

DB2 9 Application Development exam 733 prep, Part 4: Embedded SQL programming

Build applications that interact with DB2

Skill Level: Introductory

[Roger E. Sanders \(rsanders@netapp.com\)](mailto:rsanders@netapp.com)
Senior Manager - IBM Alliance Engineering
Network Appliance, Inc.

15 Feb 2007

This tutorial introduces you to embedded SQL programming and walks you through how to construct an embedded SQL application. This tutorial introduces the process for converting one or more high-level programming language source code files containing embedded SQL into an executable application. This is the fourth in a series of nine tutorials designed to help you prepare for the DB2 Application Developer Certification exam (Exam 733).

Section 1. Before you start

About this tutorial

This tutorial introduces you to embedded SQL programming and walks you through the basic steps of constructing an embedded SQL application. This tutorial also provides you an introduction to converting one or more high-level programming language source code files containing embedded SQL into an executable application. In this tutorial, you will learn:

- How SQL statements are embedded in a high-level programming language source code file

- The steps involved in developing an embedded SQL application
- What host variables are, how they are created, and how they are used
- What indicator variables are, how they are created, and when they are used
- How to analyze the contents of an SQLCA data structure variable
- How to establish a database connection from an embedded SQL application
- How to capture and process errors when they occur
- How to convert source code files containing embedded SQL into an executable application

This is the fourth in a series of nine tutorials designed to help you prepare for the DB2 Application Developer Certification exam (Exam 733). The material in this tutorial covers the objectives in Section 4 of the exam, entitled "Embedded SQL programming." You can view these objectives at <http://www.ibm.com/certify/tests/obj733.shtml>.

You do not need a copy of DB2 Universal Database to complete this tutorial. However, you can download a free trial version of [IBM DB2 Universal Database Enterprise Edition](#), and a free copy of DB2 Express-C 9 from the [DB2 Express-C downloads page](#).

Objectives

After completing this tutorial, you should be able to:

- Establish a database connection within an embedded SQL application
- Execute SQL statements within an embedded SQL application
- Analyze results and handle common errors encountered when SQL statements are executed in an embedded SQL application

Prerequisites

To understand some of the material presented in this tutorial, you should be familiar with the following terms:

- **Object:** Anything in a database that can be created or manipulated with SQL (for example tables, views, indexes, and packages).

- **Table:** A logical structure used to present data as a collection of unordered rows with a fixed number of columns. Each column contains a set of values, each value of the same data type (or a subtype of the column's data type); the definitions of the columns make up the table structure, and the rows contain the actual table data.
 - **DB2 optimizer:** A component of the SQL precompiler that chooses an access plan for a Data Manipulation Language (DML) SQL statement by modeling the execution cost of several alternative access plans and choosing the one with the lowest estimated cost.
-

Section 2. Introduction to embedded SQL programming

Structured Query Language and embedded SQL

Structured Query Language (SQL) is a standardized language used to manipulate database objects and the data they contain. SQL is comprised of several different statements that are used to define, alter, and destroy database objects, and to insert, modify, delete, and retrieve data values. But because SQL is nonprocedural in nature, SQL is not a general-purpose programming language. (SQL statements are executed by the DB2 Database Manager, not by the operating system.) Therefore, database applications are usually developed by combining the decision and sequence control of a high-level programming language with the data storage, manipulation, and retrieval capabilities of SQL. Several methods are available for merging SQL with a high-level programming language, but the simplest approach is to embed SQL statements directly into the high-level programming language source code files that are used to create an application. This technique is known as *embedded SQL programming*.

One of the drawbacks to developing applications using embedded SQL is that high-level programming language compilers do not recognize, and therefore cannot interpret, SQL statements embedded in a source code file. Because of this, source code files containing embedded SQL statements must be preprocessed (by a process known as *precompiling*) before they can be compiled and linked to produce an executable application. To facilitate this preprocessing, each SQL statement embedded in a high-level programming language source code file must be prefixed with the keywords `EXEC SQL` and terminated with either a semicolon (in C/C++) or the keyword `END-EXEC` (in COBOL). Preprocessing is performed by a special tool known as the *SQL precompiler*; when the SQL precompiler encounters the `EXEC`

SQL keywords, it replaces the text that follows (until it finds a semicolon (;) or the keyword `END-EXEC`) with a DB2-specific function call that forwards the SQL statement encountered to the DB2 Database Manager for processing.

Likewise, the DB2 Database Manager cannot work directly with high-level programming language variables. Instead, it must use special variables known as *host variables* to move data between an application and a database. (We will take a closer look at host variables in section below entitled "[Declaring host variables.](#)") Host variables look like any other high-level programming language variable; so, to be set apart, they must be defined in a special section known as a *declare section*. Also, in order for the SQL precompiler to distinguish host variables from other text in an SQL statement, all references to host variables must be preceded by a colon (:).

Static SQL

A *static SQL* statement is an SQL statement that can be hardcoded in an application program at development time because information about the structure and objects (i.e., tables, column, and data types) with which it is intended to interact with is known in advance. Since the details of a static SQL statement are known at development time, the work of analyzing the statement and selecting the optimum data access plan for executing the statement is performed by the DB2 optimizer as part of the development process. Because their operational form is stored in the database (as a *package*) and does not have to be generated at application runtime, static SQL statements execute quickly.

The downside to this approach is that all static SQL statements must be prepared (in other words, their access plans must be generated and stored in the database) before they can be executed. Furthermore, static SQL statements cannot be altered at runtime, and each application that uses static SQL must *bind* its operational packages to every database with which the application will interact. Additionally, because static SQL applications require prior knowledge of database objects, changes made to those objects after an application has been developed can produce undesirable results.

The following are examples of static SQL statements:

```
SELECT COUNT(*) FROM employee

UPDATE employee SET lastname = 'Jones' WHERE empid = '001'

SELECT MAX(salary), MIN(salary) INTO :MaxSalary, :MinSalary FROM employee
```

Generally, static SQL statements are well suited for high-performance applications that execute predefined operations against a known set of database objects.

Dynamic SQL

Although static SQL statements are relatively easy to incorporate into an application, their use is somewhat limited because their format must be known in advance.

Dynamic SQL statements, on the other hand, are much more flexible because they can be constructed at application runtime; information about a dynamic SQL statement's structure and the objects with which it plans to interact does not need to be known in advance. Furthermore, because dynamic SQL statements do not have a precoded, fixed format, the data objects they reference can change without affecting the statement (provided that objects referenced by the statement are not deleted).

Even though dynamic SQL statements are generally more flexible than static SQL statements, they are usually more complicated to incorporate into an application. And because the work of analyzing the statement to select the best access plan is performed at application runtime (again, by the DB2 optimizer), dynamic SQL statements can take longer to execute than their static SQL counterparts. (Because dynamic SQL statements can take advantage of the database statistics available at application runtime, there are some cases in which a dynamic SQL statement will execute faster than an equivalent static SQL statement, but those are the exception and not the norm.)

The following are examples of dynamic SQL statements:

```
SELECT COUNT(*) FROM ?
INSERT INTO EMPLOYEES VALUES (?, ?)
DELETE FROM DEPARTMENT WHERE DEPTID = ?
```

Generally, dynamic SQL statements are well suited for applications that interact with a rapidly changing database or that allow users to define and execute ad-hoc queries.

Section 3. Constructing an embedded SQL application

Declaring host variables

Earlier, we saw that the DB2 Database Manager relies on *host variables* to move

data between an application and a database. We also saw that host variables are defined in a special section known as a *declare section*, and that this is what distinguishes them from other high-level programming language variables. So just how are declare sections written?

