Embedded SQL

- After completing this chapter, you should be able to
  - ▷ work with **programming language (PL) interfaces** to an RDBMS, the basis for database **application development,**
  - ▷ develop (simple) programs that use **Embedded SQL,**

    Syntax of Embedded SQL, how to preprocess/compile **C programs containing embedded SQL statements**, usage of **host variables**, **error handling, indicator variables,** *etc.*
  - ▷ explain the use of **cursors** (and why they are needed to interface with a PL).

Embedded SQL

Overview

Introduction (1)

- SQL is a database language, but **not a programming language.**
  - ▷ Complex queries (and updates) may be expressed using rather short SQL commands.

    Writing equivalent code in C would take significantly more time.
  - ▷ SQL, however, is **not functionally complete.**

    Not every computable function on the database states is expressible in SQL. Otherwise, **termination of query evaluation** could not be guaranteed,

Introduction (2)

- SQL is used directly for **ad-hoc queries** or **one-time updates** of the data.
- **Repeating tasks** have to be supported by **application programs** written in some PL.

  Internally, these programs generate SQL commands which are then shipped to the DBMS.
- Most database user do *not* know SQL or are even unaware that they interact with a DBMS.
- Even if a user knows SQL, an application program might be more effective than the plain SQL console.

  Think of **visual representation** of query results or **sanity checks** during data entry.

Introduction (3)

- Languages/tools widely used for **database application programming:**
  - ▷ **SQL scripts,**

    Like UNIX shell scripting language but interpreted by non-interactive SQL console.
  - ▷ **C with Embedded SQL,**
  - ▷ **C with library procedure calls (ODBC),**
  - ▷ **Java with library procedure calls (JDBC),**
  - ▷ **Scripting languages** (Perl/DBI, PHP (LAMP), Python/DB-API, . . .),
  - ▷ **Web interfaces** (CGI, Java Servlets, . . .).

Introduction (4)

- Almost always, developers work with more than one language (*e.g.*, C and SQL) to develop an application.
  This leads to several problems:
  - ▷ The interface is not smooth: **type systems differ** and the infamous impedance mismatch problem.

    Impedance mismatch: SQL is declarative and set-oriented. Most PLs are imperative and record- (tuple-) oriented.
  - ▷ SQL commands are spread throughout the application code and can never be optimized as a whole database workload.
  - ▷ Query evaluation plans should be persistently kept inside the DBMS between program executions, but programs are external to the DBMS.

Introduction (5)

- Note that these problems could be avoided with **real database programming languages,** *i.e.*, a tight integration of DBMS and PL compiler and runtime environment.
  Proposed solutons:
  - ▷ **Persistent programming languages** (*e.g.*, Napier88, Tycoon, Pascal/R [Pascal with type `relation`]),
  - ▷ **stored procedures,**

    Application code stored inside DBMS, DBMS kernel has built-in language interpreter or calls upon external interpreter.
  - ▷ **object-oriented DBMS,**

    OODBMS stores methods (behaviour) along with data.
  - ▷ **deductive DBMS.**

    DBMS acts as huge fact storage for a Prolog-style PL.

Making Good Use of SQL

- Too often, application programs use a relational DBMS only to make records persistent, but perform all computation in the PL.

  Such programs typically retrieve single rows (records) one-by-one and perform joins and aggregations by themselves.

- Using more powerful SQL commands might
  - ▷ simplify the program, and
  - ▷ **significantly** improve the performance.

    There is a considerable overhead for executing an SQL statement: send to DBMS server, compile command, send result back. The fewer SQL statements sent, the better.

Example Database (recap)

STUDENTS			
SID	FIRST	LAST	EMAIL
101	Ann	Smith	...
102	Michael	Jones	(null)
103	Richard	Turner	...
104	Maria	Brown	...

EXERCISES			
CAT	ENO	TOPIC	MAXPT
H	1	Rel.Alg.	10
H	2	SQL	10
M	1	SQL	14

RESULTS			
SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

Embedded SQL

Overview

1. Introduction and Overview
2. Embedded SQL

Embedded SQL (1)

- **Embdedded SQL** inserts specially marked SQL statements into program source texts written in C, C++, Cobol, and other PLs.
- Inside SQL statements, **variables of the PL** may be used where SQL allows a constant term only (parameterized queries).

