4 Application Programming

4.1 PL/SQL

4.1.1 Introduction

The development of database applications typically requires language constructs similar to those that can be found in programming languages such as C, C++, or Pascal. These constructs are necessary in order to implement complex data structures and algorithms. A major restriction of the database language SQL, however, is that many tasks cannot be accomplished by using only the provided language elements.

PL/SQL (Procedural Language/SQL) is a procedural extension of Oracle-SQL that offers language constructs similar to those in imperative programming languages. PL/SQL allows users and designers to develop complex database applications that require the usage of control structures and procedural elements such as procedures, functions, and modules.

The basic construct in PL/SQL is a *block*. Blocks allow designers to combine logically related (SQL-) statements into units. In a block, constants and variables can be declared, and variables can be used to store query results. Statements in a PL/SQL block include SQL statements, control structures (loops), condition statements (if-then-else), exception handling, and calls of other PL/SQL blocks.

PL/SQL blocks that specify procedures and functions can be grouped into *packages*. A package is similar to a module and has an interface and an implementation part. Oracle offers several predefined packages, for example, input/output routines, file handling, job scheduling etc. (see directory `$ORACLE_HOME/rdbms/admin`).

Another important feature of PL/SQL is that it offers a mechanism to process query results in a tuple-oriented way, that is, one tuple at a time. For this, *cursors* are used. A cursor basically is a pointer to a query result and is used to read attribute values of selected tuples into variables. A cursor typically is used in combination with a loop construct such that each tuple read by the cursor can be processed individually.

In summary, the major goals of PL/SQL are to

- increase the expressiveness of SQL,
- process query results in a tuple-oriented way,
- optimize combined SQL statements,
- develop modular database application programs,
- reuse program code, and
- reduce the cost for maintaining and changing applications.

4.1.2 Structure of PL/SQL-Blocks

PL/SQL is a block-structured language. Each block builds a (named) program unit, and blocks can be nested. Blocks that build a procedure, a function, or a package must be named. A PL/SQL block has an optional declare section, a part containing PL/SQL statements, and an optional exception-handling part. Thus the structure of a PL/SQL looks as follows (brackets [ ] enclose optional parts):

[<Block header>]
[**declare**
    <Constants>
    <Variables>
    <Cursors>
    <User defined exceptions>]
begin
    <PL/SQL statements>
    [**exception**
        <Exception handling>]
end;

The block header specifies whether the PL/SQL block is a procedure, a function, or a package. If no header is specified, the block is said to be an *anonymous* PL/SQL block. Each PL/SQL block again builds a PL/SQL statement. Thus blocks can be nested like blocks in conventional programming languages. The scope of declared variables (i.e., the part of the program in which one can refer to the variable) is analogous to the scope of variables in programming languages such as C or Pascal.

4.1.3 Declarations

Constants, variables, cursors, and exceptions used in a PL/SQL block must be declared in the declare section of that block. Variables and constants can be declared as follows:

<variable name> [**constant**] <data type> [**not null**] [:= <expression>];

Valid data types are SQL data types (see Section 1.1) and the data type **boolean**. Boolean data may only be *true*, *false*, or *null*. The **not null** clause requires that the declared variable must always have a value different from *null*. <expression> is used to initialize a variable. If no expression is specified, the value *null* is assigned to the variable. The clause **constant** states that once a value has been assigned to the variable, the value cannot be changed (thus the variable becomes a constant). Example:

```
declare
    hire_date       date;                       /* implicit initialization with null */
    job_title       varchar2(80) := 'Salesman';
    emp_found       boolean;                    /* implicit initialization with null */
    salary_incr     constant number(3,2) := 1.5;                  /* constant */
    ...
begin ... end;
```

Instead of specifying a data type, one can also refer to the data type of a table column (so-called anchored declaration). For example, `EMP.Empno%TYPE` refers to the data type of the column `Empno` in the relation `EMP`. Instead of a single variable, a record can be declared that can store a complete tuple from a given table (or query result). For example, the data type `DEPT%ROWTYPE` specifies a record suitable to store all attribute values of a complete row from the table `DEPT`. Such records are typically used in combination with a cursor. A field in a record can be accessed using <record name>.<column name>, for example, `DEPT.Deptno`.

A cursor declaration specifies a set of tuples (as a query result) such that the tuples can be processed in a tuple-oriented way (i.e., one tuple at a time) using the **fetch** statement. A cursor declaration has the form

cursor <cursor name> [(<list of parameters>)] **is** <**select** statement>;

The cursor name is an undeclared identifier, not the name of any PL/SQL variable. A parameter has the form <parameter name> <parameter type>. Possible parameter types are **char**, **varchar2**, **number**, **date** and **boolean** as well as corresponding subtypes such as **integer**. Parameters are used to assign values to the variables that are given in the **select** statement.