The beginning of a declare section is defined by the SQL statement `BEGIN DECLARE SECTION`, while the end is defined by the statement `END DECLARE SECTION`. Thus, a typical declare section in a C/C++ source code file looks something like this:

```
...
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
char    EmployeeID[7];
char    WorkDept[4];
char    Job[9];
char    Sex[2];
double  Salary;
double  Bonus;
double  Commision;
EXEC SQL END DECLARE SECTION;
...
```

A declare section can be coded anywhere high-level programming language variable declarations can be coded in a source code file. And although a source code file typically contains only one declare section, multiple declare sections are allowed.

Host variables that transfer data to a database are known as *input host variables*, while host variables that receive data from a database are known as *output host variables*. Regardless of whether a host variable is used for input or output, its attributes must be appropriate for the context in which it is used. Thus, you must define host variables such that their data types and lengths are compatible with the data types and lengths of the database objects they are intended to work with. Also, each host variable used in an application must be assigned a unique name -- duplicate names in the same file are not allowed, even when multiple declare sections are used. A tool known as the *Declaration Generator* can be used to generate host variable declarations for the columns of a given table in a database. This tool creates embedded SQL declaration source code files, which can then be inserted into C/C++, Java language, COBOL, and FORTRAN applications. For more information about this utility, refer to the `db2dc1gn` command in the DB2 Command Reference.

So how are host variables used to move data between an application and a database? The easiest way to answer this question is by examining a simple embedded SQL source code fragment, such as the one in Listing 1.

Listing 1. Proper use of host variables

```
...
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
    char      EmployeeNo[7];
    char      LastName[16];
EXEC SQL END DECLARE SECTION;
...

// Retrieve A Record From The Database
EXEC SQL SELECT empno, lastname
    INTO :EmployeeNo, :LastName
    FROM employee
    WHERE empno = '000100';

// Do Something With The Results
...
```

In this example, when the SQL statement `SELECT empno, lastname FROM employee WHERE empno = '000100'` is executed, the results are transferred to the host variables `EmployeeNo` and `LastName`. The actual commands and functions used to make this transfer happen are added when the source code file is precompiled.

Declaring indicator variables

By default, columns in a DB2 database table can contain null values. And because null values are not stored the same way in which conventional data is stored, special provisions must be made if an application intends to work with null data. Null values cannot be retrieved and copied to host variables in the same manner that other data values can. Instead, a special flag must be examined to determine whether a specific value is meant to be null. And in order to obtain the value of this flag, a special host variable known as an *indicator variable* (or *null indicator variable*) must be associated with a host variable that has been assigned to a nullable column.

Because indicator variables must be accessible by both the DB2 Database Manager and the application program, they too must be defined inside a declare section. Furthermore, they must be assigned a data type that is compatible with the DB2 `SMALLINT` data type. Thus, the code used to define an indicator variable in a C/C++ source code file will look something like this:

```
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
    short  SalaryNullIndicator;
EXEC SQL END DECLARE SECTION;
```

An indicator variable is *associated* with a specific host variable when it follows the host variable in an SQL statement. And once associated with a host variable, an indicator variable can be examined as soon as its corresponding host variable has

been populated. If the indicator variable contains a negative value, that indicates that a null value was found and that the value of the corresponding host variable should thus be ignored. Otherwise, the value of the corresponding host variable is valid.

Again, in order to understand how indicator variables are used, it helps to look at a source code fragment. Listing 2, written in the C programming language, shows one example of how indicator variables are defined and used.

Listing 2. Proper use of indicator variables

```
...
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
    char      EmployeeNo[7];
    double    Salary;      // Salary - Used If SalaryNI Is Positive ( >= 0 )
    short     SalaryNI;    // Salary NULL Indicator - Used
                        // To Determine If Salary
                        // Value Should Be NULL
EXEC SQL END DECLARE SECTION;
...

// Declare A Static Cursor
EXEC SQL DECLARE cursor1 CURSOR FOR SELECT empno, DOUBLE(salary)
    FROM employee;

// Open The Cursor
EXEC SQL OPEN cursor1;

// If The Cursor Was Opened Successfully, Retrieve And
// Display All Records Available
while (sqlca.sqlcode == SQL_RC_OK)
{
    // Retrieve The Current Record From The Cursor
    EXEC SQL FETCH cursor1 INTO :EmployeeNo, :Salary :SalaryNI;

    // If The Salary Value For The Record Is NULL, ...
    if (SalaryNI < 0)
    {
        printf("No salary information is available for ");
        printf("employee %s\n", EmployeeNo);
    }
}

// Close The Open Cursor
EXEC SQL CLOSE C1;
...
```

Indicator variables can also be used to send null values to a database when an insert or update operation is performed. When processing `INSERT` and `UPDATE` SQL statements, the DB2 Database Manager examines the values of any indicator variables provided first; if one or more indicators contain a negative value, it assigns a null value to the appropriate column, provided the column is nullable. (If the indicator variable is set to zero or contains a positive number, or if no indicator variable is used, the DB2 Database Manager assigns the value stored in the corresponding host variable to the appropriate column instead.) Thus, the code used in a C/C++ source code file to assign a null value to a column in a table would look

something like this:

```
ValueInd = -1;
EXEC SQL INSERT INTO tabl VALUES (:Value :ValueInd);
```

The SQLCA data structure

So far, we've only looked at how host variables and indicator variables are used to move data between embedded SQL applications and database objects. However, there are times when an embedded SQL application needs to communicate with the DB2 Database Manager itself. Two special SQL data structures are used to establish this vital communication link: the *SQL Communications Area (SQLCA)* data structure and the *SQL Descriptor Area (SQLDA)* data structure.

The SQLCA data structure contains a collection of elements that are updated by the DB2 Database Manager every time an SQL statement or a DB2 administrative API function is executed. This data structure must exist before the DB2 Database Manager can populate it. Therefore, any application that contains embedded SQL or that calls one or more administrative APIs must define at least one SQLCA data structure variable. In fact, such an application will not compile successfully if an SQLCA data structure variable does not exist.

So just what does an SQLCA data structure look like? Table 1 lists the elements that make up an SQLCA data structure variable.

Table 1. SQLCA data structure elements

Element name	Data type	Description
sqlcaid	CHAR(8)	An eye catcher for storage dumps. To help visually identify the data structure, this element normally contains the value "SQLCA". If line number information is returned from parsing an SQL procedure body, the sixth byte contains the character L.
sqlcabc	INTEGER	The size, in bytes, of the SQLCA data structure itself. This element should always contain the value 136.

sqlcode	INTEGER	The SQL return code value. A value of 0 indicates successful execution, a positive value indicates successful execution with warnings, and a negative value indicates an error. Refer to the DB2 Message Reference, Volumes 1 and 2 product manuals (see Resources for a link) to obtain more information about specific SQL return code values.
sqlerrml	SMALLINT	The size, in bytes, of the data stored in the sqlerrmc element of this structure. This value can be any number between 0 and 70; a value of 0 indicates that no data has been stored in the sqlerrmc field.
sqlerrmc	CHAR(70)	One or more error message tokens, separated by the value "0xFF", that are to be substituted for variables in the descriptions of warning or error conditions. This element is also used when a successful connection is established.
sqlerrp	CHAR(8)	A diagnostic value that represents the type of DB2 server currently being used. This value begins with a three-letter code identifying the product version and release, and is followed by five digits that identify the modification level of the product. For

		example, SQL09010 means DB2 Version 9, release 1, modification level 0. If the <code>sqlcode</code> element contains a negative value, this element will contain an eight-character code that identifies the module that reported the error.
<code>sqlerrd</code>	INTEGER ARRAY	An array of six integer values that provide additional diagnostic information when an error occurs. (Refer to Table 2 for more information about the diagnostic information that can be returned in this element.)
<code>sqlwarn</code>	CHAR(11)	An array of character values that serve as warning indicators; each element of the array contains either a blank or the letter <code>w</code> . If compound SQL is used, this field will contain an accumulation of the warning indicators that were set for all substatements executed in the compound SQL statement block. (Refer to Table 3 for more information about the types of warning information that can be returned in this element.)
<code>sqlstate</code>	CHAR(5)	The SQLSTATE value that identifies the outcome of the most recently executed SQL statement. (We'll discuss these in more detail in the section below entitled SQLSTATES .)