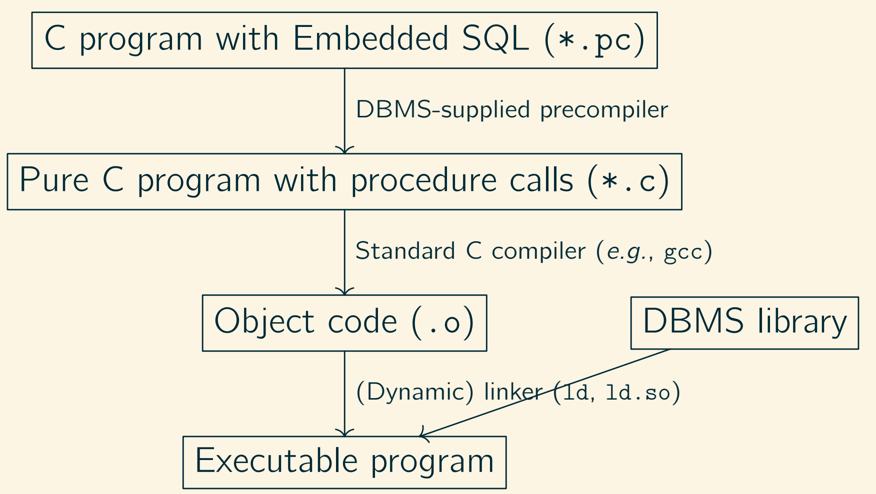
Insert a row into table `RESULTS`:

```
EXEC SQL INSERT INTO RESULTS(SID, CAT, ENO, POINTS)
         VALUES (:sid, :cat, :eno, :points);
```

▷ Here, `sid` *etc.* are C variables and the above may be emdbedded into any C source text.

Embedded SQL (2)

Compilation/linkage of Embedded SQL programs



A Mini C Recap (1)

- The **C** programming language was designed by Dennis Ritchie around 1972 at Bell Labs.

Traditional first C program.

```
#include <stdio.h>
int main (void)
{
    printf ("Hello, world!\n");  /* \n = newline */
    return 0;
}
```

Execution starts at mandatory procedure `main`. Return value `0` is a signal to the OS that the execution went OK (also see `exit()`). Header file `"stdio.h"` contains declaration of library function `printf` used for output. Braces (`{`, `}`) enclose nested statement blocks.

A Mini C Recap (2)

- In C, a **variable declaration** is written as

$$\langle \mathit{Type} \rangle \, \langle \mathit{Variable} \rangle \; ;$$

Declare integer variable `sid`:

```
int sid; /* student ID */
```

- There are integer types of different size, *e.g.*, `long` and `short`.

The type `short` (or `short int`) typically is 16 bits wide: $-32768 \ldots 32767$. Type `int` corresponds to the word size of the machine (today: 32 bits). Type `long` is at least 32 bits wide. Integer types may be modified with the `unsigned` prefix, *e.g.*, `unsigned short` has the range $0 \ldots 65535$.

A Mini C Recap (3)

- The type `char` is used to represent characters (today, effectively an 8 bit value).

  The type `unsigned char` is guaranteed to provide the value range 0...255.

Declaration of an array of characters `a[0]..a[19]`:

```
char a[20];
```

- In C, **strings** are represented as such character arrays. A null character (`'\0'`) is used to mark the string end.

  String `"xyz"` is represented as `a[0] = 'x'`, `a[1] = 'y'`, `a[2] = 'z'`, `a[3] = '\0'`.

A Mini C Recap (4)

- **Variable assignment:**

```
sid = 101;
```

- **Conditional statement:**

```
if (retcode == 0)    /* == is equality */
    printf ("Ok!\n");
else
    printf ("Error!"\n);
```

- C has no **Boolean** type but uses the type `int` instead to represent truth values: 0 represents *false*, anything is *true*.

