# ECPGPlus Guide

## Version 13

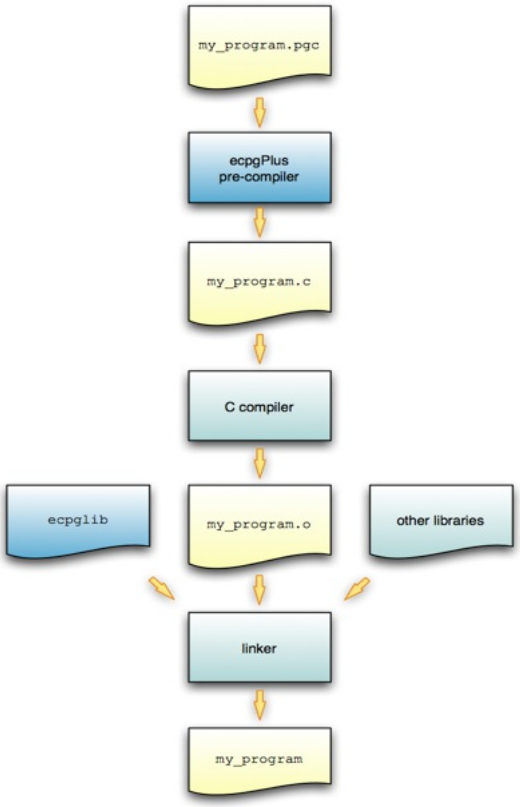


Built at 2021-03-01T17:59:05

# 1 ECPGPlus - Overview

EDB has enhanced ECPG (the PostgreSQL pre-compiler) to create ECPGPlus. ECPGPlus is a Pro*C-compatible version of the PostgreSQL C pre-compiler. ECPGPlus translates a program that combines C code and embedded SQL statements into an equivalent C program. As it performs the translation, ECPGPlus verifies that the syntax of each SQL construct is correct.

The following diagram charts the path of a program containing embedded SQL statements as it is compiled into an executable:




Compilation of a program containing embedded SQL statements

To produce an executable from a C program that contains embedded SQL statements, pass the program (`my_program.pgc` in the diagram above) to the ECPGPlus pre-compiler. ECPGPlus translates each SQL statement in `my_program.pgc` into C code that calls the `ecpglib` API, and produces a C program (`my_program.c`). Then, pass the C program to a C compiler; the C compiler generates an object file (`my_program.o`). Finally, pass the object file (`my_program.o`), as well as the `ecpglib` library file, and any other required libraries to the linker, which in turn produces the executable (`my_program`).

While the ECPGPlus preprocessor validates the *syntax* of each SQL statement, it cannot validate the *semantics*. For example, the preprocessor will confirm that an `INSERT` statement is syntactically correct, but it cannot confirm that the table mentioned in the `INSERT`

statement actually exists.

**Behind the Scenes**

A client application contains a mix of C code and SQL code comprised of the following elements:

- C preprocessor directives
- C declarations (variables, types, functions, ...)
- C definitions (variables, types, functions, ...)
- SQL preprocessor directives
- SQL statements

For example:

```
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 }
```

In the above code fragment:

- Line 1 specifies a directive to the C preprocessor.

C preprocessor directives may be interpreted or ignored; the option is controlled by a command line option (`-C PROC`) entered when you invoke ECPGPlus. In either case, ECPGPlus copies each C preprocessor directive to the output file (4) without change; any C preprocessor directive found in the source file will appear in the output file.

- Line 2 specifies a directive to the SQL preprocessor.

  SQL preprocessor directives are interpreted by the ECPGPlus preprocessor, and are not copied to the output file.

- Lines 4 through 6 contain C declarations.

  C declarations are copied to the output file without change, except that each `VARCHAR` declaration is translated into an equivalent `struct` declaration.

- Lines 10 through 14 contain an embedded-SQL declaration section.

  C variables that you refer to within SQL code are known as `host variables`. If you invoke the ECPGPlus preprocessor in Pro*C mode (`-C PROC`), you may refer to *any* C variable within a SQL statement; otherwise you must declare each host variable within a `BEGIN/END DECLARATION SECTION` pair.

- Lines 16 through 19 contain a SQL statement.

  SQL statements are translated into calls to the ECPGPlus run-time library.

- Lines 21 through 23 contain C code.

  C code is copied to the output file without change.

Any SQL statement must be prefixed with `EXEC SQL` and extends to the next (unquoted) semicolon. For example:

```
printf(“Updating employee salaries\n”);

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

printf(“Employee salaries updated\n”);
```

When the preprocessor encounters the code fragment shown above, it passes the C code (the first line and the last line) to the output file without translation and converts each `EXEC SQL` statement into a call to an `ecpglib` function. The result would appear similar to the following:

```
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”);
```

## Installation and Configuration

On Windows, ECPGPlus is installed by the Advanced Server installation wizard as part of the `Database Server` component. On Linux, install with the `edb-asxx-server-devel` RPM package where `xx` is the Advanced Server version number. By default, the executable is located in:

On Windows:

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

On Linux:

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

When invoking the ECPGPlus compiler, the executable must be in your search path (`%PATH%` on Windows, `$PATH` on Linux). For example, the following commands set the search path to include the directory that holds the ECPGPlus executable file `ecpg`.

On Windows:

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

On Linux:

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

## Constructing a Makefile

A `makefile` contains a set of instructions that tell the `make` utility how to transform a program written in C (that contains embedded SQL) into a C program. To try the examples in this guide, you will need:

- a C compiler (and linker)
- the `make` utility
- ECPGPlus preprocessor and library
- a `makefile` that contains instructions for ECPGPlus

The following code is an example of a `makefile` for the samples included in this guide. To use the sample code, save it in a file named `makefile` in the directory that contains the source code file.

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

.SUFFIXES: .pgc,.pc

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

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

The first two lines use the `pg_config` program to locate the necessary header files and library directories:

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

The `pg_config` program is shipped with Advanced Server.

`make` knows that it should use the `CFLAGS` variable when running the C compiler and `LDFLAGS` and `LDLIBS` when invoking the linker. ECPG programs must be linked against the ECPG run-time library (`-lecpg`) and the libpq library (`-lpq`)

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

The sample `makefile` instructs make how to translate a `.pgc` or a `.pc` file into a C program. Two lines in the `makefile` specify the mode in which the source file will be compiled. The first compile option is:

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

The first option tells `make` how to transform a file that ends in `.pgc` (presumably, an ECPG source file) into a file that ends in `.c` (a C program), using community ECPG (without the ECPGPlus enhancements). It invokes the ECPG pre-compiler with the `-c` flag (instructing the compiler to convert SQL code into C), using the value of the `INCLUDES` variable and the name of the `.pgc` file.

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

The second option tells make how to transform a file that ends in `.pg` (an ECPG source file) into a file that ends in `.c` (a C program), using the ECPGPlus extensions. It invokes the ECPG pre-compiler with the `-c` flag (instructing the compiler to convert SQL code into C), as well as the `-C PROC` flag (instructing the compiler to use ECPGPlus in Pro*C-compatibility mode), using the value of the `INCLUDES` variable and the name of the `.pgc` file.

When you run `make`, pass the name of the ECPG source code file you wish to compile. For example, to compile an ECPG source code file named `customer_list.pgc`, use the command:

```
make customer_list
```

The `make` utility consults the `makefile` (located in the current directory), discovers that the `makefile` contains a rule that will compile `customer_list.pgc` into a C program (`customer_list.c`), and then uses the rules built into `make` to compile `customer_list.c` into an executable program.

## ECPGPlus Command Line Options

In the sample `makefile` shown above, `make` includes the `-C` option when invoking ECPGPlus to specify that ECPGPlus should be invoked in Pro*C compatible mode.

If you include the `-C PROC` keywords on the command line, in addition to the ECPG syntax, you may use Pro*C command line syntax; for example:

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

To display a complete list of the other ECPGPlus options available, navigate to the ECPGPlus installation directory, and enter:

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

The command line options are:

| Option | Description |
|---|---|
| -c | Automatically generate C code from embedded SQL code. |
| -C *mode* | Use the `-C` option to specify a compatibility mode:<br><br>`INFORMIX`<br><br>`INFORMIX_SE`<br><br>`PROC` |
| -D *symbol* | Define a preprocessor *symbol*.<br><br>The *-D* keyword is not supported when compiling in *PROC mode.* Instead, use the Oracle-style *'DEFINE='* clause. |
| -h | Parse a header file, this option includes option `-c`. |
| -i | Parse system, include files as well. |
| -I directory | Search *directory* for `include` files. |
| -o *outfile* | Write the result to *outfile*. |
| -r *option* | Specify run-time behavior; *option* can be:<br><br>`no_indicator` - Do not use indicators, but instead use special values to represent NULL values.<br><br>`prepare` - Prepare all statements before using them.<br><br>`questionmarks` - Allow use of a question mark as a placeholder.<br><br>`usebulk` - Enable bulk processing for INSERT, UPDATE and DELETE statements that operate on host variable arrays. |
| --regression | Run in regression testing mode. |
| -t | Turn on `autocommit` of transactions. |
| -l | Disable `#line` directives. |
| --help | Display the help options. |

| Option | Description |
| --- | --- |
| --version | Output version information. |

!!! Note If you do not specify an output file name when invoking ECPGPlus, the output file name is created by stripping off the `.pgc` filename extension, and appending `.c` to the file name.

# 2 Using Embedded SQL

Each of the following sections leads with a code sample, followed by an explanation of each section within the code sample.

## Example - A Simple Query

The first code sample demonstrates how to execute a `SELECT` statement (which returns a single row), storing the results in a group of host variables. After declaring host variables, it connects to the `edb` sample database using a hard-coded role name and the associated password, and queries the `emp` table. The query returns the values into the declared host variables; after checking the value of the `NULL` indicator variable, it prints a simple result set onscreen and closes the connection.

```
/************************************************************
 * 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';

  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;
}
/*************************************************************\*
```

The code sample begins by including the prototypes and type definitions for the C `stdio` library, and then declares the `main` function:

```
#include <stdio.h>

int main(void)
{
```

Next, the application declares a set of host variables used to interact with the database server:

```
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;
```

Please note that if you plan to pre-compile the code in `PROC` mode, you may omit the `BEGIN DECLARE…END DECLARE` section. For more information about declaring host variables, refer to the Declaring Host Variables.

The data type associated with each variable within the declaration section is a C data type. Data passed between the server and the client application must share a compatible data type; for more information about data types, see the Supported C Data Types.

The next statement instructs the server how to handle an error:

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

If the client application encounters an error in the SQL code, the server will print an error message to `stderr` (standard error), using the `sqlprint()` function supplied with `ecpglib`. The next `EXEC SQL` statement establishes a connection with Advanced Server:

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

In our example, the client application connects to the `edb` database, using a role named `alice` with a password of `1safepwd`.

The code then performs a query against the `emp` table:

```
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;
```

The query returns information about employee number `7369`.

The `SELECT` statement uses an `INTO` clause to assign the retrieved values (from the `empno`, `ename`, `sal` and `comm` columns) into the `:v_empno`, `:v_ename`, `:v_sal` and `:v_comm` host variables (and the `:v_comm_ind` null indicator). The first value retrieved is assigned to the first variable listed in the `INTO` clause, the second value is assigned to the second variable, and so on.

The `comm` column contains the commission values earned by an employee, and could potentially contain a `NULL` value. The statement includes the `INDICATOR` keyword, and a host variable to hold a null indicator.

The code checks the null indicator, and displays the appropriate on-screen results:

```
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);
```

If the null indicator is `0` (that is, `false`), the `comm` column contains a meaningful value, and the `printf` function displays the commission. If the null indicator contains a non-zero value, `comm` is `NULL`, and `printf` displays a value of `NULL`. Please note that a host variable (other than a null indicator) contains no meaningful value if you fetch a `NULL` into that host variable; you must use null indicators to identify any value which may be `NULL`.

The final statement in the code sample closes the connection to the server:

```
EXEC SQL DISCONNECT;
}
```

## Using Indicator Variables

The previous example included an *indicator variable* that identifies any row in which the value of the `comm` column (when returned by the server) was `NULL`. An indicator variable is an extra host variable that denotes if the content of the preceding variable is `NULL` or truncated. The indicator variable is populated when the contents of a row are stored. An indicator variable may contain the following values:

| Indicator Value | Denotes |
|---|---|
| If an indicator variable is less than `0`. | The value returned by the server was `NULL`. |
| If an indicator variable is equal to `0`. | The value returned by the server was not `NULL`, 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. |

When including an indicator variable in an `INTO` clause, you are not required to include the optional `INDICATOR` keyword.