Table 2 outlines the elements of the `sqlca.sqlerrd` array.

Table 2. sqlerrd array elements

Array element	Description
sqlerrd[0]	<p>If a connection has successfully been established, this element will contain the expected difference in length of mixed character data (CHAR data types) when it is converted from the application code page used to the database code page used. A value of 0 or 1 indicates that no expansion is anticipated; a positive value greater than 1 indicates a possible expansion in length; and a negative value indicates a possible reduction in length.</p> <p>If an SQL procedure has successfully been executed, this element will contain the return status of the procedure.</p>
sqlerrd[1]	<p>If a connection has successfully been established, this element will contain the expected difference in length of mixed character data (CHAR data types) when it is converted from the database code page used to the application code page used. A value of 0 or 1 indicates that no expansion is anticipated; a positive value greater than 1 indicates a possible expansion in length; and a negative value indicates a possible reduction in length.</p> <p>If the SQLCA data structure contains information for a NOT ATOMIC compound SQL statement, this element will contain the number of substatements that encountered one or more errors (if any).</p>
sqlerrd[2]	<p>If the SQLCA data structure contains information for a CONNECT SQL statement that</p>

executed successfully, this element will contain the value 1 if the connected database is updatable and the value 2 if the connected database is read-only.

If the SQLCA data structure contains information for a `PREPARE SQL` statement that executed successfully, this element will contain an estimate of the number of rows that will be returned in a result data set when the prepared statement is executed.

If the SQLCA data structure contains information for an `INSERT`, `UPDATE`, or `DELETE SQL` statement that executed successfully, this element will contain a count of the number of rows that were affected by the operation.

If the SQLCA data structure contains information for an `OPEN SQL` statement that executed successfully and the corresponding cursor used to perform insert, update, and delete operations, this element will contain a count of the number of rows that were affected by the operation.

If the SQLCA data structure contains information for a `CREATE PROCEDURE SQL` statement that encountered errors when parsing the SQL procedure body, this element will contain the line number where the parsing error was encountered.

If the SQLCA data structure contains information for a compound SQL statement, this element will contain a count of the number of rows that were affected by the substatements in the compound SQL statement block.

sqlerrd[3]

If the SQLCA data structure

	<p>contains information for a <code>CONNECT</code> SQL statement that executed successfully, this element will contain the value 0 if one-phase commit from a down-level client is being used, the value 1 if one-phase commit is being used, the value 2 if one-phase, read-only commit is being used, and the value 3 if two-phase commit is being used.</p> <p>If the SQLCA data structure contains information for a <code>PREPARE</code> SQL statement that executed successfully, this element will contain a relative cost estimate of the resources needed to prepare the statement specified.</p> <p>If the SQLCA data structure contains information for a compound SQL statement, this element will contain a count of the number of substatements in the compound SQL statement block that executed successfully.</p>
sqlerrrd[4]	<p>If the SQLCA data structure contains information for a <code>CONNECT</code> SQL statement that executed successfully, this element will contain the value 0 if server authentication is being used; the value 1 if client authentication is being used; the value 2 if authentication is being handled by DB2 Connect; the value 4 if <code>SERVER_ENCRYPT</code> authentication is being used; the value 5 if DB2 Connect with encryption authentication is being used; the value 7 if <code>KERBEROS</code> authentication is being used; the value 9 if <code>GSSPLUGIN</code> authentication is being used; the value 11 if <code>DATA_ENCRYPT</code> authentication is being used; and the value 255 if the way in which authentication is being handled cannot be determined.</p> <p>If the SQLCA data structure</p>

	<p>contains information for anything else, this element will contain a count of the total number of rows that were inserted, updated, or deleted as a result of the <code>DELETE</code> rule of one or more referential integrity constraints or the activation of one or more triggers.</p> <p>If the SQLCA data structure contains information for a compound SQL statement, this element will contain a count of all such rows for each substatement in the compound SQL statement block that executed successfully.</p>
<code>sqlerrd[5]</code>	<p>For partitioned databases, this element contains the partition number of the partition that encountered an error or warning. If no errors or warnings were encountered, this element will contain the partition number of the partition that serves as the coordinator node.</p>

And finally, Table 3 outlines the elements of the of the `sqlca.sqlwarn` array.

Table 3. sqlwarn array elements

Array element	Description
<code>sqlwarn[0]</code>	<p>This element is blank if all other elements in the array are blank; it contains the character <code>w</code> if one or more of the other elements available is not blank.</p>
<code>sqlwarn[1]</code>	<p>This element contains the character <code>w</code> if the value for a column with a character string data type was truncated when it was assigned to a host variable.</p> <p>This element contains the character <code>N</code> if the null-terminator for a character string was truncated.</p> <p>This element contains the character <code>A</code> if a <code>CONNECT</code> or <code>ATTACH</code> operation was successful and the authorization name for</p>

	<p>the connection is longer than 8 bytes.</p> <p>This element contains the character <code>P</code> if the <code>PREPARE</code> statement relative cost estimate stored in <code>sqlerrd[3]</code> exceeds the amount that can be stored in an <code>INTEGER</code> data type or was less than 1, and either the <code>CURRENT EXPLAIN MODE</code> or the <code>CURRENT EXPLAIN SNAPSHOT</code> special register is set to a value other than <code>NO</code>.</p>
<code>sqlwarn[2]</code>	<p>This element contains the character <code>w</code> if null values were eliminated from the arguments passed to a function.</p> <p>If the <code>SQLCA</code> data structure contains information for a <code>CONNECT SQL</code> statement that executed successfully, this element will contain the character <code>D</code> if the database is in a quiesced state, or the character <code>I</code> if the instance is in a quiesced state.</p>
<code>sqlwarn[3]</code>	<p>This element contains the character <code>w</code> if the number of values retrieved does not equal the number of host variables provided.</p> <p>This element contains the character <code>Z</code> if the number of result data set locators specified with an <code>ASSOCIATE LOCATORS SQL</code> statement is less than the actual number of result data sets returned by a called procedure.</p>
<code>sqlwarn[4]</code>	<p>This element contains the character <code>w</code> if an <code>UPDATE</code> or <code>DELETE SQL</code> statement that does not contain a <code>WHERE</code> clause was prepared.</p>
<code>sqlwarn[5]</code>	<p>This element contains the character <code>E</code> if an error was tolerated during <code>SQL</code> statement execution.</p>

sqlwarn[6]	This element contains the character <code>w</code> if the result of a date calculation was adjusted to avoid an invalid date value.
sqlwarn[7]	This element contains the character <code>E</code> if the SQLCA data structure contains information for a <code>CONNECT SQL</code> statement that executed successfully, and the <code>DYN_QUERY_MGMT</code> database configuration parameter is enabled.
sqlwarn[8]	This element contains the character <code>w</code> if a character that could not be converted was replaced with a substitution character.
sqlwarn[9]	This element contains the character <code>w</code> if one or more errors in an arithmetic expression were ignored during column function processing.
sqlwarn[10]	This element contains the character <code>w</code> if a conversion error occurred while converting a character data value in another element of the SQLCA data structure variable.

