Introduction à Pro*C - Embedded SQL

par Vincent Rogier

Date de publication : 19 juin 2008

Dernière mise à jour :

I - Introduction

Embedded SQL is a method of combining the computing power of a high-level language like C/C++ and the database manipulation capabilities of SQL.
It allows you to execute any SQL statement from an application program.
Oracle's embedded SQL environment is called Pro*C.
A Pro*C program is compiled in two steps. First, the Pro*C precompiler recognizes the SQL statements embedded in the program, and replaces them with appropriate calls to the functions in the SQL runtime library.
The output is pure C/C++ code with all the pure C/C++ portions intact. Then, a regular C/C++ compiler is used to compile the code and produces the executable.
For details, see the section on Demo Programs.

II - Syntaxe Pro*C

II-A - SQL

All SQL statements need to start with EXEC SQL and end with a semicolon ";".
You can place the SQL statements anywhere within a C/C++ block, with the restriction that the declarative statements do not come after the executable statements.
As an example:

```
    {
        int a;
        /* ... */
        EXEC SQL SELECT salary INTO :a
                 FROM Employee
                 WHERE SSN=876543210;
        /* ... */
        printf("The salary is %d\n", a);
        /* ... */
    }
```

II-B - Directives du préprocesseur

The C/C++ preprocessor directives that work with Pro*C are #include and #if.
Pro*C does not recognize #define. For example, the following code is invalid:

```
    #define THE_SSN 876543210
    /* ... */
    EXEC SQL SELECT salary INTO :a
             FROM Employee
             WHERE SSN = THE_SSN;    /* INVALID */
```

II-C - Labels

You can connect C/C++ labels with SQL as in:

```
    EXEC SQL WHENEVER SQLERROR GOTO error_in_SQL;
    /* ... */
error_in_SQL:
    /* do error handling */
```

III - Variables hôte

III-A - Bases

Host variables are the key to the communication between the host program and the database.
A host variable expression must resolve to an lvalue (i.e., it can be assigned). You can declare host variables according to C syntax, as you declare regular C variables.
The host variable declarations can be placed wherever C variable declarations can be placed (C++ users need to use a declare section; see the section on C++ Users).
The C datatypes that can be used with Oracle include:

- char
- char[n]
- int
- short
- long
- float
- double
- VARCHAR[n]

VARCHAR[n] is a psuedo-type recognized by the Pro*C precompiler. It is used to represent blank-padded, variable-length strings. Pro*C precompiler will convert it into a structure with a 2-byte length field and a n-byte character array.
You cannot use register storage-class specifier for the host variables.
A host variable reference must be prefixed with a colon ":" in SQL statements, but should not be prefixed with a colon in C statements.
When specifying a string literal via a host variable, the single quotes must be omitted; Pro*C understands that you are specifying a string based on the declared type of the host variable.
C function calls and most of the pointer arithmetic expressions cannot be used as host variable references even though they may indeed resolve to lvalues.
The following code illustrates both legal and illegal host variable references:

```
int deptnos[3] = { 000, 111, 222 };
int get_deptno() { return deptnos[2]; }
int *get_deptnoptr() { return &(deptnos[2]); }
int main() {
    int x; char *y; int z;
    /* ... */
    EXEC SQL INSERT INTO emp(empno, ename, deptno)
        VALUES(:x, :y, :z);         /* LEGAL */
    EXEC SQL INSERT INTO emp(empno, ename, deptno)
        VALUES(:x + 1,              /* LEGAL: the reference is to x */
               'Big Shot',          /* LEGAL: but not really a host var */
               :deptnos[2]);        /* LEGAL: array element is fine */
    EXEC SQL INSERT INTO emp(empno, ename, deptno)
        VALUES(:x, :y,
               :(*(deptnos+2)));    /* ILLEGAL: although it has an
lvalue */
    EXEC SQL INSERT INTO emp(empno, ename, deptno)
        VALUES(:x, :y,
               :get_deptno());      /* ILLEGAL: no function calls */
    EXEC SQL INSERT INTO emp(empno, ename, deptno)
        VALUES(:x, :y,
               :(*get_depnoptr())); /* ILLEGAL: although it has an lvalue */
    /* ... */
}
```

