

このサンプルでは、最小限のエラー処理が提供されます。アプリケーションで SQL エラーが発生すると、エラーメッセージが画面に出力されます。

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

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

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

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

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

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

次の宣言セクションは、アプリケーションが接続するデータベースの名前、SQL ステートメントの内容、および結果セット内の列数のホスト変数（存在する場合）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、デフォルトの資格情報を使用してデータベースに接続します。

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

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

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

アプリケーションは、PREPARE ステートメントを使用して、ユーザーが指定した文字列の構文を確認します。

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

また、DESCRIBE ステートメントを使用して、クエリのメタデータを SQL 記述子に移動します。

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

次に、アプリケーションは記述子を調べて結果セットの列数を検出し、それをホスト変数 col_count に保存します。

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

列数がゼロの場合、エンドユーザーは SELECT ステートメントを入力していません。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
{
```

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

```
int row;
```

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

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

アプリケーションは、準備されたステートメントのカーソルを宣言して開きます。

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

結果セットの行をループします。

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

次に、FETCH を使用して、カーソルから次の行を記述子に取得します。

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

アプリケーションは、FETCH が失敗しなかったことを確認します。FETCH が失敗すると、アプリケーションは結果セットの最後に到達して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。その場合、アプリケーションは行のメタデータを出力します。

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

次に、行番号を含むレコードヘッダーを出力します。

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

次に、行の各列をループします。

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

アプリケーションは行記述子 (row_desc) を調べて、列値:val、ヌル標識:ind、および列名:name を前に宣言したホスト変数にコピーします。コンマ区切りのリストを使用して、記述子から複数の項目を取得できます。

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

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

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

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

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

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

exit( 0 );
}
```

print_meta_data() ファンクションは、記述子からメタデータを抽出し、各列の名前、データ型、長さを出力します。

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

アプリケーションはホスト変数を宣言します。

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

次に、アプリケーションは、データ型値（`numeric`）をデータ型名にマップする文字列の配列を定義します。記述子で見つかった数値を使用して、この配列にインデックスを付けます。たとえば、指定された列の型が2であることがわかった場合、`types[2]`と書くことでその型の名前（`NUMERIC`）を見つけることができます。

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

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

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

アプリケーションは、このアプリケーションの先頭に定義された列ヘッダーを出力します。

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

次に、記述子で見つかった各列をループし、各列の名前、タイプ、および長さを出力します。

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

現在の列の名前、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[]` 配列で見つかった型コード）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";
```

次に、列番号、名前、型名、および長さを出力します。

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

次のコマンドでサンプル アプリケーションを呼び出します。

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

アプリケーションは次を返します。

```
__OUTPUT__
col field          data          ret
num name          type          len
-----
01: empno         NUMERIC          0004
02: ename         CHARACTER_VARYING 0004
03: job           CHARACTER_VARYING 0007
```

(次のページに続く)

```

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

```

2 rows

1.4 Building and executing dynamic SQL statements

次の例は、動的 SQL ステートメントを構築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さまざまな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、Oracle 動的 SQL 方法 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、Oracle 動的 SQL 方法 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み、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、Oracle 動的 SQL 方法 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、不明な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、Oracle 動的 SQL 方法 4 で使用される手法に対応しています。

1.4.1 例：パラメーターなしで非クエリステートメントを実行する

この例では、EXECUTE IMMEDIATE コマンドを使用して SQL ステートメントを実行する方法を示します。ステートメントのテキストは、アプリケーションを実行するまでわかりません。EXECUTE IMMEDIATE を使用して、結果セットを返すステートメントを実行することはできません。EXECUTE IMMEDIATE を使用して、パラメータープレースホルダーを含むステートメントを実行することはできません。

EXECUTE IMMEDIATE ステートメントは、実行のたびに SQL ステートメントを解析して計画するため、アプリケーションのパフォーマンスに悪影響を及ぼす可能性があります。同じステートメントを繰り返し実行する予定がある場合は、

例：指定された数のプレースホルダーを使用して非クエリステートメントを実行する で説明されている PREPARE/EXECUTE

テクニックの使用を検討してください。

```

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

static void handle_error(void);

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

    EXEC SQL WHENEVER SQLERROR DO handle_error();

    EXEC SQL CONNECT :argv[1];

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

    EXEC SQL EXECUTE IMMEDIATE :insertStmt;

    fprintf(stderr, "ok\n");

    EXEC SQL COMMIT RELEASE;

    exit(EXIT_SUCCESS);
}

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

```

(次のページに続く)

(前のページからの続き)

```
EXEC SQL WHENEVER SQLERROR CONTINUE;
EXEC SQL ROLLBACK RELEASE;

exit(EXIT_FAILURE);
}

/ ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****
→***/
```

コードサンプルでは、C の `stdio`、`string`、および `stdlib` ライブラリのプロトタイプと型定義を含め、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;
```

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

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

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

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

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

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

EXEC SQL EXECUTE IMMEDIATE :insertStmt;
```

`EXECUTE IMMEDIATE` コマンドが失敗すると、ECPGPlus は `handle_error()` ファンクションを呼び出します。これは、ユーザーにエラーメッセージを表示した後、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ユーザーにメッセージ (`ok`) を表示し、変更をコミットし、サーバーから切断して、アプリケーションを終了します。

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

EXEC SQL COMMIT RELEASE;
```

(次のページに続く)

(前のページからの続き)

```
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内容を出力し、エラーハンドラーをリセットし、変更をロールバックし、データベースから切断して、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);
}
```

1.4.2 例：指定された数のプレースホルダーを使用して非クエリステートメントを実行する

既知の数のパラメータープレースホルダーを含む非クエリコマンドを実行するには、最初にステートメントを `PREPARE`（ステートメントハンドルを提供）し、次にステートメントハンドルを使用してステートメントを `EXECUTE` する必要があります。アプリケーションはステートメントを実行するときに、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、毎回異なる値が提供される場合があります。

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

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

static void handle_error(void);

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

(次のページに続く)

```

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);
}
/ ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****
->***/

```

コードサンプルでは、C の `stdio`、`string`、`stdlib`、および `sqlca` ライブラリのプロトタイプと型定義を含め、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;

```

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

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

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

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

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

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

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

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

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

EXECUTE コマンドが失敗すると、ECPGPlus は `handle_error()` ファンクションを呼び出します。これは、ユーザーにエラーメッセージを表示した後、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ユーザーにメッセージ (ok) を表示し、変更をコミットし、サーバーから切断して、アプリケーションを終了します。

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

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS);
}
```

ECPGPlus は、SQL エラーが発生するたびに `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内容を出力し、エラーハンドラーをリセットし、変更をロールバックし、データベースから切断して、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);
}
```

1.4.3 例：既知の数のプレースホルダーを使用してクエリを実行する

この例では、既知の数の入力パラメーターと結果セットの既知の列を使用してクエリを実行する方法を示します。このメソッドは、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、カーソルを開き、結果セットを反復処理します。

```

/ ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****
→ ***/
# 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;

```

(次のページに続く)

(前のページからの続き)