A Mini C Recap (5)

- **Print an integer** (`printf`: print formatted):

```
printf ("The current student ID is: %d\n", sid);
```

  First argument is a format string that determines number and type of further arguments. Format specifiers like `%d` (print `int` in decimal notation) consume further elements in order.

- **Read an integer** (`%d`: in decimal notation):

```
ok = scanf ("%d", &sid);
```

  `&sid` denotes a pointer to variable `sid`. Since C knows *call by value* only, references are implemented in terms of pointers. Library function `scanf` returns the number of converted format elements (here `1` if no problems occur). Trailing newlines are not read.

A Mini C Recap (6)

- Suppose that variable `name` is declared as

```
char name[21];
```

- In C, variable assignment via = does *not* work for strings (arrays), instead use the library function `strncpy` (declared in header file `"string.h"`):

```
strncpy (name, "Smith", 21);
```

  The C philosophy is that = should correspond to a single machine instruction. In C, the programmer is responsible to avoid string/buffer overruns during copying. This is *the* source of nasty bugs and security holes. `strncpy` never copies more characters than specified in the last argument.

A Mini C Recap (7)

- To read an entire line of characters (user input) from the terminal, use

```
fgets (name, 21, stdin);
name[(strlen (name) - 1] = '\0';  /* overwrite '\n' */
```

  `fgets` reads no more than characters than one less than specified by the second argument. The trailing newline is stored and a `'\0'` is placed to mark the string end. `stdin` denotes the terminal (if not redirected). Library function `strlen` does the obvious.

Host Variables (1)

- If SQL is embedded in C, then C is the **host language.** C variables which are to be used in SQL statements are referred to as **host variables.**
- Note that SQL uses a **type system** which is quite different from the C type system.

  For example, C has no type `DATE` and no C type corresponds to `NUMERIC(30)`.

- In addition, C has no notion of null values.
- Even if there is a natural correspondence between an SQL type and a C type, the value **storage format** might be considerable different.

  Think of endianness, for example.

Host Variables (2)

- Oracle, for example, stores variable length strings (SQL type `VARCHAR(`*n*`)`) as length information followed by an array of characters. C uses `'\0'`-terminated `char` arrays.
- Oracle stores numbers with mantissa and exponent (scientific notation) with the mantissa represented in BCD (4 bits/digit). C uses a binary representation.
- **Type/storage format conversion** has to take place whenever data values are passed to/from the DBMS.
  - ▷ The **precompiler** can help quite a lot here, but some work remains for the programmer.

Host Variables (3)

- The DBMS maintains a translation table between *internal types* and *external types* (host language types) and possible conversions between these.
- In Embedded SQL, many conversion happen **automatically**, *e.g.*, `NUMERIC(`*p*`)`, $p < 10$, into the C type `int` (32 bits).

  Also, `NUMERIC(`*p*`,`*s*`)` may be mapped to `double`, although precision may be lost.
- For `VARCHAR(`*n*`)`, however, the program either prepares C a `struct` that corresponds to the DBMS storage format or explicitly states that a conversion to `'\0'`-terminated C strings is to be done.

Host Variables (4)

- The precompiler must be able to extract and understand the **declaration of the host variables.**
- Usually, the Embedded SQL precompiler does not fully "understand" the C syntax (with all its oddities).

Correct C declaration syntax?

```
unsigned short int          short int unsigned
unsigned int short          int unsigned short
short unsigned int          int short unsigned
```

- Thus, variable declarations relevant to the precompiler must be enclosed in `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION`.

Host Variables (5)

- The declaration section might look as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   int     sid;                 /* student ID */
   VARCHAR first[20];           /* student first name */
   char    last[21];            /* student last name */
   EXEC SQL VAR last IS STRING(21);
EXEC SQL END DECLARE SECTION;
```

▷ `sid` is a standard C integer variable, the DBMS will automatically convert to and from `NUMERIC(`p`)`.

▷ `last` is a standard C character array (string).

The conversion to/from this format is explicitly requested (note: due to `'\0'`-termination, max. string length is 20).

Host Variables (6)

- `VARCHAR first[20]` is *not* a standard C data type.
  - ▷ The precompiler translates this declaration into