III-B - Pointeurs

You can define pointers using the regular C syntax, and use them in embedded SQL statements. As usual, prefix them with a colon:

```
int *x;
/* ... */
EXEC SQL SELECT xyz INTO :x FROM ...;
```

The result of this SELECT statement will be written into *x, not x.

III-C - Structures

Structures can be used as host variables, as illustrated in the following example:

```
typedef struct {
    char name[21];    /* one greater than column length; for '\0' */
    int SSN;
} Emp;
/* ... */
Emp bigshot;
/* ... */
EXEC SQL INSERT INTO emp (ename, eSSN)
    VALUES (:bigshot);
```

III-D - Tableaux

Host arrays can be used in the following way:

```
int emp_number[50];
char name[50][11];
/* ... */
EXEC SQL INSERT INTO emp(emp_number, name)
    VALUES (:emp_number, :emp_name);
```

which will insert all the 50 tuples in one go.
Arrays can only be single dimensional. The example char name[50][11] would seem to contradict that rule.
However, Pro*C actually considers name a one-dimensional array of strings rather than a two-dimensional array of characters.
You can also have arrays of structures.
When using arrays to store the results of a query, if the size of the host array (say n) is smaller than the actual number of tuples returned by the query, then only the first n result tuples will be entered into the host array.

III-E - Indicateurs

Indicator variables are essentially "NULL flags" attached to host variables.
You can associate every host variable with an optional indicator variable.
An indicator variable must be defined as a 2-byte integer (using the type short) and, in SQL statements, must be prefixed by a colon and immediately follow its host variable.
Or, you may use the keyword INDICATOR in between the host variable and indicator variable.
Here is an example:

```
short indicator_var;
EXEC SQL SELECT xyz INTO :host_var:indicator_var
```

```
        FROM ...;
    /* ... */
    EXEC SQL INSERT INTO R
        VALUES(:host_var INDICATOR :indicator_var, ...);
```

You can use indicator variables in the INTO clause of a SELECT statement to detect NULL's or truncated values in the output host variables.
The values Oracle can assign to an indicator variable have the following meanings:

Valeur	Signification
-1	The column value is NULL, so the value of the host variable is indeterminate.
0	Oracle assigned an intact column value to the host variable.
<0	Oracle assigned a truncated column value to the host variable. The integer returned by the indicator variable is the original length of the column value.
-2	Oracle assigned a truncated column variable to the host variable, but the original column value could not be determined.

You can also use indicator variables in the VALUES and SET clause of an INSERT or UPDATE statement to assign NULL's to input host variables.
The values your program can assign to an indicator variable have the following meanings:

Valeur	Signification
-1	Oracle will assign a NULL to the column, ignoring the value of the host variable.
<=0	Oracle will assign the value of the host variable to the column.

III-F - Equivalences

Oracle recognizes two kinds of datatypes: internal and external.

- Internal datatypes specify how Oracle stores column values in database tables.
- External datatypes specify the formats used to store values in input and output host variables.

At precompile time, a default Oracle external datatype is assigned to each host variable.
Datatype equivalencing allows you to override this default equivalencing and lets you control the way Oracle interprets the input data and formats the output data.
The equivalencing can be done on a variable-by-variable basis using the VAR statement. The syntax is:

```
    EXEC SQL VAR <host_var> IS <type_name> [ (<length>) ];
```

For example, suppose you want to select employee names from the emp table, and then pass them to a routine that expects C-style '\0'-terminated strings.
You need not explicitly '\0'-terminate the names yourself.
Simply equivalence a host variable to the STRING external datatype, as follows:

```
    char emp_name[21];
    EXEC SQL VAR emp_name IS STRING(21);
```

The length of the ename column in the emp table is 20 characters, so you allot emp_name 21 characters to accommodate the '\0'-terminator.
STRING is an Oracle external datatype specifically designed to interface with C-style strings.
When you select a value from the ename column into emp_name, Oracle will automatically '\0'-terminate the value for you.
You can also equivalence user-defined datatypes to Oracle external datatypes using the TYPE statement.