```

    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);
}

/ ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****
→ ***/

```

コードサンプルでは、C の `stdio`、`string`、`stdlib`、`stdbool`、および `sqlca` ライブラリのプロトタイプと型定義を含め、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];

```

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

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

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

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

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

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

次に、プログラムはカーソル `empCursor` を宣言して開き、準備された `SELECT` ステートメントのパラメーターマーカーをユーザー指定の値に置き換えます。`OPEN` ステートメントには、クエリ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る `USING` 句が含まれています。

```
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繰り返して、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);
}
```

次に、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()` 関数を呼び出します。`handle_error()` ファンクションは、エラーメッセージの内容を出力し、エラーハンドラーをリセットし、変更をロールバックし、データベースから切断して、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);
}
```

1.4.4 例：変数の数が不明なクエリを実行する

この例は、結果セットに不明な数の入力パラメーターまたは列を使用してクエリを実行する方法を示しています。この種のクエリは、ユーザーにクエリのテキストの入力を求める場合、またはユーザーが多くの条件（フィルター）から選択するフォームからクエリが組み立てられる場合に発生する可能性があります。

```

/ ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****
→ ***/
# include <stdio.h>
# include <stdlib.h>
# include <sqllda.h>
# include <sqlcpr.h>

SQLDA *params;
SQLDA *results;

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

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

    EXEC SQL WHENEVER SQLERROR sqlprint;

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

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

    EXEC SQL PREPARE stmt FROM :stmtText;

    EXEC SQL DECLARE dynCursor CURSOR FOR stmt;

    bindParams();

    EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

```

(次のページに続く)

```
    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);
        }
    }
}

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;
```

(次のページに続く)

(前のページからの続き)

```

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;
}

EXEC SQL WHENEVER NOT FOUND DO break;

```

(次のページに続く)

```

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 の `stdio` および `stdlib` ライブラリのプロトタイプと型定義を含めることから始まります。さらに、プログラムには `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 構造で見つかったメタデータを照会す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 構造へのポインターを宣言します。最初の SQLDA 構造体 (`params`) は、動的クエリテキストで見つかったパラメーターマーカーのメタデータを記述するために使用されます。2 番目の SQLDA 構造体 (`results`) には、メタデータと動的クエリの実行によって取得された結果セットの両方が含まれています。

```

SQLDA *params;
SQLDA *results;

```

次に、プログラムは2つのヘルパー関数を宣言します。これらは、コードサンプルの末尾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 および password）は、データベースサーバーへの接続に使用されます。3番目のホスト変数（stmtText）は、実行するクエリのテキストを含む NULL で終わる C 文字列です。これらの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ます。プログラムは実行を開始すると、エラーハンドラを設定してからデータベースサーバに接続します。

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

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

次に、プログラムは `sqlald()` 関数を呼び出して、各記述子に必要なメモリを割り当てます。各記述子には、次の配列へのポインターが含まれています。

- 列名
- インジケーター名
- データ型
- 長さ
- データ値

SQLDA 記述子を割り当てるときは、結果セットで検出されると予想される列の最大数（SELECT -list 記述子の場合）または動的クエリーテキスト（バインド変数記述子の場合）を指定します。この場合、20 個以下の列と 20 個のパラメーターが予想されることを指定します。また、各列またはパラメーター名、および各標識変数名の最大長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 構造体の完全な説明については、

[SQLDA 構造](#) を参照してください。

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

SELECT -list 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 `bindParam()` 関数を呼び出します。`bindParam()` ファンクションは、バインド記述子 (`params`) を調べ、動的クエリーで見つかった各パラメーターマーカーの代わりに値の入力を求めます。

```
bindParam();
```

最後に、プログラムはカーソルを開き（ユーザーが指定したパラメーター値を使用）、`displayResultSet()` ファンクションを呼び出して、クエリによって生成された結果セットを出力します。

```
EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

displayResultSet();
}
```

`bindParam()` ファンクションは、動的クエリーにパラメーターマーカーが含まれているかどうかを判断します。その場合、各パラメーターの値の入力をユーザーに求め、その値を対応するマーカーにバインドします。`DESCRIBE BIND VARIABLE` ステートメントは、各パラメーターマーカーを説明する情報を `params` `SQLDA` 構造体に入力します。

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

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

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

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

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

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

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

(次のページに続く)

(前のページからの続き)

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

特定のパラメーターの値の入力を求めた後、プログラムは次のように設定してその値をパラメーターにバインドします。

- 値のデータ型を示す `params->T[i]`
- `params->L[i]` を値の長さ (`fgets()` によって追加された末尾の改行文字をトリムするために 1 を引きます)
- `params->V[i]` は、ユーザーが提供する NULL で終わる文字列のコピーを指します

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

`displayResultSet()` ファンクションは、結果セットの各行をループし、各列で見つかった値を出力します。`displayResultSet()` は、`DESCRIBE SELECT LIST` ステートメントを実行することによって開始します。このステートメントは、結果セットの各列の説明を `SQLDA` 記述子 (`results`) に設定します。

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

動的ステートメントが列を返さない場合（つまり、動的ステートメントが `SELECT` ステートメントではない場合）、`results->F` には 0 が含まれます。ステートメントが記述子に収まるより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、その数はクエリによって返された列の数を示します。

```
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 で見つかった型コードをデコードするために、プログラムは sqlnul() ファンクションを呼び出します ([SQLDA 構造](#)の SQLDA 構造体の T メンバーの説明を参照してください)。sqlnul() へのこの呼び出しは、型コードのみを含むように results->T[col] を変更します (null 可能性フラグは null_permitted にコピーされます)。DESCRIBE SELECT LIST ステートメントは各列の型と各列の NULL 可能性を T 配列にエンコードするためです。

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

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

数値 (ここで results->T[col] = 2) の場合、プログラムは sqlprc() 関数を呼び出して、列の長さから精度と位取りを抽出します。文字列形式で数値を保持するために必要なバイト数を計算するために、displayResultSet() は精度 (つまり、最大桁数) で始まり、符号文字、小数点、および NULL 終了文字用に 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、プログラムはハードコードされた長さ 30 を使用します。実際のアプリケーションでは、必要な領域の量をより注意深く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` ターミネータ用に 1 バイトを追加します。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領域の量を計算した後、プログラムは次のことを行います。

- 値を保持するのに十分なメモリを割り当てます
- `results->V[col]` で見つかったバイト数を示すように `results->L[col]` を設定します
- `(results->T[col])` ~1 列の型コードを設定して、次の `FETCH` ステートメントに `NULL` で終了する文字列の形式で値を返すように指示します

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

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

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ーを定義します。

```
EXEC SQL WHENEVER NOT FOUND DO break;

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

プログラムは `FETCH` ステートメントを実行して、カーソルの次の行を `results` 記述子にフェッチします。`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場合）、ループの先頭の前に `EXEC SQL WHENEVER` 指示子が見つかったため、制御はループの最後に移動します。

