C++, STL und Qt

C++, STL und Qt

Objektorientiertes Programmieren mit C++, Benutzen der STL-Standard-Bibliothek und graphisches User-Interface Qt

Dipl.Ing. Christoph Stockmayer, Schwaig, sto@stockmayer.de

1. Einleitung

Graphische Programme in C++ schreibt man systemunabhängig mit der Qt-Bibliothek, die bei Linux meist mit dabei ist (für Privatabwender kostenlos, für kommerzielle Benutzung lizenzpflichtig) und auch für Windows existiert (lizenzpflichtig). Auch das KDE-System basiert auf dieser Bibliothek. Benötigt man Standard-Funktionalität wie Strings oder Container, entnimmt man diese aus der C++Standardbibliothek (Standard Template Library). Dieser Artikel beschreibt das sinnvolle Zusammenspiel dieser Komponenten am Beispiel eines kleinen Adressprogramms.

2. STL: sortiertes Speichern in einer map

Die Speicherung der Adressen kann sortiert in einer `map` erfolgen: Die `map` ist ein Container, der schlüsselorientiert Daten enthält. Da die `map` als Template-Klasse realisiert ist, ist es möglich, sowohl den Typ des Schlüssels, als auch den Typ der Daten selber festzulegen. Sollen mehrere gleiche Schlüssel speicherbar sein, ist eine `multimap` zu verwenden.

Beim Schlüssel entscheiden wir uns beim Nachnamen für den Typ `string`, die Daten sind in einem Objekt vom Typ `adress` zusammengefaßt:

```
#include <string>
#include <map>
typedef map<string, adress, Vergleich<string> > Datenbank;
```

Damit nicht immer wieder die Template-Notation verwendet werden muß, wurde hier zunächst ein `typedef`-Typ vereinbart. Der erste Typ ist der Typ des Schlüssels, der zweite Typ der der Daten. Es handelt sich hier um eine eigene Klasse, die die restlichen Adressdaten enthält:

```
class adress
{
        public:
                adress(string _v, string _s,
                       string _p, string _o, string _t);
                adress();
                const string &getVorname()  const;
                const string &getStrasse()  const;
                const string &getPLZ()      const;
                const string &getOrt()      const;
                const string &getTelephon() const;
        private:
                string vorname;
                string strasse;
                string plz;
                string ort;
                string tel;
};
```

Auf die im `private`-Bereich abgelegten Daten vom Typ `string` kann mit Hilfe von Zugriffsmethoden lesenderweise zugegriffen werden (`const`).

Der dritte Typ des `typedefs` ist die Angabe der gewünschten Sortierung in der `map`. Wird das vorhandene `less` verwendet, wird aufsteigend sortiert:

```
typedef map<string, adress, less<string> > Datenbank;
```

Wir wollen aber Groß- und Kleinschreibung zusammensortieren (also nicht unterscheiden), so müssen wir eine eigene Sortierklasse schreiben, ein Funktionsobjekt oder auch Functor genannt:

```
template <class T>
class Vergleich
{
        public:
                Vergleich() {}
                int operator()(const T &s1, const T &s2)
                {
                        return toUpper(s1) < toUpper(s2);
                }
};
```

Diese Klasse besteht im wesentlichen aus einer Überlagerung des `()`-Operators, der zwei zu vergleichende Elemente bekommt und `0` zurückliefern muß, wenn gleich, ansonsten > oder <, je nach Sortierfolge. Dies wird erreicht, in dem die Strings zunächst in Großbuchstaben konvertiert und dann verglichen werden (<-Operator der `string`-Klasse).

`toUpper` könnte etwa so programmiert werden:

```
#include <cctype>

string toUpper(const string &s)  // wandelt klein in GROSS
{
        int l = s.length();
        int i = 0;
        string u;
        while(i < l)
        {
                if(islower(s[i]))
                        u += toupper(s[i]);
                else
                        u += s[i];
                i++;
        }
        return u;
}
```

3. Zugriff auf Elemente der `map`

Um auf Elemente eines Containers zuzugreifen, verwendet die STL Iteratoren. Diese sind wie Pointer zu verstehen, kennen den `*`-Operator und können mit Hilfe von `++` hoch bzw. `--` runtergezählt werden. Ein solcher Iterator muß angelegt und initialisiert werden:

```
Datenbank daten;        // map-Objekt anlegen
```

```
Datenbank::iterator iter = daten.begin();
```

Ist die Iteratation fertig, steht der Iterator auf `daten.end()`.

