

1.5 Client Programmierung (1)

Interaktive Eingabe von SQL

- DBS liefern SQL-Interpreter mit
· (Oracle: *sqlplus*, PostgreSQL: *psql*)
- Nicht praktikabel für Endanwender

Frage:

- Wie DB-Zugriffe automatisieren (programmieren)?
- *Serverseitige Programmierung*
Im DB-Server hinterlegt und von allen Anwendungen genutzt
- *Clientseitige Programmierung*
SQL-Kommandos werden aus Anwendungsprogramm aufgerufen

1

1.5 Client Programmierung (3)

Clientseitige Programmierung

- vorherrschende Form der DB-Programmierung
- Anwendungsprogramm führt nur
· *elementare SQL-Kommandos* durch
- Ablauflogik wird in anderer Sprache
· (*Host Language*, z.B. C++) programmiert

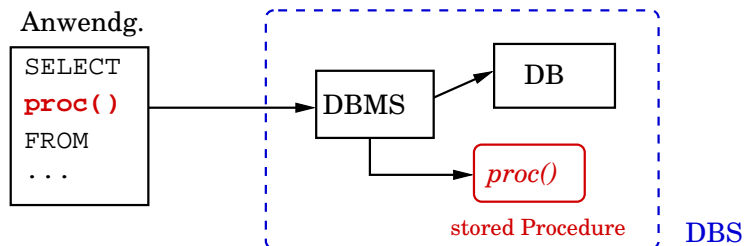
Frage:

- Wie können aus Host Language heraus
· SQL-Kommandos ausgeführt werden?

3

1.5 Client Programmierung (2)

Serverseitige Programmierung



- Hauptanwendungsgebiet: Trigger
- Vorteil: greift unabhängig davon, wie auf die
· Daten zugegriffen wird (z.B. auch SQL-Interpreter)

2

1.5 Client Programmierung (4)

SQL und Host Language

Methode	Beschreibung
SQL-Script	Batch Aufruf SQL-Interpreter. Keine Einbindung in Host Language.
embedded SQL (ESQL)	Mischen von SQL und Host Language. Präprozessor übersetzt <i>exec sql</i> Statements.
Call Level Interface (CLI)	Routinen in Host Language, SQL-Kommando ggf. als Parameter. <u>Natives CLI</u> : auf konkretes DBS zugeschnittene Bibliothek, z.B. <i>oci</i> (Oracle), <i>libpq</i> (Postgres) <u>Abstraktes CLI</u> : DBS-unabhängige abstrakte Bibliothek. Für konkretes DBS "Treiber" nötig. Beispiele: <i>odbc</i> , <i>bde</i> , <i>perl-dbi</i>

4

1.5 Client Programmierung (5)

Typische Einsatzgebiete

Schnittstelle	Einsatzgebiet
SQL-Script	Einfache Administrative Aufgaben, z.B. User anlegen, DB-Schema einspielen
ODBC	DB-unabhängige Massensoftware, z.B. Office-Pakete
Perl-DBI	<i>cron</i> -gesteuerte Serverprozesse, als CGI-Script in Web-Programmierung
ESQL native CLI's	Implementierung eigener abstrakter Interfaces, DBS-spezifische Tools, Individualsoftware, Programmierung Treiber für abstrakte CLI's

5

1.5.1 SQL-Script (2)

Alternative: "Fernsteuerung" des SQL-Interpreters durch Umlenkung von *stdin*

□ In der Shell:

```
#!/bin/sh
psql <<EOF
/* SQL-Kommandos */
EOF
```

□ Im C-Programm:

- unter Unix mit *popen()* (Einweg-Pipe)
oder *pipe()* + *fork()* + *dup2()* + *exec()* (Zweiwege-Pipe)
- unter Windows NT/2000/XP mit *CreatePipe()* + *DuplicateHandle()* + *CreateProcess()* + *CreateThread()*

7

1.5.1 SQL-Script (1)

SQL-Interpreter wie *psql* können nicht nur interaktiv verwendet werden:

- Ausführen einer externen Datei (z.B. *script.sql*)
 - mit SQL-Kommandos
 - innerhalb *psql*-Session mit Metakommando: `\i script.sql`
 - mit entsprechender Aufrufoption: `psql -f script.sql`
 - *script.sql* kann auch Metakommandos enthalten
- Übergabe eines SQL-Kommandos als Kommandozeilenparameter
 - Beispiel: `psql -c "truncate table produkt;"`

6

1.5.1 SQL-Script (3)

Nachteile

- keine Kontrollflusssteuerung (nur SQL)
- Fehlerbehandlung schwierig

Kann umgangen werden bei *psql -c*

- Kontrollfluss durch Shell-Befehle
- Abfragen des Exit-Codes möglich
- Aber: großer Overhead,
 - da pro SQL-Befehl ein Aufruf von *psql*