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

`FETCH` ステートメントは、結果記述子の次のメンバーを移入します。

- `*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らない場合、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 (`*results->I[col]` が列の値が `NULL` であることを示している場合を除く)。

- `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。結果セットは調整されません。代わりに、各値はコンマで前の値と区切られます。

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

printf("\n");
}
}
}

/ ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****      ****
→***/
```

1.5 Error handling

ECPGPlus は、埋め込み SQL コードのエラーを検出して処理する 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次のことができます。

- `sqlca` データ構造でエラーメッセージを調べ、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処理を提供します。
- ECPGPlus コンパイラにエラー処理コードを追加するように指示する `EXEC SQL WHENEVER` 指示子を含めます。

1.5.1 `sqlca` でのエラー処理

`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` が使用するグローバル変数です。INSERT または SELECT ステートメントなどの SQL ステートメントを実行した後、`sqlca` の内容を検査して、ステートメントが正常に完了したかどうかを判断できます。

`sqlca` の構造は次のとおりです。

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
```

(次のページに続く)

(前のページからの続き)

```

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

```

次の指示子を使用して、`sqlca` 機能を実装します。

```
EXEC SQL INCLUDE sqlca;
```

`ecpg` 指示子をインクルードする場合、クライアントアプリケーションのヘッダー宣言で `sqlca.h` ファイルを `#include` する必要はありません。

EDB Postgres Advanced Server の `sqlca` 構造体には、次のメンバーが含まれています。

- `sqlcaid` — 文字列が含まれています: "SQLCA"。
- `sqlabc` — `sqlabc` には、`sqlca` 構造体のサイズが含まれています。
- `sqlcode` — `sqlcode` メンバーは SQL 92 で非推奨になりました。EDB Postgres Advanced Server は、下位互換性のために `sqlcode` をサポートしています。新しいコードを記述するときに `sqlstate` メンバーを使用します。

`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ます。負の値は、警告またはエラーを示します。

ステートメントがエラーなしで処理される場合、`sqlcode` には 0 の値が含まれます。ステートメントの実行中にクライアントアプリケーションでエラーまたは警告が発生した場合、`sqlcode` には返された最後の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 ステートメントが行を返さないか、影響しないことを示します。SQL 標準では他の `sqlcode` 値が定義されていないため、各条件に割り当てられる値はデータベースによって異なることに注意してください。

`sqlerrm` は `sqlca` に埋め込まれた構造体であり、2 つのメンバーで構成されています。

- `sqlerrml` — 現在 `sqlerrmc` に格納されているエラーメッセージの長さが含まれています。
- `sqlerrmc` — `sqlstate` に格納されているコードに関連付けられたヌル終了のメッセージテキストが含まれています。メッセージが 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` — 文字列 "NOT SET" が含まれています。

`sqlerrd` は、6 つの要素を含む配列です。

- `sqlerrd[1]` — 処理された行の OID が含まれています (該当する場合)。

- `sqlerrd[2]` — 処理または返された行の数が含まれます。
- `sqlerrd[0]`、`sqlerrd[3]`、`sqlerrd[4]`、`sqlerrd[5]` は未使用です。

`sqlwarn` は、8 文字を含む配列です。

- `sqlwarn[0]` — `sqlwarn` の他の要素が 'W' に設定されている場合、'W' の値が含まれます。
- `sqlwarn[1]` — 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、'W' の値が含まれます。
- `sqlwarn[2]` — クライアントアプリケーションで致命的でない警告が発生した場合、'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` は、クライアントアプリケーションからのステートメントの実行後の SQL 準拠のステータスコードを含む 5 文字の配列です。ステートメントがエラーなしで処理される場合、`sqlstate` には `00000` の値が含まれます。 `sqlstate` はヌル終了文字列ではありません。

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

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

sqlstate	sqlcode (depre- cated)	Symbolic name	Description
YE001	-12	ECPG_OUT_OF_MEMORY	メモリを使い果たしました。
YE002	-200	ECPG_UNSUPPORTED	プリプロセッサが認識でき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	は、コマンドが予想するよりも多くの変数を指定します。
07001, or 07002	-202	ECPG_TOO_FEW_ARGUMENTS	が指定した変数は、コマンドが予想するよりも少ないです。
21000	-203	ECPG_TOO_MANY_ROWS	は複数の行を返しましたが、ステートメントは単一の行を受け取るように準備されました。

次のページに続く

表 4 – 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、文字列値を数値形式に変換します。
42804	-205	<i>ECPG_UINT_FORMAT</i>	ホスト変数 (C コードで定義) が unsigned INT で、選択されたデータは unsigned INT に変換できない型です。`ecpglib` は、`strtoul()` ファンクションを使用して、文字列値を数値形式に変換します。
42804	-206	<i>ECPG_FLOAT_FORMAT</i>	ホスト変数 (C コードで定義) は FLOAT 型ですが、選択されたデータは FLOAT に変換できません。`ecpglib` は、`strtod()` ファンクションを使用して、文字列値を数値形式に変換します。
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 に保存できません。
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>	クライアントから返された NULL 値に NULL インジケータ変数が指定されませんでした。(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>	サーバーが配列を返しましたが、対応するホスト変数に配列を保存できません。
42804	-215	<i>ECPG_DATA_NOT_ARRAY</i>	サーバーは、配列値を必要とするホスト変数に配列ではない値を返しました。
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てしました。
YE002	-221	<i>ECPG_NOT_CONN</i>	クライアントアプリケーションは、割り当てられたが閉じられた接続を使用しようとしてしました。
26000	-230	<i>ECPG_INVALID_STMT</i>	声明は準備されていませんでした。

次のページに続く

表 4 – 前のページからの続き

sqlstate	sqlcode (depre- cated)	Symbolic name	Description
33000	-240	<i>ECPG_UNKNOWN_DESCRIPTOR</i>	指定された記述子が見つかりません。
07009	-241	<i>ECPG_INVALID_DESCRIPTOR_INDEX</i>	インデックスが範囲外です。
YE002	-242	<i>ECPG_UNKNOWN_DESCRIPTOR_ITEM</i>	セッションが無効な記述子項目を要求しました (内部エラー)。
07006	-243	<i>ECPG_VAR_NOT_NUMERIC</i>	動的 SQL ステートメントが、数値以外のホスト変数に数値を戻しました。
07006	-244	<i>ECPG_VAR_NOT_CHAR</i>	動的 SQL ステートメントが CHAR 値を戻しましたが、ホスト変数が CHAR ではありません。
	-400	<i>ECPG_PGSQL</i>	サーバーがエラーメッセージを返しました。結果のメッセージにはエラーテキストが含まれています。
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開始、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はデータベースに接続できませんでした。
02000	100	<i>ECPG_NOT_FOUND</i>	最後のコマンドで行を取得または処理したか、カーソルの最後に到達しました。

