1.5 Client Programmierung

Interaktive Eingabe von SQL

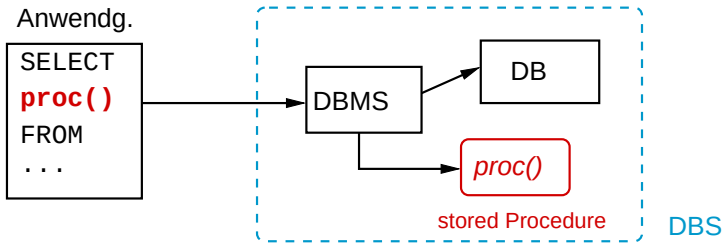
- DBS liefern SQL-Interpreter mit
  (Oracle: *sqlplus*, PostgreSQL: *psql*)
- Nicht praktikabel für Endanwender

Automatisierung der DB-Zugriffe

- *Serverseitige* Programmierung
  Im DB-Server hinterlegt und von allen Anwendungen genutzt
- *Clientseitige* Programmierung
  SQL-Kommandos werden aus Anwendungsprogramm aufgerufen

1.5 Client Programmierung

Serverseitige Programmierung



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

1.5 Client Programmierung

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?

1.5 Client Programmierung

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 *exec sql* Statements.
Call Level Interface (CLI)	Routinen in Host Language, SQL-Kommando ggf. als Parameter. Natives CLI: auf konkretes DBS zugeschnittene Bibliothek, z.B. *oci* (Oracle), *libpq* (Postgres) Abstraktes CLI: DBS-unabhängige abstrakte Bibliothek. Für konkretes DBS "Treiber" nötig. Beipiele: *odbc, bde, perl-dbi*

1.5 Client Programmierung

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	*cron*-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

1.5.1 SQL-Script

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;"*

1.5.1 SQL-Script

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 mit *CreatePipe()* + *DuplicateHandle()* + *CreateProcess()* + *CreateThread()*

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

1.5.1 SQL-Script

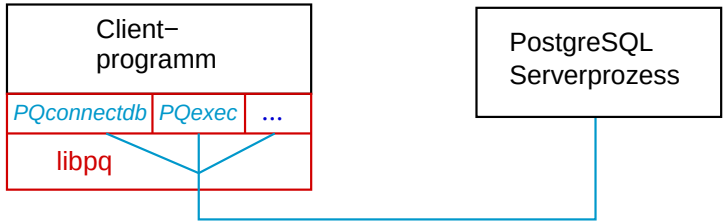
Beispiel: *dropuser* von PostgreSQL

```
#!/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
```


1.5.2 natives CLI

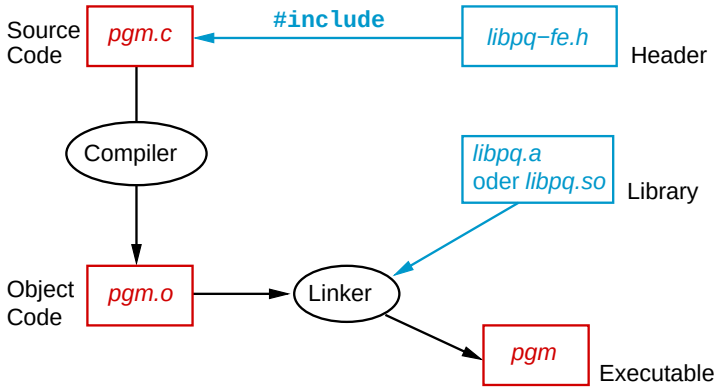
Call Level Interface

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



1.5.2 natives CLI

Erzeugung Clientprogramm:



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

1.5.2 natives CLI

Klassifikation *libpq*-Routinen:

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

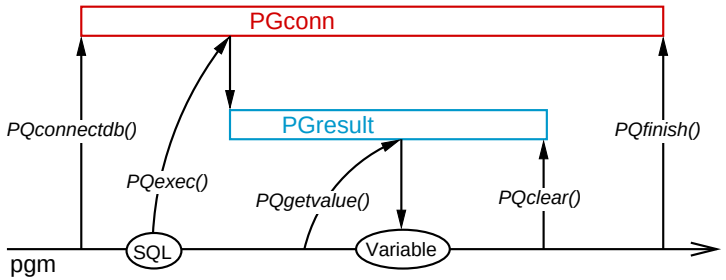
Wie die ANSI C *stdio*-Bibliothek ist *libpq* eine mit C-Mittlen realisierte objektorientierte Bibliothek