=> Einsatz begrenzt auf einfache Aufgaben

8

1.5.1 SQL-Script (4)

Beispiel: *dropuser* von PostgreSQL

```
#!/bin/sh

# (...)
# Commandline Parsing schaufelt
# zu löschenden User in Variable $DelUser

psql $PSQLOPT -d template1 -c "DROP USER $DelUser"

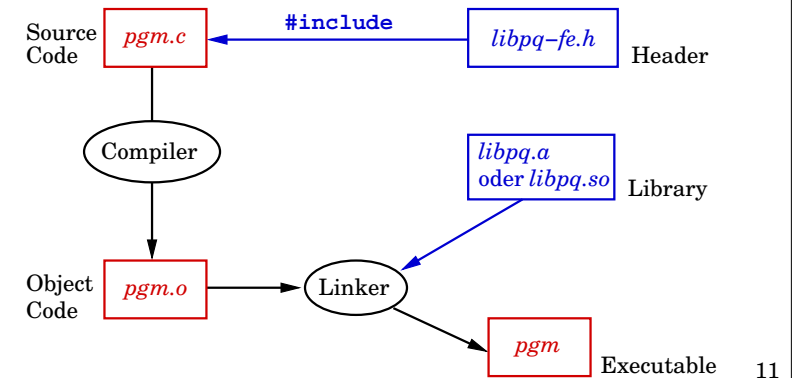
# Abfragen Exitcode
if [ "$?" -ne 0 ]; then
    echo "deletion of user \"$DelUser\" failed" 1>&2
    exit 1
fi

exit 0
```

9

1.5.2 natives CLI (2)

Erzeugung Clientprogramm:

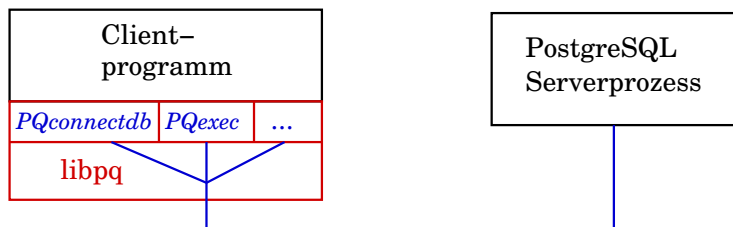


11

1.5.2 natives CLI (1)

Call Level Interface

Zugriff über vom DBS-Hersteller bereitgestellte Bibliotheksroutinen (Oracle: *oci*, Postgres: *libpq*)



10

1.5.2 natives CLI (3)

Konkrete Schritte der Programmierung:

- Source Code in Editor erstellen
Funktionsprototypen mit `#include <libpq-fe.h>` einbinden
- Zu Object Code compilieren
`gcc -c -I/usr/include/pgsql pgm.c`
"/usr/inlude/pgsql" ist Verzeichnis mit Postgres-Headern
- Mit libpq linken
`gcc -o pgm pgm.o -L/usr/lib/pgsql -lpq`
"/usr/lib/pgsql" ist Verzeichnis mit Postgres-Libraries

12

1.5.2 natives CLI (4)

Klassifikation *libpq*-Routinen:

- Verbindungsaufbau, -abbau
· *PQconnectdb()*, *PQfinish()*, *PQstatus()*
- Ausführen von SQL-Statements
· *PQexec()*, *PQresultStatus()*, *PQcmdTuples()*, *PQclear()*
- Verarbeiten von Abfrageergebnissen
· *PQntuples()*, *PQgetvalue()*, *PQgetlength()*

Wie die ANSI C *stdio*-Bibliothek ist *libpq* eine mit C-Mitteln realisierte objektorientierte Bibliothek

- Funktionsparameter zuvor konstruierte Strukturen
- Destruktoren müssen selbst aufgerufen werden

13

1.5.2 natives CLI (6)

Beispiel Verbindungsaufbau, -abbau

```
PGconn* conn;

/* Login */
conn = PQconnectdb("dbname=db user=usr ...");

/* Fehlerprüfung */
if (PQstatus(conn) == CONNECTION_BAD) {
    /* ... */
}

/* (...) */

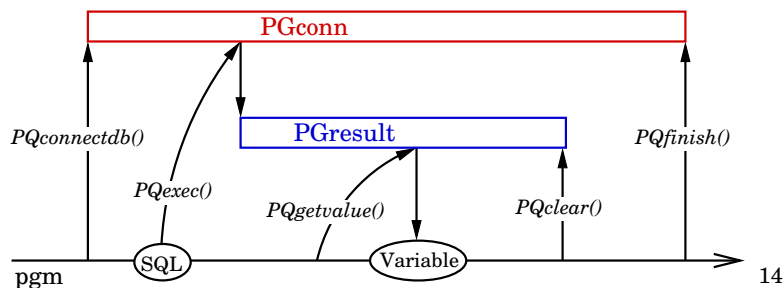
/* Logout */
PQfinish(conn);
```

15

1.5.2 natives CLI (5)

Hauptobjekte in *libpq*:

Datentyp	Konstruktor	Destruktor
PGconn	PQconnectdb()	PQfinish()
PGresult	PQexec()	PQclear()