```
struct { unsigned short len;
         unsigned char  arr[20];
} first;
```

which is a C type whose **memory layout** exactly matches the DBMS-internal `VARCHAR(20)` representation.

  - ▷ The conversion from a standard C char array `s` could be done as follows:

```
first.len = MIN (strlen (s), 20);
strncpy (first.arr, s, 20);
```


Host Variables (7)

- The variables in the `DECLARE SECTION` may be global as well as local.
- The types of these variables must be such that the precompiler can interpret them.

  Especially, non-standard user-defined types (`typedef`) are not allowed here.

- In SQL statements, host variables are prefixed with a colon (`:`) and may thus have the same name as table columns.

Error Checking (1)

- Similar coding guidelines apply whether the program interacts with the operating system or with the DBMS: **after every interaction check for possible error conditions.**
- One possibility to do this is to declare a special variable

```
char SQLSTATE[6];
```

- As required by the SQL-92 standard, if this variable is declared, the DBMS stores a **return code** whenever an SQL statement has been executed.

  `SQLSTATE` contains error class and subclass codes. First two characters `"00"` indicate "*okay*" and, for example, `"02"` indicates "*no more tuples to be returned*".

Error Checking (2)

- An alternative is the SQL **communication area** `sqlca` (a C `struct`) which can be declared via

```
EXEC SQL INCLUDE SQLCA;
```

  ▷ Component `sqlca.sqlcode` then contains the return code, for example, 0 for "*okay*", 1403: "*no more tuples*".

  ▷ Component `sqlca.sqlerrm.sqlerrmc` contains the error messag text, `sqlca.sqlerrm.sqlerrl` contains its length:

```
printf ("%.*s\n", sqlca.sqlerrm.sqlerrml,
                  sqlca.sqlerrm.sqlerrmc);
```

Error Checking (3)

- The precompiler supports the programmer in enforcing a consistent error checking discipline:

  `EXEC SQL WHENEVER SQLERROR GOTO` ⟨*Label*⟩`;`

  or

  `EXEC SQL WHENEVER SQLERROR DO` ⟨*Stmt*⟩`;`

  ▷ The C statement ⟨*Stmt*⟩ typically is a C procedure call to an error handling routine (any C statement is allowed).

- Such `WHENEVER SQLERROR` declarations may be cancelled via

  `EXEC SQL WHENEVER SQLERROR CONTINUE;`

Example (1)

```
/* program to enter a new exercise */

#include <stdio.h>
EXEC SQL INCLUDE SQLCA;    /* SQL communication area */
EXEC SQL BEGIN DECLARE SECTION;
  VARCHAR user[128];       /* DB user name */
  VARCHAR pw[32];          /* password */
  VARCHAR cat[1];
  int     eno;
  int     points;
  VARCHAR topic[42];
EXEC SQL END DECLARE SECTION;
...
```

Example (2)

```
...
/* called in case of (non-SQL) errors */
void fail (const char msg[])
{
    /* print error message */
    fprintf (stderr, "Error: %s\n", msg);

    /* close DB connection */
    EXEC SQL ROLLBACK WORK RELEASE;

    /* terminate */
    exit (1);
}
...
```


Example (3)

```
...
int main (void)
{
    char line[80];

    /* catch SQL errors */
    EXEC SQL WHENEVER SQLERROR GOTO error;

    /* log into DBMS */
    strncpy (user.arr, "grust", 128);
    user.len = strlen (user.arr);
    strncpy (pw.arr, "******", 32);
    pw.len = strlen (pw.arr);
    EXEC SQL CONNECT :user IDENTIFIED BY :pw;
...
```

Example (4)

```
...
/* read CAT, ENO of new exercise */
printf ("Enter data of new exercise:\n");
printf ("Category (H,M,F) and number (e.g., M6): ");
fgets (line, 80, stdin);
if (line[0] != 'H' && line[0] != 'M' &&
    line[0] != 'F')
    fail ("Invalid category");
cat.arr[0] = line[0];
cat.len    = 1;
if (sscanf (line + 1, "%d", &eno) != 1)
    fail ("Invalid number");
...
```


Example (5)