You may omit an indicator variable if you are certain that a query will never return a `NULL` value into the corresponding host variable. If you omit an indicator variable and a query

returns a `NULL` value, `ecpglib` will raise a run-time error.

## Declaring Host Variables

You can use a *host variable* in a SQL statement at any point that a value may appear within that statement. A host variable is a C variable that you can use to pass data values from the client application to the server, and return data from the server to the client application. A host variable can be:

- an array
- a `typedef`
- a pointer
- a `struct`
- any scalar C data type

The code fragments that follow demonstrate using host variables in code compiled in `PROC` mode, and in non-`PROC` mode. The SQL statement adds a row to the `dept` table, inserting the values returned by the variables `v_deptno`, `v_dname` and `v_loc` into the `deptno` column, the `dname` column and the `loc` column, respectively.

If you are compiling in `PROC` mode, you may omit the `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION` directives. `PROC` mode permits you to use C function parameters as host variables:

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

If you are not compiling in `PROC` mode, you must wrap embedded variable declarations with the `EXEC SQL BEGIN DECLARE SECTION` and the `EXEC SQL END DECLARE SECTION` directives, as shown below:

```
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);
}
```

You can also include the `INTO` clause in a `SELECT` statement to use the host variables to retrieve information:

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

Each column returned by the `SELECT` statement must have a type-compatible target variable in the `INTO` clause. This is a simple example that retrieves a single row; to retrieve more than one row, you must define a cursor, as demonstrated in the next example.

## Example - Using a Cursor to Process a Result Set

The code sample that follows demonstrates using a cursor to process a result set. There are four basic steps involved in creating and using a cursor:

1. Use the `DECLARE CURSOR` statement to define a cursor.
2. Use the `OPEN CURSOR` statement to open the cursor.
3. Use the `FETCH` statement to retrieve data from a cursor.
4. Use the `CLOSE CURSOR` statement to close the cursor.

After declaring host variables, our example connects to the `edb` database using a user-supplied role name and password, and queries the `emp` table. The query returns the values into a cursor named `employees`. The code sample then opens the cursor, and loops through the result set a row at a time, printing the result set. When the sample detects the end of the result set, it closes the connection.

```
/************************************************************
 * 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);
  }
  EXEC SQL CLOSE employees;
  EXEC SQL DISCONNECT;
}
/*************************************************************
```

The code sample begins by including the prototypes and type definitions for the C `stdio` library, and then declares the `main` function:

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

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

Next, the application declares a set of host variables used to interact with the database server:

```
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[]` is an array that contains the command line arguments entered when the user runs the client application. `argv[1]` contains the first command line argument (in this case, a `username`), and `argv[2]` contains the second command line argument (a `password`); please note that we have omitted the error-checking code you would normally include a real-world application. The declaration initializes the values of `username` and `password`, setting them to the values entered when the user invoked the client application.

You may be thinking that you could refer to `argv[1]` and `argv[2]` in a SQL statement (instead of creating a separate copy of each variable); that will not work. All host variables must be declared within a `BEGIN/END DECLARE SECTION` (unless you are compiling in `PROC` mode). Since `argv` is a function *parameter* (not an automatic variable), it cannot be declared within a `BEGIN/END DECLARE SECTION`. If you are compiling in `PROC` mode, you can refer to *any* C variable within a SQL statement.

The next statement instructs the server to respond to an SQL error by printing the text of the error message returned by ECPGPlus or the database server:

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

Then, the client application establishes a connection with Advanced Server:

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

The `CONNECT` statement creates a connection to the `edb` database, using the values found in the `:username` and `:password` host variables to authenticate the application to the server when connecting.

The next statement declares a cursor named `employees`:

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

`employees` will contain the result set of a `SELECT` statement on the `emp` table. The query returns employee information from the following columns: `empno`, `ename`, `sal` and `comm`. Notice that when you declare a cursor, you do not include an `INTO` clause - instead, you specify the target variables (or descriptors) when you `FETCH` from the cursor.

Before fetching rows from the cursor, the client application must `OPEN` the cursor:

```
EXEC SQL OPEN employees;
```

In the subsequent `FETCH` section, the client application will loop through the contents of the cursor; the client application includes a `WHENEVER` statement that instructs the server to `break` (that is, terminate the loop) when it reaches the end of the cursor:

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

The client application then uses a `FETCH` statement to retrieve each row from the cursor `INTO` the previously declared host variables:

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

The `FETCH` statement uses an `INTO` clause to assign the retrieved values into the `:v_empno`, `:v_ename`, `:v_sal` and `:v_comm` host variables (and the `:v_comm_ind` null indicator). The first value in the cursor is assigned to the first variable listed in the `INTO` clause, the second value is assigned to the second variable, and so on.

The `FETCH` statement also includes the `INDICATOR` keyword and a host variable to hold a null indicator. If the `comm` column for the retrieved record contains a `NULL` value, `v_comm_ind` is set to a non-zero value, indicating that the column is `NULL`.

The code then checks the null indicator, and displays the appropriate on-screen results:

```
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);
}
```

If the null indicator is `0` (that is, `false`), `v_comm` contains a meaningful value, and the `printf` function displays the commission. If the null indicator contains a non-zero value, `comm` is `NULL`, and `printf` displays the string `'NULL'`. Please note that a host variable (other than a null indicator) contains no meaningful value if you fetch a `NULL` into that host variable; you must use null indicators for any value which may be `NULL`.

The final statements in the code sample close the cursor `(employees)`, and the connection to the server:

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

# 3 Using Descriptors

Dynamic SQL allows a client application to execute SQL statements that are composed at runtime. This is useful when you don't know the content or form a statement will take when you are writing a client application. ECPGPlus does *not* allow you to use a host variable in place of an identifier (such as a table name, column name or index name); instead, you should use dynamic SQL statements to build a string that includes the information, and then execute that string. The string is passed between the client and the server in the form of a *descriptor*. A descriptor is a data structure that contains both the data and the information about the shape of the data.

A client application must use a `GET DESCRIPTOR` statement to retrieve information from a descriptor. The following steps describe the basic flow of a client application using dynamic SQL:

1. Use an `ALLOCATE DESCRIPTOR` statement to allocate a descriptor for the result set (select list).
2. Use an `ALLOCATE DESCRIPTOR` statement to allocate a descriptor for the input parameters (bind variables).

3. Obtain, assemble or compute the text of an SQL statement.
4. Use a `PREPARE` statement to parse and syntax-check the SQL statement.
5. Use a `DESCRIBE` statement to describe the select list into the select-list descriptor.
6. Use a `DESCRIBE` statement to describe the input parameters into the bind-variables descriptor.
7. Prompt the user (if required) for a value for each input parameter. Use a `SET DESCRIPTOR` statement to assign the values into a descriptor.
8. Use a `DECLARE CURSOR` statement to define a cursor for the statement.
9. Use an `OPEN CURSOR` statement to open a cursor for the statement.
10. Use a `FETCH` statement to fetch each row from the cursor, storing each row in select-list descriptor.
11. Use a `GET DESCRIPTOR` command to interrogate the select-list descriptor to find the value of each column in the current row.
12. Use a `CLOSE CURSOR` statement to close the cursor and free any cursor resources.

A descriptor may contain the attributes listed in the table below:

| Field | Type | Attribute Description |
|---|---|---|
| `CARDINALITY` | `integer` | The number of rows in the result set. |
| `DATA` | N/A | The data value. |
| `DATETIME_INTERVAL_CODE` | `integer` | If `TYPE` is `9`:<br>`1 - DATE`<br>`2 - TIME`<br>`3 - TIMESTAMP`<br>`4 - TIME WITH TIMEZONE`<br>`5 - TIMESTAMP WITH TIMEZONE` |
| `DATETIME_INTERVAL_PRECISION` | `integer` | Unused. |
| `INDICATOR` | `integer` | Indicates a `NULL` or truncated value. |
| `KEY_MEMBER` | `integer` | Unused (returns `FALSE`). |
| `LENGTH` | `integer` | The data length (as stored on server). |
| `NAME` | `string` | The name of the column in which the data resides. |
| `NULLABLE` | `integer` | Unused (returns `TRUE`). |

| Field | Type | Attribute Description |
|---|---|---|
| OCTET_LENGTH | integer | The data length (in bytes) as stored on server. |
| PRECISION | integer | The data precision (if the data is of numeric type). |
| RETURNED_LENGTH | integer | Actual length of data item. |
| RETURNED_OCTET_LENGTH | integer | Actual length of data item. |
| SCALE | integer | The data scale (if the data is of numeric type). |
| TYPE | integer | A numeric code that represents the data type of the column:<br>1 - SQL3_CHARACTER<br>2 - SQL3_NUMERIC<br>3 - SQL3_DECIMAL<br>4 - SQL3_INTEGER<br>5 - SQL3_SMALLINT<br>6 - SQL3_FLOAT<br>7 - SQL3_REAL<br>8 - SQL3_DOUBLE_PRECISION<br>9 - SQL3_DATE_TIME_TIMESTAMP<br>10 - SQL3_INTERVAL<br>12 - SQL3_CHARACTER_VARYING<br>13 - SQL3_ENUMERATED<br>14 - SQL3_BIT<br>15 - SQL3_BIT_VARYING<br>16 - SQL3_BOOLEAN |

## Example - Using a Descriptor to Return Data

The following simple application executes an SQL statement entered by an end user. The code sample demonstrates:

- how to use a SQL descriptor to execute a `SELECT` statement.
- how to find the data and metadata returned by the statement.

The application accepts an SQL statement from an end user, tests the statement to see if it includes the `SELECT` keyword, and executes the statement.

When invoking the application, an end user must provide the name of the database on which the SQL statement will be performed, and a string that contains the text of the query.

For example, a user might invoke the sample with the following command:

```
./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
{
 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" );

 }
 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++ )
{

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" );
}

/************************************************************
```

The code sample begins by including the prototypes and type definitions for the C `stdio` and `stdlib` libraries, SQL data type symbols, and the `SQLCA` (SQL communications area) structure:

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

The sample provides minimal error handling; when the application encounters an SQL error, it prints the error message to screen:

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

The application includes a forward-declaration for a function named `print_meta_data()` that will print the metadata found in a descriptor:

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

The following code specifies the column header information that the application will use when printing the metadata:

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

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

The following declaration section identifies the host variables that will contain the name of the database to which the application will connect, the content of the SQL Statement, and a host variable that will hold the number of columns in the result set (if any).

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

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

The application connects to the database (using the default credentials):

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

Next, the application allocates an SQL descriptor to hold the metadata for a statement:

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

The application uses a `PREPARE` statement to syntax check the string provided by the user:

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

and a `DESCRIBE` statement to move the metadata for the query into the SQL descriptor.

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

Then, the application interrogates the descriptor to discover the number of columns in the result set, and stores that in the host variable `col_count`.

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

If the column count is zero, the end user did not enter a `SELECT` statement; the application uses an `EXECUTE IMMEDIATE` statement to process the contents of the statement:

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

If the statement executes successfully, the application performs a `COMMIT`:

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

If the statement entered by the user is a `SELECT` statement (which we know because the column count is non-zero), the application declares a variable named `row`.

```
int row;
```

Then, the application allocates another descriptor that holds the description and the values of a specific row in the result set:

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

The application declares and opens a cursor for the prepared statement:

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

Loops through the rows in result set:

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

Then, uses a `FETCH` to retrieve the next row from the cursor into the descriptor:

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

The application confirms that the `FETCH` did not fail; if the `FETCH` fails, the application has reached the end of the result set, and breaks the loop:

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

The application checks to see if this is the first row of the cursor; if it is, the application prints the metadata for the row.

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

Next, it prints a record header containing the row number:

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

Then, it loops through each column in the row:

```
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;
```

The application interrogates the row descriptor `(row_desc)` to copy the column value `(:val)`, null indicator `(:ind)` and column name `(:name)` into the host variables declared above. Notice that you can retrieve multiple items from a descriptor using a comma-separated list.

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

If the null indicator `(ind)` is negative, the column value is `NULL`; if the null indicator is greater than `0`, the column value is too long to fit into the val host variable (so we print `<truncated>)`; otherwise, the null indicator is `0` (meaning `NOT NULL`) so we print the value. In each case, we prefix the value (or `<null>` or `<truncated>`) with the name of the column.

