IBM i
Version 7.3

Database
Embedded SQL programming

IBM
Before using this information and the product it supports, read the information in "Notices" on page 191.

This edition applies to IBM i 7.3 (product number 5770-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

This document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
# Contents

**Embedded SQL programming** ........ 1

<p>| What's new for IBM i 7.3 ............ 1 |
| PDF file for Embedded SQL programming ........ 1 |
| Common concepts and rules for using embedded SQL ........ 2 |
| Writing applications that use SQL ........ 2 |
| Using host variables in SQL statements ........ 2 |
| Assignment rules for host variables in SQL statements ........ 4 |
| Indicator variables in applications that use SQL ........ 4 |
| Indicator variables used with host structures ........ 5 |
| Indicator variables used to assign special values ........ 5 |
| Handling SQL error return codes using the SQLCA ........ 7 |
| Using the SQL diagnostics area ........ 7 |
| Updating applications to use the SQL diagnostics area ........ 8 |
| IBM i programming model ........ 8 |
| Additional notes on using the SQL diagnostics area ........ 8 |
| Example: SQL routine exception ........ 9 |
| Example: Logging items from the SQL diagnostics area ........ 9 |
| Handling exception conditions with the WHENEVER statement ........ 10 |
| Coding SQL statements in C and C++ applications ........ 11 |
| Defining the SQL communication area in C and C++ applications that use SQL ........ 11 |
| Defining SQL descriptor areas in C and C++ applications that use SQL ........ 13 |
| Embedding SQL statements in C and C++ applications that use SQL ........ 15 |
| Comments in C and C++ applications that use SQL ........ 15 |
| Continuation for SQL statements in C and C++ applications that use SQL ........ 15 |
| Including code in C and C++ applications that use SQL ........ 16 |
| Margins in C and C++ applications that use SQL ........ 16 |
| Names in C and C++ applications that use SQL ........ 16 |
| NULLs and NULLs in C and C++ applications that use SQL ........ 16 |
| Statement labels in C and C++ applications that use SQL ........ 16 |
| Preprocessor sequence for C and C++ applications that use SQL ........ 16 |
| Trigraphs in C and C++ applications that use SQL ........ 16 |
| WHENEVER statement in C and C++ applications that use SQL ........ 17 |
| Using host variables in C and C++ applications that use SQL ........ 17 |
| Declaring host variables in C and C++ applications that use SQL ........ 17 |
| Numeric host variables in C and C++ applications that use SQL ........ 17 |
| Character host variables in C and C++ applications that use SQL ........ 18 |
| Graphic host variables in C and C++ applications that use SQL ........ 21 |
| Binary host variables in C and C++ applications that use SQL ........ 23 |
| LOB host variables in C and C++ applications that use SQL ........ 24 |
| XML host variables in C and C++ applications that use SQL ........ 27 |
| ROWID host variables in C and C++ applications that use SQL ........ 29 |
| Result set locator host variables in C and C++ applications that use SQL ........ 30 |
| Using host structures in C and C++ applications that use SQL ........ 30 |
| Host structure declarations in C and C++ applications that use SQL ........ 31 |
| Host structure indicator array in C and C++ applications that use SQL ........ 34 |
| Using arrays of host structures in C and C++ applications that use SQL ........ 34 |
| Host structure array in C and C++ applications that use SQL ........ 35 |
| Host structure array indicator structure in C and C++ applications that use SQL ........ 38 |
| Using pointer data types in C and C++ applications that use SQL ........ 38 |
| Using typedef in C and C++ applications that use SQL ........ 39 |
| Using ILE C compiler external file descriptions in C and C++ applications that use SQL ........ 40 |
| Determining equivalent SQL and C or C++ data types ........ 40 |
| Notes on C and C++ variable declaration and usage ........ 43 |
| Using indicator variables in C and C++ applications that use SQL ........ 43 |
| Coding SQL statements in COBOL applications ........ 44 |
| Defining the SQL communication area in COBOL applications that use SQL ........ 44 |
| Defining SQL descriptor areas in COBOL applications that use SQL ........ 45 |
| Embedding SQL statements in COBOL applications that use SQL ........ 47 |
| Comments in COBOL applications that use SQL ........ 47 |
| Continuation for SQL statements in COBOL applications that use SQL ........ 47 |
| Including code in COBOL applications that use SQL ........ 48 |
| Margins in COBOL applications that use SQL ........ 48 |</p>
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using indicator variables in REXX applications that use SQL</td>
<td>136</td>
</tr>
<tr>
<td>Preparing and running a program with SQL statements</td>
<td>136</td>
</tr>
<tr>
<td>Basic processes of the SQL precompiler</td>
<td>136</td>
</tr>
<tr>
<td>Input to the SQL precompiler</td>
<td>137</td>
</tr>
<tr>
<td>Source file CCSIDs in the SQL precompiler</td>
<td>138</td>
</tr>
<tr>
<td>Output from the SQL precompiler</td>
<td>138</td>
</tr>
<tr>
<td>Listing</td>
<td>138</td>
</tr>
<tr>
<td>Temporary source file members created by the SQL precompiler</td>
<td>138</td>
</tr>
<tr>
<td>Sample SQL precompiler output</td>
<td>139</td>
</tr>
<tr>
<td>Non-ILE SQL precompiler commands</td>
<td>143</td>
</tr>
<tr>
<td>Compiling a non-ILE application program that uses SQL</td>
<td>144</td>
</tr>
<tr>
<td>ILE SQL precompiler commands</td>
<td>144</td>
</tr>
<tr>
<td>Compiling an ILE application program that uses SQL</td>
<td>144</td>
</tr>
<tr>
<td>Setting compiler options using the precompiler commands</td>
<td>145</td>
</tr>
<tr>
<td>Interpreting compile errors in applications that use SQL</td>
<td>146</td>
</tr>
<tr>
<td>Binding an application that uses SQL</td>
<td>146</td>
</tr>
<tr>
<td>Program references in applications that use SQL</td>
<td>147</td>
</tr>
<tr>
<td>Displaying SQL precompiler options</td>
<td>148</td>
</tr>
<tr>
<td>Running a program with embedded SQL</td>
<td>148</td>
</tr>
<tr>
<td>Running a program with embedded SQL:</td>
<td></td>
</tr>
<tr>
<td>DDM considerations</td>
<td>148</td>
</tr>
<tr>
<td>Override considerations</td>
<td>148</td>
</tr>
<tr>
<td>Running a program with embedded SQL:</td>
<td></td>
</tr>
<tr>
<td>SQL return codes</td>
<td>149</td>
</tr>
<tr>
<td>Example programs: Using Db2 for i statements</td>
<td>149</td>
</tr>
<tr>
<td>Example: SQL statements in ILE C and C++ programs</td>
<td>151</td>
</tr>
<tr>
<td>Example: SQL statements in COBOL and ILE</td>
<td></td>
</tr>
<tr>
<td>COBOL programs</td>
<td>157</td>
</tr>
<tr>
<td>Example: SQL statements in PL/I programs</td>
<td>166</td>
</tr>
<tr>
<td>Example: SQL statements in RPG/400 programs</td>
<td>171</td>
</tr>
<tr>
<td>Example: SQL statements in ILE RPG programs</td>
<td>177</td>
</tr>
<tr>
<td>Example: SQL statements in REXX programs</td>
<td>183</td>
</tr>
<tr>
<td>Report produced by example programs that use SQL</td>
<td></td>
</tr>
<tr>
<td>CL command descriptions for host language precompilers</td>
<td>188</td>
</tr>
<tr>
<td>Create SQL COBOL Program command</td>
<td>188</td>
</tr>
<tr>
<td>Create SQL ILE COBOL Object command</td>
<td>189</td>
</tr>
<tr>
<td>Create SQL ILE C Object command</td>
<td>189</td>
</tr>
<tr>
<td>Create SQL ILE C++ Object command</td>
<td>189</td>
</tr>
<tr>
<td>Create SQL PL/I Program command</td>
<td>189</td>
</tr>
<tr>
<td>Create SQL RPG Program command</td>
<td>189</td>
</tr>
<tr>
<td>Create SQL ILE RPG Object command</td>
<td>189</td>
</tr>
<tr>
<td>Related information for Embedded SQL programming</td>
<td>189</td>
</tr>
<tr>
<td>Notices</td>
<td>191</td>
</tr>
<tr>
<td>Programming interface information</td>
<td>193</td>
</tr>
<tr>
<td>Trademarks</td>
<td>193</td>
</tr>
<tr>
<td>Terms and conditions</td>
<td>193</td>
</tr>
</tbody>
</table>
Embedded SQL programming

This topic collection explains how to create database applications in host languages that use Db2® for i SQL statements and functions.

Note: By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 190.

What's new for IBM i 7.3
Read about new or significantly changed information for the Embedded SQL programming topic collection.

- The ILE RPG precompiler supports RPG free form H, F, D, and P specs.
  - "Determining equivalent SQL and ILE RPG data types" on page 116
- The ILE COBOL and ILE RPG precompilers support a timestamp data type that allows fractional seconds values from 0 to 12.

What's new as of April 2019
The ILE RPG precompiler allows constants to be used in SQL.

What's new as of August 2018
- The precompiler includes detailed error information in the SQL0312 and SQL5011 precompile messages. For more information, see "Interpreting compile errors in applications that use SQL" on page 146.

What's new as of March 2017
- The precompiler COMPILEOPT parameter recognizes EVENTF as an option. For more information, see "Setting compiler options using the precompiler commands" on page 146.
- The ILE RPG precompiler generates more free-form declarations and code.

How to see what's new or changed
To help you see where technical changes have been made, this information uses:
- The ➜ image to mark where new or changed information begins.
- The ◄ image to mark where new or changed information ends.

In PDF files, you might see revision bars (!) in the left margin of new and changed information.

To find other information about what's new or changed this release, see the Memo to users.

PDF file for Embedded SQL programming
You can view and print a PDF file of this information.

To view or download the PDF version of this document, select Embedded SQL programming.

Saving PDF files
To save a PDF on your workstation for viewing or printing:
1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click Save.

**Downloading Adobe Reader**

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (http://get.adobe.com/reader/).

**Related reference:**
“Related information for Embedded SQL programming” on page 189

Product manuals and other information center topic collections contain information that relates to the Embedded SQL programming topic collection. You can view or print any of the PDF files.

---

**Common concepts and rules for using embedded SQL**

Here are some common concepts and rules for using SQL statements in a host language.

**Writing applications that use SQL**

You can create database applications in host languages that use Db2 for i SQL statements and functions. To use embedded SQL, you must have the licensed program IBM® DB2® Query Manager and SQL Development Kit for i installed. Additionally, you must have the compilers for the host languages you want to use installed.

**Related concepts:**

- “Coding SQL statements in C and C++ applications” on page 11
- “Coding SQL statements in COBOL applications” on page 44
- “Coding SQL statements in PL/I applications” on page 73
- “Coding SQL statements in RPG/400 applications” on page 88
- “Coding SQL statements in ILE RPG applications” on page 98

The RPG/400® licensed program supports both RPG II and RPG III programs.

**Using host variables in SQL statements**

When your program retrieves data, the values are put into data items that are defined by your program and that are specified with the INTO clause of a SELECT INTO or FETCH statement. The data items are called host variables.
A *host variable* is a field in your program that is specified in an SQL statement, usually as the source or target for the value of a column. The host variable and column must have compatible data types. Host variables cannot be used to identify SQL objects, such as tables or views, except in the DESCRIBE TABLE statement.

A *host structure* is a group of host variables used as the source or target for a set of selected values (for example, the set of values for the columns of a row). A *host structure array* is an array of host structures that is used in the multiple-row FETCH and blocked INSERT statements.

**Note:** By using a host variable instead of a literal value in an SQL statement, you give the application program the flexibility to process different rows in a table or view.

For example, instead of coding an actual department number in a WHERE clause, you can use a host variable set to the department number you are currently interested in.

Host variables are commonly used in SQL statements in these ways:

- **In a WHERE clause:** You can use a host variable to specify a value in the predicate of a search condition, or to replace a literal value in an expression. For example, if you have defined a field called EMPID that contains an employee number, you can retrieve the name of the employee whose number is 000110 with:

  ```sql
  MOVE '000110' TO EMPID.
  EXEC SQL
  SELECT LASTNAME
  INTO :PGM-LASTNAME
  FROM CORPDATA.EMPLOYEE
  WHERE EMPNO = :EMPID
  END-EXEC.
  ```

- **As a receiving area for column values (named in an INTO clause):** You can use a host variable to specify a program data area that is to contain the column values of a retrieved row. The INTO clause names one or more host variables that you want to contain column values returned by SQL. For example, suppose you are retrieving the EMPNO, LASTNAME, and WORKDEPT column values from rows in the CORPDATA.EMPLOYEE table. You could define a host variable in your program to hold each column, then name the host variables with an INTO clause. For example:

  ```sql
  EXEC SQL
  SELECT EMPNO, LASTNAME, WORKDEPT
  INTO :CBLEMPNO, :CBLNAME, :CBLDEPT
  FROM CORPDATA.EMPLOYEE
  WHERE EMPNO = :EMPID
  END-EXEC.
  ```

  In this example, the host variable CBLEMPNO receives the value from EMPNO, CBLNAME receives the value from LASTNAME, and CBLDEPT receives the value from WORKDEPT.

- **As a value in a SELECT clause:** When specifying a list of items in the SELECT clause, you are not restricted to the column names of tables and views. Your program can return a set of column values intermixed with host variable values and literal constants. For example:

  ```sql
  MOVE '000220' TO PERSON.
  EXEC SQL
  SELECT "A", LASTNAME, SALARY, :RAISE,
       SALARY + :RAISE
  INTO :PROCESS, :PERSON-NAME, :EMP-SAL,
       :EMP-RAISE, :EMP-TTL
  FROM CORPDATA.EMPLOYEE
  WHERE EMPNO = :PERSON
  END-EXEC.
  ```

  The results are:

<table>
<thead>
<tr>
<th>PROCESS</th>
<th>PERSON-NAME</th>
<th>EMP-SAL</th>
<th>EMP-RAISE</th>
<th>EMP-TTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>LUTZ</td>
<td>29840</td>
<td>4476</td>
<td>34316</td>
</tr>
</tbody>
</table>
• As a value in other clauses of an SQL statement:
  – The SET clause in an UPDATE statement
  – The VALUES clause in an INSERT statement
  – The CALL statement

Related concepts:
DB2 for i5/OS SQL reference

Assignment rules for host variables in SQL statements
SQL values are assigned to host variables during the running of FETCH, SELECT INTO, SET, and VALUES INTO statements. SQL values are assigned from host variables during the running of INSERT, UPDATE, and CALL statements.

All assignment operations observe the following rules:
• Numbers and strings are compatible:
  – Numbers can be assigned to character or graphic string columns or host variables.
  – Character and graphic strings can be assigned to numeric columns or numeric host variables.
• All character and DBCS graphic strings are compatible with UCS-2 and UTF-16 graphic columns if conversion is supported between the CCSIDs. All graphic strings are compatible if the CCSIDs are compatible. All numeric values are compatible. Conversions are performed by SQL whenever necessary. All character and DBCS graphic strings are compatible with UCS-2 and UTF-16 graphic columns for assignment operations, if conversion is supported between the CCSIDs. For the CALL statement, character and DBCS graphic parameters are compatible with UCS-2 and UTF-16 parameters if conversion is supported.
• Binary strings are only compatible with binary strings.
• A null value cannot be assigned to a host variable that does not have an associated indicator variable.
• Different types of date/time values are not compatible. Dates are only compatible with dates or string representations of dates; times are only compatible with times or string representations of times; and timestamps are only compatible with timestamps or string representations of timestamps.

Related concepts:
i5/OS globalization

Related reference:
DECLARE VARIABLE
Numeric assignments
String assignments
Datetime assignments

Indicator variables in applications that use SQL
An indicator variable is a halfword integer variable used to communicate additional information about its associated host variable.
• If the value for the result column is null, SQL puts a -1 in the indicator variable.
• If you do not use an indicator variable and the result column is a null value, a negative SQLCODE is returned.
• If the value for the result column causes a data mapping error, SQL sets the indicator variable to -2.

You can also use an indicator variable to verify that a retrieved string value has not been truncated. If truncation occurs, the indicator variable contains a positive integer that specifies the original length of the string. If the string represents a large object (LOB), and the original length of the string is greater than 32 767, the value that is stored in the indicator variable is 32 767, because no larger value can be stored in a halfword integer.
Always test the indicator variable first. If the value of the indicator variable is less than zero, you know the value of the result column should not be used. When the database manager returns a null value, the host variable might or might not be set to the default value for the result column's data type (0 for numeric, blanks for fixed length character, etc).

You specify an indicator variable (preceded by a colon) immediately after the host variable. For example:

```sql
EXEC SQL
    SELECT COUNT(*), AVG(SALARY)
    INTO :PLICNT, :PLISAL:INDNULL
    FROM CORPDATA.EMPLOYEE
    WHERE EDLEVEL < 18
END-EXEC.
```

You can then test INDNULL in your program to see if it contains a negative value. If it does, you know SQL returned a null value (if its value is -1) or a data mapping error (if its value is -2). If the indicator value is not negative, the value returned in PLISAL can be used.

**Related reference:**

- Predicates

**Indicator variables used with host structures:**

You can specify an *indicator array* (defined as an array of halfword integer variables) to support a host structure.

If the results column values returned to a host structure can be null, you can add an indicator array name after the host structure name. This allows SQL to notify your program about each null value returned to a host variable in the host structure.

For example, in COBOL:

```cobol
01 SAL-REC.
   10 MIN-SAL PIC S9(6)V99 USAGE COMP-3.
   10 AVG-SAL PIC S9(6)V99 USAGE COMP-3.
   10 MAX-SAL PIC S9(6)V99 USAGE COMP-3.
01 SALTABLE.
02 SALIND PIC S9999 USAGE COMP-4 OCCURS 3 TIMES.
01 EDUC-LEVEL PIC S9999 COMP-4.
...
MOVE 20 TO EDUC-LEVEL.
...
EXEC SQL
    SELECT MIN(SALARY), AVG(SALARY), MAX(SALARY)
    INTO :SAL-REC:SALIND
    FROM CORPDATA.EMPLOYEE
    WHERE EDLEVEL>:EDUC-LEVEL
END-EXEC.
```

In this example, SALIND is an array that contains three values, each of which can be tested for a negative value. SQL selects the values for the result row and puts them into the host structure. If MIN-SAL is to return a null value, the corresponding indicator variable, SALIND(1), is set to -1. Your program must check the corresponding indicator variables first to determine which, if any, selected result variables contain the null value.

**Indicator variables used to assign special values:**

You can use an indicator variable to set a null value for a column in an INSERT or UPDATE statement.

There are two forms of indicators for INSERT and UPDATE statements: normal indicators and extended indicators. When you use normal indicators, an indicator set to any negative value is interpreted as the
null value. When you use extended indicators, the negative values have several different meanings. Both forms of indicators can be used for inserts and updates that are part of a MERGE statement as well.

When processing update and insert using normal indicators, SQL checks the indicator variable (if it exists). If it contains a negative value, the column value is set to null. If it contains a value greater than -1, the column is set from the associated host variable value.

For example, you can specify that a value be updated in a column, but you know that an actual value is not always known. To provide the capability to set a column to a null value, you can write the following statement:

```sql
EXEC SQL
UPDATE CORPDATA.EMPLOYEE
SET PHONENO = :NEWPHONE:PHONEIND
   WHERE EMPNO = :EMPID
END-EXEC.
```

When NEWPHONE contains a non-null value, set PHONEIND to zero; otherwise, to tell SQL that NEWPHONE contains a null value, set PHONEIND to a negative value.

Using extended indicators provides your application with more flexibility when writing INSERT and UPDATE statements. In addition to providing the null value, you can set an indicator to indicate that the default value for a column is used or that the corresponding column is not updated at all.

For extended indicators, the indicator values are interpreted as follows:

- An indicator value of 0 means the value for the host variable is assigned to the column.
- An indicator value of -1, -2, -3, -4, or -6 means the null value is assigned to the column.
- An indicator value of -5 means the default value for the column is assigned.
- An indicator value of -7 means that the column is not assigned. This value causes the column to be treated as though it were not listed in the insert or update column list. For an INSERT statement it means the default value is used.

To write an UPDATE statement that can conditionally update several different fields, write it as follows:

```sql
EXEC SQL
UPDATE CORPDATA.EMPLOYEE
SET PHONENO = :NEWPHONE:PHONEIND,
    LASTNAME = :LASTNAME:LASTNAMEIND,
    WORKDEPT = :WORKDEPT:WORKDEPTIND,
    EDLEVEL = :EDLEVEL:EDLEVELIND
   WHERE EMPNO = :EMPID
END-EXEC.
```

With this one UPDATE statement, you can update any or all of the columns listed in the SET clause. For example, if you only want to update the EDLEVEL column, set the EDLEVEL variable to the new value and the EDLEVELIND indicator to 0. Set the other three indicators (PHONEIND, LASTNAMEIND, and WORKDEPTIND) to -7. This causes the statement to be processed as though you had written it this way.

```sql
EXEC SQL
UPDATE CORPDATA.EMPLOYEE
SET EDLEVEL = :EDLEVEL:EDLEVELIND
   WHERE EMPNO = :EMPID
END-EXEC.
```

You can use extended indicators only if they are explicitly enabled for your program. To specify that your program supports extended indicators, use *EXTIND on the OPTION parameter of the precompiler command or EXTIND(*YES) on the SET OPTION statement.
Handling SQL error return codes using the SQLCA

When an SQL statement is processed in your program, SQL places a return code in the SQLCODE and SQLSTATE fields. The return codes indicate the success or failure of the running of your statement.

If SQL encounters an error while processing the statement, the SQLCODE is a negative number and SUBSTR(SQLSTATE,1,2) is not '00', '01', or '02'. If SQL encounters an exception but valid condition while processing your statement, the SQLCODE is a positive number and SUBSTR(SQLSTATE,1,2) is '01' or '02'. If your SQL statement is processed without encountering an error or warning condition, the SQLCODE is zero and the SQLSTATE is '00000'.

**Note:** There are situations when a zero SQLCODE is returned to your program and the result might not be satisfactory. For example, if a value was truncated as a result of running your program, the SQLCODE returned to your program is zero. However, one of the SQL warning flags (SQLWARN1) indicates truncation. In this case, the SQLSTATE is not '00000'.

**Attention:** If you do not test for negative SQLCODEs or specify a WHENEVER SQLERROR statement, your program will continue to the next statement. Continuing to run after an error can produce unpredictable results.

The main purpose for SQLSTATE is to provide common return codes for common return conditions among the different IBM relational database systems. SQLSTATEs are particularly useful when handling problems with distributed database operations.

Because the SQLCA is a valuable problem-diagnosis tool, it is a good idea to include in your application programs the instructions necessary to display some of the information contained in the SQLCA. Especially important are the following SQLCA fields:

- **SQLCODE**
  - Return code.
- **SQLSTATE**
  - Return code.
- **SQLERRD(3)**
  - The number of rows updated, inserted, or deleted by SQL.
- **SQLWARN0**
  - If set to W, at least one of the SQL warning flags (SQLWARN1 through SQLWARNA) is set.

**Related concepts:**
- DB2 for i5/OS SQL reference
- SQL messages and codes

**Using the SQL diagnostics area**

The SQL diagnostics area is used to keep the returned information for an SQL statement that has been run in a program. It contains all the information that is available to you as an application programmer through the SQLCA.

There are additional values available to provide more detailed information about your SQL statement including connection information. More than one condition can be returned from a single SQL statement. The information in the SQL diagnostics area is available for the previous SQL statement until the next SQL statement is run.

To access the information from the diagnostics area, use the GET DIAGNOSTICS statement. In this statement, you can request multiple pieces of information at one time about the previously run SQL
statement. Each item is returned in a host variable. You can also request to get a string that contains all the diagnostic information that is available. Running the GET DIAGNOSTICS statement does not clear the diagnostics area.

Related reference:
GET DIAGNOSTICS

Updating applications to use the SQL diagnostics area
You might consider changing your applications to use the SQL diagnostics area instead of the SQL communication area (SQLCA), because the SQL diagnostics area provides some significant advantages over the SQLCA.

One of the best reasons is that the SQLERRM field in the SQLCA is only 70 bytes in length. This is often insufficient for returning meaningful error information to the calling application. Additional reasons for considering the SQL diagnostics area are multiple row operations, and long column and object names. Reporting even simple warnings is sometimes difficult within the restrictions of the 136 byte SQLCA. Quite often, the returned tokens are truncated to fit the restrictions of the SQLCA.

Current applications include the SQLCA definition by using the following:
EXEC SQL INCLUDE SQLCA; /* Existing SQLCA */

With the conversion to using the SQL diagnostics area, the application would first declare a stand-alone SQLSTATE variable:
char SQLSTATE[6]; /* Stand-alone sqlstate */

And possibly a stand-alone SQLCODE variable:
long int SQLCODE; /* Stand-alone sqlcode */

The completion status of the SQL statement is verified by checking the stand-alone SQLSTATE variable. If upon the completion of the current SQL statement, the application chooses to retrieve diagnostics, the application would run the SQL GET DIAGNOSTICS statement:
char hv1[256];
long int hv2;
EXEC SQL GET DIAGNOSTICS :hv1 = COMMAND_FUNCTION,
:hv2 = COMMAND_FUNCTION_CODE;

IBM i programming model
In the IBM i Integrated Language Environment® (ILE), the SQL diagnostics area is scoped to a thread and an activation group. This means that for each activation group in which a thread runs SQL statements, a separate diagnostics area exists for the activation.

Additional notes on using the SQL diagnostics area
In an application program, the SQLCA is replaced with an implicit or a stand-alone SQLSTATE variable, which must be declared in the program.

With multiple condition areas existing in the SQL diagnostics area, the most severe error or warning is returned in the first diagnostics area. There is no specific ordering of the multiple conditions, except that the first diagnostics area will contain the information for the SQLSTATE that is also returned in the SQLSTATE variable.

With the SQLCA, the application program provides the storage for the SQLCA that is used to communicate the results of the run of an SQL statement. With the SQL diagnostics area, the database manager manages the storage for the diagnostics, and the GET DIAGNOSTICS statement is provided to retrieve the contents of the diagnostics area.
Note that the SQLCA will continue to be supported for application programs. Also, the GET DIAGNOSTICS statement can be used in an application program that uses the SQLCA.

**Example: SQL routine exception**

In this application example, a stored procedure signals an error when an input value is out of range.

EXEC SQL CREATE PROCEDURE check_input (IN p1 INT)  
LANGUAGE SQL READS SQL DATA  
test: BEGIN  
IF p1 < 0 THEN  
  SIGNAL SQLSTATE '99999'  
  SET MESSAGE_TEXT = 'Bad input value';  
END IF;  
END test;

The calling application checks for a failure and retrieves the information about the failure from the SQL diagnostics area:

```sql
EXEC SQL CALL check_input(:hv1);
if (strncmp(SQLSTATE, "99999", 5) == 0)  
  EXEC SQL GET DIAGNOSTICS CONDITION 1  
  :hv2 = RETURNED_SQLSTATE,  
  :hv3 = MESSAGE_TEXT;
else  
{
}
```

**Example: Logging items from the SQL diagnostics area**

In this example, an application needs to log all errors for security reasons. The log can be used to monitor the health of a system or to monitor for inappropriate use of a database.

For each SQL error that occurs, an entry is placed in the log. The entry includes when the error occurred, what user was using the application, what type of SQL statement was run, the returned SQLSTATE value, and the message number and corresponding complete message text.

```sql
EXEC SQL WHENEVER SQLERROR GOTO error;  
(application code)
error:  
EXEC SQL GET DIAGNOSTICS :stmt_command = COMMAND_FUNCTION,  
  :error_count = NUMBER;  
for (condition_number=1; condition_number<=error_count; ++condition_number)  
{
  EXEC SQL GET DIAGNOSTICS CONDITION :condition_number
```
:auth_id = DB2_AUTHORIZATION_ID,
:errorMessage = RETURNED_SQLSTATE,
:msgid = DB2_MESSAGE_ID,
:msgtext = DB2_MESSAGE_TEXT;

EXEC SQL INSERT INTO error_log VALUES('CURRENT_TIMESTAMP,
:stmt_command,
:condition_number,
:auth_id,
:errorMessage,
:msgid,
:msgtext);

Related reference:
GET DIAGNOSTICS

**Handling exception conditions with the WHENEVER statement**

The WHENEVER statement causes SQL to check the SQLSTATE and SQLCODE and continue processing your program, or branch to another area in your program if an error, exception, or warning exists as a result of running an SQL statement.

An exception condition handling subroutine (part of your program) can then examine the SQLCODE or SQLSTATE field to take an action specific to the error or exception situation.

**Note:** The WHENEVER statement is not allowed in REXX procedures.

The WHENEVER statement allows you to specify what you want to do whenever a general condition is true. You can specify more than one WHENEVER statement for the same condition. When you do this, the first WHENEVER statement applies to all subsequent SQL statements in the source program until another WHENEVER statement is specified.

The WHENEVER statement looks like this:

EXEC SQL
WHENEVER condition action
END-EXEC.

There are three conditions you can specify:

**SQLWARNING**
Specify SQLWARNING to indicate what you want done when SQLWARN0 = W or SQLCODE contains a positive value other than 100 (SUBSTR(SQLSTATE,1,2) = '01').

**Note:** SQLWARN0 could be set for several different reasons. For example, if the value of a column was truncated when it was moved into a host variable, your program might not regard this as an error.

**SQLERROR**
Specify SQLERROR to indicate what you want done when an error code is returned as the result of an SQL statement (SQLCODE < 0) (SUBSTR(SQLSTATE,1,2) > '02').

**NOT FOUND**
Specify NOT FOUND to indicate what you want done when an SQLCODE of +100 and a SQLSTATE of '02000' is returned because:

- After a single-row SELECT is issued or after the first FETCH is issued for a cursor, the data the program specifies does not exist.
- After a subsequent FETCH, no more rows satisfying the cursor select-statement are left to retrieve.
- After an UPDATE, a DELETE, or an INSERT, no row meets the search condition.
You can also specify the action you want taken:

**CONTINUE**
This causes your program to continue to the next statement.

**GO TO label**
This causes your program to branch to an area in the program. The label for that area may be preceded with a colon. The WHENEVER ... GO TO statement:
- Must be a section name or an unqualified paragraph name in COBOL
- Is a label in PL/I and C
- Is the label of a TAG in RPG

For example, if you are retrieving rows using a cursor, you expect that SQL will eventually be unable to find another row when the FETCH statement is issued. To prepare for this situation, specify a WHENEVER NOT FOUND GO TO ... statement to cause SQL to branch to a place in the program where you issue a CLOSE statement in order to close the cursor properly.

**Note:** A WHENEVER statement affects all subsequent source SQL statements until another WHENEVER is encountered.

In other words, all SQL statements coded between two WHENEVER statements (or following the first, if there is only one) are governed by the first WHENEVER statement, regardless of the path the program takes.

Because of this, the WHENEVER statement must precede the first SQL statement it is to affect. If the WHENEVER follows the SQL statement, the branch is not taken on the basis of the value of the SQLCODE and SQLSTATE set by that SQL statement. However, if your program checks the SQLCODE or SQLSTATE directly, the check must be done after the SQL statement is run.

The WHENEVER statement does not provide a CALL to a subroutine option. For this reason, you might want to examine the SQLCODE or SQLSTATE value after each SQL statement is run and call a subroutine, rather than use a WHENEVER statement.

### Coding SQL statements in C and C++ applications

To embed SQL statements in an ILE C or C++ program, you need to be aware of some unique application and coding requirements. This topic also defines the requirements for host structures and host variables.

**Note:** By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 190.

**Related concepts:**
- "Writing applications that use SQL" on page 2

You can create database applications in host languages that use Db2 for i SQL statements and functions.

**Related reference:**
- "Example programs: Using Db2 for i statements" on page 149
- "Example: SQL statements in ILE C and C++ programs" on page 151

Here is a sample application that shows how to code SQL statements in each of the languages that Db2 for i supports.

This example program is written in the C programming language.

### Defining the SQL communication area in C and C++ applications that use SQL

A C or C++ program can be written to use the SQLCA to check return status for embedded SQL statements, or the program can use the SQL diagnostics area to check return status.
When using the SQLCA, a C or C++ program that contains SQL statements must include one or both of the following:

- An SQLCODE variable declared as long SQLCODE
- An SQLSTATE variable declared as char SQLSTATE[6]

Or,
- An SQLCA (which contains an SQLCODE and SQLSTATE variable).

The SQLCODE and SQLSTATE values are set by the database manager after each SQL statement is run. An application can check the SQLCODE or SQLSTATE value to determine whether the last SQL statement was successful.

You can code the SQLCA in a C or C++ program directly or by using the SQL INCLUDE statement. When coding it directly, initialize the SQLCA using the following statement:

```c
struct sqlca sqlca = {0x0000000000000000};
```

Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```sql
EXEC SQL INCLUDE SQLCA;
```

A standard declaration includes a structure definition and a data area that are named sqlca.

The SQLCODE, SQLSTATE, and SQLCA variables must appear before any executable statements. The scope of the declaration must include the scope of all SQL statements in the program.

The included C and C++ source statements for the SQLCA are:

```c
struct sqlca {
    unsigned char sqlcaid[8];
    long sqlcabc;
    long sqlcode;
    short sqlerrml;
    unsigned char sqlerrmc[70];
    unsigned char sqlerrp[8];
    long sqlerrd[6];
    unsigned char sqlwarn[11];
    unsigned char sqlstate[5];
};
#define SQLCODE sqlca.sqlcode
#define SQLWARN0 sqlca.sqlwarn[0]
#define SQLWARN1 sqlca.sqlwarn[1]
#define SQLWARN2 sqlca.sqlwarn[2]
#define SQLWARN3 sqlca.sqlwarn[3]
#define SQLWARN4 sqlca.sqlwarn[4]
#define SQLWARN5 sqlca.sqlwarn[5]
#define SQLWARN6 sqlca.sqlwarn[6]
#define SQLWARN7 sqlca.sqlwarn[7]
#define SQLWARN8 sqlca.sqlwarn[8]
#define SQLWARN9 sqlca.sqlwarn[9]
#define SQLWARN10 sqlca.sqlwarn[10]
#define SQLSTATE sqlca.sqlstate
struct sqlca sqlca = {0x0000000000000000};
```

When a declare for SQLCODE is found in the program and the precompiler provides the SQLCA, SQLCADE replaces SQLCODE. When a declare for SQLSTATE is found in the program and the precompiler provides the SQLCA, SQLSTOTE replaces SQLSTATE.

**Note:** Many SQL error messages contain message data that is of varying length. The lengths of these data fields are embedded in the value of the SQLCA sqlerrmc field. Because of these lengths, printing the value of sqlerrmc from a C or C++ program might give unpredictable results.
Related concepts:

"Using the SQL diagnostics area" on page 7

The SQL diagnostics area is used to keep the returned information for an SQL statement that has been run in a program. It contains all the information that is available to you as an application programmer through the SQLCA.

Related reference:

SQL communication area

GET DIAGNOSTICS

Defining SQL descriptor areas in C and C++ applications that use SQL

There are two types of SQL descriptor areas. One is defined with the ALLOCATE DESCRIPTOR statement. The other is defined using the SQL descriptor area (SQLDA) structure. In this topic, only the SQLDA form is discussed.

The following statements can use an SQLDA:

- EXECUTE...USING DESCRIPTOR descriptor-name
- FETCH...USING DESCRIPTOR descriptor-name
- OPEN...USING DESCRIPTOR descriptor-name
- DESCRIBE statement-name INTO descriptor-name
- DESCRIBE CURSOR cursor-name INTO descriptor-name
- DESCRIBE INPUT statement-name INTO descriptor-name
- DESCRIBE PROCEDURE procedure-name INTO descriptor-name
- DESCRIBE TABLE host-variable INTO descriptor-name
- PREPARE statement-name INTO descriptor-name
- CALL...USING DESCRIPTOR descriptor-name

Unlike the SQLCA, more than one SQLDA can be in the program, and an SQLDA can have any valid name. The following list includes the statements that require a SQLDA. You can code an SQLDA in a C or C++ program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard SQLDA declaration:

EXEC SQL INCLUDE SQLDA;

A standard declaration includes only a structure definition with the name 'sqlda'.

C and C++ declarations that are included for the SQLDA are:

```c
struct sqlda {
    unsigned char sqldaid[8];
    long sqldabc;
    short sqln;
    short sqld;
    struct sqlvar {
        short sqltype;
        short sqllen;
        union {
            unsigned char *sqldata;
            long long sqld_result_set_locator; }
        union {
            short *sqlind;
            long sqld_row_change;
            long sqld_result_set_rows; }
    struct sqlname {
        short length;
    }
```
unsigned char data[30];
    } sqlname;
} sqlvar[1];

One benefit from using the INCLUDE SQLDA SQL statement is that you also get the following macro definition:

#define SQLDASIZE(n) (sizeof(struct sqlda) + (n-1)*sizeof(struct sqlvar))

This macro makes it easy to allocate storage for an SQLDA with a specified number of SQLVAR elements. In the following example, the SQLDASIZE macro is used to allocate storage for an SQLDA with 20 SQLVAR elements.

#include <stdlib.h>
EXEC SQL INCLUDE SQLDA;

struct sqlda *mydaptr;
short numvars = 20;

mydaptr = (struct sqlda *) malloc(SQLDASIZE(numvars));
mydaptr->sqln = 20;

Here are other macro definitions that are included with the INCLUDE SQLDA statement:

GETSQLDOUBLED(daptr)
    Returns 1 if the SQLDA pointed to by daptr has been doubled, or 0 if it has not been doubled. The SQLDA is doubled if the seventh byte in the SQLDAID field is set to '2'.

SETSQDLOUDBED(daptr, newvalue)
    Sets the seventh byte of SQLDAID to a new value.

GETSQLDALONGLEN(daptr,n)
    Returns the length attribute of the nth entry in the SQLDA to which daptr points. Use this only if the SQLDA was doubled and the nth SQLVAR entry has a LOB data type.

SETSQDALONGLEN(daptr,n,len)
    Sets the SQLLONGLEN field of the SQLDA to which daptr points to len for the nth entry. Use this only if the SQLDA was doubled and the nth SQLVAR entry has a LOB datatype.

GETSQLDLENPTR(daptr,n)
    Returns a pointer to the actual length of the data for the nth entry in the SQLDA to which daptr points. The SQLDATALEN pointer field returns a pointer to a long (4 byte) integer. If the SQLDATALEN pointer is zero, a NULL pointer is returned. Use this only if the SQLDA has been doubled.

SETSQDLENPTR(daptr,n,ptr)
    Sets a pointer to the actual length of the data for the nth entry in the SQLDA to which daptr points. Use this only if the SQLDA has been doubled.

When you have declared an SQLDA as a pointer, you must reference it exactly as declared when you use it in an SQL statement, just as you would for a host variable that was declared as a pointer. To avoid compiler errors, the type of the value that is assigned to the sqldata field of the SQLDA must be a pointer of unsigned character. This helps avoid compiler errors. The type casting is only necessary for the EXECUTE, OPEN, CALL, and FETCH statements where the application program is passing the address of the host variables in the program. For example, if you declared a pointer to an SQLDA called mydaptr, you would use it in a PREPARE statement as:

EXEC SQL PREPARE mysname INTO ::mydaptr FROM :mysqlstring;

SQLDA declarations can appear wherever a structure definition is allowed. Normal C scope rules apply.
Dynamic SQL is an advanced programming technique. With dynamic SQL, your program can develop and then run SQL statements while the program is running. A SELECT statement with a variable SELECT list (that is a list of the data to be returned as part of the query) that runs dynamically requires an SQL descriptor area (SQLDA). This is because you will not know in advance how many or what type of variables to allocate in order to receive the results of the SELECT.

Related concepts:
- Dynamic SQL applications

Related reference:
- SQL descriptor area

**Embedding SQL statements in C and C++ applications that use SQL**

SQL statements can be coded in a C or C++ program wherever executable statements can appear.

Each SQL statement must begin with EXEC SQL and end with a semicolon (;). The EXEC SQL keywords must be on one line. The remaining part of the SQL statement can be on more than one line.

*Example:* An UPDATE statement coded in a C or C++ program might be coded in the following way:

```c
EXEC SQL
UPDATE DEPARTMENT
SET MGRNO = :MGR_NUM
WHERE DEPTNO = :INT_DEPT ;
```

Do not use `#pragma convert` to modify the CCSID of any literals that could be used by SQL. All literals used by SQL are assumed to be in the CCSID of the source file.

**Comments in C and C++ applications that use SQL**

In addition to using SQL comments (--), you can include C comments (/\*...\*/ and single-line comments (comments that start with //) within embedded SQL statements whenever a blank is allowed, except between the keywords EXEC and SQL.

Comments can span any number of lines. You cannot nest comments.

**Continuation for SQL statements in C and C++ applications that use SQL**

SQL statements can be contained in one or more lines.

You can split an SQL statement wherever a blank can appear. The backslash (\) can be used to continue a string constant or delimited identifier. Identifiers that are not delimited cannot be continued.

Constants containing DBCS data may be continued across multiple lines in two ways:

- If the character at the right margin of the continued line is a shift-in and the character at the left margin of the continuation line is a shift-out, then the shift characters located at the left and right margin are removed.
  
  This SQL statement has a valid graphic constant of G’<AABBCCDDEEFFGGHHIIJJKKK>'. The redundant shifts at the margin are removed.

  ```sql
  *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  EXEC SQL SELECT * FROM GRAPHTAB WHERE GRAPHCOL = G’<AABBCCDDEEFFGGHH>
  <IIJJKKK>’;
  ```

- It is possible to place the shift characters outside of the margins. For this example, assume the margins are 5 and 75. This SQL statement has a valid graphic constant of G’<AABBCCDDEEFFGGHHIIJJKKK>'.

  ```sql
  *...+(....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
  EXEC SQL SELECT * FROM GRAPHTAB WHERE GRAPHCOL = G’<AABBCCDD> <EFFFFGGHHIIJJKKK>’;
  ```
Including code in C and C++ applications that use SQL

You can include SQL statements, C, or C++ statements by embedding the following SQL statement in the source code.