1.5.2 EXEC SQL WHENEVER

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

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

この指示子は、プログラムにエラー処理コードを挿入するように ECPG コンパイラーに指示します。

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

SQLERROR

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

SQLWARNING

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

NOT FOUND

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

上記のいずれかの条件が発生した場合、クライアントアプリケーションが次のいずれかのアクションを実行するように指定できます。

CONTINUE

CONTINUE を指定して、現在の `condition` を無視して処理を続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 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 コードに CONTINUE ステートメントを生成します。条件に遭遇すると、ループ内の残りのコードをスキップし、次の反復に進みます。ループでのみ使用できます。

GOTO label または GO TO label

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

SQLPRINT

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

STOP

`exit(1)` を呼び出して、エラーを通知し、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 パラメーターで指定されたパラメーターを使用して、名前 `parameter` で指定された C ファンクションを呼び出します。

1.5.3 例

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

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

注釈: ECPGPlus コンパイラは、クライアントアプリケーションが上から下に実行されない場合でも、プログラムを上から下に処理します。コンパイラ指示子は各行に順番に適用され、コンパイラが別の指

示子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へでない場合は、コンパイル中に見逃される可能性のあるプログラムの部分にエラー処理指示子を追加することを検討してください。

1.6 Reference

ECPGPlus には次の言語要素があります。

- C プリプロセッサ指示子
- 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

1.6.1 C プリプロセッサ指示子

ECPGPlus C プリプロセッサは、ECPGPlus を呼び出すモードに応じて 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 を使用して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。
- データ型が ECPG と互換性がある限り、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
# undef symbolName
# defined(symbolName)
```

プリプロセッサ指示子は、コンパイラーが受け取るコードに影響を与えるか、指示するために使用されます。たとえば、次のコードサンプルを考えます。

```
# 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を使用して ECPGPlus を呼び出すとします。

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

ECPGPlus は、フラグメント全体を変更せずに出力ファイルにコピーします。次のトークンのみを ECPG パーサーに送信します。

```
long long customerBalance;
```

一方、次のコマンドライン引数を使用して ECPGPlus を呼び出すとします。

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

ECPG パーサーは次のトークンを受け取ります。

```
double customerBalance;
```

コードがプリプロセッサ指示子を使用して、コンパイラーに送信されるコードをフィルター処理する場合、ECPG パーサーは処理されたトークンストリームのみを認識しますが、完全なコードは元のコードに保持されます。

EXEC 指示子で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
```

たとえば、コードに次のものが含まれているとします。

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

(次のページに続く)

(前のページからの続き)

```
EXEC ORACLE ENDIF;  
BALANCE_TYPE customerBalance;
```

次のコマンドライン引数を使用して ECPGPlus を呼び出します。

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

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

```
long long customerBalance;
```

注釈: *EXEC ORACLE* プリプロセッサ指示子は、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 プリコンパイラーオプションの使用

ECPGPlus を互換モードで使用する場合、`SELECT_ERROR` プリコンパイラーオプションを使用して、ホスト変数が対応できる以上の行を含む結果セットを処理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場合、`SELECT` ステートメントはエラーメッセージを返します。`SELECT` ステートメントがホスト変数が収容できるよりも多くの行を返した場合にエラーメッセージを抑制するには、`NO` を指定します。

`EXEC ORACLE OPTION` 指示子で `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そのまま出力ファイルにコピーされます。
- `EXEC SQL BEGIN/END DECLARE` セクションでホスト変数として使用する各 C 変数の型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 コアドキュメントで説明されている動作を実装します。

1.6.2 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取得すると、ECPGPlus は各 SQL データ型をデータが返される C 変数の型にマップします。

一般に、ECPGPlus はほとんどの SQL サーバー型を同様の C 型に変換できますが、すべての組み合わせが有効なわけではありません。たとえば、ECPGPlus は SQL 文字値を C 整数値に変換しようとしませんが、SQL 文字値に数字以外の文字が含まれている場合、実行時に変換が失敗する場合があります。

逆もまた真です。アプリケーションがサーバーに値を送信すると、ECPGPlus は C データ型を必要な SQL 型に変換しようとしませんが、C 値を必要な SQL 型に変換できない場合、実行時に変換が失敗する可能性があります。

ECPGPlus は、任意の SQL 型を C 文字値 (char[n] または varchar[n]) に変換できます。SQL 型を char[n] または varchar[n] との間で変換しても安全ですが、多くの場合、int、double、または float などのより自然な C 型を使用すると便利です。

サポートされている C の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
- typedef によって作成された同等の

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

- timestamp
- interval
- date
- decimal
- numeric

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

1.6.3 タイプコード

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

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、データベースにある値の型を示すために使用されます。`DESCRIBE SELECT LIST` ステートメントは、次の値を使用してデータ型配列 (`descriptor->T[column]`) 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>
189	<i>INTERVAL YEAR TO MONTH</i>
190	<i>INTERVAL DAY TO SECOND</i>
232	<i>TIMESTAMP LOCAL_TZ</i>

1.6.4 SQLDA 構造

Oracle 動的 SQL 方法 4 は、SQLDA データ構造を使用して、動的 SQL 文のデータとメタデータを保持します。SQLDA 構造体は、動的ステートメントのテキストまたは動的ステートメントの結果セットにあるパラメーターマーカーに対応する入力パラメーターのセットを記述することができます。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        */
};
```

パラメーター

N - maximum number of entries

N 構造体メンバーには、SQLDA が記述できるエントリの最大数が含まれています。このメンバーは、SQLDA 構造体を割り当てるときに `sqlald()` ファンクションによって移入されます。OPEN または FETCH ステートメントで記述子を使用する前に、記述された値の実際の数に N を設定する必要があります。

V - data values

V 構造体メンバーは、データ値の配列へのポインターです。

- SELECT -list 記述子の場合、V は FETCH ステートメントによって返される値の配列を指します。配列内の各メンバーは、結果セットの列に対応します。
- バインド記述子の場合、V はパラメーター値の配列を指します。記述子を使用するカーソルを開く前に、この配列に値を設定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領域を割り当てる必要があります。SELECT -list 値に領域を割り当てる方法の例については、

`displayResultSet` を参照してください。

L - length of each data value

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

持するのに十分な大きさのバッファを指している場合、L[5] には値 21（文字列の文字の 20 バイトと NULL ターミネータの 1 バイト）が含まれる必要があります。アプリケーションは、L 配列の各メンバーを設定する必要があります。

T - data types

T 構造体メンバーは、記述子で記述された列（またはパラメーター）ごとに 1 つのデータ型の配列を指します。

- バインド記述子の場合、T 配列の各メンバーを設定して、ECPGPlus に各パラメーターのデータ型を指示する必要があります。
- SELECT -リスト記述子の場合、DESCRIBE SELECT LIST ステートメントは、対応する列で見つかったデータの型を反映するように T 配列のメンバーを設定します。

