Embedded SQL

- SQL-Anweisungen mit dem Präfix `EXEC SQL`, eingefügt in C-Programm namens *dateiname.pc*

- Precompiler *proc* erzeugt daraus 'normales' C-Programm *dateiname.c*

- dieses mit herkömmlichem C-Compiler übersetzen *dateiname.o*

- beim Linken DB-Bibliotheken hinzubinden

- Embedded SQL ist standardisiert

Embedded SQL

- Übergabe von Daten an SQL: mit `:var` Wert der C-Variablen `var` übergeben

```
EXEC SQL INSERT INTO artikel VALUES (:anr, :bez);
```

- dazu: in SQL-Anweisungen auftretende C-Variablen dem Precompiler in Deklarationsblock bekannt machen:

```
EXEC SQL BEGIN DECLARE SECTION;
NUMBER anr;
VARCHAR2 bez;
int zaehler;
EXEC SQL END DECLARE SECTION;
```

- Typumwandlungen:
  - meist automatisch (z.B. zwischen `NUMBER` und `int`; z.T. geht Genauigkeit verloren (bei `float`))
  - Strings: aus `char feld[n]` wird SQL-Typ `CHAR(n)` (Zeichenkette fester Länge) -> Auffüllen mit Leerzeichen notwendig
  - Strings: `VARCHAR2 str[n]` wird vom Precompiler übersetzt in
    ```
    struct{
    unsigned short len;
    unsigned char arr[n];
    } str;
    ```
    wird str durch Oracle mit String belegt, ist str nicht '\0'-terminiert!
  - Strings: alternativ
    `char s[n]; EXEC SQL VAR s IS STRING(n);`

Embedded SQL

Fehlerbehandlung:

- nach jeder SQL-Anweisung sollte Fehlerabfrage durchgeführt werden

- DB schreibt Return-Code in Variable `SQLSTATE`; Variable muß selbst deklariert werden:
  `char SQLSTATE[6];`

- alternativ: Variable `long SQLCODE` deklarieren; Wert 0 bedeutet 'fehlerfreie Ausführung', Wert 100 bedeutet 'keine weiteren Daten', negative Werte bedeuten Fehler

- weitere Infos zu Fehlern enthält Struktur namens SQL-Communications-Area; Deklaration mit `EXEC SQL INCLUDE SQLCA;`
  Variable heißt dann `sqlca`; in `sqlca.sqlcode` steht Return-Code entsprechend `SQLCODE`,
  in `sqlca.sqlerrm.sqlerrmc` Text der Fehlermeldung (70 Zeichen lang),
  in `sqlca.sqlerrd[2]` Anzahl Tupel, die von letzter SQL-Anweisung betroffen waren

Embedded SQL

Forts. Fehlerbehandlung:

- mit `WHENEVER`-Anweisung automatische Fehlerabfrage nach jeder SQL-Anweisung möglich:

```
EXEC SQL WHENEVER SQLERROR GOTO label;
```

- anstelle von GOTO alternativ: `CONTINUE`
- anstelle von SQLERROR alternativ: `NOT FOUND`
- Beispiel:

```
EXEC SQL WHENEVER SQLERROR GOTO fehler;
...
fehler: printf("\nFehler! %70s\n",sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK RELEASE;
exit(1);
```

- mehrere `WHENEVER`-Statements erlaubt; `CONTINUE` hebt dabei frühere `GOTO`-Anweisung auf

Embedded SQL

Anmelden bei der Datenbank:

- `EXEC SQL CONNECT benutzer IDENTIFIED BY passwort;`

  oder

  `EXEC SQL CONNECT benutzer/passwort;`

- ```
  VARCHAR userid[10];
  VARCHAR passwd[9];
  ....
  strcpy( (char *) userid.arr, "jutta");
  userid.len = strlen( (char *) userid.arr);
  strcpy( (char *) passwd.arr, "geheim!");
  passwd.len = strlen( (char *) passwd.arr);
  EXEC SQL CONNECT :userid IDENTIFIED BY :passwd;
  ```

