

# Statische Pro\*C-Programme — Erste Schritte

Jürgen Kalinski  
Informatik III, Universität Bonn  
email: cully@cs.uni-bonn.de

3. November 1998

## 1 Einführung

Die Einbettung von SQL-Anfragen in C-Programme ist unerlässlich, sobald man mit den in ORACLE gespeicherten Tupeln komplexe Operationen durchführen will. Man denke in diesem Zusammenhang beispielsweise an die graphische Präsentation geometrischer Daten oder rechenintensive Auswertungen numerischer Daten. Zu diesem Zweck bietet ORACLE Spracherweiterungen für diverse Programmiersprachen an, die es erlauben, in einer Wirtssprache (z.B. C, COBOL, FORTRAN, ...) SQL-Statements abzusetzen. Ein Pre-Compiler übersetzt diese Programme dann vollständig in die Wirtssprache, so daß anschließend die End-Übersetzung stattfinden kann. Man beachte, daß aufgrund dieses Konzeptes das ursprüngliche Programm mit ORACLE-eigenen Anweisungen angereichert wird (was sich spätestens beim Debuggen bemerkbar macht). Im folgenden wird die C-Schnittstelle PRO\*C in ihren einfachsten Zügen dargestellt. Vertrautheit des Lesers mit SQL wird vorausgesetzt.

Bei der Illustration der wesentlichen Konzepte werden wir uns auf das folgende Beispielschema einschließlich Instanz beziehen:

```
CREATE TABLE Buch (  
  ID      NUMBER(8,0)  NOT NULL,  
  Titel   VARCHAR2(50) NOT NULL,  
  Jahr    NUMBER(4,0),  
  PRIMARY KEY (ID)  
)
```

Buch	ID	Titel	Jahr
	17	DB	1984
	19	IS	-

Es lassen sich *statische* von *dynamischen* PRO\*C-Programmen unterscheiden. Im ersten Fall sind die auszuführenden SQL-Statements von ihrer Struktur her schon zur Compile-Zeit bekannt. Man weiß, welche Relationen abgefragt oder geändert werden und welche Attribute betroffen sind; zur Laufzeit geht

es nur darum, Platzhalter-Variablen konkrete Werte zuzuweisen. Dies kann in beiden Richtungen geschehen: *Eingabe-Variablen* dienen dazu, Werte aus dem C-Programm heraus in SQL-Anfragen einzubinden; im Unterschied dazu dienen *Ausgabe-Variablen* der Überführung von SQL-Daten in C-Variablen. Ein typisches Beispiel für eine statische PRO\*C-Anwendung wäre ein Ausleihvorgang einer Bibliotheksdatenbank: Während die Benutzernummer und ein Buch-Identifikator mit jeweils unterschiedlichen Werten gefüllt werden, sehen die zugrundeliegenden Updates strukturell stets gleich aus. Demgegenüber dürfte eine natürlichsprachliche Datenbank-Schnittstelle ein dynamisches PRO\*C-Programm erfordern.

## 2 Ein erstes Programm

Jedes Pro\*C-Programm endet mit dem Suffix `.pc`. Zur Vorbereitung der Pre-Compilation kopiere man sich das Make-File

```
~cully/public_html/ORACLE/make.mk
```

ins eigene Verzeichnis. (Die Benutzerkennung `'demo-user'` und das Passwort `'demo-pass'` sind durch den ORACLE-Benutzernamen das ORACLE-Passwort zu ersetzen.)

Schließlich gebe man im Make-File statt `'Test'` den Namen des PRO\*C-Programms an. Der Aufruf des Pre-Compilers erfolgt mit

```
make -f make.mk Test
```

Zur Datei `'Test.pc'` wird dann ein ausführbares File `'Test'` erzeugt. Das Demo-Makefile setzt einige der — wie stets in ORACLE — im Übermaß vorhandenen Optionen. Den fortgeschrittenen Benutzer ermutigen wir, seiner Kreativität und Experimentierlust freien Lauf zu lassen.