FETCH ステートメントを実行して、対応する値を特定のデータ型に変換する前に T 配列のメンバーを変更できます。たとえば、DESCRIBE SELECT LIST ステートメントが、特定の列が DATE 型であることを示している場合、対応する T メンバーを変更して、次の FETCH ステートメントがその値を NULL で終わる文字列の形式で返すように要求できます。T 配列の各メンバーは数値型コードです（型コードのリストは [タイプコード](#) を参照）。DESCRIBE SELECT LIST ステートメントが返すタイプコードは、FETCH ステートメントが期待するものとは異なります。DESCRIBE SELECT LIST ステートメントの実行後、T の各メンバーは、データ型と、対応する列が NULL 可能かどうかを示すフラグをエンコードします。sqlnul() 関数を使用して、T 配列のメンバーから型コードと nullable フラグを抽出できます。sqlnul() 関数のシグネチャは次のとおりです。

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

たとえば、results という名前の記述子の 3 列目の型コードと nullable 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 構造体メンバーは、標識変数の配列を指します。この配列は、アプリケーションが sqlalld() 関数を呼び出して記述子を割り当てるときに割り当てられます。

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

F - number of entries

F 構造体メンバーは、記述子で記述されている値の数を示します。N 構造体のメンバーは、記述子が記述できる値の最大数を示します。F は、値の実際の数を示します。F メンバーの値は、DESCRIBE ステートメントを実行するときに ECPGPlus によって設定されます。F は、正、負、またはゼロです。

- SELECT リスト記述子の場合、結果セットの列数が記述子で許可される値の最大数以下である場合（N 構造体メンバーによって決定）、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

ントでない場合、0が含まれます。クエリが **N** 構造体メンバーで許可されているよりも多くの列を返す場合、負の値が含まれます。

- バインド記述子の場合、ステートメントで見つかったパラメーターの数が記述子で許可される値の最大数以下である場合、**F** には正の数が含まれます（**N** 構造体メンバーによって決定されます）。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、0が含まれます。**N** 構造体メンバーで許可されているよりも多くのパラメーターマーカーがステートメントに含まれている場合、負の値が含まれます。

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

S - column/parameter names

S 構造体メンバーは、**NULL** で終了する文字列の配列を指します。

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

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

M - maximum column/parameter name length

M 構造体メンバーは、長さの配列を指します。この配列の各メンバーは、**S** 配列の対応するメンバーの最大長を指定します（つまり、**M[0]** は、**S[0]** で見つかった列/パラメーター名の最大長を指定します）。この配列は、**sqlald()** 関数によって設定されます。

C - actual column/parameter name length

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

この配列は、**DESCRIBE** ステートメントによって設定されます。

X - indicator variable names

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

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

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

このアレイは 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。

1.6.5 ECPGPlus ステートメント

埋め込み SQL ステートメントを使用すると、クライアントアプリケーションはサーバーと対話できます。埋め込み指示子は、ECPGPlus コンパイラーへの指示です。

C プログラムに EDB Postgres Advanced Server SQL ステートメントを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 (;) で終了する必要があります。C プログラムでは、SQL ステートメントは次の形式を取ります。

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

sql_command_body は、標準 SQL ステートメントを表します。ホスト変数は、SQL ステートメントが値式を必要とする場所であればどこでも使用できます。値式のホスト変数の置換の詳細については、

[ホスト変数の宣言](#) を参照してください。

ECPGPlus は、一部のステートメントの PostgreSQL サーバー側の構文を拡張します。構文の違いは、以下のリファレンス情報に記載されています。他の SQL コマンドでサポートされている構文の完全なリファレンスについては、

[PostgreSQL core documentation](#) を参照してください。

割り当てディスクリプター

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

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

そこで：

- array_size は、記述子に割り当てる配列の要素数を指定する変数です。array_size は、INTEGER 値またはホスト変数です。
- descriptor_name は、記述子の名前または記述子の名前を含むホスト変数です。この値は、識別子、引用符で囲まれた文字列リテラル、またはホスト変数の形式を取ることができます。
- variable_count は、記述子内のホスト変数の最大数を指定します。variable_count のデフォルト値は 100 です。

次のコードフラグメントは、配列 (`emp_array`) として処理できる `emp_query` という名前の記述子を割り当てます。

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

コール

CALL ステートメントを使用して、サーバーで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、EDB Postgres 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、関数を呼び出すために使用されます。

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

2 番目の形式は、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>]);
```

そこで：

- `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。プログラム名は、スキーマ修飾、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、プログラムが期待する仮引数に対応します。各仮引数には、IN パラメーター、OUT パラメーター、または INOUT パラメーターを指定できます。
- `:ret_variable` は、プログラムがファンクションの場合に戻り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の場合に返されるインジケータ値を受け取るホスト変数を指定します。

たとえば、次のステートメントは、`:ename` ホスト変数に含まれる値を使用して `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ます。

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

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解放します。クライアントアプリケーションは、閉じたカーソルから行をフェッチできません。CLOSE ステートメントの構文は次のとおりです。

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

`cursor_name` は、ステートメントによってクローズされたカーソルの名前です。カーソル名は、識別子またはホスト変数の形式を取ることができます。

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、カーソルを再度オープンしない限り、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 する必要はありません。

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

```
EXEC SQL CLOSE emp_cursor;
```

アプリケーションが終了すると、カーソルは自動的にクローズされます。

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的にし、他のユーザーが見えるようにします。構文は次のとおりです。

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

`database_name` は、作業が存在するデータベースの名前を含むデータベースまたはホスト変数の名前です。この値は、引用符なしの文字列リテラルまたはホスト変数の形式を取ることができます。

互換性のために、ECPGPlus はエラーなしで COMMENT 句を受け入れますが、COMMENT 句に含まれるテキストは保存しません。

コミットの実行後に現在の接続を閉じるには、RELEASE 句を含めます。

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

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

デフォルトでは、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場合に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
```

接続する

CONNECT ステートメントを使用して、データベースへの接続を確立します。CONNECT ステートメントは 2 つの形式で使用できます。1 つは Oracle データベースと互換性があり、もう 1 つはそうではありません。

最初の形式は Oracle データベースと互換性があります。

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

そこで：

- **user_name** は、クライアントアプリケーションがサーバーへの接続に使用するロールを含むホスト変数です。
- **password** は、そのロールに関連付けられたパスワードを含むホスト変数です。
- **connection_id** は、データベースへの接続に使用されるスラッシュ区切りのユーザー名とパスワードを含むホスト変数です。

AT 句を含めて、接続を確立す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。値は変数または文字列リテラルで指定します。

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

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

CONNECT ステートメントの最初の形式を使用して、クライアントアプリケーションは、次のコマンドを使用して、接続ロールの ID を含む **user** という名前のホスト変数と、関連するパスワードを含む **password** という名前のホスト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形式を使用して、**:connection_id** という名前の単一のホスト変数を使用して接続を確立することもできます。次の例では、**connection_id** には、ユーザーのスラッシュ区切りのロール名と関連するパスワードが含まれています。

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

CONNECT ステートメントの 2 番目の形式の構文は次のとおりです。

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

credentials は次のいずれかです。

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

2 番目の形式では：

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。database_name を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指定します。

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

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

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

そこで：

- hostname は、データベースが存在するサーバーの名前または IP アドレスです。
- port は、サーバーがリッスンするポートです。

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

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

接続の名前を指定しない場合、ecpglib は接続に DEFAULT の名前を割り当てます。任意の EXEC SQL ステートメントで名前（DEFAULT）で接続を参照できます。

