# Part 17: Application Programming I

Elmasri/Navathe:

- Section 10.5, "Programming Oracle Applications"

I have also used the following sources:

- R. Sunderraman: Oracle Programming — A Primer, Addison-Wesley, 1999.
  Chapter 3, "Embedded SQL", Chapter 5, "Oracle JDBC".
- Michael Gertz: Oracle/SQL Tutorial, 1999. `http://www.db.cs.ucdavis.edu/teaching/sqltutorial/`
- Oracle8 Application Developer's Guide, Oracle Corporation, 1997, Part No. A58241-01.
- Pro*C/C++ Precompiler Programmer's Guide, Release 8.0, Oracle Corporation, 1997, Part No. A58233-01.
- Kernighan/Ritchie: The C Programming Language, 2nd Edition, Prentice Hall, 1988.
- Don Chamberlin: A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.
- Microsoft ODBC 3.0, Programmers Reference and SDK Guide, Microsoft Press, 1997.
- Roger E. Sanders: ODBC 3.5 Developer's Guide. McGraw-Hill, 1999.
- SQL Server Books Online, "Building SQL Server Applications".
- Art Taylor: JDBC Developer's Resource, 2nd Edition. Prentice Hall, 1999.





## Objectives

- You should be able to name a few languages/interfaces/tools which can be used for application program development.
- You should know the steps to translate a C-program with Embedded SQL to an executable program.
- You should be able to explain how host variables can be used in embedded SQL statements.
- You should know the notion of a cursor and the steps involved in using a cursor.
- You should when to use indicator variables.
- You should have some general understanding of ODBC and JDBC.





## Overview

1. Introduction and Overview
2. Embedded SQL
3. ODBC
4. JDBC

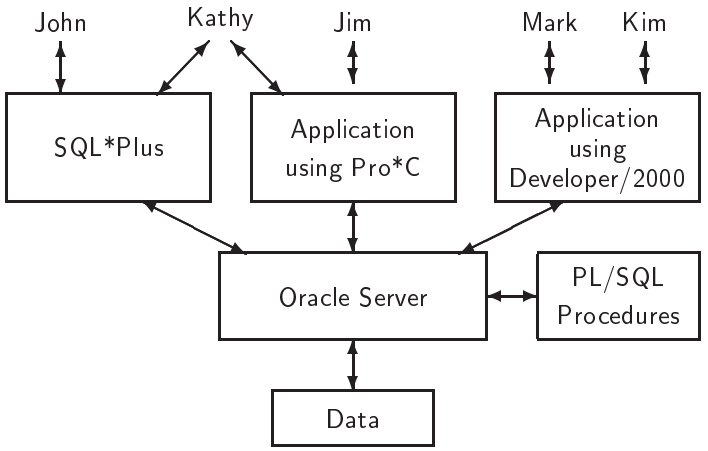




## Introduction (1)

- SQL is a DB language, but not a programming language.
- You can write powerful queries with simple SQL-commands.
  Writing something similar in C would take much more time.
- SQL is not a language for defining user interfaces or complex computations on the data.
- You use SQL directly for ad-hoc queries or one-time updates of the data.
- Repeating tasks have to be supported by application programs written in a programming language.
  These programs do use SQL internally for data access. So if you use application programs, you still use SQL indirectly.

## Introduction (2)








## Introduction (3)

Some Languages/Tools for Application Programming:

- SQL*Plus Scripts
- C with Embedded SQL (Pro*C)
- C with procedure calls (OCI, ODBC)
- Java with procedure calls (JDBC)
- Form-based applications developed with Developer/2000 4GL (Fourth Generation Language): GUI + DB + Rules(?)
- Web Interface: CGI-Program, PL/SQL-Procedures, Java Program, HTML with embedded procedural code, ...





## Introduction (4)

- So often you have to work with more than one language (e.g. C and SQL) to develop the application.
- The interface is often not very smooth (e.g. different type systems) and there is the "impedance mismatch problem".
  SQL is a declarative, set-oriented language. Most programming languages are imperative, tuple-oriented.
- The DB optimizer can look only at a single SQL commands at a time, it does not know what happens in between.
- Query evaluation plans for the SQL statements in the program should be kept between program executions, but programs are external to the DB.





## Introduction (5)

- Therefore, database research aims at integrated systems consisting of a programming language and a database.
- Proposed solutions are, e.g.
  - Persistant programming languages.
  - Pascal/R: An Pascal-extension with the type "relation".
  - 4GLs
  - Procedures stored in the DB Server (e.g. PL/SQL).
  - Object-oriented databases.
  - Deductive databases.