Der Aufbau eines PRO\*C-Programms wird aus dem folgenden Beispiel ersichtlich.

```
#include <stdio.h>
#include <sqlca.h>

EXEC SQL BEGIN DECLARE SECTION;
    char *username = "demo-user";
    char *password = "demo-pass";
    int jahr;
EXEC SQL END DECLARE SECTION;

int main()
{
    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    EXEC SQL SELECT Jahr
        INTO :jahr
        FROM Buch
        WHERE ID=17;

    printf("%d\n",jahr);

    EXEC SQL COMMIT WORK RELEASE;
    . . . .
}
```

Zunächst werden in der `'DECLARE SECTION'` die Wirts- oder Host-Variablen (hier: `'username'`, `'password'` und `'jahr'`) zum Datenaustausch zwischen dem

C-Programm und ORACLE aufgeführt. Aus Sicht des C-Programms handelt es sich um ganz normale Variablen, die (fast) keiner Sonderbehandlung bedürfen. Gegenüber dem Embedded-SQL-Precompiler werden sie mit einem vorangestellten Doppelpunkt (':jahr') kenntlich gemacht und auf diese Weise auch von gleichnamigen Attributen unterschieden.<sup>1</sup>

Diese Anweisungen werden eingerahmt von der An- und Abmeldung beim Datenbanksystem. Die Anweisung 'EXEC SQL CONNECT' stellt die Verbindung zum ORACLE-Server her.

Man beachte, daß die obige **SELECT**-Anweisung wegen der Selektion über den Buch-Identifikator genau ein Tupel zurückliefert. Damit sind aber auch schon direkt die Probleme des obigen Programms angesprochen:

- Wie erkennt man innerhalb des C-Programms, daß der Wirtsvariablen im SQL-Teil *kein* Wert zugewiesen werden konnte (z.B. bei **WHERE ID=18**)? In diesem Fall wäre die Variable **jahr** beim **printf** nicht initialisiert.
- Ähnlich fatal wäre es, wenn zwar ein Tupel mit dem angegebenen Identifikator in der Datenbank vorliegt, unter dem **Jahr**-Attribut aber einen Nullwert besitzt (z.B. bei **WHERE ID=19**).
- Die Wirtsvariable kann nur einen Ergebniswert aufnehmen. Wie sind Fälle zu behandeln, in denen die **SELECT**-Anweisung mehr als ein Tupel zurückliefert (z.B. wenn die **WHERE**- Bedingung weggelassen wird).

---

<sup>1</sup>NB: Obwohl im C-Teil ein Gleichheitszeichen die Wertzuweisung bezeichnet, steht sie im SQL-Teil stets für den Gleichheitstest!

### 3 Communications Area

Die *SQL Communications Area* 'sqlca' liefert über Komponenten des

```
struct sqlca
```

Informationen über Fehler, die bei der Bearbeitung einer SQL-Anweisung aufgetreten sein könnten, sowie darüber, ob die Anweisung ein Tupel selektiert hat. So etwa ist 'sqlca.sqlcode':

= 0 bei fehlerfreier Bearbeitung der ausgeführten SQL-Anweisung (Dies heißt bei einem **SELECT** insbesondere, daß das Ergebnis nicht leer ist.)

> 0 bei einer leeren Ergebnisrelation oder falls ein **FETCH** das Cursor-Ende erreicht hat (s.u.).

< 0 bei „harten“ ORACLE-, Netzwerk- oder Systemfehlern.

```
#include <stdio.h>
#include <sqlca.h>

EXEC SQL BEGIN DECLARE SECTION;
    char *username = "demo-user";
    char *password = "demo-pass";
    int jahr;
EXEC SQL END DECLARE SECTION;

int main()
{
    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    EXEC SQL SELECT Jahr
        INTO :jahr
        FROM Buch
        WHERE ID=18;

    if (sqlca.sqlcode==0) {
        printf("%d\n",jahr);
    }
    else {
        printf("Fehler oder kein Tupel selektiert!\n");
    }

    EXEC SQL COMMIT WORK RELEASE;
    . . . . .
}
```