14

1.5.2 natives CLI (7)

Beispiel Non-Select

```
PGconn *conn;
PGresult *res;

/* Absetzen SQL-Statement */
res = PQexec(conn,
    "DELETE FROM produkt WHERE preis>'3.0'");

if (PQresultStatus(res) == PGRES_COMMAND_OK) {
    /* Rückmeldung Auswirkungen */
    printf("%s Sätze gelöscht\n", PQcmdTuples(res));
}
else {
    /* Fehlerbehandlung */
}

/* Speicher freigeben nicht vergessen! */
PQclear(res);
```

16

1.5.2 natives CLI (8)

Beispiel Select

```
PGconn *conn;
PGresult *res;

/* Absetzen SQL-Statement */
res = PQexec(conn, "SELECT username FROM pg_user");

if (PQresultStatus(res) == PGRES_TUPLES_OK) {
    /* Ausgabe Ergebnisse */
    for (i = 0; i < PQntuples(res); i++)
        printf("%2d. %s\n", i+1, PQgetvalue(res,i,0));
}
else {
    /* Fehlerbehandlung */
}

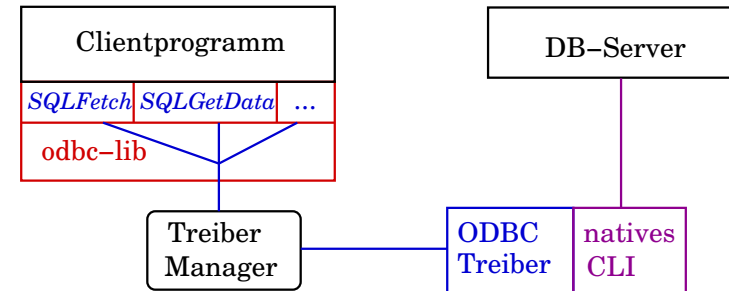
/* Speicher freigeben nicht vergessen! */
PQclear(res);
```

17

1.5.3 abstraktes CLI (1)

abstraktes Call Level Interface

- Zugriff über DBS-unabhängige Bibliotheksroutinen
- passender "Treiber" wird zur Laufzeit vom "Treiber-Manager" geladen



19

1.5.2 natives CLI (9)

weiterführende Literatur zu *libpq*:

- Hartwig: *PostgreSQL - Professionell und praxisnah*
· Kapitel 9.1 (Semesterapparat TWY Hart)
- PostgreSQL Programmer's Guide:
· *Client Interfaces - libpq*. Im PG-Paket enthalten.
· Online verfügbar unter
· <http://www.postgresql.org/docs/>

Hartwig beschreibt auch die C++ Bibliothek *libpq++*.
Achtung: Informationen sind veraltet!

18

1.5.3 abstraktes CLI (2)

Vorteile

- Programm läuft (im Prinzip) mit beliebigem DBS
Aber: ggf. abhängig vom SQL-Dialekt
- keine Bindung an konkretes DBS zur Compilezeit
=> geeignet für Massensoftware (z.B. Office-Pakete)
- kann auch *ohne* DBS verwendet werden:
z.B. gibt es Perl-DBI Treiber für Text Files

Nachteile

- kleinster gemeinsamer Nenner
- fortgeschrittene DBS-Features nicht nutzbar
- langsamer als direkt natives CLI
- erfordert Infrastruktur und Konfiguration
- im Einzelfall doch Fallunterscheidung DBS nötig
Beispiel: implizite Transaktionen in Oracle

20

1.5.3 abstraktes CLI (3)

Überblick

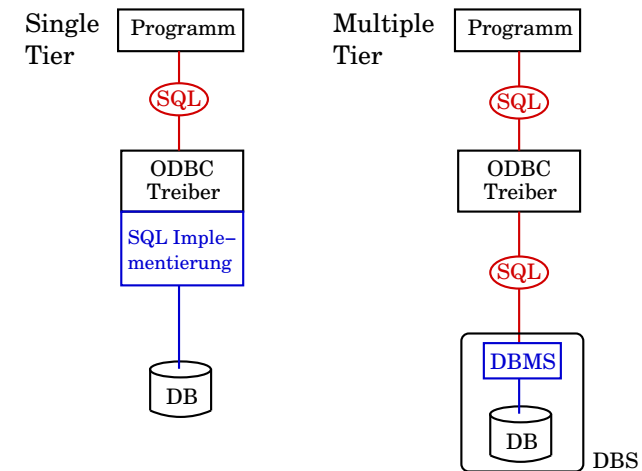
Abstraktion	Host Language	Hersteller
Open Database Connectivity (ODBC)	C Visual Basic	Microsoft offener Standard
Java Database Connectivity (JDBC)	Java	Sun offener Standard
Borland Database Engine (BDE)	Object Pascal C++	Borland
Perl Database Interface (DBI)	Perl	Tim Bunce offener Standard

Wir schauen uns konkret an:

- ODBC (prinzipieller Aufbau)
- Perl-DBI (Praktikumsaufgabe)