```
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" );
}
```

When the loop terminates, the application prints the number of rows fetched, and exits:

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

exit( 0 );
}
```

The `print_meta_data()` function extracts the metadata from a descriptor and prints the name, data type, and length of each column:

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

```
{
```

The application declares host variables:

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

The application then defines an array of character strings that map data type values `(numeric)` into data type names. We use the numeric value found in the descriptor to index into this array. For example, if we find that a given column is of type `2`, we can find the name of that type `(NUMERIC)` by writing `types[2]`.

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

The application retrieves the column count from the descriptor. Notice that the program refers to the descriptor using a host variable `(desc)` that contains the name of the descriptor. In most scenarios, you would use an identifier to refer to a descriptor, but in this case, the caller provided the descriptor name, so we can use a host variable to refer to the descriptor.

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

The application prints the column headers (defined at the beginning of this application):

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

Then, loops through each column found in the descriptor, and prints the name, type and length of each column.

```
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;
```

It retrieves the name, type code, and length of the current column:

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

If the numeric type code matches a 'known' type code (that is, a type code found in the `types[]` array), it sets `type_name` to the name of the corresponding type; otherwise, it sets `type_name` to `"unknown"`.

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

and prints the column number, name, type name, and length:

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

If you invoke the sample application with the following command:

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

The application returns:

```
col field                data                ret
num name                 type                len
--- -------------------- ------------------- ---
01: empno                NUMERIC             0004
02: ename                CHARACTER_VARYING   0004
03: job                  CHARACTER_VARYING   0007
04: mgr                  NUMERIC             0004
05: hiredate             DATE_TIME           0018
06: sal                  NUMERIC             0007
07: comm                 NUMERIC             0000
08: deptno               NUMERIC             0002

[RECORD 1]
  empno                : 7902
  ename                : FORD
  job                  : ANALYST
  mgr                  : 7566
  hiredate             : 03-DEC-81 00:00:00
  sal                  : 3000.00
  comm                 : <null>
  deptno               : 20

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

2 rows
```

# 4 Building and Executing Dynamic SQL Statements

The following examples demonstrate four techniques for building and executing dynamic SQL statements. Each example demonstrates processing a different combination of statement and input types:

- The first example demonstrates processing and executing a SQL statement that does not contain a `SELECT` statement and does not require input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 1.
- The second example demonstrates processing and executing a SQL statement that does not contain a `SELECT` statement, and contains a known number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 2.
- The third example demonstrates processing and executing a SQL statement that may contain a `SELECT` statement, and includes a known number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 3.
- The fourth example demonstrates processing and executing a SQL statement that may contain a `SELECT` statement, and includes an unknown number of input variables. This example corresponds to the techniques used by Oracle Dynamic SQL Method 4.

## Example - Executing a Non-query Statement Without Parameters

The following example demonstrates how to use the `EXECUTE IMMEDIATE` command to execute a SQL statement where the text of the statement is not known until you run the application. You cannot use `EXECUTE IMMEDIATE` to execute a statement that returns a result set. You cannot use `EXECUTE IMMEDIATE` to execute a statement that contains parameter placeholders.

The `EXECUTE IMMEDIATE` statement parses and plans the SQL statement each time it executes, which can have a negative impact on the performance of your application. If you plan to execute the same statement repeatedly, consider using the `PREPARE/EXECUTE` technique described in the next example.

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

static void handle_error(void);

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

```
{
  char    *insertStmt;

  EXEC SQL WHENEVER SQLERROR DO handle_error();

  EXEC SQL CONNECT :argv[1];

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

  EXEC SQL EXECUTE IMMEDIATE :insertStmt;

  fprintf(stderr, "ok\n");

  EXEC SQL COMMIT RELEASE;

  exit(EXIT_SUCCESS);
}


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

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK RELEASE;

  exit(EXIT_FAILURE);
}

/*************************************************************/
```

The code sample begins by including the prototypes and type definitions for the C `stdio`, `string`, and `stdlib` libraries, and providing basic infrastructure for the program:

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

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

The example then sets up an error handler; ECPGPlus calls the `handle_error()` function whenever a SQL error occurs:

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

Then, the example connects to the database using the credentials specified on the command line:

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

Next, the program uses an `EXECUTE IMMEDIATE` statement to execute a SQL statement, adding a row to the `dept` table:

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

EXEC SQL EXECUTE IMMEDIATE :insertStmt;
```

If the `EXECUTE IMMEDIATE` command fails for any reason, ECPGPlus will invoke the `handle_error()` function (which terminates the application after displaying an error message to the user). If the `EXECUTE IMMEDIATE` command succeeds, the application displays a message `(ok)` to the user, commits the changes, disconnects from the server, and terminates the application.

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

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS);
}
```

ECPGPlus calls the `handle_error()` function whenever it encounters a SQL error. The `handle_error()` function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application.

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

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK RELEASE;

  exit(EXIT_FAILURE);
}
```

## Example - Executing a Non-query Statement with a Specified Number of Placeholders

To execute a non-query command that includes a known number of parameter placeholders, you must first `PREPARE` the statement (providing a *statement handle*), and then `EXECUTE` the statement using the statement handle. When the application executes the statement, it must provide a *value* for each placeholder found in the statement.

When an application uses the `PREPARE/EXECUTE` mechanism, each SQL statement is parsed and planned once, but may execute many times (providing different *values* each time).

ECPGPlus will convert each parameter value to the type required by the SQL statement, if possible; if not possible, ECPGPlus will report an error.

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

static void handle_error(void);

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

  EXEC SQL WHENEVER SQLERROR DO handle_error();

  EXEC SQL CONNECT :argv[1];

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

  EXEC SQL PREPARE stmtHandle FROM :stmtText;

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

  fprintf(stderr, "ok\n");

  EXEC SQL COMMIT RELEASE;

  exit(EXIT_SUCCESS);
}
```

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

  exit(EXIT_FAILURE);
}
/*************************************************************/
```

The code sample begins by including the prototypes and type definitions for the C `stdio, string, stdlib`, and `sqlca` libraries, and providing basic infrastructure for the program:

```
#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;
```

The example then sets up an error handler; ECPGPlus calls the `handle_error()` function whenever a SQL error occurs:

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

Then, the example connects to the database using the credentials specified on the command line:

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

Next, the program uses a `PREPARE` statement to parse and plan a statement that includes three parameter markers - if the `PREPARE` statement succeeds, it will create a statement handle that you can use to execute the statement (in this example, the statement handle is named `stmtHandle`). You can execute a given statement multiple times using the same statement handle.

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

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

After parsing and planning the statement, the application uses the `EXECUTE` statement to execute the statement associated with the statement handle, substituting user-provided values for the parameter markers:

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

If the `EXECUTE` command fails for any reason, ECPGPlus will invoke the `handle_error()` function (which terminates the application after displaying an error message to the user). If the `EXECUTE` command succeeds, the application displays a message `(ok)` to the user, commits the changes, disconnects from the server, and terminates the application.

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

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS);
}
```

ECPGPlus calls the `handle_error()` function whenever it encounters a SQL error. The `handle_error()` function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application.

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

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

## Example - Executing a Query With a Known Number of Placeholders

This example demonstrates how to execute a *query* with a known number of input parameters, and with a known number of columns in the result set. This method uses the `PREPARE` statement to parse and plan a query, before opening a cursor and iterating through the result set.

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

static void handle_error(void);

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

  EXEC SQL WHENEVER SQLERROR DO handle_error();

  EXEC SQL CONNECT :argv[1];

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

  EXEC SQL DECLARE empCursor CURSOR FOR queryHandle;

  EXEC SQL OPEN empCursor USING :argv[2];

  EXEC SQL WHENEVER NOT FOUND DO break;

  while(true)
  {

    EXEC SQL FETCH empCursor INTO :empno, :ename;

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

  EXEC SQL CLOSE empCursor;

  EXEC SQL COMMIT RELEASE;

  exit(EXIT_SUCCESS);
}
```

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

  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK RELEASE;

  exit(EXIT_FAILURE);
}

/************************************************************/
```

The code sample begins by including the prototypes and type definitions for the C `stdio, string, stdlib, stdbool`, and `sqlca` libraries, and providing basic infrastructure for the program:

```
#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];
```

The example then sets up an error handler; ECPGPlus calls the `handle_error()` function whenever a SQL error occurs:

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

Then, the example connects to the database using the credentials specified on the command line:

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

Next, the program uses a `PREPARE` statement to parse and plan a query that includes a single parameter marker - if the `PREPARE` statement succeeds, it will create a statement handle that you can use to execute the statement (in this example, the statement handle is named `stmtHandle`). You can execute a given statement multiple times using the same statement handle.

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

The program then declares and opens the cursor, `empCursor`, substituting a user-provided value for the parameter marker in the prepared `SELECT` statement. Notice that the `OPEN` statement includes a `USING` clause: the `USING` clause must provide a *value* for each placeholder found in the query:

```
EXEC SQL DECLARE empCursor CURSOR FOR stmtHandle;

EXEC SQL OPEN empCursor USING :argv[2];

EXEC SQL WHENEVER NOT FOUND DO break;

while(true)
{
```

The program iterates through the cursor, and prints the employee number and name of each employee in the selected department:

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

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

The program then closes the cursor, commits any changes, disconnects from the server, and terminates the application.

```
EXEC SQL CLOSE empCursor;

EXEC SQL COMMIT RELEASE;

exit(EXIT_SUCCESS);
}
```

The application calls the `handle_error()` function whenever it encounters a SQL error. The `handle_error()` function prints the content of the error message, resets the error handler, rolls back any changes, disconnects from the database, and terminates the application.

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

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

  exit(EXIT_FAILURE);
}
```

## Example - Executing a Query With an Unknown Number of Variables

The next example demonstrates executing a query with an unknown number of input parameters and/or columns in the result set. This type of query may occur when you prompt the user for the text of the query, or when a query is assembled from a form on which the user chooses from a number of conditions (i.e., a filter).

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

SQLDA *params;
SQLDA *results;

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

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

  EXEC SQL WHENEVER SQLERROR sqlprint;

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

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

  EXEC SQL PREPARE stmt FROM :stmtText;

  EXEC SQL DECLARE dynCursor CURSOR FOR stmt;

  bindParams();

  EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

  displayResultSet(20);
}

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

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

     params->N = params->F;

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

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

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

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

```
}

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

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

     results->N = results->F;

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

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

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

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

            if (precision == 0)
               precision = 38;

            length = precision + 3;
            break;
           }

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

```
          }

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

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

      EXEC SQL WHENEVER NOT FOUND DO break;

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

        EXEC SQL FETCH dynCursor USING DESCRIPTOR results;

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


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

The code sample begins by including the prototypes and type definitions for the C `stdio` and `stdlib` libraries. In addition, the program includes the `sqlda.h` and `sqlcpr.h` header files. `sqlda.h` defines the SQLDA structure used throughout this example. `sqlcpr.h` defines a small set of functions used to interrogate the metadata found in an SQLDA structure.

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

Next, the program declares pointers to two SQLDA structures. The first SQLDA structure `(params)` will be used to describe the metadata for any parameter markers found in the dynamic query text. The second SQLDA structure `(results)` will contain both the metadata and the result set obtained by executing the dynamic query.

```
SQLDA *params;
SQLDA *results;
```

The program then declares two helper functions (defined near the end of the code sample):

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

Next, the program declares three host variables; the first two (`username` and `password`) are used to connect to the database server; the third host variable `(stmtTxt)` is a NULL-terminated C string containing the text of the query to execute. Notice that the values for these three host variables are derived from the command-line arguments. When the program begins execution, it sets up an error handler and then connects to the database server:

```
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;
```

Next, the program calls the `sqlald()` function to allocate the memory required for each descriptor. Each descriptor contains (among other things):

- a pointer to an array of column names
- a pointer to an array of indicator names
- a pointer to an array of data types
- a pointer to an array of lengths

- a pointer to an array of data values.

When you allocate an `SQLDA` descriptor, you specify the maximum number of columns you expect to find in the result set (for `SELECT`-list descriptors) or the maximum number of parameters you expect to find the dynamic query text (for bind-variable descriptors) - in this case, we specify that we expect no more than 20 columns and 20 parameters. You must also specify a maximum length for each column (or parameter) name and each indicator variable name - in this case, we expect names to be no more than 64 bytes long.

See SQLDA Structure section for a complete description of the `SQLDA` structure.

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

After allocating the `SELECT`-list and bind descriptors, the program prepares the dynamic statement and declares a cursor over the result set.