Die Schnittstelle C-SQL muß zudem damit fertig werden, daß in der Datenbank Attribute einen speziellen Nullwert haben können, der in der Programmiersprache C nicht existiert. Auch hier wird über einen speziellen Mechanismus die Aussage „Nullwert oder nicht“ separat abgehandelt. Zu Attributen, die Nullwerte annehmen können, muß zusätzlich zur Wirts-Variablen eine Indikator-Variable verwendet werden.

```
#include <stdio.h>
#include <sqlca.h>

EXEC SQL BEGIN DECLARE SECTION;
    char *username = "demo-user";
    char *password = "demo-pass";
    int jahr;
    short ind_jahr;
EXEC SQL END DECLARE SECTION;

int main()
{
    EXEC SQL CONNECT :username IDENTIFIED BY :password;

    EXEC SQL SELECT Jahr
        INTO :jahr:ind_jahr
        FROM Buch
        WHERE ID=19;

    if ((sqlca.sqlcode==0) && (ind_jahr!=-1)) {
        printf("%d\n",jahr);
    }
    else {
        printf("Fehler / kein Tupel selektiert / Nullwert!\n");
    }

    EXEC SQL COMMIT WORK RELEASE;
    . . . .
}
```

## 4 Strings

Die Behandlung von String-Variablen erfordert eine gewisse Umsicht: In C werden Strings grundsätzlich mit einem '\0' beendet, in ORACLE nicht. Ein String, der im C-Programm initialisiert und in einer SQL-Anweisung zum Beispiel in einer Selektion verwendet wird, führt so im allgemeinen zu unerwarteten Ergebnissen: ORACLE selektiert tatsächlich nach dem vollständigen Character-Array einschließlich '\0', das aber gar nicht in der Datenbank steht; umgekehrt bringen Ausgabevariablen Probleme mit sich, weil das Fehlen des Endzeichens '\0' mit den üblichen C-Routinen kollidieren wird.

Für Textfelder gibt es deshalb in PRO\*C den speziellen Datentyp 'VARCHAR', und eine Deklaration wie 'VARCHAR titel[100]' wird vom Pre-Compiler in ein struct umgesetzt:

```
struct {
    unsigned short    len;
    unsigned char     arr[100];
} titel;
```

Aus ORACLE-Sicht gibt die 'len'-Komponente die aktuelle Länge des 'arr'-Strings an (unabhängig davon, wo in dem String eventuell '\0' steht). Andererseits kann man im C-Programm nun die Länge des Strings abfragen und ein '\0' anfügen.

```

    . . . . .
#include <stdio.h>
    . . . . .
EXEC SQL BEGIN DECLARE SECTION;
    . . . . .
    VARCHAR titel[100];
EXEC SQL END DECLARE SECTION;

int main()
{
    . . . . .
EXEC SQL SELECT Titel
    INTO :titel
    FROM Buch
    WHERE ID=17;

    if (sqlca.sqlcode==0) {
        /* Stringende auf '\0' setzen.      */
        titel.arr[titel.len] = '\0';
        printf("%s\n",titel.arr);
    }
    else {
        printf("Fehler oder kein Tupel selektiert!\n");
    }
}
```

```
    }  
    . . . .  
}
```

Da `VARCHAR`-Deklarationen in `unsigned` umgesetzt werden, ist bei der Verwendung von `strcpy` und ähnlichen Funktionen explizites Type Casting erforderlich.

## 5 Cursors

Falls das Ergebnis einer **SELECT**-Anweisung aus mehr als einem Tupel bestehen kann, greift man im allgemeinen auf sogenannte *Cursor* zurück. Ein Cursor ermöglicht das sequentielle Durchlaufen einer Ergebnisrelation. Die Vorgehensweise ist dem Durchlauf durch eine sequentielle Datei vollkommen analog:

- Eine **'DECLARE CURSOR'**-Anweisung bindet einen Cursor-Bezeichner an eine SQL-Anweisung.
- Eine **'OPEN'**-Anweisung bereitet den sequentiellen Durchlauf vor.
- Das **'FETCH'** holt nach und nach die Ergebnis-Tupel ein.
- Eine **'CLOSE'**-Anweisung schließt den Cursor.