Both an SQLCA data structure variable and an SQLDA data structure variable (which we will look at next) can be created by embedding the appropriate form of the `INCLUDE SQL` statement (`INCLUDE SQLCA` and `INCLUDE SQLDA`, respectively) in an embedded SQL source code file.

The SQLDA data structure

The SQL Descriptor Area (SQLDA) data structure contains a collection of elements that are used to provide detailed information to `PREPARE`, `OPEN`, `FETCH`, and `EXECUTE SQL` statements. This data structure consists of a header followed by an array of structures, each of which describes a single host variable or a single column in a result data set. Table 4 lists the elements that make up an SQLDA data structure variable.

Table 4. SQLDA data structure elements

Element name	Data type	Description
sqldaid	CHAR(8)	An eye catcher for storage dumps. To

		<p>help visually identify the data structure, this element normally contains the value "SQLDA". The seventh byte of this field is a flag named SQLDOUBLED; the DB2 Database manager sets this byte to 2 if two sqlvar entries have been created for each column, or if any host variable being described is a structured type, or to BLOB, CLOB, or DBCLOB data type.</p>
sqldabc	INTEGER	<p>The size, in bytes, of the SQLDA data structure itself. The value assigned to this element is determined using the equation $sqldabc = 16 + (44 * sqln)$ (for 32-bit operating systems) or $sqldabc = 16 + (56 * sqln)$ (for 64-bit operating systems).</p>
sqln	SMALLINT	<p>The total number of elements in the sqlvar array.</p>
sqld	SMALLINT	<p>Either the number of columns in the result data set returned by a DESCRIBE or PREPARE SQL statement, or the number of host variables described by the elements in the sqlvar array.</p>
sqlvar	STRUCTURE ARRAY	<p>An array of data structures that contain information about host variables or columns in a result data set.</p>

In addition to this basic information, an SQLDA data structure variable contains an arbitrary number of occurrences of `sqlvar` data structures (also referred to as *SQLVAR variables*). The information stored in each SQLVAR variable is dependent upon the location where the corresponding SQLDA data structure variable is used: when used with a `PREPARE` or a `DESCRIBE` SQL statement, each SQLVAR variable will contain information about a column that will exist in the result data set produced when the prepared SQL statement is executed. (If any of the columns have a large object (LOB) or user-defined data type, the number of SQLVAR variables used will be doubled and the seventh byte of the character string value stored in the `sqldaid` element of the SQLDA data structure variable will be assigned the value 2.) On the other hand, when the SQLDA data structure variable is used with an `OPEN`, `FETCH`, or `EXECUTE` SQL statement, each SQLVAR variable will contain information about a host variable whose value is to be passed to the DB2 Database Manager during statement execution.

Two types of SQLVAR variables are used: *base SQLVARs* and *secondary SQLVARs*. Base SQLVARs contain basic information (such as data type code, length attribute, column name, host variable address, and indicator variable address) for result data set columns or host variables. The elements that make up a base SQLVAR data structure variable are shown in Table 5.

Table 5. SQLVAR data structure elements

Element name	Data type	Description
<code>sqltype</code>	SMALLINT	The data type of a host variable/parameter marker used, or the data type of a column in the result data set produced.
<code>sqllen</code>	SMALLINT	The length, in bytes, of a host variable used, or the size of a column in the result data set produced.
<code>sqldata</code>	Pointer	A pointer to a location in memory where the data for a host variable used is stored, or a pointer to a location in memory where data for a column in the result data set produced is to be stored.
<code>sqlind</code>	Pointer	A pointer to a location in memory where the data for the null

		indicator variable associated with a host variable used is stored, or a pointer to a location in memory where the data for the null indicator variable associated with a column in the result data set produced is to be stored.
sqlname	VARCHAR(30)	The unqualified name of a host variable or a column in the result data set produced.

Secondary SQLVARs, on the other hand, contain either the distinct data type name for distinct data types or the length attribute of the column or host variable and a pointer to the buffer that contains the actual length of the data for LOB data types. Secondary SQLVAR entries are only present if the number of SQLVAR entries is doubled because LOBs or distinct data types are used: if locators or file reference variables are used to represent LOB data types, secondary SQLVAR entries are not needed.

The information stored in an SQLDA data structure variable, along with the information stored in any corresponding SQLVAR variables, may be placed there manually (using the appropriate programming language statements), or it can be generated automatically by executing the DESCRIBE SQL statement.

Establishing a database connection

In order to perform any type of operation against a database, you must first establish a connection to that database. With embedded SQL applications, database connections are made (and in some cases are terminated) by executing the CONNECT SQL statement. (The RESET option of the CONNECT statement is used to terminate a connection.) During the connection process, information needed to establish a connection -- such as an authorization ID and a corresponding password of an authorized user -- is passed to the appropriate database for validation. Often, this information is collected at application runtime and forwarded to the CONNECT statement by way of one or more host variables. Thus, the code used to establish a database connection in a C/C++ source code file typically looks something like Listing 3.

Listing 3. Establishing a database connection

```
...
```

```
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
    char    DataSource[129] = {"SAMPLE"};
    char    UserID[129] = {"USER1"};
    char    Password[129] = {"User1PWD"};
EXEC SQL END DECLARE SECTION;

...

// Connect To The Appropriate Database
EXEC SQL CONNECT TO :DataSource USER :UserID USING :Password;
...

// Terminate The Database Connection
EXEC SQL CONNECT RESET;
...
```

Embedded SQL applications can use two types of connection semantics. These types, known simply as *Type 1* and *Type 2*, support two very different behaviors: Type 1 connections support only one database connection per transaction (referred to as a *remote unit of work*) while Type 2 connections support multiple database connections per transaction (referred to as an *application-directed distributed unit of work*). Essentially, when Type 1 connections are used, an application can only be connected to one database at a time. Once a connection to a database is established and a transaction is started, that transaction must be committed or rolled back before another database connection can be established. On the other hand, when Type 2 connections are used, an application can be connected to several different databases at the same time, and each database connection will have its own set of transactions.

The actual type of connection semantics used by an embedded SQL application is determined by the value assigned to the `CONNECT`, `SQLRULES`, `DISCONNECT`, and `SYNCPPOINT` SQL precompiler options when the application is precompiled.

Preparing and executing SQL statements

When static SQL statements are embedded in an application program, they are executed as they are encountered. However, when dynamic SQL statements are used, they can be processed in one of two ways:

- **Prepare and execute:** This approach separates the preparation of the SQL statement from its actual execution and is typically used when an SQL statement is to be executed repeatedly. This method is also used when an application needs advance information about the columns that will exist in the result data set produced when a `SELECT` SQL statement is executed. The SQL statements `PREPARE` and `EXECUTE` are used to process dynamic SQL statements in this manner.

- **Execute immediately:** This approach combines the preparation and the execution of an SQL statement into a single step and is typically used when an SQL statement is to be executed only once. This method is also used when the application does not need additional information about the result data set that will be produced, if any, when the SQL statement is executed. The SQL statement `EXECUTE IMMEDIATE` is used to process dynamic SQL statements in this manner.

Dynamic SQL statements that are prepared and executed (using either method) at runtime are not allowed to contain references to host variables. They can, however, contain parameter markers in place of constants and expressions. Parameter markers are represented by the question mark character (?), and they indicate where in the SQL statement the current value of one or more host variables (or elements of an SQLDA data structure variable) are to be substituted when the statement is executed. Therefore, parameter markers are typically used where a host variable would be referenced if the SQL statement being executed were static. Two types of parameter markers are available: *typed* and *untyped*.