- Funktionsparameter zuvor konstruierte Strukturen
- Destruktoren müssen selbst aufgerufen werden

1.5.2 natives CLI (5)

Hauptobjekte in *libpq*:

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



1.5.2 natives CLI

Beispiel Verbindungsaufbau, -abbau

```
PGconn* conn;

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

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

/* (...) */

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


1.5.2 natives CLI

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);
```

1.5.2 natives CLI

Beispiel Select

```
PGconn *conn;
PGresult *res;

/* Absetzen SQL-Statement */
res = PQexec(conn, "SELECT usename 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);
```


1.5.2 natives CLI

weiterführende Literatur zu *libpq*:

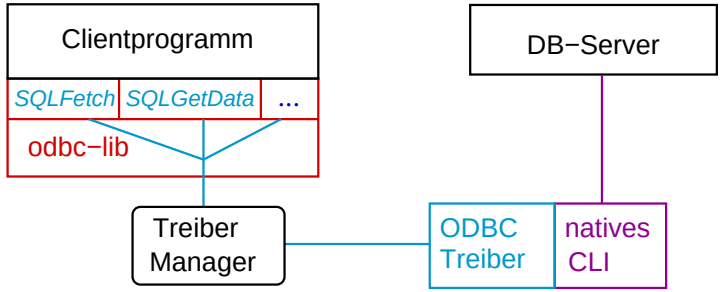
- Hartwig: *PostgreSQL - Professionell und praxisnah.* Kapitel 9.1 (SemesterapparatTWY Hart)
- PostgreSQL Programmer's Guide: *Client Interfaces - libpq.* Im PG-Paket enthalten. Online verfügbar unter *http://www.postgresql.org/docs/*

Hartwig beschreibt auch die C++ Bibliothek *libpq++*. Achtung: diese Schnittstelle ist veraltet!

1.5.3 abstraktes CLI

Abstraktes Call Level Interface

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



1.5.3 abstraktes CLI

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

1.5.3 abstraktes CLI (3)

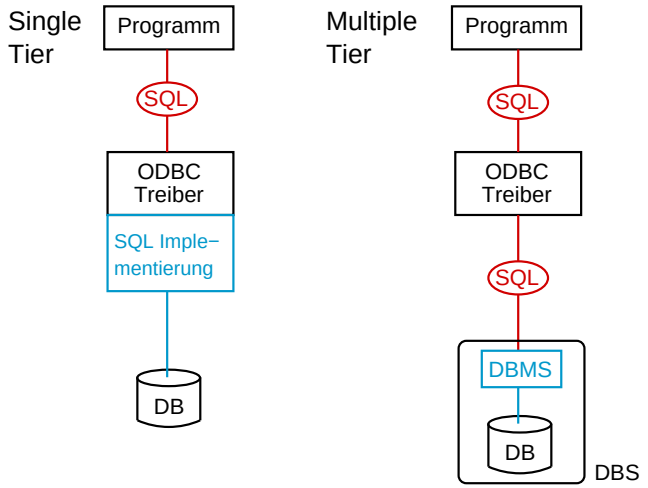
Überblick

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 (nur in schriftl. Unterlagen)

1.5.3 abstraktes CLI



1.5.3 abstraktes CLI

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 Windows 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