Auch hier kann mit `'sqlca.sqlcode'` überprüft werden, ob das Cursor-Ende schon erreicht wurde (entspricht dem End-of-File).

```
EXEC SQL BEGIN DECLARE SECTION;
    . . . .
    int   jahr;
    short ind_jahr;
EXEC SQL END DECLARE SECTION;

int main()
{
    . . . .
    /* Cursor mycur deklarieren.          */
    EXEC SQL DECLARE mycur CURSOR FOR
        SELECT Jahr FROM Buch;

    /* Cursor oeffnen.                    */
    EXEC SQL OPEN mycur;
    EXEC SQL FETCH mycur INTO :jahr:ind_jahr;

    while (sqlca.sqlcode==0) {
        /* End of Cursor nicht erreicht.  */
        if (ind_jahr!=-1) {
            printf("%d\n",jahr);
        }
        EXEC SQL FETCH mycur INTO :jahr:ind_jahr;
    }

    /* Cursor schliessen.                 */
    EXEC SQL CLOSE mycur;
    . . . .
}
```

Statt nun für jeweils ein Tupel eine **FETCH**-Anweisung vom Datenbank-Client an den Server abzusetzen, kann man auch mit einem **FETCH** gleich ein ganzes Array füllen lassen. Die Variable `sqlca.sqlerrd[2]` wird durch das Öffnen des Cursors auf 0 gesetzt und mit jedem **FETCH** um die Anzahl der eingeholten Tupel erhöht; sie gibt somit die Anzahl *insgesamt* schon eingelesener Tupel an.

```
#define FETCHSIZE 100
    /* Mit einem FETCH jeweils FETCHSIZE */
    /* auf einen Streich. */
EXEC SQL BEGIN DECLARE SECTION;
    . . . .
    VARCHAR titel[FETCHSIZE][100];
EXEC SQL END DECLARE SECTION;

int main()
{
    unsigned allFetched = 0;
    unsigned lastFetched = 0;
    unsigned i;
    . . . .
    /* Cursor mycur deklarieren. */
EXEC SQL DECLARE mycur CURSOR FOR
    SELECT Titel FROM Buch;
    /* Cursor oeffnen. */
EXEC SQL OPEN mycur;

    do {
        EXEC SQL FETCH mycur INTO :titel;
        lastFetched = sqlca.sqlerrd[2] - allFetched;
        allFetched = sqlca.sqlerrd[2];
        /* Das FETCH uebertrug lastFetched */
        /* Tupel nach titel. */
        for (i=0; i<lastFetched; i++) {
            /* Stringende auf '\0' setzen. */
            titel[i].arr[titel[i].len] = '\0';
            printf("%s\n", titel[i].arr);
        }
    } while (lastFetched==FETCHSIZE);
    /* Weiter einlesen, solange das Array */
    /* vollstaendig aufgefuellt wird. */

    /* Cursor schliessen. */
EXEC SQL CLOSE mycur;
    . . . .
}
```

Es ist möglich, die einen Cursor definierende SQL-Anweisung erst zur Laufzeit anzugeben. Genau genommen, existieren hier verschiedene Implementierungsvarianten, je nachdem, welche Teile der SQL-Anweisung schon zur Compilezeit festliegen und welche erst zur Laufzeit angegeben werden können. Es folgt ein Beispiel für die einfachste Variante. Die SQL-Anweisung wird in einem String abgelegt, der an ORACLE übergeben wird.

```
EXEC SQL BEGIN DECLARE SECTION;
    . . . . .
    VARCHAR sqlstmt[2000];
    VARCHAR titel[FETCHSIZE][100];
EXEC SQL END DECLARE SECTION;

int main()
{
    . . . . .
    strcpy((char*)sqlstmt.arr,"SELECT Titel ");
    strcat((char*)sqlstmt.arr,"FROM Buch");
    sqlstmt.len = strlen((char*)sqlstmt.arr);
    /* Die SQL-Anweisung wird in          */
    /* sqlstmt.arr abgelegt.              */
    . . . . .
    EXEC SQL PREPARE S FROM :sqlstmt;
    EXEC SQL DECLARE mycur CURSOR FOR S;
    /* Cursor mycur wird an diesen String */
    /* gebunden.                          */
    EXEC SQL OPEN mycur;
    /* Cursor oeffnen.                    */
    /* Nun geht's weiter wie zuvor.      */
    EXEC SQL FETCH mycur INTO :titel;
    . . . . .
    EXEC SQL CLOSE mycur;
    . . . . .
}
```