21

1.5.3 abstraktes CLI (5)



23

1.5.3 abstraktes CLI (4)

verbreitete ODBC-Irrtümer

- ODBC ist nur für Windows-Programme
 - Infrastruktur gibt es für Win32, Unix, MacOS, OS/2
 - ODBC-Treiber aber oft vom DBS-Hersteller nur für Win32 mitgeliefert => Treiber von Drittanbieter beziehen
- ODBC ist langsam
 - unzulässige Verallgemeinerung der Erfahrungen mit Access + VisualBasic (historisch erste ODBC-Umgebung)
 - nicht gültig für "Multiple-Tier" Treiber, da dabei die ODBC-Abstraktionsschicht nur geringer Overhead ist

Single Tier: Treiber implementiert SQL-Abfragen

Multiple Tier: Treiber reicht SQL an DBS weiter

22

1.5.3 abstraktes CLI (6)

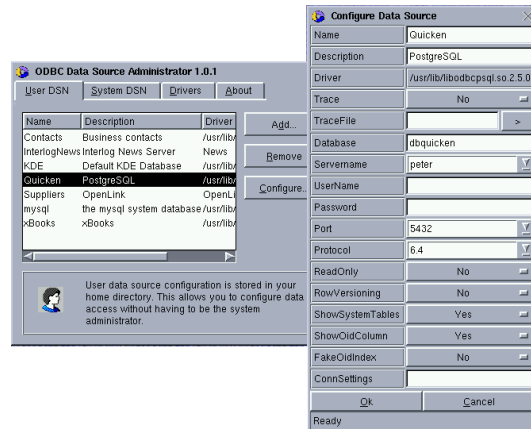
ODBC Data Sources

- Abstraktion Verbindungsparameter
 - zu verwendender ODBC-Treiber
 - Treiber-spezifische Parameter (z.B. *pgghost*, *pgdatabase*, ...)
- einem Satz Verbindungsparameter wird ein *Data Source Name (DSN)* zugewiesen
- statt Parameter beim Login zu übergeben, gibt das Anwendungsprogramm den DSN an
- Zuordnung DSN zu Parametern:
 - hinterlegt in Datei (Unix) oder Registry (Win32)
 - ODBC-Infrastruktur stellt Config-Tool(s) bereit

24

1.5.3 abstraktes CLI (7)

DSN-Konfiguration



25

1.5.3 abstraktes CLI (9)

Vereinfachtes Beispiel ODBC-Connection:

```
SQLHENV      sqlenv; /* Handle ODBC environment */
long         rc;      /* result of functions */
SQLHDBC      sqlconn; /* Handle connection */

/* 1. Allocate Environment Handle and register Version */
rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &sqlenv);
rc = SQLSetEnvAttr(sqlenv, SQL_ATTR_ODBC_VERSION, (void*)SQL_OV_ODBC3, 0);

/* 2. Allocate Connection Handle, set Timeout */
rc = SQLAllocHandle(SQL_HANDLE_DBC, sqlenv, &sqlconn);
SQLSetConnectAttr(sqlconn, SQL_LOGIN_TIMEOUT, (SQLPOINTER*)5, 0);

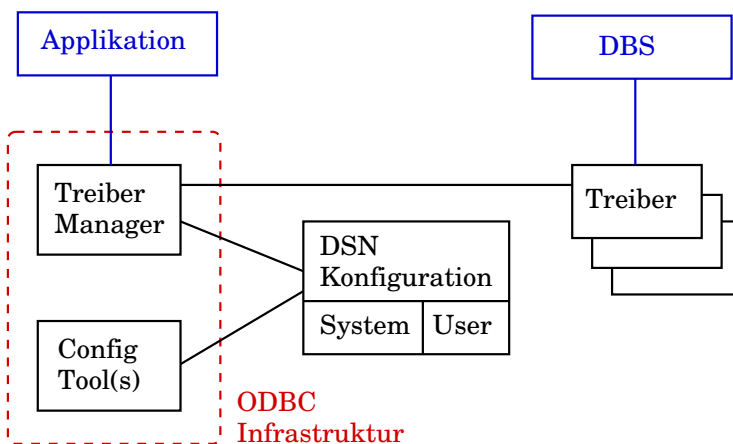
/* 3. Connect to the Datasource "web" */
rc = SQLConnect(sqlconn, (SQLCHAR*)"web", SQL_NTS,
                (SQLCHAR*)"christa", SQL_NTS, (SQLCHAR*)"", SQL_NTS);

/* Typical Errorcheck */
if ((rc != SQL_SUCCESS) && (rc != SQL_SUCCESS_WITH_INFO))
    { /* Error Handling */ }

/* Free Resources */
SQLFreeHandle(SQL_HANDLE_DBC, sqlconn);
SQLFreeHandle(SQL_HANDLE_ENV, sqlenv);
```

27

1.5.3 abstraktes CLI (8)



26

1.5.3 abstraktes CLI (10)