```
EXEC SQL PREPARE stmt FROM :stmtText;

EXEC SQL DECLARE dynCursor CURSOR FOR stmt;
```

Next, the program calls the `bindParams()` function. The `bindParams()` function examines the bind descriptor `(params)` and prompt the user for a value to substitute in place of each parameter marker found in the dynamic query.

```
bindParams();
```

Finally, the program opens the cursor (using the parameter values supplied by the user, if any) and calls the `displayResultSet()` function to print the result set produced by the query.

```
EXEC SQL OPEN dynCursor USING DESCRIPTOR params;

displayResultSet();
}
```

The `bindParams()` function determines whether the dynamic query contains any parameter markers, and, if so, prompts the user for a value for each parameter and then binds that value to the corresponding marker. The `DESCRIBE BIND VARIABLE` statement populates the `params` SQLDA structure with information describing each parameter marker.

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

If the statement contains no parameter markers, `params->F` will contain 0. If the statement contains more parameters than will fit into the descriptor, `params->F` will contain a negative number (in this case, the absolute value of `params->F` indicates the number of parameter markers found in the statement). If `params->F` contains a positive number, that number indicates how many parameter markers were found in the statement.

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

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

Next, the program executes a loop that prompts the user for a value, iterating once for each parameter marker found in the statement.

```
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);
```

After prompting the user for a value for a given parameter, the program *binds* that value to the parameter by setting `params->T[i]` to indicate the data type of the value (see `Type Codes` for a list of type codes), `params->L[i]` to the length of the value (we subtract one to trim off the trailing new-line character added by `fgets()`), and `params->V[i]` to point to a copy of the NULL-terminated string provided by the user.

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

The `displayResultSet()` function loops through each row in the result set and prints the value found in each column. `displayResultSet()` starts by executing a `DESCRIBE SELECT LIST` statement - this statement populates an SQLDA descriptor `(results)`

with a description of each column in the result set.

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

If the dynamic statement returns no columns (that is, the dynamic statement is not a `SELECT` statement), `results->F` will contain 0. If the statement returns more columns than will fit into the descriptor, `results->F` will contain a negative number (in this case, the absolute value of `results->F` indicates the number of columns returned by the statement). If `results->F` contains a positive number, that number indicates how many columns where returned by the query.

```
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;
```

Next, the program enters a loop, iterating once for each column in the result set:

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

To decode the type code found in `results->T`, the program invokes the `sqlnul()` function (see the description of the `T` member of the SQLDA structure in the The SQLDA Structure). This call to `sqlnul()` modifies `results->T[col]` to contain only the type code (the nullability flag is copied to `null_permitted`). This step is necessary because the `DESCRIBE SELECT LIST` statement encodes the type of each column and the nullability of each column into the `T` array.

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

After decoding the actual data type of the column, the program modifies the results descriptor to tell ECPGPlus to return each value in the form of a NULL-terminated string. Before modifying the descriptor, the program must compute the amount of space required to hold each value. To make this computation, the program examines the maximum length of each

column `(results->V[col])` and the data type of each column `(results->T[col])`.

For numeric values (where `results->T[col] = 2`), the program calls the `sqlprc()` function to extract the precision and scale from the column length. To compute the number of bytes required to hold a numeric value in string form, `displayResultSet()` starts with the precision (that is, the maximum number of digits) and adds three bytes for a sign character, a decimal point, and a NULL terminator.

```
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;
   }
```

For date values, the program uses a somewhat arbitrary, hard-coded length of 30. In a real-world application, you may want to more carefully compute the amount of space required.

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

For a value of any type other than date or numeric, `displayResultSet()` starts with the maximum column width reported by `DESCRIBE SELECT LIST` and adds one extra byte for the NULL terminator. Again, in a real-world application you may want to include more careful calculations for other data types.

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

After computing the amount of space required to hold a given column, the program allocates enough memory to hold the value, sets `results->L[col]` to indicate the number of bytes found at `results->V[col]`, and set the type code for the column `(results->T[col])` to `1` to instruct the upcoming `FETCH` statement to return the value in the form of a NULL-terminated string.

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

At this point, the results descriptor is configured such that a `FETCH` statement can copy each value into an appropriately sized buffer in the form of a NULL-terminated string.

Next, the program defines a new error handler to break out of the upcoming loop when the cursor is exhausted.

```
EXEC SQL WHENEVER NOT FOUND DO break;

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

The program executes a `FETCH` statement to fetch the next row in the cursor into the `results` descriptor. If the `FETCH` statement fails (because the cursor is exhausted), control transfers to the end of the loop because of the `EXEC SQL WHENEVER` directive found before the top of the loop.

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

The `FETCH` statement will populate the following members of the results descriptor:

- `*results->I[col]` will indicate whether the column contains a NULL value `(-1)` or a non-NULL value `(0)`. If the value non-NULL but too large to fit into the space provided, the value is truncated and `*results->I[col]` will contain a positive value.
- `results->V[col]` will contain the value fetched for the given column `(unless *results->I[col]` indicates that the column value is NULL).
- `results->L[col]` will contain the length of the value fetched for the given column

Finally, `displayResultSet()` iterates through each column in the result set, examines the corresponding NULL indicator, and prints the value. The result set is not aligned - instead, each value is separated from the previous value by a comma.

```
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");
}
}
}
/*************************************************************/
```

# 5 Error Handling

ECPGPlus provides two methods to detect and handle errors in embedded SQL code:

- A client application can examine the `sqlca` data structure for error messages, and supply customized error handling for your client application.
- A client application can include `EXEC SQL WHENEVER` directives to instruct the ECPGPlus compiler to add error-handling code.

## Error Handling with sqlca

`sqlca` (SQL communications area) is a global variable used by `ecpglib` to communicate information from the server to the client application. After executing a SQL statement (for example, an `INSERT` or `SELECT` statement) you can inspect the contents of `sqlca` to determine if the statement has completed successfully or if the statement has failed.

`sqlca` has the following structure:

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

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

} sqlca;
```

Use the following directive to implement `sqlca` functionality:

```
EXEC SQL INCLUDE sqlca;
```

If you include the `ecpg` directive, you do not need to `#include` the `sqlca.h` file in the client application's header declaration.

The Advanced Server `sqlca` structure contains the following members:

`sqlcaid`

`sqlcaid` contains the string: `"SQLCA"`.

`sqlabc`

`sqlabc` contains the size of the `sqlca` structure.

`sqlcode`

The `sqlcode` member has been deprecated with SQL 92; Advanced Server supports `sqlcode` for backward compatibility, but you should use the `sqlstate` member when writing new code.

`sqlcode` is an integer value; a positive `sqlcode` value indicates that the client application has encountered a harmless processing condition, while a negative value indicates a warning or error.

If a statement processes without error, `sqlcode` will contain a value of `0`. If the client application encounters an error (or warning) during a statement's execution, `sqlcode` will contain the last code returned.

The SQL standard defines only a positive value of 100, which indicates that he most recent SQL statement processed returned/affected no rows. Since the SQL standard does not define

other `sqlcode` values, please be aware that the values assigned to each condition may vary from database to database.

`sqlerrm` is a structure embedded within `sqlca`, composed of two members:

`sqlerrml`

`sqlerrml` contains the length of the error message currently stored in `sqlerrmc`.

`sqlerrmc`

`sqlerrmc` contains the null-terminated message text associated with the code stored in `sqlstate`. If a message exceeds 149 characters in length, `ecpglib` will truncate the error message.

`sqlerrp`

`sqlerrp` contains the string `"NOT SET"`.

`sqlerrd` is an array that contains six elements:

- `sqlerrd[1]` contains the OID of the processed row (if applicable).
- `sqlerrd[2]` contains the number of processed or returned rows.
- `sqlerrd[0], sqlerrd[3], sqlerrd[4]` and `sqlerrd[5]` are unused.

`sqlwarn` is an array that contains 8 characters:

- `sqlwarn[0]` contains a value of `'W'` if any other element within `sqlwarn` is set to `'W'`.
- `sqlwarn[1]` contains a value of `'W'` if a data value was truncated when it was stored in a host variable.
- `sqlwarn[2]` contains a value of `'W'` if the client application encounters a non-fatal warning.
- `sqlwarn[3], sqlwarn[4], sqlwarn[5], sqlwarn[6]`, and `sqlwarn[7]` are unused.

`sqlstate`

`sqlstate` is a 5 character array that contains a SQL-compliant status code after the execution of a statement from the client application. If a statement processes without error, `sqlstate` will contain a value of `00000`. Please note that `sqlstate` is *not* a null-terminated string.

`sqlstate` codes are assigned in a hierarchical scheme:

- The first two characters of `sqlstate` indicate the general class of the condition.
- The last three characters of `sqlstate` indicate a specific status within the class.

If the client application encounters multiple errors (or warnings) during an SQL statement's execution `sqlstate` will contain the last code returned.

The following table lists the `sqlstate` and `sqlcode` values, as well as the symbolic name and error description for the related condition:

| sqlstate | sqlcode (Deprecated) | Symbolic Name | Description |
|---|---|---|---|
| `YE001` | `-12` | `ECPG_OUT_OF_MEMORY` | Virtual memory is exhausted. |
| `YE002` | `-200` | `ECPG_UNSUPPORTED` | The preprocessor has generated an unrecognized item. Could indicate incompatibility between the preprocessor and the library. |
| `07001`, or `07002` | `-201` | `ECPG_TOO_MANY_ARGUMENTS` | The program specifies more variables than the command expects. |
| `07001`, or `07002` | `-202` | `ECPG_TOO_FEW_ARGUMENTS` | The program specified fewer variables than the command expects. |
| `21000` | `-203` | `ECPG_TOO_MANY_MATCHES` | The SQL command has returned multiple rows, but the statement was prepared to receive a single row. |

| sqlstate | sqlcode (Deprecated) | Symbolic Name | Description |
|---|---|---|---|
| 42804 | -204 | ECPG_INT_FORMAT | The host variable (defined in the C code) is of type INT, and the selected data is of a type that cannot be converted into an INT. ecpglib uses the strtol() function to convert string values into numeric form. |
| 42804 | -205 | ECPG_UINT_FORMAT | The host variable (defined in the C code) is an unsigned INT, and the selected data is of a type that cannot be converted into an unsigned INT. ecpglib uses the strtoul() function to convert string values into numeric form. |
| 42804 | -206 | ECPG_FLOAT_FORMAT | The host variable (defined in the C code) is of type FLOAT, and the selected data is of a type that cannot be converted into an FLOAT. ecpglib uses the strtod() function to convert string values into numeric form. |

| sqlstate | sqlcode (Deprecated) | Symbolic Name | Description |
|---|---|---|---|
| 42804 | -211 | ECPG_CONVERT_BOOL | The host variable (defined in the C code) is of type BOOL, and the selected data cannot be stored in a BOOL. |
| YE002 | -2-1 | ECPG_EMPTY | The statement sent to the server was empty. |
| 22002 | -213 | ECPG_MISSING_INDICATOR | A NULL indicator variable has not been supplied for the NULL value returned by the server (the client application has received an unexpected NULL value). |
| 42804 | -214 | ECPG_NO_ARRAY | The server has returned an array, and the corresponding host variable is not capable of storing an array. |
| 42804 | -215 | ECPG_DATA_NOT_ARRAY | The server has returned a value that is not an array into a host variable that expects an array value. |
| 08003 | -220 | ECPG_NO_CONN | The client application has attempted to use a non-existent connection. |

| sqlstate | sqlcode (Deprecated) | Symbolic Name | Description |
|---|---|---|---|
| YE002 | -221 | ECPG_NOT_CONN | The client application has attempted to use an allocated, but closed connection. |
| 26000 | -230 | ECPG_INVALID_STMT | The statement has not been prepared. |
| 33000 | -240 | ECPG_UNKNOWN_DESCRIPTOR | The specified descriptor is not found. |
| 07009 | -241 | ECPG_INVALID_DESCRIPTOR_INDEX | The descriptor index is out-of-range. |
| YE002 | -242 | ECPG_UNKNOWN_DESCRIPTOR_ITEM | The client application has requested an invalid descriptor item (internal error). |
| 07006 | -243 | ECPG_VAR_NOT_NUMERIC | A dynamic statement has returned a numeric value for a non-numeric host variable. |
| 07006 | -244 | ECPG_VAR_NOT_CHAR | A dynamic SQL statement has returned a CHAR value, and the host variable is not a CHAR. |
| | -400 | ECPG_PGSQL | The server has returned an error message; the resulting message contains the error text. |

| sqlstate | sqlcode (Deprecated) | Symbolic Name | Description |
|---|---|---|---|
| 08007 | -401 | ECPG_TRANS | The server cannot start, commit or rollback the specified transaction. |
| 08001 | -402 | ECPG_CONNECT | The client application's attempt to connect to the database has failed. |
| 02000 | 100 | ECPG_NOT_FOUND | The last command retrieved or processed no rows, or you have reached the end of a cursor. |

## EXEC SQL WHENEVER

Use the `EXEC SQL WHENEVER` directive to implement simple error handling for client applications compiled with ECPGPlus. The syntax of the directive is:

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

This directive instructs the ECPG compiler to insert error-handling code into your program.

The code instructs the client application that it should perform a specified action if the client application detects a given condition. The *condition* may be one of the following:

`SQLERROR`

A `SQLERROR` condition exists when `sqlca.sqlcode` is less than zero.

`SQLWARNING`

A `SQLWARNING` condition exists when `sqlca.sqlwarn[0]` contains a `'W'`.

`NOT FOUND`

A `NOT FOUND` condition exists when `sqlca.sqlcode` is `ECPG_NOT_FOUND` (when a query returns no data).

You can specify that the client application perform one of the following *actions* if it encounters one of the previous conditions:

`CONTINUE`

Specify `CONTINUE` to instruct the client application to continue processing, ignoring the current `condition`. `CONTINUE` is the default action.

`DO CONTINUE`

An action of `DO CONTINUE` will generate a `CONTINUE` statement in the emitted C code that if it encounters the condition, skips the rest of the code in the loop and continues with the next iteration. You can only use it within a loop.

`GOTO label`

or

`GO TO label`

Use a C `goto` statement to jump to the specified `label`.

`SQLPRINT`

Print an error message to `stderr` (standard error), using the `sqlprint()` function. The `sqlprint()` function prints `sql error`, followed by the contents of `sqlca.sqlerrm.sqlerrmc`.

`STOP`

Call `exit(1)` to signal an error, and terminate the program.

`DO BREAK`

Execute the C `break` statement. Use this action in loops, or `switch` statements.

`CALL name(args)`

or

`DO name(args)`

Invoke the C function specified by the name `parameter`, using the parameters specified in the `args` parameter.

**Example:**

The following code fragment prints a message if the client application encounters a warning, and aborts the application if it encounters an error:

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

!!! Note The ECPGPlus compiler processes your program from top to bottom, even though the client application may not *execute* from top to bottom. The compiler directive is applied to each line in order, and remains in effect until the compiler encounters another directive. If the control of the flow within your program is not top-to-bottom, you should consider adding error-handling directives to any parts of the program that may be inadvertently missed during compilation.

# 6 Reference

The sections that follow describe ecpgPlus language elements:

- C-Preprocessor Directives
- Supported C Data Types
- Type Codes
- The SQLDA Structure
- ECPGPlus Statements

## C-preprocessor Directives

The ECPGPlus C-preprocessor enforces two behaviors that are dependent on the mode in which you invoke ECPGPlus:

- `PROC` mode
- non-`PROC` mode

### Compiling in PROC Mode

In `PROC` mode, ECPGPlus allows you to:

- Declare host variables outside of an `EXEC SQL BEGIN/END DECLARE SECTION`.
- Use any C variable as a host variable as long as it is of a data type compatible with ECPG.

When you invoke ECPGPlus in `PROC` mode (by including the `-C PROC` keywords), the ECPG compiler honors the following C-preprocessor directives:

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

Pre-processor directives are used to effect or direct the code that is received by the compiler. For example, using the following code sample:

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

If you invoke ECPGPlus with the following command-line arguments:

```
ecpg –C PROC –DHAVE_LONG_LONG=1
```

ECPGPlus will copy the entire fragment (without change) to the output file, but will only send the following tokens to the ECPG parser:

```
long long customerBalance;
```

On the other hand, if you invoke ECPGPlus with the following command-line arguments:

```
ecpg –C PROC –DHAVE_LONG_LONG=0
```

The ECPG parser will receive the following tokens:

```
double customerBalance;
```

If your code uses preprocessor directives to filter the code that is sent to the compiler, the complete code is retained in the original code, while the ECPG parser sees only the processed token stream.

You can also use compatible syntax when executing the following preprocessor directives with an `EXEC` directive:

```
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
```

For example, if your code includes the following:

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

If you invoke ECPGPlus with the following command-line arguments:

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

ECPGPlus will send the following tokens to the output file, and the ECPG parser:

```
long long customerBalance;
```

!!! Note The `EXEC ORACLE` pre-processor directives only work if you specify `-C PROC` on the ECPG command line.

**Using the SELECT_ERROR Precompiler Option**

When using ECPGPlus in compatible mode, you can use the `SELECT_ERROR` precompiler option to instruct your program how to handle result sets that contain more rows than the host variable can accommodate. The syntax is:

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

The default value is `YES`; a `SELECT` statement will return an error message if the result set

exceeds the capacity of the host variable. Specify `NO` to instruct the program to suppress error messages when a `SELECT` statement returns more rows than a host variable can accommodate.

Use `SELECT_ERROR` with the `EXEC ORACLE OPTION` directive.

**Compiling in non-PROC Mode**

If you do not include the `-C PROC` command-line option:

- C preprocessor directives are copied to the output file without change.
- You must declare the type and name of each C variable that you intend to use as a host variable within an `EXEC SQL BEGIN/END DECLARE` section.

When invoked in non-`PROC` mode, ECPG implements the behavior described in the PostgreSQL Core documentation.

## Supported C Data Types

An ECPGPlus application must deal with two sets of data types: SQL data types (such as `SMALLINT`, `DOUBLE PRECISION` and `CHARACTER VARYING`) and C data types (like `short`, `double` and `varchar[n]`). When an application fetches data from the server, ECPGPlus will map each SQL data type to the type of the C variable into which the data is returned.

In general, ECPGPlus can convert most SQL server types into similar C types, but not all combinations are valid. For example, ECPGPlus will try to convert a SQL character value into a C integer value, but the conversion may fail (at execution time) if the SQL character value contains non-numeric characters. The reverse is also true; when an application sends a value to the server, ECPGPlus will try to convert the C data type into the required SQL type. Again, the conversion may fail (at execution time) if the C value cannot be converted into the required SQL type.

ECPGPlus can convert any SQL type into C character values `(char[n]` or `varchar[n])`. Although it is safe to convert any SQL type to/from `char[n]` or `varchar[n]`, it is often convenient to use more natural C types such as `int`, `double`, or `float`.

The supported C data types are:

- `short`
- `int`
- `unsigned int`
- `long long int`

- `float`
- `double`
- `char[n+1]`
- `varchar[n+1]`
- `bool`
- and any equivalent created by a `typedef`

In addition to the numeric and character types supported by C, the `pgtypeslib` run-time library offers custom data types (and functions to operate on those types) for dealing with date/time and exact numeric values:

- `timestamp`
- `interval`
- `date`
- `decimal`
- `numeric`

To use a data type supplied by `pgtypeslib`, you must `#include` the proper header file.

## Type Codes

The following table contains the type codes for *external* data types. An external data type is used to indicate the type of a C host variable. When an application binds a value to a parameter or binds a buffer to a `SELECT`-list item, the type code in the corresponding SQLDA descriptor `(descriptor->T[column])` should be set to one of the following values:

| Type Code | Host Variable Type (C Data Type) |
|---|---|
| `1, 2, 8, 11, 12, 15, 23, 24, 91, 94, 95, 96, 97` | `char[]` |
| `3` | `int` |
| `4, 7, 21` | `float` |
| `5, 6` | `null-terminated string (char[length+1])` |
| `9` | `varchar` |
| `22` | `double` |
| `68` | `unsigned int` |

The following table contains the type codes for *internal* data types. An internal type code is used to indicate the type of a value as it resides in the database. The `DESCRIBE SELECT LIST` statement populates the data type array `(descriptor->T[column])` using the

following values.

| Internal Type Code | Server Type |
|---|---|
| 1 | VARCHAR2 |
| 2 | NUMBER |
| 8 | LONG |
| 11 | ROWID |
| 12 | DATE |
| 23 | RAW |
| 24 | LONG RAW |
| 96 | CHAR |
| 100 | BINARY FLOAT |
| 101 | BINARY DOUBLE |
| 104 | UROWID |
| 187 | TIMESTAMP |
| 188 | TIMESTAMP W/TIMEZONE |
| 189 | INTERVAL YEAR TO MONTH |
| 190 | INTERVAL DAY TO SECOND |
| 232 | TIMESTAMP LOCAL_TZ |

## The SQLDA Structure

Oracle Dynamic SQL method 4 uses the SQLDA data structure to hold the data and metadata for a dynamic SQL statement. A SQLDA structure can describe a set of input parameters corresponding to the parameter markers found in the text of a dynamic statement or the result set of a dynamic statement. The layout of the SQLDA structure is:

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