## 6 Fehler

Im Falle eines ORACLE-Fehlers ist man (zumindest bei der Programmentwicklung) an der entsprechenden Fehlermeldung interessiert. Sie wird ausgeworfen, wenn dem Programm die folgenden Anweisungen *vorangestellt* werden.

```
void error_output()
{
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
    printf("%s\n", sqlca.sqlerrm.sqlerrmc);
}
```

```
EXEC SQL WHENEVER SQLERROR DO error_output();
. . . . .
```

Bei „harten“ Fehlern wird man kein ‘COMMIT’, sondern ein ‘ROLLBACK’ der Transaktion vornehmen wollen. Eine Möglichkeit, diesen Fehlerausstieg zu erreichen, besteht darin, das Label ‘`sqlerror`’ anzuspringen.

```
. . . . .
int main()
{
    EXEC SQL WHENEVER SQLERROR GOTO sqlerror;
    . . . . .
    EXEC SQL COMMIT WORK RELEASE;
    return 0;

sqlerror:
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    return 1;
}
```

## 7 Abschließende Bemerkungen

Das PRO\*C-Programm wird vom Pre-Compiler reichlich mit zusätzlichen Anweisungen versehen. Um sich das Arbeiten mit dem generierten C-Quellcode zu vereinfachen, empfiehlt es sich, die SQL-Anweisungen deutlich sichtbar in Kommentaren einzuschliessen. Die selbst geschriebenen C-Anweisungen lassen sich dann leichter identifizieren.

In den obigen Beispielen waren die Wirtsvariablen stets global vereinbart worden. Diese Vorgehensweise ist zwar unschön, aber im allgemeinen notwendig. Der Pre-Compiler zieht zum Beispiel Variablen aus einer Cursor-Deklaration in das Öffnen des Cursors hinüber. Wenn beide Anweisungen in verschiedenen Funktionen stehen und eine in der Deklaration verwendete Variable lokal definiert ist, verursacht die Pre-Compilation eine Fehlermeldung des C-Compilers.

Ein Blick in den vom Pre-Compiler generierten C-Quellcode zeigt zudem, daß Cursor-Deklarationen im Grunde genommen auskommentiert werden. Die eingebettete 'SELECT'-Anweisung wird in einer globalen Stringvariablen abgelegt und beim Öffnen des Cursors erst ausgeführt. Man achte also darauf, daß im 'SELECT'-Teil verwendeten Eingabevariablen nicht vor dem Öffnen des Cursors ein neuer Wert zugewiesen wird.

Die Cursor-Deklaration muß statisch vor der 'OPEN'-Anweisung und anderen Cursor-Zugriffen stehen. Angenommen beide Anweisungen werden auf zwei verschiedene Funktionen verteilt werden, so daß 'func1' die Deklaration und 'func2' das Öffnen eines Cursors übernimmt. Es reicht nun nicht aus sicherzustellen, daß 'func1' zur Laufzeit vor 'func2' ausgeführt wird: Die erstgenannte Funktion muß auch statisch im PRO\*C-Programm vor der letztgenannten stehen.

Umfangreiche Online-Dokumentation zu ORACLE kann mit jedem HTML-Browser unter

```
/usr/datasystems/oracle/doc01
```

gelesen werden. Unter

```
/usr/datasystems/oracle/orahome/precomp/demo/proc
```

ist eine Vielzahl von Beispielprogrammen zu finden.