The syntax is:

```
EXEC SQL TYPE <user_type> IS <type_name> [ (<length>) ] [REFERENCE];
```

You can declare a user-defined type to be a pointer, either explicitly, as a pointer to a scalar or structure, or implicitly as an array, and then use this type in a TYPE statement.
In these cases, you need to use the REFERENCE clause at the end of the statement, as shown below:

```
typedef unsigned char *my_raw;
EXEC SQL TYPE my_raw IS VARRAW(4000) REFERENCE;
my_raw buffer;
/* ... */
buffer = malloc(4004);
```

Here we allocated more memory than the type length (4000) because the precompiler also returns the length, and may add padding after the length in order to meet the alignment requirement on your system.

IV - SQL dynamique

While embedded SQL is fine for fixed applications, sometimes it is important for a program to dynamically create entire SQL statements.
With dynamic SQL, a statement stored in a string variable can be issued.
PREPARE turns a character string into a SQL statement, and EXECUTE executes that statement.
Consider the following example.

```
char *s = "INSERT INTO emp VALUES(1234, 'jon', 3)";
EXEC SQL PREPARE q FROM :s;
EXEC SQL EXECUTE q;
```

Alternatively, PREPARE and EXECUTE may be combined into one statement:

```
char *s = "INSERT INTO emp VALUES(1234, 'jon', 3)";
EXEC SQL EXECUTE IMMEDIATE :s;
```

V - Transactions

Oracle PRO*C supports transactions as defined by the SQL standard. A transaction is a sequence of SQL statements that Oracle treats as a single unit of work.
A transaction begins at your first SQL statement. A transaction ends when you issue "EXEC SQL COMMIT" (to make permanent any database changes during the current transaction) or "EXEC SQL ROLLBACK" (to undo any changes since the current transaction began).
After the current transaction ends with your COMMIT or ROLLBACK statement, the next executable SQL statement will automatically begin a new transaction.
If your program exits without calling EXEC SQL COMMIT, all database changes will be discarded.

VI - Gestion des erreurs

After each executable SQL statement, your program can find the status of execution either by explicit checking of SQLCA, or by implicit checking using the WHENEVER statement.
These two ways are covered in details below.

VI-A - SQLCA

SQLCA (SQL Communications Area) is used to detect errors and status changes in your program.
This structure contains components that are filled in by Oracle at runtime after every executable SQL statement.
To use SQLCA you need to include the header file sqlca.h using the #include directive.
In case you need to include sqlca.h at many places, you need to first undefine the macro SQLCA with #undef SQLCA.
The relevant chunk of sqlca.h follows:

```
#ifndef SQLCA
#define SQLCA 1

struct sqlca {
    /* ub1 */ char sqlcaid[8];
    /* b4 */ long sqlabc;
    /* b4 */ long sqlcode;
    struct {
        /* ub2 */ unsigned short sqlerrml;
        /* ub1 */ char sqlerrmc[70];
    } sqlerrm;
    /* ub1 */ char sqlerrp[8];
    /* b4 */ long sqlerrd[6];
    /* ub1 */ char sqlwarn[8];
    /* ub1 */ char sqlext[8];
};
/* ... */
```

The fields in sqlca have the following meaning:

Membre	Définition
sqlcaid	This string component is initialized to "SQLCA" to identify the SQL Communications Area.
sqlcabc	This integer component holds the length, in bytes, of the SQLCA structure.
sqlcode	This integer component holds the status code of the most recently executed SQL statement: 0 : No error <0 : Statement executed but exception detected. This occurs when Oracle cannot find a row that meets your WHERE condition or when a SELECT INTO or FETCH returns no rows. >0 : Oracle did not execute the statement because of an error. When such errors occur, the current transaction should, in most cases, be rolled back.
sqlerrm	This embedded structure contains the following two components: sqlerrml - Length of the message text stored in sqlerrmc sqlerrmc - Up to 70 characters of the message text corresponding to the error code stored in sqlcode.
sqlerrp	Reserved for future use.
sqlerrp	This array of binary integers has six elements: sqlerrd[0] - Future use sqlerrd[1] - Future use sqlerrd[2] - Numbers of rows processed by the most recent SQL statement sqlerrd[3] - Future use sqlerrd[4] - Future use sqlerrd[5] - Offset that specifies the character position at which a parse error begins in the most recent SQL statement.
sqlwarn	This array of single characters has eight elements used as warning flags. Oracle sets a flag by assigning to it the character 'W'. sqlwarn[0] - Set if any other flag is set; sqlwarn[1] - Set if a truncated column value was assigned to an output host variable. sqlwarn[2] - Set if a NULL column value is not used in computing a SQL aggregate such as AVG or SUM. sqlwarn[3] - Set if the number of columns in SELECT does not equal the number of host variables specified in INTO. sqlwarn[4] - Set if every row in a table was processed by an UPDATE or DELETE statement without a WHERE clause. sqlwarn[5] - Set if a procedure/function/package/package body creation command fails because of a PL/SQL compilation error. sqlwarn[6] - No longer in use. sqlwarn[7] - No longer in use.
sqlext	Reserved for future use.

SQLCA can only accommodate error messages up to 70 characters long in its sqlerrm component. To get the full text of longer (or nested) error messages, you need the sqlglm() function:

```
void sqlglm(char *msg_buf, size_t *buf_size, size_t *msg_length);
```

where msg_buf is the character buffer in which you want Oracle to store the error message; buf_size specifies the size of msg_buf in bytes; Oracle stores the actual length of the error message in *msg_length. The maximum length of an Oracle error message is 512 bytes.

VI-B - clause WHENEVER

This statement allows you to do automatic error checking and handling. The syntax is:

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

Oracle automatically checks SQLCA for [condition], and if such condition is detected, your program will automatically perform [action].
[condition] can be any of the following:

- SQLWARNING - sqlwarn[0] is set because Oracle returned a warning
- SQLERROR - sqlcode is negative because Oracle returned an error
- NOT FOUND - sqlcode is positive because Oracle could not find a row that meets your WHERE condition, or a SELECT INTO or FETCH returned no rows

[action] can be any of the following:

- CONTINUE - Program will try to continue to run with the next statement if possible
- DO - Program transfers control to an error handling function
- GOTO [label] - Program branches to a labeled statement
- STOP - Program exits with an exit() call, and uncommitted work is rolled back

Some examples of the WHENEVER statement:

```
    EXEC SQL WHENEVER SQLWARNING DO print_warning_msg();
    EXEC SQL WHENEVER NOT FOUND GOTO handle_empty;
```

Here is a more concrete example:

```
/* code to find student name given id */
/* ... */
for (;;) {
    printf("Give student id number : ");
    scanf("%d", &id);
    EXEC SQL WHENEVER NOT FOUND GOTO notfound;
    EXEC SQL SELECT studentname INTO :st_name
             FROM   student
             WHERE  studentid = :id;
    printf("Name of student is %s.\n", st_name);
    continue;
notfound:
    printf("No record exists for id %d!\n", id);
}
/* ... */
```

Note that the WHENEVER statement does not follow regular C scoping rules. Scoping is valid for the entire program. For example, if you have the following statement somewhere in your program (such as before a loop):