A typed parameter marker is any parameter marker that is specified along with its target data type when it is referenced in an SQL statement. Typed parameter markers have this form:

```
CAST( ? AS DataType )
```

This notation does not imply that a function is called; rather, it promises that the data type of the value replacing the parameter marker at application runtime will be either the data type specified or a data type that can be converted to the data type specified. For example, consider the following SQL statement:

```
UPDATE employee SET lastname = CAST(? AS VARCHAR(12)) WHERE empno = '000050'
```

In this example, the value for the LASTNAME column is provided at application runtime, and the data type of that value will be either a `VARCHAR(12)` data type or a data type that can be converted to a `VARCHAR(12)` data type.

An untyped parameter marker, on the other hand, is specified without a target data type and has the form of a single question mark (?). The actual data type for an untyped parameter marker is determined by the context in which it is used. For example, in the following SQL statement, the value for the LASTNAME column is provided at application runtime, and it is assumed that the data type of that value will be compatible with the data type that has been assigned to the LASTNAME column of the EMPLOYEE table.

```
UPDATE employee SET lastname = ? WHERE empno = '000050'
```

When parameter markers are used in embedded SQL applications, values that are to be substituted for them must be provided, as additional parameter values, to the EXECUTE or the EXECUTE IMMEDIATE SQL statement when either is used to execute the SQL statement specified. Listing 4, written in the C programming language, illustrates how actual values might be provided for parameter markers that have been coded in a simple UPDATE SQL statement.

Listing 4. Using parameter markers

```
...
// Define The SQL Host Variables Needed
EXEC SQL BEGIN DECLARE SECTION;
    char    SQLStmt[80];
    char    JobType[10];
EXEC SQL END DECLARE SECTION;
...

// Define A Dynamic UPDATE SQL Statement That Uses A Parameter Marker
strcpy(SQLStmt, "UPDATE employee SET job = ? ");
strcat(SQLStmt, "WHERE job = 'DESIGNER'");

// Populate The Host Variable That Will Be Used In Place Of The Parameter Marker
strcpy(JobType, "MANAGER");

// Prepare The SQL Statement
EXEC SQL PREPARE SQL_STMT FROM :SQLStmt;

// Execute The SQL Statement
EXEC SQL EXECUTE SQL_STMT USING :JobType;
...
```

Retrieving and processing results

Regardless of whether static SQL statements or dynamic SQL statements are used in an embedded SQL application, once a statement has been executed, any results produced will need to be retrieved and processed. If the SQL statement was anything other than a SELECT or a VALUES statement, the only additional processing required after execution is a check of the SQLCA data structure variable to ensure that the statement executed as expected. However, if a query was executed, especially a query that returns multiple rows, additional steps may be required in order to retrieve data from the result data set produced.

When multiple rows are returned to an application by a query, DB2 can use a mechanism known as a *cursor* to retrieve values from the result data set produced. The name *cursor* probably originated from the blinking cursor found on early computer screens, and just as that cursor indicated the current position on the screen and identified where typed words would appear next, a DB2 cursor indicates

the current position in a result data set (i.e., the current row) and identifies the row of data that will be returned to the application next. The following steps must be performed, in the order shown, if a cursor is to be incorporated into an embedded SQL application:

1. Declare (define) a cursor along with its type (read-only or updatable), and associate it with the desired query. This is done by executing the `DECLARE CURSOR` statement.
2. Open the cursor. This will cause the corresponding query to be executed and a result data set to be produced. This is done by executing the `OPEN` statement.
3. Retrieve (fetch) each row stored in the result data set, one by one, until an end-of-data condition occurs. This is done by repeatedly executing the `FETCH` statement; host variables or an `SQLDA` data structure variable are used in conjunction with a `FETCH` statement to extract a row of data from a result data set. Each time a row is retrieved from the result data set, the cursor is automatically moved to the next row.
4. If appropriate, modify or delete the current row -- provided the cursor is an updatable cursor. This is done by executing the `UPDATE` statement or the `DELETE` statement.
5. Close the cursor. This action will cause the result data set that was produced when the corresponding query was executed to be deleted. This is done by executing the `CLOSE` statement.

Now that you have seen the steps that must be performed in order to use a cursor, you can take a look at how these steps are coded in an embedded SQL application. Listing 5, written in the C programming language, illustrates how a cursor would be used to retrieve the results of a simple query.

Listing 5. Retrieving results using a cursor

```
...
// Declare The SQL Host Memory Variables
EXEC SQL BEGIN DECLARE SECTION;
    char    EmployeeNo[7];
    char    LastName[16];
EXEC SQL END DECLARE SECTION;
...
```

```
// Declare A Cursor
EXEC SQL DECLARE cursor1 CURSOR FOR
    SELECT empno, lastname
    FROM employee
    WHERE job = 'DESIGNER';

// Open The Cursor
EXEC SQL OPEN cursor1;

// Fetch The Records
while (sqlca.sqlcode == SQL_RC_OK)
{
    // Retrieve A Record
    EXEC SQL FETCH cursor1
    INTO :EmployeeNo, :LastName;

    // Process The Information Retrieved
    if (sqlca.sqlcode == SQL_RC_OK)
        ...
}

// Close The Cursor
EXEC SQL CLOSE cursor1;
...
```

If you know in advance that only one row of data will be produced in response to a query, you can copy the contents of that row directly to host variables without using a cursor by executing either the `SELECT INTO` SQL statement or the `VALUES INTO` statement. Like the `SELECT` statement, the `SELECT INTO` statement can be used to construct complex queries. However, unlike the `SELECT` statement, the `SELECT INTO` statement requires a list of valid host variables to be supplied as part of its syntax, and cannot be used dynamically. Additionally, if the result data set produced when the `SELECT INTO` statement is executed contains more than one record, the operation will fail and an error will be generated. (If the result data set produced is empty, a `NOT FOUND` warning will be generated instead.)

Like the `SELECT INTO` statement, the `VALUES INTO` statement can be used to retrieve the data associated with a single record and copy it to one or more host variables. And, like the `SELECT INTO` statement, when the `VALUES INTO` statement is executed, all data retrieved is stored in a result data set. If this result data set contains only one record, the first value in that record is copied to the first host variable specified, the second value is copied to the second host variable specified, and so on. If the result data set produced contains more than one record, the operation will fail and an error will be produced. (Again, if the result data set produced is empty, a `NOT FOUND` warning will be generated.)

Managing transactions

A *transaction* (also known as a *unit of work*) is a sequence of one or more SQL operations grouped together as a single unit, usually within an application process. A given transaction can be comprised of any number of SQL operations, from a single operation to many hundreds or even thousands, depending upon what is considered

a single step within your business logic.

The initiation and termination of a single transaction defines points of consistency within a database: either the effects of all operations performed within a transaction are applied to the database and made permanent (committed), or the effects of all operations performed are backed out (rolled back) and the database is returned to the state it was in before the transaction was started. In most cases, transactions are initiated the first time an executable SQL statement is executed after a connection to a database has been established, or immediately after a pre-existing transaction has been terminated. Once initiated, transactions can be implicitly terminated using the *autocommit* feature or explicitly by executing either the `COMMIT` or the `ROLLBACK` SQL statement. If the autocommit feature is enabled, each executable SQL statement is treated as a single transaction; if the statement executes successfully, any changes made by the statement are applied to the database; if the statement fails, all changes are discarded.