Example: We want to retrieve the following attribute values from the table `EMP` in a tuple-oriented way: the job title and name of those employees who have been hired after a given date, and who have a manager working in a given department.

```
cursor employee_cur (start_date date, dno number) is
    select JOB, ENAME from EMP E where HIREDATE > start_date
    and exists (select * from EMP
                where E.MGR = EMPNO and DEPTNO = dno);
```

If (some) tuples selected by the cursor will be modified in the PL/SQL block, the clause **for update**[(<column(s)>)] has to be added at the end of the cursor declaration. In this case selected tuples are locked and cannot be accessed by other users until a **commit** has been issued. Before a declared cursor can be used in PL/SQL statements, the cursor must be opened, and after processing the selected tuples the cursor must be closed. We discuss the usage of cursors in more detail below.

Exceptions are used to process errors and warnings that occur during the execution of PL/SQL statements in a controlled manner. Some exceptions are internally defined, such as ZERO_DIVIDE. Other exceptions can be specified by the user at the end of a PL/SQL block. User defined exceptions need to be declared using <name of exception> **exception**. We will discuss exception handling in more detail in Section 4.1.5

4.1.4 Language Elements

In addition to the declaration of variables, constants, and cursors, PL/SQL offers various language constructs such as variable assignments, control structures (loops, if-then-else), procedure and function calls, etc. However, PL/SQL does not allow commands of the SQL data definition language such as the **create table** statement. For this, PL/SQL provides special packages.

Furthermore, PL/SQL uses a modified **select** statement that requires each selected tuple to be assigned to a record (or a list of variables).

There are several alternatives in PL/SQL to a assign a value to a variable. The most simple way to assign a value to a variable is

declare
    counter **integer** := 0;
    ...
begin
    counter := counter + 1;

Values to assign to a variable can also be retrieved from the database using a **select** statement

select <column(s)> **into** <matching list of variables>
from <table(s)> **where** <condition>;

It is important to ensure that the **select** statement retrieves at most one tuple ! Otherwise it is not possible to assign the attribute values to the specified list of variables and a run-time error occurs. If the **select** statement retrieves more than one tuple, a cursor must be used instead. Furthermore, the data types of the specified variables must match those of the retrieved attribute values. For most data types, PL/SQL performs an automatic type conversion (e.g., from **integer** to **real**).

Instead of a list of single variables, a record can be given after the keyword **into**. Also in this case, the **select** statement must retrieve at most one tuple !

declare
    `employee_rec EMP%ROWTYPE;`
    `max_sal EMP.SAL%TYPE;`
begin
    select `EMPNO, ENAME, JOB, MGR, SAL, COMM, HIREDATE, DEPTNO`
    into `employee_rec`
    from `EMP` **where** `EMPNO` = 5698;
    select max(`SAL`) **into** `max_sal` **from** `EMP`;
    ...
end;

PL/SQL provides **while**-loops, two types of **for**-loops, and continuous loops. Latter ones are used in combination with cursors. All types of loops are used to execute a sequence of statements multiple times. The specification of loops occurs in the same way as known from imperative programming languages such as C or Pascal.

A **while**-loop has the pattern

[<< <label name> >>]
while <condition> **loop**
    <sequence of statements>;
end loop [<label name>] ;

A loop can be named. Naming a loop is useful whenever loops are nested and inner loops are completed unconditionally using the **exit** <label name>; statement.

Whereas the number of iterations through a **while** loop is unknown until the loop completes, the number of iterations through the **for** loop can be specified using two integers.

[<< <label name> >>]
for <index> **in** [**reverse**] <lower bound>..<upper bound> **loop**
    <sequence of statements>
end loop [<label name>] ;

The loop counter <index> is declared implicitly. The scope of the loop counter is only the **for** loop. It overrides the scope of any variable having the same name outside the loop. Inside the **for** loop, <index> can be referenced like a constant. <index> may appear in expressions, but one cannot assign a value to <index>. Using the keyword **reverse** causes the iteration to proceed downwards from the higher bound to the lower bound.