```
  char  **X; /* Indicator names              */
  short  *Y; /* Indicator name maximum lengths */
  short  *Z; /* Indicator name actual lengths     */
};
```

**Parameters**

`N - maximum number of entries`

The N structure member contains the maximum number of entries that the SQLDA may describe. This member is populated by the `sqlald()` function when you allocate the SQLDA structure. Before using a descriptor in an `OPEN` or `FETCH` statement, you must set `N` to the *actual* number of values described.

`V - data values`

The `V` structure member is a pointer to an array of data values.

- For a `SELECT`-list descriptor, `V` points to an array of values returned by a `FETCH` statement (each member in the array corresponds to a column in the result set).
- For a bind descriptor, `V` points to an array of parameter values (you must populate the values in this array before opening a cursor that uses the descriptor).

Your application must allocate the space required to hold each value. Refer to displayResultSet function for an example of how to allocate space for `SELECT`-list values.

`L - length of each data value`

The `L` structure member is a pointer to an array of lengths. Each member of this array must indicate the amount of memory available in the corresponding member of the `V` array. For example, if `V[5]` points to a buffer large enough to hold a 20-byte NULL-terminated string, `L[5]` should contain the value 21 (20 bytes for the characters in the string plus 1 byte for the NULL-terminator). Your application must set each member of the `L` array.

`T - data types`

The `T` structure member points to an array of data types, one for each column (or parameter) described by the descriptor.

- For a bind descriptor, you must set each member of the `T` array to tell ECPGPlus the data type of each parameter.
- For a `SELECT`-list descriptor, the `DESCRIBE SELECT LIST` statement sets each member of the `T` array to reflect the type of data found in the corresponding column.

You may change any member of the `T` array before executing a `FETCH` statement to force ECPGPlus to convert the corresponding value to a specific data type. For example, if the

`DESCRIBE SELECT LIST` statement indicates that a given column is of type `DATE`, you may change the corresponding `T` member to request that the next `FETCH` statement return that value in the form of a NULL-terminated string. Each member of the `T` array is a numeric type code (see Type Codes for a list of type codes). The type codes returned by a `DESCRIBE SELECT LIST` statement differ from those expected by a `FETCH` statement. After executing a `DESCRIBE SELECT LIST` statement, each member of `T` encodes a data type *and* a flag indicating whether the corresponding column is nullable. You can use the `sqlnul()` function to extract the type code and nullable flag from a member of the T array. The signature of the `sqlnul()` function is as follows:

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

For example, to find the type code and nullable flag for the third column of a descriptor named results, you would invoke `sqlnul()` as follows:

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

### I - indicator variables

The `I` structure member points to an array of indicator variables. This array is allocated for you when your application calls the `sqlald()` function to allocate the descriptor.

- For a `SELECT`-list descriptor, each member of the `I` array indicates whether the corresponding column contains a NULL (non-zero) or non-NULL (zero) value.
- For a bind parameter, your application must set each member of the `I` array to indicate whether the corresponding parameter value is NULL.

### F - number of entries

The `F` structure member indicates how many values are described by the descriptor (the `N` structure member indicates the *maximum* number of values which may be described by the descriptor; `F` indicates the actual number of values). The value of the `F` member is set by ECPGPlus when you execute a `DESCRIBE` statement. `F` may be positive, negative, or zero.

- For a `SELECT`-list descriptor, `F` will contain a positive value if the number of columns in the result set is equal to or less than the maximum number of values permitted by the descriptor (as determined by the `N` structure member); 0 if the statement is *not* a `SELECT` statement, or a negative value if the query returns more columns than allowed by the `N` structure member.
- For a bind descriptor, `F` will contain a positive number if the number of parameters found in the statement is less than or equal to the maximum number of values permitted by the descriptor (as determined by the `N` structure member); 0 if the statement contains no parameters markers, or a negative value if the statement contains more parameter markers than allowed by the `N` structure member.

If `F` contains a positive number (after executing a `DESCRIBE` statement), that number reflects the count of columns in the result set (for a `SELECT`-list descriptor) or the number of parameter markers found in the statement (for a bind descriptor). If `F` contains a negative value, you may compute the absolute value of `F` to discover how many values (or parameter markers) are required. For example, if `F` contains `-24` after describing a `SELECT` list, you know that the query returns 24 columns.

### `S - column/parameter names`

The `S` structure member points to an array of NULL-terminated strings.

- For a `SELECT`-list descriptor, the `DESCRIBE SELECT LIST` statement sets each member of this array to the name of the corresponding column in the result set.
- For a bind descriptor, the `DESCRIBE BIND VARIABLES` statement sets each member of this array to the name of the corresponding bind variable.

In this release, the name of each bind variable is determined by the left-to-right order of the parameter marker within the query - for example, the name of the first parameter is always `?0`, the name of the second parameter is always `?1`, and so on.

### `M - maximum column/parameter name length`

The `M` structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the `S` array (that is, `M[0]` specifies the maximum length of the column/parameter name found at `S[0]`). This array is populated by the `sqlald()` function.

### `C - actual column/parameter name length`

The `C` structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the `S` array (that is, `C[0]` specifies the actual length of the column/parameter name found at `S[0]`).

This array is populated by the `DESCRIBE` statement.

### `X - indicator variable names`

The `X` structure member points to an array of NULL-terminated strings -each string represents the name of a NULL indicator for the corresponding value.

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

### `Y - maximum indicator name length`

The `Y` structure member points to an array of lengths. Each member in this array specifies the *maximum* length of the corresponding member of the `X` array (that is, `Y[0]` specifies the

maximum length of the indicator name found at `X[0]`).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

`Z - actual indicator name length`

The `Z` structure member points to an array of lengths. Each member in this array specifies the *actual* length of the corresponding member of the `X` array (that is, `Z[0]` specifies the actual length of the indicator name found at `X[0]`).

This array is not used by ECPGPlus, but is provided for compatibility with Pro*C applications.

## ECPGPlus Statements

An embedded SQL statement allows your client application to interact with the server, while an embedded directive is an instruction to the ECPGPlus compiler.

You can embed any Advanced Server SQL statement in a C program. Each statement should begin with the keywords `EXEC SQL`, and must be terminated with a semi-colon (;). Within the C program, a SQL statement takes the form:

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

Where `sql_command_body` represents a standard SQL statement. You can use a host variable anywhere that the SQL statement expects a value expression. For more information about substituting host variables for value expressions, refer to Declaring Host Variables.

ECPGPlus extends the PostgreSQL server-side syntax for some statements; for those statements, syntax differences are outlined in the following reference sections. For a complete reference to the supported syntax of other SQL commands, refer to the *PostgreSQL Core Documentation* available at:

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

## ALLOCATE DESCRIPTOR

Use the `ALLOCATE DESCRIPTOR` statement to allocate an SQL descriptor area:

```
EXEC SQL [FOR <array_size>] ALLOCATE DESCRIPTOR
```

```
<descriptor_name>
    [WITH MAX <variable_count>];
```

Where:

`array_size` is a variable that specifies the number of array elements to allocate for the descriptor. `array_size` may be an `INTEGER` value or a host variable.

`descriptor_name` is the host variable that contains the name of the descriptor, or the name of the descriptor. This value may take the form of an identifier, a quoted string literal, or of a host variable.

`variable_count` specifies the maximum number of host variables in the descriptor. The default value of `variable_count` is `100`.

The following code fragment allocates a descriptor named `emp_query` that may be processed as an array `(emp_array)`:

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

## CALL

Use the `CALL` statement to invoke a procedure or function on the server. The `CALL` statement works only on Advanced Server. The `CALL` statement comes in two forms; the first form is used to call a *function*:

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

The second form is used to call a *procedure*:

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

Where:

`program_name` is the name of the stored procedure or function that the `CALL` statement invokes. The program name may be schema-qualified or package-qualified (or both); if you do not specify the schema or package in which the program resides, ECPGPlus will use the value of `search_path` to locate the program.

`actual_arguments` specifies a comma-separated list of arguments required by the program. Note that each `actual_argument` corresponds to a formal argument expected by the program. Each formal argument may be an `IN` parameter, an `OUT` parameter, or an

`INOUT` parameter.

`:ret_variable` specifies a host variable that will receive the value returned if the program is a function.

`:ret_indicator` specifies a host variable that will receive the indicator value returned, if the program is a function.

For example, the following statement invokes the `get_job_desc` function with the value contained in the `:ename` host variable, and captures the value returned by that function in the `:job` host variable:

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

## CLOSE

Use the `CLOSE` statement to close a cursor, and free any resources scurrently in use by the cursor. A client application cannot fetch rows from a closed cursor. The syntax of the `CLOSE` statement is:

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

Where:

`cursor_name` is the name of the cursor closed by the statement. The cursor name may take the form of an identifier or of a host variable.

The `OPEN` statement initializes a cursor. Once initialized, a cursor result set will remain unchanged unless the cursor is re-opened. You do not need to `CLOSE` a cursor before re-opening it.

To manually close a cursor named `emp_cursor`, use the command:

```
EXEC SQL CLOSE emp_cursor;
```

A cursor is automatically closed when an application terminates.

## COMMIT

Use the `COMMIT` statement to complete the current transaction, making all changes

permanent and visible to other users. The syntax is:

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

Where:

`database_name` is the name of the database (or host variable that contains the name of the database) in which the work resides. This value may take the form of an unquoted string literal, or of a host variable.

For compatibility, ECPGPlus accepts the `COMMENT` clause without error but does *not* store any text included with the `COMMENT` clause.

Include the `RELEASE` clause to close the current connection after performing the commit.

For example, the following command commits all work performed on the `dept` database and closes the current connection:

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

By default, statements are committed only when a client application performs a `COMMIT` statement. Include the `-t` option when invoking ECPGPlus to specify that a client application should invoke `AUTOCOMMIT` functionality. You can also control `AUTOCOMMIT` functionality in a client application with the following statements:

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

and

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

## CONNECT

Use the `CONNECT` statement to establish a connection to a database. The `CONNECT` statement is available in two forms - one form is compatible with Oracle databases, the other is not.

The first form is compatible with Oracle databases:

```
EXEC SQL CONNECT
  {{:<user_name> IDENTIFIED BY :<password>}   | :
```

```
<connection_id>}
  [AT <database_name>]
  [USING :database_string]
  [ALTER AUTHORIZATION :new_password];
```

Where:

`user_name` is a host variable that contains the role that the client application will use to connect to the server.

`password` is a host variable that contains the password associated with that role.

`connection_id` is a host variable that contains a slash-delimited user name and password used to connect to the database.

Include the `AT` clause to specify the database to which the connection is established. `database_name` is the name of the database to which the client is connecting; specify the value in the form of a variable, or as a string literal.

Include the `USING` clause to specify a host variable that contains a null-terminated string identifying the database to which the connection will be established.

The `ALTER AUTHORIZATION` clause is supported for syntax compatibility only; ECPGPlus parses the `ALTER AUTHORIZATION` clause, and reports a warning.

Using the first form of the `CONNECT` statement, a client application might establish a connection with a host variable named `user` that contains the identity of the connecting role, and a host variable named `password` that contains the associated password using the following command:

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

A client application could also use the first form of the `CONNECT` statement to establish a connection using a single host variable named `:connection_id`. In the following example, `connection_id` contains the slash-delimited role name and associated password for the user:

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

The syntax of the second form of the `CONNECT` statement is:

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

Where `credentials` is one of the following:

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

In the second form:

`database_name` is the name or identity of the database to which the client is connecting. Specify `database_name` as a variable, or as a string literal, in one of the following forms:

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

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

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

Where:

`hostname` is the name or IP address of the server on which the database resides.

`port` is the port on which the server listens.

You can also specify a value of `DEFAULT` to establish a connection with the default database, using the default role name. If you specify `DEFAULT` as the target database, do not include a `connection_name` or `credentials`.

`connection_name` is the name of the connection to the database. `connection_name` should take the form of an identifier (that is, not a string literal or a variable). You can open multiple connections, by providing a unique `connection_name` for each connection.

If you do not specify a name for a connection, `ecpglib` assigns a name of `DEFAULT` to the connection. You can refer to the connection by name (`DEFAULT`) in any `EXEC SQL` statement.

`CURRENT` is the most recently opened or the connection mentioned in the most-recent `SET CONNECTION TO` statement. If you do not refer to a connection by name in an `EXEC SQL` statement, ECPG assumes the name of the connection to be `CURRENT`.

`user_name` is the role used to establish the connection with the Advanced Server database. The privileges of the specified role will be applied to all commands performed through the connection.

`password` is the password associated with the specified `user_name`.

The following code fragment uses the second form of the `CONNECT` statement to establish a connection to a database named `edb`, using the role `alice` and the password associated with that role, `1safepwd`:

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

The name of the connection is `acctg_conn`; you can use the connection name when changing the connection name using the `SET CONNECTION` statement.

## DEALLOCATE DESCRIPTOR

Use the `DEALLOCATE DESCRIPTOR` statement to free memory in use by an allocated descriptor. The syntax of the statement is:

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

Where:

`descriptor_name` is the name of the descriptor. This value may take the form of a quoted string literal, or of a host variable.

The following example deallocates a descriptor named `emp_query`:

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

## DECLARE CURSOR

Use the `DECLARE CURSOR` statement to define a cursor. The syntax of the statement is:

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

Where:

`database_name` is the name of the database on which the cursor operates. This value may take the form of an identifier or of a host variable. If you do not specify a database name, the default value of `database_name` is the default database.

`cursor_name` is the name of the cursor.

`select_statement` is the text of the `SELECT` statement that defines the cursor result set; the `SELECT` statement cannot contain an `INTO` clause.

`statement_name` is the name of a SQL statement or block that defines the cursor result set.

The following example declares a cursor named `employees`:

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

The cursor generates a result set that contains the employee number, employee name, salary and commission for each employee record that is stored in the `emp` table.

## DECLARE DATABASE

Use the `DECLARE DATABASE` statement to declare a database identifier for use in subsequent SQL statements (for example, in a `CONNECT` statement). The syntax is:

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

Where:

`database_name` specifies the name of the database.

The following example demonstrates declaring an identifier for the `acctg` database:

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

After invoking the command declaring `acctg` as a database identifier, the `acctg` database can be referenced by name when establishing a connection or in `AT` clauses.

This statement has no effect and is provided for Pro*C compatibility only.

## DECLARE STATEMENT

Use the `DECLARE STATEMENT` directive to declare an identifier for an SQL statement. Advanced Server supports two versions of the `DECLARE STATEMENT` directive:

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

and

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

Where:

`statement_name` specifies the identifier associated with the statement.

`database_name` specifies the name of the database. This value may take the form of an identifier or of a host variable that contains the identifier.

A typical usage sequence that includes the `DECLARE STATEMENT` directive might be:

```
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;
```

This statement has no effect and is provided for Pro*C compatibility only.

## DELETE

Use the `DELETE` statement to delete one or more rows from a table. The syntax for the ECPGPlus `DELETE` statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that an expression is allowed. The syntax is:

```
[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> ]
```

Where:

Include the `FOR exec_count` clause to specify the number of times the statement will

execute; this clause is valid only if the `VALUES` clause references an array or a pointer to an array.

`table` is the name (optionally schema-qualified) of an existing table. Include the `ONLY` clause to limit processing to the specified table; if you do not include the `ONLY` clause, any tables inheriting from the named table are also processed.

`alias` is a substitute name for the target table.

`using_list` is a list of table expressions, allowing columns from other tables to appear in the `WHERE` condition.

Include the `WHERE` clause to specify which rows should be deleted. If you do not include a `WHERE` clause in the statement, `DELETE` will delete all rows from the table, leaving the table definition intact.

`condition` is an expression, host variable or parameter marker that returns a value of type `BOOLEAN`. Those rows for which `condition` returns true will be deleted.

`cursor_name` is the name of the cursor to use in the `WHERE CURRENT OF` clause; the row to be deleted will be the one most recently fetched from this cursor. The cursor must be a non-grouping query on the `DELETE` statements target table. You cannot specify `WHERE CURRENT OF` in a `DELETE` statement that includes a Boolean condition.

The `RETURN/RETURNING` clause specifies an `output_expression` or `host_variable_list` that is returned by the `DELETE` command after each row is deleted:

- `output_expression` is an expression to be computed and returned by the `DELETE` command after each row is deleted. `output_name` is the name of the returned column; include * to return all columns.

- `host_variable_list` is a comma-separated list of host variables and optional indicator variables. Each host variable receives a corresponding value from the `RETURNING` clause.

For example, the following statement deletes all rows from the `emp` table where the `sal` column contains a value greater than the value specified in the host variable, `:max_sal:`

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

For more information about using the `DELETE` statement, see the PostgreSQL Core documentation available at:

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

## DESCRIBE

Use the `DESCRIBE` statement to find the number of input values required by a prepared statement or the number of output values returned by a prepared statement. The `DESCRIBE` statement is used to analyze a SQL statement whose shape is unknown at the time you write your application.

The `DESCRIBE` statement populates an `SQLDA` descriptor; to populate a SQL descriptor, use the `ALLOCATE DESCRIPTOR` and `DESCRIBE...DESCRIPTOR` statements.

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

or

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

Where:

`statement_name` is the identifier associated with a prepared SQL statement or PL/SQL block.

`descriptor` is the name of C variable of type `SQLDA*`. You must allocate the space for the descriptor by calling `sqlald()` (and initialize the descriptor) before executing the `DESCRIBE` statement.

When you execute the first form of the `DESCRIBE` statement, ECPG populates the given descriptor with a description of each input variable *required* by the statement. For example, given two descriptors:

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

You might prepare a query that returns information from the `emp` table:

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

The command requires one input variable (for the parameter marker (?)).

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

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

When you execute the second form, ECPG populates the given descriptor with a description of each value *returned* by the statement. For example, the following statement returns three values:

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

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

Before *executing* the statement, you must bind a variable for each input value and a variable for each output value. The variables that you bind for the input values specify the actual values used by the statement. The variables that you bind for the output values tell ECPGPlus where to put the values when you execute the statement.

This is alternate Pro*C compatible syntax for the `DESCRIBE DESCRIPTOR` statement.

## DESCRIBE DESCRIPTOR

Use the `DESCRIBE DESCRIPTOR` statement to retrieve information about a SQL statement, and store that information in a SQL descriptor. Before using `DESCRIBE DESCRIPTOR`, you must allocate the descriptor with the `ALLOCATE DESCRIPTOR` statement. The syntax is:

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

Where:

`statement_name` is the name of a prepared SQL statement.

`descriptor_name` is the name of the descriptor. `descriptor_name` can be a quoted string value or a host variable that contains the name of the descriptor.

If you include the `INPUT` clause, ECPGPlus populates the given descriptor with a description of each input variable *required* by the statement.

For example, given two descriptors:

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

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

You might prepare a query that returns information from the `emp` table:

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

The command requires one input variable (for the parameter marker (?)).

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

After describing the bind variables for this statement, you can examine the descriptor to find the number of variables required and the type of each variable.

If you do not specify the `INPUT` clause, `DESCRIBE DESCRIPTOR` populates the specified descriptor with the values returned by the statement.

If you include the `OUTPUT` clause, ECPGPlus populates the given descriptor with a description of each value *returned* by the statement.

For example, the following statement returns three values:

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

After describing the select list for this statement, you can examine the descriptor to find the number of returned values and the name and type of each value.

## DISCONNECT

Use the `DISCONNECT` statement to close the connection to the server. The syntax is:

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

Where:

`connection_name` is the connection name specified in the `CONNECT` statement used to establish the connection. If you do not specify a connection name, the current connection is closed.

Include the `CURRENT` keyword to specify that ECPGPlus should close the most-recently used connection.

Include the `DEFAULT` keyword to specify that ECPGPlus should close the connection named `DEFAULT`. If you do not specify a name when opening a connection, ECPGPlus assigns the name, `DEFAULT`, to the connection.

Include the `ALL` keyword to instruct ECPGPlus to close all active connections.

The following example creates a connection (named `hr_connection`) that connects to the `hr` database, and then disconnects from the connection:

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

## EXECUTE

Use the `EXECUTE` statement to execute a statement previously prepared using an `EXEC SQL PREPARE` statement. The syntax is:

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

Where:

`array_size` is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`statement_name` specifies the name assigned to the statement when the statement was created (using the `EXEC SQL PREPARE` statement).

Include the `USING` clause to supply values for parameters within the prepared statement:

- Include the `DESCRIPTOR` `SQLDA_descriptor` clause to provide an SQLDA descriptor value for a parameter.
- Use a `host_variable` (and an optional `indicator_variable`) to provide a user-specified value for a parameter.

The following example creates a prepared statement that inserts a record into the `emp` table:

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

Each time you invoke the prepared statement, provide fresh parameter values for the statement:

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

## EXECUTE DESCRIPTOR

Use the `EXECUTE` statement to execute a statement previously prepared by an `EXEC SQL PREPARE` statement, using an SQL descriptor. The syntax is:

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

Where:

`array_size` is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`statement_identifier` specifies the identifier assigned to the statement with the `EXEC SQL PREPARE` statement.

Include the `USING` clause to specify values for any input parameters required by the prepared statement.

Include the `INTO` clause to specify a descriptor into which the `EXECUTE` statement will write the results returned by the prepared statement.

`descriptor_name` specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

The following example executes the prepared statement, `give_raise`, using the values contained in the descriptor `stmtText:`

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

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

## EXECUTE...END EXEC

Use the `EXECUTE...END-EXEC` statement to embed an anonymous block into a client application. The syntax is:

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

Where:

`database_name` is the database identifier or a host variable that contains the database identifier. If you omit the `AT` clause, the statement will be executed on the current default database.

`anonymous_block` is an inline sequence of PL/pgSQL or SPL statements and declarations. You may include host variables and optional indicator variables within the block; each such variable is treated as an `IN/OUT` value.

The following example executes an anonymous block:

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

!!! Note The `EXECUTE...END EXEC` statement is supported only by Advanced Server.

## EXECUTE IMMEDIATE

Use the `EXECUTE IMMEDIATE` statement to execute a string that contains a SQL command. The syntax is:

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

Where:

`database_name` is the database identifier or a host variable that contains the database identifier. If you omit the `AT` clause, the statement will be executed on the current default database.

`command_text` is the command executed by the `EXECUTE IMMEDIATE` statement.

This dynamic SQL statement is useful when you don't know the text of an SQL statement (ie., when writing a client application). For example, a client application may prompt a (trusted) user for a statement to execute. After the user provides the text of the statement as a string value, the statement is then executed with an `EXECUTE IMMEDIATE` command.

The statement text may not contain references to host variables. If the statement may contain parameter markers or returns one or more values, you must use the `PREPARE` and `DESCRIBE` statements.

The following example executes the command contained in the `:command_text` host variable:

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

## FETCH

Use the `FETCH` statement to return rows from a cursor into an SQLDA descriptor or a target list of host variables. Before using a `FETCH` statement to retrieve information from a cursor, you must prepare the cursor using `DECLARE` and `OPEN` statements. The statement syntax is:

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

Where:

`array_size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`cursor` is the name of the cursor from which rows are being fetched, or a host variable that contains the name of the cursor.

If you include a `USING` clause, the `FETCH` statement will populate the specified SQLDA descriptor with the values returned by the server.

If you include an `INTO` clause, the `FETCH` statement will populate the host variables (and

optional indicator variables) specified in the `target_list`.

The following code fragment declares a cursor named `employees` that retrieves the `employee number`, `name` and `salary` from the `emp` table:

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

## FETCH DESCRIPTOR

Use the `FETCH DESCRIPTOR` statement to retrieve rows from a cursor into an SQL descriptor. The syntax is:

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

Where:

`array_size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`cursor` is the name of the cursor from which rows are fetched, or a host variable that contains the name of the cursor. The client must `DECLARE` and `OPEN` the cursor before calling the `FETCH DESCRIPTOR` statement.

Include the `INTO` clause to specify an SQL descriptor into which the `EXECUTE` statement will write the results returned by the prepared statement. `descriptor_name` specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor. Prior to use, the descriptor must be allocated using an `ALLOCATE DESCRIPTOR` statement.

The following example allocates a descriptor named `row_desc` that will hold the description and the values of a specific row in the result set. It then declares and opens a cursor for a prepared statement (`my_cursor`), before looping through the rows in result set, using a `FETCH` to retrieve the next row from the cursor into the descriptor:

```
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';
```

## GET DESCRIPTOR

Use the `GET DESCRIPTOR` statement to retrieve information from a descriptor. The `GET DESCRIPTOR` statement comes in two forms. The first form returns the number of values (or columns) in the descriptor.

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

The second form returns information about a specific value (specified by the `VALUE column_number` clause).

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

Where:

`array_size` is an integer value or a host variable that contains an integer value that specifies the number of rows to be processed. If you specify an `array_size`, the `host_variable` must be an array of that size; for example, if `array_size` is `10`, `:host_variable` must be a 10-member array of `host_variables`. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`descriptor_name` specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the `VALUE` clause to specify the information retrieved from the descriptor.

- `column_number` identifies the position of the variable within the descriptor.
- `host_variable` specifies the name of the host variable that will receive the value of the item.
- `descriptor_item` specifies the type of the retrieved descriptor item.

ECPGPlus implements the following `descriptor_item` types:

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

The following code fragment demonstrates using a `GET DESCRIPTOR` statement to obtain the number of columns entered in a user-provided string:

```
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;
```

The example allocates an SQL descriptor (named `parse_desc`), before using a `PREPARE` statement to syntax check the string provided by the user `(:stmt)`. A `DESCRIBE` statement moves the user-provided string into the descriptor, `parse_desc`. The call to `EXEC SQL GET DESCRIPTOR` interrogates the descriptor to discover the number of columns `(:col_count)` in the result set.

## INSERT

Use the `INSERT` statement to add one or more rows to a table. The syntax for the ECPGPlus `INSERT` statement is the same as the syntax for the SQL statement, but you can use parameter markers and host variables any place that a value is allowed. The syntax is:

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

Where:

Include the `FOR exec_count` clause to specify the number of times the statement will

execute; this clause is valid only if the `VALUES` clause references an array or a pointer to an array.

`table` specifies the (optionally schema-qualified) name of an existing table.

`column` is the name of a column in the table. The column name may be qualified with a subfield name or array subscript. Specify the `DEFAULT VALUES` clause to use default values for all columns.

`expression` is the expression, value, host variable or parameter marker that will be assigned to the corresponding column. Specify `DEFAULT` to fill the corresponding column with its default value.

`query` specifies a `SELECT` statement that supplies the row(s) to be inserted.

`output_expression` is an expression that will be computed and returned by the `INSERT` command after each row is inserted. The expression can refer to any column within the table. Specify * to return all columns of the inserted row(s).

`output_name` specifies a name to use for a returned column.

The following example adds a row to the `employees` table:

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

!!! Note The `INSERT` statement uses a host variable `(:ename)` to specify the value of the `ename` column.

For more information about using the `INSERT` statement, see the PostgreSQL Core documentation available at:

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

## OPEN

Use the `OPEN` statement to open a cursor. The syntax is:

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

Where `parameters` is one of the following:

```
DESCRIPTOR <SQLDA_descriptor>
```

or

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

Where:

`array_*size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`cursor` is the name of the cursor being opened.

`parameters` is either `DESCRIPTOR SQLDA_descriptor` or a comma-separated list of `host variables` (and optional `indicator variables`) that initialize the cursor. If specifying an `SQLDA_descriptor`, the descriptor must be initialized with a `DESCRIBE` statement.

The `OPEN` statement initializes a cursor using the values provided in `parameters`. Once initialized, the cursor result set will remain unchanged unless the cursor is closed and re-opened. A cursor is automatically closed when an application terminates.

The following example declares a cursor named `employees`, that queries the `emp` table, returning the `employee number`, `name`, `salary` and `commission` of an employee whose name matches a user-supplied value (stored in the host variable, `:emp_name`).

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

After declaring the cursor, the example uses an `OPEN` statement to make the contents of the cursor available to a client application.

## OPEN DESCRIPTOR

Use the `OPEN DESCRIPTOR` statement to open a cursor with a SQL descriptor. The syntax is:

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

Where:

`array_size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`cursor` is the name of the cursor being opened.

`descriptor_name` specifies the name of an SQL descriptor (in the form of a single-quoted string literal) or a host variable that contains the name of an SQL descriptor that contains the query that initializes the cursor.

For example, the following statement opens a cursor (named `emp_cursor`), using the host variable, `:employees`:

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

## PREPARE

Prepared statements are useful when a client application must perform a task multiple times; the statement is parsed, written and planned only once, rather than each time the statement is executed, saving repetitive processing time.

Use the `PREPARE` statement to prepare an SQL statement or PL/pgSQL block for execution. The statement is available in two forms; the first form is:

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

The second form is:

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

Where:

`database_name` is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the `AT` clause, the statement will execute against the current default database.

`statement_name` is the identifier associated with a prepared SQL statement or PL/SQL block.

`sql_statement` may take the form of a `SELECT` statement, a single-quoted string literal or host variable that contains the text of an SQL statement.

To include variables within a prepared statement, substitute placeholders (`$1, $2, $3`, etc.) for statement values that might change when you `PREPARE` the statement. When you `EXECUTE` the statement, provide a value for each parameter. The values must be provided in the order in which they will replace placeholders.

The following example creates a prepared statement (named `add_emp`) that inserts a record into the `emp` table:

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

Each time you invoke the statement, provide fresh parameter values for the statement:

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

!!! Note A client application must issue a `PREPARE` statement within each session in which a statement will be executed; prepared statements persist only for the duration of the current session.

## ROLLBACK

Use the `ROLLBACK` statement to abort the current transaction, and discard any updates made by the transaction. The syntax is:

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

Where:

`database_name` is the database identifier or a host variable that contains the database identifier against which the statement will execute. If you omit the `AT` clause, the statement will execute against the current default database.

Include the `TO` clause to abort any commands that were executed after the specified `savepoint`; use the `SAVEPOINT` statement to define the `savepoint`. If you omit the `TO` clause, the `ROLLBACK` statement will abort the transaction, discarding all updates.

Include the `RELEASE` clause to cause the application to execute an `EXEC SQL COMMIT RELEASE` and close the connection.

Use the following statement to rollback a complete transaction:

```
EXEC SQL ROLLBACK;
```

Invoking this statement will abort the transaction, undoing all changes, erasing any savepoints, and releasing all transaction locks. If you include a savepoint (`my_savepoint` in the following example):

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

Only the portion of the transaction that occurred after the `my_savepoint` is rolled back; `my_savepoint` is retained, but any savepoints created after `my_savepoint` will be erased.

Rolling back to a specified savepoint releases all locks acquired after the savepoint.

## SAVEPOINT

Use the `SAVEPOINT` statement to define a `savepoint`; a savepoint is a marker within a transaction. You can use a `ROLLBACK` statement to abort the current transaction, returning the state of the server to its condition prior to the specified savepoint. The syntax of a `SAVEPOINT` statement is:

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

Where:

`database_name` is the database identifier or a host variable that contains the database identifier against which the savepoint resides. If you omit the `AT` clause, the statement will execute against the current default database.

`savepoint_name` is the name of the savepoint. If you re-use a `savepoint_name`, the original savepoint is discarded.

Savepoints can only be established within a transaction block. A transaction block may contain multiple savepoints.

To create a savepoint named `my_savepoint`, include the statement:

```
EXEC SQL SAVEPOINT my_savepoint;
```

## SELECT

ECPGPlus extends support of the `SQL SELECT` statement by providing the `INTO host_variables` clause. The clause allows you to select specified information from an Advanced Server database into a host variable. The syntax for the `SELECT` statement is:

```
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 ]
[...]]
```

Where:

`database_name` is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

`host_variables` is a list of host variables that will be populated by the `SELECT` statement. If the `SELECT` statement returns more than a single row, `host_variables` must be an array.

ECPGPlus provides support for the additional clauses of the SQL `SELECT` statement as documented in the PostgreSQL Core documentation available at:

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

To use the `INTO host_variables` clause, include the names of defined host variables when specifying the `SELECT` statement. For example, the following `SELECT` statement populates the `:emp_name` and `:emp_sal` host variables with a list of `employee names` and `salaries`:

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

The enhanced `SELECT` statement also allows you to include parameter markers (question marks) in any clause where a value would be permitted. For example, the following query contains a parameter marker in the `WHERE` clause:

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

This `SELECT` statement allows you to provide a value at run-time for the `dept_no` parameter marker.

## SET CONNECTION

There are (at least) three reasons you may need more than one connection in a given client application:

- You may want different privileges for different statements;
- You may need to interact with multiple databases within the same client.
- Multiple threads of execution (within a client application) cannot share a connection concurrently.

The syntax for the `SET CONNECTION` statement is:

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

Where:

`connection_name` is the name of the connection to the database.

To use the `SET CONNECTION` statement, you should open the connection to the database using the second form of the `CONNECT` statement; include the AS clause to specify a `connection_name`.

By default, the current thread uses the current connection; use the `SET CONNECTION` statement to specify a default connection for the current thread to use. The default connection is only used when you execute an `EXEC SQL` statement that does not explicitly specify a connection name. For example, the following statement will use the default connection because it does not include an `AT connection_name` clause. :

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

This statement will not use the default connection because it specifies a connection name using the `AT connection_name` clause:

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

For example, a client application that creates and maintains multiple connections (such as):

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

and

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

Can change between the connections with the `SET CONNECTION` statement:

```
SET CONNECTION acctg_conn;
```

or

```
SET CONNECTION hr_conn;
```

The server will use the privileges associated with the connection when determining the privileges available to the connecting client. When using the `acctg_conn` connection, the client will have the privileges associated with the role, `alice`; when connected using `hr_conn`, the client will have the privileges associated with `bob`.

## SET DESCRIPTOR

Use the `SET DESCRIPTOR` statement to assign a value to a descriptor area using information provided by the client application in the form of a host variable or an integer value. The statement comes in two forms; the first form is:

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

The second form is:

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

Where:

`array_size` is an integer value or a host variable that contains an integer value specifying the number of rows to fetch. If you omit the `FOR` clause, the statement is executed once for each member of the array.

`descriptor_name` specifies the name of a descriptor (as a single-quoted string literal), or a host variable that contains the name of a descriptor.

Include the `VALUE` clause to describe the information stored in the descriptor.

- `column_number` identifies the position of the variable within the descriptor.
- `descriptor_item` specifies the type of the descriptor item.
- `host_variable` specifies the name of the host variable that contains the value of the item.

ECPGPlus implements the following `descriptor_item` types:

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

For example, a client application might prompt a user for a dynamically created query:

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

To execute a dynamically created query, you must first `prepare` the query (parsing and validating the syntax of the query), and then `describe` the `input` parameters found in the query using the `EXEC SQL DESCRIBE INPUT` statement.

```
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';
```

After describing the query, the `query_params` descriptor contains information about each parameter required by the query.

For this example, we'll assume that the user has entered:

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

In this case, the descriptor describes two parameters, one for `sal > ?` and one for `job = ?`.

To discover the number of parameter markers (question marks) in the query (and therefore, the number of values you must provide before executing the query), use:

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

Then, you can use `EXEC SQL GET DESCRIPTOR` to retrieve the name of each parameter. You can also use `EXEC SQL GET DESCRIPTOR` to retrieve the type of each parameter (along with the number of parameters) from the descriptor, or you can supply each `value` in the form of a character string and ECPG will convert that string into the required data type.

The data type of the first parameter is `numeric`; the type of the second parameter is `varchar`. The name of the first parameter is `sal`; the name of the second parameter is `job`.

Next, loop through each parameter, prompting the user for a value, and store those values in host variables. You can use `GET DESCRIPTOR … COUNT` to find the number of parameters in the query.

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

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

Use `GET DESCRIPTOR` to copy the name of the parameter into the `param_name` host variable:

```
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 */
```

To associate a `value` with each parameter, you use the `EXEC SQL SET DESCRIPTOR`

statement. For example:

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

Now, you can use the `EXEC SQL EXECUTE DESCRIPTOR` statement to execute the prepared statement on the server.

## UPDATE

Use an `UPDATE` statement to modify the data stored in a table. The syntax is:

```
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] [,
...] ]
```

Where:

`database_name` is the name of the database (or host variable that contains the name of the database) in which the table resides. This value may take the form of an unquoted string literal, or of a host variable.

Include the `FOR exec_count` clause to specify the number of times the statement will execute; this clause is valid only if the `SET` or `WHERE` clause contains an array.

ECPGPlus provides support for the additional clauses of the SQL `UPDATE` statement as documented in the PostgreSQL Core documentation available at:

https://www.postgresql.org/docs/current/static/sql-update.html

A host variable can be used in any clause that specifies a value. To use a host variable, simply substitute a defined variable for any value associated with any of the documented `UPDATE` clauses.

The following `UPDATE` statement changes the job description of an employee (identified by the `:ename` host variable) to the value contained in the `:new_job` host variable, and increases the employees salary, by multiplying the current salary by the value in the `:increase` host variable:

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

The enhanced `UPDATE` statement also allows you to include parameter markers (question marks) in any clause where an input value would be permitted. For example, we can write the same update statement with a parameter marker in the `WHERE` clause:

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

This `UPDATE` statement could allow you to prompt the user for a new value for the `job` column and provide the amount by which the `sal` column is incremented for the employee specified by `:ename`.

## WHENEVER

Use the `WHENEVER` statement to specify the action taken by a client application when it encounters an SQL error or warning. The syntax is:

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

The following table describes the different conditions that might trigger an `action`:

| Condition | Description |
|---|---|
| `NOT FOUND` | The server returns a `NOT FOUND` condition when it encounters a `SELECT` that returns no rows, or when a `FETCH` reaches the end of a result set. |
| `SQLERROR` | The server returns an `SQLERROR` condition when it encounters a serious error returned by an SQL statement. |
| `SQLWARNING` | The server returns an `SQLWARNING` condition when it encounters a non-fatal warning returned by an SQL statement. |

The following table describes the actions that result from a client encountering a `condition`:

| Action | Description |
|---|---|
| `CALL function [([args])]` | Instructs the client application to call the named `function`. |
| `CONTINUE` | Instructs the client application to proceed to the next statement. |
| `DO BREAK` | Instructs the client application to a C break statement. A break statement may appear in a `loop` or a `switch` statement. If executed, the break statement terminate the `loop` or the `switch` statement. |
| `DO CONTINUE` | Instructs the client application to emit a C `continue` statement. A `continue` statement may only exist within a loop, and if executed, will cause the flow of control to return to the top of the loop. |
| `DO function ([args])` | Instructs the client application to call the named `function`. |
| `GOTO label` or `GO TO label` | Instructs the client application to proceed to the statement that contains the `label`. |
| `SQLPRINT` | Instructs the client application to print a message to standard error. |
| `STOP` | Instructs the client application to stop execution. |

The following code fragment prints a message if the client application encounters a warning, and aborts the application if it encounters an error:

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

Include the following code to specify that a client should continue processing after warning a user of a problem:

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

Include the following code to call a function if a query returns no rows, or when a cursor reaches the end of a result set:

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