## Overview

1. Introduction and Overview
2. Embedded SQL
3. ODBC
4. JDBC





## Embedded SQL (1)

- You can use SQL in your C, C++, Cobol, ... programs.
- Embedded SQL means that you insert SQL commands as specially marked statements in your program.
- You can use variables of your programming language everywhere where SQL allows a constant.
- For example, insert a new record in the "Course"-table:

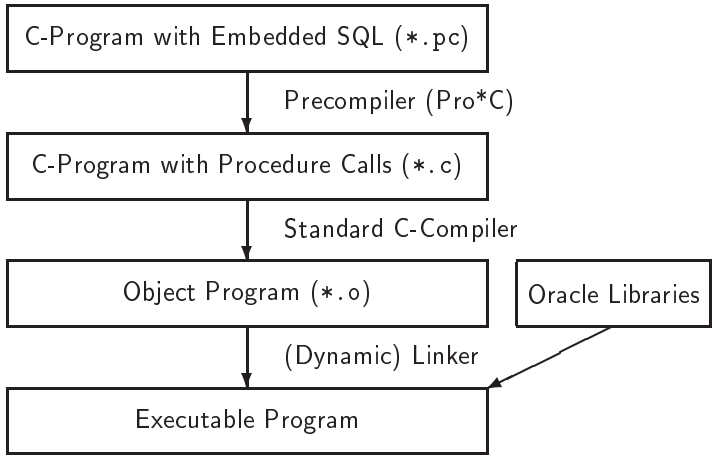
```
EXEC SQL INSERT INTO Course(CRN, Title, Name)
         VALUES (:crn, :title, :instructor);
```

- Here `crn`, `title`, `instructor` are normal "C" variables, and this statement can be in between normal "C" statements.





## Embedded SQL (2)








## Host Variables (1)

- If SQL is embedded in C, then C is the "host language". C-variables which should be used in SQL statements are called "host variables".
- Note that the database has a type system which is quite different from the "C" type system.
  E.g. "C" has no type "`DATE`", and no C type corresponds to "`NUMBER(30)`".
- In addition, "C" has no notion of null values.
- Furthermore, even if there is a natural correspondence between an SQL data type and a C data type, the storage format Oracle uses can differ from the C representation.

## Host Variables (2)

- E.g. Oracle stores strings (`VARCHAR`) with the length followed by an array of characters, whereas C uses an array of characters with a special marker at the end (null character).
  Also Oracle stores numbers with mantissa and exponent (scientific notation), where the mantissa is stored in a kind of BCD format (4 bits for every digit). But C uses a binary representation.
- So some sort of type / memory format conversion has to take place whenever data values are passed from the DB to the program or vice versa.
- The precompiler can do quite a lot, but some work remains for the programmer.





## Host Variables (3)

- Oracle has a list of "internal types" (i.e. Oracle types) and "external types" (i.e. types of C and other host languages) and possible conversions between them.
- Many conversions are done automatically, e.g. from `NUMBER`($p$), $p$ not big, into the C-type "`int`".
  Also `NUMBER`($p$,$s$) can be converted to and from the C-type "`float`", although precision may be lost.
- However, for `VARCHAR`, you either have to use an unusual C-type which corresponds to Oracle's representation, or you must explicitly state that you want to use a character array with the C-convention for marking the string end.





## Host Variables (4)

- The precompiler must be able to understand the declaration of those variables which should be used in SQL statements.
- The precompiler does not necessarily understand the full "C" syntax. In general it can do its job by looking only at the statements specially marked with the prefix "`EXEC SQL`".
  It is also not really desirable that the precompiler checks the complete C syntax. Then you would have to write "C" which is acceptable to two compilers.
- So the variable declarations relevant to the precompiler must be enclosed in "`EXEC SQL BEGIN DECLARE SECTION`" and "`EXEC SQL END DECLARE SECTION`".





## Host Variables (5)

- E.g. the declare section can look as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
      int crn;
      varchar title[40];
      char name[21];
      EXEC SQL VAR name IS STRING(21);
EXEC SQL END DECLARE SECTION;
```

- In C, a variable declaration is written as

  ⟨`Type`⟩ ⟨`Variable`⟩`;`

  But e.g. for array types, on the right hand side something like an expression is written, which would give the type on the left.

## Host Variables (6)

- So "crn" is a standard C integer variable, and Oracle can convert it automatically to and from NUMBER($p$).
- In C, you assign a value in the form

```
crn = 22268;
```

- You can read an integer in decimal notation with

```
scanf("%d", &crn);
```

"&crn" denotes a pointer to crn. C has only "call by value": You must pass the address of crn such that scanf can set it. "scanf" leaves a line end in the input. See also the example.

- You can print an integer (to the terminal) with