- CURRENT は、最後に開かれたか、最新の SET CONNECTION TO ステートメントで言及されている接続です。EXEC SQL ステートメントで接続を名前参照しない場合、ECPG は接続の名前を CURRENT と想定します。
- user_name は、EDB Postgres 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です。

次のコードフラグメントは、CONNECT ステートメントの 2 番目の形式を使用して、ロール alice とそのロールに関連付けられたパスワード 1safepwd を使用して edb という名前のデータベースへの接続を確立します。

```
EXEC SQL CONNECT TO edb AS acctg_conn
USER alice IDENTIFIED BY 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を変更するときに、接続名を使用できます。

割り当て解除ディスクリプター

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

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

`descriptor_name` は記述子の名前です。この値は、引用符で囲まれた文字列リテラルまたはホスト変数の形式を取ることができます。

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

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

カーソルを宣言

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>);
```

そこで：

- `database_name` は、カーソルが操作するデータベースの名前です。この値は、ID またはホスト変数の形式を取ることができます。データベース名を指定しない場合、`database_name` のデフォルト値がデフォルトのデータベースです。
- `cursor_name` はカーソルの名前です。
- `select_statement` は、カーソル結果セットを定義する **SELECT** ステートメントのテキストです。**SELECT** ステートメントに **INTO** 句を含めることはできません。
- `statement_name` は、カーソル結果セットを定義する SQL ステートメントまたはブロックの名前です。

次の例では、`employees` という名前のカーソルを宣言します。

```
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、給与、および歩合を含む結果セットを生成します。

宣言データベース

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（たとえば、CONNECT ステートメント）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;
```

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、接続を確立するときに、または AT 句で acctg データベースを参照できます。

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。

宣言文

DECLARE STATEMENT 指示子を使用して、SQL ステートメントの識別子を宣言します。EDB Postgres Advanced Server は、2つのバージョンの DECLARE STATEMENT 指示子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>;
```

そこで：

- 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、ID または ID を含むホスト変数の形式を取ることができます。

DECLARE STATEMENT 指示子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。

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、式が許可されている任意の場所でパラメーターマーカーとホスト変数を使用できます。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> ]
```

- **FOR exec_count** 句を含めて、ステートメントの実行回数を指定します。この句は、**VALUES** 句が配列または配列へのポインター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、処理を指定されたテーブルに制限します。**ONLY** 句を含めない場合、名前付けたテーブルを継承するテーブルも処理されます。
- **alias** は、対象表の代用名です。
- **using_list** はテーブル式のリストであり、他のテーブルの列を **WHERE** 条件に表示できます。
- **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、テーブル定義はそのままにします。
- **condition** は、**BOOLEAN** 型の値を返す式、ホスト変数、またはパラメーターマーカーです。**condition** が **true** を返す行は削除されます。
- **cursor_name** は、**WHERE CURRENT OF** 句で使用するカーソルの名前です。削除される行は、このカーソルから最後にフェッチされた行です。カーソルは、**DELETE** ステートメントのターゲットテーブルに対する非グループ化クエリである必要があります。ブール条件を含む **DELETE** ステートメントで **WHERE CURRENT OF** を指定することはできません。

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

- **output_expression** は、各行が削除された後に **DELETE** コマンドによって計算されて返される式です。**output_name** は、返される列の名前です。すべての列を返すには*を含めます。
- **host_variable_list** は、ホスト変数とオプションの標識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。

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

```
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、[PostgreSQL core documentation](#) を参照してください。

説明する

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

DESCRIBE ステートメントは、SQLDA 記述子を設定します。SQL 記述子を設定するには、ALLOCATE DESCRIPTOR および DESCRIBE...DESCRIPTOR ステートメントを使用します。

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

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

そこで：

- `statement_name` は、準備された SQL 文または PL/SQL ブロックに関連付けられた識別子です。
- `descriptor` は、SQLDA* 型の C 変数の名前です。sqlald() を呼び出して記述子用の領域を割り当て、DESCRIBE ステートメントを実行する前に記述子を初期化する必要があります。

DESCRIBE ステートメントの最初の形式を実行すると、ECPG は指定された記述子にステートメントで `_required_` 各入力変数の説明を入力します。たとえば、2 つの記述子が与えられた場合：

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

emp テーブルから情報を返すクエリを準備できます。

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

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

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

この文のバインド変数を説明した後、記述子を調べて、必要な変数の数と各変数の型を確認できます。

2 番目のフォームを実行すると、ECPG は指定された記述子にステートメントによって返された各値の説明を入力します。たとえば、次のステートメントは 3 つの値を返します。

```
EXEC SQL DESCRIBE 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説明した後、記述子を調べて、返された値の数、および各値の名前と型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。出力値にバインドした変数は、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 Pro*C 互換の代替構文です。

詳細説明

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

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

そこで：

- `statement_name` は、準備された SQL ステートメントの名前です。
- `descriptor_name` は、記述子の名前です。`descriptor_name` は、引用符で囲まれた文字列値または記述子の名前を含むホスト変数です。

`INPUT` 句を含めると、ECPGPlus は指定された記述子にステートメントに必要な各入力変数の説明を入力します。

たとえば、2 つの記述子が与えられた場合：

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

`emp` テーブルから情報を返すクエリを準備できます。

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

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

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

この文のバインド変数を説明した後、記述子を調べて、必要な変数の数と各変数の型を確認できます。

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

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

たとえば、次のステートメントは 3 つの値を返します。

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

このステートメントの選択リストを説明した後、記述子を調べて、返された値の数、および各値の名前と型を見つけることができます。

切断する

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];
```

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

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

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

すべてのアクティブな接続を閉じるには、ALL キーワードを含めます。

次の例では、hr データベースに接続してから切断する hr_connection という名前の接続を作成します。

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

実行する

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以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>]]];
```

そこで：

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

- statement_name は、EXEC SQL PREPARE ステートメントを使用してステートメントを作成したときにステートメントに割り当てられた名前を指定します。

USING 句を含めて、準備されたステートメントのパラメーターに値を指定します。

- DESCRIPTOR SQLDA_descriptor 句を含めて、パラメーターに SQLDA 記述子値を提供します。
- host_variable (およびオプションの indicator_variable) を使用して、パラメーターにユーザー指定の値を提供します。

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

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

準備されたステートメントを呼び出すたびに、ステートメントに新しいパラメーター値を指定します。

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

EXECUTE DESCRIPTOR

EXECUTE ステートメントを使用して、SQL 記述子を使用して、EXEC SQL PREPARE ステートメントで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>];
```

そこで：

- `array_size` は、処理する行数を指定する整数値またはホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。
- `statement_identifier` は、EXEC SQL PREPARE ステートメント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。
- `descriptor_name` は、記述子の名前（一重引用符で囲まれた文字列リテラルとして）、または記述子の名前を含むホスト変数を指定します。

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

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

次の例では、記述子 `stmtText:` に含まれる値を使用して、準備されたステートメント `give_raise` を実行します。

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

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;
```