```
...
/* read TOPIC of new exercise */
printf ("Topic of the exercise: ");
fgets ((char *) topic.arr, 42, stdin);
topic.len = strlen (topic.arr) - 1;  /* remove '\n' */

/* read MAXPT for new exercise */
printf ("Maximum number of points: ");
fgets (line, 80, stdin);
if (sscanf (line, "%d", &points) != -1)
    fail ("Invalid number");
...
```

Example (6)

```
...
/* show read exercise data */
printf ("%c %d [%s]: %d points\n",
        cat.arr[0], eno, title.arr, maxpt);

/* execute SQL INSERT statement */
EXEC SQL INSERT INTO
         EXERCISES (CAT, ENO, TOPIC, MAXPT)
         VALUES (:cat, :eno, :topic, :points);

/* end transaction, log off */
EXEC SQL COMMIT WORK RELEASE;
...
```


Example (7)

```
...
/* terminate program (success) */
return 0;

/* jumped to in case of SQL errors */
error:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    fprintf (stderr, "DBMS Error: %.*s\n",
             sqlca.sqlerrm.sqlerrml,
             sqlca.sqlerrm.sqlerrc);
    EXEC SQL ROLLBACK WORK RELEASE;
    exit (EXIT_FAILURE);
...
```

Simple Queries (1)

- The above example shows how to pass values **from the program into the DBMS** (*e.g.*, for `INSERT`).
- Now the task is to extract values **from the database into host variables.**
- If is it guaranteed that a query can **return at most one tuple,** the following may be used:

`SELECT INTO`: read student tuple specified by `sid`.

```
EXEC SQL SELECT  FIRST, LAST
         INTO    :first, :last
         FROM    STUDENTS
         WHERE   SID = :sid
```


Simple Queries (2)

- It is an error if the `SELECT INTO` yields more than one row.

`SELECT INTO` using a "soft key".

```
EXEC SQL SELECT  SID
         INTO    :sid
         FROM    STUDENTS
         WHERE   FIRST = :first
         AND     LAST = :last
```

▷ The DBMS will execute the statement without warning as long as there is at most one `SID` returned. A result of two or more tuples will raise an SQL error.

Simple Queries (3)

- After issuing a SELECT statement, the program is expected to check whether a row was found at all. (An empty result is no error, but then the INTO host variables are **undefined.**)

①
```
if (sqlca.sqlcode == 0)
    ... process returned tuple data ...
```

②
```
EXEC SQL WHENEVER NOT FOUND GOTO empty;
EXEC SQL SELECT ... INTO ...;
    ... process returned tuple data ...
empty:
    ... no tuple returned ...
```


General Queries (1)

- In general, a SQL query will yield a table, *i.e.*, more than a single tuple. Since C lacks a type equivalent to the relational table concept, **the query result must be read tuple-by-tuple** in a loop.
  - ▷ A DBMS-maintained **cursor** points into the table, marking the next tuple to be read.

Declaring a SQL cursor:

```
EXEC SQL DECLARE c1 CURSOR FOR
         SELECT  CAT, ENO, POINTS
         FROM    RESULTS
         WHERE   SID = :sid
```

  - ▷ Note: at this point, the query is not yet executed and the value of :sid is immaterial.

General Queries (2)

- The next step is to **open the cursor:**

```
EXEC SQL OPEN c1;
```

  - ▷ This initiates **query evaluation** and the then current value of the query parameter `:sid` is used.
  - ▷ The program may **close** the cursor and **reopen** it again with a different value of `:sid`.

General Queries (3)

- The query result may then be read **one tuple at a time** into host variables

FETCH

```
EXEC SQL WHENEVER NOT FOUND GOTO done;
while (1) {   /* while (forever) */
    EXEC SQL FETCH c1 INTO :cat, :eno, :points;
    ... process result tuple data ...
}
done:
    ... all tuples processed ...
```

General Queries (4)

- Other variants:

①
```
EXEC SQL WHENEVER NOT FOUND DO break;
while (1) {   /* while (forever) */
    EXEC SQL FETCH c1 INTO :cat, :eno, :points;
    ... process result tuple data ...
}
... all tuples processed ...
```

②
```
EXEC SQL FETCH c1 INTO :cat, :eno, :points;
while (sqlca.sqlcode == 0) {
    ... process result tuple data ...
    EXEC SQL FETCH c1 INTO :cat, :eno, :points;
}
... all tuples processed ...
```


General Queries (5)

- The last step is to **close** the cursor:

```
EXEC SQL CLOSE c1;
```

  ▷ Open cursors allocate memory and, more importantly, **retain locks on the data** which can get in the way of other concurrent users.

Positioned Updates/Deletes

- A program can refer to tuple last FETCHed in UPDATE and DELETE commands:

```
EXEC SQL UPDATE RESULTS SET POINTS = :points
         WHERE CURRENT OF c1;
```

  ▷ This is helpful if the new attribute value (here: `points`) is computed by the C program (*e.g.*, read from the terminal) and not by an SQL query.

Null Values (1)

- If a column value in a query result can possibly yield NULL, the program is required to declare two host variables: one variable will receive the **data value** (if any), the other will **indicate whether the value is NULL.**

  ▷ Such variables are called **indicator variables** (normally of C type `short`).

  ▷ The indicator variable will be set ot `-1` if NULL was returned by the query (otherwise set to `0`).

Null Values (2)

Cursor declared to fetch student data:

```
EXEC SQL DECLARE stud CURSOR FOR
         SELECT  FIRST, LAST, EMAIL
         FROM    STUDENTS;
```

- An indicator variable may be attached to any variable in an SQL statment, *e.g.*:

```
EXEC SQL FETCH stud INTO :first, :last,
                         :email INDICATOR :null;
```

- It is an error to `FETCH` a `NULL` value without indicator variables set up (this includes the result of aggregation fuctions!).
- Indicator variables may also be used during `INSERT` to insert `NULL` column values into the DB.

Dynamic SQL (1)

- Up to here, table and column names were already known at program **compile time.** At runtime, the current value of host variables is inserted into these **static SQL statements.**
  - ▷ In the case of static SQL, the precompiler checks the existence of tables and columns (via lookups in the DBMS **data dictionary**).
  - ▷ In some systems (*e.g.*, IBM DB2), static queries are already **optimized** at compile time and the resulting query evaluation plans are stored in the database.

Dynamic SQL (2)

- In contrast, it is possible to compose strings containing SQL statements at **runtime** and then to ship the string to the DBMS for execution.

  This is exactly how the the **SQL console** application is built.

- If the SQL command is *not* a query (whose result needs to be consumed), dynamic execution works as follows:

```
EXEC SQL EXECUTE IMMEDIATE :sql_cmd;
```


Dynamic SQL (3)

- A problem of the dynamic SQL approach is that the command has to compiled (into a query evaluation plan) every time it is submitted to the DBMS. Query optimization may be costly.

  The DBMS may cache recent query evaluation plans. These may be reused if a query is re-issued (possibly with different host variable values).

- If an SQL statement is executed several times with different host variables values, the DBMS can be explicitly asked to **precompile** ("prepare") the query using `EXEC SQL PREPARE` and then calling

  `EXECUTE ... USING` ⟨*Variables*⟩`;`

Dynamic SQL (4)

- Note that, for dynamic queries, the **result schema** (tuple format) is not known until runtime.

  This rules out the use a construct like `SELECT INTO`.

- In this case, an SQL **descriptor area** (`SQLDA`) is used to obtain information about the result columns (column names, types)..

  ▷ The SQL `DESCRIBE` statement stores the number, names, and datatypes of the result columns of a dynamic query in the `SQLDA`.

  The `SQLDA` also contains slots for pointers to variables which will contain the retrieved data values (the `FETCH` host variables).

Dynamic SQL (5)

- The sequence of steps:

  ① Allocate an `SQLDA` (SQL-92: `ALLOCATE DESCRIPTOR`).

  ② Compose the query string.

  ③ Compile the query using `PREPARE`.

  ④ Use `OPEN` to execute the query and open a result cursor.

  ⑤ Fill the `SQLDA` using `DESCRIBE`.

  ⑥ Allocate variables for the query result (place pointers in `SQLDA`).

  ⑦ Call `FETCH` repeatedly to read the result tuples.