Weitergehende ODBC-Features:

- Anfragen an die Fähigkeiten des Treibers
- Anfragen an den System Catalog
- Anfragen über verfügbare Datenquellen und gesetzte Optionen
- ODBC verwendet eigenen SQL-Dialekt, der in SQL-Dialekt des DBS übersetzt wird
- kompliziertere SQL-Statements können an DBS "durchgereicht" werden (=> Abhängigkeit von SQL-Dialekt)

ODBC Referenzen:

- Kyle Geiger: *Inside ODBC*. Microsoft Press 1995
- ODBC Infrastruktur für Unix: <http://www.unixodbc.org/>
- Microsoft ODBC Seite: <http://www.microsoft.com/data/odbc/>

28

1.5.3 abstraktes CLI (11)

Was ist Perl?

- portable Scriptsprache von Larry Wall
- Mischung aus *C*, *awk* und *sh*
- Hauptsächlich *C* mit zahlreichen abkürzenden Notationen zur Emulation von *awk*
- => leicht zu schreiben, aber evtl. schwer zu lesen

Meinungen zu Perl:

- "Practical Extracting and Report Language"
- "Perl is awk with skin cancer."
- "Perl wird nicht mehr weiterentwickelt, weil alle Sonderzeichen aufgebraucht sind."

29

1.5.3 abstraktes CLI (13)

Perl "Crash-Kurs" später => Praktikumsaufgabe

An dieser Stelle nur Kurzüberblick:

- Kommentare von '#' bis Zeilenende
- Variablen haben als erstes Zeichen Typkennung:
 - \$bla - skalare Variable
 - @bla - Array Variable
 - %bla - Hash Variable (Array mit Key statt Index)
- keine weitergehende explizite Typunterscheidung (insbesondere kein *int*, *float*, *char*)
- ansonsten sehr C-ähnlich:
 - Kommandoabschluss mit ';' ;
 - Blockbildung mit '{ ... }'
 - Kontrollflusssteuerung mit *if*, *for*, *while*

31

1.5.3 abstraktes CLI (12)

Programme in *Scriptsprachen* können nicht direkt vom OS ausgeführt werden, sondern brauchen eine Laufzeitumgebung, den *Script-Interpreter*.

- Aufruf des Perlscripts *script*:
 - `perl script arg1 arg2 ...`
 - *arg1*, ... sind Kommandozeilenargumente für *script*
- Unter Unix (oder bei Aufruf aus der Cygwin-Shell) kann *script* direkt ausführbar gemacht werden:
 - als erste Zeile des Scripts einfügen: `#!/usr/bin/perl`
 - Script ausführbar machen mit: `chmod +x script`

Methode geht mit vielen Scriptsprachen (*sh*, *perl*, *wish*)

30

1.5.3 abstraktes CLI (14)

Beispiel für Perl Code:

```
# erstelle %list mit Usern und Passwörtern
srand();
for ($nr = 1; $nr <= 35; $nr++)
{
    # Username: Prefix "dbs" + laufende Nr
    $uid = sprintf("dbs%02d", $nr);
    # zufällige Generierung Passwortzeichen
    $pwd = "";
    for ($i = 0; $i < 8; $i++) {
        $pwd .= chr(int(rand 42) + 48);
    }
    # Zuordnung User => Passwort
    $list{$uid} = $pwd;
}
```

32

1.5.3 abstraktes CLI (15)

Perl DBI

Wie bei ODBC braucht der DB-Zugriff mit dem *Perl Database Interface* (DBI) zwei Komponenten:

- Die abstrakte DBI-Bibliothek wird als Modul eingebunden mit

```
use DBI;
```
- Der Database Driver (DBD) wird dynamisch beim Aufruf der *connect* Methode geladen

```
DBI->connect("dbi:Pg:", "uid", "pwd");
DBI->connect("dbi:Oracle:", "uid", "pwd");
```

33

1.5.3 abstraktes CLI (17)

Verbindungsaufbau

```
$dsn = "dbi:Pg:dbname=test;host=dbs;port=5432";
%attr = (AutoCommit => 0,
        PrintError => 0, RaiseError => 0);

$dbh = DBI->connect($dsn, "uid", "pwd", \%attr);
if (!$dbh) {
    # Fehlerbehandlung
}
```

- Parameter *\$dsn* entspricht "Data Source Name" bei ODBC: enthält Treiber + (optional) treiberspezifische Optionen
- DBI-Attribute *%attr* können auch direkt angegeben werden:

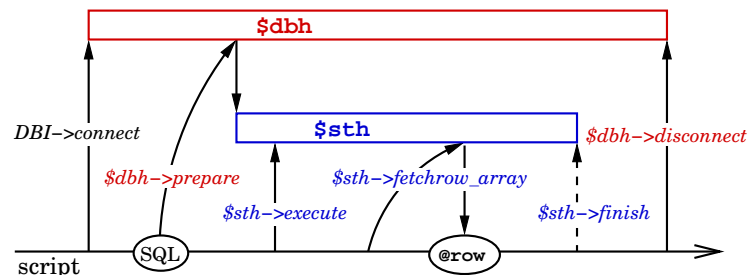
```
DBI->connect(..., {AutoCommit => 0, ...})
```