そこで：

- `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格納するホスト変数です。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 のステートメントと宣言のインラインシーケンスです。ブロックにはホスト変数とオプションの標識変数を含めることができます。このような変数はそれぞれ IN/OUT 値として扱われます。

次の例では、無名ブロックを実行します。

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

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

すぐに実行

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>;
```

そこで：

- `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格納するホスト変数です。AT 句を省略すると、ステートメントは現在のデフォルトデータベースで実行されます。

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

この動的 SQL ステートメントは、クライアントアプリケーションを作成するときに SQL ステートメントのテキストがわからない場合に役立ちます。たとえば、クライアントアプリケーションは、実行するステートメントを信頼できるユーザーに求める場合があります。ユーザーがステートメントのテキストを文字列値として入力すると、ステートメントが EXECUTE IMMEDIATE コマンドで実行されます。

ステートメントテキストには、ホスト変数への参照を含めることはできません。ステートメントにパラメーターマーカーが含まれている場合、または 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します。

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;
```

取得

FETCH ステートメントを使用して、カーソルから SQLDA 記述子またはホスト変数のターゲットリストに行を戻します。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> };
```

そこで：

- array_size は、取得する行数を指定する整数値またはホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

- cursor は、行がフェッチされるカーソルの名前、またはカーソルの名前を含むホスト変数です。

USING 句を含めると、FETCH ステートメントは、指定された SQLDA 記述子にサーバーから返された値を設定します。

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

次のコードフラグメントは、emp テーブルから employee number、name、および salary を取得する employees という名前のカーソルを宣言しています。

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

フェッチディスクリプタ

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

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

そこで：

- **array_size** は、取得する行数を指定する整数値またはホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。
- **cursor** は、行がフェッチされるカーソルの名前、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。
- **descriptor_name** は、記述子の名前（一重引用符で囲まれた文字列リテラルとして）または記述子の名前を含むホスト変数を指定します。使用する前に、ALLOCATE DESCRIPTOR ステートメントを使用して記述子を割り当てる必要があります。

INTO 句を含めて、準備されたステートメントから返された結果を EXECUTE ステートメントが書き込む SQL 記述子を指定します。

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

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

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

記述子を取得する

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

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

2 番目の形式は、特定の値（VALUE column_number 句で指定）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> {,...}};
```

そこで：

- **array_size** は、処理する行数を指定する整数値またはホスト変数です。**array_size** を指定する場合、**host_variable** はそのサイズの配列である必要があります。たとえば、**array_size** が 10 の場合、

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

- descriptor_name は、ディスクリプタ名を一重引用符で囲んだ文字列リテラルまたはディスクリプタ名を格納するホスト変数で指定します。

VALUE 句を含めて、記述子から取得した情報を指定します。

- column_number は、記述子内の変数の位置を識別します。
- host_variable は、項目の値を受け取るホスト変数名を指定します。
- descriptor_item は、取得する記述子項目の種類を指定します。

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

次のコードフラグメントは、GET DESCRIPTOR ステートメントを使用して、ユーザー指定の文字列に入力された列の数を取得する方法を示しています。

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

この例では、PREPARE ステートメントを使用してユーザー:stmt が提供した文字列の構文を確認する前に、parse_desc という名前の SQL 記述子を割り当てます。DESCRIBE ステートメントは、ユーザー指定の文字列を記述子 parse_desc に移動します。EXEC SQL GET DESCRIPTOR の呼び出しは、記述子を調べて、結果セット内の (:col_count) 列の数を検出します。

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、値が許可される任意の場所でパラメーターマーカーとホスト変数を使用できます。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>] [, ...]]
```

FOR `exec_count` 句を含めて、ステートメントの実行回数を指定します。この句は、VALUES 句が配列または配列へのポインター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）名前を指定します。
- `column` は、テーブルの列の名前です。列名は、サブフィールド名または配列の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た式、値、ホスト変数、またはパラメーターマーカーです。DEFAULT を指定して、対応する列をデフォルト値で埋めます。
- `query` は、挿入する行を提供する SELECT ステートメントを指定します。
- `output_expression` は、各行が挿入された後に INSERT コマンドによって計算されて返される式です。式は、テーブル内の任意の列を参照できます。挿入された行のすべての列を返すには、* 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);
```

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

INSERT ステートメントの使用の詳細については、[PostgreSQL core documentation](#) を参照してください。

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>, ... ]
```

そこで：

- `array_size` は、取得する行数を指定する整数値またはホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。
- `cursor` は、オープンされるカーソルの名前です。
- `parameters` は、DESCRIPTOR `SQLDA_descriptor` またはカーソルを初期化する `host variables` とオプションの `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、記述子は DESCRIBE ステートメントで初期化する必要があります。

OPEN ステートメントは、`parameters` で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クローズされます。

次の例では、`emp` テーブルを照会する `employees` という名前のカーソルを宣言します。ホスト変数 `emp_name` に格納されているユーザー指定の値と名前が一致する従業員の `employee number`、`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て、カーソルの内容をクライアントアプリケーションで使えるようにします。

オープンディスクリプター

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

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

そこで：

- `array_size` は、取得する行数を指定する整数値またはホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに 1 回実行されます。

- `cursor` は、オープンされるカーソルの名前です。
- `descriptor_name` は、SQL 記述子名をシングルクォーテーション文字列リテラル、またはカーソルを初期化するクエリーを含む SQL 記述子名を含むホスト変数で指定します。

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

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

準備する

準備されたステートメントは、クライアントアプリケーションがタスクを複数回実行する必要がある場合に役立ちます。ステートメントは、実行されるたびにではなく、1 回だけ解析、書き込まれ、計画されます。このアプローチにより、反復処理時間が節約されます。

PREPARE ステートメントを使用して、実行する SQL ステートメントまたは PL/pgSQL ブロックを準備します。ステートメントは 2 つの形式で入手できます。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 番目の形式は次のとおりです。

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

そこで：

- `database_name` 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。
- `statement_name` は、準備された SQL 文または PL/SQL ブロックに関連付けられた識別子です。
- `sql_statement` は、SELECT ステートメント、一重引用符で囲まれた文字列リテラル、または SQL ステートメントのテキストを含むホスト変数の形式を取ることができます。

準備されたステートメントに変数を含めるには、ステートメントを PREPARE した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、各パラメーターの値を指定します。プレースホルダーを置き換える順序で値を指定します。

次の例では、`emp` テーブルにレコードを挿入する `add_emp` という名前の準備済みステートメントを作成します。

```
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、ステートメントに新しいパラメーター値を指定します。

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

注釈: クライアントアプリケーションは、ステートメントが実行される各セッションで `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のみ持続します。

ロールバック

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

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

`database_name` は、ステートメントの実行対象とな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。