```
EXEC SQL INCLUDE member-name;
```

You cannot use C and C++ #include statements to include SQL statements or declarations of C or C++ host variables that are referred to in SQL statements.

Margins in C and C++ applications that use SQL

When you precompile using a source member, you must code SQL statements within the margins that are specified by the MARGINS parameter on the `CRTSQLCI` or `CRTSQLCPP` command.

If the MARGINS parameter is specified as `*SRCFILE`, the record length of the source file will be used. If a value is specified for the right margin and that value is larger than the source record length, the entire record will be read. The value will also apply to any included members. For example, if a right margin of 200 is specified and the source file has a record length of 80, only 80 columns of data will be read from the source file. If an included source member in the same precompile has a record length of 200, the entire 200 from the include will be read.

When you precompile using a source stream file, the MARGINS parameter is ignored; the entire file is read. Any source stream file included using the SQL INCLUDE statement is read up to the length of the longest line in the primary source stream file, which is specified on the SRCSTMF parameter.

If EXEC SQL does not start within the margins, the SQL precompiler does not recognize the SQL statement.

Related concepts:

- “CL command descriptions for host language precompilers” on page 188

The IBM DB2 Query Manager and SQL Development Kit for i licensed program provides commands for precompiling programs coded in these programming languages.

Names in C and C++ applications that use SQL

You can use any valid C or C++ variable name for a host variable. It is subject to these restrictions.

Do not use host variable names or external entry names that begin with SQL, RDI, or DSN in any combination of uppercase or lowercase letters. These names are reserved for the database manager. The length of host variable names is limited to 128.

If the name SQL in any combination of uppercase or lowercase letters is used, unpredictable results might occur.

NULLs and NULs in C and C++ applications that use SQL

C, C++, and SQL use the word null, but for different meanings.

The C and C++ languages have a null character (NUL), a null pointer (NULL), and a null statement (just a semicolon (;)). The C NUL is a single character that compares equal to 0. The C NULL is a special reserved pointer value that does not point to any valid data object. The SQL null value is a special value that is distinct from all non-null values and denotes the absence of a (non-null) value.

Statement labels in C and C++ applications that use SQL

Executable SQL statements can be preceded with a label.

Preprocessor sequence for C and C++ applications that use SQL

You must run the SQL preprocessor before the C or C++ preprocessor. You cannot use C or C++ preprocessor directives within SQL statements.
Trigraphs in C and C++ applications that use SQL
Some characters from the C and C++ character set are not available on all keyboards. You can enter these characters into a C or C++ source program by using a sequence of three characters that is called a trigraph.

The following trigraph sequences are supported within host variable declarations:
- ??( left bracket
- ??) right bracket
- ??< left brace
- ??> right brace
- ??= pound
- ??/ backslash

WHENEVER statement in C and C++ applications that use SQL
The target for the GOTO clause in an SQL WHENEVER statement must be within the scope of any SQL statements affected by the WHENEVER statement.

Using host variables in C and C++ applications that use SQL
All host variables used in SQL statements must be explicitly declared prior to their first use.

In C, the C statements that are used to define the host variables should be preceded by a BEGIN DECLARE SECTION statement and followed by an END DECLARE SECTION statement. If a BEGIN DECLARE SECTION and END DECLARE SECTION are specified, all host variable declarations used in SQL statements must be between the BEGIN DECLARE SECTION and the END DECLARE SECTION statements. Host variables declared using a typedef identifier also require a BEGIN DECLARE SECTION and END DECLARE SECTION; however, the typedef declarations do not need to be between these two sections.

In C++, the C++ statements that are used to define the host variables must be preceded by a BEGIN DECLARE SECTION statement and followed by an END DECLARE SECTION statement. You cannot use any variable that is not between the BEGIN DECLARE SECTION statement and the END DECLARE SECTION statement as a host variable.

All host variables within an SQL statement must be preceded by a colon (:).

The names of host variables must be unique within the program, even if the host variables are in different blocks or procedures.

An SQL statement that uses a host variable must be within the scope of the statement in which the variable was declared.

Host variables cannot be union elements.

Host variables cannot contain continuation characters within the name.

Declaring host variables in C and C++ applications that use SQL
The C and C++ precompilers recognize only a subset of valid C and C++ declarations as valid host variable declarations.

Numeric host variables in C and C++ applications that use SQL:

This figure shows the syntax for valid numeric host variable declarations.
**Numeric**

```plaintext
auto extern static const volatile
float double decimal (precision, scale)
_Decimal32
_Decimal64
_Decimal128
long long
signed long short
sqlint32 sqlint64

variable-name = expression ;
```

**Notes:**

1. Precision and scale must be integer constants. Precision may be in the range from 1 to 63. Scale may be in the range from 0 to the precision.
2. If using the decimal data type, the header file decimal.h must be included.
3. If using sqlint32 or sqlint64, the header file sqlsystm.h must be included.
4. _Decimal32, _Decimal64, and _Decimal128 are only supported for C.

**Character host variables in C and C++ applications that use SQL:**

There are three valid forms for character host variables.

These forms are:
- Single-character form
- NUL-terminated character form
- VARCHAR structured form

In addition, an SQL VARCHAR declare can be used to define a varchar host variable.

All character types are treated as unsigned.

**Single-character form**
NUL-terminated character form

Notes:
1. The length must be an integer constant that is greater than 1 and not greater than 32 741.
2. If the *CNULRQD option is specified on the CRTSQLCI or CRTSQLCPPI command, the input host variables must contain the NUL-terminator. Output host variables are padded with blanks, and the last character is the NUL-terminator. If the output host variable is too small to contain both the data and the NUL-terminator, the following actions are taken:
   - The data is truncated
   - The last character is the NUL-terminator
   - SQLWARN1 is set to 'W'
3. If the *NOCNULRQD option is specified on the CRTSQLCI or CRTSQLCPPI command, the input variables do not need to contain the NUL-terminator. The following applies to output host variables.
   - If the host variable is large enough to contain the data and the NUL-terminator, then the following actions are taken:
     - The data is returned, but the data is not padded with blanks
     - The NUL-terminator immediately follows the data
   - If the host variable is large enough to contain the data but not the NUL-terminator, then the following actions are taken:
     - The data is returned
     - A NUL-terminator is not returned
     - SQLWARN1 is set to 'N'
   - If the host variable is not large enough to contain the data, the following actions are taken:
     - The data is truncated
     - A NUL-terminator is not returned
     - SQLWARN1 is set to 'W'

VARCHAR structured form
Notes:
1. `length` must be an integer constant that is greater than 0 and not greater than 32740.
2. `var-1` and `var-2` must be simple variable references and cannot be used individually as integer and character host variables.
3. The struct tag can be used to define other data areas, but these cannot be used as host variables.
4. The VARCHAR structured form should be used for bit data that may contain the NULL character. The VARCHAR structured form will not be ended using the null-terminator.
5. `_Packed` must not be used in C++. Instead, specify `#pragma pack(1)` prior to the declaration and `#pragma pack()` after the declaration.

Note: You can use `#pragma pack (reset)` instead of `#pragma pack()` because they are the same.

```c
#pragma pack(1)
struct VARCHAR {
    short len;
    char s[10];
} vstring;
#pragma pack()
```

Example:
EXEC SQL BEGIN DECLARE SECTION;

/* valid declaration of host variable vstring */
struct VARCHAR {
    short len;
    char s[10];
} vstring;

/* invalid declaration of host variable wstring */
struct VARCHAR wstring;

SQL VARCHAR form
**Notes:**

1. VARCHAR can be in mixed case.
2. `length` must be an integer constant that is greater than 0 and not greater than 32,740.
3. The SQL VARCHAR form should be used for bit data that may contain the NULL character. The SQL VARCHAR form will not be ended using the nul-terminator.

**Example**

The following declaration:
```
VARCHAR vstring[528]="mydata";
```

Results in the generation of the following structure:
```
_Packed struct { short len;
    char data[528];}
 vstring={6, "mydata"};
```

The following declaration:
```
VARCHAR vstring1[111],
    vstring2[222]="mydata",
    vstring3[333]="more data";
```

Results in the generation of the following structures:
```
_Packed struct { short len;
    char data[111];}
 vstring1;
_Packed struct { short len;
    char data[222];}
 vstring2={6, "mydata"};
_Packed struct { short len;
    char data[333];}
 vstring3={9, "more data"};
```

**Graphic host variables in C and C++ applications that use SQL:**

There are three valid forms for graphic host variables.
- Single-graphic form
- NUL-terminated graphic form
- VARGRAPHIC structured form

**Single-graphic form**

```c
auto extern static const volatile wchar_t variable-name = expression;
```

**NUL-terminated graphic form**
Notes:
1. `length` must be an integer constant that is greater than 1 and not greater than 16371.
2. If the `*CNULRQD` option is specified on the `CRTSQLCI` or `CRTSQLCPP` command, then input host variables must contain the graphic NUL-terminator (/0/). Output host variables are padded with DBCS blanks, and the last character is the graphic NUL-terminator. If the output host variable is too small to contain both the data and the NUL-terminator, the following actions are taken:
   - The data is truncated
   - The last character is the graphic NUL-terminator
   - SQLWARN1 is set to 'W'
If the `*NOCNULRQD` option is specified on the `CRTSQLCI` or `CRTSQLCPP` command, the input host variables do not need to contain the graphic NUL-terminator. The following is true for output host variables.
   - If the host variable is large enough to contain the data and the graphic NUL-terminator, the following actions are taken:
     - The data is returned, but is not padded with DBCS blanks
     - The graphic NUL-terminator immediately follows the data
   - If the host variable is large enough to contain the data but not the graphic NUL-terminator, the following actions are taken:
     - The data is returned
     - A graphic NUL-terminator is not returned
     - SQLWARN1 is set to 'N'
   - If the host variable is not large enough to contain the data, the following actions are taken:
     - The data is truncated
     - A graphic NUL-terminator is not returned
     - SQLWARN1 is set to 'W'

**VARGRAPHIC structured form**
Notes:
1. length must be an integer constant that is greater than 0 and not greater than 16370.
2. var-1 and var-2 must be simple variable references and cannot be used as host variables.
3. The struct tag can be used to define other data areas, but these cannot be used as host variables.
4. _Packed must not be used in C++. Instead, specify #pragma pack(1) prior to the declaration and #pragma pack() after the declaration.

Example
EXEC SQL BEGIN DECLARE SECTION;

/* valid declaration of host variable graphic string */
struct VARGRAPH {
  short len;
  wchar_t s[10];
} vstring;

/* invalid declaration of host variable wstring */
struct VARGRAPH wstring;

Binary host variables in C and C++ applications that use SQL:

C and C++ do not have variables that correspond to the SQL binary data types. To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a C language structure in the output source member.

**BINAR**

**VARBINAR**
Notes:
1. For BINARY host variables, the length must be in the range 1 to 32 766.
2. For VARBINARY and BINARY VARYING host variables, the length must in the range 1 to 32 740.
3. SQL TYPE IS, BINARY, VARBINARY, and BINARY VARYING can be in mixed case.

**BINARY example**

The following declaration:

```c
SQL TYPE IS BINARY(4) myBinField;
```

Results in the generation of the following code:

```c
char myBinField[4];
```

**VARBINARY example**

The following declaration:

```c
SQL TYPE IS VARBINARY(12) myVarBinField;
```

Results in the generation of the following structure:

```c
_Packed struct myVarBinField_t {
    short length;
    char data[12];
} myVarBinField;
```

**LOB host variables in C and C++ applications that use SQL:**

C and C++ do not have variables that correspond to the SQL data types for LOBs (large objects). To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a C language structure in the output source member.

**LOB host variable**
variable-name

= SQL_CLOB_INIT("init-data")
= SQL_DBCLOB_INIT("init-data")
= SQL_BLOB_INIT("init-data")

Notes:
2. For BLOB and CLOB, $1 \leq \text{length} \leq 2^{14}$
3. For DBCLOB, $1 \leq \text{length} \leq 1 \times 10^{23}$
4. SQL TYPE IS, BLOB, CLOB, DBCLOB, K, M, G can be in mixed case.
5. The maximum length allowed for the initialization string is 32 766 bytes.
6. The initialization length, init-len, must be a numeric constant (that is, it cannot include K, M, or G).
7. If the LOB is not initialized within the declaration, then no initialization will be done within the precompiler generated code.
8. The precompiler generates a structure tag which can be used to cast to the host variable's type.
9. Pointers to LOB host variables can be declared, with the same rules and restrictions as for pointers to other host variable types.
10. CCSID processing for LOB host variables will be the same as the processing for other character and graphic host variable types.
11. If a DBCLOB is initialized, it is the user's responsibility to prefix the string with an 'L' (indicating a wide-character string).

CLOB example

The following declaration:
SQL TYPE IS CLOB(128K) var1, var2 = {10, "data2data2"};

The precompiler will generate for C:
_PACKED struct var1_t {
    unsigned long length;
    char data[131072];
} var1, var2={10,"data2data2"};

DBCLOB example

The following declaration:
SQL TYPE IS DBCLOB(128K) my_dbclob;

The precompiler will then generate:
_PACKED struct my_dbclob_t {
    unsigned long length;
    wchar_t data[131072];
} my_dbclob;

BLOB example

The following declaration:
static SQL TYPE IS BLOB(128K)
    my_blob=SQL_BLOB_INIT("mydata");

Results in the generation of the following structure:
static struct my_blob_t {
    unsigned long length;
    char data[131072];
} my_blob=SQL_BLOB_INIT("my_data");

LOB locator

Notes:
1. SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR, DBCLOB_LOCATOR can be in mixed case.
2. init-value permits the initialization of pointer locator variables. Other types of initialization will have no meaning.
3. Pointers to LOB locators can be declared with the same rules and restrictions as for pointers to other host variable types.

CLOB locator example

The following declaration:
static SQL TYPE IS CLOB_LOCATOR my_locator;

Results in the following generation:
static long int unsigned my_locator;

LOB and DBCLOB locators have similar syntax.

LOB file reference variable

Notes:
1. SQL TYPE IS, BLOB_FILE, CLOB_FILE, DBCLOB_FILE can be in mixed case.
2. Pointers to LOB File Reference Variables can be declared, with the same rules and restrictions as for pointers to other host variable types.
CLOB file reference example

The following declaration:
static SQL TYPE IS CLOB_FILE my_file;

Results in the generation of the following structure:
static _Packed struct {
    unsigned long name_length;
    unsigned long data_length;
    unsigned long file_options;
    char name[255];
} my_file;

BLOB and DBCLOB file reference variables have similar syntax.

The precompiler generates declarations for the following file option constants. You can use these constants to set the file_options variable when you use file reference host variables.

- SQL_FILE_READ (2)
- SQL_FILE_CREATE (8)
- SQL_FILE_OVERWRITE (16)
- SQL_FILE_APPEND (32)

Related reference:
LOB file reference variables

XML host variables in C and C++ applications that use SQL:

C and C++ do not have variables that correspond to the SQL data type for XML. To create host variables that can be used with this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a C language structure in the output source member.

XML host variable

Notes:
2. For BLOB and CLOB, $1 \leq length \leq 2\, 147\, 483\, 647$
3. For DBCLOB, $1 \leq length \leq 1\, 073\, 741\, 823$
4. SQL TYPE IS, XML AS, BLOB, CLOB, DBCLOB, K, M, G can be in mixed case.
5. The maximum length allowed for the initialization string is 32 766 bytes.
6. The initialization length, init-len, must be a numeric constant (that is, it cannot include K, M, or G).
7. If the XML variable is not initialized within the declaration, then no initialization will be done within the precompiler generated code.

8. The precompiler generates a structure tag which can be used to cast to the host variable’s type.

9. Pointers to XML host variables can be declared, with the same rules and restrictions as for pointers to other host variable types.

10. The CCSID value for an XML host variable can be explicitly set by the DECLARE VARIABLE statement. Otherwise, the value specified by the SQL_XML_DATA_CCSID QAQQINI option will be used. The default for this QAQQINI option is CCSID 1208.

11. If XML AS DBCLOB is initialized, it is the user’s responsibility to prefix the string with an ‘L’ (indicating a wide-character string).

**XML example**

The following declaration:

```sql
SQL TYPE IS XML AS CLOB(5000) var1;
```

The precompiler will generate for C:

```c
_Packed struct var1_t {
    unsigned long length;
    char data[5000];
} var1;
```

**XML locator**

```sql
SQL TYPE IS XML AS LOCATOR my_locator;
```

Results in the following generation:

```c
static long int unsigned my_locator;
```

**Notes:**

1. SQL TYPE IS, XML AS LOCATOR can be in mixed case.
2. *init-value* permits the initialization of pointer locator variables. Other types of initialization will have no meaning.
3. Pointers to XML locators can be declared with the same rules and restrictions as for pointers to other host variable types.

**XML locator example**

The following declaration:

```sql
static SQL TYPE IS XML AS LOCATOR my_locator;
```

Results in the following generation:

```c
static long int unsigned my_locator;
```
XML file reference variable

```
auto extern static volatile SQL TYPE IS XML AS CLOB_FILE DBCLOB_FILE BLOB_FILE
```

```
variable-name = init-value
```

Notes:
1. SQL TYPE IS, XML AS, BLOB_FILE, CLOB_FILE, DBCLOB_FILE can be in mixed case.
2. Pointers to XML File Reference Variables can be declared, with the same rules and restrictions as for pointers to other host variable types.

XML file reference example

The following declaration:
```
static SQL TYPE IS XML AS CLOB_FILE my_file;
```

Results in the generation of the following structure:
```
static _Packed struct {
    unsigned long name_length;
    unsigned long data_length;
    unsigned long file_options;
    char name[255];
} my_file;
```

The precompiler generates declarations for the following file option constants. You can use these constants to set the file_options variable when you use file reference host variables.

- SQL_FILE_READ (2)
- SQL_FILE_CREATE (8)
- SQL_FILE_OVERWRITE (16)
- SQL_FILE_APPEND (32)

ROWID host variables in C and C++ applications that use SQL:

C and C++ do not have a variable that corresponds to the SQL data type ROWID. To create host variables that can be used with this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a C language structure in the output source member.

```
SQL TYPE IS ROWID variable-name
```

Note: SQL TYPE IS ROWID can be in mixed case.
ROWID example

The following declaration:
SQL TYPE IS ROWID myrowid, myrowid2;

Results in the generation of the following structure:
_Packed struct { short len;
    char data[40];}
myrowd1, myrowid2;

Result set locator host variables in C and C++ applications that use SQL:

C and C++ do not have a variable that corresponds to the SQL result set locator data type. To create host variables that can be used for this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a C language structure in the output source member.

Result set locator

►►SQL TYPE IS RESULT_SET_LOCATOR variable-name ;◄◄

Note: SQL TYPE IS RESULT_SET_LOCATOR can be in mixed case.

Result set locator example

The following declaration:
SQL TYPE IS RESULT_SET_LOCATOR resloc1;

Results in the generation of the following structure:
long long unsigned resloc1;

Using host structures in C and C++ applications that use SQL

In C and C++ programs, you can define a host structure, which is a named set of elementary C or C++ variables.

Host structures have a maximum of two levels, even though the host structure might itself occur within a multilevel structure. An exception is the declaration of a varying-length string, which requires another structure.

A host structure name can be a group name whose subordinate levels name elementary C or C++ variables. For example:

struct {
    struct {
        char c1;
        char c2;
    } b_st;
} a_st;

In this example, b_st is the name of a host structure consisting of the elementary items c1 and c2.

You can use the structure name as a shorthand notation for a list of scalars, but only for a two-level structure. You can qualify a host variable with a structure name (for example, structure.field). Host structures are limited to two levels. (For example, in the above host structure example, the a_st cannot be referred to in SQL.) A structure cannot contain an intermediate level structure. In the previous example,
A host structure for SQL data has two levels and can be thought of as a named set of host variables. After the host structure is defined, you can refer to it in an SQL statement instead of listing the several host variables (that is, the names of the host variables that make up the host structure).

For example, you can retrieve all column values from selected rows of the table CORPDATA.EMPLOYEE with:

```c
struct { char empno[7];
    struct {
        short int firstname_len;
        char firstname_text[12];
    } firstname;
    char midint,
    struct {
        short int lastname_len;
        char lastname_text[15];
    } lastname;
    char workdept[4];
} pemp1;

......
strcpy(pemp1.empno, "000220");
......
exec sql
SELECT *
    INTO :pemp1
    FROM corpdata.employee
    WHERE empno=:pemp1.empno;
```

Notice that in the declaration of pemp1, two varying-length string elements are included in the structure: firstname and lastname.

### Host structure declarations in C and C++ applications that use SQL

These figures show the valid syntax for host structure declarations.

### Host structures
variable-name = expression;

varchar-structure:

struct { signed short int var-3; var-3 }

char var-4[length];

Host structures (continued)

vargraphic-structure:

struct { signed short int var-6; wchar_t var-7[length]; }
binary:

SQL TYPE IS BINARY (length)
VARBINARY
BINARY VARYING

lob:

SQL TYPE IS CLOB (length)
DBCLOB
BLOB
CLOB_LOCATOR
DBCLOB_LOCATOR
BLOB_LOCATOR
CLOB_FILE
DBCLOB_FILE
BLOB_FILE

xml:

SQL TYPE IS XML AS CLOB (length)
DBCLOB
BLOB
LOCATOR
CLOB_FILE
DBCLOB_FILE
BLOB_FILE

SQL-varchar:

VARCHAR variable-name [length]

rowid:

SQL TYPE IS ROWID

result-set-locator:

SQL TYPE IS RESULT_SET_LOCATOR

Notes:

1. For details on declaring numeric, character, graphic, binary, LOB, XML, ROWID, and result set locator host variables, see the notes under numeric, character, graphic, binary, LOB, XML, ROWID, and result set locator host variables.

2. A structure of a short int followed by either a char or wchar_t array is always interpreted by the SQL C and C++ precompilers as either a VARCHAR or VARGRAPHIC structure.

3. _Packed must not be used in C++. Instead, specify #pragma pack(1) prior to the declaration and #pragma pack() after the declaration.
#pragma pack(1)
struct {
    short myshort;
    long mylong;
    char mychar[5];
} a_st;
#pragma pack()

4. If using sqlint32 or sqlint64, the header file sqlsystm.h must be included.
5. _Decimal32, _Decimal64, and _Decimal128 are only supported for C.

Host structure indicator array in C and C++ applications that use SQL
This figure shows the valid syntax for host structure indicator array declarations.

Host structure indicator array

Note: Dimension must be an integer constant between 1 and 32 767.

Using arrays of host structures in C and C++ applications that use SQL
In C and C++ programs, you can define a host structure array that has the dimension attribute. Host structure arrays have a maximum of two levels, even though the array might occur within a multiple-level structure. Another structure is not needed if a varying-length character string or a varying-length graphic string is not used.

In this C example,
struct {
    _Packed struct{
        char c1_var[20];
        short c2_var;
    } b_array[10];
} a_struct;

and in this C++ example,

```
#pragma pack(1)
struct {
    struct{
        char c1_var[20];
        short c2_var;
    } b_array[10];
} a_struct;
#pragma pack()
```

the following are true:
- All of the members in b_array must be valid variable declarations.
- The _Packed attribute must be specified for the struct tag.
- b_array is the name of an array of host structures containing the members c1_var and c2_var.
- b_array may only be used on the blocked forms of FETCH statements and INSERT statements.
- c1_var and c2_var are not valid host variables in any SQL statement.
- A structure cannot contain an intermediate level structure.

For example, in C you can retrieve 10 rows from the cursor with:

```c
_Packed struct {char first_initial;
    char middle_initial;
    _Packed struct {short lastname_len;
        char lastname_data[15];
    } lastname;
    double total_salary;
} employee_rec[10];
struct { short inds[4];
} employee_inds[10];
...
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT SUBSTR(FIRSTNME,1,1), MIDINIT, LASTNAME,
           SALARY+BONUS+COMM
    FROM CORPDATA.EMPLOYEE;
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 FOR 10 ROWS INTO :employee_rec:employee_inds;
...
```

**Host structure array in C and C++ applications that use SQL**

The figure shows the valid syntax for host structure array declarations.

```sql
auto extern static const volatile _Packed struct { tag }
```
IBM i: Embedded SQL programming
Notes:
1. For details on declaring numeric, character, graphic, binary, LOB, XML, ROWID, and result set locator host variables, see the notes under numeric-host variables, character-host variables, graphic-host variables, binary host variables, LOB host variables, XML host variables, ROWID host variables, and result set locator host variables.
2. The struct tag can be used to define other data areas, but these cannot be used as host variables.
3. Dimension must be an integer constant between 1 and 32,767.
4. _Packed must not be used in C++. Instead, specify #pragma pack(1) prior to the declaration and #pragma pack() after the declaration.
5. If using sqlint32 or sqlint64, the header file sqlsystm.h must be included.
6. _Decimal32, _Decimal64, and _Decimal128 are only supported for C.

**Host structure array indicator structure in C and C++ applications that use SQL**

The figure shows the valid syntax for host structure array indicator structure declarations.

### Host Structure Array Indicator Structure

```
struct tag {  
  _Packed
  var-1[dimenion-1] ;

  variable-name[dimension-2] = expression ;

  int ,
  short var-1[dimension-1] ;
}
```

**Notes:**
1. The struct tag can be used to define other data areas, but they cannot be used as host variables.
2. dimension-1 and dimension-2 must both be integer constants between 1 and 32,767.
3. _Packed must not be used in C++. Instead, specify #pragma pack(1) prior to the declaration and #pragma pack() after the declaration.

**Using pointer data types in C and C++ applications that use SQL**

You can also declare host variables that are pointers to the supported C and C++ data types, with the following restrictions.

- If a host variable is declared as a pointer, then that host variable must be declared with asterisks followed by a host variable. The following examples are all valid:

```c
short *mynum; /* Ptr to an integer */
long **mynumptr; /* Ptr to a ptr to a long integer */
char *mychar; /* Ptr to a single character */
char(*mychara)[20]; /* Ptr to a char array of 20 bytes */
struct {
  short mylen; /* Ptr to a variable char array of 30 bytes */
  char mydata[30];
} *myvarchar;
```

**Note:** Parentheses are only allowed when declaring a pointer to a NUL-terminated character array, in which case they are required. If the parentheses were not used, you would be declaring an array of pointers rather than the desired pointer to an array. For example:

```c
char (*a)[10]; /* pointer to a null-terminated char array */
char *a[10]; /* pointer to an array of pointers */
```

- If a host variable is declared as a pointer, then no other host variable can be declared with that same name within the same source file. For example, the second declaration below would be invalid:

```c
char *a[10]; /* pointer to an array of pointers */
```
char *mychar; /* This declaration is valid */
char mychar; /* But this one is invalid */

- When a host variable is referenced within an SQL statement, that host variable must be referenced exactly as declared, with the exception of pointers to NUL-terminated character arrays. For example, the following declaration required parentheses:

```c
char (*mychara)[20]; /* ptr to char array of 20 bytes */
```

However, the parentheses are not allowed when the host variable is referenced in an SQL statement, such as a SELECT:

```c
EXEC SQL SELECT name INTO :*mychara FROM mytable;
```

- Only the asterisk can be used as an operator over a host variable name.
- The maximum length of a host variable name is affected by the number of asterisks specified, as these asterisks are considered part of the name.
- Pointers to structures are not usable as host variables except for variable character structures. Also, pointer fields in structures are not usable as host variables.
- SQL requires that all specified storage for based host variables be allocated. If the storage is not allocated, unpredictable results can occur.

### Using typedef in C and C++ applications that use SQL

You can also use the typedef declarations to define your own identifiers that will be used in place of C type specifiers such as short, float, and double.

The typedef identifiers used to declare host variables must be unique within the program, even if the typedef declarations are in different blocks or procedures. If the program contains BEGIN DECLARE SECTION and END DECLARE SECTION statements, the typedef declarations do not need to be contained with the BEGIN DECLARE SECTION and END DECLARE SECTION. The typedef identifier will be recognized by the SQL precompiler within the BEGIN DECLARE SECTION. The C and C++ precompilers recognize only a subset of typedef declarations, the same as with host variable declarations.

Examples of valid typedef statements:

- Declaring a long typedef and then declaring host variables which reference the typedef.

```c
typedef long int LONG_T;
LONG_T I1, *I2;
```

- The character array length may be specified in either the typedef or on the host variable declaration but not in both.

```c
typedef char NAME_T[30];
typedef char CHAR_T;
CHAR_T name1[30]; /* Valid */
NAME_T name2; /* Valid */
NAME_T name3[10]; /* Not valid for SQL use */
```

- The SQL TYPE IS statement may be used in a typedef.

```c
typedef SQL TYPE IS CLOB(5K) CLOB_T;
CLOB_T clob_var1;
```

- Storage class (auto, extern, static), volatile, or const qualifiers may be specified on the host variable declaration.

```c
typedef short INT_T;
typedef short INT2_T;
static INT_T i1;
volatile INT2_T i2;
```

- typedefs of structures are supported.

```c
typedef _Packed struct {char dept[3];
    char deptname[30];
    long Num_employees;} DEPT_T;
DEPT_T dept_rec;
DEPT_T dept_array[20]; /* use for blocked insert or fetch */
```
Using ILE C compiler external file descriptions in C and C++ applications that use SQL

You can use the C or C++ #pragma mapinc directive with the #include directive to include external file descriptions in your program.

When used with SQL, only a particular format of the #pragma mapinc directive is recognized by the SQL precompiler. If all of the required elements are not specified, the precompiler ignores the directive and does not generate host variable structures. The required elements are:

- Include name
- Externally described file name
- Format name or a list of format names
- Options
- Conversion options

The library name, union name, conversion options, and prefix name are optional. Although typedef statements coded by the user are not recognized by the precompiler, those created by the #pragma mapinc and #include directives are recognized. SQL supports input, output, both, and key values for the options parameter. For the conversion options, the supported values are D, p, z, _P, and 1BYTE_CHAR. These options may be specified in any order except that both D and p cannot be specified. Unions declared using the typedef union created by the #pragma mapinc and #include directive cannot be used as host variables in SQL statements; the members of the unions can be used. Structures that contain the typedef structure cannot be used in SQL statements; the structure declared using the typedef can be used.

To retrieve the definition of the sample table DEPARTMENT described in Db2 for i sample tables in the SQL programming topic collection, you can code the following:

```c
#pragma mapinc("dept","CORPDATA/DEPARTMENT(*ALL)","both")
#include "dept"
CORPDATA_DEPARTMENT_DEPARTMENT_both_t Dept_Structure;
```

A host structure named Dept_Structure is defined with the following elements: DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT. These field names can be used as host variables in SQL statements.

**Note:** DATE, TIME, and TIMESTAMP columns generate character host variable definitions. They are treated by SQL with the same comparison and assignment rules as a DATE, TIME, and TIMESTAMP column. For example, a date host variable can be compared only against a DATE column or a character string that is a valid representation of a date.

If the GRAPHIC or VARGRAPHIC column has a UCS-2 CCSID, the generated host variable will have the UCS-2 CCSID assigned to it. If the GRAPHIC or VARGRAPHIC column has a UTF-16 CCSID, the generated host variable will have the UTF-16 CCSID assigned to it.

Although zoned, binary (with nonzero scale fields), and, optionally, decimal are mapped to character fields in ILE C, SQL will treat these fields as numeric. By using the extended program model (EPM) routines, you can manipulate these fields to convert zoned and packed decimal data. For more information, see the [ILE C/C++ Language Reference](https://www.ibm.com) topic.

CLOB, BLOB, and DBCLOB columns in the external file are ignored. No host variable definition will be generated in the host structure for these types.

**Determining equivalent SQL and C or C++ data types**

The precompiler determines the base SQLTYPE and SQLLEN of host variables based on the table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.
Table 1. C or C++ declarations mapped to typical SQL data types

<table>
<thead>
<tr>
<th>C or C++ data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>long int</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long long int</td>
<td>492</td>
<td>8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>decimal(p, s)</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p, s)</td>
</tr>
<tr>
<td>_Decimal32</td>
<td>996</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>_Decimal64</td>
<td>996</td>
<td>8</td>
<td>DECFLOAT(16)</td>
</tr>
<tr>
<td>_Decimal128</td>
<td>996</td>
<td>16</td>
<td>DECFLOAT(34)</td>
</tr>
<tr>
<td>float</td>
<td>480</td>
<td>4</td>
<td>FLOAT (single precision)</td>
</tr>
<tr>
<td>double</td>
<td>480</td>
<td>8</td>
<td>FLOAT (double precision)</td>
</tr>
<tr>
<td>single-character form</td>
<td>452</td>
<td>1</td>
<td>CHAR(1)</td>
</tr>
<tr>
<td>NUL-terminated character form</td>
<td>460</td>
<td>length</td>
<td>VARCHAR (length - 1)</td>
</tr>
<tr>
<td>VARCHAR structured form</td>
<td>448</td>
<td>length</td>
<td>VARCHAR (length)</td>
</tr>
<tr>
<td>single-graphic form</td>
<td>468</td>
<td>1</td>
<td>GRAPHIC(1)</td>
</tr>
<tr>
<td>NUL-terminated single-graphic form</td>
<td>400°</td>
<td>length</td>
<td>VARGRAPHIC (length - 1)</td>
</tr>
<tr>
<td>VARGRAPHIC structured form</td>
<td>464</td>
<td>length</td>
<td>VARGRAPHIC (length)</td>
</tr>
</tbody>
</table>

You can use the following table to determine the C or C++ data type that is equivalent to a given SQL data type.

Table 2. SQL data types mapped to typical C or C++ declarations

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>C or C++ data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>short int</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>long int</td>
<td></td>
</tr>
<tr>
<td>BIGINT</td>
<td>long long int</td>
<td></td>
</tr>
<tr>
<td>DECIMAL(p, s)</td>
<td>decimal(p, s)</td>
<td>p is a positive integer from 1 to 63, and s is a positive integer from 0 to 63.</td>
</tr>
<tr>
<td>NUMERIC(p, s) or nonzero scale binary</td>
<td>No exact equivalent</td>
<td>Use DECIMAL (p, s).</td>
</tr>
<tr>
<td>DECFLOAT(16)</td>
<td>_Decimal64</td>
<td>Only supported in C.</td>
</tr>
<tr>
<td>DECFLOAT(34)</td>
<td>_Decimal128</td>
<td>Only supported in C.</td>
</tr>
<tr>
<td>FLOAT (single precision)</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>FLOAT (double precision)</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>CHAR(1)</td>
<td>single-character form</td>
<td></td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>No exact equivalent</td>
<td>If n &gt; 1, use NUL-terminated character form.</td>
</tr>
<tr>
<td>SQL data type</td>
<td>C or C++ data type</td>
<td>Notes</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------</td>
<td>-------</td>
</tr>
</tbody>
</table>
| VARCHAR(n)    | NUL-terminated character form | Allow at least \( n+1 \) to accommodate the NUL-terminator. If data can contain character NULs (\(/0\)), use VARCHAR structured form or SQL VARCHAR. 
\( n \) is a positive integer. The maximum value of \( n \) is 32740. |
| CLOB          | None              | Use SQL TYPE IS to declare a CLOB in C or C++. |
| GRAPHIC (1)   | single-graphic form | |
| GRAPHIC (n)   | No exact equivalent | |
| VARGRAPHIC(n) | NUL-terminated graphic form | If \( n > 1 \), use NUL-terminated graphic form. |
|               | VARGRAPHIC structured form | If data can contain graphic NUL values (\(/0/0\)), use VARGRAPHIC structured form. Allow at least \( n+1 \) to accommodate the NUL-terminator. 
\( n \) is a positive integer. The maximum value of \( n \) is 16370. |
| DBCLOB        | None              | Use SQL TYPE IS to declare a DBCLOB in C or C++. |
| BINARY        | None              | Use SQL TYPE IS to declare a BINARY in C or C++. |
| VARBINARY     | None              | Use SQL TYPE IS to declare a VARBINARY in C or C++. |
| BLOB          | None              | Use SQL TYPE IS to declare a BLOB in C or C++. |
| DATE          | NUL-terminated character form | If the format is *USA, *ISO, *JIS, or *EUR, allow at least 11 characters to accommodate the NUL-terminator. If the format is *MDY, *YMD, or *DMY, allow at least 9 characters to accommodate the NUL-terminator. If the format is *JUL, allow at least 7 characters to accommodate the NUL-terminator. |
|               | VARCHAR structured form | If the format is *USA, *ISO, *JIS, or *EUR, allow at least 10 characters. If the format is *MDY, *YMD, or *DMY, allow at least 8 characters. If the format is *JUL, allow at least 6 characters. |
| TIME          | NUL-terminated character form | Allow at least 7 characters (9 to include seconds) to accommodate the NUL-terminator. |
|               | VARCHAR structured form | Allow at least 6 characters; 8 to include seconds. |
### Table 2. SQL data types mapped to typical C or C++ declarations (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>C or C++ data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>NUL-terminated character form</td>
<td>Allow at least 20 characters (33 to include all fractional seconds) to accommodate the NUL-terminator. If n is less than 33, truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td></td>
<td>VARCHAR structured form</td>
<td>Allow at least 19 characters. To include all fractional seconds, allow 32 characters. If the number of characters is less than 32, truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td>XML</td>
<td>None</td>
<td>Use SQL TYPE IS to declare XML in C or C++.</td>
</tr>
<tr>
<td>DATALINK</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a ROWID in C or C++.</td>
</tr>
<tr>
<td>Result set locator</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a result set locator in C or C++.</td>
</tr>
</tbody>
</table>

### Notes on C and C++ variable declaration and usage

Single quotation marks (') and quotation marks ("") have different meanings in C, C++, and SQL.

C and C++ use quotation marks to delimit string constants and single quotation marks to delimit character constants. In contrast, SQL uses quotation marks for delimited identifiers and uses single quotation marks to delimit character string constants. Character data in SQL is distinct from integer data.

### Using indicator variables in C and C++ applications that use SQL

An indicator variable is a two-byte integer (short int).

You can also specify an indicator structure (defined as an array of halfword integer variables) to support a host structure.

Indicator variables are declared in the same way as host variables. The declarations of the two can be mixed in any way that seems appropriate to you.

### Example

Given the statement:

```sql
EXEC SQL FETCH CLS_CURSOR INTO :ClsCd,
    :Day :DayInd,
    :Bgn :BgnInd,
    :End :EndInd;
```

Variables can be declared as follows:

```c
EXEC SQL BEGIN DECLARE SECTION;
char ClsCd[8];
char Bgn[9];
char End[9];
short Day, DayInd, BgnInd, EndInd;
EXEC SQL END DECLARE SECTION;
```

### Related reference:

References to variables

---

Embedded SQL programming 43
An indicator variable is a halfword integer variable used to communicate additional information about its associated host variable.

**Coding SQL statements in COBOL applications**

There are unique application and coding requirements for embedding SQL statements in a COBOL program. In this topic, requirements for host structures and host variables are defined.

The IBM i products support more than one COBOL compiler. The IBM DB2 Query Manager and SQL Development Kit for i licensed program only supports the OPM COBOL and ILE COBOL programming languages.

**Note:** By using the code examples, you agree to the terms of the Code license and disclaimer information on page 190.

**Related concepts:**

- **Writing applications that use SQL** on page 2
- You can create database applications in host languages that use Db2 for i SQL statements and functions.

**Related reference:**

- **Example programs: Using Db2 for i statements** on page 149
- Here is a sample application that shows how to code SQL statements in each of the languages that Db2 for i supports.

**Defining the SQL communication area in COBOL applications that use SQL**

A COBOL program can be written to use the SQL communication area (SQLCA) to check return status for embedded SQL statements, or the program can use the SQL diagnostics area to check return status.

To use the SQL diagnostics area instead of the SQLCA, use the SET OPTION SQL statement with the option SQLCA = *NO.

When using the SQLCA, a COBOL program that contains SQL statements must include one or both of the following:

- An SQLCODE variable declared as PICTURE S9(9) BINARY, PICTURE S9(9) COMP-4, or PICTURE S9(9) COMP.
- An SQLSTATE variable declared as PICTURE X(5).

Or,

- An SQLCA (which contains an SQLCODE and SQLSTATE variable).

The SQLCODE and SQLSTATE values are set by the database manager after each SQL statement is run. An application can check the SQLCODE or SQLSTATE value to determine whether the last SQL statement was successful.

The SQLCA can be coded in a COBOL program either directly or by using the SQL INCLUDE statement. When coding it directly, make sure it is initialized. Using the SQL INCLUDE statement requests the inclusion of a standard declaration:

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```
The SQLCODE, SQLSTATE, and SQLCA variable declarations must appear in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program and can be placed wherever a record description entry can be specified in those sections.

When you use the INCLUDE statement, the SQL COBOL precompiler includes COBOL source statements for the SQLCA:

```cobol
01 SQLCA.
   05 SQLCAID PIC X(8). VALUE '0000000000000000'.
   05 SQLABC   PIC S9(9) BINARY.
   05 SQLCODE  PIC S9(9) BINARY.
   05 SQLERRM. 
      49 SQLERRML PIC S9(4) BINARY.
      49 SQLERRMC PIC X(70).
   05 SQLERRD OCCURS 6 TIMES
      PIC S9(9) BINARY.
   05 SQLWARN. 
      10 SQLWARN0 PIC X.
      10 SQLWARN1 PIC X.
      10 SQLWARN2 PIC X.
      10 SQLWARN3 PIC X.
      10 SQLWARN4 PIC X.
      10 SQLWARN5 PIC X.
      10 SQLWARN6 PIC X.
      10 SQLWARN7 PIC X.
      10 SQLWARN8 PIC X.
      10 SQLWARN9 PIC X.
      10 SQLWARNA PIC X.
   05 SQLSTATE PIC X(5).
```

For ILE COBOL, the SQLCA is declared using the GLOBAL clause. SQLCODE is replaced with SQLCADE when a declaration for SQLCODE is found in the program and the SQLCA is provided by the precompiler. SQLSTATE is replaced with SQLSTATE when a declaration for SQLSTATE is found in the program and the SQLCA is provided by the precompiler.

**Related concepts:**

- [“Using the SQL diagnostics area” on page 7](#)

The SQL diagnostics area is used to keep the returned information for an SQL statement that has been run in a program. It contains all the information that is available to you as an application programmer through the SQLCA.

**Related reference:**

- SQL communication area

### Defining SQL descriptor areas in COBOL applications that use SQL

There are two types of SQL descriptor areas (SQLDAs). One is defined with the ALLOCATE DESCRIPTOR statement. The other is defined using the SQLDA structure. In this topic, only the SQLDA form is discussed.

The following statements can use an SQLDA:

- EXECUTE...USING DESCRIPTOR descriptor-name
- FETCH...USING DESCRIPTOR descriptor-name
- OPEN...USING DESCRIPTOR descriptor-name
- CALL...USING DESCRIPTOR descriptor-name
- DESCRIBE statement-name INTO descriptor-name
- DESCRIBE CURSOR cursor-name INTO descriptor-name
- DESCRIBE INPUT statement-name INTO descriptor-name
- DESCRIBE PROCEDURE procedure-name INTO descriptor-name
DESCRIBE TABLE host-variable INTO descriptor-name

PREPARE statement-name INTO descriptor-name

Unlike the SQLCA, there can be more than one SQLDA in a program. The SQLDA can have any valid name. An SQLDA can be coded in a COBOL program directly or added with the INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard SQLDA declaration:

EXEC SQL INCLUDE SQLDA END-EXEC.

The COBOL declarations included for the SQLDA are:

```
1 SQLDA.
  05 SQLDAID PIC X(8).
  05 SQLDABC PIC S9(9) BINARY.
  05 SQLN PIC S9(4) BINARY.
  05 SQLD PIC S9(4) BINARY.
  05 SQLVAR OCCURS 0 TO 409 TIMES DEPENDING ON SQLD.
    10 SQLVAR.
      15 SQLTYPE PIC S9(4) BINARY.
      15 SQLLEN PIC S9(4) BINARY.
      15 FILLER REDEFINES SQLLEN.
      20 SQLPRECISION PIC X.
      20 SQLSCALE PIC X.
      15 SQLRES PIC X(12).
      15 SQLDATA POINTER.
      15 SQL-RESULT-SET-LOCATOR-R REDEFINES SQLDATA.
      20 SQL-RESULT-SET-LOCATOR PIC S9(18) BINARY.
      15 SQLIND POINTER.
      15 SQL-ROW-CHANGE-SQL-R REDEFINES SQLIND.
      20 SQLD-ROW-CHANGE PIC S9(9) BINARY.
      15 SQLRESULT-SET-ROWS-R PIC S9(9) BINARY.
      15 SQLNAME.
      49 SQLNAMEL PIC S9(4) BINARY.
      49 SQLNAMEC PIC X(30).
    10 SQLVAR2 REDEFINES SQLVAR.
      15 SQLVAR2-RESERVED-1 PIC S9(9) BINARY.
      15 SQLLONGLEN REDEFINEDS SQLVAR2-RESERVED-1
          PIC S9(9) BINARY.
      15 SQLVAR2-RESERVED-2 PIC X(28).
      15 SQLDATALEN POINTER.
      15 SQLDATATYPE-NAME.
        49 SQLDATATYPE_NAMEL PIC S9(4) BINARY.
        49 SQLDATATYPE_NAMEC PIC X(30).
```

Figure 1. INCLUDE SQLDA declarations for COBOL

SQLDA declarations must appear in the WORKING-STORAGE SECTION or LINKAGE SECTION of your program and can be placed wherever a record description entry can be specified in those sections. For ILE COBOL, the SQLDA is declared using the GLOBAL clause.

Dynamic SQL is an advanced programming technique. With dynamic SQL, your program can develop and then run SQL statements while the program is running. A SELECT statement with a variable SELECT list (that is, a list of the data to be returned as part of the query) that runs dynamically requires an SQL descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT.

Related concepts:
Dynamic SQL applications

Related reference:
SQL descriptor area
Embedding SQL statements in COBOL applications that use SQL

SQL statements can be coded in COBOL program sections as in this table.

<table>
<thead>
<tr>
<th>SQL statement</th>
<th>Program section</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN</td>
<td>WORKING-STORAGE SECTION or LINKAGE SECTION</td>
</tr>
<tr>
<td>DECLARE</td>
<td></td>
</tr>
<tr>
<td>SECTION</td>
<td></td>
</tr>
<tr>
<td>END</td>
<td></td>
</tr>
<tr>
<td>DECLARE</td>
<td></td>
</tr>
<tr>
<td>VARIABLE</td>
<td></td>
</tr>
<tr>
<td>DECLARE</td>
<td></td>
</tr>
<tr>
<td>STATEMENT</td>
<td></td>
</tr>
<tr>
<td>WORKING-STORAGE SECTION or LINKAGE SECTION</td>
<td></td>
</tr>
<tr>
<td>INCLUDE</td>
<td></td>
</tr>
<tr>
<td>SQLCA</td>
<td></td>
</tr>
<tr>
<td>INCLUDE</td>
<td></td>
</tr>
<tr>
<td>SQLDA</td>
<td></td>
</tr>
<tr>
<td>INCLUDE member-name</td>
<td>DATA DIVISION or PROCEDURE DIVISION</td>
</tr>
<tr>
<td>Other</td>
<td>PROCEDURE DIVISION</td>
</tr>
</tbody>
</table>

Each SQL statement in a COBOL program must begin with EXEC SQL and end with END-EXEC. If the SQL statement appears between two COBOL statements, the period is optional and might not be appropriate. The EXEC SQL keywords must appear all on one line, but the remainder of the statement can appear on the next and subsequent lines.

Example

An UPDATE statement coded in a COBOL program might be coded as follows:

```cobol
EXEC SQL
UPDATE DEPARTMENT
SET MGRNO = :MGR-NUM
WHERE DEPTNO = :INT-DEPT
END-EXEC.
```

Comments in COBOL applications that use SQL

In addition to SQL comments (--), you can include COBOL comment lines (* or / in column 7) within embedded SQL statements except between the keywords EXEC and SQL. COBOL debugging lines (D in column 7) are treated as comment lines by the precompiler.

Continuation for SQL statements in COBOL applications that use SQL

The line continuation rules for SQL statements are the same as those for other COBOL statements, except that EXEC SQL must be specified within one line.

If you continue a string constant from one line to the next, the first nonblank character in the next line must be either an apostrophe or a quotation mark. If you continue a delimited identifier from one line to the next, the first nonblank character in the next line must be either an apostrophe or a quotation mark.

Constants containing DBCS data can be continued across multiple lines by placing the shift-in character in column 72 of the continued line and the shift-out after the first string delimiter of the continuation line.

This SQL statement has a valid graphic constant of G’<AABBCCDDEEFFGHHIIJJKKK>’. The redundant shifts are removed.
Including code in COBOL applications that use SQL

SQL statements or COBOL host variable declaration statements can be included by embedding the following SQL statement in the source code where the statements are to be embedded.