Nun kann mit `++iter` von Adresse zu Adresse gegangen werden und mit

```
(*iter).first
```

auf den Schlüssel und mit

```
(*iter).second
```

auf die Daten zugegriffen werden (die Daten sind hier ein Objekt der `adress`-Klasse).
Aber auch ein Zugriff (lesender oder schreibenderweise) über den `[]`-Operator ist möglich:

```
string nachname = "Meier";
daten[nachname];
```

wobei das Ergebnis eine Referenz auf ein `adress`-Objekt ist (Zugriff über `get`-Methoden).

4. Qt-Elemente und Slots

Für die graphische Bedienung der Adressen brauchen wir im wesentlichen Textfelder und Knöpfe, die z.B. mit V- oder H-Boxen oder Gitter eingepaßt werden:

```
#include <qpushbutton.h>
#include <qlineedit.h>
#include <qvbox.h>
#include <qlabel.h>

QPushButton *suche = new QPushButton("&Suche",  ...);
new QLabel("Nachname",  ...);
QLineEdit *nachname  = new QLineEdit("", ...);
```

Die graphischen Elemente werden in einer hierarchischen Anordnung (horizontal positioniert) miteinander verbunden:

```
w1 = new QHBox(...);            // Knöpfe horizontal
w1->setSpacing(10);
clear     = new QPushButton("Lösche Fel&der", w1);
speichere = new QPushButton("S&peichere", w1);
```

Zur Verfeinerung können noch eine Menue-Leiste (für das Speichern der Adressen in verschiedenen Dateien), ein Verlaufsbalken für die Anzeige des Lade- oder Speichervorgangs und ähnliches integriert werden.

Damit auf Knopfdruck etwas getan wird, verbinden wir die Signale der Knöpfe mit selbst definierten Slots unserer Datenbank-Klasse. Dadurch werden bei Knopfdruck Methoden dieser Klasse aktiviert (die Makros `SIGNAL` und `SLOT` helfen in Qt die Signale und Methoden mit ihren Übergabeparametern zu verifizieren):

```
DB *db = new DB(w, "Datenbank", ..., file);
db->connect(speichere, SIGNAL(clicked()), SLOT(speichereText()));
```

```
db->connect(suche, SIGNAL(clicked()), SLOT(suchText()));
db->connect(clear, SIGNAL(clicked()), SLOT(clear()));
db->connect(loesche, SIGNAL(clicked()), SLOT(loesche()));
db->connect(printe, SIGNAL(clicked()), SLOT(print()));
db->connect(up, SIGNAL(clicked()), SLOT(up()));
db->connect(down, SIGNAL(clicked()), SLOT(down()));
```

Die Datenbank-Klasse muß diese Methoden in Form von Slots zur Verfügung stellen:

```
#include <qobject.h>

class DB : public QObject
{
        Q_OBJECT

        public:
                DB(QObject *parent, const char *name,
                   QLabel *_lanz,
                   QLineEdit *nn, QLineEdit *vn,
                   ... );
                ~DB();
                void load();
                bool save();

        public slots:
                void speichereText();
                void suchText();
                void clear();
                void print();
                void loesche();
                void getFile();
                void up();
                void down();

        private:
                QObject *parent;

                ...

                Datenbank daten;
                Datenbank::iterator it;
};
```

Zwei Voraussetzungen müssen eingehalten werden:
- die Slot-Klasse erbt von `QObject`
- das Makro `Q_OBJECT` wird zu Beginn (ohne `;`) aufgerufen.

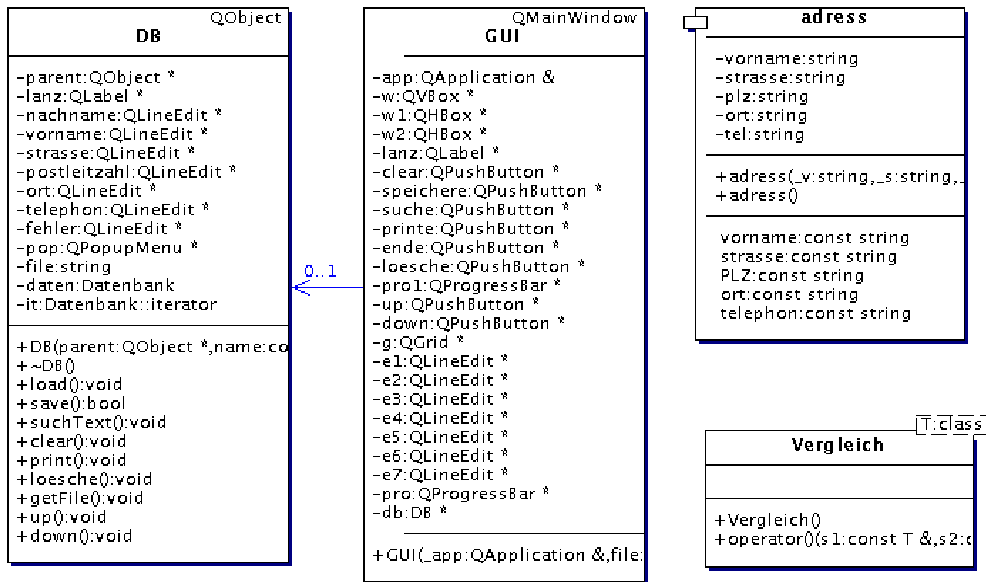
Das Schlüsselwort `slots` gehört nicht zur Sprache von C++, sondern wird später vom `moc` von Qt umgesetzt.

