

Programmieren in C++ (9): Objektorientierte C++-Schnittstelle einer Bibliothek, Msql++

Silke Seehusen, Fachhochschule Lübeck

Es gibt viele Bibliotheken, die eine C-Schnittstelle zur Verfügung stellen. Nur zu relativ wenigen Bibliotheken gibt es C++-Schnittstellen. An einem Beispiel, der API zu SQL, wird gezeigt, wie eine solche Schnittstelle entwickelt werden kann, und es werden insbesondere die Vor- und Nachteile aufgezeigt.

C-Schnittstellen

Viele Programmpakete kommen mit einer C-Schnittstelle in Form einer Menge von Funktionen, Typen und Konstanten. Als Beispiele seien hier mathematische Bibliotheken und die *X11*-Bibliothek erwähnt, aber auch kleinere Pakete wie *Tcl/Tk*, *Jpeg*, Schnittstellen zu Datenbanken und fensterorientierten Benutzungsschnittstellen.

C++-Schnittstellen zu solchen Standardpaketen sind seltener, da aus C++ auf die C-Schnittstelle direkt zugegriffen werden kann. Sogar die Standard-C++-Bibliothek stellt einige Teile der Standard-ANSI-C-Bibliothek wie mathematische Funktionen, Funktionen für Zeichenketten, variable Argumentlisten, Datum und Uhrzeit direkt zur Verfügung, allerdings unter anderen Include-Namen, z.B. `<cstdlibarg>` statt `<stdarg.h>`.

Ein Entwurfsziel von C++ ist, die Entwicklung größerer Programmpakete zu unterstützen. Dazu gehören auch Bibliotheken.

Wir wollen im folgenden an einem Beispiel untersuchen, welche Vor- und Nachteile eine C++-Schnittstelle einer Bibliothek haben kann.

Mini SQL

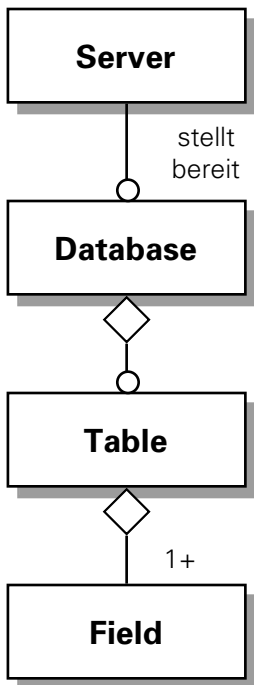
Als Beispiel wird eine C++-Schnittstelle für Mini SQL entwickelt, die auf der C-Schnittstelle aufbaut.

Mini SQL, auch mSQL genannt, ist eine sogenannte leichtgewichtige Datenbankmaschine [1]. Sie stellt als Datenbanksprache eine echte Teilmenge von ANSI SQL (*Structured Query Language*) zur Verfügung, eben ein Mini-SQL. Sie ist zur Verwaltung von kleineren Datenbeständen geeignet. Mini SQL unterstützt z.B. keine Sichten (*views*), Teilanfragen (*sub-queries*) und implizite Funktionen (z.B. *avq()*).

Mini SQL ist als einfacher Datenbankdienst von www-Servern beliebt geworden. Es ist für nicht kommerzielle Nutzung kostenfrei und per *ftp* erhältlich.

Da die Datenbanksprache eine Teilmenge von SQL ist, kann eine Anwendung relativ leicht auf ein anderes Datenbankverwaltungssystem umgestellt werden.

Bei Mini SQL wird eine Datenbasis von einem Server bereitgestellt und verwaltet. Eine Datenbasis besteht aus einer Menge von Datenbanken. Jede Datenbank besteht aus einer Menge von Tabellen. Eine Tabelle hat mindestens ein Feld:



Von SQL vorgegeben ist, daß sowohl eine Datenbank als auch eine Tabelle in einer Datenbank eindeutig durch ihren Namen identifiziert werden. Diese Namen werden in den Anfragen verwendet. Zur Datenbankmaschine gehört, daß ein Server durch seinen Namen (z.B. Rechnername) ausgewählt wird.