```cobol
EXEC SQL INCLUDE member-name END-EXEC.
```

Margins in COBOL applications that use SQL

You must code SQL statements in columns 12 through 72. If EXEC SQL starts before the specified margin (that is, before column 12), the SQL precompiler does not recognize the statement.

Sequence numbers in COBOL applications that use SQL

The source statements generated by the SQL precompiler are generated with the same sequence number as the SQL statement.

Names in COBOL applications that use SQL

Any valid COBOL variable name can be used for a host variable and is subject to the following restrictions:

- Do not use host variable names or external entry names that begin with 'SQL', 'RDI', or 'DSN'. These names are reserved for the database manager.

- Using structures that contain FILLER may not work as expected in an SQL statement. It is recommended that all fields within a COBOL structure be named to avoid unexpected results.

COBOL compile-time options in COBOL applications that use SQL

The COBOL PROCESS statement can be used to specify the compile-time options for the COBOL compiler.

Although the PROCESS statement will be recognized by the COBOL compiler when it is called by the precompiler to create the program; the SQL precompiler itself does not recognize the PROCESS statement. Therefore, options that affect the syntax of the COBOL source such as APOST and QUOTE should not be specified in the PROCESS statement. Instead *APOST and *QUOTE should be specified in the OPTION parameter of the CRTSQLCBL and CRTSQLCBLI commands.

Statement labels in COBOL applications that use SQL

Executable SQL statements in the PROCEDURE DIVISION can be preceded by a paragraph name.

WHENEVER statement in COBOL applications that use SQL

The target for the GOTO clause in an SQL WHENEVER statement must be a section name or unqualified paragraph name in the PROCEDURE DIVISION.

Multiple source COBOL programs and the SQL COBOL precompiler

The SQL COBOL precompiler does not support precompiling multiple source programs separated with the PROCESS statement.

Using host variables in COBOL applications that use SQL

All host variables used in SQL statements must be explicitly declared prior to their first use.
The COBOL statements that are used to define the host variables should be preceded by a BEGIN DECLARE SECTION statement and followed by an END DECLARE SECTION statement. If a BEGIN DECLARE SECTION and END DECLARE SECTION are specified, all host variable declarations used in SQL statements must be between the BEGIN DECLARE SECTION and the END DECLARE SECTION statements.

All host variables within an SQL statement must be preceded by a colon (:).

Host variables cannot be records or elements.

To accommodate using dashes within a COBOL host variable name, blanks must precede and follow a minus sign.

**Declaring host variables in COBOL applications that use SQL**

The COBOL precompiler only recognizes a subset of valid COBOL declarations as valid host variable declarations.

**Numeric host variables in COBOL applications that use SQL:**

This figure shows the syntax for valid integer host variable declarations.

**BIGINT and INTEGER and SMALLINT**

![Diagram of valid integer host variable declarations]

Notes:

1. BINARY, COMPUTATIONAL-4, COMP-4, COMPUTATIONAL-5, and COMP-5 are equivalent. A portable application should code BINARY, because COMPUTATIONAL-4, COMP-4, COMPUTATIONAL-5, and COMP-5 are IBM extensions that are not supported in International Organization for Standardization (ISO)/ANSI COBOL. The picture-string associated with these types must have the form S9(i)V9(d) (or S9...9V9...9, with i and d instances of 9). i + d must be less than or equal to 18.
2. level-1 indicates a COBOL level between 2 and 48.
3. COMPUTATIONAL-5, and COMP-5 are only supported for ILE COBOL.

The following figure shows the syntax for valid decimal host variable declarations.

**DECIMAL**

![Diagram of valid decimal host variable declarations]
Notes:
1. PACKED-DECIMAL, COMPUTATIONAL-3, and COMP-3 are equivalent. A portable application should code PACKED-DECIMAL, because COMPUTATIONAL-3 and COMP-3 are IBM extensions that are not supported in ISO/ANS COBOL. The picture-string associated with these types must have the form S9(i)V9(d) (or S9..9V9..9, with i and d instances of 9). i + d must be less than or equal to 63.
2. COMPUTATIONAL and COMP are equivalent. The picture strings associated with these and the data types they represent are product-specific. Therefore, COMP and COMPUTATIONAL should not be used in a portable application. In an OPM COBOL program, the picture-string associated with these types must have the form S9(i)V9(d) (or S9..9V9..9, with i and d instances of 9). i + d must be less than or equal to 63.
3. level-1 indicates a COBOL level between 2 and 48.

The following figure shows the syntax for valid numeric host variable declarations.

**Numeric**

```
01 variable-name PICTURE PIC IS picture-string.
```

**display clause:**

```
DISPLAY IS SIGN LEADING SEPARATE CHARACTER.
```

Notes:
1. The picture-string associated with SIGN LEADING SEPARATE and DISPLAY must have the form S9(i)V9(d) (or S9..9V9..9, with i and d instances of 9). i + d must be less than or equal to 18.
2. level-1 indicates a COBOL level between 2 and 48.

Floating-point host variables in COBOL applications that use SQL:

This figure shows the syntax for valid floating-point host variable declarations. Floating-point host variables are only supported for ILE COBOL.
Floating-point

```
    01    variable-name  USAGE COMPUTATIONAL-1
    -77    level-1

    VALUE IS numeric-constant
```

Notes:
1. COMPUTATIONAL-1 and COMP-1 are equivalent. COMPUTATIONAL-2 and COMP-2 are equivalent.
2. level-1 indicates a COBOL level between 2 and 48.

Character host variables in COBOL applications that use SQL:

There are two valid forms of character host variables: fixed-length strings and varying-length strings.

Fixed-length character strings

```
    01    variable-name  PICTURE IS picture-string
    -77    level-1

    VALUE IS string-constant
```

Notes:
1. The picture-string associated with these forms must be X(m) (or XXX...X, with m instances of X) with 1 ≤ m ≤ 32 766.
2. level-1 indicates a COBOL level between 2 and 48.

Varying-length character strings

```
    01    variable-name  . 49 var-1 PICTURE IS picture-string-1
    level-1
```

Notes:
1. The picture-string associated with these forms must be X(m) (or XXX...X, with m instances of X) with 1 ≤ m ≤ 32 766.
2. level-1 indicates a COBOL level between 2 and 48.
Notes:
1. The *picture-string-1* associated with these forms must be S9(m) or S9..9 with *m* instances of 9. *m* must be from 1 to 4.
   
   Note that the database manager uses the full size of the S9(m) variable even though OPM COBOL only recognizes values up to the specified precision. This can cause data truncation errors when COBOL statements are being run, and might effectively limit the maximum length of variable-length character strings to the specified precision.
2. The *picture-string-2* associated with these forms must be either X(m), or XX..X, with *m* instances of X, and with 1 ≤ *m* ≤ 32 740.
3. *var-1* and *var-2* cannot be used as host variables.
4. *level-1* indicates a COBOL level between 2 and 48.
5. COMPTATIONAL-5 and COMP-5 are only supported for ILE COBOL.

**Graphic host variables in COBOL applications that use SQL:**

Graphic host variables are only supported in ILE COBOL.

There are two valid forms of graphic host variables:
- Fixed-length graphic strings
- Varying-length graphic strings

**Fixed-length graphic strings**
Notes:
1. The picture-string associated with the DISPLAY-1 form must be G(m) (or GGG...G, with m instances of G) or N(m) (or NNN...N, with m instances of N) with 1 ≤ m ≤ 16 383.
2. The picture-string associated with the NATIONAL form must be N(m) (or NNN...N, with m instances of N) with 1 ≤ m ≤ 16 383. NATIONAL is only supported for ILE COBOL. The CCSID is always 1200.
3. You cannot specify a variable that is declared as NATIONAL on the DECLARE VARIABLE statement.
4. level-1 indicates a COBOL level between 2 and 48.

Varying-length graphic strings

Notes:
1. The picture-string-1 associated with these forms must be S9(m) or S9...9 with m instances of 9. m must be from 1 to 4.
   Note that the database manager uses the full size of the S9(m) variable even though OPM COBOL only recognizes values up to the specified precision. This can cause data truncation errors when COBOL statements are being run, and might effectively limit the maximum length of variable-length graphic strings to the specified precision.
2. The picture-string-2 associated with the DISPLAY-1 form must be G(m), GG...G with m instances of G, N(m), or NN...N with m instances of N, and with 1 ≤ m ≤ 16 370.
3. The picture-string-2 associated with the NATIONAL form must be N(m) (or NNN...N, with m instances of N) with 1 ≤ m ≤ 16 383. NATIONAL is only supported for ILE COBOL. The CCSID is always 1200.
   You cannot specify a variable that is declared as NATIONAL on the DECLARE VARIABLE statement.
4. The variables var-1 and var-2 cannot be used as host variables.
5. level-1 indicates a COBOL level between 2 and 48.
6. COMPUTATIONAL-5 and COMP-5 are only supported for ILE COBOL.

**Binary host variables in COBOL applications that use SQL:**

COBOL does not have variables that correspond to the SQL binary data types. To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source member.

**BINARY and VARBINARY**

```sql
01 variable-name IS SQL TYPE IS BINARY (length). VARBINARY BINARY VARYING.
```

**Notes:**
1. For BINARY host variables, the length must be in the range 1 to 32766.
2. For VARBINARY or BINARY VARYING host variables, the length must be in the range 1 to 32740.
3. SQL TYPE IS, BINARY, VARBINARY, and BINARY VARYING can be in mixed case.

**BINARY Example**

The following declaration:

```cobol
01 MY-BINARY SQL TYPE IS BINARY(200).
```

Results in the generation of the following code:

```cobol
01 MY-BINARY PIC X(200).
```

**VARBINARY Example**

The following declaration:

```cobol
01 MY-VARBINARY SQL TYPE IS VARBINARY(250).
```

Results in the generation of the following structure:

```cobol
01 MY-VARBINARY,
   49 MY-VARBINARY-LENGTH PIC 9(5) BINARY.
   49 MY-VARBINARY-DATA PIC X(250).
```

**LOB host variables in COBOL applications that use SQL:**

COBOL does not have variables that correspond to the SQL data types for LOBs (large objects). To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source member.

LOB host variables are only supported in ILE COBOL.

**LOB host variables**

```sql
01 variable-name IS SQL TYPE IS CLOB (lob-length). DBCLOB BLOB.
```
Notes:
1. For BLOB and CLOB, $1 \leq \text{lob-length} \leq 15,728,640$
2. For DBCLOB, $1 \leq \text{lob-length} \leq 7,864,320$
3. SQL TYPE IS, BLOB, CLOB, DBCLOB can be in mixed case.

**CLOB example**

The following declaration:

```sql
01 MY-CLOB SQL TYPE IS CLOB(16384).
```

Results in the generation of the following structure:

```sql
01 MY-CLOB.
   49 MY-CLOB-LENGTH PIC 9(9) BINARY.
   49 MY-CLOB-DATA PIC X(16384).
```

**DBCLOB example**

The following declaration:

```sql
01 MY-DBCLOB SQL TYPE IS DBCLOB(8192).
```

Results in the generation of the following structure:

```sql
01 MY-DBCLOB.
   49 MY-DBCLOB-LENGTH PIC 9(9) BINARY.
   49 MY-DBCLOB-DATA PIC G(8192) DISPLAY-1.
```

**BLOB example**

The following declaration:

```sql
01 MY-BLOB SQL TYPE IS BLOB(16384).
```

Results in the generation of the following structure:

```sql
01 MY-BLOB.
   49 MY-BLOB-LENGTH PIC 9(9) BINARY.
   49 MY-BLOB-DATA PIC X(16384).
```

**LOB locator**

```sql
01—variable-name USAGE IS SQL TYPE IS CLOB-LOCATOR.
```

Notes:
1. SQL TYPE IS, BLOB-LOCATOR, CLOB-LOCATOR, DBCLOB-LOCATOR can be in mixed case.
2. LOB locators cannot be initialized in the SQL TYPE IS statement.

CLOB and DBCLOB locators have similar syntax.

**BLOB locator example**

The following declaration:

```sql
01 MY-LOCATOR SQL TYPE IS BLOB_LOCATOR.
```
Results in the following generation:

```
01 MY-LOCATOR PIC 9(9) BINARY.
```

**LOB file reference variable**

```
01 variable-name
    USAGE IS SQL TYPE IS CLOB-FILE.
```

**Note:** SQL TYPE IS, BLOB-FILE, CLOB-FILE, DBCLOB-FILE can be in mixed case.

**BLOB file reference example**

The following declaration:

```
01 MY-FILE SQL TYPE IS BLOB-FILE.
```

Results in the generation of the following structure:

```
01 MY-FILE.
    49 MY-FILE-NAME-LENGTH PIC 9(9) COMP-5.
    49 MY-FILE-DATA-LENGTH PIC 9(9) COMP-5.
    49 MY-FILE-OPTIONS PIC 9(9) COMP-5.
    49 MY-FILE-NAME PIC X(255).
```

CLOB and DBCLOB file reference variables have similar syntax.

The precompiler generates declarations for the following file option constants. You can use these constants to set the xxx-FILE-OPTIONS variable when you use file reference host variables.

- SQL_FILE_READ (2)
- SQL_FILE_CREATE (8)
- SQL_FILE_OVERWRITE (16)
- SQL_FILE_APPEND (32)

**Related reference:**
[LOB file reference variables](#)

**XML host variables in COBOL applications that use SQL:**

COBOL does not have variables that correspond to the SQL data type for XML. To create host variables that can be used with this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source member.

XML host variables are only supported in ILE COBOL.

**XML host variables**

```
01 variable-name
    USAGE IS SQL TYPE IS XML AS CLOB.  
```

56 IBM i: Embedded SQL programming
Notes:
1. For BLOB and CLOB, \(1 \leq \text{lob-length} \leq 15,728,640\)
2. For DBCLOB, \(1 \leq \text{lob-length} \leq 7,864,320\)
3. SQL TYPE IS, XML AS, BLOB, CLOB, DBCLOB can be in mixed case.
4. The CCSID value for an XML host variable can be explicitly set by the DECLARE VARIABLE statement. Otherwise, the value specified by the SQL.XML_DATA.CCSID QAQQINI option will be used. The default for this QAQQINI option is CCSID 1208.

XML example

The following declaration:

```
01 MY-XML SQL TYPE IS CLOB(5000).
```

Results in the generation of the following structure:

```
01 MY-XML.
   49 MY-XML-LENGTH PIC 9(9) BINARY.
   49 MY-XML-DATA PIC X(5000).
```

XML locator

```
01 variable-name IS USAGE SQL TYPE IS XML-LOCATOR.
```

Notes:
1. SQL TYPE IS, XML AS, XML-LOCATOR can be in mixed case.
2. LOB locators cannot be initialized in the SQL TYPE IS statement.

XML locator example

The following declaration:

```
01 MY-LOCATOR SQL TYPE IS XML-LOCATOR.
```

Results in the following generation:

```
01 MY-LOCATOR PIC 9(9) BINARY.
```

XML file reference variable

```
01 variable-name IS USAGE SQL TYPE IS XML AS CLOB-FILE DBCLOB-FILE BLOB-FILE.
```

Note: SQL TYPE IS, XML AS, BLOB-FILE, CLOB-FILE, DBCLOB-FILE can be in mixed case.
XML file reference example

The following declaration:

01 MY-FILE SQL TYPE IS XML AS CLOB-FILE.

Results in the generation of the following structure:

01 MY-FILE.
   49 MY-FILE-NAME-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-DATA-LENGTH PIC S9(9) COMP-5.
   49 MY-FILE-FILE-OPTIONS PIC S9(9) COMP-5.
   49 MY-FILE-NAME PIC X(255).

The precompiler generates declarations for the following file option constants. You can use these constants to set the xxx-FILE-OPTIONS variable when you use file reference host variables.

- SQL_FILE_READ (2)
- SQL_FILE_CREATE (8)
- SQL_FILE_OVERWRITE (16)
- SQL_FILE_APPEND (32)

Datetime host variables in COBOL applications that use SQL:

This figure shows the syntax for valid date, time, and timestamp host variable declarations. Datetime host variables are supported only for ILE COBOL.

Datetime host variable

```
01 variable-name OF IS FORMAT DATE format-options IS SIZE integer
```

Notes:
1. level-1 indicates a COBOL level between 2 and 48.
2. format-options indicates valid datetime options that are supported by the COBOL compiler. See the ILE COBOL Language Reference manual for details.
3. The value for the TIMESTAMP SIZE integer must be 19 to 32.

ROWID host variables in COBOL applications that use SQL:

COBOL does not have a variable that corresponds to the SQL data type ROWID. To create host variables that can be used with this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source member.

ROWID

```
01 variable-name SQL TYPE IS ROWID.
```

Note: SQL TYPE IS ROWID can be in mixed case.
The following declaration:

```
01 MY-ROWID SQL TYPE IS ROWID.
```

Results in the generation of the following structure:

```
01 MY-ROWID.
   49 MY-ROWID-LENGTH PIC 9(2) BINARY.
   49 MY-ROWID-DATA PIC X(40).
```

**Result set locator host variables in COBOL applications that use SQL:**

COBOL does not have a variable that corresponds to the SQL result set locator data type. To create host variables that can be used with this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a COBOL language structure in the output source member.

**Result set locator**

```
01 variable-name IS USAGE SQL TYPE IS RESULT_SET_LOCATOR.
```

**Note:** SQL TYPE IS RESULT_SET_LOCATOR can be in mixed case.

**Result set locator example**

The following declaration:

```
01 RSLOC1 SQL TYPE IS RESULT_SET_LOCATOR.
```

Results in the generation of the following structure:

```
01 RSLOC1 PIC 9(18) BINARY.
```

**Using host structures in COBOL applications that use SQL**

A host structure is a named set of host variables that is defined in your program’s DATA DIVISION.

Host structures have a maximum of two levels, even though the host structure might itself occur within a multilevel structure. An exception is the declaration of a varying-length character string, which requires another level that must be level 49.

A host structure name can be a group name whose subordinate levels name basic data items. For example:

```
01 A
   02 B
      03 C1 PICTURE ...
      03 C2 PICTURE ...
```

In this example, B is the name of a host structure consisting of the basic items C1 and C2.

When writing an SQL statement using a qualified host variable name (for example, to identify a field within a structure), use the name of the structure followed by a period and the name of the field. For example, specify B.C1 rather than C1 OF B or C1 IN B. However, this guideline applies only to qualified names within SQL statements; you cannot use this technique for writing qualified names in COBOL statements.

A host structure is considered complete if any of the following items are found:
• A COBOL item that must begin in area A
• Any SQL statement (except SQL INCLUDE)

After the host structure is defined, you can refer to it in an SQL statement instead of listing the several host variables (that is, the names of the data items that comprise the host structure).

For example, you can retrieve all column values from selected rows of the table CORPDATA.EMPLOYEE with:

```cobol
01 PEMPL.
   10 EMPNO    PIC X(6).
   10 FIRSTNME.
      49 FIRSTNME-LEN PIC S9(4) USAGE BINARY.
      49 FIRSTNME-TEXT PIC X(12).
   10 MIDINIT  PIC X(1).
   10 LASTNAME.
      49 LASTNAME-LEN PIC S9(4) USAGE BINARY.
      49 LASTNAME-TEXT PIC X(15).
   10 WORKDEPT PIC X(3).
...
MOVE "000220" TO EMPNO.
...
EXEC SQL
   SELECT * INTO :PEMPL 
   FROM CORPDATA.EMPLOYEE 
   WHERE EMPNO = :EMPNO 
END-EXEC.
```

Notice that in the declaration of PEMPL, two varying-length string elements are included in the structure: FIRSTNME and LASTNAME.

**Host structure in COBOL applications that use SQL**

This figure shows the syntax for the valid host structure.

```
►►level-1—variable-name—►

►level-2—var-1—PICTURE►IS◄picture-string—usage-clause—►
  —floating-point—
  —varchar-string—
  —vargraphic-string—
  —binary—
  —lob—
  —xml—
  —datetime—
  —rowid—
  —result-set-locator—

usage-clause:
```
**display-clause:**

- `DISPLAY`  
- `IS`  
- `SIGN`  
- `LEADING`  
- `SEPARATE`  
- `CHARACTER`

**floating-point:**

- `USAGE`  
- `IS`  
- `COMPUTATIONAL-1`  
- `COMP-1`  
- `COMPUTATIONAL-2`  
- `COMP-2`  
- `VALUE`  
- `IS`  
- `constant`

**varchar-string:**

- `49`  
- `var-2`  
- `PICTURE`  
- `IS`  
- `picture-string-1`

- `USAGE`  
- `IS`  
- `BINARY`  
- `COMPUTATIONAL-4`  
- `COMP-4`  
- `COMPUTATIONAL-5`  
- `COMP-5`  
- `VALUE`  
- `IS`  
- `numeric-constant`

- `49`  
- `var-3`  
- `PICTURE`  
- `IS`  
- `picture-string-2`
Notes:
1. level-1 indicates a COBOL level between 1 and 47.
2. level-2 indicates a COBOL level between 2 and 48 where level-2 > level-1.
3. Graphic host variables, LOB host variables, XML host variables, floating-point host variables, and COMP-5 host variables are only supported for ILE COBOL.
4. For details on declaring numeric, character, graphic, binary, LOB, XML, ROWID, and result set locator host variables, see the notes under numeric-host variables, character-host variables, graphic-host variables, binary host variables, LOB host variables, XML host variables, ROWID, and result set locator host variables.
5. The variable format-options indicates valid datetime options that are supported by the COBOL compiler. See the ILE COBOL Language Reference manual for details.
6. The value for the TIMESTAMP SIZE integer must be 19 to 32.

Host structure indicator array in COBOL applications that use SQL
This figure shows the syntax for valid host structure indicator array declarations.

Host structure indicator array
Notes:
1. Dimension must be an integer between 1 and 32,767.
2. level-1 must be an integer between 2 and 48.
3. BINARY, COMPUTATIONAL-4, COMP-4, COMPUTATIONAL-5, and COMP-5 are equivalent. A portable application should code BINARY because COMPUTATIONAL-4, COMP-4, COMPUTATIONAL-5, and COMP-5 are IBM extensions that are not supported in ISO/ANSI COBOL. The picture-string associated with these types must have the form S9(i) (or S9...9, with i instances of 9). i must be less than or equal to 4.

Using host structure arrays in COBOL applications that use SQL
A host structure array is a named set of host variables that is defined in the program's Data Division and has an OCCURS clause.

Host structure arrays have a maximum of two levels, even though the host structure can occur within a multiple level structure. A varying-length string requires another level, level 49. A host structure array name can be a group name whose subordinate levels name basic data items.

In these examples, the following are true:
• All members in B-ARRAY must be valid.
• B-ARRAY cannot be qualified.
• B-ARRAY can only be used on the blocked form of the FETCH and INSERT statements.
• B-ARRAY is the name of an array of host structures containing items C1-VAR and C2-VAR.
• The SYNCHRONIZED attribute must not be specified.
• C1-VAR and C2-VAR are not valid host variables in any SQL statement. A structure cannot contain an intermediate level structure.

To retrieve 10 rows from the CORPDATA.DEPARTMENT table, use the following example:

EXEC SQL
DECLARE C1 CURSOR FOR
    SELECT *
    FROM CORPDATA.DEPARTMENT
END-EXEC.
EXEC SQL
  FETCH C1 FOR 10 ROWS INTO :DEPT :IND-ARRAY
END-EXEC.

Host structure array in COBOL applications that use SQL
These figures show the syntax for valid host structure array declarations.

level-1—variable-name—OCCURS—dimension—

level-2—var-1
  PICTURE picture-string-1—usage-clause—

usage-clause:

  USAGE IS constant

  VALUE IS constant

  DISPLAY display-clause

  NATIONAL

display-clause:

  DISPLAY SIGN IS CHARACTER

  LEADING SEPARATE

floating-point:
binary:

lob:

xml:

datetime:
rowid:

```
| SQL TYPE IS ROWID |
```

result set locator:

```
| SQL TYPE IS RESULT_SET_LOCATOR |
```

Notes:
1. `level-1` indicates a COBOL level between 2 and 47.
2. `level-2` indicates a COBOL level between 3 and 48 where `level-2 > level-1`.
3. Graphic host variables, LOB host variables, XML host variables, and floating-point host variables are only supported for ILE COBOL.
4. For details on declaring numeric, character, graphic, binary LOB, XML, ROWID, and result set locator host variables, see the notes under numeric-host variables, character-host variables, graphic-host variables, binary host variables, LOB, XML, ROWID, and result set locator host variables.
5. Dimension must be an integer constant between 1 and 32,767.
6. The variable `format-options` indicates valid datetime options that are supported by the COBOL compiler. See the [ILE COBOL Language Reference](#) manual for details.
7. The value for the `TIMESTAMP SIZE` integer must be 19 to 32.

Host array indicator structure in COBOL applications that use SQL

This figure shows the valid syntax for host structure array indicators.

```
TIMES

IS

PICTURE

picture-string

USAGE

BINARY

COMP-4

COMP-5

IS

constant
```

Notes:
1. `level-1` indicates a COBOL level between 2 and 48.
2. `level-2` indicates a COBOL level between 3 and 48 where `level-2 > level-1`.
3. Dimension must be an integer constant between 1 and 32,767.
4. BINARY, COMPUTATIONAL-4, COMP-4, COMPUTATIONAL-5, and COMP-5 are equivalent. A portable application should code BINARY, because COMPUTATIONAL-4, COMP-4, COMPUTATIONAL-5, and COMP-5 are IBM extensions that are not supported in ISO/ANSI COBOL. The `picture-string` associated with these types must have the form S9(i) (or S9..9, with i instances of 9). i must be less than or equal to 4.
Using external file descriptions in COBOL applications that use SQL

SQL uses the COPY DD-format-name, COPY DD-ALL-FORMATS, COPY DDS-format-name, COPY DDR-format-name, COPY DDR-ALL-FORMATS, COPY DDSR-format-name, COPY DDS-ALL-FORMATS, and COPY DDSR-ALL-FORMATS to retrieve host variables from the file definitions.

If the REPLACING option is specified, only complete name replacing is done. Var-1 is compared against the format name and the field name. If they are equal, var-2 is used as the new name.

**Note:** You cannot retrieve host variables from file definitions that have field names which are COBOL reserved words. You must place the COPY DDx-format statement within a COBOL host structure.

To retrieve the definition of the sample table DEPARTMENT described in Db2 for i sample tables in the SQL programming concepts topic collection, you can code the following:

```
01 DEPARTMENT-STRUCTURE.
   COPY DDS-ALL-FORMATS OF DEPARTMENT.
```

A host structure named DEPARTMENT-STRUCTURE is defined with an 05 level field named DEPARTMENT-RECORD that contains four 06 level fields named DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT. These field names can be used as host variables in SQL statements.

For more information about the COBOL COPY verb, see the ILE COBOL Language Reference and COBOL/400 User’s Guide at IBM Publications Center.

CLOB, BLOB, and DBCLOB columns in the external file are ignored. No host variable definition will be generated in the host structure for these types.

Using external file descriptions for host structure arrays in COBOL applications that use SQL

Because COBOL creates an extra level when including externally described data, the OCCURS clause must be placed on the preceding 04 level. The structure cannot contain any additional declares at the 05 level.

If the file contains fields that are generated as FILLER, the structure cannot be used as a host structure array.

For device files, if INDARA is not specified and the file contains indicators, the declaration cannot be used as a host structure array. The indicator area is included in the generated structure and causes the storage for records to not be contiguous.

For example, the following shows how to use COPY–DDS to generate a host structure array and fetch 10 rows into the host structure array:

```
01 DEPT.
   04 DEPT-ARRAY OCCURS 10 TIMES.
   COPY DDS-ALL-FORMATS OF DEPARTMENT.
   ...

EXEC SQL DECLARE C1 CURSOR FOR
   SELECT * FROM CORPDATA.DEPARTMENT
END EXEC.

EXEC SQL OPEN C1
END-EXEC.

EXEC SQL FETCH C1 FOR 10 ROWS INTO :DEPARTMENT
END-EXEC.
```
Note: DATE, TIME, and TIMESTAMP columns will generate character host variable definitions that are treated by SQL with the same comparison and assignment rules as the DATE, TIME, or TIMESTAMP column. For example, a date host variable can only be compared against a DATE column or a string which is a valid representation of a date.

Although GRAPHIC and VARGRAPHIC are mapped to character variables in OPM COBOL, SQL considers these GRAPHIC and VARGRAPHIC variables. If the GRAPHIC or VARGRAPHIC column has a UCS-2 CCSID, the generated host variable has the UCS-2 CCSID assigned to it. If the GRAPHIC or VARGRAPHIC column has a UTF-16 CCSID, the generated host variable has the UTF-16 CCSID assigned to it.

Determining equivalent SQL and COBOL data types

The precompiler determines the base SQLTYPE and SQLLEN of host variables based on this table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.

Table 3. COBOL declarations mapped to typical SQL data types

<table>
<thead>
<tr>
<th>COBOL data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S9(i)V9(d) COMP-3 or S9(i)V9(d) COMP or S9(i)V9(d) PACKED-DECIMAL</td>
<td>484</td>
<td>i+d in byte 1, d in byte 2</td>
<td>DECIMAL(i+d,d)</td>
</tr>
<tr>
<td>S9(i)V9(d) DISPLAY SIGN LEADING SEPARATE</td>
<td>504</td>
<td>i+d in byte 1, d in byte 2</td>
<td>No exact equivalent use DECIMAL(i+d,d) or NUMERIC (i+d,d)</td>
</tr>
<tr>
<td>S9(i)V9(d)DISPLAY</td>
<td>488</td>
<td>i+d in byte 1, d in byte 2</td>
<td>NUMERIC(i+d,d)</td>
</tr>
<tr>
<td>S9(i) BINARY or S9(i) COMP-4 or S9(i) COMP-5 where i is from 1 to 4</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>S9(i) BINARY or S9(i) COMP-4 or S9(i) COMP-5 where i is from 5 to 9</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>S9(i) BINARY or S9(i) COMP-4 or S9(i) COMP-5 where i is from 10 to 18.</td>
<td>492</td>
<td>8</td>
<td>BIGINT</td>
</tr>
<tr>
<td>Not supported by OPM COBOL.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S9(i)V9(d) BINARY or S9(i)V9(d) COMP-4 or S9(i)V9(d) COMP-5 where i+d ≤ 4</td>
<td>500</td>
<td>i+d in byte 1, d in byte 2</td>
<td>No exact equivalent use DECIMAL(i+d,d) or NUMERIC (i+d,d)</td>
</tr>
<tr>
<td>S9(i)V9(d) BINARY or S9(i)V9(d) COMP-4 or S9(i)V9(d) COMP-5 where 4 &lt; i+d ≤ 9</td>
<td>496</td>
<td>i+d in byte 1, d in byte 2</td>
<td>No exact equivalent use DECIMAL(i+d,d) or NUMERIC (i+d,d)</td>
</tr>
<tr>
<td>COMP-1</td>
<td>480</td>
<td>4</td>
<td>FLOAT(single precision)</td>
</tr>
<tr>
<td>Not supported by OPM COBOL.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMP-2</td>
<td>480</td>
<td>8</td>
<td>FLOAT(double precision)</td>
</tr>
<tr>
<td>Not supported by OPM COBOL.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-length character data</td>
<td>452</td>
<td>m</td>
<td>CHAR(m)</td>
</tr>
<tr>
<td>Varying-length character data</td>
<td>448</td>
<td>m</td>
<td>VARCHAR(m)</td>
</tr>
<tr>
<td>Fixed-length graphic data</td>
<td>468</td>
<td>m</td>
<td>GRAPHIC(m)</td>
</tr>
<tr>
<td>Not supported by OPM COBOL.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3. COBOL declarations mapped to typical SQL data types (continued)

<table>
<thead>
<tr>
<th>COBOL data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Varying-length graphic data</td>
<td></td>
<td>464</td>
<td>m</td>
</tr>
<tr>
<td>Not supported by OPM COBOL.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td></td>
<td>384</td>
<td></td>
</tr>
<tr>
<td>Not supported by OPM COBOL.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td></td>
<td>388</td>
<td></td>
</tr>
<tr>
<td>Not supported by OPM COBOL.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP SIZE n</td>
<td></td>
<td>392</td>
<td>n</td>
</tr>
<tr>
<td>Not supported by OPM COBOL.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following table can be used to determine the COBOL data type that is equivalent to a given SQL data type.

Table 4. SQL data types mapped to typical COBOL declarations

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>COBOL data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>S9(m) COMP-4 or S9(m) COMP-5</td>
<td>m is from 1 to 4</td>
</tr>
<tr>
<td>INTEGER</td>
<td>S9(m) COMP-4 or S9(m) COMP-5</td>
<td>m is from 5 to 9</td>
</tr>
<tr>
<td>BIGINT</td>
<td>S9(m) COMP-4 or S9(m) COMP-5 for ILE COBOL.</td>
<td>m is from 10 to 18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not supported by OPM COBOL.</td>
</tr>
<tr>
<td>DECIMAL(p,s)</td>
<td>If p&lt;64: S9(p-s)V9(s) PACKED-DECIMAL or S9(p-s)V9(s) COMP or S9(p-s)V9(s) COMP-3. If p&gt;63: Not supported</td>
<td>p is precision; s is scale. 0&lt;=s&lt;=p&lt;=63. If s=0, use S9(p) or S9(p)V. If s=p, use SV9(s).</td>
</tr>
<tr>
<td>NUMERIC(p,s)</td>
<td>If p&lt;19: S9(p-s)V9(s) DISPLAY If p&gt;18: Not supported</td>
<td>p is precision; s is scale. 0&lt;=s&lt;=p&lt;=18. If s=0, use S9(p) or S9(p)V. If s=p, use SV9(s).</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>FLOAT(single precision)</td>
<td>COMP-1 for ILE COBOL.</td>
<td></td>
</tr>
<tr>
<td>FLOAT(double precision)</td>
<td>COMP-2 for ILE COBOL.</td>
<td></td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>Fixed-length character string</td>
<td>32766≥n≥1</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>Varying-length character string</td>
<td>32740≥n≥1</td>
</tr>
<tr>
<td>CLOB</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a CLOB for ILE COBOL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not supported by OPM COBOL.</td>
</tr>
</tbody>
</table>
Table 4. SQL data types mapped to typical COBOL declarations (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>COBOL data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>GRAPHIC(n)</td>
<td>Fixed-length graphic string for ILE COBOL.</td>
<td>16383≥n≥1</td>
</tr>
<tr>
<td></td>
<td>Not supported by OPM COBOL.</td>
<td></td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>Varying-length graphic string for ILE COBOL.</td>
<td>16370≥n≥1</td>
</tr>
<tr>
<td></td>
<td>Not supported by OPM COBOL.</td>
<td></td>
</tr>
<tr>
<td>DBCLOB</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a DBCLOB for ILE COBOL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not supported by OPM COBOL.</td>
</tr>
<tr>
<td>BINARY</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a BINARY.</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a VARBINARY.</td>
</tr>
<tr>
<td>BLOB</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a BLOB.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not supported by OPM COBOL.</td>
</tr>
<tr>
<td>DATE</td>
<td>Fixed-length character string or DATE for ILE COBOL.</td>
<td>If the format is *USA, *JIS, *EUR, or *ISO, allow at least 10 characters. If the format is *YMD, *DMY, or *MDY, allow at least 8 characters. If the format is *JUL, allow at least 6 characters.</td>
</tr>
<tr>
<td>TIME</td>
<td>Fixed-length character string or TIME for ILE COBOL.</td>
<td>Allow at least 6 characters; 8 to include seconds.</td>
</tr>
<tr>
<td>TIMESTAMP(n)</td>
<td>Fixed-length character string or TIMESTAMP(19) when n = 0, TIMESTAMP(20+n) when n &gt; 0 for ILE COBOL.</td>
<td>To include all fractional seconds, n must be 12. If n is less than 12, truncation occurs on the fractional seconds part.</td>
</tr>
<tr>
<td>XML</td>
<td>None</td>
<td>Use SQL TYPE IS to declare an XML.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not supported by OPM COBOL.</td>
</tr>
<tr>
<td>DATALINK</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a ROWID.</td>
</tr>
<tr>
<td>Result set locator</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a result set locator.</td>
</tr>
</tbody>
</table>

Notes on COBOL variable declaration and usage
Any level 77 data description entry can be followed by one or more REDEFINES entries. However, the names in these entries cannot be used in SQL statements.

Unpredictable results may occur when a structure contains levels defined below a FILLER item.

The COBOL declarations for SMALLINT, INTEGER, and BIGINT data types are expressed as a number of decimal digits. The database manager uses the full size of the integers and can place larger values in the host variable than would be allowed in the specified number of digits in the COBOL declaration. However, this can cause data truncation or size errors when COBOL statements are being run. Ensure that the size of numbers in your application is within the declared number of digits.
Using indicator variables in COBOL applications that use SQL

An indicator variable is a two-byte integer (PIC S9(m) USAGE BINARY, where m is from 1 to 4).

You can also specify an indicator structure (defined as an array of halfword integer variables) to support a host structure.

Indicator variables are declared in the same way as host variables, and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

**Example**

Given the statement:

```sql
EXEC SQL FETCH CLS_CURSOR INTO :CLS-CD,
    :NUMDAY :NUMDAY-IND,
    :BGN :BGN-IND,
    :ENDCLS :ENDCLS-IND
END-EXEC.
```

The variables can be declared as follows:

```sql
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
77 CLS-CD PIC X(7).
77 NUMDAY PIC S9(4) BINARY.
77 BGN PIC X(8).
77 ENDCLS PIC X(8).
77 NUMDAY-IND PIC S9(4) BINARY.
77 BGN-IND PIC S9(4) BINARY.
77 ENDCLS-IND PIC S9(4) BINARY.
EXEC SQL END DECLARE SECTION END-EXEC.
```

Related reference:

- References to variables
- “Indicator variables in applications that use SQL” on page 4

An indicator variable is a halfword integer variable used to communicate additional information about its associated host variable.

**Coding SQL statements in PL/I applications**

There are some unique application and coding requirements for embedding SQL statements in a PL/I program. In this topic, requirements for host structures and host variables are defined.

**Note:** By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 190.

**Related concepts:**
- “Writing applications that use SQL” on page 2

You can create database applications in host languages that use Db2 for i SQL statements and functions.

**Related reference:**
- “Example programs: Using Db2 for i statements” on page 149
- “Example: SQL statements in PL/I programs” on page 166

Here is a sample application that shows how to code SQL statements in each of the languages that Db2 for i supports.

This example program is written in the PL/I programming language.
Defining the SQL communication area in PL/I applications that use SQL

A PL/I program that contains SQL statements must include one or both of these fields.

- An SQLCODE variable declared as FIXED BINARY(31)
- An SQLSTATE variable declared as CHAR(5)

Or,

- An SQLCA (which contains an SQLCODE and SQLSTATE variable).

The SQLCODE and SQLSTATE values are set by the database manager after each SQL statement is run. An application can check the SQLCODE or SQLSTATE value to determine whether the last SQL statement was successful.

The SQLCA can be coded in a PL/I program either directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard SQLCA declaration:

```sql
EXEC SQL INCLUDE SQLCA ;
```

The scope of the SQLCODE, SQLSTATE, and SQLCA variables must include the scope of all SQL statements in the program.

The included PL/I source statements for the SQLCA are:

```pli
DCL 1 SQLCA,
   2 SQLCAID CHAR(8),
   2 SQLCAID FIXED(31) BINARY,
   2 SQLCODE FIXED(31) BINARY,
   2 SQLERRM CHAR(70) VAR,
   2 SQLERRP CHAR(8),
   2 SQLERROR FIXED(31) BINARY,
   2 SQLWARNM CHAR(8),
   3 SQLWARN0 CHAR(1),
   3 SQLWARN1 CHAR(1),
   3 SQLWARN2 CHAR(1),
   3 SQLWARN3 CHAR(1),
   3 SQLWARN4 CHAR(1),
   3 SQLWARN5 CHAR(1),
   3 SQLWARN6 CHAR(1),
   3 SQLWARN7 CHAR(1),
   3 SQLWARN8 CHAR(1),
   3 SQLWARN9 CHAR(1),
   3 SQLWARN10 CHAR(1),
   2 SQLSTATE CHAR(5);
```

SQLCODE is replaced with SQLCADE when a declare for SQLCODE is found in the program and the SQLCA is provided by the precompiler. SQLSTATE is replaced with SQLSTOTE when a declare for SQLSTATE is found in the program and the SQLCA is provided by the precompiler.

Related reference:

SQL communication area

Defining SQL descriptor areas in PL/I applications that use SQL

There are two types of SQL descriptor areas. One is defined with the ALLOCATE DESCRIPTOR statement. The other is defined using the SQLDA structure. In this topic, only the SQLDA form is discussed.

The following statements can use an SQLDA:

- EXECUTE...USING DESCRIPTOR descriptor-name
- FETCH...USING DESCRIPTOR descriptor-name

74 | IBM i: Embedded SQL programming
• OPEN...USING DESCRIPTOR descriptor-name
• CALL...USING DESCRIPTOR descriptor-name
• DESCRIBE statement-name INTO descriptor-name
• DESCRIBE CURSOR cursor-name INTO descriptor-name
• DESCRIBE INPUT statement-name INTO descriptor-name
• DESCRIBE PROCEDURE procedure-name INTO descriptor-name
• DESCRIBE TABLE host-variable INTO descriptor-name
• PREPARE statement-name INTO descriptor-name

Unlike the SQLCA, there can be more than one SQLDA in a program, and an SQLDA can have any valid name. An SQLDA can be coded in a PL/I program either program directly or by using the SQL INCLUDE statement. Using the SQL INCLUDE statement requests the inclusion of a standard SQLDA declaration:

EXEC SQL INCLUDE SQLDA ;

The included PL/I source statements for the SQLDA are:

DCL 1 SQLDA BASED(SQLDAPTR),
  2 SQLDAID CHAR(8),
  2 SQLDABC FIXED(31) BINARY,
  2 SQLN FIXED(15) BINARY,
  2 SQLD FIXED(15) BINARY,
  2 SQLVAR(99),
  3 SQLTYPE FIXED(15) BINARY,
  3 SQLLEN FIXED(15) BINARY,
  3 SQLRES CHAR(12),
  3 SQLDATA PTR,
  3 SQLIND PTR,
  3 SQLNAME CHAR(30) VAR;
DCL SQLDAPTR PTR;

Dynamic SQL is an advanced programming technique. With dynamic SQL, your program can develop and then run SQL statements while the program is running. A SELECT statement with a variable SELECT list (that is, a list of the data to be returned as part of the query) that runs dynamically requires an SQL descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT.

Related concepts:
Dynamic SQL applications

Related reference:
SQL descriptor area

Embedding SQL statements in PL/I applications that use SQL

The first statement of the PL/I program must be a PROCEDURE statement. SQL statements can be coded in a PL/I program wherever executable statements can appear.

Each SQL statement in a PL/I program must begin with EXEC SQL and end with a semicolon (;). The key words EXEC SQL must appear all on one line, but the remainder of the statement can appear on the next and subsequent lines.

Example: Embedding SQL statements in PL/I applications that use SQL

You can code an UPDATE statement in a PL/I program as in this example.

EXEC SQL UPDATE DEPARTMENT
  SET MGRNO = :MGR_NUM
  WHERE DEPTNO = :INT_DEPT ;

Embedded SQL programming 75
Comments in PL/I applications that use SQL
In addition to SQL comments (--), you can include PL/I comments (/...*/ in embedded SQL statements wherever a blank is allowed, except between the keywords EXEC and SQL.

Continuation for SQL statements in PL/I applications that use SQL
The line continuation rules for SQL statements are the same as those for other PL/I statements, except that EXEC SQL must be specified within one line.

Constants containing DBCS data can be continued across multiple lines by placing the shift-in and shift-out characters outside of the margins. This example assumes margins of 2 and 72. This SQL statement has a valid graphic constant of G’<AABBCCDEFFGHHIIJJKKK>’.

EXEC SQL SELECT * FROM GRAPHTAB WHERE GRAPHCOL = G’<AABBCCDDD>EEFFGHHIIJJKKK’;

Including code in PL/I applications that use SQL
SQL statements or PL/I host variable declaration statements can be included by placing the following SQL statement at the point in the source code where the statements are to be embedded.

EXEC SQL INCLUDE member-name;

Margins in PL/I applications that use SQL
You must code SQL statements within the margins specified by the MARGINS parameter on the CRTSQLPLI command. If EXEC SQL does not start within the specified margins, the SQL precompiler will not recognize the SQL statement.

Related concepts:
“CL command descriptions for host language precompilers” on page 188
The IBM DB2 Query Manager and SQL Development Kit for i licensed program provides commands for precompiling programs coded in these programming languages.

Names in PL/I applications that use SQL
Any valid PL/I variable name can be used for a host variable and is subject to these restrictions.

Do not use host variable names or external entry names that begin with 'SQL', 'RDI', or 'DSN'. These names are reserved for the database manager.

Statement labels in PL/I applications that use SQL
All executable SQL statements, like PL/I statements, can have a label prefix.

WHENEVER statement in PL/I applications that use SQL
The target for the GOTO clause in an SQL WHENEVER statement must be a label in the PL/I source code and must be within the scope of any SQL statements affected by the WHENEVER statement.

Using host variables in PL/I applications that use SQL
All host variables used in SQL statements must be explicitly declared.

The PL/I statements that are used to define the host variables should be preceded by a BEGIN DECLARE SECTION statement and followed by an END DECLARE SECTION statement. If a BEGIN DECLARE SECTION and END DECLARE SECTION are specified, all host variable declarations used in SQL statements must be between the BEGIN DECLARE SECTION and the END DECLARE SECTION statements.
All host variables within an SQL statement must be preceded by a colon (:).

The names of host variables must be unique within the program, even if the host variables are in different blocks or procedures.

An SQL statement that uses a host variable must be within the scope of the statement in which the variable was declared.

Host variables must be scalar variables. They cannot be elements of an array.

**Declaring host variables in PL/I applications that use SQL**
The PL/I precompiler only recognizes a subset of valid PL/I declarations as valid host variable declarations.

Only the names and data attributes of the variables are used by the precompilers; the alignment, scope, and storage attributes are ignored. Even though alignment, scope, and storage are ignored, there are some restrictions on their use that, if ignored, may result in problems when compiling PL/I source code that is created by the precompiler. These restrictions are:

- A declaration with the EXTERNAL scope attribute and the STATIC storage attribute must also have the INITIAL storage attribute.
- If the BASED storage attribute is coded, it must be followed by a PL/I element-locator-expression.

**Numeric-host variables in PL/I applications that use SQL:**

This figure shows the syntax for valid scalar numeric-host variable declarations.

**Numeric**

```
DECLARE DCL variable-name,
     (variable-name)
```

```
BINARY FIXED BIN (precision)
DECIMAL FIXED DEC (precision)
```

```
FLOAT (precision, scale)
```

```
PICTURE picture-string;
```

**Notes:**

1. (BINARY, BIN, DECIMAL, or DEC) and (FIXED or FLOAT) and (precision, scale) can be specified in any order.
2. A picture-string in the form '9...9V9...R' indicates a numeric host variable. The R is required. The optional V indicates the implied decimal point.
3. A picture-string in the form 'S9...9V9...9' indicates a sign leading separate host variable. The S is required. The optional V indicates the implied decimal point.
Character-host variables in PL/I applications that use SQL:

This figure shows the syntax for valid scalar character-host variables.

Character

\[
\text{DECLARE}\ DCL \text{variable-name}, \left(\text{variable-name}\right) \text{CHARACTER} \text{CHAR} \left(\text{length}\right) \text{VARYING}\ VAR; \\
\text{Alignment and/or Scope and/or Storage}
\]

Notes:
1. The variable \textit{length} must be an integer constant not greater than 32766 if VARYING or VAR is not specified.
2. If VARYING or VAR is specified, \textit{length} must be a constant no greater than 32740.

Binary host variables in PL/I applications that use SQL:

PL/I does not have variables that correspond to the SQL binary data types. To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a PL/I language structure in the output source member.

\textbf{BINARY and VARBINARY}

\[
\text{DECLARE}\ DCL \text{variable-name} \text{SQL TYPE IS BINARY VARBINARY} \left(\text{length}\right); \\
\text{alignment and/or scope and/or storage}
\]

Notes:
1. For BINARY host variables, the length must be in the range 1 to 32766.
2. For VARBINARY and BINARY VARYING host variables, the length must be in the range 1 to 32740.
3. SQL TYPE IS, BINARY, VARBINARY, BINARY VARYING can be in mixed case.

\textbf{BINARY example}

The following declaration:
\[\text{DCL MY\_BINARY SQL TYPE IS BINARY(100);}\]

Results in the generation of the following code:
\[\text{DCL MY\_BINARY CHARACTER(100);}\]

\textbf{VARBINARY example}

The following declaration:
\[\text{DCL MY\_VARBINARY SQL TYPE IS VARBINARY(250);}\]

Results in the generation of the following code:
\[\text{DCL MY\_VARBINARY CHARACTER(250) VARYING;}\]
LOB host variables in PL/I applications that use SQL:

PL/I does not have variables that correspond to the SQL data types for LOBs (large objects). To create host variables that can be used with these data types, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a PL/I language structure in the output source member.

The following figure shows the syntax for valid LOB host variables.

**LOB**

```
DECLARE
  variable-name,
  (variable-name)
SQL TYPE IS
  CLOB
  BLOB
  (lob-length);
```

Notes:
1. For BLOB and CLOB, 1 ≤ lob-length ≤ 32,766
2. SQL TYPE IS, BLOB, CLOB can be in mixed case.

**CLOB example**

The following declaration:
```
DCL MY_CLOB SQL TYPE IS CLOB(16384);
```

Results in the generation of the following structure:
```
DCL 1 MY_CLOB,
  3 MY_CLOB_LENGTH BINARY FIXED (31) UNALIGNED,
  3 MY_CLOB_DATA CHARACTER (16384);
```

**BLOB example**

The following declaration:
```
DCL MY_BLOB SQL TYPE IS BLOB(16384);
```

Results in the generation of the following structure:
```
DCL 1 MY_BLOB,
  3 MY_BLOB_LENGTH BINARY FIXED (31) UNALIGNED,
  3 MY_BLOB_DATA CHARACTER (16384);
```

The following figure shows the syntax for valid LOB locators.

**LOB locator**

```
DECLARE
  variable-name,
  (variable-name)
SQL TYPE IS
  CLOB_LOCATOR
  DBCLOB_LOCATOR
  BLOB_LOCATOR;
```

Note: SQL TYPE IS, BLOB_LOCATOR, CLOB_LOCATOR, DBCLOB_LOCATOR can be in mixed case.

**CLOB locator example**

The following declaration:
```
DCL MY_LOCATOR SQL TYPE IS CLOB_LOCATOR;
```
Results in the following generation:

```
DCL MY_LOCATOR BINARY FIXED(31) UNALIGNED;
```

BLOB and DBCLOB locators have similar syntax.

The following figure shows the syntax for valid LOB file reference variables.

**LOB file reference variable**

```
DECLARE variable-name SQL TYPE IS CLOB_FILE;
```

**Note:** SQL TYPE IS, BLOB_FILE, CLOB_FILE, and DBCLOB_FILE can be in mixed case.

**CLOB file reference example**

The following declaration:

```
DCL MY_FILE SQL TYPE IS CLOB_FILE;
```

Results in the generation of the following structure:

```
DCL 1 MY_FILE,
  3 MY_FILE_NAME_LENGTH BINARY FIXED(31) UNALIGNED,
  3 MY_FILE_DATA_LENGTH BINARY FIXED(31) UNALIGNED,
  3 MY_FILE_FILE_OPTIONS BINARY FIXED(31) UNALIGNED,
  3 MY_FILE_NAME CHAR(255);
```

BLOB and DBCLOB file reference variables have similar syntax.

The pre-compiler will generate declarations for the following file option constants:

- SQL_FILE_READ (2)
- SQL_FILE_CREATE (8)
- SQL_FILE_OVERWRITE (16)
- SQL_FILE_APPEND (32)

**Related reference:**

LOB file reference variables

**ROWID host variables in PL/I applications that use SQL:**

PL/I does not have a variable that corresponds to the SQL data type ROWID. To create host variables that can be used with this data type, use the SQL TYPE IS clause. The SQL precompiler replaces this declaration with a PL/I language structure in the output source member.

**ROWID**

```
DECLARE variable-name SQL TYPE IS ROWID;
```

**Note:** SQL TYPE IS ROWID can be in mixed case.
ROWID example

The following declaration:
DCL MY_ROWID SQL TYPE IS ROWID;

Results in the following generation:
DCL MY_ROWID CHARACTER(40) VARYING;

Using host structures in PL/I applications that use SQL

In PL/I programs, you can define a host structure, which is a named set of elementary PL/I variables. A host structure name can be a group name whose subordinate levels name elementary PL/I variables.

For example:
DCL 1 A,
  2 B,
    3 C1 CHAR(...),
    3 C2 CHAR(...);

In this example, B is the name of a host structure consisting of the elementary items C1 and C2.

You can use the structure name as shorthand notation for a list of scalars. You can qualify a host variable with a structure name (for example, STRUCTURE.FIELD). Host structures are limited to two levels. (For example, in the above host structure example, the A cannot be referred to in SQL.) A structure cannot contain an intermediate level structure. In the previous example, A could not be used as a host variable or referred to in an SQL statement. However, B is the first level structure. B can be referred to in an SQL statement. A host structure for SQL data is two levels deep and can be thought of as a named set of host variables. After the host structure is defined, you can refer to it in an SQL statement instead of listing the several host variables (that is, the names of the host variables that make up the host structure).

For example, you can retrieve all column values from selected rows of the table CORPDATA.EMPLOYEE with:
DCL 1 PEMPL,
  5 EMPNO CHAR(6),
  5 FIRSTNAME CHAR(12) VAR,
  5 MIDINIT CHAR(1),
  5 LASTNAME CHAR(15) VAR,
  5 WORKDEPT CHAR(3);

EMPID = '000220';
...
EXEC SQL
  SELECT * INTO :PEMPL
  FROM CORPDATA.EMPLOYEE
  WHERE EMPNO = :EMPID;

Host structures in PL/I applications that use SQL

This figure shows the syntax for valid host structure declarations.

Host structures

```
DECLARE 1-variable-name
DCL level-1-variable-name,...
Scope and/or storage
```
data-types:

- BINARY
- BIN
- FIXED
- FLOAT
- (precision)
- UNALIGNED
- DECIMAL
- DEC
- FIXED
- FLOAT
- (precision, scale)
- UNALIGNED
- PICTURE
- picture-string
- CHARACTER
- CHAR
- (length)
- VARYING
- VAR
- ALIGNED
- SQL TYPE IS
- CLOB
- (lob-length)
- BLOB
- K
- SQL TYPE IS
- CLOB_LOCATOR
- DBCLOB_LOCATOR
- BLOB_LOCATOR
- SQL TYPE IS
- CLOB_FILE
- DBCLOB_FILE
- BLOB_FILE
- SQL TYPE IS
- BINARY
- (length)
- VARBINARY
- BINARY VARYING

Notes:
1. level-1 indicates that there is an intermediate level structure.
2. level-1 must be an integer constant between 1 and 254.
3. level-2 must be an integer constant between 2 and 255.
4. For details on declaring numeric, character, LOB, ROWID, and binary host variables, see the notes under numeric-host variables, character-host variables, LOB host variables, ROWID host variables, and binary host variables.

Host structure indicator arrays in PL/I applications that use SQL
This figure shows the syntax for valid host structure indicator array declarations.

Host structure indicator array

Note: Dimension must be an integer constant between 1 and 32766.
Using host structure arrays in PL/I applications that use SQL

In PL/I programs, you can define a host structure array.

In these examples, the following are true:
- B_ARRAY is the name of a host structure array that contains the items C1_VAR and C2_VAR.
- B_ARRAY cannot be qualified.
- B_ARRAY can only be used with the blocked forms of the FETCH and INSERT statements.
- All items in B_ARRAY must be valid host variables.
- C1_VAR and C2_VAR are not valid host variables in any SQL statement. A structure cannot contain an intermediate level structure. A_STRUCT cannot contain the dimension attribute.

DCL 1 A_STRUCT,
  2 B_ARRAY(10),
  3 C1_VAR CHAR(20),
  3 C2_FIXED BIN(15) UNALIGNED;

To retrieve 10 rows from the CORPDATA.DEPARTMENT table, do the following:

DCL 1 DEPT(10),
  5 DEPTNO CHAR(3),
  5 DEPTNAME CHAR(29) VAR,
  5 MGRNO CHAR(6),
  5 ADMRDEPT CHAR (3);
DCL 1 IND_ARRAY(10),
  5 IND(4) FIXED BIN(15);
EXEC SQL
  DECLARE C1 CURSOR FOR
    SELECT *
    FROM CORPDATA.DEPARTMENT;
EXEC SQL
  FETCH C1 FOR 10 ROWS INTO :DEPT :IND_ARRAY;

Host structure array in PL/I applications that use SQL

This syntax diagram shows the syntax for valid host structure array declarations.

Host structure array

```
DECLARE 1-variable-name-(dimension),
  level-1-variable-name,
  ...

level-2 var-1 data-types ;
  ...
```

Data-types:
Notes:
1. *level-1* indicates that there is an intermediate level structure.
2. *level-1* must be an integer constant between 1 and 254.
3. *level-2* must be an integer constant between 2 and 255.
4. For details on declaring numeric, character, LOB, ROWID, and binary host variables, see the notes under numeric-host variables, character-host variables, LOB host variables, ROWID, and binary host variables.
5. Dimension must be an integer constant between 1 and 32,767.

**Host structure array indicator in PL/I applications that use SQL:**

This figure shows the syntax diagram for the declaration of a valid host structure array indicator.

```
DECLARE 1-variable-name-(dimension) Scope and/or storage;
DCL level-1-variable-name, level-2-identifier-(dimension-2) BINARY FIXED precision.
```

Notes:
1. *level-1* indicates that there is an intermediate level structure.
2. *level-1* must be an integer constant between 1 and 254.
3. *level-2* must be an integer constant between 2 and 255.
4. Dimension-1 and dimension-2 must be integer constants between 1 and 32,767.

**Using external file descriptions in PL/I applications that use SQL**

You can use the PL/I `%INCLUDE` directive to include the definitions of externally described files in a source program.
When used with SQL, only a particular format of the %INCLUDE directive is recognized by the SQL precompiler. That directive format must have the following three elements or parameter values, otherwise the precompiler ignores the directive. The required elements are file name, format name, and element type. There are two optional elements supported by the SQL precompiler: prefix name and COMMA.

The structure is ended normally by the last data element of the record or key structure. However, if in the %INCLUDE directive the COMMA element is specified, then the structure is not ended.

To include the definition of the sample table DEPARTMENT described in Db2 for i sample tables in the SQL programming topic collection, you can code:

```
DCL 1 TDEPT_STRUCTURE,
  %INCLUDE DEPARTMENT(DEPARTMENT,RECORD);
```

In the above example, a host structure named TDEPT_STRUCTURE would be defined having four fields. The fields would be DEPTNO, DEPTNAME, MGRNO, and ADMRDEPT.

For device files, if INDARA is not specified and the file contains indicators, the declaration cannot be used as a host structure array. The indicator area is included in the generated structure and causes the storage to not be contiguous.

```
DCL 1 DEPT_REC(10),
  %INCLUDE DEPARTMENT(DEPARTMENT,RECORD);
EXEC SQL DECLARE C1 CURSOR FOR
  SELECT * FROM CORPDATA.DEPARTMENT;
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 FOR 10 ROWS INTO :DEPT_REC;
```

**Note:** DATE, TIME, and TIMESTAMP columns will generate host variable definitions that are treated by SQL with the same comparison and assignment rules as a DATE, TIME, and TIMESTAMP column. For example, a date host variable can only be compared with a DATE column or a character string that is a valid representation of a date.

Although decimal and zoned fields with precision greater than 15 and binary with nonzero scale fields are mapped to character field variables in PL/I, SQL considers these fields to be numeric.

Although GRAPHIC and VARGRAPHIC are mapped to character variables in PL/I, SQL considers these to be GRAPHIC and VARGRAPHIC host variables. If the GRAPHIC or VARGRAPHIC column has a UCS-2 CCSID, the generated host variable will have the UCS-2 CCSID assigned to it. If the GRAPHIC or VARGRAPHIC column has a UTF-16 CCSID, the generated host variable will have the UTF-16 CCSID assigned to it.

CLOB, BLOB, and DBCLOB columns in the external file are ignored. No host variable definition will be generated in the host structure for these types.

**Determining equivalent SQL and PL/I data types**

The precompiler determines the base SQLTYPE and SQLLEN of host variables based on this table.

If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.
Table 5. PL/I declarations mapped to typical SQL data types

<table>
<thead>
<tr>
<th>PL/I data type</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN FIXED(p) where p is in the range 1 to 15</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>BIN FIXED(p) where p is in the range 16 to 31</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>DEC FIXED(p,s)</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>BIN FLOAT(p) p is in the range 1 to 24</td>
<td>480</td>
<td>4</td>
<td>FLOAT (single precision)</td>
</tr>
<tr>
<td>BIN FLOAT(p) p is in the range 25 to 53</td>
<td>480</td>
<td>8</td>
<td>FLOAT (double precision)</td>
</tr>
<tr>
<td>DEC FLOAT(m) m is in the range 1 to 7</td>
<td>480</td>
<td>4</td>
<td>FLOAT (single precision)</td>
</tr>
<tr>
<td>DEC FLOAT(m) m is in the range 8 to 16</td>
<td>480</td>
<td>8</td>
<td>FLOAT (double precision)</td>
</tr>
<tr>
<td>PICTURE picture string (numeric)</td>
<td>488</td>
<td>p in byte 1, s in byte 2</td>
<td>NUMERIC (p,s)</td>
</tr>
<tr>
<td>PICTURE picture string (sign leading separate)</td>
<td>504</td>
<td>p in byte 1, s in byte 2</td>
<td>No exact equivalent, use NUMERIC(p,s).</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>CHAR(n) VARYING</td>
<td>448</td>
<td>n</td>
<td>VARCHARC(n)</td>
</tr>
</tbody>
</table>

The following table can be used to determine the PL/I data type that is equivalent to a given SQL data type.

Table 6. SQL data types mapped to typical PL/I declarations

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>PL/I equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>BIN FIXED(p)</td>
<td>p is a positive integer from 1 to 15.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>BIN FIXED(p)</td>
<td>p is a positive integer from 16 to 31.</td>
</tr>
<tr>
<td>BIGINT</td>
<td>No exact equivalent</td>
<td>Use DEC FIXED(18).</td>
</tr>
<tr>
<td>DECIMAL(p,s) or NUMERIC(p,s)</td>
<td>DEC FIXED(p) or DEC FIXED(p,s) or PICTURE picture-string</td>
<td>s (the scale factor) and p (the precision) are positive integers. p is a positive integer from 1 to 31. s is a positive integer from 0 to p.</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>Not supported</td>
<td>Not supported.</td>
</tr>
<tr>
<td>FLOAT (single precision)</td>
<td>BIN FLOAT(p) or DEC FLOAT(m)</td>
<td>p is a positive integer from 1 to 24.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m is a positive integer from 1 to 7.</td>
</tr>
<tr>
<td>FLOAT (double precision)</td>
<td>BIN FLOAT(p) or DEC FLOAT(m)</td>
<td>p is a positive integer from 25 to 53.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>m is a positive integer from 8 to 16.</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>CHAR(n)</td>
<td>n is a positive integer from 1 to 32766.</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>CHAR(n) VARYING</td>
<td>n is a positive integer from 1 to 32740.</td>
</tr>
<tr>
<td>CLOB</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a CLOB.</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>Not supported</td>
<td>Not supported.</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>Not supported</td>
<td>Not supported.</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>Not supported</td>
<td>Not supported.</td>
</tr>
<tr>
<td>BINARY</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a BINARY.</td>
</tr>
</tbody>
</table>
Table 6. SQL data types mapped to typical PL/I declarations (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>PL/I equivalent</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARBINARY</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a VARBINARY.</td>
</tr>
<tr>
<td>BLOB</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a BLOB.</td>
</tr>
<tr>
<td>DATE</td>
<td>CHAR(n)</td>
<td>If the format is *USA, *JIS, *EUR, or *ISO, n must be at least 10 characters. If the format is *YMD, *DMY, or *MDY, n must be at least 8 characters. If the format is *JUL, n must be at least 6 characters.</td>
</tr>
<tr>
<td>TIME</td>
<td>CHAR(n)</td>
<td>n must be at least 6; to include seconds, n must be at least 8.</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>CHAR(n)</td>
<td>n must be at least 19. To include microseconds at full precision, n must be 26; if n is less than 26, truncation occurs on the microseconds part.</td>
</tr>
<tr>
<td>XML</td>
<td>Not supported</td>
<td>Not supported.</td>
</tr>
<tr>
<td>DATALINK</td>
<td>Not supported</td>
<td>Not supported.</td>
</tr>
<tr>
<td>ROWID</td>
<td>None</td>
<td>Use SQL TYPE IS to declare a ROWID.</td>
</tr>
<tr>
<td>Result set locator</td>
<td>Not supported</td>
<td>Not supported.</td>
</tr>
</tbody>
</table>

Using indicator variables in PL/I applications that use SQL

An indicator variable is a two-byte integer (BIN FIXED(p), where p is 1 to 15).

You can also specify an indicator structure (defined as an array of halfword integer variables) to support a host structure.

Indicator variables are declared in the same way as host variables and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

Example

Given the statement:

```sql
EXEC SQL FETCH CLS_CURSOR INTO :CLS_CD,
    :DAY :DAY_IND,
    :BGN :BGN_IND,
    :END :END_IND;
```

Variables can be declared as follows:

```pli
EXEC SQL BEGIN DECLARE SECTION;
DCL CLS_CD    CHAR(7);
DCL DAY      BIN FIXED(15);
DCL BGN      CHAR(8);
DCL END      CHAR(8);
DCL (DAY_IND, BGN_IND, END_IND) BIN FIXED(15);
EXEC SQL END DECLARE SECTION;
```

Related reference:

References to variables

"Indicator variables in applications that use SQL" on page 4

An indicator variable is a halfword integer variable used to communicate additional information about its
associated host variable.

**Differences in PL/I because of structure parameter passing techniques**

The PL/I precompiler attempts to use the structure parameter passing technique, if possible. This structure parameter passing technique provides better performance for most PL/I programs using SQL.

The precompiler generates code where each host variable is a separate parameter when the following conditions are true:

- A PL/I %INCLUDE compiler directive is found that copies external text into the source program.
- The data length of the host variables referred to in the statement is greater than 32 703. Because SQL uses 64 bytes of the structure, 32703 + 64 = 32767, the maximum length of a data structure.
- The PL/I precompiler estimates that it could possibly exceed the PL/I limit for user-defined names.
- A sign leading separate host variable is found in the host variable list for the SQL statement.

**Related concepts:**

Application design tips for database performance

---

**Coding SQL statements in RPG/400 applications**

The RPG/400 licensed program supports both RPG II and RPG III programs.

SQL statements can only be used in RPG III programs. RPG II and AutoReport are NOT supported. All referrals to RPG in this guide apply to RPG III or ILE RPG only.

This topic describes the unique application and coding requirements for embedding SQL statements in a RPG/400 program. Requirements for host variables are defined.

**Note:** By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 190.

For more information about programming using RPG, see the manuals RPG/400 User’s Guide and RPG/400 Reference at IBM Publications Center.

**Related concepts:**

“Writing applications that use SQL” on page 2

You can create database applications in host languages that use Db2 for i SQL statements and functions.

**Related reference:**

“Example programs: Using Db2 for i statements” on page 149

Here is a sample application that shows how to code SQL statements in each of the languages that Db2 for i supports.

“Example: SQL statements in RPG/400 programs” on page 171

This example program is written in the RPG programming language.

---

**Defining the SQL communication area in RPG/400 applications that use SQL**

The SQL precompiler automatically places the SQLCA in the input specifications of the RPG/400 program prior to the first calculation specification.

INCLUDE SQLCA should not be coded in the source program. If the source program specifies INCLUDE SQLCA, the statement will be accepted, but it is redundant. The SQLCA, as defined for RPG/400:

<table>
<thead>
<tr>
<th>ISQLCA</th>
<th>DS</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>I*</td>
<td>SQL COMMUNICATION AREA</td>
<td>SQL</td>
</tr>
<tr>
<td>I I</td>
<td>X’0000000000000000’</td>
<td>1</td>
</tr>
<tr>
<td>I I</td>
<td>B 9 120SQLABC</td>
<td>SQL</td>
</tr>
</tbody>
</table>
Note: Variable names in RPG/400 are limited to 6 characters. The standard SQLCA names have been changed to a length of 6. RPG/400 does not have a way of defining arrays in a data structure without also defining them in the extension specification. SQLERR is defined as character with SQLER1 through 6 used as the names of the elements.

Related reference:

**SQL communication area**

Defining SQL descriptor areas in RPG/400 applications that use SQL

There are two types of SQL descriptor areas. One is defined with the ALLOCATE DESCRIPTOR statement. The other is defined using the SQLDA structure. In this topic, only the SQLDA form is discussed.

The following statements can use an SQLDA:

- EXECUTE...USING DESCRIPTOR descriptor-name
- FETCH...USING DESCRIPTOR descriptor-name
- OPEN...USING DESCRIPTOR descriptor-name
- CALL...USING DESCRIPTOR descriptor-name
- DESCRIBE statement-name INTO descriptor-name
- DESCRIBE CURSOR cursor-name INTO descriptor-name
- DESCRIBE INPUT statement-name INTO descriptor-name
- DESCRIBE PROCEDURE procedure-name INTO descriptor-name
- DESCRIBE TABLE host-variable INTO descriptor-name
- PREPARE statement-name INTO descriptor-name

Unlike the SQLCA, there can be more than one SQLDA in a program and an SQLDA can have any valid name.

Dynamic SQL is an advanced programming technique. With dynamic SQL, your program can develop and then run SQL statements while the program is running. A SELECT statement with a variable SELECT list (that is, a list of the data to be returned as part of the query) that runs dynamically requires an SQL
descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT.

Because the SQLDA uses pointer variables that are not supported by RPG/400, an INCLUDE SQLDA statement cannot be specified in an RPG/400 program. An SQLDA must be set up by a C, C++, COBOL, PL/I, or ILE RPG program and passed to the RPG program in order to use it.

Related concepts:
- Dynamic SQL applications

Related reference:
- SQL descriptor area

**Embedding SQL statements in RPG/400 applications that use SQL**

SQL statements coded in an RPG/400 program must be placed in the calculation section. This requires that a C be placed in position 6.

SQL statements can be placed in detail calculations, in total calculations, or in an RPG/400 subroutine. The SQL statements are run based on the logic of the RPG/400 statements.

The keywords EXEC SQL indicate the beginning of an SQL statement. EXEC SQL must occupy positions 8 through 16 of the source statement, preceded by a / in position 7. The SQL statement may start in position 17 and continue through position 74.

The keyword END-EXEC ends the SQL statement. END-EXEC must occupy positions 8 through 16 of the source statement, preceded by a slash (/) in position 7. Positions 17 through 74 must be blank.

Both uppercase and lowercase letters are acceptable in SQL statements.

**Example: Embedding SQL statements in RPG/400 applications that use SQL**

An UPDATE statement coded in an RPG/400 program might be coded as this example shows.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
C/EXEC SQL UPDATE DEPARTMENT
C+ SET MANAGER = :MGRNUM
C+ WHERE DEPTNO = :INTDEP
C/END-EXEC
```

**Comments in RPG/400 applications that use SQL**

In addition to SQL comments (--), RPG/400 comments can be included within SQL statements wherever a blank is allowed, except between the keywords EXEC and SQL.

To embed an RPG/400 comment within the SQL statement, place an asterisk (*) in position 7.

**Continuation for SQL statements in RPG/400 applications that use SQL**

When additional records are needed to contain the SQL statement, positions 9 through 74 can be used. Position 7 must be a + (plus sign), and position 8 must be blank.

Constants containing DBCS data can be continued across multiple lines by placing the shift-in character in position 75 of the continued line and placing the shift-out character in position 8 of the continuation line. This SQL statement has a valid graphic constant of G’<AABBCCDDEEFFGHHIIJJKK’.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
C/EXEC SQL SELECT * FROM GRAHTAB
C+ WHERE GRAPHCOL = G’<AABB’
C+<CCDDEEFFGHHIIJJKK’
C/END-EXEC
```

**Including code in RPG/400 applications that use SQL**

SQL statements and RPG/400 calculation specifications can be included by embedding the SQL statement.
The /COPY statement can be used to include SQL statements or RPG/400 specifications.

Sequence numbers in RPG/400 applications that use SQL
The sequence numbers of the source statements generated by the SQL precompiler are based on the *NOSEQSRC/*SEQSRC keywords of the OPTION parameter on the CRTSQLRPG command.

When *NOSEQSRC is specified, the sequence number from the input source member is used. For *SEQSRC, the sequence numbers start at 000001 and are incremented by 1.

Names in RPG/400 applications that use SQL
Any valid RPG variable name can be used for a host variable and is subject to these restrictions.

Do not use host variable names or external entry names that begin with 'SQ', 'SQL', 'RDI', or 'DSN'. These names are reserved for the database manager.

Statement labels in RPG/400 applications that use SQL
A TAG statement can precede any SQL statement. Code the TAG statement on the line preceding EXEC SQL.

WHENEVER statement in RPG/400 applications that use SQL
The target for the GOTO clause must be the label of the TAG statement. The scope rules for the GOTO/TAG must be observed.

Using host variables in RPG/400 applications that use SQL
All host variables used in SQL statements must be explicitly declared. LOB, XML, ROWID, result set locator, and binary host variables are not supported in RPG/400.

SQL embedded in RPG/400 does not use the SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements to identify host variables. Do not put these statements in the source program.

All host variables within an SQL statement must be preceded by a colon (:).

The names of host variables must be unique within the program.

Declaring host variables in RPG/400 applications that use SQL
The SQL RPG/400 precompiler only recognizes a subset of RPG/400 declarations as valid host variable declarations.

Most variables defined in RPG/400 can be used in SQL statements. A partial listing of variables that are not supported includes the following:
- Indicator field names (*INxx)
- Tables
- UDATE
- UDAY
- UMONTH
- UYEAR
- Look-ahead fields
- Named constants
Fields used as host variables are passed to SQL, using the CALL/PARM functions of RPG/400. If a field cannot be used in the result field of the PARM, it cannot be used as a host variable.

**Using host structures in RPG/400 applications that use SQL**

The RPG/400 data structure name can be used as a host structure name if subfields exist in the data structure. The use of the data structure name in an SQL statement implies that it is the list of subfield names that make up the data structure.

When subfields are not present for the data structure, then the data structure name is a host variable of character type. This allows character variables larger than 256, because data structures can be up to 9999.

In the following example, BIGCHR is an RPG/400 data structure without subfields. SQL treats any referrals to BIGCHR as a character string with a length of 642.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...*
IBIGCHR      DS       642
```

In the next example, PEMPL is the name of the host structure consisting of the subfields EMPNO, FIRSTN, MIDINT, LASTNAME, and DEPTNO. The referral to PEMPL uses the subfields. For example, the first column of EMPLOYEE is placed in EMPNO, the second column is placed in FIRSTN, and so on.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...*
IPEMPL     DS
I           01   06 EMPNO
I           07   18 FIRSTN
I           19   19 MIDINT
I           20   34 LASTNA
I           35   37 DEPTNO
...
C  MOVE '000220' EMPNO
...
C/EXEC SQL
C+  SELECT * INTO :PEMPL
C+  FROM CORPDATA.EMPLOYEE
C+  WHERE EMPNO = :EMPNO
C/END-EXEC
```

When writing an SQL statement, referrals to subfields can be qualified. Use the name of the data structure, followed by a period and the name of the subfield. For example, PEMPL.MIDINT is the same as specifying only MIDINT.

**Using host structure arrays in RPG/400 applications that use SQL**

A host structure array is defined as an occurrence data structure. An occurrence data structure can be used on the SQL FETCH statement when fetching multiple rows.

In these examples, the following are true:

- All items in BARRAY must be valid host variables.
- All items in BARRAY must be contiguous. The first FROM position must be 1 and there cannot be overlaps in the TO and FROM positions.
- For all statements other than the multiple-row FETCH and blocked INSERT, if an occurrence data structure is used, the current occurrence is used. For the multiple-row FETCH and blocked INSERT, the occurrence is set to 1.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7...*
IBARRAY    DS       10
I           01  20 C1VAR
I           02  21 C2VAR
```
The following example uses a host structure array called DEPT and a multiple-row FETCH statement to retrieve 10 rows from the DEPARTMENT table.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....*
E INDS 4 4 0
I DEPT DS 10
I 01 03 DEPTNO
I 04 32 DEPTNM
I 33 38 MGRNO
I 39 41 ADMRD
IINDARR DS 10
I B 1 80INDS
...
```

```
C/EXEC SQL
C+ DECLARE C1 CURSOR FOR
C+ SELECT * 
C+ FROM CORPDATA.DEPARTMENT 
C/END-EXEC
C/EXEC SQL
C+ OPEN C1
C/END-EXEC
C/EXEC SQL
C+ FETCH C1 FOR 10 ROWS INTO :DEPT:INDARR
C/END-EXEC
```

Using external file descriptions in RPG/400 applications that use SQL

The SQL precompiler processes the RPG/400 source in much the same manner as the ILE RPG compiler. This means that the precompiler processes the /COPY statement for definitions of host variables.

Field definitions for externally described files are obtained and renamed, if different names are specified. The external definition form of the data structure can be used to obtain a copy of the column names to be used as host variables.

In the following example, the sample table DEPARTMENT is used as a file in an RPG/400 program. The SQL precompiler retrieves the field (column) definitions for DEPARTMENT for use as host variables.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....*
FTDEPT IP E DISK
F DEPT TDEPT KRENAMEDEPTREC
I DEPTNAME DEPTN
I ADMRDEPT ADMRD
```

Note: Code an F-spec for a file in your RPG program only if you use RPG/400 statements to do I/O operations to the file. If you use only SQL statements to do I/O operations to the file, you can include the external definition by using an external data structure.

In the following example, the sample table is specified as an external data structure. The SQL precompiler retrieves the field (column) definitions as subfields of the data structure. Subfield names can be used as host variable names, and the data structure name TDEPT can be used as a host structure name. The field names must be changed because they are greater than six characters.

```
*...1....+....2....+....3....+....4....+....5....+....6....+....7....*
ITDEPT E DSDEPARTMENT
I DEPTNAME DEPTN
I ADMRDEPT ADMRD
```

Note: DATE, TIME, and TIMESTAMP columns will generate host variable definitions that are treated by SQL with the same comparison and assignment rules as a DATE, TIME, and TIMESTAMP column. For example, a date host variable can only be compared against a DATE column or a character string that is a valid representation of a date.
Although varying-length columns generate fixed-length character-host variable definitions, to SQL they are varying-length character variables.

Although GRAPHIC and VARGRAPHIC columns are mapped to character variables in RPG/400, SQL considers these GRAPHIC and VARGRAPHIC variables. If the GRAPHIC or VARGRAPHIC column has a UCS-2 CCSID, the generated host variable will have the UCS-2 CCSID assigned to it. If the GRAPHIC or VARGRAPHIC column has a UTF-16 CCSID, the generated host variable will have the UTF-16 CCSID assigned to it.

CLOB, BLOB, and DBCLOB columns in the external file are ignored. No host variable definition will be generated in the host structure for these types.

**External file description considerations for host structure arrays in RPG/400 applications that use SQL**

Field definitions for externally described files, including renaming of fields, are recognized by the SQL precompiler.

The external definition form of the data structure can be used to obtain a copy of the column names to be used as host variables.

In the following example, the DEPARTMENT table is included in the RPG/400 program and is used to declare a host structure array. A multiple-row FETCH statement is then used to retrieve 10 rows into the host structure array.

```
*...1....+.2....+.3....+.4....+.5....+.6....*
ITDEPT  E DSDEPARTMENT  10
  I  DEPARTMENT  DEPTN
  I  ADMRDEPT  ADMRD
...
C/EXEC SQL
C+  DECLARE C1 CURSOR FOR
C+  SELECT *
C+    FROM CORPDATA.DEPARTMENT
C/END-EXEC

C/EXEC SQL
C+  FETCH C1 FOR 10 ROWS INTO :TDEPT
C/END-EXEC
```

**Determining equivalent SQL and RPG/400 data types**

The precompiler determines the base SQLTYPE and SQLLEN of host variables based on the table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.

<table>
<thead>
<tr>
<th>RPG/400 data type</th>
<th>Col 43</th>
<th>Col 52</th>
<th>Other RPG/400 coding</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Structure subfield</td>
<td>blank</td>
<td>blank</td>
<td>Length = n when n ≤ 256</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>Data structure (without subfields)</td>
<td>n/a</td>
<td>n/a</td>
<td>Length = n when n ≤ 9999</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>Input field</td>
<td>blank</td>
<td>blank</td>
<td>Length = n when n ≤ 256</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
</tbody>
</table>
Table 7. RPG/400 declarations mapped to typical SQL data types (continued)

<table>
<thead>
<tr>
<th>RPG/400 data type</th>
<th>Col 43</th>
<th>Col 52</th>
<th>Other RPG/400 coding</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculation result field</td>
<td>n/a</td>
<td>blank</td>
<td>Length = n where n ≤ 256</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>Data Structure subfield</td>
<td>B</td>
<td>0</td>
<td>Length = 2</td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>Data Structure subfield</td>
<td>B</td>
<td>0</td>
<td>Length = 4</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Data Structure subfield</td>
<td>B</td>
<td>1-4</td>
<td>Length = 2</td>
<td>500</td>
<td>2</td>
<td>DECIMAL(4,s)</td>
</tr>
<tr>
<td>Data Structure subfield</td>
<td>B</td>
<td>1-9</td>
<td>Length = 4</td>
<td>496</td>
<td>4</td>
<td>DECIMAL(9,s)</td>
</tr>
<tr>
<td>Data Structure subfield</td>
<td>P</td>
<td>0 to 9</td>
<td>Length = n where n is 1 to 16</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>Input field</td>
<td>P</td>
<td>0 to 9</td>
<td>Length = n where n is 1 to 16</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>Input field</td>
<td>blank</td>
<td>0 to 9</td>
<td>Length = n where n is 1 to 30</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>Input field</td>
<td>B</td>
<td>0 to 4 if n = 2; 0 to 9 if n = 4</td>
<td>Length = 2 or 4</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>Calculation result field</td>
<td>n/a</td>
<td>0 to 9</td>
<td>Length = n where n is 1 to 30</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p,s)</td>
</tr>
<tr>
<td>Data Structure subfield</td>
<td>blank</td>
<td>0 to 9</td>
<td>Length = n where n is 1 to 30</td>
<td>488</td>
<td>p in byte 1, s in byte 2</td>
<td>NUMERIC(p,s)</td>
</tr>
</tbody>
</table>

Use the information in the following table to determine the RPG/400 data type that is equivalent to a given SQL data type.

Table 8. SQL data types mapped to typical RPG/400 declarations

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>RPG/400 data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>Subfield of a data structure. B in position 43, length must be 2 and 0 in position 52 of the subfield specification.</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>Subfield of a data structure. B in position 43, length must be 4 and 0 in position 52 of the subfield specification.</td>
<td></td>
</tr>
<tr>
<td>SQL data type</td>
<td>RPG/400 data type</td>
<td>Notes</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------</td>
<td>-------</td>
</tr>
<tr>
<td>BIGINT</td>
<td>No exact equivalent</td>
<td>Use P in position 43 and 0 in position 52 of the subfield specification.</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Subfield of a data structure. P in position 43 and 0 through 9 in position 52 of the subfield specification. OR Defined as numeric and not a subfield of a data structure.</td>
<td>Maximum length of 16 (precision 30) and maximum scale of 9.</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>Subfield of the data structure. Blank in position 43 and 0 through 9 in position 52 of the subfield OR Defined as numeric and not a subfield of a data structure.</td>
<td>Maximum length of 30 (precision 30) and maximum scale of 9.</td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>FLOAT (single precision)</td>
<td>No exact equivalent</td>
<td>Use one of the alternative numeric data types described above.</td>
</tr>
<tr>
<td>FLOAT (double precision)</td>
<td>No exact equivalent</td>
<td>Use one of the alternative numeric data types described above.</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>Subfield of a data structure or input field. Blank in positions 43 and 52 of the specification. OR Calculation result field defined without decimal places.</td>
<td>n can be from 1 to 256.</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>Data structure name with no subfields in the data structure.</td>
<td>n can be from 1 to 9999.</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>No exact equivalent</td>
<td>Use a character host variable large enough to contain the largest expected VARCHAR value.</td>
</tr>
<tr>
<td>CLOB</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>BINARY</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>BLOB</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>DATE</td>
<td>Subfield of a data structure. Blank in position 52 of the subfield specification. OR Field defined without decimal places.</td>
<td>If the format is *USA, *JIS, *EUR, or *ISO, the length must be at least 10. If the format is *YMD, *DMY, or *MDY, the length must be at least 8. If the format is *JUL, the length must be at least 6.</td>
</tr>
<tr>
<td>TIME</td>
<td>Subfield of a data structure. Blank in position 52 of the subfield specification. OR Field defined without decimal places.</td>
<td>Length must be at least 6; to include seconds, length must be at least 8.</td>
</tr>
</tbody>
</table>
Table 8. SQL data types mapped to typical RPG/400 declarations (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>RPG/400 data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIMESTAMP</td>
<td>Subfield of a data structure. Blank in position 52 of</td>
<td>Length must be at least 19. To include microseconds at full precision,</td>
</tr>
<tr>
<td></td>
<td>the subfield specification.</td>
<td>length must be 26. If length is less than 26, truncation occurs</td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>on the microseconds part.</td>
</tr>
<tr>
<td></td>
<td>Field defined without decimal places.</td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>DATALINK</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>ROWID</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
<tr>
<td>Result set locator</td>
<td>Not supported</td>
<td>Not supported</td>
</tr>
</tbody>
</table>

**Assignment rules in RPG/400 applications that use SQL**

RPG/400 associates precision and scale with all numeric types.