Processing Cursors: Before a cursor can be used, it must be opened using the **open** statement

open <cursor name> [(<list of parameters>)] ;

The associated **select** statement then is processed and the cursor references the first selected tuple. Selected tuples then can be processed one tuple at a time using the **fetch** command

fetch <cursor name> **into** <list of variables>;

The **fetch** command assigns the selected attribute values of the current tuple to the list of variables. After the **fetch** command, the cursor advances to the next tuple in the result set. Note that the variables in the list must have the same data types as the selected values. After all tuples have been processed, the **close** command is used to disable the cursor.

close <cursor name>;

The example below illustrates how a cursor is used together with a continuous loop:

```
declare
    cursor emp_cur is select * from EMP;
    emp_rec EMP%ROWTYPE;
    emp_sal EMP.SAL%TYPE;
begin
    open emp_cur;
    loop
        fetch emp_cur into emp_rec;
        exit when emp_cur%NOTFOUND;
        emp_sal := emp_rec.sal;
        <sequence of statements>
    end loop;
    close emp_cur;
    ...
end;
```

Each loop can be completed unconditionally using the **exit** clause:

exit [<block label>] [**when** <condition>]

Using **exit** without a block label causes the completion of the loop that contains the **exit** statement. A condition can be a simple comparison of values. In most cases, however, the condition refers to a cursor. In the example above, `%NOTFOUND` is a predicate that evaluates to *false* if the most recent `fetch` command has read a tuple. The value of <cursor name>`%NOTFOUND` is *null* before the first tuple is fetched. The predicate evaluates to *true* if the most recent **fetch** failed to return a tuple, and *false* otherwise. `%FOUND` is the logical opposite of `%NOTFOUND`.

Cursor **for** loops can be used to simplify the usage of a cursor:

[<< <label name> >>]
for <record name> **in** <cursor name>[(<list of parameters>)] **loop**
    <sequence of statements>
end loop [<label name>];

A record suitable to store a tuple fetched by the cursor is implicitly declared. Furthermore, this loop implicitly performs a **fetch** at each iteration as well as an **open** before the loop is entered and a **close** after the loop is left. If at an iteration no tuple has been fetched, the loop is automatically terminated without an **exit**.

It is even possible to specify a query instead of <cursor name> in a **for** loop:

for <record name> **in** (<**select** statement>) **loop**
    <sequence of statements>
end loop;

That is, a cursor needs not be specified before the loop is entered, but is defined in the **select** statement.

Example:

for `sal_rec` **in** (**select** `SAL + COMM total` **from** `EMP`) **loop**
    ...;
end loop;

`total` is an alias for the expression computed in the **select** statement. Thus, at each iteration only one tuple is fetched. The record `sal_rec`, which is implicitly defined, then contains only one entry which can be accessed using `sal_rec.total`. Aliases, of course, are not necessary if only attributes are selected, that is, if the **select** statement contains no arithmetic operators or aggregate functions.

For conditional control, PL/SQL offers **if-then-else** constructs of the pattern

if <condition> **then** <sequence of statements>
[**elsif**] <condition> **then** <sequence of statements>
...
[**else**] <sequence of statements> **end if**;

Starting with the first condition, if a condition yields *true*, its corresponding sequence of statements is executed, otherwise control is passed to the next condition. Thus the behavior of this type of PL/SQL statement is analogous to if-then-else statements in imperative programming languages.

Except data definition language commands such as **create table**, all types of SQL statements can be used in PL/SQL blocks, in particular **delete**, **insert**, **update**, and **commit**. Note that in PL/SQL only **select** statements of the type **select** <column(s)> **into** are allowed, i.e., selected attribute values can only be assigned to variables (unless the **select** statement is used in a subquery). The usage of **select** statements as in SQL leads to a syntax error. If **update** or **delete** statements are used in combination with a cursor, these commands can be restricted to currently fetched tuple. In these cases the clause **where current of**<cursor name> is added as shown in the following example.

Example: The following PL/SQL block performs the following modifications: All employees having 'KING' as their manager get a 5% salary increase.

```
declare
    manager EMP.MGR%TYPE;
    cursor emp_cur (mgr_no number) is
        select SAL from EMP
        where MGR = mgr_no
    for update of SAL;
begin
    select EMPNO into manager from EMP
    where ENAME = 'KING';
    for emp_rec in emp_cur(manager) loop
        update EMP set SAL = emp_rec.sal * 1.05
        where current of emp_cur;
    end loop;
    commit;
end;
```

Remark: Note that the record `emp_rec` is implicitly defined. We will discuss another version of this block using parameters in Section 4.1.6.

4.1.5 Exception Handling

A PL/SQL block may contain statements that specify exception handling routines. Each error or warning during the execution of a PL/SQL block raises an exception. One can distinguish between two types of exceptions:

- system defined exceptions
- user defined exceptions (which must be declared by the user in the declaration part of a block where the exception is used/implemented)

System defined exceptions are always automatically raised whenever corresponding errors or warnings occur. User defined exceptions, in contrast, must be raised explicitly in a sequence of statements using **raise** <exception name>. After the keyword **exception** at the end of a block, user defined exception handling routines are implemented. An implementation has the pattern

when <exception name> **then** <sequence of statements>;

The most common errors that can occur during the execution of PL/SQL programs are handled by system defined exceptions. The table below lists some of these exceptions with their names and a short description.