- Connect-Anweisung entspricht nicht SQL92-Standard
  ```

Embedded SQL

Abmelden bei der Datenbank:

- bei erfolgreichem Programm: `EXEC SQL COMMIT RELEASE;`
- Durch Commit werden Transaktionen dauerhaft gemacht; durch Release werden für Anwendung reservierte Resourcen freigegeben, Verbindung zur Datenbank beendet
- im Fehlerfall: `EXEC SQL ROLLBACK RELEASE;`
- alle noch nicht mittels Commit dauerhaft gemachten Änderungen werden zurückgenommen, durch Release wie oben Abmeldung bei der Datenbank
- in SQL92: `DISCONNECT`-Kommando

Embedded SQL

Arbeiten in C mit SQL-Anfrageergebnissen:

Bei genau einem Ergebnistupel: mit `INTO` arbeiten

Beispiel:

```
VARCHAR bez;
...
EXEC SQL SELECT artname INTO :bez FROM artikel
WHERE artnr = 15;
if(sqlca.sqlcode == 0) {
bez.arr[bez.len]='\0';
printf("Bezeichnung zu Artnr. 15 ist %s\n",bez.arr);
}
..}
```

Embedded SQL

Arbeiten in C mit SQL-Anfrageergebnissen:

- Bei mehreren Ergebnistupeln: nacheinander in Schleife auslesen, mit Cursor
- `EXEC SQL DECLARE c1 CURSOR FOR sfw-Anfrage;`
- `EXEC SQL DECLARE c CURSOR`
  `SELECT artnr FROM artikel WHERE artname = :aname;`
- dann Cursor öffnen: `EXEC SQL OPEN c;`
  jetzt wird Anfrage ausgeführt mit aktuellem Wert von `aname`
- mit Fetch nächstes Tupel holen: `EXEC SQL FETCH c INTO :artikelnr;`
- zuletzt Cursor schließen: `EXEC SQL CLOSE c;`
- bei `UPDATE` und `DELETE` zuletzt mit Fetch geholtes Tupel ansprechbar: `EXEC SQL DELETE FROM artikel WHERE CURRENT OF c;`
  dazu bei Cursor-Deklaration `FOR UPDATE` hinzufügen

Embedded SQL

Nullwerte:

- Wenn Anfrageergebnis Nullwert liefern kann: neben Variable noch Indikator-Variable angeben
- Indikatorvariable ist -1, wenn Nullwert geliefert wurde, sonst 0
- Syntax: `variable INDICATOR ind-variable`
- Schlüsselwort INDICATOR kann auch weggelassen werden, Indikator-Variable zuvor als `short` deklarieren
- Beispiel:

```
short v1_ind, v2_ind;
...
EXEC SQL FETCH c1 INTO :liefname, :artikelnr,
:preis INDICATOR :v1_ind;
...
EXEC SQL FETCH c2 INTO :artikelnr, :bez :v2_ind;
```

Embedded SQL

- innerhalb von Embedded SQL: PL/SQL-Blöcke
- dazu:
  `EXEC SQL EXECUTE ...- - PL/SQL-Block END-EXEC;`
- Precopiler mit Option `SQLCHECK=SEMANTICS` aufrufen
- Relationen, die in PL/SQL-Block benutzt werden, mit `EXEC SQL DECLARE relname TABLE (attr1 Typ1,...);` bekannt machen
- Beispiel:

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

EXEC SQL BEGIN DECLARE SECTION;
VARCHAR userid[9];
VARCHAR passwd[9];
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE SQLCA;

int main(void)
{
char user[9];
char passwort[9];

printf("Ihr Login-Name?");
scanf("%s",user);
printf("\nIhr Passwort?");
scanf("%s",passwort);


EXEC SQL WHENEVER SQLERROR GOTO fehler;
EXEC SQL WHENEVER NOT FOUND GOTO ende;
```

```
strcpy( (char *) userid.arr, user);
userid.len = strlen( (char *) userid.arr);
strcpy( (char *) passwd.arr, passwort);
passwd.len = strlen( (char *) passwd.arr);
EXEC SQL CONNECT :userid IDENTIFIED BY :passwd;

printf("Alle Milch-Artikel: \n");
printf("ab jetzt in der Tabelle 'milchprod'\n");


EXEC SQL DECLARE artikel TABLE
(artnr number(4),
artname VARCHAR2(40)
);
EXEC SQL DECLARE milchprod TABLE (
nr number(4),
bez varchar2(40)
);
EXEC SQL EXECUTE
DECLARE
CURSOR c IS SELECT * FROM artikel WHERE artname LIKE '%milch%';
BEGIN
FOR rec IN c LOOP
INSERT INTO milchprod
VALUES (rec.artnr,rec.artname);
END LOOP;
END;
END-EXEC;

ende: EXEC SQL COMMIT RELEASE;
return 0;

fehler: printf("\nFehler! %70s\n",sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK RELEASE;
exit(1);
}
```