RPG/400 defines numeric operations, assuming the data is in packed format. This means that operations involving binary variables include an implicit conversion to packed format before the operation is performed (and back to binary, if necessary). Data is aligned to the implied decimal point when SQL operations are performed.

**Using indicator variables in RPG/400 applications that use SQL**

An indicator variable is a two-byte integer.

See the entry for the SMALLINT SQL data type in Table 7 on page 94.

An indicator structure can be defined by declaring the variable as an array with an element length of 4,0 and declaring the array name as a subfield of a data structure with B in position 43.

Indicator variables are declared in the same way as host variables and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

**Related reference:**

References to variables

“Indicator variables in applications that use SQL” on page 4

An *indicator variable* is a halfword integer variable used to communicate additional information about its associated host variable.

**Example: Using indicator variables in RPG/400 applications that use SQL**

This example shows declaring indicator variables in RPG.

Given the statement:

```c
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
C/EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,
C+ :DAY :DAYIND,
C+ :BGN :BGNIND,
C+ :END :ENDIND
C/END-EXEC
```

variables can be declared as follows:

```c
*...1....+....2....+....3....+....4....+....5....+....6....+....7...
I  DS
I   1  7 CLSCD
I   8  90DAY
I  10 110DAYIND
```

References to variables

“Indicator variables in applications that use SQL” on page 4

An *indicator variable* is a halfword integer variable used to communicate additional information about its associated host variable.

Example: Using indicator variables in RPG/400 applications that use SQL

This example shows declaring indicator variables in RPG.
Differences in RPG/400 because of structure parameter passing techniques

The SQL RPG/400 precompiler attempts to use the structure parameter passing technique, if possible.

The precompiler generates code where each host variable is a separate parameter when the following conditions are true:

- The data length of the host variables, referred to in the statement, is greater than 9935. Because SQL uses 64 bytes of the structure, 9935 + 64 = 9999, the maximum length of a data structure.
- An indicator is specified on the statement where the length of the indexed indicator name plus the required index value is greater than six characters. The precompiler must generate an assignment statement for the indicator with the indicator name in the result field that is limited to six characters ("INDIC,1" requires seven characters).
- The length of a host variable is greater than 256. This can happen when a data structure without subfields is used as a host variable, and its length exceeds 256. Subfields cannot be defined with a length greater than 256.

Related concepts:
- Application design tips for database performance

Correctly ending a called RPG/400 program that uses SQL

SQL run time builds and maintains data areas (internal SQLDAs) for each SQL statement that contains host variables.

These internal SQLDAs are built the first time the statement is run and then reused on subsequent executions of the statement to increase performance. The internal SQLDAs can be reused as long as there is at least one SQL program active. The SQL precompiler allocates static storage used by SQL run time to manage the internal SQLDAs properly.

If an RPG/400 program containing SQL is called from another program that also contains SQL, the RPG/400 program should not set the Last Record (LR) indicator on. Setting the LR indicator on causes the static storage to be re-initialized the next time the RPG/400 program is run. Re-initializing the static storage causes the internal SQLDAs to be rebuilt, thus causing a performance degradation.

An RPG/400 program containing SQL statements that is called by a program that also contains SQL statements, should be ended one of two ways:

- By the RETRN statement
- By setting the RT indicator on.

This allows the internal SQLDAs to be used again and reduces the total run time.

Coding SQL statements in ILE RPG applications

You need to be aware of the unique application and coding requirements for embedding SQL statements in an ILE RPG program. In this topic, the coding requirements for host variables are defined.

- Not all features implemented by the ILE RPG compiler are supported by the SQL precompiler.
- If a control specification is not present in the program source, the RPG compiler searches for a data area containing control specification keywords. This feature is not supported by the SQL precompiler.
**Note:** By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 190.

For more information about programming using ILE RPG, see the ILE RPG Programmer's Guide topic and the ILE RPG Language Reference topic.

**Related concepts:**
- “Writing applications that use SQL” on page 2

You can create database applications in host languages that use Db2 for i SQL statements and functions.

**Related reference:**
- “Example programs: Using Db2 for i statements” on page 149
- “Example: SQL statements in ILE RPG programs” on page 177

Here is a sample application that shows how to code SQL statements in each of the languages that Db2 for i supports.

This example program is written in the ILE RPG programming language.

## Defining the SQL communication area in ILE RPG applications that use SQL

The SQL precompiler automatically places the SQL communication area (SQLCA) in the definition specifications of the ILE RPG program before the first calculation specification, unless a SET OPTION SQLCA = "NO" statement is found.

INCLUDE SQLCA should not be coded in the source program. If the source program specifies INCLUDE SQLCA, the statement will be accepted, but it is redundant. The SQLCA source statements for ILE RPG are:

```sql
// SQL COMMUNICATION AREA
DCL-DS SQLCA;
SQLCAID CHAR(8) INZ('X'0000000000000000');
SQLAID CHAR(8) OVERLAY(SQLCAID);
SQLABC INT(10);
SQLABCD BINDEC(9) OVERLAY(SQLABC);
SQLCODE INT(10);
SQLCOD BINDEC(9) OVERLAY(SQLCODE);
SQLERRML INT(5);
SQLERML BINDEC(4) OVERLAY(SQLERRML);
SQLERRMC CHAR(70);
SQLERM CHAR(70) OVERLAY(SQLERRMC);
SQLERRP CHAR(8);
SQLERP CHAR(8) OVERLAY(SQLERRP);
SQLERR CHAR(24);
SQLERR1 BINDEC(9) OVERLAY(SQLERR:*NEXT);
SQLERR2 BINDEC(9) OVERLAY(SQLERR:*NEXT);
SQLERR3 BINDEC(9) OVERLAY(SQLERR:*NEXT);
SQLERR4 BINDEC(9) OVERLAY(SQLERR:*NEXT);
SQLERR5 BINDEC(9) OVERLAY(SQLERR:*NEXT);
SQLERR6 BINDEC(9) OVERLAY(SQLERR:*NEXT);
SQLERRD INT(10) DIM(6) OVERLAY(SQLERR);    
SQLWRN CHAR(11);
SQLWR0 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR1 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR2 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR3 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR4 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR5 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR6 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR7 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR8 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWR9 CHAR(1) OVERLAY(SQLWRN:*NEXT);
SQLWRN CHAR(1) OVERLAY(SQLWRN:*NEXT);
```
If a SET OPTION SQLCA = *NO statement is found, the SQL precompiler automatically places SQLCODE and SQLSTATE variables in the definition specification. They are defined as follows when the SQLCA is not included:

DCL-S SQLCODE INT(10);
DCL-S SQLSTATE CHAR(5);

Related reference:
SQL communication area

Defining SQL descriptor areas in ILE RPG applications that use SQL

There are two types of SQL descriptor areas (SQLDAs). One is defined with the ALLOCATE DESCRIPTOR statement. The other is defined using the SQLDA structure. In this topic, only the SQLDA form is discussed.

The following statements can use an SQLDA:

- EXECUTE...USING DESCRIPTOR descriptor-name
- FETCH...USING DESCRIPTOR descriptor-name
- OPEN...USING DESCRIPTOR descriptor-name
- CALL...USING DESCRIPTOR descriptor-name
- DESCRIBE statement-name INTO descriptor-name
- DESCRIBE CURSOR cursor-name INTO descriptor-name
- DESCRIBE INPUT statement-name INTO descriptor-name
- DESCRIBE PROCEDURE procedure-name INTO descriptor-name
- DESCRIBE TABLE host-variable INTO descriptor-name
- PREPARE statement-name INTO descriptor-name

Unlike the SQLCA, there can be more than one SQLDA in a program and an SQLDA can have any valid name.

Dynamic SQL is a programming technique. With dynamic SQL, your program can develop and then run SQL statements while the program is running. A SELECT statement with a variable SELECT list (that is, a list of columns to be returned as part of the query) that runs dynamically requires an SQL descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT.

You can specify an INCLUDE SQLDA statement in an ILE RPG program. If an INCLUDE SQLDA statement is found anywhere in your program, the SQLDA structure is generated one time as part of the global definitions in your program.

EXEC SQL INCLUDE SQLDA;

The INCLUDE SQLDA generates the following data structure.

```
// SQL DESCRIPTOR AREA
DCL-DS SQLDA;
  SQLDAID CHAR(8);
  SQLDABC INT(10);
  SQLN INT(5);
  SQLD INT(5);
  SQL_VAR CHAR(80) DIM(SQL_NUM);
  *N POINTER OVERLAY(SQL_VAR:17);
  *N POINTER OVERLAY(SQL_VAR:33);
```
The user is responsible for the definition of SQL_NUM. SQL_NUM must be defined as a numeric constant with the dimension required for SQL_VAR.

The INCLUDE SQLDA generates two data structures. The second data structure is used to setup and reference the part of the SQLDA that contains the field descriptions.

To set the field descriptions of the SQLDA the program sets up the field description in the subfields of SQLVAR and then assigns SQLVAR to SQL_VAR(n), where n is the number of the field in the SQLDA. This is repeated until all the field descriptions are set.

When the SQLDA field descriptions are to be referenced the user assigns SQLVAR(n) to SQL_VAR where n is the number of the field description to be processed.

**Related concepts:**
- Dynamic SQL applications

**Related reference:**
- SQL descriptor area

**Embedding SQL statements in ILE RPG applications that use SQL**

SQL statements coded in an ILE RPG program can be placed in the calculation section or in a free-form calculation block.

SQL statements can be placed in detail calculations, in total calculations, or in RPG subroutines. The SQL statements are run based on the logic of the RPG statements.

Both uppercase and lowercase letters are acceptable in SQL statements.

**Fixed-form RPG**

The keywords EXEC SQL indicate the beginning of an SQL statement. EXEC SQL must occupy positions 8 through 16 of the source statement, preceded by a / in position 7. The SQL statement may start in position 17 and continue through position 80.

The keyword END-EXEC ends the SQL statement. END-EXEC must occupy positions 8 through 16 of the source statement, preceded by a slash (/) in position 7. Positions 17 through 80 must be blank.
An UPDATE statement coded in an ILE RPG program might be coded as follows:

C/EXEC SQL UPDATE DEPARTMENT
C+ SET MANAGER = :MGRNUM
C+ WHERE DEPTNO = :INTDEP
C/END-EXEC

**Free-form RPG**

Each SQL statement must begin with EXEC SQL and end with a semicolon (;). The EXEC SQL keywords must be on one line. The remaining part of the SQL statement can be on more than one line. Each SQL statement should start on a new line. No other statement should be on the same line as the SQL statement.

An UPDATE statement coded in free form might be coded in the following way:

EXEC SQL UPDATE DEPARTMENT
   SET MGRNO = :MGR_NUM
   WHERE DEPTNO = :INT_DEP;

**Comments in ILE RPG applications that use SQL**

In addition to SQL comments (--), ILE RPG comments can be included within SQL statements wherever SQL allows a blank character.

**Fixed-form RPG**

To embed an ILE RPG comment within the SQL statement, place an asterisk (*) in position 7.

**Free-form RPG**

Bracketed comments (/.../*/) are allowed within embedded SQL statements whenever a blank is allowed, except between the keywords EXEC and SQL. Comments can span any number of lines. Single-line comments (//) can also be used.

**Continuation for SQL statements in ILE RPG applications that use SQL**

SQL statements can be continued across many records in ILE RPG.

**Fixed-form RPG**

When additional records are needed to contain the SQL statement, positions 9 through 80 can be used. Position 7 must be a plus sign (+), and position 8 must be blank. Position 80 of the continued line is concatenated with position 9 of the continuation line.

Constants containing DBCS data can be continued across multiple lines by placing the shift-in character in position 81 of the continued line and placing the shift-out character in position 8 of the continuation line.

In this example, the SQL statement has a valid graphic constant of G'<AABBCCDDEEFFGHHIIJJKK>'.

C/EXEC SQL SELECT * FROM GRAPHTAB WHERE GRAPHCOL = G'<AABBCCDDEEFFGHHIIJJKK>'
C+<FFGHHIIJJKK'>
C/END-EXEC

**Free-form RPG**

SQL statements can be contained on one or more lines. To continue an SQL statement across multiple lines, the SQL statement can be split wherever a blank is allowed. The plus sign (+) can be used to indicate a continuation of a string constant. The literal continues with the first nonblank character on the next line.
Including code in ILE RPG applications that use SQL
To include SQL statements and RPG specifications in ILE RPG applications, use the SQL INCLUDE statement.

C/EXEC SQL INCLUDE member-name
C/END-EXEC

RPG directives are handled by the SQL precompiler according to the value of the RPG preprocessor options parameter (RPGPPOPT).

Related reference:
[“Using directives in ILE RPG applications that use SQL”]
RPG directives are handled by the SQL precompiler according to the value of the RPG preprocessor options parameter (RPGPPOPT). If the RPG preprocessor is used, the SQL precompile will run using the expanded preprocessed source.

Using directives in ILE RPG applications that use SQL
RPG directives are handled by the SQL precompiler according to the value of the RPG preprocessor options parameter (RPGPPOPT). If the RPG preprocessor is used, the SQL precompile will run using the expanded preprocessed source.

- When the value is *NONE, the RPG preprocessor is not called to preprocess the RPG source. When a source stream file is precompiled, no directives are recognized by SQL. When a source member is precompiled, the only directive that is handled by the SQL precompiler is /COPY. Nested /COPY statements are not handled. In fully free-form source mode, /COPY statements that start in any column are handled by the SQL precompiler. In column-limited source mode, only /COPY statements that start in column 7 are handled by the SQL precompiler. All /COPY statements must reference a source member; they cannot reference a source stream file. All other directives are ignored until the RPG compiler is called. This means that all RPG and SQL statements within conditional logic blocks are processed unconditionally by the SQL precompiler.

- When the value is *LVL1, the RPG preprocessor will be called to preprocess the RPG source. All /COPY statements are expanded, even nested /COPY statements, and the conditional compilation directives will be handled.

- When the value is *LVL2, the RPG preprocessor will be called to preprocess the RPG source. All /COPY and /INCLUDE statements are expanded and the conditional compilation directives will be handled.

- When *LVL1 or *LVL2 is used, there is a possibility that the expanded source generated by the RPG preprocessor will become very large and reach a resource limit due to the expansion of the /COPY and /INCLUDE statements. If this happens you must either break up your source into smaller pieces, or not use the RPG preprocessor.

Related reference:
[“Including code in ILE RPG applications that use SQL”]
To include SQL statements and RPG specifications in ILE RPG applications, use the SQL INCLUDE statement.

Sequence numbers in ILE RPG applications that use SQL
The sequence numbers of the source statements generated by the SQL precompiler are based on the *NOSEQSRC/*SEQSRC keywords of the OPTION parameter on the CRTSQLRPGD command.

When *NOSEQSRC is specified, the sequence number from the input source member is used. For *SEQSRC, the sequence numbers start at 000001 and are incremented by 1.

Names in ILE RPG applications that use SQL
Any valid ILE RPG variable name can be used for a host variable with these restrictions.

- Do not use host variable names or external entry names that begin with the characters SQ, SQL, RDI, or DSN. These names are reserved for the database manager.
- The maximum length of host variable names is 64 characters.
Statement labels in ILE RPG applications that use SQL
A TAG statement can precede any SQL statement. Code the TAG statement on the line preceding EXEC SQL.

WHENEVER statement in ILE RPG applications that use SQL
The target for the GOTO clause must be the label of the TAG statement. The scope rules for the GOTO/TAG must be observed.

Using host variables in ILE RPG applications that use SQL
All host variables used in SQL statements must be explicitly declared.

SQL embedded in ILE RPG does not use the SQL BEGIN DECLARE SECTION and END DECLARE SECTION statements to identify host variables. Do not put these statements in the source program.

All host variables within an SQL statement must be preceded by a colon (:). Names of host variables do not need to be unique within the program. The precompiler recognizes variables with the same name in different procedures and scopes them correctly.

An SQL statement that uses a host variable must be within the scope of the variable's declaration. A DECLARE CURSOR statement must be in the same scope as its OPEN statement to guarantee that any variables used by the DECLARE CURSOR will use the definition in the scope of the OPEN.

If an error stating that a host variable is not defined or not usable is issued, look at the cross-reference in the precompiler listing to see how the precompiler defined the variable. To generate a cross-reference in the listing, run the precompile command with *XREF specified on the OPTIONS parameter.

Declaring host variables in ILE RPG applications that use SQL
The SQL ILE RPG precompiler only recognizes a subset of valid ILE RPG declarations as valid host variable declarations.

Most variables defined in ILE RPG can be used in SQL statements. A partial listing of variables that are not supported includes the following:

- Unsigned integers
- Pointer
- Tables
- UDATETIME
- DAYTIME
- UMONTH
- UYEAR
- Look-ahead fields
- Named constants which refer to figurative constants or built-in functions
  - Multiple dimension arrays
  - Definitions requiring the resolution of %SIZE or %ELEM
  - Definitions requiring the resolution of constants unless the constant is used in OCCURS, DIM, OVERLAY, or POS and the constant is declared before it is used in the OCCURS, DIM, OVERLAY, or POS.

Fields used as host variables are passed to SQL using the CALL/Parm functions of ILE RPG. If a field cannot be used in the result field of the Parm, it cannot be used as a host variable.

Date and time host variables are always assigned to corresponding date and time subfields in the structures generated by the SQL precompiler. The generated date and time subfields are declared using
the format and separator specified by the DATFMT, DATSEP, TIMFMT, and TIMSEP parameters on the CRTSQLRPGI command or with the SET OPTION statement. Conversion from the user declared host variable format to the precompile specified format occurs on assignment to and from the SQL generated structure. If the DATFMT parameter value is a system format (*MDY, *YMD, *DMY, or *JUL), then all input and output host variables must contain date values within the range 1940-2039. If any date value is outside of this range, then the DATFMT on the precompile must be specified as one of the IBM SQL formats of *ISO, *USA, *EUR, or *JIS.

The SQL precompiler accepts variables defined as VARYING(2), but does not support VARYING(4). The LEN keyword is recognized, but only lengths up to the SQL limit for the data type are accepted.

Graphic host variables will use the RPG CCSID value if one is specified. An SQL DECLARE VARIABLE statement cannot be used to change the CCSID of a host variable whose CCSID has been defined in RPG, or a host variable that is defined as UCS-2 or UTF-16.

The precompiler will generate an RPG logical (indicator) variable as a character of length 1. This type can be used wherever SQL allows a character host variable. It cannot be used as an SQL indicator variable. It is up to the user to make sure that only values of 1 or 0 are assigned to it.

The precompiler supports EXTNANE, but does not support EXTNANE(filename : fnname : fieldtype), where fieldtype is *NULL, *ALL, *INPUT, *OUTPUT, or *KEY.

The precompiler supports LIKERE(intrecname), but does not support the optional second parameter.

The precompiler supports EXTDESC(literal), but does not support EXTDESC(constant).

For SQLTYPE declarations, the precompiler only supports the CCSID keyword. All other keywords are ignored.

**Declaring binary host variables in ILE RPG applications that use SQL:**

ILE RPG does not have variables that correspond to the SQL binary data types.

To create host variables that can be used with these data types, use the SQLTYPE keyword. The SQL precompiler replaces this declaration with an ILE RPG language declaration in the output source member. Binary declarations can be either standalone or within a data structure.

**BINARY example**

- The following declaration in free-form:
  ```sql
  DCL-S MYBINARY SQLTYPE(BINARY:50);
  ```
  results in the generation of the following code:
  ```sql
  DCL-S MYBINARY CHAR(50) CCSID(*HEX);
  ```
- The following declaration in fixed-form:
  ```sql
  D MYBINARY S SQLTYPE(BINARY:50)
  ```
  results in the generation of the following code:
  ```sql
  D MYBINARY S 50A CCSID(*HEX)
  ```

**VARBINARY example**

- The following declaration in free-form:
  ```sql
  DCL-S MYVARBINARY SQLTYPE(VARBINARY:100);
  ```
  results in the generation of the following code:
  ```sql
  DCL-S MYVARBINARY VARCHAR(100) CCSID(*HEX)
  ```
- The following declaration in fixed-form:
D MYVARBINARY S SQLTYPE(VARBINARY:100)
results in the generation of the following code:
D MYVARBINARY S 100A VARYING CCSID(*HEX)

Notes:
1. For BINARY host variables, the length must be in the range 1 to 32766.
2. For VARBINARY host variables, the length must be in the range 1 to 32740.
3. BINARY and VARBINARY host variables are allowed to be declared in host structures.
4. SQLTYPE, BINARY, and VARBINARY can be in mixed case.
5. SQLTYPE must be between positions 44 to 80 for fixed-form declarations.
6. When a BINARY or VARBINARY is declared as a standalone host variable, position 24 must contain
   the character S and position 25 must be blank for fixed-form declarations.
7. The standalone field indicator S in position 24 for fixed-form declarations should be omitted when a
   BINARY or VARBINARY host variable is declared in a host structure.

Declaring LOB host variables in ILE RPG applications that use SQL:

ILE RPG does not have variables that correspond to the SQL data types for LOBs (large objects).
To create host variables that can be used with these data types, use the SQLTYPE keyword. The SQL
precompiler replaces this declaration with an ILE RPG language structure in the output source member.
LOB declarations can be either standalone or within a data structure.

LOB host variables in ILE RPG applications that use SQL:

Here are some examples of LOB host variables (CLOB, DBCLOB, BLOB) in ILE RPG applications.

CLOB example
• The following declaration in free-form:
  DCL-S MYCLOB SQLTYPE(CLOB:1000);
results in the generation of the following structure:
  DCL-DS MYCLOB;
  MYCLOB_LEN UNS(10);
  MYCLOB_DATA CHAR(1000);
  END-DS MYCLOB;
• The following declaration in fixed-form:
  D MYCLOB S SQLTYPE(CLOB:1000)
results in the generation of the following structure:
  D MYCLOB DS
  D MYCLOB_LEN 10U
  D MYCLOB_DATA 1000A

DBCLOB example
• The following declaration in free-form:
  DCL-S MYDBCLOB SQLTYPE(DBCLOB:400);
results in the generation of the following structure:
  DCL-DS MYDBCLOB;
  MYDBCLOB_LEN UNS(10);
  MYDBCLOB_DATA GRAPH(400);
  END-DS MYDBCLOB;
• The following declaration in fixed-form:
  D MYDBCLOB S SQLTYPE(DBCLOB:400)
results in the generation of the following structure:

```
D MYDBCLOB     DS
D MYDBCLOB_LEN  10U
D MYDBCLOB_DATA 400G
```

**BLOB example**

- The following declaration in free-form:
  
  ```
  DCL-S MYBLOB SQLTYPE(BLOB:500);
  ```
  
  results in the generation of the following structure:

  ```
  DCL-DS MYBLOB;
  MYBLOB_LEN UNS(10);
  MYBLOB_DATA CHAR(500) CCSID(*HEX);
  END-DS MYBLOB;
  ```

- The following declaration in fixed-form:

  ```
  D MYBLOB S SQLTYPE(BLOB:500)
  ```

  results in the generation of the following structure:

  ```
  D MYBLOB     DS
  D MYBLOB_LEN  10U
  D MYBLOB_DATA 500A CCSID(*HEX)
  ```

**Notes:**

1. For BLOB and CLOB, \(1 \leq \text{lob-length} \leq 16,773,100\).
2. For DBCLOB, \(1 \leq \text{lob-length} \leq 8,386,550\).
3. LOB host variables are allowed to be declared in host structures.
4. LOB host variables are not allowed in host structure arrays. LOB locators should be used instead.
5. LOB host variables declared in structure arrays cannot be used as standalone host variables.
6. SQLTYPE, BLOB, CLOB, DBCLOB can be in mixed case.
7. SQLTYPE must be between positions 44 to 80 for fixed-form declarations.
8. When a LOB is declared as a stand-alone host variable, position 24 must contain the character 'S' and position 25 must be blank for fixed-form declarations.
9. The stand-alone field indicator S in position 24 for fixed-form declarations should be omitted when a LOB is declared in a host structure.
10. LOB host variables cannot be initialized.

**LOB locators in ILE RPG applications that use SQL:**

LOB, CLOB, and DBCLOB locators have similar syntax. Here is an example of a BLOB locator.

**Example: BLOB locator**

- The following declaration in free-form:

  ```
  DCL-S MYBLOB SQLTYPE(BLOB_LOCATOR);
  ```

  results in the following generation:

  ```
  DCL-S MYBLOB UNS(10);
  ```

- The following declaration in fixed-form:

  ```
  D MYBLOB S SQLTYPE(BLOB_LOCATOR)
  ```

  results in the following generation:

  ```
  D MYBLOB S 10U
  ```

**Notes:**

1. LOB locators are allowed to be declared in host structures.
2. SQLTYPE, BLOB_LOCATOR, CLOB_LOCATOR, DBCLOB_LOCATOR can be in mixed case.
3. SQLTYPE must be between positions 44 to 80 for fixed-form declarations.
4. When a LOB locator is declared as a standalone host variable, position 24 must contain the character ‘S’ and position 25 must be blank for fixed-form declarations.
5. The standalone field indicator S in position 24 for fixed-form declarations should be omitted when a LOB locator is declared in a host structure.
6. LOB locators cannot be initialized.

**LOB file reference variables in ILE RPG applications that use SQL:**

Here is an example of a CLOB file reference variable in ILE RPG. BLOB and DBCLOB file reference variables have similar syntax.

**CLOB file reference example**

- The following declaration in free-form:
  ```
  DCL-S MY_FILE SQLTYPE(CLOB_FILE);
  ```
  results in the generation of the following structure:
  ```
  DCL-DS MY_FILE;
  MY_FILE_NL UNS(10);
  MY_FILE_DL UNS(10);
  MY_FILE_FO UNS(10);
  MY_FILE_NAME CHAR(255);
  END-DS MY_FILE;
  ```
- The following declaration in fixed-form:
  ```
  D MY_FILE S SQLTYPE(CLOB_FILE)
  ```
  results in the generation of the following structure:
  ```
  D MY_FILE DS
  D MY_FILE_NL 10U
  D MY_FILE_DL 10U
  D MY_FILE_FO 10U
  D MY_FILE_NAME 255A
  ```

**Notes:**

1. LOB file reference variables are allowed to be declared in host structures.
2. SQLTYPE, BLOB_FILE, CLOB_FILE, DBCLOB_FILE can be in mixed case.
3. SQLTYPE must be between positions 44 to 80 for fixed-form declarations.
4. When a LOB file reference is declared as a standalone host variable, position 24 must contain the character ‘S’ and position 25 must be blank for fixed-form declarations.
5. The standalone field indicator ‘S’ in position 24 for fixed-form declarations should be omitted when a LOB file reference variable is declared in a host structure.
6. LOB file reference variables cannot be initialized.

The pre-compiler will generate declarations for the following file option constants. You can use these constants to set the xxx_FO variable when you use file reference host variables.

- SQFRD (2)
- SQFCRT (8)
- SQFOVR (16)
- SQFAPP (32)

**Related reference:**

[LOB file reference variables](#)
Declaring XML host variables in ILE RPG applications that use SQL:

ILE RPG does not have variables that correspond to the SQL data type for XML.

To create host variables that can be used with this data type, use the SQLTYPE keyword. The SQL precompiler replaces this declaration with an ILE RPG language structure in the output source member. XML declarations can be either standalone or within a data structure.

**XML host variables in ILE RPG applications that use SQL**

Here are some examples of XML host variables in ILE RPG applications.

**XML AS CLOB example**

- The following declaration in free-form:
  
  `DCL-S MYXMLCLOB SQLTYPE(XML_CLOB:3000);`

  results in the generation of the following structure:

  ```
  DCL-DS MYXMLCLOB;
  MYXMLCLOB_LEN UNS(10);
  MYXMLCLOB_DATA CHAR(3000);
  END-DS MYXMLCLOB;
  ```

- The following declaration in fixed-form:
  
  `D MYXMLCLOB S SQLTYPE(XML_CLOB:3000)`

  results in the generation of the following structure:

  ```
  D MYXMLCLOB DS
  D MYXMLCLOB_LEN 10U
  D MYXMLCLOB_DATA 3000A
  ```

**XML AS DBCLOB example**

- The following declaration in free-form:
  
  `DCL-S MYXMLDBCLOB SQLTYPE(XML_DBCLOB:400);`

  results in the generation of the following structure:

  ```
  DCL-DS MYXMLDBCLOB;
  MYXMLDBCLOB_LEN UNS(10);
  MYXMLDBCLOB_DATA UCS2(400) CCSID(13488);
  END-DS MYXMLDBCLOB;
  ```

- The following declaration in fixed-form:
  
  `D MYXMLDBCLOB S SQLTYPE(XML_DBCLOB:400)`

  results in the generation of the following structure:

  ```
  D MYXMLDBCLOB DS
  D MYXMLDBCLOB_LEN 10U
  D MYXMLDBCLOB_DATA 400C
  ```

**XML AS BLOB example**

- The following declaration in free-form:
  
  `DCL-S MYXMLBLOB SQLTYPE(XML_BLOB:780);`

  results in the generation of the following structure:

  ```
  DCL-DS MYXMLBLOB;
  MYXMLBLOB_LEN UNS(10);
  MYXMLBLOB_DATA CHAR(780) CCSID(+HEX);
  END-DS MYXMLBLOB;
  ```

- The following declaration in fixed-form:
  
  `D MYXMLBLOB S SQLTYPE(XML_BLOB:780)`

  results in the generation of the following structure:

  ```
  D MYXMLBLOB DS
  D MYXMLBLOB_LEN 10U
  D MYXMLBLOB_DATA 780C
  ```
results in the generation of the following structure:

D MYXMLBLOB DS
D MYXMLBLOB_LEN 10U
D MYXMLBLOB_DATA 780A CCSID(*HEX)

Notes:
1. For XML_BLOB and XML_CLOB, \(1 \leq \text{lob-length} \leq 16773\) 100.
2. For XML_DBCLOB, \(1 \leq \text{lob-length} \leq 8386\) 550.
3. XML host variables are allowed to be declared in host structures.
4. XML host variables are not allowed in host structure arrays. XML locators should be used instead.
5. XML host variables declared in structure arrays cannot be used as standalone host variables.
6. SQLTYPE, XML_BLOB, XML_CLOB, XML_DBCLOB can be in mixed case.
7. SQLTYPE must be between positions 44 to 80 for fixed-form declarations.
8. The CCSID value for an XML host variable can be explicitly set by the DECLARE VARIABLE statement. Otherwise, the value specified by the SQL_XML_DATA_CCSID QAQQINI option will be used. The default for this QAQQINI option is CCSID 1208.
9. When XML is declared as a standalone host variable, position 24 must contain the character ‘S’ and position 25 must be blank for fixed-form declarations.
10. The stand-alone field indicator S in position 24 for fixed-form declarations should be omitted when XML is declared in a host structure.
11. XML host variables cannot be initialized.

XML locators in ILE RPG applications that use SQL

Here is an example of an XML locator.

XML locator example
- The following declaration in free-form:
  
  DCL-S MYXMLLOC SQLTYPE(XML_LOCATOR);

results in the following generation:

  DCL-S MYXMLLOC UNS(10);

- The following declaration in fixed-form:
  
  D MYXMLLOC S SQLTYPE(XML_LOCATOR)

results in the following generation:

  D MYXMLLOC S 10U

Notes:
1. XML locators are allowed to be declared in host structures.
2. SQLTYPE, XML_LOCATOR can be in mixed case.
3. SQLTYPE must be between positions 44 to 80 for fixed-form declarations.
4. When an XML locator is declared as a standalone host variable, position 24 must contain the character ‘S’ and position 25 must be blank for fixed-form declarations.
5. The standalone field indicator S in position 24 for fixed-form declarations should be omitted when an XML locator is declared in a host structure.
6. XML locators cannot be initialized.

XML file reference variables in ILE RPG applications that use SQL

Here is an example of a XML file reference variable in ILE RPG.
XML file reference example

- The following declaration in free-form:
  
  DCL-S MY_XMLFILE SQLTYPE(XML_CLOB_FILE);

  results in the generation of the following structure:

  DCL-DS MY_XMLFILE;
  MY_XMLFILE_DL UNS(10);
  MY_XMLFILE_FO UNS(10);
  MY_XMLFILE_NAME CHAR(255);
  END-DS MY_XMLFILE;

- The following declaration in fixed-form:

  D MY_XMLFILE S SQLTYPE(XML_CLOB_FILE)

  results in the generation of the following structure:

  D MY_XMLFILE DS
  D MY_XMLFILE_DL 10U
  D MY_XMLFILE_FO 10U
  D MY_XMLFILE_NAME 255A

XML_BLOB_FILE and XML_DBCLOB_FILE file reference variables have similar syntax.

Notes:
1. XML file reference variables are allowed to be declared in host structures.
2. SQLTYPE, XML_BLOB_FILE, XML_CLOB_FILE, XML_DBCLOB_FILE can be in mixed case.
3. SQLTYPE must be between positions 44 to 80 for fixed-form declarations.
4. When an XML file reference is declared as a standalone host variable, position 24 must contain the character 'S' and position 25 must be blank for fixed-form declarations.
5. The standalone field indicator 'S' in position 24 for fixed-form declarations should be omitted when an XML file reference variable is declared in a host structure.
6. XML file reference variables cannot be initialized.

The precompiler will generate declarations for the following file option constants. You can use these constants to set the xxx_FO variable when you use file reference host variables.

- SQFRD (2)
- SQFCRT (8)
- SQFOVR (16)
- SQFAPP (32)

Declaring ROWID variables in ILE RPG applications that use SQL:

ILE RPG does not have a variable that corresponds to the SQL data type ROWID.

To create host variables that can be used with this data type, use the SQLTYPE keyword. The SQL precompiler replaces this declaration with an ILE RPG language declaration in the output source member. ROWID declarations can be either standalone or within a data structure.

ROWID example

- The following declaration in free-form:

  DCL-S MY_ROWID SQLTYPE(ROWID);

  results in the following generation:

  DCL_S MY_ROWID VARCHAR(40) CCSID(*HEX);

- The following declaration in fixed-form:
D MY_ROWID S SQLTYPE(ROWID)

results in the following generation:
D MY_ROWID S 40A VARYING CCSID(*HEX)

Notes:
1. SQLTYPE, ROWID can be in mixed case.
2. ROWID host variables are allowed to be declared in host structures.
3. SQLTYPE must be between positions 44 and 80 for fixed-form declarations.
4. When a ROWID is declared as a standalone host variable, position 24 must contain the character 'S' and position 25 must be blank for fixed-form declarations.
5. The standalone field indicator 'S' in position 24 for fixed-form declarations should be omitted when a ROWID is declared in a host structure.
6. ROWID host variables cannot be initialized.

Declaring result set locator variables in ILE RPG applications that use SQL:

ILE RPG does not have a variable that corresponds to the SQL result set locator data type.

To create host variables that can be used with this data type, use the SQLTYPE keyword. The SQL precompiler replaces this declaration with an ILE RPG language declaration in the output source member. Result set locator declarations can be either standalone or within a data structure.

Result set locator example
• The following declaration in free-form:
  DCL-S RS_LOC1 SQLTYPE(RESULT_SET_LOCATOR);
results in the following generation:
  DCL-S RS_LOC1 INT(20);
• The following declaration in fixed-form:
  D RS_LOC1 S SQLTYPE(RESULT_SET_LOCATOR)
results in the following generation:
  D RS_LOC1 S 20I 0

Notes:
1. SQLTYPE, RESULT_SET_LOCATOR can be in mixed case.
2. Result set locator host variables are allowed to be declared in host structures.
3. SQLTYPE must be between positions 44 and 80 for fixed-form declarations.
4. When a result set locator is declared as a standalone host variable, position 24 must contain the character 'S' and position 25 must be blank for fixed-form declarations.
5. The standalone field indicator 'S' in position 24 for fixed-form declarations should be omitted when a result set locator is declared in a host structure.
6. Result set locator host variables cannot be initialized.

Using host structures in ILE RPG applications that use SQL

The ILE RPG data structure name can be used as a host structure name. The use of the data structure name in an SQL statement implies the specification of the list of subfield names that make up the data structure.

When a data structure contains one or more unnamed subfields, the data structure name cannot be used as a host structure in an SQL statement. The named subfields can be used as host variables.

The following must be considered when using a data structure as a host structure name:
• All subfields must be valid host variable declarations.
• All subfields must have a name.
• No subfield can be defined with the DIM or OCCUR keywords.

Any subfields within a data structure which are defined out of order or that overlap earlier bytes in the data structure will be skipped by the precompiler. This includes bytes that were skipped for alignment or positioning. For example, in the following structure declaration the FIRSTNAME subfield will be skipped since it is defined to use storage prior to the LASTNAME subfield.

```
DCL-DS PEMPL;
  LASTNAME CHAR(10) POS(11);
  FIRSTNAME CHAR(10) POS(1);
END-DS;
```

In the following example, BIGCHR is an ILE data structure without subfields. SQL treats any references to BIGCHR as a character string with a length of 642.

```
DCL-DS BIGCHR LEN(642);
END-DS;
```

In the next example, PEMPL is the name of the host structure consisting of the subfields EMPNO, FIRSTN, MIDINT, LASTNA, and DEPTNO. A reference to PEMPL uses the subfields. For example, the first column of CORPDATA.EMPLOYEE is placed in EMPNO, the second column is placed in FIRSTN, and so on.

```
DCL-DS PEMPL;
  EMPNO CHAR(6);
  FIRSTN CHAR(12);
  MIDINT CHAR(1);
  LASTNA CHAR(15);
  DEPTNO CHAR(3);
END-DS;
...
EMPNO = '000220';
...
EXEC SQL SELECT * INTO :PEMPL
  FROM CORPDATA.EMPLOYEE
  WHERE EMPNO = :EMPNO;
```

When writing an SQL statement, references to subfields that are not in a QUALIFIED data structure can be qualified. Use the name of the data structure, followed by a period and the name of the subfield. For example, PEMPL.MIDINT is the same as specifying only MIDINT. If the data structure has the QUALIFIED keyword, then the subfield must be referenced using the data structure name to qualify the subfield name.

In this example, there are two data structures, one QUALIFIED and one not QUALIFIED, that contain the same subfield names:

```
DCL-DS fststruct;
  sub1 INT(5);
  sub2 INT(10);
  sub3 INT(20);
  sub4 INT(10);
END-DS;

DCL-DS secstruct QUALIFIED;
  sub1 CHAR(4);
  sub2 CHAR(12);
  sub3 INT(20);
  myvar CHAR(5);
  sub5 CHAR(20);
END-DS;

DCL-S myvar INT(10);
```
Referencing secstruct.sub1 as a host variable will be a character variable with a length of 4.

sub1 as a host variable will have an SQL data type of small integer. It picks up its attributes from the data structure that is not QUALIFIED.

A host variable reference to myvar will use the standalone declaration to pick up the data type of integer. If you use secstruct.myvar, the character variable in the QUALIFIED structure will be used.

You cannot refer to sub5 without qualifying it with secstruct because it is in a QUALIFIED data structure.

The precompiler will recognize a host structure defined using the LIKEDS keyword. However, the SQL syntax for a host variable only allows using a single level of qualification in an SQL statement. This means that if a data structure DS has a subfield S1 which is defined like a data structure with a subfield S2, an SQL statement cannot refer to S2 using the fully qualified host variable name of DS.S1.S2. If you use S1.S2 as the host variable reference, the precompiler will recognize it as DS.S1.S2. The following additional restrictions apply:

- The top level structure, DS, cannot be an array.
- S1.S2 must be unique. That is, there must be no other valid names in the program ending with S1.S2, such as a structure S1 with a subfield S1.S2, or a structure DS3 with a subfield DS3.S0.S1.S2.

Example

DCL-DS CustomerInfo QUALIFIED;
   Name CHAR(20);
   Address CHAR(50);
END-DS;

DCL-DS ProductInfo QUALIFIED;
   Number CHAR(5);
   Description CHAR(20);
   Cost PACKED(9:2);
END-DS;

DCL-DS SalesTransaction QUALIFIED;
   Buyer LIKEDS(CustomerInfo);
   Seller LIKEDS(CustomerInfo);
   NumProducts INT(10);
   Product LIKEDS(ProductInfo) DIM(10);
END-DS;

EXEC SQL
   SELECT * INTO :CustomerInfo.Name, :Buyer.Name FROM MYTABLE;

CustomerInfo.Name will be recognized as a reference to the QUALIFIED structure’s variable. Buyer.Name will be defined as SalesTransaction.Buyer.Name.

You cannot use SalesTransaction.Buyer.Name in an SQL statement because only one level of qualification is allowed in SQL syntax. You cannot use Product.Cost in an SQL statement because cost is in a dimensioned array.

If there is a SalesTransaction2 defined like SalesTransaction, then the subfields that are structures cannot be used in SQL statements. Because only one level of qualification is supported by SQL, a reference to Buyer.Name is ambiguous.

Using host structure arrays in ILE RPG applications that use SQL

A host structure array is defined as an occurrence data structure or a data structure with the keyword DIM coded. Both types of data structures can be used on the SQL FETCH or INSERT statement when processing multiple rows.
The following list of items must be considered when using a data structure with multiple row blocking support:

- All subfields must be valid host variables.
- All subfields must be contiguous. The first FROM position must be 1 and there cannot be overlaps in the TO and FROM positions.
- No subfields using the POS or OVERLAY keywords can be included in the data structure.
- If the date and time format and separator of date and time subfields within the host structure are not the same as the DATFMT, DATSEP, TIMFMT, and TIMSEP parameters on the CRTSQLRPGI command (or in the SET OPTION statement), then the host structure array is not usable.

For all statements, other than the blocked FETCH and blocked INSERT, if an occurrence data structure is used, the current occurrence is used. For the blocked FETCH and blocked INSERT, the occurrence is set to 1.

The following example uses a host structure array called DEPARTMENT and a blocked FETCH statement to retrieve 10 rows from the DEPARTMENT table.

DCL-DS DEPARTMENT OCCURS(10);
   DEPTNO CHAR(3);
   DEPTNM CHAR(29);
   MGRNO CHAR(6);
   ADMRD CHAR(3);
END-DS;

DCL-DS IND_ARRAY OCCURS(10);
   IND(Int(5)) DIM(4);
END-DS;

EXEC SQL DECLARE C1 CURSOR FOR
   SELECT * FROM CORPDATA.DEPARTMENT;

EXEC SQL FETCH C1 FOR 10 ROWS INTO :DEPARTMENT:IND_ARRAY;

Blocked FETCH and blocked INSERT are the only SQL statements that allow a data structure with the DIM keyword. Data structures with the DIM keyword cannot have a name that starts with 'TAB'. If the data structure name starts with 'TAB', the data structure is considered a table instead of an array and is not supported by SQL. A host variable reference with a subscript like MyStructure(index).Mysubfield is not supported by SQL.

Example

DCL-DS fststruct DIM(10) QUALIFIED;
   sub1 INT(5);
   sub2 INT(10);
   sub3 INT(20);
   sub4 INT(10);
END-DS;

EXEC SQL FETCH C1 FOR 10 ROWS INTO :fststruct;

Using external file descriptions in ILE RPG applications that use SQL

Field definitions for externally described files, including renaming of fields, are recognized by the SQL precompiler. The external definition form of the data structure can be used to obtain a copy of the column names to be used as host variables.

How date and time field definition are retrieved and processed by the SQL precompiler depends on whether *NOCVTDT or *CVTDT is specified on the OPTION parameter of the CRTSQLRPGI command. If *NOCVTDT is specified, then date and time field definitions are retrieved including the format and separator. If *CVTDT is specified, then the format and separator are ignored when date and time field
definitions are retrieved, and the precompiler assumes that the variable declarations are date/time host variables in character format. *CVTDT is a compatibility option for the ILE RPG precompiler.