Exception name	Number	Remark
CURSOR_ALREADY_OPEN	ORA-06511	You have tried to open a cursor which is already open
INVALID_CURSOR	ORA-01001	Invalid cursor operation such as fetching from a closed cursor
NO_DATA_FOUND	ORA-01403	A **select** ... **into** or **fetch** statement returned no tuple
TOO_MANY_ROWS	ORA-01422	A **select** ... **into** statement returned more than one tuple
ZERO_DIVIDE	ORA-01476	You have tried to divide a number by 0

Example:

```
declare
    emp_sal EMP.SAL%TYPE;
    emp_no EMP.EMPNO%TYPE;
    too_high_sal exception;
begin
    select EMPNO, SAL into emp_no, emp_sal
    from EMP where ENAME = 'KING';
    if emp_sal * 1.05 > 4000 then raise too_high_sal
    else update EMP set SQL ...
    end if;
    exception
        when NO_DATA_FOUND  -- no tuple selected
            then rollback;
        when too_high_sal then insert into high_sal_emps values(emp_no);
        commit;
end;
```

After the keyword **when** a list of exception names connected with **or** can be specified. The last **when** clause in the exception part may contain the exception name **others**. This introduces the default exception handling routine, for example, a **rollback**.

If a PL/SQL program is executed from the SQL*Plus shell, exception handling routines may contain statements that display error or warning messages on the screen. For this, the procedure **raise_application_error** can be used. This procedure has two parameters <error_number> and <message_text>. <error_number> is a negative integer defined by the user and must range between -20000 and -20999. <error_message> is a string with a length up to 2048 characters. The concatenation operator "||" can be used to concatenate single strings to one string. In order to display numeric variables, these variables must be converted to strings using the function **to_char**. If the procedure **raise_application_error** is called from a PL/SQL block, processing the PL/SQL block terminates and all database modifications are undone, that is, an implicit **rollback** is performed in addition to displaying the error message.

Example:

if `emp_sal` $*$ 1.05 > 4000
then raise_application_error(-20010, 'Salary increase for employee with Id '
|| **to_char**(`Emp_no`) || ' is too high');

4.1.6 Procedures and Functions

PL/SQL provides sophisticated language constructs to program procedures and functions as stand-alone PL/SQL blocks. They can be called from other PL/SQL blocks, other procedures and functions. The syntax for a procedure definition is

create [**or replace**] **procedure** <procedure name> [(<list of parameters>)] **is**
<declarations>
begin
<sequence of statements>
[**exception**
<exception handling routines>]
end [<procedure name>];

A function can be specified in an analogous way

create [**or replace**] **function** <function name> [(<list of parameters>)]
return <data type> **is**
...

The optional clause **or replace** re-creates the procedure/function. A procedure can be deleted using the command **drop procedure** <procedure name> (**drop function** <function name>). In contrast to anonymous PL/SQL blocks, the clause **declare** may not be used in procedure/function definitions.

Valid parameters include all data types. However, for **char**, **varchar2**, and **number** no length and scale, respectively, can be specified. For example, the parameter **number**(6) results in a compile error and must be replaced by **number**. Instead of explicit data types, implicit types of the form `%TYPE` and `%ROWTYPE` can be used even if constrained declarations are referenced. A parameter is specified as follows:

<parameter name> [**IN** | **OUT** | **IN OUT**] <data type> [{ := | `DEFAULT`} <expression>]

The optional clauses **IN, OUT**, and **IN OUT** specify the way in which the parameter is used. The default mode for a parameter is **IN**. **IN** means that the parameter can be referenced inside the procedure body, but it cannot be changed. **OUT** means that a value can be assigned to the parameter in the body, but the parameter's value cannot be referenced. **IN OUT** allows both assigning values to the parameter and referencing the parameter. Typically, it is sufficient to use the default mode for parameters.

Example: The subsequent procedure is used to increase the salary of all employees who work in the department given by the procedure's parameter. The percentage of the salary increase is given by a parameter, too.

```
create procedure raise_salary(dno number, percentage number DEFAULT 0.5) is
    cursor emp_cur (dept_no number) is
        select SAL from EMP where DEPTNO = dept_no
        for update of SAL;
    empsal number(8);
begin
    open emp_cur(dno); - - Here dno is assigned to dept_no
    loop
        fetch emp_cur into empsal;
        exit when emp_cur%NOTFOUND;
        update EMP set SAL = empsal * ((100 + percentage)/100)
        where current of emp_cur;
    end loop;
    close emp_cur;
    commit;
end raise_salary;
```

This procedure can be called from the SQL*Plus shell using the command

```
execute raise_salary(10, 3);
```

If the procedure is called only with the parameter 10, the default value 0.5 is assumed as specified in the list of parameters in the procedure definition. If a procedure is called from a PL/SQL block, the keyword **execute** is omitted.