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

RELEASE 句を含めて、アプリケーションに EXEC SQL COMMIT RELEASE を実行させ、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中止され、すべての変更が元に戻され、セーブポイントが消去され、すべてのトランザクションロックが解放されます。セーブポイント（次の例では `my_savepoint`）を含めるとします。

```
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、その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。

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

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

そこで：

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

セーブポイントはトランザクションブロックでのみ設定できます。トランザクションブロックには、複数のセーブポイントを含めることができます。

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

```
EXEC SQL SAVEPOINT my_savepoint;
```

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。この句を使用すると、EDB Postgres Advanced Server データベースから指定した情報を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 ] [...]]
```

そこで：

- `database_name` は、テーブルが存在するデータベースの名前を含むデータベースまたはホスト変数の名前です。この値は、引用符なしの文字列リテラルまたはホスト変数の形式を取ることができます。
- `host_variables` は、SELECT ステートメントによって入力された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、[PostgreSQL core documentation](#) に記載されている SQL SELECT ステートメントの追加の句のサポートを提供します。

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されたホスト変数の名前を含めます。たとえば、次の SELECT ステートメントは、`:emp_name` および `:emp_sal` ホスト変数に従業員の名前と給与のリストを設定します。

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

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

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

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

接続を設定する

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理由は少なくとも3つあります。

- ステートメントごとに異なる特権が必要になる場合があります。
- 同じクライアントで複数のデータベースと対話する必要がある場合。
- クライアントアプリケーションでの複数の実行スレッドは、接続を同時に共有できません。

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>;
```

`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、2 番目の形式の CONNECT ステートメントを使用してデータベースへの接続を開きます。`connection_name` を指定するには、AS 句を含めます。

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

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

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

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

たとえば、クライアントアプリケーションが次のいずれかのアプローチを使用して複数の接続を作成および維持するとします。

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

```
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、接続しているクライアントが使用できる特権を判断するときに、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、クライアントにはロール alice に関連付けられた特権があります。hr_conn を使用して接続すると、クライアントには bob に関連付けられた特権があります。

セットディスクリプター

SET DESCRIPTOR 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がホスト変数または整数値の形式で提供する情報を使用して、記述子領域に値を割り当てます。ステートメントには2つの形式があります。最初の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 番目の形式は次のとおりです。

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

そこで：

- array_size は、取得する行数を指定する整数値またはホスト変数です。FOR 句を省略すると、ステートメントは配列のメンバーごとに1回実行されます。

- `descriptor_name` は、ディスクリプタ名を一重引用符で囲んだ文字列リテラルまたはディスクリプタ名を格納するホスト変数で指定します。

VALUE 句を含めて、記述子に格納されている情報を説明します。

- `column_number` は、記述子内の変数の位置を識別します。
- `descriptor_item` は、記述子項目の種類を指定します。
- `host_variable` は、項目の値を格納するホスト変数名を指定します。

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

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

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

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

動的に作成されたクエリを実行するには、最初にクエリを準備（クエリの構文を解析および検証）してから、EXEC SQL DESCRIBE INPUT ステートメントを使用してクエリで見つかった入力パラメーターを説明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;
```

`query_params` 記述子には、クエリを記述した後、クエリに必要な各パラメーターに関する情報が含まれています。

この例では、ユーザーが次のように入力したとします。

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

この場合、記述子には2つのパラメーターが記述されています。1つは `sal > ?` 用、もう1つは `job = ?` 用です。

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

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

次に、EXEC SQL GET DESCRIPTOR を使用して各パラメーターの名前を取得できます。EXEC SQL GET DESCRIPTOR を使用して、パラメーターの数とともに記述子から各パラメーターの型を取得することもできます。または、各 value を文字列の形式で指定すると、ECPG はその文字列を必要なデータ型に変換します。

最初のパラメーターのデータ型は numeric です。第 2 引数の型は varchar です。最初のパラメーターの名前は sal です。2 番目のパラメーターの名前は job です。

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

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

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

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

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

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

value を各パラメーターに関連付けるには、EXEC SQL SET DESCRIPTOR ステートメントを使用します。例：

```
EXEC SQL SET DESCRIPTOR query_params
    VALUE :param_number DATA = :reply;
EXEC SQL SET DESCRIPTOR query_params
    VALUE :param_number INDICATOR = :reply_ind;
}
```

これで、EXEC SQL EXECUTE DESCRIPTOR ステートメントを使用して、サーバーで準備されたステートメントを実行できます。

更新

UPDATE ステートメントを使用して、テーブルに格納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] [, ...] ]
```

database_name は、テーブルが存在するデータベースの名前を含むデータベースまたはホスト変数の名前です。この値は、引用符なしの文字列リテラルまたはホスト変数の形式を取ることができます。

FOR exec_count 句を含めて、ステートメントの実行回数を指定します。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、[PostgreSQL core documentation](#) に記載されている SQL UPDATE ステートメントの追加の句のサポートを提供します。

値を指定する句でホスト変数を使用できます。ホスト変数を使用するには、文書化された UPDATE 句に関連付けられた値を定義された変数に置き換えます。

次の UPDATE ステートメントは、従業員（:ename ホスト変数で識別）の職務記述書を:new_job ホスト変数に含まれる値に変更します。現在の給与に:increase ホスト変数の値を掛けて、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、入力値が許可されている句にパラメーターマーカー（疑問符）を含めることもできます。たとえば、WHERE 句にパラメーターマーカーを使用して、同じ更新ステートメントを書くことができます。

```
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、ユーザーに job 列の新しい値の入力を求め、:ename で指定された従業員の sal 列のインクリメント量を提供できます。

いつでも

WHENEVER ステートメントを使用して、SQL エラーまたは警告が発生したときにクライアントアプリケーションが実行するアクションを指定します。構文は次のとおりです。

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

次の表で、action をトリガーする可能性のあるさまざまな条件について説明します。

Condition	Description
<i>NOT FOUND</i>	サーバーは、行を返さない `SELECT` を検出した場合、または `FETCH` が結果セットの最後に到達した場合、`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る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警告を検出すると、`SQLWARNING` 条件を返します。

次の表では、クライアントが condition に遭遇した結果として発生するアクションについて説明します。

Action	Description
<i>CALL function</i> <i>[[args]]</i>	Call the named <i>function</i> .
<i>CONTINUE</i>	Proceed to the next statement.
<i>DO BREAK</i>	Emit a C break statement. A break statement can appear in a <i>loop</i> or a <i>switch</i> statement. If executed, the break statement terminates the <i>loop</i> or the <i>switch</i> statement.
<i>DO CONTINUE</i>	Emit a C <i>continue</i> statement. A <i>continue</i> statement can exist only in a loop. If executed, it causes the flow of control to return to the top of the loop.
<i>DO function</i> <i>[[args]]</i>	Call the named <i>function</i> .
<i>GOTO label</i> または <i>GO TO label</i>	Proceed to the statement that contains the label.
<i>SQLPRINT</i>	Print a message to standard error.
<i>STOP</i>	Stop executing.

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

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

次のコードを含めて、クライアントが問題を警告した後に処理を続行するように指定します。

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
```

クエリが行を返さない場合、またはカーソルが結果セットの最後に到達した場合に関数を呼び出すには、次のコードを含めます。

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