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

All SQL statements that occur after this line in the file would be affected. Make sure you use the following line to cancel the effect of WHENEVER when it is no longer needed (such as after your loop):

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
```

VI - Programme de démonstration

Note: The demo programs will create and use four tables named DEPT, EMP, PAY1, and PAY2. Be careful if any table in your database happens to have the same name!
Several demo programs are available in /afs/ir/class/cs145/code/proc on the leland system.
They are named sample*.pc (for C users) and cppdemo*.pc (for C++ users).
".pc" is the extension for Pro*C code. Do not copy these files manually, since there are a couple of customizations to do.
To download and customize the demo programs, follow the instructions below:

1 Make sure that you have run source /afs/ir/class/cs145/all.env
2 In your home directory, run load_samples [db_username> [db_passwd] [sample_dir], where [sample_dir] is the name of the directory where you wish to put demo programs (e.g., load_samples sally etaoinshrdlu cs145_samples)
3 cd [sample_dir]
4 Run make samples (or make cppsamples for C++) to compile all demo programs

Step (2) will set up the sample database, create a new directory as specified in [sample_dir], and copy the demo files into that directory. It will also change the user name and password in the sample programs to be yours, so that you do not have to type in your username and password every time when running a sample program. However, sample1 and cppdemo1 do provide an interface for the user to input the username and password, in case you would like to learn how to do it.
If you happen to make any mistake when entering username or password in Step (2), just run clean_samples [db_username] [db_passwd] [sample_dir] in your home directory, and then repeat Steps (2) to (4).
For Step (4), you can also compile each sample program separately. For example, make sample1 compiles sample1.pc alone.
The compilation process actually has two phases:

```
proc iname=sample1.pc
```

Converts the embedded SQL code to corresponding library calls and outputs sample1.c

```
cc [a_number_of_flags_here] sample1.c
```

Generates the executable sample1
To compile your own code, say, foo.pc, just change a few variables in Makefile: Add the program name foo to variable SAMPLES and the source file name foo.pc to variable SAMPLE_SRC.
Then, do make foo after foo.pc is ready.
foo.pc will be precompiled to foo.c and then compiled to foo, the executable.
C++ users will need to add their program name to CPPSAMPLES instead of SAMPLES, and source file name to CPPSAMPLE_SRC instead of SAMPLE_SRC.
The demo programs operate on the following tables:

```
CREATE TABLE DEPT
    (DEPTNO    NUMBER(2) NOT NULL,
     DNAME     VARCHAR2(14),
     LOC       VARCHAR2(13));

CREATE TABLE EMP
    (EMPNO     NUMBER(4) NOT NULL,
     ENAME     VARCHAR2(10),
     JOB       VARCHAR2(9),
     MGR       NUMBER(4),
     HIREDATE  DATE,
     SAL       NUMBER(7, 2),
     COMM      NUMBER(7, 2),
```

```
     DEPTNO    NUMBER(2));

CREATE TABLE PAY1
    (ENAME     VARCHAR2(10),
     SAL       NUMBER(7, 2));