Functions have the same structure as procedures. The only difference is that a function returns a value whose data type (unconstrained) must be specified.

Example:

```
create function get_dept_salary(dno number) return number is
    all_sal number;
begin
    all_sal := 0;
    for emp_sal in (select SAL from EMP where DEPTNO = dno
                    and SAL is not null) loop
```

```
        all_sal := all_sal + emp_sal.sal;
    end loop;
    return all_sal;
end get_dept_salary;
```

In order to call a function from the SQL*Plus shell, it is necessary to first define a variable to which the return value can be assigned. In SQL*Plus a variable can be defined using the command **variable** <variable name> <data type>;, for example, **variable** `salary` **number**. The above function then can be called using the command **execute** `:salary` := `get_dept_salary`(20); Note that the colon ":" must be put in front of the variable.

Further information about procedures and functions can be obtained using the **help** command in the SQL*Plus shell, for example, **help** [**create**] **function**, **help subprograms**, **help stored subprograms**.

4.1.7 Packages

It is essential for a good programming style that logically related blocks, procedures, and functions are combined into modules, and each module provides an interface which allows users and designers to utilize the implemented functionality. PL/SQL supports the concept of modularization by which modules and other constructs can be organized into *packages*. A package consists of a package specification and a package body. The package specification defines the interface that is visible for application programmers, and the package body implements the package specification (similar to header- and source files in the programming language C).

Below a package is given that is used to combine all functions and procedures to manage information about employees.

```
create package manage_employee as -- package specification
   function hire_emp (name varchar2, job varchar2, mgr number, hiredate date,
                      sal number, comm number default 0, deptno number)
            return number;
   procedure fire_emp (emp_id number);
   procedure raise_sal (emp_id number, sal_incr number);
end manage_employee;

create package body manage_employee as
   function hire_emp (name varchar2, job varchar2, mgr number, hiredate date,
                      sal number, comm number default 0, deptno number)
            return number is
   -- Insert a new employee with a new employee Id
   new_empno number(10);
   begin
     select emp_sequence.nextval into new_empno from dual;
```

```
    insert into emp values(new_empno, name, job, mgr, hiredate,
                           sal, comm, deptno);
    return new_empno;
  end hire_emp;

  procedure fire_emp(emp_id number) is
  -- deletes an employee from the table EMP
  begin
    delete from emp where empno = emp_id;
    if SQL%NOTFOUND then -- delete statement referred to invalid emp_id
    raise_application_error(-20011, 'Employee with Id ' ||
                            to_char(emp_id) || ' does not exist.');
    end if;
  end fire_emp;

  procedure raise_sal(emp_id number, sal_incr number) is
  -- modify the salary of a given employee
  begin
    update emp set sal = sal + sal_incr
    where empno = emp_id;
    if SQL%NOTFOUND then
    raise_application_error(-20012, 'Employee with Id ' ||
                            to_char(emp_id) || ' does not exist');
    end if;
  end raise_sal;
end manage_employee;
```

Remark: In order to compile and execute the above package, it is necessary to create first the required sequence (**help sequence**):

create sequence emp_sequence **start with** 8000 **increment by** 10;

A procedure or function implemented in a package can be called from other procedures and functions using the statement <package name>.<procedure name>[(<list of parameters>)]. Calling such a procedure from the SQL*Plus shell requires a leading **execute**.

Oracle offers several predefined packages and procedures that can be used by database users and application developers. A set of very useful procedures is implemented in the package DBMS_OUTPUT. This package allows users to display information to their SQL*Plus session's screen as a PL/SQL program is executed. It is also a very useful means to debug PL/SQL programs that have been successfully compiled, but do not behave as expected. Below some of the most important procedures of this package are listed:

Procedure name	Remark
DBMS_OUTPUT.ENABLE	enables output
DBMS_OUTPUT.DISABLE	disables output
DBMS_OUTPUT.PUT(<string>)	appends (displays) <string> to output buffer
DBMS_OUTPUT.PUT_LINE(<string>)	appends <string> to output buffer and appends a new-line marker
DBMS_OUTPUT.NEW_LINE	displays a new-line marker

Before strings can be displayed on the screen, the output has to be enabled either using the procedure DBMS_OUTPUT.ENABLE or using the SQL*Plus command **set serveroutput on** (before the procedure that produces the output is called).