```
printf("%d", crn);
```





## Host Variables (7)

- "name" is an array of 21 characters (index range 0..20).
- It can contain a string of up to 20 characters, followed by the ASCII NUL character ('\0') to mark the end of the string.
  E.g. "Brass" is stored as name[0]='B', name[1]='r', ..., name[5]='\0'. Here an explicit type-declaration for the precompiler is required (or the option CHAR_MAP=STRING).
- Strings can be read e.g. with "scanf("%s", name);".
  For arrays, always the memory addres is passed, thus no "&".
  This command will read only a string consisting of non-blank characters (a word). To read an entire line use "gets(name);" or "fgets(name,21,stdin); name[strlen(name)-1]='\0';".
- You can print strings with "printf("%s", name);".





## Host Variables (8)

- "varchar title[40]" is not a standard C data type.
- The precompiler translates it into

```
struct{
      unsigned short len;
      unsigned char arr[40];
} title;
```

- One possibility to read this is by

```
scanf("%s", title.arr);
title.len = strlen(title.arr);
```

Then only strings up to 39 characters can be read. If the input is longer, other variables on your stack will be overwritten!





## Host Variables (9)

- In the same way, Oracle strings can be printed by transforming them first into C strings:

```
title.arr[title.len] = '\0';
printf("%s", title.arr);
```

- You can assign/copy a string in this way:

```
strcpy(title.arr, "Database Management");
title.len = strlen(title.arr);
```

- You need space for the additional NUL byte in each of these operations. You can declare the host variable longer than the corresponding database column.

## Host Variables (10)

- The variables in the `DECLARE SECTION` can be global or local.
- The types of these variables must be such that the precompiler can understand them.
- Especially, no non-standard types (defined by the programmer with `typedef`) are allowed.
- Host variables can have the same name as database columns since they are marked in the SQL commands with a ":".





## Error Checking (1)

- After every execution of an SQL command you have to check whether an error happened.
- One possibilities for this is to declare a variable

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

- As required by the SQL-92 standard, if such a variable is declared, Oracle stores a "Return Code" in it after executing an SQL command.

  The first two characters contain a "class code", e.g. `'00'` means "ok" and `'02'` means that there are no more tuples to be returned by the query. The following three characters can contain a "subclass code".





## Error Checking (2)

- Another possibility, only retained for compatibility with the SQL-86 standard, is to declare a variable

```
long SQLCODE;
```

  Here 0 means successful execution, 100 means that there are no further tuples, and negative values denote errors.
- Oracle (but not the standard) in addition offers a data structure called "SQL Communications Area", containing various information including a return code.
- You declare this structure with

```
EXEC SQL INCLUDE SQLCA;
```





## Error Checking (3)

- The variable is called "`sqlca`".
- It is a record, the component "`sqlca.sqlcode`" contains the return code.

  It is 0 if the SQL command executed normally, and 1403 if there are no further tuples.
- The component "`sqlca.sqlerrm.sqlerrmc`" contains the text of an error message, and "`sqlca.sqlerrm.sqlerrml`" contains its number of characters. You can print it with

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

  You can get the message also with the procedure `sqlglm`.

## Error Checking (4)

- You can tell the precompiler to automatically insert an error check after every "EXEC SQL" command:

  EXEC SQL WHENEVER SQLERROR GOTO ⟨Label⟩;

- The DO clause allows to execute any C-statement:

  EXEC SQL WHENEVER SQLERROR DO error();

  The DO-clause is an Oracle extension, otherwise the WHENEVER declaration is part of SQL-86 and SQL-92.

- You can cancel previous WHENEVER-declarations with

  EXEC SQL WHENEVER SQLERROR CONTINUE;

  A WHENEVER declaration applies to all EXEC SQL statements which follow in in the program text, until the next WHENEVER declaration.





## Example (1)

```
(1)  #include <stdio.h>
(2)  EXEC SQL INCLUDE SQLCA;
(3)  EXEC SQL BEGIN DECLARE SECTION;
(4)     VARCHAR user[128];
(5)     VARCHAR passwd[32];
(6)     int crn;
(7)     varchar title[40];
(8)     char name[21];
(9)  EXEC SQL END DECLARE SECTION;
(10)
(11) main()
```





## Example (2)