Putting it all together

Now that we have examined some of the basic components that are used to construct embedded SQL applications, let's see how they all come together to produce an embedded SQL application that interacts with a DB2 database. A simple embedded SQL application, written in the C programming language, that obtains and prints employee identification numbers, last names, and salaries for all employees who have the job title "DESIGNER", using static SQL, would look something like Listing 6.

Listing 6. A simple embedded SQL application that uses static SQL

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

int main()
{
    // Include The SQLCA Data Structure Variable
    EXEC SQL INCLUDE SQLCA;

    // Define The SQL Host Variables Needed
    EXEC SQL BEGIN DECLARE SECTION;
        char    EmployeeNo[7];
        char    LastName[16];
        double  Salary;
        short   SalaryNI;
    EXEC SQL END DECLARE SECTION;

    // Connect To The Appropriate Database
    EXEC SQL CONNECT TO sample USER db2admin USING ibmdb2;

    // Declare A Static Cursor
    EXEC SQL DECLARE cursor1 CURSOR FOR
        SELECT empno,
```

```

        lastname,
        DOUBLE(salary)
    FROM employee
    WHERE JOB = 'DESIGNER';

    // Open The Cursor
    EXEC SQL OPEN cursor1;

    // If The Cursor Was Opened Successfully, Retrieve And
    // Display All Records Available
    while (sqlca.sqlcode == SQL_RC_OK)
    {
        // Retrieve The Current Record From The Cursor
        EXEC SQL FETCH cursor1
            INTO :EmployeeNo,
                :LastName,
                :Salary :SalaryNI;

        // Display The Record Retrieved
        if (sqlca.sqlcode == SQL_RC_OK)
        {
            printf("%-8s %-16s ", EmployeeNo,
                LastName);
            if (SalaryNI >= 0)
                printf("%lf\n", Salary);
            else
                printf("Unknown\n");
        }
    }

    // Close The Open Cursor
    EXEC SQL CLOSE cursor1;

    // Commit The Transaction
    EXEC SQL COMMIT;

    // Terminate The Database Connection
    EXEC SQL CONNECT RESET;

    // Return Control To The Operating System
    return(0);
}

```

On the other hand, a simple embedded SQL application written in the C programming language that uses dynamic SQL to change the job titles for all users with the title of "DESIGNER" to "MANAGER" might look something like Listing 7.

Listing 7. A simple embedded SQL application that uses dynamic SQL

```

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

int main()
{
    // Include The SQLCA Data Structure Variable
    EXEC SQL INCLUDE SQLCA;

    // Define The SQL Host Variables Needed
    EXEC SQL BEGIN DECLARE SECTION;
    char    DataSource[129] = {"SAMPLE"};
    char    UserID[129] = {"USER1"};

```

```
    char    Password[129] = {"User1PWD"};
    char    SQLStmt[80];
    char    JobType[10];
EXEC SQL END DECLARE SECTION;

// Connect To The Appropriate Database
EXEC SQL CONNECT TO :DataSource USER :UserID USING :Password;

// Define A Dynamic UPDATE SQL Statement That Uses A
// Parameter Marker
strcpy(SQLStmt, "UPDATE employee SET JOB = ? ");
strcat(SQLStmt, "WHERE job = 'DESIGNER'");

// Populate The Host Variable That Will Be Used In
// Place Of The Parameter Marker
strcpy(JobType, "MANAGER");

// Prepare The SQL Statement
EXEC SQL PREPARE SQL_STMT FROM :SQLStmt;

// Execute The SQL Statement
EXEC SQL EXECUTE SQL_STMT USING :JobType;

// Commit The Transaction
EXEC SQL COMMIT;

// Terminate The Database Connection
EXEC SQL DISCONNECT CURRENT;

// Return Control To The Operating System
return(0);
}
```

Section 4. Diagnostics and error handling

Using the WHENEVER statement

Earlier, we saw that the SQL Communications Area (SQLCA) data structure contains a collection of elements that are updated by the DB2 Database Manager each time an SQL statement is executed. One element of that structure, the `sqlcode` element, is assigned a value that indicates the success or failure of the SQL statement executed. (A value of 0 indicates successful execution, a positive value indicates successful execution with warnings, and a negative value indicates that an error occurred.) At a minimum, an embedded SQL application should always check the `sqlcode` value produced (often referred to as the *SQL return code*) immediately after an SQL statement is executed. If an SQL statement fails to execute as expected, users should be notified that an error or warning condition occurred; in addition, whenever possible they should be provided with diagnostic information sufficient to allow them to locate and correct the problem.

As you might imagine, checking the SQL return code after each statement is executed can add additional overhead to an application, especially one that contains a large number of SQL statements. However, since every SQL statement coded in an embedded SQL application must be processed by the SQL precompiler, it is possible to have the precompiler automatically generate the source code needed to check SQL return codes. This is accomplished by embedding one or more forms of the `WHENEVER` SQL statement in a source code file.

When used, the `WHENEVER` statement tells the SQL precompiler to generate source code that evaluates SQL return codes and branches to a specified label whenever an error, warning, or out-of-data condition occurs. (If the `WHENEVER` statement is not used, the default behavior is to ignore SQL return codes and continue processing as if no problems have been encountered.) Four forms of the `WHENEVER` statement are available, one for each of the three different types of error/warning conditions the `WHENEVER` statement can be used to check for, and one to turn error checking off:

- `WHENEVER SQLERROR GOTO [Label]`: Instructs the precompiler to generate source code that evaluates SQL return codes and branches to the label specified whenever a negative `sqlcode` value is generated.
- `WHENEVER SQLWARNING GOTO [Label]`: Instructs the precompiler to generate source code that evaluates SQL return codes and branches to the label specified whenever a positive `sqlcode` value (other than the value 100) is generated.
- `WHENEVER NOT FOUND GOTO [Label]`: Instructs the precompiler to generate source code that evaluates SQL return codes and branches to the label specified whenever an `sqlcode` value of 100 or an `sqlstate` value of 02000 is generated. (The value 100 is used to indicate that no records were found that matched the selection criteria specified or that the end of a result data set has been reached.)
- `WHENEVER [SQLERROR | SQL WARNING | NOT FOUND] CONTINUE:` Instructs the precompiler to ignore the SQL return code and continue with the next instruction in the application.

A source code file can contain any combination of these four forms of the `WHENEVER` statement, and the order in which the first three forms appear is insignificant. However, once any form of the `WHENEVER` statement is used, the SQL return codes of all subsequent SQL statements executed will be evaluated and processed accordingly until the application ends or until another `WHENEVER` statement alters this behavior.

Listing 8, written in the C programming language, illustrates how the `WHENEVER` statement can be used to trap and process out-of-data errors.

Listing 8. Handling errors with the `WHENEVER` statement

```
...
// Include The SQLCA Data Structure Variable
EXEC SQL INCLUDE SQLCA;

// Set Up Error Handler
EXEC SQL WHENEVER NOT FOUND GOTO NOT_FOUND_HANDLER;

// Connect To The Appropriate Database
EXEC SQL CONNECT TO sample USER db2admin USING ibmdb2;

// Execute A SELECT INTO SQL Statement (If A "DATA NOT FOUND" Situation Occurs,
// The Code Will Branch To The NOT_FOUND_HANDLER Label)
EXEC SQL SELECT empno INTO :EmployeeNo
      FROM rsanders.employee
      WHERE job = 'CODER';
...