If the GRAPHIC or VARGRAPHIC column has a UCS-2 CCSID, the generated host variable will have the UCS-2 CCSID assigned to it. If the GRAPHIC or VARGRAPHIC column has a UTF-16 CCSID, the generated host variable will have the UTF-16 CCSID assigned to it.

CLOB, BLOB, and DBCLOB columns in the external file are ignored. No host variable definition will be generated in the host structure for these types.

In the following example, the sample table DEPARTMENT is used as a file in an ILE RPG program. The SQL precompiler retrieves the field (column) definitions for DEPARTMENT for use as host variables.

```sql
DCL-F DEPARTMENT DISK(*EXT) USAGE(*INPUT) RENAME(ORIGREC:DEPTREC);
```

Note: Code an F-spec for a file in your ILE RPG program only if you use ILE RPG statements to do I/O operations to the file. If you use only SQL statements to do I/O operations to the file, you can include the external definition of the file (table) by using an external data structure.

In the following example, the sample table is specified as an external data structure. The SQL precompiler retrieves the field (column) definitions as subfields of the data structure. Subfield names can be used as host variable names, and the data structure name TDEPT can be used as a host structure name. The example shows that the field names can be renamed if required by the program.

```sql
DCL-DS TDEPT EXT
  EXTNAMES('DEPARTMENT');
    DEPTN EXTFLD('DEPTNAME')
    ADMRD EXTFLD('ADMRDEPT')
END-DS;
```

**External file description considerations for host structure arrays in ILE RPG applications that use SQL**

For device files, if INDARA was not specified and the file contains indicators, the declaration is not used as a host structure array. The indicator area is included in the structure that is generated and would cause the storage to be separated.

If OPTION(*NOCVTDT) is specified and the date and time format and separator of date and time field definitions within the file are not the same as the DATFMT, DATSEP, TIMFMT, and TIMSEP parameters on the CRTSQLRPGI command, then the host structure array is not usable.

In the following example, the DEPARTMENT table is included in the ILE RPG program and used to declare a host structure array. A blocked FETCH statement is then used to retrieve 10 rows into the host structure array.

```sql
DCL-DS DEPARTMENT EXT OCCURS(10);
END-DS;

EXEC SQL DECLARE C1 CURSOR FOR
  SELECT * FROM CORPDATA.DEPARTMENT;

EXEC SQL FETCH C1 FOR 10 ROWS INTO :DEPARTMENT;

...```

**Determining equivalent SQL and ILE RPG data types**

The precompiler determines the base SQLTYPE and SQLLEN of host variables according to this table. If a host variable appears with an indicator variable, the SQLTYPE is the base SQLTYPE plus one.
Table 9. ILE RPG declarations mapped to typical SQL data types

<table>
<thead>
<tr>
<th>RPG data type</th>
<th>RPG coding</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data structure</td>
<td>Free-form:</td>
<td>452</td>
<td>n</td>
<td>CHAR(n)</td>
</tr>
<tr>
<td>(without subfields)</td>
<td>• DCL-DS name LEN(n) END-DS;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-form:</td>
<td>• Length = n where n ≤ 32766.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zoned data</td>
<td>Free-form:</td>
<td>488</td>
<td>p in byte 1, s in byte 2</td>
<td>NUMERIC(p, s) where p is the number of digits and s is the number of decimal places</td>
</tr>
<tr>
<td></td>
<td>• ZONED(p:s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-form:</td>
<td>• Defined on Definition specification as subfield with data type S or blank.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification with data type S.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Input specification with data type S or blank.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Packed data</td>
<td>Free-form:</td>
<td>484</td>
<td>p in byte 1, s in byte 2</td>
<td>DECIMAL(p, s) where p is the number of digits and s is the number of decimal places</td>
</tr>
<tr>
<td></td>
<td>• PACKED(p:s)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed-form:</td>
<td>• Defined on Definition specification with decimal positions (pos 69-70) not blank.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification subfield with data type P.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification with data type P or blank.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Input specification with data type P.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 9. ILE RPG declarations mapped to typical SQL data types (continued)

<table>
<thead>
<tr>
<th>RPG data type</th>
<th>RPG coding</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-byte binary with zero decimal positions</td>
<td><strong>Free-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BINDEC(digits)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>where 1 &lt;= digits &lt;= 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification as subfield</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with from and to positions and data type B and byte length 2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification with data type B and digits from 1 to 4.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Input specification with data type B and byte length 2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>4-byte binary with zero decimal positions</td>
<td><strong>Free-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BINDEC(digits)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>where 5 &lt;= digits &lt;= 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification as subfield</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with from and to positions and data type B and byte length 4.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification with data type B and digits from 5 to 9.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Input specification with data type B and byte length 4.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td>2-byte integer</td>
<td><strong>Free-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• INT(5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification as subfield</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>with from and to positions and data type I and byte length 2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification with data type I and digits 5.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Input specification with data type I and byte length 2.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>500</td>
<td>2</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>RPG data type</td>
<td>RPG coding</td>
<td>SQLTYPE of host variable</td>
<td>SQLLEN of host variable</td>
<td>SQL data type</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------</td>
<td>--------------------------</td>
<td>-------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>4-byte integer</td>
<td><strong>Free-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• INT(10)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification as subfield with from and to positions and data type I and byte length 4.</td>
<td>496</td>
<td>4</td>
<td>INTEGER</td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification with data type I and digits 10.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Input specification with data type I and byte length 4.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-byte integer</td>
<td><strong>Free-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• INT(20)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification as subfield with from and to positions and data type I and byte length 8.</td>
<td>492</td>
<td>8</td>
<td>BIGINT</td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification with data type I and digits 20.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Input specification with data type I and byte length 8.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>short float</td>
<td><strong>Free-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FLOAT(4)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Data type = F, length = 4.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>long float</td>
<td><strong>Free-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FLOAT(8)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Data type = F, length = 8.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Character</td>
<td><strong>Free-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• CHAR(n)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Data type = A or blank, decimal positions blank, length between 1 and 32766.</td>
<td>452</td>
<td>n</td>
<td>CHAR (n) where n is the length</td>
</tr>
</tbody>
</table>
### Table 9. ILE RPG declarations mapped to typical SQL data types (continued)

<table>
<thead>
<tr>
<th>RPG data type</th>
<th>RPG coding</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
</table>
| Character varying length greater than 254 | **Free-form:**  
  - VARCHAR(n)  
  **Fixed-form:**  
  - Data type = A or blank, decimal positions blank, VARYING keyword on Definition specification or format *VAR on Input specification. | 448 | n | VARCHAR (n) where n is the length |
| Character varying length between 1 and 254 | **Free-form:**  
  - VARCHAR(n)  
  **Fixed-form:**  
  - Data type = A or blank, decimal positions blank, VARYING keyword on Definition specification or format *VAR on Input specification. | 456 | n | VARCHAR (n) where n is the length |
| graphic                            | **Free-form:**  
  - GRAPHR(n)  
  **Fixed-form:**  
  - Defined on Definition specification as subfield with from and to positions and data type G and byte-length b.  
  - Defined on Definition specification with data type G and length n.  
  - Defined on Input specification with data type G and byte-length b and VARYING keyword. | 468 | m | GRAPHIC(m) where m = n or m = b/2 |
| varying graphic                    | **Free-form:**  
  - VARGRAPH(n)  
  **Fixed-form:**  
  - Defined on Definition specification as subfield with from and to positions and data type G and byte-length b and VARYING keyword.  
  - Defined on Definition specification with data type G and length n and VARYING keyword.  
  - Defined on Input specification with data type G and byte-length b and format *VAR. | 464 | m | VARGRAPHIC(m) where m = n or m = (b-2)/2 |

---

IBM i: Embedded SQL programming

120
### Table 9. ILE RPG declarations mapped to typical SQL data types (continued)

<table>
<thead>
<tr>
<th>RPG data type</th>
<th>RPG coding</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
</table>
| UCS-2         | **Free-form:**  
• UCS2(n)  
|             | **Fixed-form:**  
• Defined on Definition specification as subfield with from and to positions and data type C and byte-length b.  
• Defined on Definition specification with data type C and length n.  
• Defined on Input specification with data type C and byte-length b. | 468 | m | GRAPHIC(m) with CCSID 13488 or CCSID 1200 where m = n or m = b/2 |
| varying UCS-2 | **Free-form:**  
• VARUCS2(n)  
|             | **Fixed-form:**  
• Defined on Definition specification as subfield with from and to positions and data type C and byte-length b and VARYING keyword.  
• Defined on Definition specification with data type C and length n and VARYING keyword.  
• Defined on Input specification with data type C and byte-length b and format *VAR. | 464 | m | VARGRAPHIC(m) with CCSID 13488 or CCSID 1200 where m = n or m = b/2 |
| Date          | **Free-form:**  
• DATE  
• DATE(fs)  
|             | **Fixed-form:**  
• Defined on Definition specification with data type D, format f and separator s from DATFMT keyword.  
• Defined on Input specification with data type D and format in pos 31-34, separator in pos 35. | 384 | | DATE DATFMT(f) DATSEP(s)¹ |
Table 9. ILE RPG declarations mapped to typical SQL data types (continued)

<table>
<thead>
<tr>
<th>RPG data type</th>
<th>RPG coding</th>
<th>SQLTYPE of host variable</th>
<th>SQLLEN of host variable</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Free-form:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TIME</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TIME(fs)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Definition specification with data type T, format f and separator s from TIMFMT keyword.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined on Input specification with data type T and format in pos 31-34, separator in pos 35.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>388</td>
<td></td>
<td></td>
<td>TIME TIMFMT(f)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TIMSEP(s)(^1)</td>
</tr>
<tr>
<td>Time</td>
<td>Free-form:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TIMESTAMP(n)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Data type Z.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>392</td>
<td></td>
<td></td>
<td>TIMESTAMP(n)</td>
</tr>
</tbody>
</table>

\(^1\)SQL creates the date/time subfield using the DATE/TIME format specified on the CRTSQLRPGI command. The conversion to the host variable DATE/TIME format occurs when the mapping is done between the host variables and the SQL-generated subfields.

The following table can be used to determine the RPG data type that is equivalent to a given SQL data type.

Table 10. SQL data types mapped to typical RPG declarations

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>RPG data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>Free-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• INT(5)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BINDEC(n) where 1 &lt;= n &lt;= 4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. I in position 40, length must be 5 and 0 in position 42.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. B in position 40, length must be ≤ 4 and 0 in position 42.</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>Free-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• INT(10)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• BINDEC(n) where 5 &lt;= n &lt;= 9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. I in position 40, length must be 10 and 0 in position 42.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. B in position 40, length must be ≤ 9 and ≥ 5 and 0 in position 42.</td>
<td></td>
</tr>
<tr>
<td>SQL data type</td>
<td>RPG data type</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>BIGINT</td>
<td>Free-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• INT(20)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. I in position 40, length must be 20 and 0 in position 42.</td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Free-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• PACKED(p:s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. P in position 40 or blank in position 40 for a non-subfield, 0 through 63 in position 41,42.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Defined as numeric on non-definition specification.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>maximum length of 32 (precision 63) and maximum scale of 63.</td>
<td></td>
</tr>
<tr>
<td>NUMERIC</td>
<td>Free-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ZONED(p:s)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. S in position 40 or blank in position 40 for a subfield, 0 through 63 in position 41,42.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum length of 63 (precision 63) and maximum scale of 63.</td>
<td></td>
</tr>
<tr>
<td>DECFLOAT</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>FLOA (single precision)</td>
<td>Free-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FLOA(4)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. F in position 40, length must be 4.</td>
<td></td>
</tr>
<tr>
<td>FLOA (double precision)</td>
<td>Free-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FLOA(8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. F in position 40, length must be 8.</td>
<td></td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>Free-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• CHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. A or blank in positions 40 and blanks in position 41,42.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Input field defined without decimal places.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Calculation result field defined without decimal places.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>n can be from 1 to 32766.</td>
<td></td>
</tr>
</tbody>
</table>
Table 10. SQL data types mapped to typical RPG declarations (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>RPG data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td><strong>Free-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• DCL-DS name LEN(n) END-DS;</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Data structure name with no subfields in the data structure. n can be from 1 to 32766.</td>
<td></td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td><strong>Free-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• VARCHAR(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. A or blank in position 40 and VARYING in positions 44-80. n can be from 1 to 32740.</td>
<td></td>
</tr>
<tr>
<td>CLOB</td>
<td>Not supported</td>
<td>Use SQLTYPE keyword to declare a CLOB.</td>
</tr>
<tr>
<td>GRAPHIC(n)</td>
<td><strong>Free-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• GRAPH(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. G in position 40. • Input field defined with G in position 36. n can be 1 to 16383.</td>
<td></td>
</tr>
<tr>
<td>VARGRAPHIC(n)</td>
<td><strong>Free-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• VARGRAPH(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification. G in position 40 and VARYING in positions 44-80. n can be from 1 to 16370.</td>
<td></td>
</tr>
<tr>
<td>DBCLOB</td>
<td>Not supported</td>
<td>Use SQLTYPE keyword to declare a DBCLOB.</td>
</tr>
<tr>
<td>BINARY</td>
<td>Not supported</td>
<td>Use SQLTYPE keyword to declare a BINARY.</td>
</tr>
<tr>
<td>VARBINARY</td>
<td>Not supported</td>
<td>Use SQLTYPE keyword to declare a VARBINARY.</td>
</tr>
<tr>
<td>BLOB</td>
<td>Not supported</td>
<td>Use SQLTYPE keyword to declare a BLOB.</td>
</tr>
<tr>
<td>DATE</td>
<td><strong>Free-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• DATE</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Fixed-form:</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A character field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification with a D in position 40. • Input field defined with D in position 36.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the format is *USA, *JIS, *EUR, or *ISO, the length must be at least 10. If the format is *YMD, *DMY, or *MDY, the length must be at least 8. If the format is *JUL, the length must be at least 6.</td>
<td></td>
</tr>
</tbody>
</table>

IBM i: Embedded SQL programming
Table 10. SQL data types mapped to typical RPG declarations (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>RPG data type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>Free-form:</td>
<td>Length must be at least 6; to include seconds, length must be at least 8.</td>
</tr>
<tr>
<td></td>
<td>TIME</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A character field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification with a T in position 40.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Input field defined with T in position 36.</td>
<td></td>
</tr>
<tr>
<td>TIMESTAMP(n)</td>
<td>Free-form:</td>
<td>For a character field, the length must be at least 19; to include fractional seconds, the length can be 21 to 32. If less than 32, some fractional seconds will be truncated.</td>
</tr>
<tr>
<td></td>
<td>TIMESTAMP(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Fixed-form:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• A character field</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Definition specification with a Z in position 40.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Input field defined with Z in position 36.</td>
<td></td>
</tr>
<tr>
<td>XML</td>
<td>Not supported</td>
<td>Use SQLTYPE keyword to declare an XML.</td>
</tr>
<tr>
<td>DATALINK</td>
<td>Not supported</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>Not supported</td>
<td>Use SQLTYPE keyword to declare a ROWID.</td>
</tr>
<tr>
<td>Result set locator</td>
<td>Not supported</td>
<td>Use SQLTYPE keyword to declare a result set locator.</td>
</tr>
</tbody>
</table>

Notes on ILE RPG variable declaration and usage
ILE RPG associates precision and scale with all numeric types.

ILE RPG defines numeric operations, assuming the data is in packed format. This means that operations involving binary variables include an implicit conversion to packed format before the operation is performed (and back to binary, if necessary). Data is aligned to the implied decimal point when SQL operations are performed.

Using indicator variables in ILE RPG applications that use SQL
An indicator variable is a binary field with length less than 5 (2 bytes).

An indicator array can be defined by declaring the variable element length of 4,0 and specifying the DIM on the definition specification.

Indicator variables are declared in the same way as host variables and the declarations of the two can be mixed in any way that seems appropriate to the programmer.

Related reference:
References to variables
“Indicator variables in applications that use SQL” on page 4
An indicator variable is a halfword integer variable used to communicate additional information about its associated host variable.
Example: Using indicator variables in ILE RPG applications that use SQL

Here is an example of declaring indicator variables in ILE RPG.

Given the statement:
EXEC SQL FETCH CLS_CURSOR INTO :CLSCD,
     :DAY :DAYIND,
     :BGN :BGNIND,
     :END :ENDIND;

variables can be declared as follows:
DCL-S CLSCD CHAR(7);
DCL-S DAY INT(5);
DCL-S DAYIND INT(5);
DCL-S BGN CHAR(8);
DCL-S BGNIND INT(5);
DCL-S END CHAR(8);
DCL-S ENDIND INT(5)

Example: SQLDA for a multiple row-area fetch in ILE RPG applications that use SQL

Here is an example of the SQL descriptor area (SQLDA) for a multiple row-area fetch in ILE RPG.

DCL-C SQL_NUM CONST(4);
EXEC SQL INCLUDE SQLDA;

DCL-DS DEPARTMENT OCCURS(10);
    DEPTNO CHAR(3);
    DEPTNM CHAR(29);
    MGRNO CHAR(6);
    ADMRD CHAR(3);
END-DS;

DCL-DS IND_ARRAY OCCURS(10);
    INDs INT(5) DIM(4);
END-DS;

// setup number of sqlda entries and length of the sqlda
sqln = 4;
sqlln = 4;
sqldabc = 336;

// setup the first entry in the sqlda
sqltype = 453;
sqllen = 3;
sq_l_var(1) = sqlvar;

// setup the second entry in the sqlda
sqltype = 453;
sqllen = 29;
sq_l_var(2) = sqlvar;
...
// setup the fourth entry in the sqlda
sqltype = 453;
sqllen = 3;
sq_l_var(4) = sqlvar;
...
EXEC SQL DECLARE C1 CURSOR FOR
    SELECT * FROM CORPDATA.DEPARTMENT;
...
EXEC SQL FETCH C1 FOR 10 ROWS
    USING DESCRIPTOR :SQLDA
    INTO :DEPARTMENT:IND_ARRAY;

126   IBM i: Embedded SQL programming
Example: Dynamic SQL in an ILE RPG application that uses SQL

Here is an example of using dynamic SQL in ILE RPG.

```sql
//**********************************************************
//* Declare program variables.                             *
//* STMT initialized to the                              *
//* listed SQL statement.                                *
//**********************************************************
DCL-S EMPNUM CHAR(6);
DCL-S NAME CHAR(15);
DCL-S STMT CHAR(500)
  INZ('SELECT LASTNAME FROM CORPDATA.EMPLOYEE +
   WHERE EMPNO = ?');
...
//**********************************************************
//* Prepare STMT as initialized in declare section       *
//**********************************************************
EXEC SQL PREPARE S1 FROM :STMT;

//**********************************************************
//* Declare Cursor for STMT                             *
//**********************************************************
EXEC SQL DECLARE C1 CURSOR FOR S1;

//**********************************************************
//* Assign employee number to use in select statement    *
//**********************************************************
EMPNUM = '000110';

//**********************************************************
//* Open Cursor                                        *
//**********************************************************
EXEC SQL OPEN C1 USING :EMPNUM;

//**********************************************************
//* Fetch record and put value of                        *
//* LASTNAME into NAME                                   *
//**********************************************************
EXEC SQL FETCH C1 INTO :NAME;

//**********************************************************
//* Program processes NAME here                         *
//**********************************************************
...

EXEC SQL CLOSE C1;
```

Coding SQL statements in REXX applications

REXX procedures do not have to be preprocessed. At run time, the REXX interpreter passes statements that it does not understand to the current active command environment for processing.

The command environment can be changed to `*EXECSQL` to send all unknown statements to the database manager in two ways:
1. CMDENV parameter on the STRREXPRC CL command
2. address positional parameter on the ADDRESS REXX command

For more information about the STRREXPRC CL command or the ADDRESS REXX command, see the REXX/400 Programmer's Guide topic and the REXX/400 Reference topic.
Note: By using the code examples, you agree to the terms of the Code license and disclaimer information on page 190.

Related concepts:
- Writing applications that use SQL on page 2
You can create database applications in host languages that use Db2 for i SQL statements and functions.

Related reference:
- Example programs: Using Db2 for i statements on page 149
Here is a sample application that shows how to code SQL statements in each of the languages that Db2 for i supports.
- Example: SQL statements in REXX programs on page 183
This example program is written in the REXX programming language.

Using the SQL communication area in REXX applications
The fields that make up the SQL communication area (SQLCA) are automatically included by the SQL/REXX interface.

An INCLUDE SQLCA statement is not required and is not allowed. The SQLCODE and SQLSTATE fields of the SQLCA contain SQL return codes. These values are set by the database manager after each SQL statement is run. An application can check the SQLCODE or SQLSTATE value to determine whether the last SQL statement was successful.

The SQL/REXX interface uses the SQLCA in a manner consistent with the typical SQL usage. However, the SQL/REXX interface maintains the fields of the SQLCA in separate variables rather than in a contiguous data area. The variables that the SQL/REXX interface maintains for the SQLCA are defined as follows:

**SQLCODE**
- The primary SQL return code.

**SQLERRMC**
- Error and warning message tokens.

**SQLERRP**
- Product code and, if there is an error, the name of the module that returned the error.

**SQLERRD\_n**
- Six variables (n is a number between 1 and 6) containing diagnostic information.

**SQLWARN\_n**
- Eleven variables (n is a number between 0 and 10) containing warning flags.

**SQLSTATE**
- The alternate SQL return code.

Related reference:
- SQL communication area

Using SQL descriptor areas in REXX applications
There are two types of SQL descriptor areas. One is defined with the ALLOCATE DESCRIPTOR statement. The other is defined using the SQL descriptor area (SQLDA) structure. Only the SQLDA form is discussed here. Allocated descriptors are not supported in REXX.

The following statements can use an SQLDA:
- EXECUTE...USING DESCRIPTOR descriptor-name
- FETCH...USING DESCRIPTOR descriptor-name
- OPEN...USING DESCRIPTOR descriptor-name
CALL...USING DESCRIPTOR descriptor-name
DESCRIBE statement-name INTO descriptor-name
DESCRIBE TABLE host-variable INTO descriptor-name

Unlike the SQLCA, more than one SQLDA can be in a procedure, and an SQLDA can have any valid name.

Each SQLDA consists of a set of REXX variables with a common stem, where the name of the stem is the descriptor-name from the appropriate SQL statements. This must be a simple stem; that is, the stem itself must not contain any periods. The SQL/REXX interface automatically provides the fields of the SQLDA for each unique descriptor name. An INCLUDE SQLDA statement is not required and is not allowed.

The SQL/REXX interface uses the SQLDA in a manner consistent with the typical SQL usage. However, the SQL/REXX interface maintains the fields of the SQLDA in separate variables rather than in a contiguous data area.

The following variables are returned to the application after a DESCRIBE, a DESCRIBE TABLE, or a PREPARE INTO statement:

stem.n.SQLNAME
  The name of the nth column in the result table.
  The following variables must be provided by the application before an EXECUTE...USING DESCRIPTOR, an OPEN...USING DESCRIPTOR, a CALL...USING DESCRIPTOR, or a FETCH...USING DESCRIPTOR statement. They are returned to the application after a DESCRIBE, a DESCRIBE TABLE, or a PREPARE INTO statement:

stem.SQLD
  Number of variable elements that the SQLDA actually contains.

stem.n.SQLTYPE
  An integer representing the data type of the nth element (for example, the first element is in stem.1.SQLTYPE).
  The following data types are not allowed:
  400/401
    NUL-terminated graphic string
  404/405
    BLOB host variable
  408/409
    CLOB host variable
  412/413
    DBCLOB host variable
  460/461
    NUL-terminated character string
  476/477
    PASCAL L-string
  496/497
    Large integer (where scale is greater than 0)
  500/501
    Small integer (where scale is greater than 0)
  504/505
    DISPLAY SIGN LEADING SEPARATE
ROWID
VARBINARY host variable
BINARY host variable
BLOB file reference variable
CLOB file reference variable
DBCLOB file reference variable
BLOB locator
CLOB locator
DBCLOB locator
Result set locator
XML host variable
Decimal floating point host variable
XML locator
stem.n.SQLDA
This contains the input value supplied by the application, or the output value fetched by SQL.
This value is converted to the attributes specified in SQLTYPE, SQLLEN, SQLPRECISION, and SQLSCALE.
stem.n.SQLIND
If the input or output value is null, this is a negative number.
Related reference:
130 IBM i: Embedded SQL programming
Embedding SQL statements in REXX applications

An SQL statement can be placed anywhere a REXX command can be placed.

Each SQL statement in a REXX procedure must begin with EXECSQL (in any combination of uppercase and lowercase letters), followed by either:

- The SQL statement enclosed in single or double quotation marks, or
- A REXX variable containing the statement. Note that a colon must not precede a REXX variable when it contains an SQL statement.

For example:

```rexx
EXECSQL "COMMIT"
```

is equivalent to:

```rexx
rexxvar = "COMMIT"
EXECSQL rexxvar
```

The command follows normal REXX rules. For example, it can optionally be followed by a semicolon (;) to allow a single line to contain more than one REXX statement. REXX also permits command names to be included within single quotation marks, for example:

```rexx
'EXECSQL COMMIT'
```

The SQL/REXX interface supports the following SQL statements:

```
ALTER FUNCTION                   EXECUTE
ALTER PROCEDURE                 EXECUTE IMMEDIATE
ALTER SEQUENCE
ALTER TABLE
CALL
CLOSE
COMMENT
COMMIT
CREATE ALIAS
CREATE FUNCTION
CREATE INDEX
CREATE PROCEDURE
CREATE SCHEMA
CREATE SEQUENCE
CREATE TABLE
CREATE TRIGGER
DECLARE CURSOR
DECLARE GLOBAL TEMPORARY TABLE
DELETE
DESCRIBE
DESCRIBE TABLE
DROP
EXECUTE
FETCH
GRANT
INSERT
LABEL
LOCK TABLE
OPEN
PREPARE
REFRESH TABLE
RELEASE SAVEPOINT
RENAME
REVOKE
ROLLBACK
SAVEPOINT
SET CURRENT DECIMAL Rounding Mode
SET CURRENT IMPLICIT XMLPARSE OPTION
SET CURRENT SCHEMA PASSWORD
SET ENCRYPTION option
SET path
SET schema
SET transaction
SET variable
UPDATE
```

The following SQL statements are not supported by the SQL/REXX interface:

---

**SQL descriptor area**
1. The blocked form of this statement is not supported.
2. These statements cannot be run directly if they contain host variables; they must be the object of a PREPARE and then an EXECUTE.
3. The SET OPTION statement can be used in a REXX procedure to change some of the processing options used for running SQL statements. These options include the commitment control level and date format. See the DB2 for i SQL reference topic for more information about the SET OPTION statement.
4. See “Handling errors and warnings in REXX applications that use SQL” on page 133 for more information.

Comments in REXX applications that use SQL
Neither SQL comments (--) nor REXX comments are allowed in strings representing SQL statements.

Continuation of SQL statements in REXX applications that use SQL
The string containing an SQL statement can be split into several strings on several lines, separated by commas or concatenation operators, according to standard REXX usage.

Including code in REXX applications that use SQL
Unlike the other host languages, support is not provided for including externally defined statements.

Margins in REXX applications that use SQL
There are no special margin rules for the SQL/REXX interface.

Names in REXX applications that use SQL
Any valid REXX name not ending in a period (.) can be used for a host variable. The name must be 64 characters or less.

Variable names should not begin with the characters ‘SQL’, ‘RDI’, ‘DSN’, ‘RXSQL’, or ‘QRW’.

Nulls in REXX applications that use SQL
Although the term null is used in both REXX and SQL, the term has different meanings in the two languages.

REXX has a null string (a string of length zero) and a null clause (a clause consisting only of blanks and comments). The SQL null value is a special value that is distinct from all non-null values and denotes the absence of a (non-null) value.

Statement labels in REXX applications that use SQL
REXX command statements can be labeled as usual.
Handling errors and warnings in REXX applications that use SQL

The WHENEVER statement is not supported by the SQL/REXX interface. You can use one of several substitutes, however.

Any of the following may be used instead:

- A test of the REXX SQLCODE or SQLSTATE variables after each SQL statement to detect error and warning conditions issued by the database manager, but not for those issued by the SQL/REXX interface.
- A test of the REXX RC variable after each SQL statement to detect error and warning conditions. Each use of the EXEC SQL command sets the RC variable to:
  
  0 Statement completed successfully.
  +10 A SQL warning occurred.
  -10 An SQL error occurred
  -100 An SQL/REXX interface error occurred.

  This can be used to detect errors and warnings issued by either the database manager or by the SQL/REXX interface.

- The SIGNAL ON ERROR and SIGNAL ON FAILURE facilities can be used to detect errors (negative RC values), but not warnings.

Using host variables in REXX applications that use SQL

REXX does not provide for variable declarations.

LOB, XML, ROWID, binary, and result set locator host variables are not supported in REXX. New variables are recognized by their appearance in assignment statements. Therefore, there is no declare section, and the BEGIN DECLARE SECTION and END DECLARE SECTION statements are not supported.

All host variables within an SQL statement must be preceded by a colon (:).

The SQL/REXX interface performs substitution in compound variables before passing statements to the database manager. For example:

```
a = 1
b = 2
EXEC SQL 'OPEN c1 USING :a.b'
```

causes the contents of x.1.2 to be passed to SQL.

Determining data types of input host variables in REXX applications that use SQL

All data in REXX is in the form of strings.

The data type of input host variables (that is, host variables used in a 'USING host variable' clause in an EXECUTE or OPEN statement) is inferred by the database manager at run time from the contents of the variable according to the table below.

These rules define either numeric, character, or graphic values. A numeric value can be used as input to a numeric column of any type. A character value can be used as input to a character column of any type, or to a date, time, or timestamp column. A graphic value can be used as input to a graphic column of any type.
Table 11. Determining data types of host variables in REXX

<table>
<thead>
<tr>
<th>Host variable contents</th>
<th>Assumed data type</th>
<th>SQL type code</th>
<th>SQL type description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A number with neither decimal point nor exponent. It can have a leading plus or minus</td>
<td>Signed integers</td>
<td>496/497</td>
<td>INTEGER</td>
</tr>
<tr>
<td>sign.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A number that includes a decimal point, but no exponent, or a number that does not</td>
<td>Packed decimal</td>
<td>484/485</td>
<td>DECIMAL(m,n)</td>
</tr>
<tr>
<td>include a decimal point or an exponent and is greater than 2147483647 or smaller than</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-2147483647.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>It can have a leading plus or minus sign. m is the total number of digits in the</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>number. n is the number of digits to the left of the decimal point (if any).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A number that is in scientific or engineering notation (that is, followed immediately</td>
<td>Floating point</td>
<td>480/481</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>by an 'E' or 'e', an optional plus or minus sign, and a series of digits). It can</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>have a leading plus or minus sign.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A string with leading and trailing single quotation marks (') or quotation marks</td>
<td>Varying-length</td>
<td>448/449</td>
<td>VARCHAR(n)</td>
</tr>
<tr>
<td>(&quot;), which has length n after removing the two delimiters, or a string with a leading</td>
<td>character string</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X or x followed by a single quotation mark (') or quotation mark (&quot;), and a trailing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>single quotation mark (') or quotation mark (&quot;). The string has a length of 2n after</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>removing the X or x and the two delimiters. Each remaining pair of characters is the</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hexadecimal representation of a single character.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>or a string of length n, which cannot be recognized as character, numeric, or graphic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>through other rules in this table</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 11. Determining data types of host variables in REXX (continued)

<table>
<thead>
<tr>
<th>Host variable contents</th>
<th>Assumed data type</th>
<th>SQL type code</th>
<th>SQL type description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A string with a leading and trailing single quotation mark ('') or quotation marks (&quot;&quot;&quot;) preceded by:</td>
<td>Varying-length graphic string</td>
<td>464/465</td>
<td>VARGRAPHIC(n)</td>
</tr>
<tr>
<td>• A string that starts with a G, g, N, or n. This is followed by a single quotation mark or a quotation mark and a shift-out (X'0E') character. This is followed by n graphic characters, each 2 characters long. The string must end with a shift-in (X'0F') character and a single quotation mark or a quotation mark (whichever the string started with).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• A string with a leading GX, Gx, gX, or gx, followed by a single quotation mark or a quotation mark and a shift-out (X'0E') character. This is followed by n graphic characters, each 2 characters long. The string must end with a shift-in (X'0F') character and a single quotation mark or a quotation mark (whichever the string started with). The string has a length of 4n after removing the GX and the delimiters. Each remaining group of 4 characters is the hexadecimal representation of a single graphic character.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undefined Variable</td>
<td>Variable for which a value has not been assigned</td>
<td>None</td>
<td>Data that is not valid was detected.</td>
</tr>
</tbody>
</table>

1 The byte immediately following the leading single quotation mark is a X'0E' shift-out character, and the byte immediately preceding the trailing single quotation mark is a X'0F' shift-in character.

The format of output host variables in REXX applications that use SQL

It is not necessary to determine the data type of an output host variable (that is, a host variable used in an ‘INTO host variable’ clause in a FETCH statement).

Output values are assigned to host variables as follows:

• Character values are assigned without leading and trailing apostrophes.

• Graphic values are assigned without a leading G or apostrophe, without a trailing apostrophe, and without shift-out and shift-in characters.

• Numeric values are translated into strings.

• Integer values do not retain any leading zeros. Negative values have a leading minus sign.

• Decimal values retain leading and trailing zeros according to their precision and scale. Negative values have a leading minus sign. Positive values do not have a leading plus sign.

• Floating-point values are in scientific notation, with one digit to the left of the decimal place. The 'E' is in uppercase.

Avoiding REXX conversion in REXX applications that use SQL

To guarantee that a string is not converted to a number or assumed to be of graphic type, strings should be enclosed in ‘“”’. Enclosing the string in single quotation marks does not work.

For example:

```plaintext
stringvar = '100'
```
causes REXX to set the variable `stringvar` to the string of characters 100 (without the single quotation marks). This is evaluated by the SQL/REXX interface as the number 100, and it is passed to SQL as such.

On the other hand,
```
stringvar = "'100"
```
causes REXX to set the variable `stringvar` to the string of characters '100' (with the single quotation marks). This is evaluated by the SQL/REXX interface as the string 100, and it is passed to SQL as such.

**Using indicator variables in REXX applications that use SQL**

An indicator variable is an integer.

Unlike other languages, a valid value must be specified in the host variable even if its associated indicator variable contains a negative value.

**Related reference:**

References to variables

"Indicator variables in applications that use SQL" on page 4

An indicator variable is a halfword integer variable used to communicate additional information about its associated host variable.

**Preparing and running a program with SQL statements**

This topic describes some of the tasks for preparing and running an application program.

**Related concepts:**

"Writing applications that use SQL" on page 2

You can create database applications in host languages that use Db2 for i SQL statements and functions.

**Basic processes of the SQL precompiler**

You must precompile and compile an application program containing embedded SQL statements before you can run it.

**Note:** SQL statements in a REXX procedure are not precompiled and compiled.

Precompiling of such programs is done by the SQL precompiler. The SQL precompiler scans each statement of the application program source and does the following:

- **Looks for SQL statements and for the definition of host variable names.** The variable names and definitions are used to verify the SQL statements. You can examine the listing after the SQL precompiler completes processing to see if any errors occurred.

- **Verifies that each SQL statement is valid and free of syntax errors.** The validation procedure supplies error messages in the output listing that help you correct any errors that occur.

- **Validates the SQL statements using the description in the database.** During the precompile, SQL statements are checked for valid table, column, and other object references. If a specified object does not exist or you are not authorized to it at the time of the precompile, complete validation will be done at run time. If an object does not exist at run time, an error occurs.

**Notes:**

1. Overrides are processed when retrieving external definitions.
2. You need some authority (at least *OBJOPR) to any tables or views referred to in the SQL statements in order to validate the SQL statements. The actual authority required to process any SQL statement is checked at run time.
3. When the RDB parameter is specified on the CRTSQLxxx commands, the precompiler accesses the specified relational database to obtain the table and view descriptions.
• **Prepares each SQL statement for compilation in the host language.** For most SQL statements, the SQL precompiler inserts a comment and a CALL statement to one of the SQL interface modules. For some SQL statements (for example, DECLARE statements), the SQL precompiler produces no host language statement except a comment.

• **Produces information about each precompiled SQL statement.** The information is stored internally in a temporary source file member, where it is available for use during the bind process.

To get complete diagnostic information when you precompile, specify either of the following:

- OPTION(*SOURCE *XREF) for CRTSQLxxx (where xxx=CBL, PLI, or RPG)
- OPTION(*XREF) OUTPUT(*PRINT) for CRTSQLxxx (where xxx=CI, CPPI, CBLI, or RPGI)

**Related concepts:**
- Database programming
- Database file management
- DB2 for i5/OS SQL reference

**Input to the SQL precompiler**

Application programming statements and embedded SQL statements are the primary input to the SQL precompiler. The statements can be in a source member or, for any ILE precompile, in a source stream file.

In PL/I, C, and C++ source members, the SQL statements must use the margins that are specified in the MARGINS parameter of the CRTSQLPLI, CRTSQLCI, and CRTSQLCPPI commands. The MARGINS parameter is ignored when you precompile from a source stream file.

The SQL precompiler assumes that the host language statements are syntactically correct. If the host language statements are not syntactically correct, the precompiler might not correctly identify SQL statements and host variable declarations. Literals and comments that are not accepted by the application language compiler can interfere with the precompiler source scanning process and cause errors.

You can use the SQL INCLUDE statement to embed secondary input from the file that is specified by the INCFILE or INCDIR parameter of the CRTSQLxxx command. The SQL INCLUDE statement causes the specified member or source stream file to be read. The included source cannot contain other precompiler INCLUDE statements, but can contain both application program and SQL statements.

When you precompile a source member, the INCFILE parameter is used to find the source that is specified in the SQL INCLUDE statement. When you precompile a source stream file, the INCDIR parameter is used. If a relative path is specified in the INCLUDE statement, the precompiler first searches the current directory. If that file is not found, the name specified on the INCLUDE statement is appended to the INCDIR value. If that is not found, the precompiler searches the directory where the input source is found. If an absolute path is specified for the INCLUDE statement, the precompiler ignores the INCDIR value. No suffixes are appended to the name specified on the INCLUDE statement.

If mixed DBCS constants are specified in the application program source, the source file must be a mixed CCSID.

You can specify many of the precompiler command parameter values directly in the input source by using the SQL SET OPTION statement. These include options such as DATFMT, COMMIT, and NAMING. By specifying them in the input source, you do not need to remember to specify them on the precompiler command.

**Note:** If a value is provided for an option on both the precompile command and on the SET OPTION statement, the value from the SET OPTION statement is used.
The RPG preprocessor options (RPGPOPT) parameter of the **CRTSQLRPGI** command has two options to call the RPG preprocessor. If *LVL1 or *LVL2 is specified, the RPG compiler will be called to preprocess the source before the SQL precompile is run. Preprocessing the SQL source will allow many compiler directives to be handled before the SQL precompile. The preprocessed source will be placed in file QSQLPRE in QTEMP. This source will be used as the input for the SQL precompile. The CCSID used by the SQL precompile is the CCSID of QSQLPRE.

**Related reference:**

| SET OPTION | Create SQL ILE RPG Object (CRTSQLRPGI) command |

**Source file CCSIDs in the SQL precompiler**

The SQL precompiler reads the source records by using the CCSID of the source file or source stream file.

When processing SQL INCLUDE statements, the include source is converted to the CCSID of the primary source if necessary. If the include source cannot be converted to the CCSID of the primary source, an error occurs.

The SQL precompiler processes SQL statements using the source CCSID. This affects variant characters the most. For example, the not sign (¬) is located at 'BAX' in CCSID 500. This means that if the CCSID of your source file is 500, SQL expects the not sign (¬) to be located at 'BA'.

If the source file CCSID is 65535, SQL processes variant characters as if they had a CCSID of 37. This means that SQL looks for the not sign (¬) at '5FX'.

**Output from the SQL precompiler**

The SQL precompiler generates two pieces of output: a listing and a source file number.

**Listing:**

The output listing is sent to the printer file that is specified by the PRTFILE parameter of the CRTSQLxxx command.

The following items are written to the printer file:

- Precompiler options
  - Options specified in the CRTSQLxxx command.
- Precompiler source
  - This output supplies precompiler source statements with the record numbers that are assigned by the precompiler, if the listing option is in effect.
- Precompiler cross-reference
  - If *XREF was specified in the OPTION parameter, this output supplies a cross-reference listing. The listing shows the precompiler record numbers of SQL statements that contain the referred to host names and column names.
- Precompiler diagnostics
  - This output supplies diagnostic messages, showing the precompiler record numbers of statements in error.
  - The output to the printer file will use a CCSID value of 65535. The data will not be converted when it is written to the printer file.

**Temporary source file members created by the SQL precompiler:**

Source statements processed by the precompiler are written to an output source file.
In the precompiler-changed source code, SQL statements have been converted to comments and calls to the SQL run time code. Include files that are processed by SQL are expanded.

The output source file is specified on the CRTSQLxxx command in the TOSRCFILE parameter.

- For languages other than C, C++, and ILE RPG, the default file is QSQLTEMP in the QTEMP library.
- For C and C++ when *CALC is specified as the output source file, QSQLTEMP will be used if the source file's record length is 92 or less. For a C or C++ source file where the record length is greater than 92, the output source file name will be generated as QSQLTxxxxx, where xxxx is the record length.
- For RPG when *CALC is specified as the output source file, QSQLTEMP1 will be used if the source file's record length is 112 or less. For an RPG source file where the record length is greater than 112, the output source file name will be generated as QSQLTxxxxx, where xxxx is the record length.

The name of the output source file member is the same as the name specified in the PGM or OBJ parameter of the CRTSQLxxx command. This member cannot be changed before being used as input to the compiler. When SQL creates the output source file, it uses the CCSID value of the source file as the CCSID value for the new file.

If the precompile generates output in a source file in QTEMP, the file can be moved to a permanent library after the precompile if you want to compile at a later time. You cannot change the records of the source member, or the attempted compile fails.

The source member that is generated by SQL as the result of the precompile should never be edited and reused as an input member to another precompile step. The additional SQL information that is saved with the source member during the first precompile will cause the second precompile to work incorrectly. Once this information is attached to a source member, it stays with the member until the member is deleted.

The SQL precompiler uses the CRTSRCPF command to create the output source file. If the defaults for this command have changed, then the results may be unpredictable. If the source file is created by the user, not the SQL precompiler, the file's attributes may be different as well. It is recommended that the user allow SQL to create the output source file. Once it has been created by SQL, it can be reused on later precompiles.

**Sample SQL precompiler output:**

The precompiler output can provide information about your program source.

To generate the listing:

- For non-ILE precompilers, specify the *SOURCE (*SRC) and *XREF options on the OPTION parameter of the CRTSQLxxx command.
- For ILE precompilers, specify OPTION(*XREF) and OUTPUT(*PRINT) on the CRTSQLxxx command.

The format of the precompiler output is:
Figure 2. Sample COBOL precompiler output format

1 A list of the options you specified when the SQL precompiler was called.
2 The date the source member was last changed.
/* This program will get the average education level and the average salary by department. */

EXEC SQL
WHENEVER SQLERROR GO TO B000-SQL-ERROR
END-EXEC.

WHENEVER statement to handle SQL errors.
EXEC SQL
DECLARE CURSOR FOR
SELECT WORKDEPT, AVG(EDLEVEL), AVG(SALARY) 
FROM CORPDATA.EMPLOYEE 
GROUP BY WORKDEPT
END-EXEC.

EXEC SQL
OPEN CURSOR
END-EXEC.

PERFORM A010-FETCH-PROCEDURE THROUGH A010-FETCH-EXIT
UNTIL SQLCODE IS = 100.

EXEC SQL
CLOSE CURSOR
END-EXEC.

CLOSE OUTFILE.
STOP RUN.

A010-FETCH-PROCEDURE.
EXEC SQL
FETCH CURS INTO :AVG-RECORD
END-EXEC.

IF SQLCODE IS = 0
MOVE WORKDEPT TO DEPT-NO
MOVE AVG-SALARY TO AVERAGE-SALARY
MOVE AVG-EDUC TO AVERAGE-EDUCATION-LEVEL
WRITE REC-1 AFTER ADVANCING 1 LINE.

A010-FETCH-EXIT.
EXIT.

B000-SQL-ERROR.
MOVE SPACES TO ERROR-RECORD.
MOVE SQLCODE TO ERROR-CODE.
MOVE "AN SQL ERROR HAS OCCURRED" TO ERROR-MESSAGE.
WRITE ERROR-RECORD AFTER ADVANCING 1 LINE.
CLOSE OUTFILE.
STOP RUN.

* * * * * END OF SOURCE * * * * *
Data names are the symbolic names used in source statements.

The define column specifies the line number at which the name is defined. The line number is generated by the SQL precompiler. **** means that the object was not defined or the precompiler did not recognize the declarations.

The reference column contains two types of information:
- The definition of the symbolic name (4)
- The line numbers where the symbolic name occurs (5)

If the symbolic name refers to a valid host variable, the data-type (6) or data-structure (7) is also noted.

Non-ILE SQL precompiler commands

The IBM DB2 Query Manager and SQL Development Kit for i licensed program includes non-ILE precompiler commands for the following host languages: CRSTCLBL (for OPM COBOL), CRSTQLPLI (for PL/I PRPQ), and CRSTLRPG (for RPG III, which is part of RPG/400).
Some options only apply to certain languages. For example, the options *APOST and *QUOTE are unique to COBOL. They are not included in the commands for the other languages.

Related concepts:
“CL command descriptions for host language precompilers” on page 188

The IBM DB2 Query Manager and SQL Development Kit for i licensed program provides commands for precompiling programs coded in these programming languages.

Compiling a non-ILE application program that uses SQL

The SQL precompiler automatically calls the host language compiler after the successful completion of a precompile, unless *NOGEN is specified.

The CRTxxxPGM command is run specifying the program name, source file name, precompiler created source member name, text, and USRPRF.

Within these languages, the following parameters are passed:

- For COBOL, the *QUOTE or *APOST is passed on the CRTCBLPGM command.
- For RPG and COBOL, SAAFLAG (*FLAG) is passed on the CRTxxxPGM command.
- For RPG and COBOL, the SRTSEQ and LANGID parameter from the CRTSQLxxx command is specified on the CRTxxxPGM command.
- For RPG and COBOL, the CVTOPT (*DATETIME *VARCHAR) is always specified on the CRTxxxPGM command.
- For COBOL and RPG, the TGTRLS parameter value from the CRTSQLxxx command is specified on the CRTxxxPGM command. TGTRLS is not specified on the CRTCBLPGM command. The program can be saved or restored to the level specified on the TGTRLS parameter of the CRTSQLPLI command.
- For PL/I, the MARGINS are set in the temporary source file.
- For all languages, the REPLACE parameter from the CRTSQLxxx command is specified on the CRTxxxPGM command.

If a package is created as part of the precompile process, the REPLACE parameter value from the CRTSQLxxx command is specified on the CRTSQLPKG command.

- For all languages, if USRPRF(*USER) or system naming (*SYS) with USRPRF(*NAMING) is specified, then USRPRF(*USER) is specified on the CRTxxxPGM command. If USRPRF(*OWNER) or SQL naming (*SQL) with USRPRF(*NAMING) is specified, then USRPRF(*OWNER) is specified on the CRTxxxPGM command.

Defaults are used for all other parameters with CRTxxxPGM commands.

You can interrupt the call to the host language compiler by specifying *NOGEN on the OPTION parameter of the precompiler command. *NOGEN specifies that the host language compiler will not be called. Using the object name in the CRTSQLxxx command as the member name, the precompiler created the source member in the output source file (specified as the TOSRCFILE parameter on the CRTSQLxxx command). You now can explicitly call the host language compilers, specify the source member in the output source file, and change the defaults. If the precompile and compile were done as separate steps, the CRTSQLPKG command can be used to create the SQL package for a distributed program.

Note: You must not change the source member in QTEMP/QSQLTEMP prior to issuing the CRTxxxPGM command or the compile will fail.

ILE SQL precompiler commands

In the IBM DB2 Query Manager and SQL Development Kit for i licensed program, these ILE precompiler commands exist: CRTSQLCI, CRTSQLCPP, CRTSQLCBL, and CRTSQLRPGI.

A precompiler command exists for each of the host languages: ILE C, ILE C++, ILE COBOL, and ILE RPG. For each command, you can specify the required parameters and use the defaults for the remaining
parameters. Some options are applicable only to one language. The defaults are applicable only to the language you are using. For example, the options *APOST and *QUOTE are unique to COBOL. They are not included in the commands for the other languages.

Related concepts:
"CL command descriptions for host language precompilers" on page 188

The IBM DB2 Query Manager and SQL Development Kit for i licensed program provides commands for precompiling programs coded in these programming languages.

Compiling an ILE application program that uses SQL

The SQL precompiler automatically calls the host language compiler after the successful completion of a precompile for the CRTSQLxxx commands, unless *NOGEN is specified.

If the *MODULE option is specified, the SQL precompiler issues the CRTxxxMOD command to create the module. If the *PGM option is specified, the SQL precompiler issues the CRTBNDxxx command to create the program. If the *SRVPGM option is specified, the SQL precompiler issues the CRTxxxMOD command to create the module, followed by the Create Service Program (CRTSRVPGM) command to create the service program. The CRTSQLCPPI command only creates *MODULE objects.

Within these languages, the following parameters are passed:

• If DBGVIEW(*SOURCE) is specified on the CRTSQLxxx command, then DBGVIEW(*ALL) is specified on both the CRTxxxMOD and CRTBNDxxx commands.

• If OUTPUT(*PRINT) is specified on the CRTSQLxxx command, it is passed on both the CRTxxxMOD and CRTBNDxxx commands.

If OUTPUT(*NONE) is specified on the CRTSQLxxx command, it is not specified on either the CRTxxxMOD command or the CRTBNDxxx command.

• The TGTRLS parameter value from the CRTSQLxxx command is specified on the CRTxxxMOD, CRTBNDxxx, and Create Service Program (CRTSRVPGM) commands.

• The REPLACE parameter value from the CRTSQLxxx command is specified on the CRTxxxMOD, CRTBNDxxx, and CRTSRVPGM commands.

If a package is created as part of the precompile process, the REPLACE parameter value from the CRTSQLxxx command is specified on the CRTSQLPKG command.

• If OBJTYPE is either *PGM or *SRVPGM, and USRPRF(*USER) or system naming (*SYS) with USRPRF(*NAMING) is specified, USRPRF(*USER) is specified on the CRTBNDxxx or the CRTSRVPGM commands.

If OBJTYPE is either *PGM or *SRVPGM, and USRPRF(*OWNER) or SQL naming (*SQL) with USRPRF(*NAMING) is specified, USRPRF(*OWNER) is specified on the CRTBNDxxx or the CRTSRVPGM commands.

• The DBGENCKEY value from the CRTSQLxxx command is specified on the CRTxxxMOD or CRTBNDxxx commands.

• For C and C++, the MARGINS are set in the temporary source file.

If the precompiler calculates that the total length of the LOB host variables is close to 15M, the TERASPACEx (*YES *TSIFC) option is specified on the CRTCMOD, CRTBNDC, or CRTCPPMOD commands.

• For C and C++, the DECFLTRND value is passed on the CRTCMOD, CRTBNDC, or CRTCPPMOD commands.

• For COBOL, the *QUOTE or *APOST is passed on the CRTBNDCBL or the CRTCBLMOD commands.

• For RPG and COBOL, the SRTSEQ and LANGID parameter from the CRTSQLxxx command is specified on the CRTxxxMOD and CRTBNDxxx commands.

• For COBOL, CVTOPT(*VARCHAR *DATETIME *PICGGRAPHIC *FLOAT) is always specified on the CRTCBLMOD and CRTBNDCBL commands. If OPTION(*NOCVTDT) is specified (the shipped command default), the additional options *DATE *TIME *TIMESTAMP are also specified for the CVTOPT.

• For RPG, if OPTION(*CVTDT) is specified, then CVTOPT(*DATETIME) is specified on the CRTRPGMOD and CRTBNDRPG commands.
You can interrupt the call to the host language compiler by specifying *NOGEN on the OPTION parameter of the precompiler command. *NOGEN specifies that the host language compiler is not called. Using the specified program name in the CRTSQLxxx command as the member name, the precompiler creates the source member in the output source file (TOSRCFILE parameter). You can now explicitly call the host language compiler, specify the source member in the output source file, and change the defaults. If the precompile and compile were done as separate steps, the CRTSQLPKG command can be used to create the SQL package for a distributed program.

If the program or service program is created later, the USRPRF parameter may not be set correctly on the CRTBNDxxx, Create Program (CRTPGM), or Create Service Program (CRTSRVPGM) command. The SQL program runs predictably only after the USRPRF parameter is corrected. If system naming is used, then the USRPRF parameter must be set to *USER. If SQL naming is used, then the USRPRF parameter must be set to *OWNER.

**Setting compiler options using the precompiler commands**

The COMPILEOPT string is available on the precompiler command and on the SET OPTION statement to allow additional parameters to be used on the compiler command.

The COMPILEOPT string is added to the compiler command built by the precompiler. This allows specifying compiler parameters without requiring a two step process of precompiling and then compiling. Do not specify parameters in the COMPILEOPT string that the SQL precompiler passes. Doing so will cause the compiler command to fail with a duplicate parameter error. It is possible that the SQL precompiler will pass additional parameters to the compiler in the future. This could lead to a duplicate parameter error, requiring your COMPILEOPT string to be changed at that time.

If "*EVENTF" is anywhere in the COMPILEOPT string specified on the precompiler command, the precompiler will behave as if "OPTION(*EVENTF)" was specified on the precompiler command. This does not apply to the COMPILEOPT string specified on the SET OPTION statement.

If "INCDIR(" is anywhere in the COMPILEOPT string, the precompiler will call the compiler using the SRCSTMF parameter.