C-API

Die Anwendungsprogrammchnittstelle (API: *Application Program Interface*) für C stellt durch Funktionen und Typdefinitionen einen Zugriff auf die Tabellen von Datenbanken zur Verfügung. Die API besteht aus 25 Funktionen, davon sind 4 als Makros implementiert, 7 Konstanten, durch **#define** festgelegt, und 5 Typdefinitionen, davon sind vier Strukturen. Ein Beispiel zur Benutzung folgt weiter unten.

C++-API Msql++

Zunächst entwickeln wir eine einfache C++-Schnittstelle zu Mini SQL, Msql++. Sie soll den objektorientierten Ansatz ausnutzen, jedoch zunächst zum Vergleich die gleiche Funktionalität wie die C-Schnittstelle besitzen. Bei der Entwicklung einer neuen Schnittstelle fallen einem natürlich viele "nette, kleine" Erweiterungen ein.

Mit der C++-Schnittstelle soll eine einfache Anwendung einfach programmierbar sein und eine komplexere Anwendung mit einem angemessenen Aufwand ebenfalls implementiert werden können.

Eine einfache Anwendung kennt nur eine Tabelle und möchte Daten der Tabelle weiterverarbeiten.



Abbildung 1: Interaktive Datenbankverwaltung

Als Beispielanwendung verwalten wir Rechnermodelle, wie in Abbildung 1 im Ausgabefenster dargestellt. Es sollen unter Verwendung von Msql++ alle Namen und Prozessoren der Modelle mit einer Taktrate von 160 MHz zusammengestellt werden:

```
Database system("system");
vector< vector<string> >
    result;
system.query(
    "SELECT Name, Prozessor
    FROM   Modell
    WHERE  Taktrate=160",
    result);
```

Die Datenbank mit dem Namen *system* wird mit dem Konstruktor *Database* eröffnet, siehe auch Abbildung 2. Die Methode *query()* schickt eine SQL-Anfrage zum Server und liefert das Ergebnis im zweiten Parameter zurück.

Einfach ausgegeben wird das Ergebnis mit:

```
cout << result;
```

Ein Verbund (*join*) von Tabellen wird in SQL durch eine normale SELECT-Anweisung durchgeführt. Im folgenden werden alle Namen der Rechner gesucht, die als CPU einen Power-PC haben. Da beim Rechner nur das Modell notiert ist, muß ein Verbund von Rechner und Modell durchgeführt werden:

```
if(system.query(
    "SELECT Rechner.Name
    FROM   Rechner,Modell
    WHERE  Rechner.Modell
           = Modell.Name
    AND   Modell.Prozessor
    LIKE  'Power-PC%'
```

```

        ORDER BY Rechner.Name",
        result))
    { cout << result; }

```

Das Ergebnis ist in der Regel eine Tabelle, die als zweidimensionale Matrix `vector<vector<string>>` dargestellt wird.

Für das Resultat ist ein Standardtyp gewählt worden, damit der Anwendungsprogrammierer nicht noch eine weitere Art lernen muß, wie die einzelnen Elemente einer Matrix durchlaufen werden. Im folgenden Beispiel sollen alle Modelle, die weniger als 16 MB Hauptspeicher haben, um 32 MB erweitert werden:

```

if(system.query(
    "SELECT Name,Hauptspeicher
    FROM   Modell
    WHERE  Hauptspeicher<16",
    result))
{ for ( unsigned int i = 0;
      i<result.size(); i++ )
  { system.query(
      "UPDATE Modell
      SET Hauptspeicher="
      +toString
      (atod(result[i][1])
      +32.0)
      +"WHERE Name=' "
      +result[i][0]+' '");
  } }

```

Die SELECT-Anfrage liefert in `result` die Namen und Hauptspeichergrößen aller Änderungskandidaten. Die UPDATE-Anfrage wird aus dem Namen des Modells, `result[i][0]`, der Primärschlüssel ist, und dem neuen Hauptspeicherwert zusammengestellt, der sich aus dem alten Wert, `result[i][1]` berechnen läßt. `toString()` wandelt hier eine Gleitkommazahl in ein Objekt vom Typ `string` um.

Die Abfragesyntax hinsichtlich der Angabe von Zeichenketten und Werten wird von SQL festgelegt. Daß die Resultattabelle nur aus Zeichenketten besteht, wird von Mini SQL festgelegt. Bei numerischen Werten muß entsprechend konvertiert werden.