// Disable All Error Handling
EXEC SQL WHENEVER NOT FOUND CONTINUE;

// Prepare To Return To The Operating System
goto EXIT;

// Define A Generic "Data Not Found" Handler
NOT_FOUND_HANDLER:
      printf("NOT FOUND: SQL Code = %d\n", sqlca.sqlcode);
      EXEC SQL ROLLBACK;
      goto EXIT;

EXIT:

// Terminate The Database Connection
EXEC SQL CONNECT RESET;

// Return Control To The Operating System
return(0);
```

Unfortunately, the code that is generated when the `WHENEVER SQL` statement is used relies on `GO TO` branching instead of call/return interfaces to transfer control to the appropriate error-handling section. As a result, when control is passed to the source code that is used to process errors and warnings, the application has no way of knowing where control came from, nor does it have any way of knowing where control should be returned to after the error or warning has been properly handled. For this reason, about the only thing an application can do when control is passed to a `WHENEVER` statement error-handling label is to display the error code generated, roll back the current transaction, and return control to the operating system.

The Get Error Message API

Among other things, most editions of DB2 contain a rich set of functions, referred to

as the *administrative APIs* (application programming interfaces), that are designed to provide services other than the data storage, manipulation, and retrieval functionality that SQL provides to DB2 applications. Essentially, any database operation that can be performed from the Command Line Processor by executing a DB2 command can be performed from within an application by calling the appropriate administrative API.

The value assigned to the `sqlcode` element of the SQLCA data structure variable each time an SQL statement is executed is actually a coded number. Furthermore, a special administrative API, called the *Get Error Message API*, can be used to translate this coded number into a meaningful description that can then be displayed to the user. The basic syntax used to call this API from a C/C++ high-level programming language source code file is as follows:

```
sqlaintp (char          *pBuffer,  
          short         sBufferSize,  
          short         sLineWidth,  
          struct sqlca  *pSQLCA);
```

And the syntax used to call this API from other high-level programming language source code files is:

```
sqlgintp (short         sBufferSize,  
          short         sLineWidth,  
          struct sqlca  *pSQLCA,  
          char          *pBuffer);
```

Let's take a closer look at the components of the syntax for this API:

- `pBuffer`: Identifies a location in memory where the Get Error Message API is to store any message text retrieved.
- `sBufferSize`: Identifies the size, in bytes, of the memory storage buffer to which any message text retrieved should be written.
- `sLineWidth`: Identifies the maximum number of characters that one line of message text should contain before a line break is inserted. A value of 0 indicates that the entire message text is to be returned without line breaks.
- `pSQLCA`: Identifies a location in memory where an SQL Communications Area (SQLCA) data structure variable is stored.

Each time the Get Error Message API is called, the value stored in the `sqlcode` element of the SQLCA data structure variable provided is used to locate and retrieve appropriate error message text from a message file that is packaged with DB2. Listing 9, written in the C programming language, illustrates how the Get Error Message API would typically be used to obtain and display the message text associated with any SQL return code generated.

Listing 9. Handling errors with the Get Error Message API

```
...
// Include The SQLCA Data Structure Variable
EXEC SQL INCLUDE SQLCA;

// Declare The Local Memory Variables
long  RetCode = SQL_RC_OK;
char  ErrorMsg[1024];
...

// Perform Some SQL Operation
...