```
EXEC SQL SET OPTION COMPILEOPT = 'OPTION(*SHOWINC +EXPMAC)
INCDIR('',/QSYS.LIB/MYLIB.LIB/MYFILE.MBR '')';
```

**Interpreting compile errors in applications that use SQL**

Sometimes you will encounter compile errors. Use the following information to interpret these errors.

If you separate the precompile and compile steps, and the source program refers to externally described files, the referred-to files must not be changed between precompile and compile steps. Otherwise, results that are not predictable might occur because the changes to the field definitions are not changed in the temporary source member.

Examples of externally described files are:
- COPY DDS in COBOL
- %INCLUDE in PL/I
- #pragma mapinc and #include in C or C++
- Externally-described files and externally-described data structures in RPG

When the SQL precompiler does not recognize host variables or host structures, messages SQL0312 and SQL5011 are written to the listing. These messages return a detailed reason why the variable was not able to be used. If the information in the message does not resolve the problem, you can try compiling the source. The compiler will not recognize the EXEC SQL statements, ignore these errors. Verify that the compiler interprets the host variable declaration as defined by the SQL precompiler for that language.
Binding an application that uses SQL

Before you can run your application program, a relationship between the program and any specified tables and views must be established. This process is called binding. The result of binding is an access plan.

The access plan is a control structure that describes the actions necessary to satisfy each SQL request. An access plan contains information about the program and about the data the program intends to use.

For a nondistributed SQL program, the access plan is stored in the program. For a distributed SQL program (where the RDB parameter is specified on the CRTSQLxxx command), the access plan is stored in the SQL package at the specified relational database.

SQL automatically attempts to bind and create access plans when the program object is created. For non-ILE compilations, this occurs as the result of running a successful CRTxxxPGM command. For ILE compilations, this occurs as the result of running a successful CRTBNDxxx, CRTPGM, or CRTSRVPGM command. If Db2 for i detects at run time that an access plan is not valid (for example, the referenced tables are in a different library) or detects that changes have occurred to the database that might improve performance (for example, the addition of indexes), a new access plan is automatically created. Binding does the following things:

1. **It revalidates the SQL statements using the description in the database.** During the bind process, the SQL statements are checked for valid table, column, and other object names. If a specified table or object does not exist at the time of the precompile or compile, the validation is done at run time. If the table or object does not exist at run time, a negative SQLCODE is returned.

2. **It selects the index needed to access the data your program wants to process.** In selecting an index, table sizes, and other factors are considered. It considers all indexes available to access the data and decides which ones (if any) to use when selecting a path to the data.

3. **It attempts to build access plans.** For each SQL statement that is valid, the bind process builds and stores an access plan in the program.

If the characteristics of a table or view your program accesses have changed, the access plan may no longer be valid. When you attempt to run a program that contains an access plan that is not valid, the system automatically attempts to rebuild the access plan. If the access plan cannot be rebuilt, a negative SQLCODE is returned. In this case, you might have to change the program's SQL statements and reissue the CRTSQLxxx command to correct the situation.

Assume that a program contains an SQL statement that refers to COLUMNNA in TABLEA and the user deletes and re-creates TABLEA so that COLUMNNA no longer exists. When you call the program, the automatic rebind will be unsuccessful because COLUMNNA no longer exists. In this case you must change the program source and reissue the CRTSQLxxx command.

Program references in applications that use SQL

All schemas, tables, views, SQL packages, and indexes referenced in SQL statements in an SQL program are placed in the object information repository (OIR) of the library when the program is created.

You can use the CL command **Display Program References (DSPGMREF)** to display all object references in the program. If the SQL naming convention is used, the library name is stored in the OIR in one of three ways:

1. **If the SQL name is fully qualified, the schema name is stored as the name qualifier.**
2. **If the SQL name is not fully qualified and the DFTRDBCOL parameter is not specified, the authorization ID of the statement is stored as the name qualifier.**
3. **If the SQL name is not fully qualified and the DFTRDBCOL parameter is specified, the schema name specified on the DFTRDBCOL parameter is stored as the name qualifier.**

If the system naming convention is used, the library name is stored in the OIR in one of three ways:
1. If the object name is fully qualified, the library name is stored as the name qualifier.
2. If the object is not fully qualified and the DFTRDBCOL parameter is not specified, *LIBL is stored.
3. If the SQL name is not fully qualified and the DFTRDBCOL parameter is specified, the schema name specified on the DFTRDBCOL parameter is stored as the name qualifier.

Displaying SQL precompiler options

When the SQL application program is successfully compiled, the Display Module (DSPMOD), the Display Program (DSPPGM), or the Display Service Program (DSPSRVPGM) command can be used to determine some of the options that were specified on the SQL precompile.

This information may be needed when the source of the program has to be changed. These same SQL precompiler options can then be specified on the CRTSQLxxx command when the program is compiled again.

The Print SQL Information (PRTSQLINF) command can also be used to determine some of the options that were specified on the SQL precompile.

Running a program with embedded SQL

Running a host language program with embedded SQL statements, after the precompile and compile have been successfully done, is the same as running any host program.

Enter the following CALL statement:

CALL pgm-name

on the system command line.

Note: After installing a new release, users may encounter message CPF2218 in QHST using any Structured Query Language (SQL) program if the user does not have *CHANGE authority to the program. Once a user with *CHANGE authority calls the program, the access plan is updated and the message will be issued.

Related concepts:
- Control language

Running a program with embedded SQL: DDM considerations

SQL does not support remote file access through distributed data management (DDM) files, except for the case when a RDB is specified in the DDM file. SQL does support remote access through Distributed Relational Database Architecture™ (DRDA).

Running a program with embedded SQL: Override considerations

You can use overrides (specified by the OVRDBF command) to direct a reference to a different table or view or to change certain operational characteristics of the program or SQL Package.

The following parameters are processed if an override is specified:

- TOFILE
- MBR
- SEQONLY
- INHWRT
- WAITRCD

All other override parameters are ignored. Overrides of statements in SQL packages are accomplished by doing both of the following:
1. Specifying the OVRSCOPE(*JOB) parameter on the OVRDBF command

IBM i: Embedded SQL programming
2. Sending the command to the application server by using the Submit Remote Command (SBMRMTCMD) command

To override tables and views that are created with long names, you can create an override using the system name that is associated with the table or view. When the long name is specified in an SQL statement, the override is found using the corresponding system name.

An alias is actually created as a DDM file. You can create an override that refers to an alias name (DDM file). In this case, an SQL statement that refers to the file that has the override actually uses the file to which the alias refers.

Related concepts:
- Database programming
- Database file management

Running a program with embedded SQL: SQL return codes

An SQL return code is sent by the database manager after the completion of each SQL statement. Your program can check the SQLCODE or SQLSTATE after every SQL statement.

Related concepts:
- SQL messages and codes

Example programs: Using Db2 for i statements

Here is a sample application that shows how to code SQL statements in each of the languages that Db2 for i supports.

The sample application gives raises based on commission.

Each sample program produces the same report, which is shown at the end of this topic. The first part of the report shows, by project, all employees working on the project who received a raise. The second part of the report shows the new salary expense for each project.

Notes about the sample programs

The following notes apply to all the sample programs:

- SQL statements can be entered in uppercase or lowercase.

1. This host language statement retrieves the external definitions for the SQL table PROJECT. These definitions can be used as host variables or as a host structure.

Notes:
1. In RPG/400, field names in an externally described structure that are longer than 6 characters must be renamed.
2. REXX does not support the retrieval of external definitions.

2. The SQL INCLUDE SQLCA statement is used to include the SQLCA for PL/I, C, and COBOL programs. For RPG programs, the SQL precompiler automatically places the SQLCA data structure into the source at the end of the Input specification section. For REXX, the SQLCA fields are maintained in separate variables rather than in a contiguous data area mapped by the SQLCA.

3. This SQL WHENEVER statement defines the host language label to which control is passed if an SQLERROR (SQLCODE < 0) occurs in an SQL statement. This WHENEVER SQLERROR statement applies to all the following SQL statements until the next WHENEVER SQLERROR statement is encountered. REXX does not support the WHENEVER statement. Instead, REXX uses the SIGNAL ON ERROR facility.
This SQL UPDATE statement updates the SALARY column, which contains the employee salary by the percentage in the host variable PERCENTAGE (PERCNT for RPG). The updated rows are those that have employee commissions greater than 2000. For REXX, this is PREPARE and EXECUTE since UPDATE cannot be run directly if there is a host variable.

This SQL COMMIT statement commits the changes made by the SQL UPDATE statement. Record locks on all changed rows are released.

Note: The program was precompiled using COMMIT(*CHG). (For REXX, *CHG is the default.)

This SQL DECLARE CURSOR statement defines cursor C1, which joins two tables, EMPLOYEE and EMPLOYEE, and returns rows for employees who received a raise (commission > 2000). Rows are returned in ascending order by project number and employee number (PROJNO and EMPNO columns). For REXX, this is PREPARE and DECLARE CURSOR since the DECLARE CURSOR statement cannot be specified directly with a statement string if it has host variables.

This SQL OPEN statement opens cursor C1 so that the rows can be fetched.

This SQL WHENEVER statement defines the host language label to which control is passed when all rows are fetched (SQLCODE = 100). For REXX, the SQLCODE must be explicitly checked.

After all rows are fetched, control is passed to this label. The SQL CLOSE statement closes cursor C1.

This SQL DECLARE CURSOR statement defines cursor C2, which joins the three tables, EMPLOYEE, PROJECT, and EMPLOYEE. The results are grouped by columns PROJNO and PROJNAME. The COUNT function returns the number of rows in each group. The SUM function calculates the new salary cost for each project. The ORDER BY 1 clause specifies that rows are retrieved based on the contents of the final results column (EMPLOYEE.PROJNO). For REXX, this is a PREPARE and DECLARE CURSOR since the DECLARE CURSOR statement cannot be specified directly with a statement string if it has host variables.

This SQL FETCH statement returns the results columns for cursor C2 and places the returned values into the corresponding elements of the host structure.

This SQL WHENEVER statement with the CONTINUE option causes processing to continue to the next statement regardless if an error occurs on the SQL ROLLBACK statement. Errors are not expected on the SQL ROLLBACK statement; however, this prevents the program from going into a loop if an error does occur. REXX does not support the WHENEVER statement. Instead, REXX uses the SIGNAL OFF ERROR facility.

This SQL ROLLBACK statement restores the table to its original condition if an error occurred during the update.

Related concepts:
- “Coding SQL statements in C and C++ applications” on page 11
- “Coding SQL statements in COBOL applications” on page 44
- “Coding SQL statements in PL/I applications” on page 73
- “Coding SQL statements in ILE RPG applications” on page 98

To embed SQL statements in an ILE C or C++ program, you need to be aware of some unique application and coding requirements. This topic also defines the requirements for host structures and host variables.

There are unique application and coding requirements for embedding SQL statements in a COBOL program. In this topic, requirements for host structures and host variables are defined.

There are some unique application and coding requirements for embedding SQL statements in a PL/I program. In this topic, requirements for host structures and host variables are defined.

You need to be aware of the unique application and coding requirements for embedding SQL statements in an ILE RPG program. In this topic, the coding requirements for host variables are defined.
The RPG/400 licensed program supports both RPG II and RPG III programs.

REXX procedures do not have to be preprocessed. At run time, the REXX interpreter passes statements that it does not understand to the current active command environment for processing.

**Example: SQL statements in ILE C and C++ programs**

This example program is written in the C programming language.

The same program would work in C++ if the following conditions are true:

- An SQL BEGIN DECLARE SECTION statement was added before line 18
- An SQL END DECLARE SECTION statement was added after line 42

**Note:** By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 190.
Figure 3. Sample C program using SQL statements
/* A sample program which updates the salaries for those employees */
/* whose current commission total is greater than or equal to the */
/* value of 'commission'. The salaries of those who qualify are */
/* increased by the value of 'percentage', retroactive to 'raise_date'.*/
/* A report is generated showing the projects that these employees */
/* have contributed to, ordered by project number and employee ID. */
/* A second report shows each project having an end date occurring */
/* after 'raise_date' (is potentially affected by the retroactive */
/* raises) with its total salary expenses and a count of employees */
/* who contributed to the project. */

short work_days = 253;  /* work days during in one year */
float commission = 2000.00;  /* cutoff to qualify for raise */
float percentage = 1.04;  /* raised salary as percentage */
char raise_date??(12??) = "1982-06-01";  /* effective raise date */
FILE *qprint;

/* Structure for report 1 */
#include "project","CORPDATA/PROJECT(PROJECT)" both p z"
#include "project"

struct {  
  CORPDATA_PROJECT PROJECT_both_t Proj_struct;
  char empno??(7??);
  char name??(30??);
  float salary;
  int rpt1;
}
rpt1;

/* Structure for report 2 */
struct {  
  char projno??(7??);
  char project_name??(37??);
  int short employee_count;
  double total_proj_cost;
  int rpt2;
}
rpt2;

2 exec sql include SQLCA;
qprint=fopen("QPRINT","w");

/* Update the selected projects by the new percentage. If an error */
/* occurs during the update, ROLLBACK the changes. */
EXEC SQL WHENEVER SQLERROR GO TO update_error;
EXEC SQL UPDATE CORPDATA/EMPLOYEE
SET SALARY = SALARY * :percentage
WHERE COMM = :commission
COMMIT;
EXEC SQL WHENEVER SQLERROR GO TO report_error;

EXEC

/* Report the updated statistics for each employee assigned to the */
/* selected projects. */
/* Write out the header for Report 1 */
fprintf(qprint," REPORT OF PROJECTS AFFECTED \nPROJECT EMPID EMPLOYEE NAME ");
fprintf(qprint, " SALARY\n");

6 exec sql
   declare cl cursor for
   select distinct projno, empproject.empno,
       lastname||'.'||firstname, salary
   from corpdata/empproject, corpdata/employee
   where empproject.empno = employee.empno and comm >= :commission
   order by projno, empno;
7 EXEC SQL
   OPEN Cl;
8 /* Fetch and write the rows to QPRINT */
8 EXEC SQL WHENEVER NOT FOUND GO TO done1;
10 do {
10 EXEC SQL
   FETCH Cl INTO :Proj_struct.PROJNO, :rpt1.empno,
       :rpt1.name,rpt1.salary;
   fprintf(qprint,"\n\n%30s %8s %-30s %8.2f",
        C1, Proj_struct.PROJNO,rpt1.empno,
        rpt1.name,rpt1.salary);
   } while (SQLCODE==0);
12 done1:

/* For all projects ending at a date later than the 'raise date' */
/* (that is, those projects potentially affected by the salary raises), */
/* generate a report containing the project number, project name */
/* the count of employees participating in the project, and the */
/* total salary cost of the project. */
/* Write out the header for Report 2 */
fprintf(qprint," ACCUMULATED STATISTICS\nPROJECT NUMBER PROJECT NAME EMPLOYEES COST\n");

11 EXEC SQL
11 DECLARE C2 CURSOR FOR
12 SELECT EMPPROJECT.PROJNO, PROJNAME, COUNT(*),
   SUM ( ( DAYS(EMENDATE) - DAYS(EMSTDATE) ) * EMPTIME *
   (DECIMAL( SALARY / :work_days ,8,2) ) )
   FROM CORPDATA/EMPPROJECT, CORPDATA/PROJECT, CORPDATA/EMPLOYEE
   WHERE EMPPROJECT.PROJNO=PROJECT.PROJNO AND
   EMPPROJECT.EMPNO=EMPLOYEE.EMPNO AND
   PREDATE > :raise date
   GROUP BY EMPPROJECT.PROJNO, PROJNAME
   ORDER BY 1;
12 EXEC SQL
12 OPEN C2;
/* Fetch and write the rows to QPRINT */
EXEC SQL WHENEVER NOT FOUND GO TO done2;

do {
  FETCH C2 INTO :rpt2;
  fprintf(qprint, "%6s %-36s %6d %9.2f",
          rpt2.projno, rpt2.project_name, rpt2.employee_count,
          rpt2.total_proj_cost);
}
while (SQLCODE==0);

done2:
EXEC SQL CLOSE C2;
goto finished;

/* Error occurred while updating table. Inform user and rollback */
EXEC SQL WHENEVER SQLERROR CONTINUE;
fprintf(qprint, "*** ERROR Occurred while updating table. SQLCODE=
      \"%5d\n\",SQLCODE);
ROLLBACK;
goto finished;

/* Error occurred while generating reports. Inform user and exit. */
EXEC SQL WHENEVER SQLERROR CONTINUE;
fprintf(qprint, "*** ERROR Occurred while generating reports. SQLCODE=\n      \"%5d\n\",SQLCODE);
goto finished;

*/ All done */
finished:
fclose(qprint);
exit(0);

* * * * * E N D   O F   S O U R C E * * * * *
CROSS REFERENCE

Data Names  Define  Reference
commission  19  FLOAT(24)
            54 75
done1       ****  LABEL
            81
done2       ****  LABEL
            126
employee_count  40  SMALL INTEGER PRECISION(4,0) IN rpt2
empno       31  VARCHAR(7) IN rpt1
            85
name        32  VARCHAR(30) IN rpt1
            86
percentage  20  FLOAT(24)
            53
project_name  39  VARCHAR(37) IN rpt2
projno      38  VARCHAR(7) IN rpt2
raise_date  21  VARCHAR(12)
            119
report_error ****  LABEL
            59
rpt1        34
rpt2        42  STRUCTURE
            130
salary      33  FLOAT(24) IN rpt1
            86
total_proj_cost  41  FLOAT(53) IN rpt2
update_error ****  LABEL
            50
work_days   18  SMALL INTEGER PRECISION(4,0)
            115
ACTNO       74  SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN CORPDATA.EMPPROJACT
BIRTHDATE   74  DATE(10) COLUMN IN CORPDATA.EMPLOYEE
BONUS       74  DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE
COMM        ****  COLUMN
            54 75
COMM        74  DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE
CORPDATA    ****  SCHEMA
            52 74 74 116 116 116
C1          71  CURSOR
            78 85 95
C2          112  CURSOR
            123 130 139
DEPTNO      27  VARCHAR(3) IN Proj struct
DEPTNO      116  CHARACTER(3) COLUMN (NOT NULL) IN CORPDATA.PROJECT
EDLEVEL     74  SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
EMENDATE   74  DATE(10) COLUMN IN CORPDATA.EMPPROJACT
EMENDATE   ****  COLUMN
            114
EMPLOYEE    ****  TABLE IN CORPDATA
            52 74 116
EMPLOYEE    ****  TABLE
            75 118
EMPNO       ****  COLUMN IN EMPPROJACT
            72 75 76 118
EMPNO       ****  COLUMN IN EMPLOYEE
            75 118
EMPNO       74  CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.EMPPROJACT
EMPNO       74  CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
EMPPROJACT  ****  TABLE
            72 75 113 117 118 120
EMPPROJACT  ****  TABLE IN CORPDATA
            74 116
To embed SQL statements in an ILE C or C++ program, you need to be aware of some unique application and coding requirements. This topic also defines the requirements for host structures and host variables.

Example: SQL statements in COBOL and ILE COBOL programs

This example program is written in the COBOL programming language.

Note: By using the code examples, you agree to the terms of the Code license and disclaimer information.
Figure 4. Sample COBOL program using SQL statements
A sample program that updates the salaries for those employees whose current commission total is greater than or equal to the value of COMMISSION. The salaries of those who qualify are increased by the value of PERCENTAGE retroactive to RAISE-DATE. A report is generated showing the projects that these employees have contributed to ordered by project number and employee ID. A second report shows each project having an end date occurring after RAISE-DATE (that is, potentially affected by the retroactive raises) with its total salary expenses and a count of employees who contributed to the project.

 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLEX.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-AS400.
 OBJECT-COMPUTER. IBM-AS400.
 INPUT-OUTPUT SECTION.

 FILE-CONTROL.
   SELECT PRINTFILE ASSIGN TO PRINTER-QPRINT
   ORGANIZATION IS SEQUENTIAL.

 DATA DIVISION.
 FILE SECTION.

 FD PRINTFILE
   BLOCK CONTAINS 1 RECORDS
   LABEL RECORDS ARE OMITTED.
 01 PRINT-RECORD PIC X(132).

 WORKING-STORAGE SECTION.
 77 WORK-DAYS PIC S9(4) BINARY VALUE 253.
 77 RAISE-DATE PIC X(11) VALUE "1982-06-01".
 77 PERCENTAGE PIC S999V99 PACKED-DECIMAL.
 77 COMMISSION PIC S99999V99 PACKED-DECIMAL VALUE 2000.00.

*******************************************************************************
* Structure for report 1. *
*******************************************************************************
 1 01 RPT1.
    COPY DDS-PROJECT OF CORPDATA-PROJECT.
   05 EMPNO PIC X(6).
   05 NAME PIC X(30).
   05 SALARY PIC S9(6)V99 PACKED-DECIMAL.
01 RPT2.
   15 PROJNO PIC X(6).
   15 PROJECT-NAME PIC X(36).
   15 EMPLOYEE-COUNT PIC S9(4) BINARY.
   15 TOTAL-PROJ-COST PIC S9(10)V99 PACKED-DECIMAL.

2 EXEC SQL
   INCLUDE SQLCA
   END-EXEC.

01 RPT1-HEADERS.
  05 RPT1-HEADER1.
    10 FILLER PIC X(21) VALUE SPACES.
    10 FILLER PIC X(111) VALUE "REPORT OF PROJECTS AFFECTED BY RAISES".
  05 RPT1-HEADER2.
    10 FILLER PIC X(9) VALUE "PROJECT".
    10 FILLER PIC X(10) VALUE "EMPID".
    10 FILLER PIC X(35) VALUE "EMPLOYEE NAME".
    10 FILLER PIC X(40) VALUE "SALARY".

01 RPT2-HEADERS.
  05 RPT2-HEADER1.
    10 FILLER PIC X(21) VALUE SPACES.
    10 FILLER PIC X(111) VALUE "ACCUMULATED STATISTICS BY PROJECT".
  05 RPT2-HEADER2.
    10 FILLER PIC X(9) VALUE "PROJECT".
    10 FILLER PIC X(38) VALUE SPACES.
    10 FILLER PIC X(16) VALUE "NUMBER OF".
    10 FILLER PIC X(10) VALUE "TOTAL".
  05 RPT2-HEADER3.
    10 FILLER PIC X(9) VALUE "NUMBER".
    10 FILLER PIC X(38) VALUE "PROJECT NAME".
    10 FILLER PIC X(16) VALUE "EMPLOYEES".
    10 FILLER PIC X(65) VALUE "COST".

01 RPT1-DATA.
  05 PROJNO PIC X(6).
  05 EMPNO PIC X(6).
  05 NAME PIC X(30).
  05 PROJECT-NAME PIC X(36).
  05 EMPLOYEE-COUNT PIC S99.
  05 FILLER PIC X(96) VALUE SPACES.

01 RPT2-DATA.
  05 PROJNO PIC X(6).
  05 EMPNO PIC X(6).
  05 EMPLOYEE-COUNT PIC ZZZ9.
  05 PROJECT-NAME PIC X(36).
  05 EMPLOYEE-COUNT PIC ZZ9.
  05 FILLER PIC X(5) VALUE SPACES.
  05 TOTAL-PROJ-COST PIC ZZZZZZZZZ9.99.
  05 FILLER PIC X(56) VALUE SPACES.
PROCEDURE DIVISION.

A000-MAIN.

MOVE 1.04 TO PERCENTAGE.

OPEN OUTPUT PRINTFILE.

**********************************************************************
/* Update the selected employees by the new percentage. If an */
/* error occurs during the update, roll back the changes. */
**********************************************************************

EXEC SQL WHENEVER SQLERROR GO TO E010-UPDATE-ERROR
END-EXEC.

EXEC SQL UPDATE CORPDATA/EMPLOYEE
SET SALARY = SALARY * :PERCENTAGE
WHERE COMM >= :COMMISSION
END-EXEC.

**********************************************************************
/* Commit changes. */
**********************************************************************

EXEC COMMIT
END-EXEC.

EXEC SQL WHENEVER SQLERROR GO TO E020-REPORT-ERROR
END-EXEC.

**********************************************************************
/* Report the updated statistics for each employee receiving */
/* a raise and the projects that the employee participates in */
**********************************************************************

**********************************************************************
/* Write out the header for Report 1. */
**********************************************************************

write print-record from rpt1-header1
before advancing 2 lines.
write print-record from rpt1-header2
before advancing 1 line.

EXEC SQL DECLARE C1 CURSOR FOR
SELECT DISTINCT projno, empprojact.empno,
lastname||"||firstnme, salary
FROM CORPDATA/EMPPROJECT, CORPDATA/EMPLOYEE
WHERE EMPPROJECT.EMPNO = EMPLOYEE.EMPNO AND
COMM >= :COMMISSION
ORDER BY projno, empno
END-EXEC.

EXEC SQL OPEN C1
END-EXEC.

PERFORM B000-GENERATE-REPORT1 THRU B010-GENERATE-REPORT1-EXIT
UNTIL SQLCODE NOT EQUAL TO ZERO.
EXEC SQL
CLOSE C1
END-EXEC.

******************************************************************************
* For all projects ending at a date later than the RAISE-DATE (that is, those projects potentially affected by the salary raises), generate a report containing the project number, project name, the count of employees participating in the project, and the total salary cost for the project.
******************************************************************************

******************************************************************************
* Write out the header for Report 2.
******************************************************************************

MOVE SPACES TO PRINT-RECORD.
WRITE PRINT-RECORD BEFORE ADVANCING 2 LINES.
WRITE PRINT-RECORD FROM RPT2-HEADER1 BEFORE ADVANCING 2 LINES.
WRITE PRINT-RECORD FROM RPT2-HEADER2 BEFORE ADVANCING 1 LINE.
WRITE PRINT-RECORD FROM RPT2-HEADER3 BEFORE ADVANCING 2 LINES.
EXEC SQL
11 DECLARE C2 CURSOR FOR
SELECT EMPPROJACT.PROJNO, PROJNAME, COUNT(*),
SUM ( (DAYS(EMENDATE)-DAYS(EMSTDATE)) * EMPTIME * DECIMAL((SALARY / :WORK-DAYS),8,2))
FROM CORPDATA/EMPPROJACT, CORPDATA/PROJECT, CORPDATA/EMPLOYEE
WHERE EMPPROJACT.PROJNO=PROJECT.PROJNO AND EMPPROJACT.EMPNO = EMPLOYEE.EMPNO AND PRENDATE > :RAISE-DATE
GROUP BY EMPPROJACT.PROJNO, PROJNAME
ORDER BY 1
END-EXEC.
EXEC SQL
OPEN C2
END-EXEC.
PERFORM C000-GENERATE-REPORT2 THRU C010-GENERATE-REPORT2-EXIT
UNTIL SQLCODE NOT EQUAL TO ZERO.
EXEC SQL
CLOSE C2
END-EXEC.

******************************************************************************
* All done.
******************************************************************************

A900-MAIN-EXIT.
CLOSE PRINTFILE.
STOP RUN.
B000-GENERATE-REPORT1.
8 EXEC SQL
WHENEVER NOT FOUND GO TO A100-DONE1
END-EXEC.
9 EXEC SQL
FETCH C1 INTO :PROJECT.PROJNO, :RPT1.EMPNO,
:RPT1.NAME, :RPT1.SALARY
END-EXEC.
MOVE CORRESPONDING RPT1 TO RPT1-DATA.
MOVE PROJNO OF RPT1 TO PROJNO OF RPT1-DATA.
WRITE PRINT-RECORD FROM RPT1-DATA BEFORE ADVANCING 1 LINE.
B010-GENERATE-REPORT1-EXIT.
EXIT.

C000-GENERATE-REPORT2.
EXEC SQL
WHENEVER NOT FOUND GO TO A200-DONE2
END-EXEC.
12 EXEC SQL
FETCH C2 INTO :RPT2
END-EXEC.
MOVE CORRESPONDING RPT2 TO RPT2-DATA.
WRITE PRINT-RECORD FROM RPT2-DATA BEFORE ADVANCING 1 LINE.
C010-GENERATE-REPORT2-EXIT.
EXIT.

E010-UPDATE-ERROR.
13 EXEC SQL
WHENEVER SQLERROR CONTINUE
END-EXEC.
MOVE SQLCODE TO CODE-EDIT.
STRING "*** ERROR Occurred while updating table. SQLCODE=
CODE-EDIT DELIMITED BY SIZE INTO PRINT-RECORD.
WRITE PRINT-RECORD.
14 EXEC SQL
ROLLBACK
END-EXEC.
STOP RUN.

E020-REPORT-ERROR.
MOVE SQLCODE TO CODE-EDIT.
STRING "*** ERROR Occurred while generating reports. SQLCODE
=" CODE-EDIT DELIMITED BY SIZE INTO PRINT-RECORD.
WRITE PRINT-RECORD.
STOP RUN.
Data Names Define Reference

ACTNO 168 SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
A100-DONE1 **** LABEL
A200-DONE2 **** LABEL

BIRTHDATE 134 DATE(10) COLUMN IN CORPDATA.EMPLOYEE
BONUS 134 DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE

CODE-EDIT 69

COMM 134 DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE
COMMISSION 43 DECIMAL(7,2) 136 170

CORPDATA **** SCHEMA

C1 165 CURSOR 174 182 250
C2 209 CURSOR 222 230 270

DEPTNO 50 CHARACTER(3) IN PROJECT
DEPTNO 213 CHARACTER(3) COLUMN (NOT NULL) IN CORPDATA.PROJECT

EDLEVEL 134 SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
EMENDATE 168 DATE(10) COLUMN IN CORPDATA.EMPLOYEE
EMENDATE **** COLUMN

EMPLOYEE **** TABLE IN CORPDATA

EMPLOYEE **** TABLE

169 216

EMPLOYEE-COUNT 63 SMALL INTEGER PRECISION(4,0) IN RPT2
EMPLOYEE-COUNT 114 IN RPT2-DATA

EMPNO 51 CHARACTER(6) IN RPT1 250

EMPNO 103 CHARACTER(6) IN RPT1-DATA
EMPNO 134 CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
EMPNO **** COLUMN IN EMPPROJECT

166 169 171 216

EMPNO **** COLUMN IN EMPLOYEE

EMPPROJECT 168 CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.EMPPROJECT
EMPPROJECT **** TABLE

166 169 210 215 216 218

EMPPROJECT **** TABLE IN CORPDATA

EMPPROJECT 168 213

EMPTIME 168 DECIMAL(5,2) COLUMN IN CORPDATA.EMPPROJECT
EMPTIME **** COLUMN

212

EMSTDATE 168 DATE(10) COLUMN IN CORPDATA.EMPPROJECT
EMSTDATE **** COLUMN

211

E010-UPDATE-ERROR **** LABEL

131

E020-REPORT-ERROR **** LABEL

148

FIRSTNAME 134 VARCHAR(12) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
FIRSTNAME **** COLUMN

167

HIREDATE 134 DATE(10) COLUMN IN CORPDATA.EMPLOYEE
JOB 134 CHARACTER(8) COLUMN IN CORPDATA.EMPLOYEE

LASTNAME 134 VARCHAR(15) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
LASTNAME **** COLUMN

167

MAJPROJ 50 CHARACTER(6) IN PROJECT
MAJPROJ 213 CHARACTER(6) COLUMN IN CORPDATA.PROJECT

MIDINIT 134 CHARACTER(1) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
NAME 52 CHARACTER(30) IN RPT1 251

NAME 105 CHARACTER(30) IN RPT1-DATA
PERCENTAGE 42 DECIMAL(5,2)
PHONENO 134 CHARACTER(4) COLUMN IN CORPDATA.EMPLOYEE
PRENDATE 50 DATE(10) IN PROJECT
PRENDATE *** COLUMN
PRSTAFF 50 DECIMAL(5,2) IN PROJECT
PRSTDATE 50 DATE(10) IN PROJECT
PROJECT 50 STRUCTURE IN RPT1
PROJECT *** TABLE IN CORPDATA
PROJECT *** TABLE
PROJECT-NAME 62 CHARACTER(36) IN RPT2
PROJECT-NAME 112 CHARACTER(36) IN RPT2-DATA
PROJNAME 50 VARCHAR(24) IN PROJECT
PROJNAME *** COLUMN
134 218
PROJNO 213 VARCHAR(24) COLUMN (NOT NULL) IN CORPDATA.PROJECT
PROJNO 50 CHARACTER(6) IN PROJECT
PROJNO 50 CHARACTER(6) IN PROJECT
250
PROJNO 61 CHARACTER(6) IN RPT2
PROJNO 101 CHARACTER(6) IN RPT1-DATA
PROJNO 110 CHARACTER(6) IN RPT2-DATA
PROJNO *** COLUMN
166 171
PROJNO 168 CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.EMPPROJECT
PROJNO *** COLUMN IN EMPPROJECT
210 215 218
PROJNO *** COLUMN IN PROJECT
215
PROJNO 213 CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.PROJECT
PRSTAFF 50 DECIMAL(5,2) IN PROJECT
PRSTAFF 213 DECIMAL(5,2) COLUMN IN CORPDATA.PROJECT
PRSTDATE 50 DATE(10) IN PROJECT
PRSTDATE 213 DATE(10) COLUMN IN CORPDATA.PROJECT
RAISE-DATE 41 CHARACTER(11)
RESPEMP 50 CHARACTER(6) IN PROJECT
RESPEMP 213 CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.PROJECT
RPT1 49
RPT1-DATA 100
RPT1-HEADERS 75
RPT1-HEADER1 76 IN RPT1-HEADERS
RPT1-HEADER2 80 IN RPT1-HEADERS
RPT2 60 STRUCTURE
270
RPT2-DATA 109
SS REFERENCE
RPT2-HEADERS 85
RPT2-HEADER1 86 IN RPT2-HEADERS
RPT2-HEADER2 90 IN RPT2-HEADERS
RPT2-HEADER3 95 IN RPT2-HEADERS
SALARY 53 DECIMAL(8,2) IN RPT1
251
SALARY 107 IN RPT1-DATA
SALARY *** COLUMN
135 135 167 212
SALARY 134 DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE
SEX 134 CHARACTER(1) COLUMN IN CORPDATA.EMPLOYEE
TOTAL-PROJ-COST 64 DECIMAL(12,2) IN RPT2
TOTAL-PROJ-COST 116 IN RPT2-DATA
WORK-DAYS 40 SMALL INTEGER PRECISION(4,0)
212
WORKDEPT 134 CHARACTER(3) COLUMN IN CORPDATA.EMPLOYEE

No errors found in source
307 Source records processed

* * * * * E N D   O F   L I S T I N G   * * * * *

Related concepts:
There are unique application and coding requirements for embedding SQL statements in a COBOL program. In this topic, requirements for host structures and host variables are defined.

**Example: SQL statements in PL/I programs**

This example program is written in the PL/I programming language.

**Note:** By using the code examples, you agree to the terms of the Code license and disclaimer information.
/* A sample program that updates the salaries for those employees */
/* whose current commission total is greater than or equal to the */
/* value of COMMISSION. The salaries of those who qualify are */
/* increased by the value of PERCENTAGE, retroactive to RAISE_DATE. */
/* A report is generated showing the projects that these employees */
/* have contributed to, ordered by project number and employee ID. */
/* A second report shows each project having an end date occurring */
/* after RAISE_DATE (that is, those projects potentially affected */
/* by the retroactive raises) with its total salary expenses and a */
/* count of employees who contributed to the project. */
*
*******************************************************************************/

PLIEX: PROC;

DCL RAISE_DATE CHAR(10);
DCL WORK_DAYS FIXED BIN(15);
DCL COMMISSION FIXED DECIMAL(8,2);
DCL PERCENTAGE FIXED DECIMAL(5,2);
DCL EMPNO CHAR(6),
NAME CHAR(30),
SALARY FIXED DECIMAL(8,2);

PROCEDURE RPT1;
1%INCLUDE PROJECT (PROJECT, RECORD,,COMMA);
15 EMPNO CHAR(6),
15 NAME CHAR(30),
15 SALARY FIXED DECIMAL(8,2);

PROCEDURE RPT2,
DCL 1 RPT2,
15 PROJNO CHAR(6),
15 PROJECT_NAME CHAR(36),
15 EMPLOYEE_COUNT FIXED BIN(15),
15 TOTL_PROJ_COST FIXED DECIMAL(10,2);

EXEC SQL INCLUDE SQLCA;

COMMISSION = 2000.00;
PERCENTAGE = 1.04;
RAISE_DATE = '1982-06-01';
WORK_DAYS = 253;
OPEN FILE(SYSPRINT);

EXEC SQL WHENEVER SQLERROR GO TO UPDATE_ERROR;
EXEC SQL WHENEVER SQLERROR GO TO REPORT_ERROR;

UPDATE CORPDATA/EMPLOYEE
SET SALARY = SALARY * :PERCENTAGE
WHERE COMM >= :COMMISSION ;
COMMIT;
EXEC SQL WHENEVER SQLERROR GO TO REPORT_ERROR;

*/
*/ File declaration for syssprint */
DCL SYSPRINT FILE EXTERNAL OUTPUT STREAM PRINT;

*/ Structure for report 1 */
DCL 1 RPT1,
1%INCLUDE PROJECT (PROJECT, RECORD,,COMMA);
15 EMPNO CHAR(6),
15 NAME CHAR(30),
15 SALARY FIXED DECIMAL(8,2);

*/ Structure for report 2 */
DCL 1 RPT2,
15 PROJNO CHAR(6),
15 PROJECT_NAME CHAR(36),
15 EMPLOYEE_COUNT FIXED BIN(15),
15 TOTL_PROJ_COST FIXED DECIMAL(10,2);

*/ Update the selected employees' salaries by the new percentage. */
*/ If an error occurs during the update, roll back the changes. */
EXEC SQL UPDATE CORPDATA/EMPLOYEE
SET SALARY = SALARY * :PERCENTAGE
WHERE COMM >= :COMMISSION ;
COMMIT;
EXEC SQL WHENEVER SQLERROR GO TO REPORT_ERROR;

*/ Commit changes */
EXEC SQL;

Page 2

Embedded SQL programming 167
DO EXEC SQL
DECLARE CI CURSOR FOR
SELECT DISTINCT projno, EMP PROJECT.empno, lastname[', ']fname, salary FROM CORPDATA/EMP.WRITEPROJECT, CORPDATA/EMPLOYEE WHERE EMPPROJECT.empno = EMPLOYEE.empno AND comm >= :COMMISSION ORDER BY projno, empno;
OPEN CI;
EXEC SQL
/* Fetch and write the rows to SYSPRINT */
DO UNTIL (SQLCODE != 0);
EXEC SQL
FIND CI INTO :RPT1.PROJNO, :RPT1.EMPNO, :RPT1.NAME,
(RPT1.SALARY); EXEC SQL
PUT FILE(SYSPRINT)
EDIT(RPT1.PROJNO, RPT1.EMPNO, RPT1.NAME, RPT1.SALARY)
(SKIP,COL(1),A,COL(10),A,COL(20),A,COL(54),F(8,2)); EXEC SQL
CLOSE CI;
EXEC SQL
/* For all projects ending at a date later than 'raise_date' */
/* (that is, those projects potentially affected by the salary raise), generate a report containing the project number, */
/* project name, the count of employees participating in the */
/* project, and the total salary cost of the project. */
EXEC SQL
/* Write out the header for Report 2 */
PUT FILE(SYSPRINT) EDIT('ACUMULATED STATISTICS BY PROJECT')
(SKIP(3),COL(22),A);
EXEC SQL
PUT FILE(SYSPRINT)
EDIT('PROJECT', 'NUMBER OF', 'TOTAL')
(SKIP(2),COL(1),A,COL(48),A,COL(63),A);
EXEC SQL
PUT FILE(SYSPRINT)
EDIT('NUMBER', 'PROJECT NAME', 'EMPLOYEES', 'COST')
(SKIP,COL(1),A,COL(10),A,COL(48),A,COL(63),A,SKIP);
EXEC SQL
DECLARE C2 CURSOR FOR
SELECT EMPPROJACT.PROJNO, PROJNAME, COUNT(*),
SUM(DAYS(EMENDATE) - DAYS(EMSTDATE)) * EMPTIME *
DECIMAL((SALARY / :WORK_DAYS),8,2) 
FROM CORPDATA/EMPPROJACT, CORPDATA/PROJECT, CORPDATA/EMPLOYEE
WHERE EMPPROJACT.PROJNO=PROJECT.PROJNO AND
EMPPROJACT.EMPNO =EMPLOYEE.EMPNO AND
PRENDATE > :RAISE_DATE
GROUP BY EMPPROJACT.PROJNO, PROJNAME
ORDER BY 1;
EXEC SQL
OPEN C2;
EXEC SQL WHENEVER NOT FOUND GO TO DONE2;
DO UNTIL (SQLCODE ^= 0);
EXEC SQL
FETCH C2 INTO :RPT2;
PUT FILE(SYSPRINT) EDIT(RPT2.PROJNO,RPT2.PROJECT_NAME,EMPLOYEE_COUNT,
TOTL_PROJ_COST) (SKIP,COL(1),A,COL(10),A,COL(50),F(4),COL(62),F(8,2));
END;
DONE2:
EXEC SQL
CLOSE C2;
GO TO FINISHED;
EXEC SQL WHENEVER SQLERROR CONTINUE;
PUT FILE(SYSPRINT) EDIT('*** ERROR Occurred while updating table. '||
'SQLCODE=',SQLCODE)(A,F(5));
EXEC SQL
ROLLBACK;
GO TO FINISHED;
EXEC SQL WHENEVER SQLERROR CONTINUE;
PUT FILE(SYSPRINT) EDIT('*** ERROR Occurred while generating reports. '||
'SQLCODE=',SQLCODE)(A,F(5));
REPORT ERROR:
EXEC SQL
PUT FILE(SYSPRINT) EDIT('*** ERROR Occurred while generating reports. '||
'SQLCODE=',SQLCODE)(A,F(5));
GO TO FINISHED;
END PLEX;
END OF SOURCE
CROSS REFERENCE
Data Names Define Reference
ACTNO 74 SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN CORPDATA.EMPPROJACT
BIRTHDATE 74 DATE(10) COLUMN IN CORPDATA.EMPLOYEE
BONUS 74 DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE
COMM **** COLUMN 52 76
COMM 74 DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE
COMMISSION 18 DECIMAL(8,2) 52 76
CORPDATA **** SCHEMA 50 74 118 118 118
C1 71 CURSOR 79 86 95
C2 114 CURSOR 125 132 141
DEPTNO 26 CHARACTER(3) IN RPT1
DEPTNO 118 CHARACTER(3) COLUMN (NOT NULL) IN CORPDATA.PROJECT
DONE1 **** LABEL 82
DONE2 **** LABEL 128
EDLEVEL 74 SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
EMENDATE 74 DATE(10) COLUMN IN CORPDATA.EMPPROJACT
EMENDATE **** COLUMN 116
EMPLOYEE **** TABLE IN CORPDATA 50 74 118
EMPLOYEE **** TABLE 75 120
EMPLOYEE_COUNT 35 SMALL INTEGER PRECISION(4,0) IN RPT2
EMPNO 27 CHARACTER(6) IN RPT1 86
EMPNO **** COLUMN IN EMPPROJACT 72 75 77 120
EMPNO **** COLUMN IN EMPLOYEE 75 120
EMPNO 74 CHARACER(6) COLUMN (NOT NULL) IN CORPDATA.EMPPROJACT
EMPNO 74 CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
EMPPROJACT **** TABLE 72 75 115 119 120 122
EMPPROJACT **** TABLE IN CORPDATA 74 118
EMPTIME 74 DECIMAL(5,2) COLUMN IN CORPDATA.EMPPROJACT
EMPTIME **** COLUMN 116
EMSTDATE 74 DATE(10) COLUMN IN CORPDATA.EMPPROJACT
EMSTDATE **** COLUMN 116
FIRSTNME **** COLUMN 73
FIRSTNME 74 VARCHAR(12) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
HIREDATE 74 DATE(10) COLUMN IN CORPDATA.EMPLOYEE
JOB 74 CHARACTER(8) COLUMN IN CORPDATA.EMPLOYEE
LASTNAME **** COLUMN 73
LASTNAME 74 VARCHAR(15) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
MAJPROJ 26 CHARACTER(6) IN RPT1
MAJPROJ 118 CHARACTER(6) COLUMN IN CORPDATA.PROJECT
MIDINIT 74 CHARACTER(1) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE
NAME 28 CHARACTER(30) IN RPT1 86
PERCENTAGE 19 DECIMAL(5,2) 51
PHONENO 74 CHARACTER(4) COLUMN IN CORPDATA.EMPLOYEE
There are some unique application and coding requirements for embedding SQL statements in a PL/I program. In this topic, requirements for host structures and host variables are defined.

**Example: SQL statements in RPG/400 programs**

This example program is written in the RPG programming language.

**Note:** By using the code examples, you agree to the terms of the "Code license and disclaimer information" on page 190.
Figure 6. Sample RPG/400 program using SQL statements
Record 1

1  IRPT1 E DSPROJECT 800
   I  PROJNAME PROJNM 900
   I  RESPEMP RESEM 1000
   I  PRSTAFF STAFF 1100
   I  PRSTDATE PRSTD 1200
   I  PRENDATE PREND 1300
   I  MAJPROJ MAJPRJ 1400
   I  
   I  DS 1600
   I  EMPNO 1 6 1700
   I  NAME 7 36 1800
   I  SALARY P 37 4121900
   I  
   I  IRPT2 DS 2100
   I  PRJNUM 1 6 2400
   I  PNAME 7 43 2500
   I  EMPCNT B 43 440 2600
   I  PRCOST P 45 492 2700
   I  
   I  DS 2800
   I  WRKDAY 1 20 2900
   I  RDATE 7 16 3000
   I  COMMI P 3 62 3100
   I  RDATE P 17 202 3200
   I  
   I  Z-ADD253 WRKDAY 3400
   I  Z-ADD2000.00 COMM 3600
   I  Z-ADD1.04 PERCNT 3700
   I  MOVE '1982-06-' RDATE 3800
   I  MOVE '01' RDATE 3900
   I  LR 4001
   I  
   I  Z-ADD253 WRKDAY 4000
   I  Z-ADD2000.00 COMM 4200
   I  Z-ADD1.04 PERCNT 4300
   I  MOVE '1982-06-' RDATE 4400
   I  MOVE '01' RDATE 4500
   I  LR 4600
   I  
   I  Z-ADD253 WRKDAY 4700
   I  Z-ADD2000.00 COMM 4800
   I  Z-ADD1.04 PERCNT 4900
   I  WHERE COMM >= :COMMI 5000
   I  
   I  SETON LR 5100
   I  
   I  Z-ADD253 WRKDAY 5200
   I  Z-ADD2000.00 COMM 5300
   I  Z-ADD1.04 PERCNT 5400
   I  WHERE COMM >= :COMMI 5500
   I  
   I  SETON LR 5600
   I  
   I  Z-ADD253 WRKDAY 5700
   I  Z-ADD2000.00 COMM 5800
   I  Z-ADD1.04 PERCNT 5900
C* Report the updated statistics for each employee assigned to
C* selected projects.
C* Write out the header for report 1.
C*
C* EXCPRECA
6 C/EXEC SQL DECLARE C1 CURSOR FOR
6 C+ SELECT DISTINCT PROJNO, EMPPROJACT.EMPNO,
6 C+ LASTNAME||'','||FIRSTNAME, SALARY
6 C+ FROM CORPDATA/EMPPROJACT, CORPDATA/EMPLOYEE
6 C+ WHERE EMPPROJACT.EMPNO = EMPLOYEE.EMPNO AND
6 C+ COMM >= :COMM
6 C+ ORDER BY PROJNO, EMPNO
C/END-EXEC
76 7 C/EXEC SQL
7 C+ OPEN C1
7 C/END-EXEC
80 C*
81 C+ Fetch and write the rows to QPRINT.
C+ END
96 C* For all project ending at a date later than the raise date
96 C* (that is, those projects potentially affected by the salary raises),
98 C+ generate a report containing the project number, project name, 
99 C+ the count of employees participating in the project, and the 
100 C+ total salary cost of the project.
101 C*
102 C* Write out the header for report 2.
103 C*
104 C* EXCPRECC
11 C/EXEC SQL
116 C+ DECLARE C2 CURSOR FOR
117 C+ SELECT EMPPROJACT.PROJNO, PROJNAME, COUNT(*),
118 C+ SUM(DAYS(EMSTDATE) - DAYS(EMSTDATE)) * EMPTIME *
119 C+ DECIMAL(SALARY/EMWEEK), 8,2)
120 C+ FROM CORPDATA/EMPPROJACT, CORPDATA/PROJECT, CORPDATA/EMPLOYEE
121 C+ WHERE EMPPROJACT.PROJNO = PROJECT.PROJNO AND
122 C+ EMPPROJACT.EMPNO = EMPLOYEE.EMPNO AND
123 C+ PRENDATE > :RDATE
124 C+ GROUP BY EMPPROJACT.PROJNO, PROJNAME
program.

* * * * * END OF SOURCE * * * * *
### CROSS REFERENCE

<table>
<thead>
<tr>
<th>Data Names</th>
<th>Define</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTNO</td>
<td>68</td>
<td>SMALL INTEGER PRECISION(0,0) COLUMN (NOT NULL) IN CORPDATA.EMPPROJECT</td>
</tr>
<tr>
<td>BIRTHDATE</td>
<td>48</td>
<td>DATE(10) COLUMN IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>BONUS</td>
<td>48</td>
<td>DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>COMM</td>
<td>48</td>
<td>DECIMAL(9,2) COLUMN IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>COMM</td>
<td>31</td>
<td>DECIMAL(7,2)</td>
</tr>
<tr>
<td>CORPDATA</td>
<td>48</td>
<td>SCHEMA 48 68 68 105 105 105</td>
</tr>
<tr>
<td>C1</td>
<td>68</td>
<td>CURSOR 77 86 92</td>
</tr>
<tr>
<td>C2</td>
<td>105</td>
<td>CURSOR 118 126 132</td>
</tr>
<tr>
<td>DEPTNO</td>
<td>8</td>
<td>CHARACTER(3) IN RPT1</td>
</tr>
<tr>
<td>DEPTNO</td>
<td>105</td>
<td>CHARACTER(3) COLUMN (NOT NULL) IN CORPDATA.PROJECT</td>
</tr>
<tr>
<td>DONE1</td>
<td>91</td>
<td>LABEL 83</td>
</tr>
<tr>
<td>DONE2</td>
<td>131</td>
<td>LABEL 123</td>
</tr>
<tr>
<td>EDLEVEL</td>
<td>48</td>
<td>SMALL INTEGER PRECISION(0,0) IN RPT2</td>
</tr>
<tr>
<td>EMPLOYEE</td>
<td>68</td>
<td>TABLE IN CORPDATA 48 68 105</td>
</tr>
<tr>
<td>EMPLOYEE</td>
<td>68</td>
<td>TABLE 68 105</td>
</tr>
<tr>
<td>EMPNO</td>
<td>17</td>
<td>CHARACTER(6) 86</td>
</tr>
<tr>
<td>EMPNO</td>
<td>48</td>
<td>CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>EMPNO</td>
<td>68</td>
<td>CHARACTER(6) COLUMN IN EMPPROJECT</td>
</tr>
<tr>
<td>EMPNO</td>
<td>68</td>
<td>COLUMN IN EMPLOYEE 68 105</td>
</tr>
<tr>
<td>EMPPROJECT</td>
<td>68</td>
<td>CHARACTER(6) COLUMN (NOT NULL) IN CORPDATA.EMPPROJECT</td>
</tr>
<tr>
<td>EMPPROJECT</td>
<td>68</td>
<td>TABLE 68 105 105 105 105</td>
</tr>
<tr>
<td>EMPNO</td>
<td>68</td>
<td>CHARACTER(6) COLUMN IN EMPLOYEE</td>
</tr>
<tr>
<td>EMPTIME</td>
<td>68</td>
<td>DECIMAL(5,2) COLUMN IN CORPDATA.EMPPROJECT</td>
</tr>
<tr>
<td>EMPTIME</td>
<td>68</td>
<td>COLUMN 105</td>
</tr>
<tr>
<td>EMSTDATE</td>
<td>68</td>
<td>DATE(10) COLUMN IN CORPDATA.EMPPROJECT</td>
</tr>
<tr>
<td>EMSTDATE</td>
<td>68</td>
<td>COLUMN 105</td>
</tr>
<tr>
<td>FINISH</td>
<td>156</td>
<td>LABEL</td>
</tr>
<tr>
<td>FIRSTNME</td>
<td>48</td>
<td>VARCHAR(12) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>FIRSTNME</td>
<td>68</td>
<td>COLUMN</td>
</tr>
<tr>
<td>HIREDATE</td>
<td>48</td>
<td>DATE(10) COLUMN IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>JOB</td>
<td>48</td>
<td>CHARACTER(8) COLUMN IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>LASTNAME</td>
<td>48</td>
<td>VARCHAR(15) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>LASTNAME</td>
<td>68</td>
<td>COLUMN</td>
</tr>
<tr>
<td>MAJPRJ</td>
<td>8</td>
<td>CHARACTER(6) IN RPT1</td>
</tr>
<tr>
<td>MAJPRJ</td>
<td>105</td>
<td>CHARACTER(6) COLUMN IN CORPDATA.PROJECT</td>
</tr>
<tr>
<td>MIDINIT</td>
<td>48</td>
<td>CHARACTER(1) COLUMN (NOT NULL) IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>NAME</td>
<td>18</td>
<td>CHARACTER(30) 86</td>
</tr>
<tr>
<td>PERCNT</td>
<td>33</td>
<td>DECIMAL(7,2) 86</td>
</tr>
<tr>
<td>PHONENO</td>
<td>48</td>
<td>CHARACTER(4) COLUMN IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>PNAME</td>
<td>25</td>
<td>CHARACTER(36) IN RPT2</td>
</tr>
<tr>
<td>PROJ</td>
<td>27</td>
<td>DECIMAL(9,2) IN RPT2</td>
</tr>
<tr>
<td>PREND</td>
<td>8</td>
<td>DATE(10) IN RPT1</td>
</tr>
<tr>
<td>PRENDATE</td>
<td>105</td>
<td>COLUMN</td>
</tr>
</tbody>
</table>
The RPG/400 licensed program supports both RPG II and RPG III programs.

Example: SQL statements in ILE RPG programs

This example program is written in the ILE RPG programming language.

Note: By using the code examples, you agree to the terms of the Code license and disclaimer information on page 190.
Source type...............RPG
Object name................CORPDATA/RPGLEEX
Source file...............CORPDATA/SRC
Member.....................OBJ
To source file..............QTEMP/QSQLTEMP1
Options....................XREF
RPG preprocessor options..NONE
Listing option..............PRINT
Target release..............VxRxMx
INCLUDE files..............SRCFILE
Commit........................CHG
Allow copy of data..........OPTIMIZE
Close SQL cursor............ENDMOD
Allow blocking...............ALLREAD
Delay PREPARE..............NO
Concurrency access resolution.............DFT
Generation level............10
Printer file...............*LBL/*QSYSPRT
Date format..................JOB
Date separator..............JOB
Time format..................HMS
Time separator...............JOB
Replace......................YES
Relational database...........LOCAL
User..........................CURRENT
RDB connect method..........DUW
Default collection..........NONE
Dynamic default collection...NONE
Package name................OBJLIB/*OBJ
Path........................NAMING
SQL rules....................DB2
Created object type.........PGM
Debugging view..............NONE
Debugging encryption key...NONE
User profile...............NAMING
Dynamic user profile.......USER
Sort sequence...............JOB
Language ID...............JOB
IBM SQL flagging............NOFLAG
ANS flagging.................NONE
Text........................SRCMBRTXT
Source file CCSID........37
Job CCSID..................37
Decimal result options:
  Maximum precision.........31
  Maximum scale...........31
  Minimum divide scale.....0
DECIMAL rounding mode......HALFEVEN
Compiler options............NONE

Figure 7. Sample ILE RPG program using SQL statements
1 CTL-OPT; 000100
2 // File declaration for QPRINT 000200
3 // 000300
4 DCL-F QPRINT(PRINTER(132)) USAGE(*OUTPUT); 000400
5 // 000500
6 // Structure for report 1. 000600
7 // 000700
8 1 DCL-DS RPT1 EXT EXTNAME('CORPDATA/PROJECT'); 000800
9 END-DS; 000900
10 // 001000
11 DCL-DS *N; 001100
12 EMPNO CHAR(6); 001200
13 NAME CHAR(30); 001300
14 SALARY PACKED(9:2); 001400
15 END-DS; 001500
16 // 001600
17 // Structure for report 2. 001700
18 // 001800
19 DCL-DS RPT2; 001900
20 PRJNUM CHAR(6); 002000
21 PNAME CHAR(36); 002100
22 EMPCT LONG DECIMAL(4:0); 002200
23 PCOST PACKED(9:2); 002300
24 END-DS; 002400
25 // 002500
26 DCL-DS *N; 002600
27 WRKDAY BINDEC(4:0); 002700
28 COMMI PACKED(7:2); 002800
29 RDATE CHAR(10); 002900
30 PERCNT PACKED(7:2); 003000
31 END-DS; 003100
32 // 003200
33 2 WRKDAY = 253; 003300
34 COMMI = 2000.00; 003400
35 PERCNT = 1.04; 003500
36 RDATE = '1982-06-01'; 003600
37 // 003700
38 // Update the selected projects by the new percentage. If an 003800
39 // error occurs during the update, roll back the changes. 003900
40 // 004000
41 3 EXEC SQL WHENEVER SQLERROR GOTO UPDERR; 004100
42 // 004200
43 EXEC SQL 004300
44 4 UPDATE CORPDATA/EMPLOYEE 004400
45 SET SALARY = SALARY * :PERCNT 004500
46 WHERE COMM >= :COMMI; 004600
47 // 004700
48 // Commit changes. 004800
49 // 004900
50 5 EXEC SQL COMMIT; 005000
51 // 005100
52 EXEC SQL WHENEVER SQLERROR GO TO RPTERR; 005200
53 // 005300
54 // Report the updated statistics for each employee assigned to 005400
55 // selected projects. 005500
56 // 005600
57 // Write out the header for report 1. 005700
58 // 005800
59 EXCEPTION RECA; 005900
60 6 EXEC SQL DECLARE C1 CURSOR FOR 006000
61 SELECT DISTINCT PROJNO, EMPPROJACT.EMPNO, 006100
62 LASTNAME || ', ' || FIRSTNAME, SALARY 006200
63 FROM CORPDATA/EMPLOYEE, CORPDATA/EMPLOYEE 006300
64 WHERE EMPPROJACT.EMPNO = EMPLOYEE.EMPNO AND 006400
65 COMM >= :COMMI 006500
66 ORDER BY PROJNO, EMPNO; 006600
67 // 006700
68 7 EXEC SQL OPEN C1; 006800
180 IBM i: Embedded SQL programming