Further packages provided by ORACLE are UTL_FILE for reading and writing files from PL/SQL programs, DBMS_JOB for job scheduling, and DBMS_SQL to generate SQL statements dynamically, that is, during program execution. The package DBMS_SQL is typically used to create and delete tables from within PL/SQL programs. More packages can be found in the directory $ORACLE_HOME/rdbms/admin.

4.1.8 Programming in PL/SQL

Typically one uses an editor such as emacs or vi to write a PL/SQL program. Once a program has been stored in a file <name> with the extension .sql, it can be loaded into SQL*Plus using the command @<name>. It is important that the last line of the file contains a slash "/".

If the procedure, function, or package has been successfully compiled, SQL*Plus displays the message PL/SQL procedure successfully completed. If the program contains errors, these are displayed in the format ORA-n <message text>, where n is a number and <message text> is a short description of the error, for example, ORA-1001 INVALID CURSOR. The SQL*Plus command **show errors** [<function|procedure|package|package body|trigger> <name>] displays all compilation errors of the most recently created or altered function (or procedure, or package etc.) in more detail. If this command does not show any errors, try **select** $*$ **from** USER_ERRORS.

Under the UNIX shell one can also use the command **oerr** ORA n to get information of the following form:

error description
Cause: *Reason for the error*
Action: *Suggested action*

4.2 Embedded SQL and Pro*C

The query language constructs of SQL described in the previous sections are suited for formulating ad-hoc queries, data manipulation statements and simple PL/SQL blocks in simple, interactive tools such as SQL*Plus. Many data management tasks, however, occur in sophisticated engineering applications and these tasks are too complex to be handled by such an interactive tool. Typically, data are generated and manipulated in computationally complex application programs that are written in a Third-Generation-Language (3GL), and which, therefore, need an interface to the database system. Furthermore, a majority of existing data-intensive engineering applications are written previously using an imperative programming language and now want to make use of the functionality of a database system, thus requiring an easy to use programming interface to the database system. Such an interface is provided in the form of *Embedded SQL*, an embedding of SQL into various programming languages, such as C, C++, Cobol, Fortran etc. Embedded SQL provides application programmers a suitable means to combine the computing power of a programming language with the database manipulation and management capabilities of the declarative query language SQL.

Since all these interfaces exhibit comparable functionalities, in the following we describe the embedding of SQL in the programming language C. For this, we base our discussion on the Oracle interface to C, called Pro*C. The emphasis in this section is placed on the description of the interface, not on introducing the programming language C.

4.2.1 General Concepts

Programs written in Pro*C and which include SQL and/or PL/SQL statements are precompiled into regular C programs using a precompiler that typically comes with the database management software (precompiler package). In order to make SQL and PL/SQL statements in a Proc*C program (having the suffix .pc) recognizable by the precompiler, they are always preceded by the keywords **EXEC SQL** and end with a semicolon ";". The Pro*C precompiler replaces such statements with appropriate calls to functions implemented in the SQL runtime library. The resulting C program then can be compiled and linked using a normal C compiler like any other C program. The linker includes the appropriate Oracle specific libraries. Figure 1 summarizes the steps from the source code containing SQL statements to an executable program.

4.2.2 Host and Communication Variables

As it is the case for PL/SQL blocks, also the first part of a Pro*C program has a declare section. In a Pro*C program, in a declare section so-called *host variables* are specified. Host variables are the key to the communication between the host program and the database. Declarations of host variables can be placed wherever normal C variable declarations can be placed. Host variables are declared according to the C syntax. Host variables can be of the following data types:

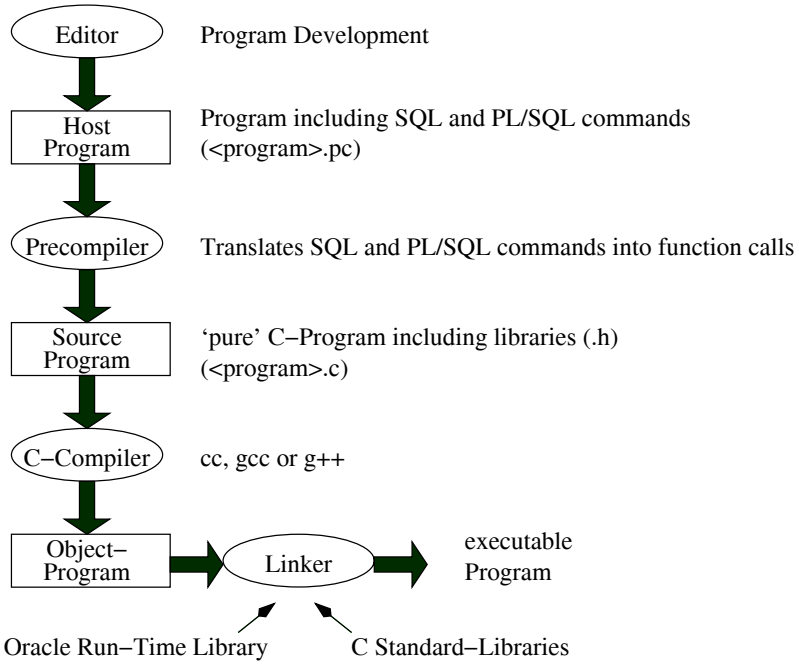

Figure 1: Translation of a Pro*C Program

char <Name>	single character
char <Name>[n]	array of n characters
int	integer
float	floating point
VARCHAR<Name>[n]	variable length strings

VARCHAR[2] is converted by the Pro*C precompiler into a structure with an n-byte character array and a 2-bytes length field. The declaration of host variables occurs in a declare section having the following pattern:

```
EXEC SQL BEGIN DECLARE SECTION
    <Declaration of host variables>
    /* e.g., VARCHAR userid[20]; */
    /* e.g., char test_ok; */
EXEC SQL END DECLARE SECTION
```

In a Pro*C program at most one such a declare section is allowed. The declaration of cursors and exceptions occurs outside of such a declare section for host variables. In a Pro*C program host variables referenced in SQL and PL/SQL statements must be prefixed with a colon ":". Note that it is not possible to use C function calls and most of the pointer expressions as host variable references.

[2]Note: all uppercase letters; **varchar2** is not allowed!

4.2.3 The Communication Area

In addition to host language variables that are needed to pass data between the database and C program (and vice versa), one needs to provide some status variables containing program runtime information. The variables are used to pass status information concerning the database access to the application program so that certain events can be handled in the program properly. The structure containing the status variables is called *SQL Communication Area* or *SQLCA*, for short, and has to be included after the declare section using the statement

EXEC SQL INCLUDE SQLCA.H

In the variables defined in this structure, information about error messages as well as program status information is maintained:

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

The fields in this structure have the following meaning:

`sqlcaid`	Used to identify the SQLCA, set to "SQLCA"
`sqlabc`	Holds the length of the SQLCA structure
`sqlcode`	Holds the status code of the most recently executed SQL (PL/SQL) statement $0 \hat{=}$ No error, statement successfully completed $> 0 \hat{=}$ Statement executed and exception detected. Typical situations are where **fetch** or **select into** returns no rows. $< 0 \hat{=}$ Statement was not executed because of an error; transaction should be rolled back explicitly.
`sqlerrm`	Structure with two components `sqlerrml`: length of the message text in `sqlerrmc`, and `sqlerrmc`: error message text (up to 70 characters) corresponding to the error code recorded in `sqlcode`
`sqlerrp`	Not used

`sqlerrd`	Array of binary integers, has 6 elements: `sqlerrd[0],sqlerrd[1],sqlerrd[3],sqlerrd[6]` not used; `sqlerrd[2]` = number of rows processed by the most recent SQL statement; `sqlerrd[4]` = offset specifying position of most recent parse error of SQL statement.
`sqlwarn`	Array with eight elements used as warning (not error!) flags. Flag is set by assigning it the character 'W'. `sqlwarn[0]`: only set if other flag is set `sqlwarn[1]`: if truncated column value was assigned to a host variable `sqlwarn[2]`: *null* column is not used in computing an aggregate function `sqlwarn[3]`: number of columns in **select** is not equal to number of host variables specified in **into** `sqlwarn[4]`: if every tuple was processed by an **update** or **delete** statement without a **where** clause `sqlwarn[5]`: procedure/function body compilation failed because of a PL/SQL error `sqlwarn[6]` and `sqlwarn[7]`: not used
`sqlext`	not used

Components of this structure can be accessed and verified during runtime, and appropriate handling routines (e.g., exception handling) can be executed to ensure a correct behavior of the application program. If at the end of the program the variable `sqlcode` contains a 0, then the execution of the program has been successful, otherwise an error occurred.