// If An Error Occurred, Obtain And Display Any Diagnostic Information Available
if (sqlca.sqlcode != SQL_RC_OK)
{
    // Retrieve The Error Message Text For The Error Code Generated
    RetCode = sqlaintp(ErrorMsg, sizeof(ErrorMsg), 70, &sqlca);
    switch (RetCode)
    {
        case -1:
            printf("ERROR : Insufficient memory.\n");
            break;
        case -3:
            printf("ERROR : Message file is inaccessible.\n");
            break;
        case -5:
            printf("ERROR : Invalid SQLCA, bad buffer, ");
            printf("or bad buffer length specified.\n");
            break;
        default:
            printf("%s\n", ErrorMsg);
            break;
    }
}
...

```

As you can see in this example, when the Get Error Message API is called, it returns a value that indicates whether or not it executed successfully. In this example, the return code produced is checked; if an error occurred, a message that explains why the API failed is returned to the user. If the API was successful, the appropriate message text is retrieved and returned to the user instead.

SQLSTATEs

In addition to SQL return codes, DB2 (as well as other relational database products)

uses a set of error message codes known as *SQLSTATEs* to provide supplementary diagnostic information for warnings and errors. *SQLSTATEs* are alphanumeric strings that are five characters (bytes) in length and have the format *ccsss*, where *cc* indicates the error message class and *sss* indicates the error message subclass. Like SQL return code values, *SQLSTATE* values are written to an element (the *sqlstate* element) of the *SQLCA* data structure variable used each time an SQL statement is executed. And just as the Get Error Message API can be used to convert any SQL return code value into a meaningful description, another API -- the *Get SQLSTATE Message API* -- can be used to convert an *SQLSTATE* value into a meaningful description as well. By including either (or both) of these APIs in your embedded SQL applications, you can always return meaningful information to the end user whenever error and/or warning conditions occur.

Section 5. Creating executable applications

The basic process

So far, we have looked at some of the basic steps used to embed SQL statements in high-level programming language source code files, but we have only hinted at how source code files containing embedded SQL statements are converted into executable applications. Once a source code file has been written, the following steps must be performed, in the order shown, before an application that interacts with a DB2 database will be created:

1. All source code files containing embedded SQL statements must be precompiled. During the precompile process, a source code file containing embedded SQL statements is converted into a source code file that is made up entirely of high-level programming language statements -- the embedded SQL statements themselves are commented out and DB2-specific function calls are stored in their place.

At the same time, a corresponding package that contains (among other things) the access plans that are to be used to process each static SQL statement embedded in the source code file is also produced. (Access plans contain optimized information that the DB2 Database Manager uses to execute SQL statements; access plans for static SQL statements are produced at precompile time, while access plans for dynamic SQL statements are produced at application runtime.) Packages produced by the SQL precompiler can be stored in the database being used by the

precompiler as they are generated, or they can be written to an external bind file and bound to any valid DB2 database later (the process of storing this package in the appropriate database is known as *binding*). Unless otherwise specified, the SQL precompiler is also responsible for verifying that all database objects, such as tables and columns, that have been referenced in static SQL statements actually exist, and that all application data types used are compatible with their database counterparts. (That's why you need a database connection in order to use the SQL precompiler.)

2. Once a source code file containing embedded SQL statements has been processed by the SQL precompiler, the source code file produced -- along with any additional source code files needed -- must be compiled by a high-level programming language compiler. This compiler is responsible for converting source code files into object modules that the linker can use to create an executable program.
3. After all source code files needed to build an application have been successfully compiled, the resulting object modules must be linked with high-level programming language libraries and DB2 libraries to create an executable program. In most cases, the program produced exists as an executable application. However, it may also exist as a shared library or a dynamic link library (DLL) that can be loaded and executed by other executable applications.
4. If the packages for the files that were processed by the SQL precompiler have not already been bound to the appropriate database, they must be bound using the bind files produced during the precompile process. This is done using a tool known as the *DB2 Binder* (or simply the Binder).

Figure 1 illustrates the basic process used to convert an embedded SQL source code file to an executable application when deferred binding is used.

Figure 1. Converting an embedded SQL source code file to an executable application

Section 6. Summary

This tutorial was designed to introduce you to embedded SQL programming and to

walk you through the steps that are used to develop an embedded SQL application. Structured Query Language (SQL) is a standardized language used to manipulate database objects and the data they contain. Because SQL is nonprocedural in nature, it is not a general-purpose programming language. Therefore, database applications are usually developed by combining the decision and sequence control of a high-level programming language with the data storage, manipulation, and retrieval capabilities of SQL. Several methods are available for merging SQL with a high-level programming language, but the simplest approach is to embed SQL statements directly into the high-level programming language source code files that are used to create an application.

One of the drawbacks to developing applications using embedded SQL is that high-level programming language compilers do not recognize, and therefore cannot interpret, SQL statements embedded in a source code file. Because of this, source code files containing embedded SQL statements must be preprocessed (by a process known as *precompiling*) before they can be compiled and linked to produce an executable application. To facilitate this preprocessing, each SQL statement embedded in a high-level programming language source code file must be prefixed with the keywords `EXEC SQL` and terminated with either a semicolon (in C/C++) or the keyword `END-EXEC` (in COBOL). Preprocessing is performed by a special tool known as the *SQL precompiler*; when the SQL precompiler encounters the `EXEC SQL` keywords, it replaces the text that follows (until it encounters a semicolon (;) or the keyword `END-EXEC`) with a DB2-specific function call that forwards the SQL statement encountered to the DB2 Database Manager for processing.

Likewise, the DB2 Database Manager cannot work directly with high-level programming language variables. Instead, it must use special variables known as *host variables* to move data between an application and a database. Host variables look like any other high-level programming language variable; so, to be set apart, they must be defined in a special section known as a *declare section*. Also, in order for the SQL precompiler to distinguish host variables from other text in an SQL statement, all references to host variables must be preceded by a colon (:).

In order to perform any type of operation against a database, you must first establish a connection to that database. With embedded SQL applications, database connections are made (and in some cases are terminated) by executing the `CONNECT SQL` statement. During the connection process, information needed to establish a connection -- such as the authorization ID and a corresponding password of an authorized user -- is passed to the appropriate database for validation. Often, this information is collected at application runtime and forwarded to the `CONNECT` statement by way of one or more host variables.

Embedded SQL applications are comprised of static and dynamic SQL statements. Static SQL statements are well suited for high-performance applications that execute predefined operations against a known set of database objects. Dynamic SQL statements are well suited for applications that interact with a rapidly changing

database or that allow users to define and execute ad-hoc queries. When static SQL statements are embedded in an application program, they are executed as they are encountered. However, when dynamic SQL statements are used, they can be processed in one of two ways:

- **Prepare and execute:** This approach separates the preparation of the SQL statement from its actual execution and is typically used when an SQL statement is to be executed repeatedly. This method is also used when an application needs advance information about the columns that will exist in the result data set produced when a `SELECT` SQL statement is executed. The SQL statements `PREPARE` and `EXECUTE` are used to process dynamic SQL statements in this manner.
- **Execute immediately:** This approach combines the preparation and the execution of an SQL statement into a single step and is typically used when an SQL statement is to be executed only once. This method is also used when the application does not need additional information about the result data set that will be produced, if any, when the SQL statement is executed. The SQL statement `EXECUTE IMMEDIATE` is used to process dynamic SQL statements in this manner.

When multiple rows are returned to an application by a query, DB2 can use a mechanism known as a *cursor* to retrieve values from the result data set produced. A DB2 cursor indicates the current position in a result data set (i.e., the current row) and identifies the row of data that will be returned to the application next. The following steps must be performed, in the order shown, if a cursor is to be incorporated into an embedded SQL application:

1. Declare (define) a cursor along with its type (read-only or updatable), and associate it with the desired query.
2. Open the cursor. This will cause the corresponding query to be executed and a result data set to be produced.
3. Retrieve (fetch) each row stored in the result data set, one by one, until an end-of-data condition occurs.
4. Close the cursor. This action will cause the result data set that was produced when the corresponding query was executed to be deleted.

The SQL Communications Area (SQLCA) data structure contains a collection of elements that are updated by the DB2 Database Manager each time an SQL statement is executed. One element of that structure, the `sqlcode` element, is assigned a value that indicates the success or failure of the SQL statement executed. (A value of 0 indicates successful execution, a positive value indicates successful execution with warnings, and a negative value indicates that an error occurred.) At a minimum, an embedded SQL application should always check the `sqlcode` value produced (often referred to as the *SQL return code*) immediately after an SQL statement is executed. If an SQL statement fails to execute as expected, users should be notified that an error or warning condition occurred; whenever possible, they should be provided with diagnostic information sufficient to allow them to locate and correct the problem.

Once a source code file has been written, the following steps must be performed, in the order shown, before an application that interacts with a DB2 database will be created:

1. All source code files containing embedded SQL statements must be precompiled.
2. Once a source code file containing embedded SQL statements has been processed by the SQL precompiler, the source code file produced -- along with any additional source code files needed -- must be compiled by a high-level programming language compiler.
3. After all source code files needed to build an application have been successfully compiled, the resulting object modules must be linked with high-level programming language libraries and DB2 libraries to create an executable program.
4. If the packages for the files that were processed by the SQL precompiler have not already been bound to the appropriate database, they must be bound using the bind files produced during the precompile process. This is done using a tool known as the *DB2 Binder* (or simply the Binder).

Resources

Learn

- For more information on the DB2 9 Family Application Development Certification exam (Exam 733), see [IBM DB2 Information Management -- Training and certification](#) for information on classes, certifications available and additional resources.
- As mentioned earlier, this tutorial is just one in a [series of nine tutorials designed](#) to help you prepare for the DB2 9 Family Application Development certification exam (Exam 733). The complete list of all tutorials in this series is provided below:
 - Database objects and programming methods
 - Data manipulation
 - XML data manipulation
 - Embedded SQL programming
 - ODBC/CLI programming
 - .NET programming
 - Java programming
 - Advanced programming
 - User-defined routines
- Before you take the DB2 9 Application Development certification exam (Exam 733), you should have already taken and passed the DB2 9 Fundamentals certification exam (Exam 730). Use the [DB2 9 Fundamentals certification prep tutorial series](#) to prepare for that exam. A set of seven tutorials covers the following topics:
 - DB2 planning
 - DB2 security
 - Accessing DB2 data
 - Working with DB2 data
 - Working with DB2 objects
 - Data concurrency
 - Introducing Xquery

- Use the [DB2 V9 Database administration certification prep series](#) to prepare for the DB2 9 Database Administration for Linux, UNIX, and Windows certification exam (Exam 731). A set of seven tutorials covers the following topics:
 - Server management
 - Data placement
 - Database access
 - Monitoring DB2 activity
 - DB2 utilities
 - High availability: Backup and recovery
 - High availability: Split mirroring and HADR
- You can learn more about database objects and client and server application development from the [DB2 Information Center](#). In particular, look to these sections:
 - Administration guide: Implementation
 - SQL reference guide
 - Getting started with database application development
 - Developing embedded SQL applications
 - Developing Java applications
 - Developing Perl and PHP applications
 - Developing SQL and external routines
- The [DB2 Message Reference](#) can provide useful information about SQL return code values.
- Stay current with [developerWorks technical events and Webcasts](#).

Get products and technologies

- Build your next development project with [IBM trial software](#), available for download directly from developerWorks.

Discuss

- [Participate in the discussion forum for this content.](#)

About the author

Roger E. Sanders



Roger E. Sanders is a Senior Manager - IBM Alliance Engineering at Network Appliance, Inc. He has been designing and developing databases and database applications for more than 20 years and has been working with DB2 Universal Database since it was first introduced with OS/2 1.3 Extended Edition. He has written articles for IDUG Solutions Journal, Certification Magazine, and developerWorks, presented and taught classes at IDUG and RUG conferences, participated in the development of the DB2 certification exams, writes a regular column for DB2 Magazine and is the author of 9 books on DB2 UDB.