In C

Was haben wir bisher gegenüber der C-API gewonnen? Das erste Beispiel, die Auswahl aller Rechner mit einer Taktrate von 160 MHz, sieht in C, ohne größere Fehlerabfragen, folgendermaßen aus:

```

int socket = mysqlConnect
              ((char*)NULL);
if ( mysqlSelectDB
      (socket,"system")
  == -1 ) /* ... */
if ( mysqlQuery ( socket,
  "SELECT Name, Prozessor
  FROM   Modell
  WHERE  Taktrate=160" )
  != -1 )
{ m_result * result;
  result = mysqlStoreResult();
  if ( result != NULL )
  { m_row row;
    while((row = mysqlFetchRow
            (result)) != NULL)
    { printf("%s\t%s\n",

```

```

    row[0]?row[0]:"(null)",
    row[1]?row[1]:"(null)");
}
mysqlFreeResult (result);
} }

```

Die Benutzung der C-Schnittstelle fordert mehr Detailwissen über die Bibliothek, um routinemäßige Aufrufe wie z.B. *mysqlConnect* zur Verbindung mit einem Server und *mysqlSelectDB* zur Auswahl der entsprechenden Datenbank nicht zu vergessen und in der richtigen Reihenfolge durchzuführen.

Des Weiteren ist die Struktur zur Übergabe der Ergebnisdaten natürlich eine sehr spezielle Struktur, *m_result*, die durch eine Folge von Funktionsaufrufen von *mysqlFetchRow* in Strukturen vom Typ *m_row* ausgelesen werden kann, wenn vorher der Aufruf von *mysqlStoreResult* nicht vergessen wurde. Und zu guter Letzt muß das Anwendungsprogramm das Resultat mit *mysqlFreeResult* aufräumen.

Vorteile der C++-API

Die Vorteile der C++-API, die mit dem Beispiel deutlich geworden sein sollten, liegen in der einfacheren Nutzung. Insbesondere werden vorgeschriebene Folgen von Funktionsaufrufen in der Bibliothek versteckt.

Wenn ein Objekt der Klasse *Database* erzeugt wird, dann wird auch die Verbindung zu dem Server eröffnet. Die Verbindung, die in der Implementierung dann durch einen Socketwert identifiziert wird, ist ein Attribut des Objekts und wird bei allen Zugriffen auf die Datenbank sozusagen automatisch eingefügt. Damit ist eine Fehlerquelle ausgeschaltet, nämlich die Übergabe einer falschen Socket-Identifikation.

Das Holen des Resultats und das Freigeben des Speicherplatzes wird durch die Methode *query()* erledigt, wobei die Freigabe letztendlich auf den Destruktor von *vector* abgebildet wird. Aber dazu ist das Konzept der Destruktoren eingeführt.

Durch das Überladen der Methode *query()* wird einmal eine Anfrage ohne und einmal mit Resultatermittlung zur Verfügung gestellt (siehe Abbildung 2).

Database			
	Rückgabewert	Argumente	Beispiel
Database		const string& dbName, const string& server = ""	Database db("system")
query	bool	const string& query	db.query("DELETE FROM Modell WHERE Name=NeXT")
query	bool	const string& query, vector<vector<string>>& table	db.query("SELECT * FROM Modell", result)
schema	bool	const string& table, vector<Field>& theSchema	db.schema("Modell", aSchema)

Abbildung 2: Klasse "Database" (Auszug)

Die Iteration über die Ergebnismenge wird mit Standarditeratoren durchgeführt, so daß keine spezielle Funktionsaufrufreihenfolge erlernt und eingehalten werden muß.

Eine große Programmiererleichterung bringt auch die Verwendung der Klasse *string* für Zeichenketten aus der Standard-C++-Bibliothek. Das Beispiel, in dem ausgewählte Modelle um 32 MB Hauptspeicher erweitert werden, nutzt *string* zur Zusammenstellung einer Anfrage extrem aus. In C wäre es wesentlich

unübersichtlicher.

Schemata et cetera

Die Vorteile, die bisher an einfachen Datenbankabfragen gezeigt wurden, verstärken sich bei etwas komplizierteren Anwendungen, in denen auch auf das Schema einer Tabelle zugegriffen werden muß. Eine Schemabeschreibung in SQL kann auch auf ein Objekt der Klasse `vector<vector<string>>` abgebildet werden, wobei ein Feld in einem Objekt der Klasse `vector<string>` beschrieben wird.