```
EXEC SQL WHENEVER NOT FOUND GO TO DONE1;
DOU SQLCOD <> 0;
EXEC SQL FETCH C1 INTO :PROJNO, :EMPNO, :NAME, :SALARY;
EXCEPT RECC;
ENDDO;

C DONE1 TAG

EXEC SQL CLOSE C1;

// For all project ending at a date later than the raise date
// (that is, those projects potentially affected by the salary raises).
// generate a report containing the project number, project name,
// the count of employees participating in the project, and the
// total salary cost of the project.
// Write out the header for report 2.
//
// order by 1;

DECLARE C2 CURSOR FOR
SELECT EMPPROJECT.PROJNO, PROJNAME, COUNT(*),
    SUM(DAYS(EMENDATE) - DAYS(EMSTDATE)) * EMPTIME *
    DECIMAL((SALARY/WRKDAY),8,2)) FROM CORPDATA/EMPPROJECT, CORPDATA/PROJECT, CORPDATA/EMPLOYEE
WHERE EMPPROJECT.PROJNO = PROJECT.PROJNO AND
    EMPPROJECT.EMPNO = EMPLOYEE.EMPNO AND
    PREDATE > :DATE;

GROUP BY EMPPROJECT.PROJNO, PROJNAME
ORDER BY 1;

EXEC SQL OPEN C2;

// Fetch and write the rows to QPRINT.
//
EXEC SQL WHENEVER NOT FOUND GO TO DONE2;
DOU SQLCOD <> 0;
EXEC SQL FETCH C2 INTO :RPT2;
EXCEPT RECC;
ENDDO;

C DONE2 TAG

EXEC SQL CLOSE C2;

GOTO FINISH

// Error occurred while updating table. Inform user and rollback
// changes.

C UPDERR TAG

EXEC SQL WHENEVER SQLERROR CONTINUE;

EXEC SQL ROLLBACK;

GOTO FINISH

// Error occurred while generating reports. Inform user and exit.

C RPTERR TAG

EXCEPT RECC;

// All done.

C FINISH TAG

*INLR = 'ON;

QPRINT E RECA 0 2 01 'REPORT OF PROJECTS AFFECTED BY RAISES'

0 58 'TOTAL BY RAISES' 013400
0 0 0 1 'PROJECT'
0 14 'EMPID' 013800
0 32 'EMPLOYEE NAME'
0 59 'SALARY'
```

Create SQL ILE RPG Object

**Embedded SQL programming**
### Data Names Define Reference

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTNO</td>
<td>SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL)</td>
<td>IN CORPDATA.EMPPROJACT</td>
</tr>
<tr>
<td>BIRTHDATE</td>
<td>DATE(10)</td>
<td>COLUMN IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>BONUS</td>
<td>DECIMAL(9,2) COLUMN</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>COMM</td>
<td>COLUMNS</td>
<td>46 65</td>
</tr>
<tr>
<td>CORPDATA</td>
<td>SCHEMA</td>
<td>44 63 63 94 94 94 94</td>
</tr>
<tr>
<td>C1</td>
<td>CURSOR</td>
<td>60 74 78</td>
</tr>
<tr>
<td>C2</td>
<td>CURSOR</td>
<td>101 107 111</td>
</tr>
<tr>
<td>DEPTNO</td>
<td>CHARACTER(3)</td>
<td>IN RPT1</td>
</tr>
<tr>
<td>EMPCNT</td>
<td>SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL)</td>
<td>IN CORPDATA.PROJECT</td>
</tr>
<tr>
<td>DONE1</td>
<td>LABEL</td>
<td>77 72</td>
</tr>
<tr>
<td>DONE2</td>
<td>LABEL</td>
<td>110 105</td>
</tr>
<tr>
<td>EDLEVEL</td>
<td>SMALL INTEGER PRECISION(4,0) COLUMN (NOT NULL)</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>EMENDATE</td>
<td>DATE(10)</td>
<td>COLUMN IN CORPDATA.EMPPROJACT</td>
</tr>
<tr>
<td>EMPPROJACT</td>
<td>TABLE</td>
<td>44 63 94</td>
</tr>
<tr>
<td>EMPNO</td>
<td>CHARACTER(6)</td>
<td>COLUMN IN EMPPROJACT</td>
</tr>
<tr>
<td>EMPNO</td>
<td>COLUMN IN EMPLOYEE</td>
<td>64 66 96</td>
</tr>
<tr>
<td>EMPNO</td>
<td>COLUMN IN EMPLOYEE</td>
<td>64 96</td>
</tr>
<tr>
<td>EMPNO</td>
<td>CHARACTER(6) CCSID 37 COLUMN (NOT NULL)</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>EMPNO</td>
<td>CHARACTER(6) CCSID 37 COLUMN (NOT NULL)</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>EMPNO</td>
<td>TABLE IN CORPDATA</td>
<td>63 94</td>
</tr>
<tr>
<td>EMPNO</td>
<td>TABLE</td>
<td>64 95 96 98</td>
</tr>
<tr>
<td>EMPTIME</td>
<td>DECIMAL(5,2) COLUMN</td>
<td>IN CORPDATA.EMPPROJACT</td>
</tr>
<tr>
<td>EMSTDATE</td>
<td>DATE(10)</td>
<td>COLUMN IN CORPDATA.EMPPROJACT</td>
</tr>
<tr>
<td>FINISH</td>
<td>LABEL</td>
<td>131 74</td>
</tr>
<tr>
<td>FIRSTNME</td>
<td>COLUMN</td>
<td>62 62</td>
</tr>
<tr>
<td>FIRSTNME</td>
<td>VARCHAR(12) CCSID 37 COLUMN (NOT NULL)</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>HIREDATE</td>
<td>DATE(10)</td>
<td>COLUMN IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>JOB</td>
<td>CHARACTER(8) CCSID 37 COLUMN</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>LASTNAME</td>
<td>COLUMN</td>
<td>62 62</td>
</tr>
<tr>
<td>LASTNAME</td>
<td>VARCHAR(15) CCSID 37 COLUMN (NOT NULL)</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>MAJPROJ</td>
<td>CHARACTER(6)</td>
<td>IN RPT1</td>
</tr>
<tr>
<td>MAJPROJ</td>
<td>CHARACTER(6) CCSID 37 COLUMN IN CORPDATA.PROJECT</td>
<td>COLUMN</td>
</tr>
<tr>
<td>MIDINIT</td>
<td>CHARACTER(1) CCSID 37 COLUMN (NOT NULL)</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>NAME</td>
<td>CHARACTER(30)</td>
<td>74 74</td>
</tr>
<tr>
<td>PERCNT</td>
<td>DECIMAL(7,2)</td>
<td>45 45</td>
</tr>
<tr>
<td>PHONENO</td>
<td>CHARACTER(4) CCSID 37 COLUMN</td>
<td>IN CORPDATA.EMPLOYEE</td>
</tr>
<tr>
<td>PNAME</td>
<td>CHARACTER(36)</td>
<td>IN RPT2</td>
</tr>
<tr>
<td>PROJNUM</td>
<td>CHARACTER(6)</td>
<td>IN RPT2</td>
</tr>
<tr>
<td>PREDATE</td>
<td>DATE(8)</td>
<td>COLUMN IN RPT1</td>
</tr>
<tr>
<td>PRENDATE</td>
<td>DATE(10)</td>
<td>COLUMN IN CORPDATA.PROJECT</td>
</tr>
<tr>
<td>PRJNUM</td>
<td>CHARACTER(6)</td>
<td>97 97</td>
</tr>
</tbody>
</table>

**IBM i: Embedded SQL programming**
You need to be aware of the unique application and coding requirements for embedding SQL statements in an ILE RPG program. In this topic, the coding requirements for host variables are defined.

**Example: SQL statements in REXX programs**

This example program is written in the REXX programming language.

**Note:** By using the code examples, you agree to the terms of the “Code license and disclaimer information” on page 190.
/* A sample program which updates the salaries for those employees */
/* whose current commission total is greater than or equal to the */
/* value of COMMISSION. The salaries of those who qualify are */
/* increased by the value of PERCENTAGE, retroactive to RAISE_DATE. */
/* A report is generated and dumped to the display which shows the */
/* projects which these employees have contributed to, ordered by */
/* project number and employee ID. A second report shows each */
/* project having an end date occurring after RAISE_DATE (i.e. is */
/* potentially affected by the retroactive raises) with its total */
/* salary expenses and a count of employees who contributed to the */
/* project. */

RC = 0
COMMISSION = 2000.00;
PERCENTAGE = 1.04;
RAISE_DATE = '1982-06-01';
WORK_DAYS = 253;

ADDRESS '*COMMAND',
'DLT FILE(CORPDATA/REPORTFILE)' ADDRESS '*COMMAND',
'CRTPF FILE(CORPDATA/REPORTFILE) RCDLEN(80)' ADDRESS '*COMMAND',
'OVRDBF FILE(STDOUT) TOFILE(CORPDATA/REPORTFILE) MBR(REPORTFILE)'

UPDATE_STMT = 'UPDATE CORPDATA/EMPLOYEE ',
'SET SALARY = SALARY * ? ',
'WHERE COMM >= ? ',
EXECSQL,
'PREPARE S1 FROM :UPDATE_STMT'
EXECSQL,
'EXECUTE S1 USING :PERCENTAGE,' EXECSQL,
'COMMIT'
ERRLOC = 'REPORT_ERROR'

Figure 8. Sample REXX Procedure Using SQL Statements
Record *...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+... 8
52 /* Report the updated statistics for each project supported by one */
53 /* of the selected employees. */
54 55 /* Write out the header for Report 1 */
56 57 SAY ' ' 58 SAY ' ' 59 SAY ' REPORT OF PROJECTS AFFECTED BY EMPLOYEE RAISES' 60 SAY ' ' 61 SAY ' PROJECT EMPID EMPLOYEE NAME SALARY' 62 SAY '------- ------ -------------- ------' 63 64 SELECT_STMT = 'SELECT DISTINCT PROJNO, EMPPROJACT.EMPNO,' 65 ' 66 LASTNAME||', '',''|FIRSTNAME, SALARY,' 67 ' FROM CORPDATA/EMPPROJACT, CORPDATA/EMPLOYEE,' 68 ' WHERE EMPPROJACT.EMPNO = EMPLOYEE.EMPNO AND ' 69 ' 70 ' COMM >= ? ' 71 ' ORDER BY PROJNO, EMPNO ' 72 EXECSQL, 73 'PREPARE S2 FROM :SELECT_STMT' 74 75 DECLARE C1 CURSOR FOR S2 76 OPEN C1 USING :COMMISSION' 77 78 /* Handle the FETCH errors and warnings inline */
79 SIGNAL OFF ERROR 80 81 /* Fetch all of the rows */
82 DO UNTIL (SQLCODE <> 0) 83 9EXECSQL, 84 'FETCH C1 INTO :RPT1.PROJNO, :RPT1.EMPNO,',' 85 ' :RPT1.NAME, :RPT1.SALARY ' 86 87 /* Process any errors that may have occurred. Continue so that */
88 /* we close the cursor for any warnings. */
89 IF SQLCODE < 0 THEN 90 SIGNAL ERROR 91 92 /* Stop the loop when we hit the EOF. Don't try to print out the */
93 /* fetched values. */
94 IF SQLCODE = 100 THEN 95 LEAVE 96 97 /* Print out the fetched row */
98 SAY RPT1.PROJNO ' ' RPT1.EMPNO ' ' RPT1.NAME ' ' RPT1.SALARY 99 100 10EXECSQL, 101 'CLOSE C1' 102 103 /* For all projects ending at a date later than 'raise_date' */
104 /* (that is, those projects potentially affected by the salary raises) */
105 /* generate a report containing the project number, project name, */
106 /* the count of employees participating in the project, and the */
107 /* total salary cost of the project. */
/* Write out the header for Report 2 */
SAY 'ACCUMULATED STATISTICS BY PROJECT'
SAY 'PROJECT NAME NUMBER OF EMPLOYEES COST'
SAY '------- ----------- ------- ----'
/* Delete the OVRDBF so that we will continue writing to the output */
/* display. */
ADDRESS '+COMMAND',
'DLTOVR FILE(STDOUT)'

/* Leave procedure with a successful or warning RC */
EXIT RC

/* Error occurred while updating the table or generating the */
/* reports. If the error occurred on the UPDATE, rollback all of */
/* the changes. If it occurred on the report generation, display the */
/* REXX RC variable and the SQLCODE and exit the procedure. */
ERROR:
13SIGNAL OFF ERROR

/* Determine the error location */
SELECT /* When the error occurred on the UPDATE statement */
WHEN ERRLOC = 'UPDATE_ERROR' THEN
  DO
    SAY '*** ERROR Occurred while updating table.',
    'SQLCODE = ' SQLCODE
    14EXECSQL,
    'ROLLBACK'
  END
/* When the error occurred during the report generation */
WHEN ERRLOC = 'REPORT_ERROR' THEN
  SAY '*** ERROR Occurred while generating reports.',
  'SQLCODE = ' SQLCODE
  OTHERWISE
    SAY '*** Application procedure logic error occurred'
END

/* Delete the OVRDBF so that we will continue writing to the */
/* output display. */
ADDRESS '+COMMAND',
'DLTOVR FILE(STDOUT)'

/* Return the error RC received from SQL. */
EXIT RC

* * * * * E N D O F S O U R C E * * * * *

Related concepts:
"Coding SQL statements in REXX applications" on page 127

REXX procedures do not have to be preprocessed. At run time, the REXX interpreter passes statements that it does not understand to the current active command environment for processing.

Report produced by example programs that use SQL

This report is produced by each of the example programs.

REPORT OF PROJECTS AFFECTED BY RAISES

<table>
<thead>
<tr>
<th>PROJECT</th>
<th>EMPID</th>
<th>EMPLOYEE NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD3100</td>
<td>000010</td>
<td>HAAS, CHRISTINE</td>
<td>54860.00</td>
</tr>
<tr>
<td>AD3110</td>
<td>000070</td>
<td>PULASKI, EVA</td>
<td>37616.80</td>
</tr>
<tr>
<td>AD3111</td>
<td>000240</td>
<td>MARINO, SALVATORE</td>
<td>29910.40</td>
</tr>
<tr>
<td>AD3113</td>
<td>000270</td>
<td>PEREZ, MARIA</td>
<td>28475.20</td>
</tr>
<tr>
<td>IF1000</td>
<td>000030</td>
<td>KHAN, SALLY</td>
<td>39780.00</td>
</tr>
<tr>
<td>IF1000</td>
<td>000140</td>
<td>NICHOLLS, HEATHER</td>
<td>29556.80</td>
</tr>
<tr>
<td>IF2000</td>
<td>000030</td>
<td>KHAN, SALLY</td>
<td>39780.00</td>
</tr>
<tr>
<td>IF2000</td>
<td>000140</td>
<td>NICHOLLS, HEATHER</td>
<td>29556.80</td>
</tr>
<tr>
<td>MA2100</td>
<td>000010</td>
<td>HAAS, CHRISTINE</td>
<td>54860.00</td>
</tr>
<tr>
<td>MA2100</td>
<td>000110</td>
<td>LUCCHESI, VICENZO</td>
<td>48360.00</td>
</tr>
<tr>
<td>MA2110</td>
<td>000010</td>
<td>HAAS, CHRISTINE</td>
<td>54860.00</td>
</tr>
<tr>
<td>MA2111</td>
<td>000200</td>
<td>BROWN, DAVID</td>
<td>28849.60</td>
</tr>
<tr>
<td>MA2111</td>
<td>000220</td>
<td>LUTZ, JENNIFER</td>
<td>31033.60</td>
</tr>
<tr>
<td>MA2112</td>
<td>000150</td>
<td>ADAMSON, BRUCE</td>
<td>26291.20</td>
</tr>
</tbody>
</table>
CL command descriptions for host language precompilers

The IBM DB2 Query Manager and SQL Development Kit for i licensed program provides commands for precompiling programs coded in these programming languages.

Related concepts:
“Non-ILE SQL precompiler commands” on page 143

The IBM DB2 Query Manager and SQL Development Kit for i licensed program includes non-ILE precompiler commands for the following host languages: CRTSQLCBL (for OPM COBOL), CRTSQLPLI (for PL/I PRPQ), and CRTSQLRPG (for RPG III, which is part of RPG/400).

Related reference:
“ILE SQL precompiler commands” on page 144

In the IBM DB2 Query Manager and SQL Development Kit for i licensed program, these ILE precompiler commands exist: CRTSQLCI, CRTSQLCPPI, CRTSQLCBLI, and CRTSQLRPGI.

Create SQL COBOL Program command

The Create SQL COBOL Program (CRTSQLCBL) command calls the SQL precompiler.

It precompiles COBOL source containing SQL statements, produces a temporary source member, and then optionally calls the COBOL compiler to compile the program.

Related reference:
Create SQL COBOL Program (CRTSQLCBL) command

188 IBM i: Embedded SQL programming
Create SQL ILE COBOL Object command

The Create SQL ILE COBOL Object (CRTSQLCBLI) command calls the SQL precompiler, which precompiles COBOL source containing SQL statements, produces a temporary source member, and then optionally calls the ILE COBOL compiler to create a module, a program, or a service program.

Related reference:
Create SQL ILE COBOL Object (CRTSQLCBLI) command

Create SQL ILE C Object command

The Create SQL ILE C Object (CRTSQLCLI) command calls the SQL precompiler, which precompiles C source containing SQL statements, produces a temporary source member, and then optionally calls the ILE C compiler to create a module, create a program, or create a service program.

Related reference:
Create SQL ILE C Object (CRTSQLCLI) command

Create SQL ILE C++ Object command

The Create SQL ILE C++ Object (CRTSQLCPPI) command calls the SQL precompiler, which precompiles C++ source containing SQL statements, produces a temporary source member, and then optionally calls the C++ compiler to create a module.

Related reference:
Create SQL C++ Object (CRTSQLCPPI) command

Create SQL PL/I Program command

The Create SQL PL/I Program (CRTSQLPLI) command calls a SQL precompiler, which precompiles PL/I source containing SQL statements, produces a temporary source member, and optionally calls the PL/I compiler to compile the program.

Related reference:
Create SQL PL/I Program (CRTSQLPLI) command

Create SQL RPG Program command

The Create SQL RPG Program (CRTSQLRPG) command calls the SQL precompiler, which precompiles the RPG source containing the SQL statements, produces a temporary source member, and then optionally calls the RPG compiler to compile the program.

Related reference:
Create SQL RPG Program (CRTSQLRPG) command

Create SQL ILE RPG Object command

The Create SQL ILE RPG Object (CRTSQLRPGI) command calls the SQL precompiler, which precompiles RPG source containing SQL statements, produces a temporary source member, and then optionally calls the ILE RPG compiler to create a module, create a program, or create a service program.

Related reference:
Create SQL ILE RPG Object (CRTSQLRPGI) command

Related information for Embedded SQL programming

Product manuals and other information center topic collections contain information that relates to the Embedded SQL programming topic collection. You can view or print any of the PDF files.
Manuals

- ILE RPG Programmer's Guide
- ILE RPG Reference
- ILE COBOL Programmer's Guide
- ILE COBOL Reference
- REXX/400 Programmer's Guide
- REXX/400 Reference
- DB2 for i SQL reference

The following manuals are not included in the IBM i Information Center. However, these manuals might be a useful reference to you. Each of the manuals is available from the IBM Publications Center as a printed hardcopy that you can order, in an online format that you can download at no charge, or both.

- COBOL/400 User's Guide (about 5,980 KB)
- COBOL/400 Reference (about 2,150 KB)
- RPG/400 User's Guide (about 2,090 KB)
- RPG/400 Reference (about 2,520 KB)

Other information

You can view or download these related topics:

- Database performance and query optimization
- SQL call level interface
- SQL messages and codes
- SQL programming

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.
Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.
Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided “AS IS”, without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.
Programming interface information

This Embedded SQL programming publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.