Embedded SQL, PL/SQL

Rekursion:

- durch rekursive PL/SQL-Funktion
- durch rekursives C-Programm mit Embedded SQL-Anweisungen
- durch 'erweitertes' Select-Statement mit `START WITH` und `CONNECT BY PRIOR`-Angaben
- Beispiel:

Embedded SQL, PL/SQL

```
#include<stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
VARCHAR userid[9];
VARCHAR passwd[9];
VARCHAR ch[40];
VARCHAR un[40];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE c CURSOR FOR
SELECT chef, angname INTO ch, un
FROM sn_ang_chef
START WITH chef='Bond'
CONNECT BY PRIOR angname=chef;

EXEC SQL INCLUDE SQLCA;

int main(void)
{...
printf("Alle Untergebenen: \n");
EXEC SQL OPEN c;
if(sqlca.sqlcode == 0) {
printf("Vorgesetzter: \t\t Untergebener:\n");
while(1){
EXEC SQL FETCH c INTO :ch, :un;
ch.arr[ch.len]='\0';
un.arr[un.len]='\0';
printf("%s\t\t %s\n",ch.arr,un.arr);
}
}
...
}
```

Embedded SQL

- bisher: statisch (SQL-Anweisung steht bei Precompiler-Lauf fest)

- auch möglich: dynamisch (zur Laufzeit wird Anfrage erst erzeugt)

- dazu

  1. SQL-Variable vom Typ statement vereinbaren:

     ```
     EXEC SQL DECLARE s1 STATEMENT;
     ```

  2. Statement-Variablen einen SQL-Anfragestring zuweisen:

     ```
     EXEC SQL PREPARE s1 FROM 'select * from liefert;'
     ```

  3. Anfrage ausführen:

     ```
     EXEC SQL EXECUTE s1 [USING :var1,...];
     ```

- Anfragestring kann Stringvariable sein (z.B. `:string1`)

- Statement kann Cursor zugeordnet und damit Ergebnisrelation durchlaufen werden (s. Bsp1)

- Anfragestring selbst kann Variablen enthalten, diese werden durch `USING ...` beim Aufruf mit akt. Variablenwerten aufgefüllt (s. Bsp2)

Embedded SQL

Beispiel1 (dynamisch)

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

EXEC SQL BEGIN DECLARE SECTION;
VARCHAR userid[9];
VARCHAR passwd[9];
VARCHAR bez[40];
int nummer;
char string1[256];

EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE s1 STATEMENT;

EXEC SQL DECLARE c CURSOR FOR
SELECT artnr, artname FROM artikel
WHERE artname like '%milch%';

EXEC SQL DECLARE c1 CURSOR FOR s1;

EXEC SQL INCLUDE SQLCA;

int main(void)
{
char user[9];
char passwort[9];
char u_bez[50];

printf("Ihr Login-Name?");
scanf("%s",user);
printf("\nIhr Passwort?");
scanf("%s",passwort);

EXEC SQL WHENEVER SQLERROR GOTO fehler;

strcpy( (char *) userid.arr, user);
userid.len = strlen( (char *) userid.arr);
strcpy( (char *) passwd.arr, passwort);
```