```
(12) {
(13)     char line[80];
(14)     /* Catch errors:  */
(15)     EXEC SQL WHENEVER SQLERROR GOTO error;
(16)     /* Log into Oracle:  */
(17)     strcpy(user.arr, "SCOTT");
(18)     user.len = strlen(user.arr);
(19)     strcpy(passwd.arr, "TIGER");
(20)     passwd.len = strlen(passwd.arr);
(21)     EXEC SQL CONNECT :user
(22)            IDENTIFIED BY :passwd;
```





## Example (3)

```
(23)     /* Read data of new course:  */
(24)     printf("Please enter CRN: ");
(25)     gets(line);
(26)     sscanf(line, "%d", &crn);
(27)     printf("Enter course title:  ");
(28)     gets((char*) title.arr);
(29)     title.len = strlen(title.arr);
(30)     printf("Enter instructor name:  ");
(31)     gets(name);
(32)     printf("CRN=%d, Title='%s', Name='%s'\n",
(33)        crn, title.arr, name);
```

## Example (4)

```
(34)
(35)      /* Execute INSERT statement:  */
(36)      EXEC SQL INSERT
(37)          INTO Course(CRN, Title, Name)
(38)          VALUES(:crn, :title, :name);
(39)
(40)      /* End transaction, Log off:  */
(41)      EXEC SQL COMMIT WORK RELEASE;
(42)
(43)      /* Terminate Program:  */
(44)      return(0);
```





## Example (5)

```
(45)      /* In case of errors:  */
(46)      error:
(47)          EXEC SQL WHENEVER SQLERROR CONTINUE;
(48)          printf("Oracle Error:  %.*s\n",
(49)              sqlca.sqlerrm.sqlerrml,
(50)              sqlca.sqlerrm.sqlerrmc);
(51)          EXEC SQL ROLLBACK WORK RELEASE;
(52)          exit(1);
(53)  }
```





## Practical Hints (1)

- You find example programs in

  `$ORACLE_HOME/precomp/demo/proc`

- Especially, this directory contains a `makefile` "`demo_proc.mk`" which calls the precompiler, the compiler, and the linker.

  The number of libraries which must be linked to your program is quite nasty.

- When the final program is executed, the environment variables need to be correctly set.

  If you execute the program yourself, this is no problem. But when it is a cgi-program started by the web server, you must set the environment variables.





## Practical Hints (2)

- You can look at the result of the precompilation.

  It is a standard C program with the `EXEC SQL` replaced by initializations of data structures and then one or more procedure calls. The original `EXEC SQL` statement is preserved in comments.

- If the C-compiler prints an error message, the line number refers to the "`*.c`" file, and not to the "`*.pc`" file.

  C has a mechanism for avoiding this (`#line`), but Pro*C doesn't use it.

- If you develop a larger program with several modules, you probably want to have only one "`*.pc`"-module, and otherwise standard C modules.

## Simple Queries (1)

- We have seen how to pass values from the program into the database (e.g. for INSERT).
- We now have to explain how to get values out of the database into program variables.
- If we know that a query can return at most one tuple, we can use the following format:

```
EXEC SQL SELECT Title, Name
         INTO   :title, :name
         FROM   Course
         WHERE  CRN = :crn;
```





## Simple Queries (2)

- It is an error if the SELECT yields more than one tuple.
- We must still check whether a tuple was found.
  If the query result was empty, the INTO variables are not set.
- One possibility is to check the status code:

```
if(sqlca.sqlcode == 0) {
     ...  print title and name ...
}
```

- Another possibility is to use

```
EXEC SQL WHENEVER NOT FOUND GOTO empty;
```

  before the SELECT...INTO... (contained in SQL-86/92).





## General Queries (1)

- If the query can yield more than one tuple, the tuples must be read one after the other in a loop.
- In order to do this, you first have to declare a "cursor" for the query:

```
EXEC SQL DECLARE c1 CURSOR FOR
         SELECT  CRN, Title
         FROM    Course
         WHERE   Name = :name;
```

- Then you have to open the cursor:

```
EXEC SQL OPEN c1;
```





## General Queries (2)

- At the OPEN, the query is actually evaluated, and the current value of ":name" is used.
- Next, we need a loop, in which we use FETCH to get a single tuple at a time into our host variables:

```
EXEC SQL WHENEVER NOT FOUND GOTO done;
while(1) { /* 1 is true, i.e.  while(forever) */
     EXEC SQL FETCH c1 INTO :crn, :title;
     ...  Process result tuple ...
}
done:  ...
```

## General Queries (3)

- You can avoid the GOTO if you use the DO clause which is not part of the standard:

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

  The C statement "break" terminates the nearest enclosing loop.
- Finally, you can also watch the return code:

```
while(sqlca.sqlcode == 0) {
```

  You can use the SQL-86 "SQLCODE" or the SQL-92 "SQLSTATE" instead to make this portable. You must check the return code after the FETCH statement, before you process the data.
- Finally, you have to close the cursor:

```
EXEC SQL CLOSE c1;
```





## Positioned Updates/Deletes

- You can refer to the tuple last FETCHed in UPDATE and DELETE commands:

```
EXEC SQL DELETE FROM Course
         WHERE CURRENT OF c1;
```

- This is helpful if you have to ask the user or if you need to compute the new attribute value in C (and not in SQL).
- In this case you should add the "FOR UPDATE" clause to the query (this locks the current tuple).

  Of course, this works only with queries where every result tuple is derived from a single source tuple. You cannot use positioned updates with join or aggregation queries.





## Null Values (1)

- If a query can yield a null value, you have to use two variables: One for the data value, and one "boolean" variable which indicates whether the value is null.
- Such variables are called "Indicator-Variables".

  They must be declared as "short" in Oracle.
- The indicator variable will be set to -1 if a null value was returned, and 0 if a normal value was returned.

  Oracle uses other codes besides these two, but this does not agree with the standard.
- It is an error if a column should be null and you try to FETCH it into a variable without indicator.





## Null Values (2)

- You can attach an indicator-variable to any variable:

```
EXEC SQL FETCH c1 INTO :crn,
                       :title INDICATOR :ind;
```

- For "crn" no indicator variable is needed since it is the primary key and thus not null.
- In Oracle, you do not need the keyword "INDICATOR".
- You can use indicator variables in any statement, e.g. also in an INSERT to set columns to null.

## Using Arrays as Host Variables

- It is not very efficient to exchange a large number of tuples one by one between database and application program.
- Oracle allows to use arrays as host variables for exchanging larger chunks of data in one step.
- However, this is not part of the SQL-92 standard.
- The number of tuples returned can be found in `sqlca.sqlerrd[2]`.
  Counts tuples fetched until now after cursor was opened.
- If you want to insert an array of values, you can specify the number of values as "`FOR :n INSERT INTO ...`".
- Oracle also allows to use records.





## Dynamic SQL (1)

- Until now, we have assumed that the table and column names were already known when the program was written, only some contents were not known until runtime.
- In this case, the precompiler can check the existence of the tables and columns.
  Oracle Pro*C has an option for this.
- In some systems, the queries are already optimized and a "query evaluation plan" (program for executing the query) is stored in the database.
  Oracle does not do this, but it cashes QEPs for the most recently executed queries.





## Dynamic SQL (2)

- It is possible to compose SQL commands only at runtime in string variables.
- If the SQL command is not a query (and does not contain host variables), it can be executed in this way:

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

- One of the problems of Dynamic SQL is that the command is compiled (into a QEP) every time it is executed, whereas it is compiled at precompilation time (on some systems) or only once when it is executed for the first time.
- It is possible to use `PREPARE` for compiling the statement and then `EXECUTE...USING` ⟨`Variables`⟩ (see manual).





## Dynamic SQL (3)

- Dynamic queries are quite complicated, since here also the result columns are not known when you write the program.
- In this case, an "SQL Descriptor Area" (SQLDA) is used.
- The `DESCRIBE` statement will store the number, names, and data-types of the resulting columns into the SQLDA.
- The SQLDA also contains place for pointers to variables which will contain the retrieved data elements.
  You allocate memory for these variables after `DESCRIBE`.
- See the manuals for more information.
  You will have to use `sqlald` (to allocate an SQLDA), `PREPARE`, `OPEN`, `DESCRIBE`, `FETCH`.

## Portability Issues (1)

- In general, the SQL standard is mainly for Embedded SQL, so that application programs become portable.
  For ad-hoc queries, a standard is not so important.
- The CASCADE project was ported from Oracle Pro*C to Microsoft ESQL (running on MS SQL Server).
  ESQL is no longer supported by Microsoft. Microsoft wants you to use ODBC.
- One problem was that the SQL commands contained non-standard constructs, only supported by Oracle, e.g.
  - the strange outer join syntax,
  - the `START WITH ...CONNECT BY ...` construct.





## Portability Issues (2)

- Another problem was that the type conversion with the host variables was different:
  - MS ESQL does not have the `VARCHAR` type for host variables (containing length and character array).
  - If you declare a host variable as `char x[`$n$`]`, it will be filled with spaces until the length $n$.
    You can have standard C strings in MS ESQL, but only if the host variable is a character pointer.
  - Elements of the `DATE` type were converted very differently into character arrays.
    It was already lucky that the `DATE` type is supported.





## Portability Issues (3)

- Using the return codes for executed SQL commands also caused problems:
  - You have to use `SQLCODE` in MS ESQL (Oracle's `sqlca` is not contained in the Standard).
  - "`WHENEVER NOT FOUND DO break;`" does not work in MS ESQL (it is also not contained in the standard).
    MS ESQL allows `WHENEVER ... CALL` ⟨`Procedure`⟩.
- Also the data dictionary is structured differently in Oracle and MS SQL Server.





## Overview

1. Introduction and Overview
2. Embedded SQL
3. ODBC
4. JDBC

## Introduction (1)

- ODBC (Open Database Connectivity) is an interface for application programs (API) to send SQL commands to a DBMS and get back the results.
- ODBC was developed by Microsoft. In parallel, an SQL Call Level Interface (CLI), which is a subset of ODBC 3.x, was standardized by X/OPEN and ISO/IEC in 1995.
- The goal of ODBC is to make application programs portable: ODBC drivers are available for a many different DB systems.
  Like printer drivers implement a common interface for printers, so that your word processing software can print on any printer, your ODBC-application can work with any DBMS which has an ODBC driver.





## Introduction (2)

- ODBC does not make the differences between systems completely invisible, but it
  - guarantees a minimal set of SQL constructs (very few),
    Every driver must support at least this minimal syntax.
  - has many functions where your program can get information about features of the DBMS, and
  - defines certain escape-sequences in SQL statements which the driver replaces by a DBMS-specific syntax.
    E.g. for date etc. constants, for the `ESCAPE` clause of the `LIKE` predicate, for outer joins, and for procedure calls.
- ODBC drivers are also available for non-relational data sources, e.g. spreadsheets and flat files.





## Introduction (3)

- ODBC is naturally very common on Windows platforms: There Oracle and DB2 come also with an ODBC driver.
- But e.g. ODBC drivers for Oracle on Solaris are available from a number of third-party vendors (e.g. Merant, Intersolv).
- Since only procedure calls are used, no precompiler is needed.
- ODBC uses dynamic SQL: SQL queries are passed as strings to the DBMS, so they can be constructed at runtime.
  It is a bit slower than static embedded SQL, where already the precompiler generates a query evaluation plan (in DB2).
- However, ODBC allows to optimize a statement only once and then execute it for different parameter values.





## Using ODBC (1)

- In order to use ODBC, you need:
  - An ODBC driver for your DBMS.
    Most DBMS for Windows come with such a driver.
  - Header-Files which define constants and types needed for calling the ODBC procedures.
    The header files as well as the driver manager are part pf the Microsoft ODBC Software Development Kit (SDK) and should come e.g. with Microsoft Visual C++.
  - The driver manager, a library (DLL), to which your ODBC application is linked.
    It is responsible for loading the correct driver and forwarding your function calls to the driver.

## Using ODBC (2)

- You need to configure an ODBC data source in the control panel.

  E.g. what ODBC driver should be used? To which database should it connect? You can specify username and password either here or in your application program.
- You communicate with the driver via handles (opaque pointers to data structures in the driver). There are 4 types of handles:
  - Environment Handles
  - Database Connection Handles (for logging into a DBMS)
  - Statement Handles (SQL statement in execution)
  - Descriptor Handles (with subtypes)





## Using ODBC (3)

- If strings are input to an ODBC function, two arguments are used: the array containing the characters and the length.
- The special constant `SQL_NTS` can be used for the length to indicate that the character array is a Null-Terminated String.
- E.g. for opening a database connection, you must specify a connection handle, the data source name, user and password (if not specified in the data source):

```
SQLConnect(conn,
           (SQLCHAR*) "MyOracleDB", SQL_NTS,
           (SQLCHAR*) "SCOTT", SQL_NTS,
           (SQLCHAR*) "TIGER", SQL_NTS);
```





## Using ODBC (4)

- Every ODBC function returns a result code. It should normally be `SQL_SUCCESS` or `SQL_SUCCESS_WITH_INFO`.

  It is good programming practice to check these result codes. This was not done in the program example shown below for space reasons. You can get a full version from `http://www2.sis.pitt.edu/~sbrass/db/odbc_ex1.c`.
- If an ODBC call fails, you can get more information with `SQLGetDiagRec`.
- The easiest way to execute an SQL statement is to pass the string to `SQLExecDirect` (using a statement handle):

```
SQLExecDirect(stmt,
     (SQLCHAR*) "SELECT ENAME FROM EMP", SQL_NTS)
```





## Using ODBC (5)

- In order to fetch result rows, you must bind C variables to columns. ODBC can do a type conversion, you only have to specify the type of the C variable. E.g. for a character array:

```
SQLBindCol(stmt, 1, SQL_C_CHAR,
     (SQLPOINTER) &name, (SQLINTEGER) 40, &ind);
```

  I.e. the first column is bound to the character array `name` of size 40. The variable `ind` will be set to the length or `SQL_NULL_DATA` for a null value. The driver will also null-terminate the string `name`.
- `SQLExecDirect` creates a cursor in the statement handle which is positioned just before the first result row.
- `SQLFetch(stmt)` will store the first/next row in the C variables bound to the columns.

## Using ODBC (6)

- If the same SQL statement should be used repeatedly with different values, you can use ? to mark the parameters:

```
SQLPrepare(stmt, (SQLCHAR*)
    "SELECT ENAME FROM EMP WHERE EMPNO = ?",
    SQL_NTS)
```

- Then you use SQLBindParameter to bind C variables to the parameter. You must specify the type of the C variable and the SQL type to which is should be converted:

```
SQLBindParameter(stmt, 1, SQL_PARAM_INPUT,
    SQL_C_SLONG, SQL_NUMERIC,4,0,/*NUMERIC(4,0)*/
    &empno, 0, NULL)
```

- The prepared statement is executed with SQLExecute(stmt).





## ODBC Example (1)

```
(1)  #include <stdio.h>
(2)  #include <windows.h>
(3)  #include <Sql.h>
(4)  #include <Sqlext.h>
(5)  #include <Sqltypes.h>
(6)  int main()
(7)  {
(8)     SQLRETURN rc;  /* Return Code */
(9)     SQLHENV env;   /* Environment Handle */
(10)    SQLHDBC conn;  /* DB Connection Handle*/
(11)    SQLHSTMT stmt; /* Statement Handle */
```





## ODBC Example (2)

```
(12)    SQLINTEGER prod_id;
(13)    SQLINTEGER id_null;
(14)    SQLCHAR prod_name[50];
(15)    SQLINTEGER name_null;
(16)
(17)    /* Allocate Environment Handle:  */
(18)    SQLAllocHandle(SQL_HANDLE_ENV,
(19)           SQL_NULL_HANDLE, &env);
(20)    SQLSetEnvAttr(env, SQL_ATTR_ODBC_VERSION,
(21)           (SQLPOINTER) SQL_OV_ODBC3,
(22)           (SQLINTEGER) 0);
```





## ODBC Example (3)

```
(23)    /* Allocate Connection Handle:  */
(24)    SQLAllocHandle(SQL_HANDLE_DBC,
(25)           env, &conn);
(26)    /* Connect to Database:  */
(27)    rc = SQLConnect(conn,
(28)           (SQLCHAR*) "Northwind", SQL_NTS,
(29)           (SQLCHAR*) "", SQL_NTS, /* User */
(30)           (SQLCHAR*) "", SQL_NTS); /* Pwd */
(31)    if(rc != SQL_SUCCESS
(32)           && rc != SQL_SUCCESS_WITH_INFO)
(33)           ...
```

## ODBC Example (4)

```
(34)    /* Allocate Statement Handle:  */
(35)    SQLAllocHandle(SQL_HANDLE_STMT,
(36)           conn, &stmt);
(37)
(38)    /* Execute a Query:  */
(39)    rc = SQLExecDirect(stmt, (SQLCHAR*)
(40)           "SELECT ProductID, ProductName "
(41)           "FROM Products ORDER BY 1",
(42)                 SQL_NTS);
(43)    if(rc != SQL_SUCCESS)
(44)           ...
```





## ODBC Example (5)

```
(45)    /* Bind Variables to Output Columns:  */
(46)    SQLBindCol(stmt, 1, SQL_C_SLONG,
(47)           (SQLPOINTER) &prod_id,0, &id_null);
(48)    SQLBindCol(stmt, 2, SQL_C_CHAR,
(49)           (SQLPOINTER) &prod_name,
(50)           (SQLINTEGER) sizeof(prod_name),
(51)           &name_null);
(52)    /* Print Query Result:  */
(53)    rc = SQLFetch(stmt);
(54)    if(rc == SQL_NO_DATA_FOUND)
(55)           printf("Query result is empty\n");
```





## ODBC Example (6)

```
(56)    else {
(57)       while(rc == SQL_SUCCESS ||
(58)              rc == SQL_SUCCESS_WITH_INFO) {
(59)          printf("%6d ", prod_id);
(60)          if(name_null == SQL_NULL_DATA)
(61)             printf("(null)\n");
(62)          else
(63)             printf("%s\n", prod_name);
(64)          rc = SQLFetch(stmt);
(65)       }
(66)    }
```





## ODBC Example (7)