35

1.5.3 abstraktes CLI (16)

DBI arbeitet als objektorientierte Bibliothek ähnlich wie *libpq* mit zwei Objekten:

Objekt	Bedeutung
<i>\$dbh</i>	Database handle object
<i>\$sth</i>	Statement handle object



34

1.5.3 abstraktes CLI (18)

Wichtige DBI-Attribute

Attribut	Bedeutung
AutoCommit	Nach jedem Statement automatisch Commit (0 = off, 1 = on (default))
PrintError	Fehler werden automatisch nach <i>stderr</i> ausgegeben (0 = off, 1 = on (default))
RaiseError	Bei Fehler wird Exception (vgl. C++) geworfen (0 = off (default), 1 = on)

Attribute werden als Hash-Referenz im vierten Argument bei *DBI->connect* übergeben:

```
DBI->connect($dsn, $uid, $pwd,
            {AutoCommit => 0, PrintError => 0});
```

36

1.5.3 abstraktes CLI (19)

Beispiel Non-Select

```
$sth = $dbh->prepare("delete from produkt");

if (!$sth->execute) {
    # Fehler
    printf ("%s\n", $dbh->errstr);
}
else {
    # Rückmeldung über Erfolg
    printf ("%d Zeilen gelöscht\n", $sth->rows);
}
```

Bemerkung:

Für Non-Select Statements gibt es auch die Methode `$dbh->do()` (Zusammenfassung von `prepare` + `execute`)

37

1.5.3 abstraktes CLI (21)

Referenzen

□ Online Kurse Perl:

- http://www.phy.uni-bayreuth.de/~btpa25/perl/perl_inhalt.html
- <http://www.pronix.de/perl/perl.html>

□ Offizielle Perl-DBI Dokumentation:

- <http://www.perldoc.com/cpan/DBI.html>
- Die beim DBI mitgelieferte Dokumentation kann mit dem Kommando `perldoc DBI` wie eine Manpage gelesen werden

Beispielprogramm *bsp-perldb* auf Homepage

39

1.5.3 abstraktes CLI (20)

Beispiel Select

```
$sth = $dbh->prepare("select * from produkt");

if (!$sth->execute) {
    # Fehler
    printf ("%s\n", $dbh->errstr);
}
else {
    # Verarbeitung der Ergebniszeilen
    while (@row = $sth->fetchrow_array) {
        # ...
    }
}

# meist optional:
$sth->finish;
```

38

1.5.4 embedded SQL (1)

Historie

- historisch erste Möglichkeit, aus Programm SQL-Kommandos abzusetzen
- 1992 in SQL2-Standard aufgenommen
- wird von den meisten DBS unterstützt (Oracle: *Pro*C*, PostgreSQL: *ecpg*)
- => ESQL Source Code leicht portabel

Konzept

- C-Code und SQL-Code werden gemischt
- SQL-Code wird durch `exec sql` kenntlich gemacht
- SQL-Code wird von Präprozessor in C-Code übersetzt

40

1.5.4 embedded SQL (2)

Ein Beispiel:

```
EXEC SQL BEGIN DECLARE SECTION;
int res;
EXEC SQL END DECLARE SECTION;

int main()
{
    EXEC SQL CONNECT TO dbname USER uid/passwd;

    EXEC SQL SELECT COUNT(*) INTO :res FROM pg_user;

    printf("Anzahl User: %d\n", res);

    return 0;
}
```

41

1.5.4 embedded SQL (4)

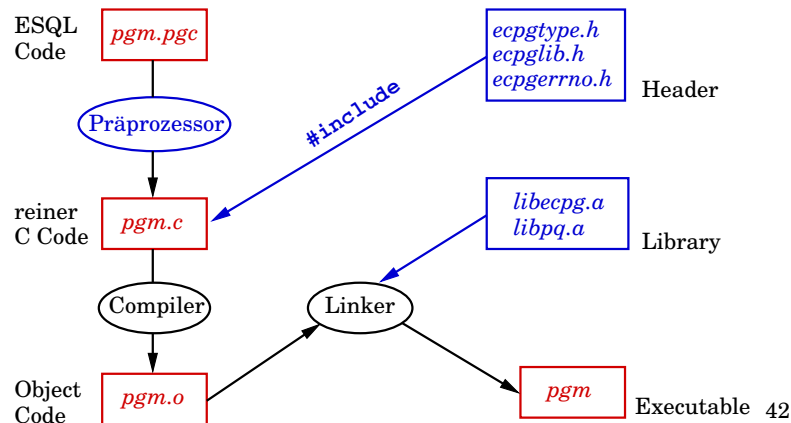
Konkrete Schritte der Programmierung:

- ESQL Source Code in Editor erstellen
Dateinamenskvention: *.pgc
- ESQL-Präprozessor aufrufen
ecpg pgm.pgc (erzeugt pgm.c)
- Zu Object Code compilieren
gcc -c -I/usr/include/postgresql pgm.c
"/usr/include/postgresql" ist Verzeichnis mit Postgres-Headern
- Mit *libecpg* und *libpq* linken
gcc -o pgm pgm.o -L/usr/lib/postgresql -lecpg -lpq
"/usr/lib/postgresql" ist Verzeichnis mit Postgres-Libraries

43

1.5.4 embedded SQL (3)

Übersetzen ESQL-Programm:



42

1.5.4 embedded SQL (5)

Probleme bei "Zwittercode"

Problem	Lösung
Variablen austausch	Definition von gemeinsamen Variablen in <i>exec sql declare</i> Abschnitt
Laufzeit-Generierung von SQL-Statements	Übernahme von Statements aus Variablen mit <i>exec sql prepare</i> ("dynamic SQL")
Navigation in Select-Ergebnissen (Tabellen)	SQL-Erweiterung: <i>Cursor</i>
Fehlerkommunikation	globale Variable (structure) <i>sqlca</i> ("SQL Communication Area")

44

1.5.4 embedded SQL (6)

gemeinsame Variablen

- Shared Variables werden als C Variablen in der *Declare Section* deklariert

```
EXEC SQL BEGIN DECLARE SECTION;
int nr;
char name[30];
EXEC SQL END DECLARE SECTION;
```

- Verwendung in SQL mit vorangestelltem Doppelpunkt

```
EXEC SQL DELETE FROM produkt WHERE nr=:nr;
```

45

1.5.4 embedded SQL (8)

dynamic SQL

Shared Variables können auch SQL-Kommandos enthalten. Zum Ausführen zwei Schritte nötig:

- Definition SQL Statement-Variable mit *prepare*

```
strcpy(stmt, "delete from produkt");
EXEC SQL PREPARE sqlstmt FROM :stmt;
```

- Ausführen der Statements mit *execute*

```
EXEC SQL EXECUTE sqlstmt;
```

47

1.5.4 embedded SQL (7)

C- und SQL-Datentyp müssen kompatibel sein

SQL-Typ	C-Typ
INTEGER	int, char[]
NUMERIC	double, char[]
CHAR(<i>n</i>), VARCHAR(<i>n</i>)	char[<i>n</i> +1]
DATE	char[12]
TIMESTAMP	char[28]

Bemerkungen:

- C-Typ *char*-Array ist immer möglich
- wenn SQL-String länger als C-String, wird abgeschnitten (aber Achtung: ggf. fehlt dann Abschluss-Null)
- Datumstypen vorzugsweise mit *to_char*, *to_date* formatieren

46

1.5.4 embedded SQL (9)

Select-Ergebnisse

Da Select-Statements im allg. *Tabellen* liefern, sind zwei Fälle bei der Auswertung zu unterscheiden:

- Ergebnis kann nur aus *einer Zeile* bestehen
⇒ Spaltenwerte können direkt in Shared Variables übergeben werden mit *select into*
- Ergebnis kann *mehrere Zeilen* enthalten
⇒ einzelne Zeilen müssen mit einem *Cursor* abgearbeitet werden

48

1.5.4 embedded SQL (10)

Single-Row Select

```
/* Deklaration Shared Variables */
EXEC SQL BEGIN DECLARE SECTION;
double preis;
char einf[12];
EXEC SQL END DECLARE SECTION;

/* Füllen Shared Variables in SQL */
EXEC SQL
  SELECT preis, to_char(einfuehrung, 'DD.MM.YY')
  INTO :preis, :einf
  FROM produkt WHERE pnr = 'P1';

/* Benutzung Shared Variables in C */
printf("preis: %0.2f; einf: %s\n", preis, einf);
```

49

1.5.4 embedded SQL (12)

Tabellen-Retrieval (2)

```
EXEC SQL BEGIN DECLARE SECTION;
char name[256];
EXEC SQL END DECLARE SECTION;

/* Cursor Definition */
EXEC SQL DECLARE cursor1 CURSOR FOR
  SELECT username FROM pg_user;
EXEC SQL OPEN cursor1;

/* Loop über Ergebnisse */
EXEC SQL WHENEVER NOT FOUND DO break;
while (1) {
  EXEC SQL FETCH cursor1 INTO :name;
  printf("%s\n", name);
}

/* Cursor Schliessen */
EXEC SQL CLOSE cursor1;
```

51

1.5.4 embedded SQL (11)

Tabellen-Retrieval (1)

- Definition eines Cursors mit *declare cursor* und Verknüpfung mit Tabellenausdruck
 - Tabellenausdruck muss Select-Statement oder entspr. SQL Statement-Variable sein
 - bei Cursordefinition wird Statement noch nicht ausgeführt
- Ausführen des Statements und Positionierung des Cursors vor die erste Zeile mit *open*
- Auslesen der Zeilen mit *fetch* in einer Schleife
- Schließen des Cursors mit *close*