```
passwd.len = strlen( (char *) passwd.arr);
EXEC SQL CONNECT :userid IDENTIFIED BY :passwd;
printf("Welche Artikel (Wort in Art.bezeichnung) interessieren Sie?
");
scanf("%s",u_bez);

strcat(string1,"select artnr, artname from artikel where artname
like '%");
strcat(string1,u_bez);
strcat(string1,"%'");

printf("\n\nIhre Anfrage lautet:\n%s\n",string1);

EXEC SQL PREPARE s1 FROM :string1;
printf("\n\nAlle %s-Artikel: \n",u_bez);
EXEC SQL EXECUTE s1;
EXEC SQL OPEN c1;

EXEC SQL WHENEVER NOT FOUND GOTO next;

if(sqlca.sqlcode == 0) {
printf("Artikelnr:\t\tArtikelname\n");
while(1){
EXEC SQL FETCH c1 INTO :nummer, :bez;
bez.arr[bez.len]='\0';
printf("%-10d\t\t %s\n",nummer,bez.arr);
}
}

next: EXEC SQL CLOSE c1;
printf("\n\n");

EXEC SQL WHENEVER NOT FOUND GOTO ende;

printf("Alle Milch-Artikel: \n");
EXEC SQL OPEN c;
if(sqlca.sqlcode == 0) {
printf("Artikelnr:\t\tArtikelname\n");
while(1){
EXEC SQL FETCH c INTO :nummer, :bez;
bez.arr[bez.len]='\0';
printf("%-10d\t\t %s\n",nummer,bez.arr);
```

```
}
}
EXEC SQL CLOSE c;
ende: EXEC SQL COMMIT RELEASE;
return 0;

fehler: printf("\nFehler! %70s\n",sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK RELEASE;
exit(1);
}
```

Embedded SQL

Beispiel2 (dynamisch)

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

EXEC SQL BEGIN DECLARE SECTION;
VARCHAR userid[9];
VARCHAR passwd[9];
VARCHAR bez[40];
int nummer;
char string1[256];
int u_bez;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE s1 STATEMENT;
EXEC SQL DECLARE c1 CURSOR FOR s1;

EXEC SQL INCLUDE SQLCA;

int main(void)
{
char user[9];
char passwort[9];

.........

printf("Welcher Artikel (Artikelnr) interessiert Sie? ");
scanf("%d",&u_bez);

EXEC SQL PREPARE s1 FROM 'select artnr, artname from artikel where
artnr = :u_bez';

printf("\n\nArtikel: \n");
EXEC SQL EXECUTE s1 USING :u_bez;
```

```
EXEC SQL OPEN c1;

EXEC SQL WHENEVER NOT FOUND GOTO ende;

if(sqlca.sqlcode == 0) {
printf("Artikelnr:\t\tArtikelname\n");
while(1){
EXEC SQL FETCH c1 INTO :nummer, :bez;
bez.arr[bez.len]='\0';
printf("%-10d\t\t %s\n",nummer,bez.arr);
}
}

EXEC SQL CLOSE c1;
printf("\n\n");

ende: EXEC SQL COMMIT RELEASE;
return 0;

fehler: printf("\nFehler! %70s\n",sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK RELEASE;
exit(1);
}
```

DB-Anwendungsprogrammierung: Trigger

Allgemein:

- Trigger sind ECA-Regeln: Event-Condition-Action

- wenn auslösendes Event auftritt und Condition erfüllt ist, wird Action ausgeführt (ähnlich Prozeduren, die nicht explizit aufgerufen werden müssen

- Trigger nicht in SQL92 enthaelten, für SQL3 vorgesehen

- Trigger in Oracle enthalten, sehr unterschiedlich zum SQL3-Vorschlag

- Trigger heißen auch 'aktive Regeln', DB mit Triggern 'aktive DB'

- Anwendungen von Triggern: Integritätsüberwachungen, Aktualisierung abgeleiteter Daten, Protokollierung von Änderungen, Anstoßen von Geschäftsprozessen