```
(67)    /* Free Handles, Close DB Connection:  */
(68)    SQLFreeHandle(SQL_HANDLE_STMT, stmt);
(69)    SQLDisconnect(conn);
(70)    SQLFreeHandle(SQL_HANDLE_DBC, conn);
(71)    SQLFreeHandle(SQL_HANDLE_ENV, env);
(72)
(73)    /* We are done:  */
(74)    printf("Query successfully completed!\n");
(75)    return(0);
(76) }
```

## Other APIs

- Microsoft has developed newer object-oriented interfaces to databases:
  - ADO: Microsoft AcitiveX Data Objects: Easier interface to OLE DB.
  - OLE DB: COM API for accessing data.
- Although both are intended to be open interfaces and support other data sources than Microsoft SQL Server, there is no independent standard for them (in contrast to ODBC, which is compatible with SQL CLI).

  However, there is an "OLE DB Provider" which can use an ODBC driver to access data.
- See: `http://www.microsoft.com/data/`





## Overview

1. Introduction and Overview

2. Embedded SQL

3. ODBC

4. JDBC





## Introduction (1)

- JDBC is an API for accessing databases from Java Programs.
- It based on the SQL CLI (ODBC), but adds a nice object-oriented interface.
- The first JDBC drivers were JDBC-ODBC bridges, i.e. they used an ODBC driver to access the data.
- Important classes are:
  - `DriverManager`: Provides access to JDBC.
  - `Connection`: Result of logging into a DB.
  - `Statement`: SQL Statement (e.g. query, update).
    `PreparedStatement`: Statements with parameters.
  - `ResultSet`: Result of a query.





## Introduction (2)

- When you have a `Connection` object, you can call its method `CreateStatement()` to create a `Statement` object.
- The `Statement` object then methods
  - `executeQuery(String)` to execute an SQL query. It returns a `ResultSet` object.
  - `executeUpdate(String)` to execute any SQL statement which is not a query. It returns the number of rows affected by the update (if it was an update).
- For a `ResultSet` object, the method `next` moves the cursor to the next row (it must be called also before the first row). It returns `false` if there are no more rows.

## Introduction (3)

- Once you have positioned the cursor on a row with `next`, you can use one of the get-methods of the `ResultSet` object to retrieve a column value.
  - You specify the column number/name as argument and
  - use `getString`, `getInt`, etc. depending on the type.
    These methods will do type conversions if needed.
- If you want to execute a statement several times with different values, use the method `prepareStatement(String)` of the `Connection` class to create an object of the class `PreparedStatement`. This class has methods `setString`, `setInt`, etc. for setting the parameters. Then you can call `executeQuery` or `executeUpdate` (without parameter).





## Introduction (4)

- The JDBC drivers for Oracle (and example programs) are in `$ORACLE_HOME/jdbc`.
  OTN members (membership is free) can also download the drivers from `http://technet.oracle.com/software/`.
- For more information, see `http://java.sun.com/products/jdbc/`
- Thanks to Kevin Ho for the example.
  I slightly modified it and may have introduced errors.





## JDBC Example (1)

```
(1)  import java.sql.*;
(2)  import java.net.URL;
(3)
(4)  class JdbcExample
(5)  {
(6)  public static void main (String args [])
(7)  {
(8)     String driver =
(9)        "oracle.jdbc.driver.OracleDriver";
(10)    String url = "jdbc:oracle:thin:" +
(11)       "@vsam.sis.pitt.edu:1521:sis";
```





## JDBC Example (2)

```
(12)    String username = "scott";
(13)    String password = "tiger";
(14)    String query = "SELECT ENAME FROM EMP";
(15)    String ename;
(16)    try {
(17)       // Load driver:
(18)       Class.forName(driver);
(19)       // Get connection:
(20)       Connection con =
(21)          DriverManager.getConnection
(22)             (url, username, password);
```

## JDBC Example (3)

```
(23)          // Execute statement:
(24)          Statement stm = con.createStatement();
(25)          ResultSet rs =
(26)              stm.executeQuery(query);
(27)
(28)          // Print results:
(29)          while (rs.next()) {
(30)              ename = rs.getString(1);
(31)              System.out.println(ename);
(32)          }
```





## JDBC Example (4)

```
(33)          rs.close();
(34)          stm.close();
(35)          con.close();
(36)      }
(37)      catch(Exception e) {
(38)          System.out.println(e);
(39)      }
(40)  }
(41)  }
```