4.2.4 Exception Handling

There are two ways to check the status of your program after executable SQL statements which may result in an error or warning: (1) either by explicitly checking respective components of the SQLCA structure, or (2) by doing automatic error checking and handling using the **WHENEVER** statement. The complete syntax of this statement is

EXEC SQL WHENEVER <condition> <action>;

By using this command, the program then automatically checks the SQLCA for <condition> and executes the given <action>. <condition> can be one of the following:

- SQLERROR: `sqlcode` has a negative value, that is, an error occurred
- SQLWARNING: In this case `sqlwarn[0]` is set due to a warning
- NOT FOUND: `sqlcode` has a positive value, meaning that no row was found that satisfies the **where** condition, or a **select into** or **fetch** statement returned no rows

<action> can be

- STOP: the program exits with an `exit()` call, and all SQL statements that have not been committed so far are rolled back

- CONTINUE: if possible, the program tries to continue with the statement following the error resulting statement
- DO <function>: the program transfers processing to an error handling function named <function>
- GOTO <label>: program execution branches to a labeled statement (see example)

4.2.5 Connecting to the Database

At the beginning of Pro*C program, more precisely, the execution of embedded SQL or PL/SQL statements, one has to connect to the database using a valid Oracle account and password. Connecting to the database occurs trough the embedded SQL statement

EXEC SQL CONNECT :<Account> **IDENTIFIED BY** :<Password>.

Both <Account> and <Password> are host variables of the type **VARCHAR** and must be specified and handled respectively (see also the sample Pro*C program in Section 4.2.7). <Account> and <Password> can be specified in the Pro*C program, but can also be entered at program runtime using, e.g., the C function **scanf**.

4.2.6 Commit and Rollback

Before a program is terminated by the c `exit` function and if no error occurred, database modifications through embedded insert, update, and delete statements must be committed. This is done by using the embedded SQL statement

EXEC SQL COMMIT WORK RELEASE;

If a program error occurred and previous non-committed database modifications need to be undone, the embedded SQL statement

EXEC SQL ROLLBACK WORK RELEASE;

has to be specified in the respective error handling routine of the Pro*C program.

4.2.7 Sample Pro*C Program

The following Pro*C program connects to the database using the database account scott/tiger. The database contains information about employees and departments (see the previous examples used in this tutorial). The user has to enter a salary which then is used to retrieve all employees (from the relation `EMP`) who earn more than the given minimum salary. Retrieving and processing individual result tuples occurs through using a PL/SQL cursor in a C while-loop.

```
/* Declarations */
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>

/* Declare section for host variables */
EXEC SQL BEGIN DECLARE SECTION;
   VARCHAR userid[20];
   VARCHAR passwd[20];
   int empno;
   VARCHAR ename[15];
   float sal;
   float min_sal;
EXEC SQL END DECLARE SECTION;

/* Load SQL Communication Area */
EXEC SQL INCLUDE SQLCA.H;

main()  /* Main program */
{  int retval;
   /* Catch errors automatically and go to error handling rountine */
   EXEC SQL WHENEVER SQLERROR GOTO error;

   /* Connect to Oracle as SCOTT/TIGER; both are host variables      */
   /* of type VARCHAR; Account and Password are specified explicitly */
   strcpy(userid.arr,"SCOTT");    /* userid.arr := "SCOTT" */
   userid.len=strlen(userid.arr); /* uid.len := 5 */
   strcpy(passwd.arr,"SCOTT");    /* passwd.arr := "TIGER" */
   passwd.len=strlen(passwd.arr); /* passwd.len := 5 */

   EXEC SQL CONNECT :userid IDENTIFIED BY :passwd;

   printf("Connected to ORACLE as: %s\n\n", userid.arr);

   /* Enter minimum salary by user */
   printf("Please enter minimum salary > ");
   retval = scanf("%f", &min_sal);

   if(retval != 1) {
      printf("Input error!!\n");
      EXEC SQL ROLLBACK WORK RELEASE;
      /* Disconnect from ORACLE */
      exit(2); /* Exit program */
   }

   /* Declare cursor; cannot occur in declare section! */
   EXEC SQL DECLARE EMP_CUR CURSOR FOR
   SELECT EMPNO,ENAME,SAL FROM EMP
```

```
   WHERE SAL>=:min_sal;

/* Print Table header, run cursor through result set */
printf("Empployee-ID      Employee-Name       Salary \n");
printf("---------------   -----------------   -------\n");
EXEC SQL OPEN EMP_CUR;
EXEC SQL FETCH EMP_CUR INTO :empno, :ename, :sal; /* Fetch 1.tuple */
while(sqlca.sqlcode==0) { /* are there more tuples ? */
   ename.arr[ename.len] = '\0';    /* "End of String" */
   printf("%15d   %-17s   %7.2f\n",empno,ename.arr,sal);
   EXEC SQL FETCH EMP_CUR INTO :empno, :ename, :sal; /* get next tuple */
}
EXEC SQL CLOSE EMP_CUR;

/* Disconnect from database and terminate program */
EXEC SQL COMMIT WORK RELEASE;
printf("\nDisconnected from ORACLE\n");
exit(0);

/* Error Handling: Print error message */
error: printf("\nError: %.70s \n",sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}
```