50

1.5.4 embedded SQL (13)

Fehlerbehandlung

Kommunikation über Erfolg der SQL-Statements über globale Variable *sqlca* (SQL Communication Area)

- Einbindung mit `EXEC SQL INCLUDE sqlca;`
- In C Struktur. Wichtigste Felder:

sqlca.sqlcode (alternativ Makro: SQLCODE)	Fehlercode letztes Statement
sqlca.sqlerrm.sqlerrmc	Fehlermeldung vom DBS

Achtung: Struktur von *sqlca* in SQL2 nicht festgelegt
Nur Werte für das Makro *SQLCODE* sind festgelegt

52

1.5.4 embedded SQL (14)

Werte für *sqlca.sqlcode* bzw. *SQLCODE*:

Wert	Bedeutung
0	kein Fehler
100	keine Datensätze gefunden
< 0	Fehler bei Ausführung Statement konkrete Codes DBS-spezifisch
> 0	Warnung (z.B. String zu klein)
≠ 100	konkrete Codes DBS-spezifisch

Bemerkung

- Fehlerabfrage in SQL2 vereinfacht mit *whenever*
EXEC SQL WHENEVER bedingung aktion;
- Mögliche *bedingung* NOT FOUND oder SQLERROR
- Fügt Abfrage hinter jedes Statement ein und führt ggf. *aktion* (z.B. DO break) aus

53

1.5.4 embedded SQL (16)

Verbindungsaufbau und -abbau

DB-Login ist in SQL nicht vorgesehen
=> spezielles ESQL-Statement erforderlich

- EXEC SQL CONNECT TO db [USER uid[/pwd]];
EXEC SQL DISCONNECT;
- Parameter *db* kann DBS-spezifische Erweiterungen unterstützen (z.B. Postgres: *dbname[host]*)
- Fehlerabfrage über *SQLCODE* bzw. *sqlca.sqlcode*

55

1.5.4 embedded SQL (15)

Beispiel Fehlerbehandlung

```
EXEC SQL DECLARE cursor1 CURSOR FOR
  SELECT username FROM pg_user;
EXEC SQL OPEN cursor1;

if (sqlca.sqlcode) {
  /* Fehler aufgetreten */
  printf("%s\n", sqlca.sqlerrm.sqlerrmc);
}
else {
  /* Loop über Ergebnisse */
  EXEC SQL WHENEVER NOT FOUND DO break;
  while (1) {
    EXEC SQL FETCH cursor1 INTO :name;
    printf("%s\n", name);
  }
}
EXEC SQL CLOSE cursor1;
```

54

1.5.5 weitere Ansätze (1)

Webanwendungen

- normale Client-Server Anwendungen sind *session-orientiert*:
Login -> umfangreiche Verarbeitung -> Logout
- Webanwendungen sind *session-los*:
einzelne Seiten werden ohne weitere Verpflichtung angefordert

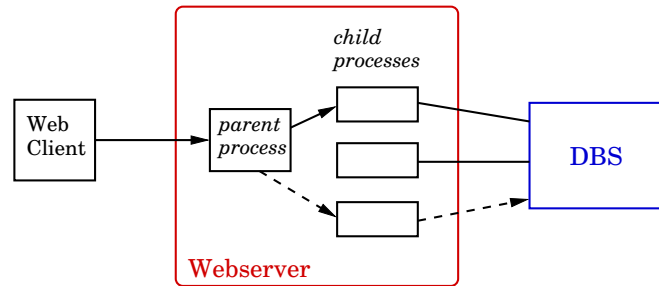
=> für jeden Seitenaufbau Login großer Overhead

- Lösung: *Persistent Database Connections*

56

1.5.5 weitere Ansätze (2)

Persistent Database Connections



Voraussetzung: Verarbeitung nicht über CGI sondern Script-Modul

57

1.5.5 weitere Ansätze (4)

Fourth Generation Language (4GL)

- proprietäre Sprache des DBS-Herstellers, die
· SQL und GUI-Programmierkonzepte vereinigt
· (Beispiele: *Informix-4GL*, *Sybase PowerBuilder*)
- Bezeichnung ist Marketing-Gag durch Gleichsetzung
· Maschinencode=1GL, Assembler=2GL, C etc.=3GL

Weiterführung dieser Idee sind spezielle *Application Development Frameworks* einzelner Hersteller

Vorteil: Customizing statt Programmierung
· => niedrige Entwicklungskosten

59

1.5.5 weitere Ansätze (3)

Application Server

Anwendungen wollen eigentlich gar keinen Datenbank-Zugriff sondern spezielle Funktionen

Idee:

- entwerfe Protokoll für diese Funktionen
- implementiere dieses Protokoll in eigenem Client-Server Modell
- Clients greifen über dieses Protokoll auf Application Server zu
- Application Server implementiert DB-Zugriffe

Vorteile:

- Datenbank-Logik zentral in Application Server
- einfachere Client-Programmierung

58