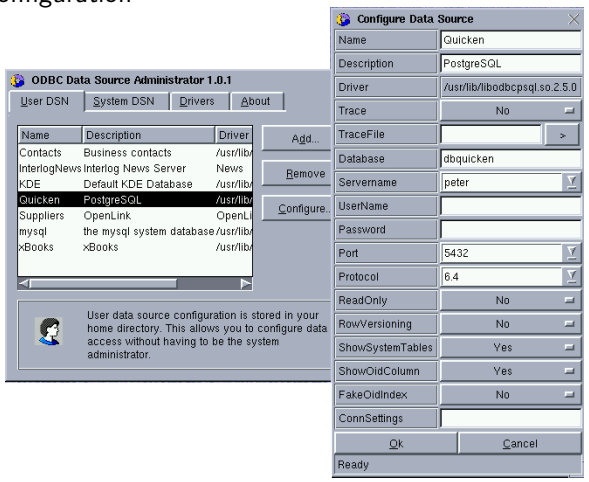
1.5.3 abstraktes CLI

ODBC Data Sources

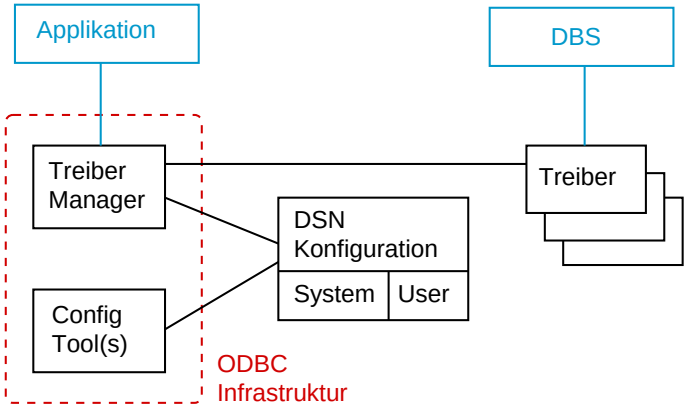
- Abstraktion Verbindungsparameter
  - zu verwendender ODBC-Treiber
  - Treiber-spezifische Parameter (z.B. *pghost*, *pgdatabase*, ...)
- einem Satz Verbindungsparameter wird ein *Data Source Name* (DSN) zugewiesen
- statt Parameter beim Login zu übergeben, gibt das Anwendungsprogramm den DSN an
- Zuordnung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

1.5.3 abstraktes CLI

DSN-Konfiguration



1.5.3 abstraktes CLI



1.5.3 abstraktes CLI

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);
```


1.5.3 abstraktes CLI

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/*

1.5.3 abstraktes CLI

Was ist Perl?

- portable Scriptsprache von Larry Wall
- Mischung aus *C*, *awk* und *sh*
- Hauptsächlich *C* mit zahlreichen abkürzenden Notationen zur Emulation von *awk*
  $\Rightarrow$ 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.“

1.5.3 abstraktes CLI

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 +xscript`

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

1.5.3 abstraktes CLI

Perl „Crash-Kurs“ separate Folien 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*

1.5.3 abstraktes CLI

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;
}
```

1.5.3 abstraktes CLI

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;
```

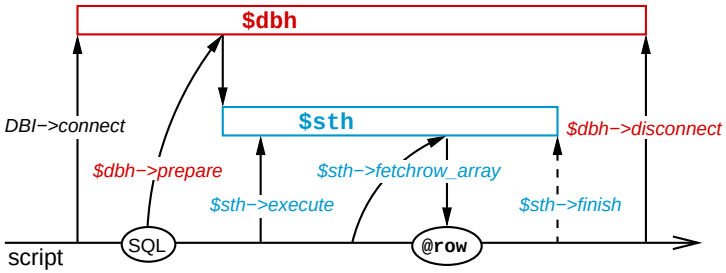
- Der Database Driver (DBD) wird dynamisch beim Aufruf der *connect* Methode geladen

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


1.5.3 abstraktes CLI

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

Objekt	Bedeutung
`$dbh`	Database handle object
`$sth`	Statement handle object



1.5.3 abstraktes CLI

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, ...})`

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 *stderr* 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});
```

1.5.3 abstraktes CLI

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*)

1.5.3 abstraktes CLI

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;
```


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-perldbi* auf Veranstaltungs-Webseite

1.5.4 embedded SQL

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

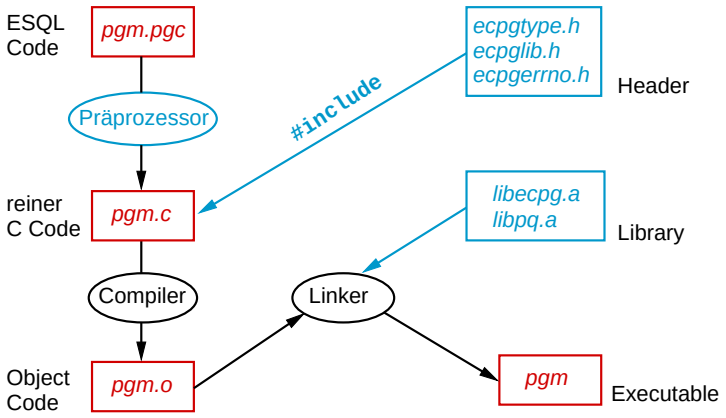
1.5.4 embedded SQL

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;
}
```