Als Alternative hat sich hier aber auch eine neue Klasse `Field` bewährt, die entsprechende Methoden auf die einzelnen Beschreibungsmerkmale eines Feldes zulassen. Das erlaubt dann zusätzliche Methoden, die z.B. aus einem Vektor von Feldbeschreibungen einen Primärschlüssel heraussuchen.

Damit läßt sich ein allgemeines Werkzeug zur interaktiven Bearbeitung von Datenbanken realisieren, wie in Abbildung 1 dargestellt. Diese Anwendung wurde mit `Msql++` und die Oberfläche mit `tcl/tk` implementiert.

Um den Zugriff auf das Schema einer Tabelle zu erleichtern, wird die Klasse `Table` zur Verfügung gestellt, die auch Methoden wie `query()` an ihre Datenbank durchreicht (siehe Abbildung 3).

Table			
	<i>Rückgabewert</i>	<i>Argumente</i>	<i>Beispiel</i>
Table		<code>const string& dbName,</code> <code>const string& tableName</code> <code>const string& server=""</code>	<code>Table Modell("system",</code> <code> "Modell")</code>
name	<code>const</code> <code>string&</code>		<code>cout<<Modell.name()</code>
primaryKeyName	<code>string</code>		<code>cout<<Modell.primaryKeyName()</code>
primaryKeyIndex	<code>int</code>		<code>if(result[0][Modell.</code> <code> primaryKeyIndex()]==0)</code>
query	<code>bool</code>	<code>const string& query</code>	<code>Modell.query("DELETE FROM</code> <code> Modell WHERE Name=NeXT")</code>
schema	<code>bool</code>	<code>vector<Field>& aSchema</code>	<code>Modell.schema(aSchema)</code>

Abbildung 3: Klasse "Table" (Auszug)

Das Einbringen einer Tabellenzeile (als Reaktion auf das Drücken des Knopfes "Ändere Zeile" aus Abbildung 1), die die Benutzerin interaktiv am Bildschirm geändert hat, ist dann relativ kurz. Im folgenden wird die Zeile `vector<string> row`, die vom Bildschirm an der Cursor-Position eingelesen wurde, in der Tabelle `Table table` ersetzt:

```
if(table.query(
    "DELETE FROM "
    +table.name()+" WHERE "
    +table.primaryKeyName()
    +" = '"+row
    [table.primaryKeyIndex()]
    + "'") )
{ if(table.query(
    "INSERT INTO "
    +table.name()+" VALUES ("
```

```

+table.wertliste(row)
+)" " ) )
// ...

```

Die Methoden *primaryKeyName()* bzw. *primaryKeyIndex()* liefern den Feldnamen bzw. Index des Primärschlüssels. Die Methode *wertliste()* macht aus einem Stringvektor eine Werteliste in der Syntax von SQL.

Entwurfsentscheidungen

Da eine ANSI-C-Schnittstelle (bis auf wenige Ausnahmen, z.B. wegen Namenskollisionen mit neuen Schlüsselwörtern) direkt von einem C++-Programm benutzt werden kann, geht es beim Entwurf der C++-Schnittstelle nicht darum, exakt dieselbe Schnittstelle bereitzustellen. Es geht vielmehr darum,

- die gesamte Funktionalität zu erhalten und
- die Benutzung des Pakets einfacher zu gestalten.

Einfacher bedeutet in diesem Zusammenhang leicht verständlich, konform mit dem objektorientierten Ansatz von C++, möglichst unter Verwendung von schon aus C++ bekannten Konzepten. Im folgenden werden die Entwurfsentscheidungen, die bei der Entwicklung von Msql++ getroffen worden sind, verallgemeinert dargestellt.

Für eine Bibliothek gibt es in der Regel genau eine Definitionsdatei. Ein Programmteil, der die Bibliothek benutzt, macht diese durch *#include <Definitionsdatei>* bekannt. Auch wenn mehrere Klassen, wie in unserem Beispiel *Database* und *Table*, zur Verfügung gestellt werden, macht die Definition in genau einer Datei die Benutzung einfacher.

Ausnahmen bilden alle größeren Bibliotheken, die mehrere Aufgabenbereiche abdecken, die sinnvoll voneinander getrennt werden können. "Sinnvoll" bedeutet hier hauptsächlich, daß die Trennung für die Benutzung offensichtlich und damit leicht verständlich ist.