CREATE TABLE PAY2
    (ENAME     VARCHAR2(10),
     SAL       NUMBER(7, 2));
```

These tables are created automatically when you run load_samples in Step (2). A few tuples are also inserted. You may like to browse the tables before running the samples on them. You can also play with them as you like (e.g., inserting, deleting, or updating tuples). These tables will be dropped automatically when you run clean_samples. Note:clean_samples also wipes out the entire [sample_dir]; make sure you move your own files to some other place before running this command!
You should take a look at the sample source code before running it. The comments at the top describe what the program does. For example, sample1 takes an employee's EMPNO and retrieve the name, salary, and commission for that employee from the table EMP.
You are supposed to study the sample source code and learn the following:

- How to connect to Oracle from the host program
- How to embed SQL in C/C++
- How to use cursors
- How to use host variables to communicate with the database
- How to use WHENEVER to take different actions on error messages.
- How to use indicator variables to detect NULL's in the output

Now, you can use these techniques to code your own database application program. And have fun!

VIII - PRO*C et C++

To get the precompiler to generate appropriate C++ code, you need to be aware of the following issues:

- Code emission by precompiler. To get C++ code, you need to set the option CODE=CPP while executing proc. C users need not worry about this option; the default caters to their needs.
- Parsing capability.

The PARSE option of proc may take the following values:

- PARSE=NONE. C preprocessor directives are understood only inside a declare section, and all host variables need to be declared inside a declare section.
- PARSE=PARTIAL. C preprocessor directives are understood; however, all host variables need to be declared inside a declare section.
- PARSE=FULL. C preprocessor directives are understood and host variables can be declared anywhere. This is the default when CODE is anything other than CPP; it is an error to specify PARSE=FULL with CODE=CPP.

So, C++ users must specify PARSE=NONE or PARSE=PARTIAL. They therefore lose the freedom to declare host variables anywhere in the code. Rather, the host variables must be encapsulated in declare sections as follows:

```
        EXEC SQL BEGIN DECLARE SECTION;
            // declarations...
        EXEC SQL END DECLARE SECTION;
```

You need to follow this routine for declaring the host and indicator variables at all the places you do so.

- File extension. You need to specify the option CPP_SUFFIX=cc or CPP_SUFFIX=C.
- Location of header files. By default, proc searches for header files like stdio.h in standard locations. However, C++ has its own header files, such as iostream.h, located elsewhere. So you need to use the SYS_INCLUDE option to specify the paths that proc should search for header files.

IX - Liste des commandes SQL supportées par Pro*C

Declarative Statements

Statement	Définition
EXEC SQL ARRAYLEN	To use host arrays with PL/SQL
EXEC SQL BEGIN DECLARE SECTION	To declare host variables
EXEC SQL END DECLARE SECTION	To declare host variables
EXEC SQL DECLARE	To name Oracle objects
EXEC SQL INCLUDE	To copy in files
EXEC SQL TYPE	To equivalence datatypes
EXEC SQL VAR	To equivalence variables
EXEC SQL WHENEVER	To handle runtime errors

Executable Statements To define and control Oracle data

Statement	Définition
EXEC SQL ALTER	
EXEC SQL ANALYZE	
EXEC SQL AUDIT	
EXEC SQL COMMENT	
EXEC SQL CONNECT	
EXEC SQL DROP	
EXEC SQL GRANT	
EXEC SQL NOAUDIT	
EXEC SQL RENAME	
EXEC SQL REVOKE	
EXEC SQL TRUNCATE	
EXEC SQL CLOSE	
EXEC SQL ALTER	

Executable Statements To query and manipulate Oracle data

Statement	Définition
EXEC SQL DELETE	
EXEC SQL EXPLAIN PLAN	
EXEC SQL FETCH	
EXEC SQL INSERT	
EXEC SQL LOCK TABLE	
EXEC SQL OPEN	
EXEC SQL SELECT	
EXEC SQL UPDATE	

Executable Statements To process transactions

Statement	Définition
EXEC SQL COMMIT	
EXEC SQL ROLLBACK	
EXEC SQL SAVEPOINT	
EXEC SQL SET TRANSACTION	

Executable Statements To use dynamic SQL

Statement	Définition
EXEC SQL DESCRIBE	
EXEC SQL EXECUTE	
EXEC SQL PREPARE	

Executable Statements To control sessions

Statement	Définition
EXEC SQL ALTER SESSION	
EXEC SQL SET ROLE	

Executable Statements To embed PL/SQL blocks

Statement	Définition
EXEC SQL EXECUTE	
END-EXEC	

X - Conclusion

XI - Ressources

- Database Systems: The Complete Book
- A First Course in Database Systems
- Gradiance SQL Tutorial.