1.5.4 embedded SQL

Übersetzen ESQL-Programm:



1.5.4 embedded SQL

Konkrete Schritte der Programmierung:

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

1.5.4 embedded SQL (5)

Probleme bei „Zwittercode“

Problem	**Lösung**
Variablenaustausch	Definiton von gemeinsamen Variablen in *exec sql declare* Abschnitt
Laufzeit-Generierung von SQL-Statements	Übernahme von Statements aus Variablen mit *exec sql prepare* (“dynamic SQL”)
Navigation in Select-Ergebnissen (Tabellen)	SQL-Erweiterung: *Cursor*
Fehlerkommunikation	globale Variable (structure) *sqlca* (“SQL Communication Area”)

1.5.4 embedded SQL

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;
```


1.5.4 embedded SQL

C- und SQL-Datentyp müssen kompatibel sein

SQL-Typ	C-Typ
INTEGER	int, char[]
NUMERIC	double, char[]
CHAR(*n*), VARCHAR(*n*)	char[*n*+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

1.5.4 embedded SQL

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;
```


1.5.4 embedded SQL

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

1.5.4 embedded SQL

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);
```


1.5.4 embedded SQL

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*

1.5.4 embedded SQL

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 usename 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;
```


1.5.4 embedded SQL

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`) `sqlca.sqlstate` (alternativ Makro: `SQLSTATE`)	Fehlercode letztes Statement
`sqlca.sqlerrm.sqlerrmc`	Fehlermeldung vom DBS

Achtung: Struktur von *sqlca* in SQL2 nicht festgelegt
Nur Werte für das Makro *SQLCODE* (SQL-92)
bzw. *SQLSTATE* (seit SQL-99) sind festgelegt

1.5.4 embedded SQL

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 $\neq$ 100	Warnung (z.B. String zu klein) 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

1.5.4 embedded SQL

sqlca.sqlstate bzw. *SQLSTATE* hierarchisch aufgebaut:

- insgesamt 5 Characters
- erste zwei Stellen Fehlerklasse
- Stellen 3 bis 5 Fehler-Subklasse

Wichtige Fehlerklassen:

Wert	Bedeutung
00	no error
01	warning
02	no data
> 02	echte Fehler

Achtung: Code muss nicht nur aus Ziffern bestehen, z.B. `XX002` für „Index corrupted“

1.5.4 embedded SQL

Beispiel Fehlerbehandlung

```
EXEC SQL DECLARE cursor1 CURSOR FOR
  SELECT usename 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;
```


1.5.4 embedded SQL

Verbindungsaufbau und -abbau

DB-Login ist in SQL nicht vorgesehen
$\Rightarrow$ 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*

1.5.5 weitere Ansätze

Webanwendungen

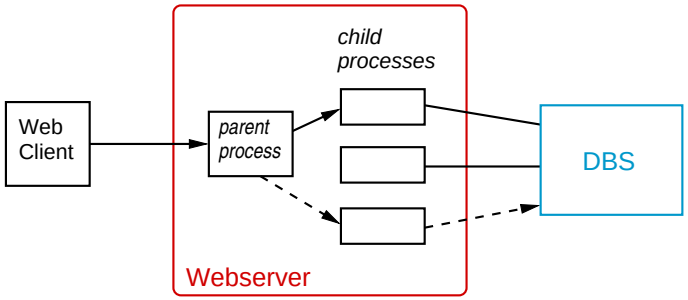
- normale Client-Server Anwendungen sind *session-orientiert*:
  Login → umfangreiche Verarbeitung → Logout
- Webanwendungen sind *session-los*:
  einzelne Seiten werden ohne weitere Verpflichtung angefordert

⇒ für jeden Seitenaufbau Login großer Overhead

Lösung: *Persistent Database Connections*

1.5.5 weitere Ansätze

Persistent Database Connections



Voraussetzung: Verarbeitung nicht über CGI sondern Script-Modul
Allgemeine Lösung (unter GNU GPL): *http://sqlrelay.sf.net/*

1.5.5 weitere Ansätze

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

1.5.5 weitere Ansätze

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
  Machinencode=1GL, Assembler=2GL, C etc.=3GL

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

Vorteil: Customizing statt Programmierung
⇒ niedrige Entwicklungskosten