Die Festlegung der Klassen ergibt sich meistens aus dem Anwendungsgebiet der Bibliothek. In unserem Beispiel sind es Datenbanken, Tabellen und Felder. Ein Objekt einer Klasse kapselt in der Regel Daten, die in einer reinen funktionalen Schnittstelle jedesmal als Parameter übergeben werden müssen. Ein Beispiel aus Msql++ ist die Socketidentifikation, nachdem eine Verbindung mit einem Server aufgebaut wurde.

Die Konstruktoren sollte man so mit voreingestellten Parametern gestalten, daß eine einfache Anwendung sehr wenige Parameter angeben muß, wenn überhaupt. Im Beispiel verlangt der Konstruktor von *Database* nur den Namen der Datenbank. Der voreingestellte Server ist der lokale Rechner. Mit einem weiteren Parameter (in Abbildung 2 nicht angegeben) kann die Datenbank auch noch bezüglich Fehlermeldungen zum Schweigen gebracht werden.

Das Überladen von Funktionen wird ausgenutzt, um gleichartige Funktionen gleich nennen, aber überflüssige Parameter vermeiden zu können. Ein schönes Beispiel ist die Methode *query()*, die nur auf Wunsch eine Resultattabelle zurückliefert.

Die Destruktoren übernehmen alle notwendigen Aufräumarbeiten, um die Anwendungsprogrammierung zu entlasten.

Entwicklungsaufwand

Die Entwicklung einer C++-Schnittstelle, die auf einer vorhandenen C-Schnittstelle aufbaut, ist nicht besonders aufwendig, aber insgesamt ist der Arbeitsaufwand für Entwicklung und insbesondere Test nicht zu unterschätzen.

Des weiteren muß die C++-Schnittstelle immer dann angepaßt werden, wenn sich

die C-Schnittstelle ändert. Aus diesem Grund veralten z.B. C++-Schnittstellen aus dem Public-Domain-Bereich sehr schnell. Die notwendige Anpassung der C++-Schnittstelle hat aber den positiven Effekt, daß die Anwendungsprogramme in der Regel nicht geändert werden müssen.

Ein solche zusätzliche Schnittstelle ist dann sinnvoll, wenn

- die Schnittstelle in einem größeren Projekt eingesetzt,
- die C-Schnittstelle ohnehin um einige Möglichkeiten erweitert oder
- die Schnittstelle z.B. aus Portabilitätsgründen gekapselt werden soll.

Zusammenfassung

Insgesamt ist eine C++-Schnittstelle leichter erlern- und benutzbar. Die Anzahl der möglichen Fehler bei Verwendung der Bibliothek wird deutlich reduziert.

- Der erste Einstieg in die Benutzung der Bibliothek gestaltet sich wesentlich einfacher.
- Die Anwendungsprogrammierung wird von der Übergabe von Parametern befreit, die nur der Übergabe von Systemzuständen dienen und für die eigentliche Anwendung keine Bedeutung haben. Damit wird die Anzahl der "Fehlbenutzungen" reduziert.
- Des weiteren wird die Einhaltung von Aufruffreihenfolgen von Funktionen von der C++-Schnittstelle garantiert.
- Die C++-Schnittstelle wird von der Anzahl Methoden und Klassen her nach meiner Erfahrung umfangreicher als das C-Äquivalent, denn sie bietet mehr Komfort.
- Der Erstellungsaufwand für eine C++-Schnittstelle lohnt sich in der Regel schon bei mehrfacher Verwendung innerhalb einer Anwendung. Ein Nachteil ist, daß die C++-Schnittstelle zusätzlich mit gewartet werden muß.

Sinnvoll ist natürlich, daß in Zukunft für eine Bibliothek gleich eine C++-Schnittstelle bereitgestellt wird. Aber vom Verzicht auf C sind wir wohl noch einige Zeit entfernt.

Quellen

Die vollständigen Programme befinden sich auf dem FTP-Server *ftp.informatik.uni-osnabrueck.de* unter *pub/hanser/um*.

Literatur

- [1] D. J. Hughes *Mini SQL, A Lightweight Database Engine*, Release 1.0.11, Hughes Technologies Pty Ltd., 1996, <http://Hughes.com.au/>.