5. Verbindung der Klassen

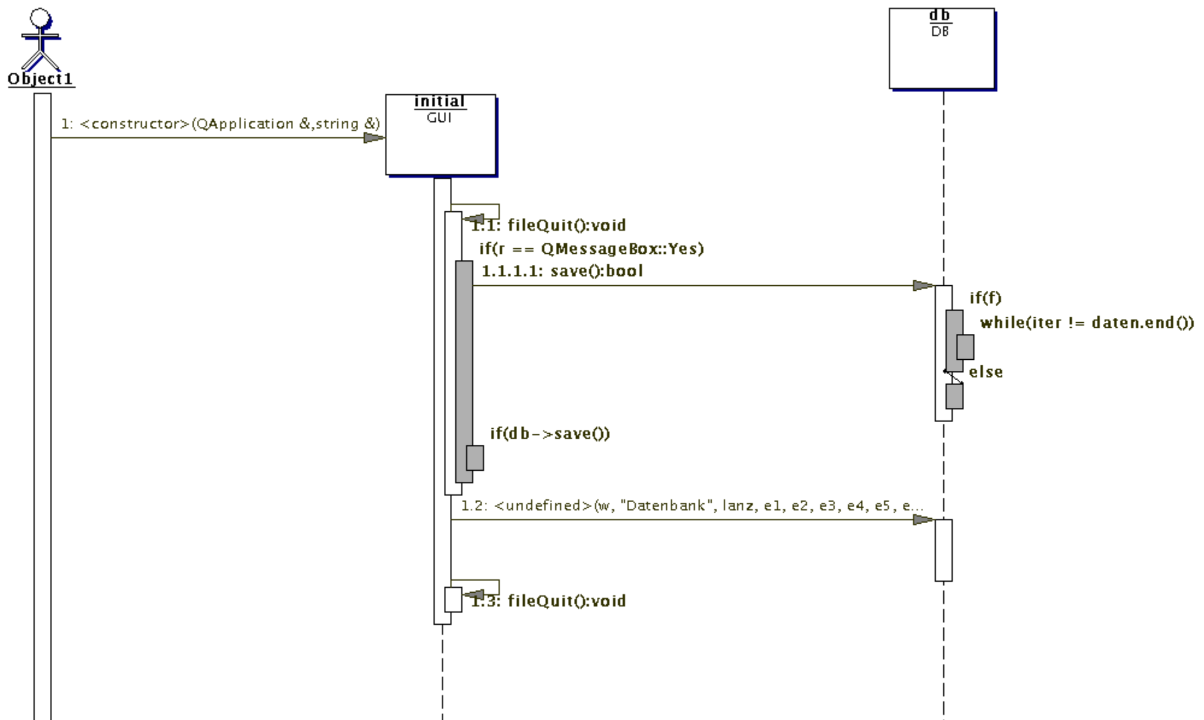
Insgesamt gibt es also 4 Klassen:
- `GUI` für die graphische Darstellung,
- `DB` für die Implementierung der Adressverwaltung mit Hilfe einer `map`,

- `adress` zur Speicherung der Daten in der `map` und
- `Vergleich` zur Realisierung der Sortierung in der `map`.

Das Klassendiagramm (in UML) zeigt dies:



Im Hauptprogramm wird ein `GUI`-Objekt angelegt, das die Graphik aufbaut und eine Verbindung zu einem Datenbank-Objekt aufbaut . Im Sequence-Diagramm wird dies deutlich:



6. Zusammensetzung des Quellcodes zum exe: Makefile

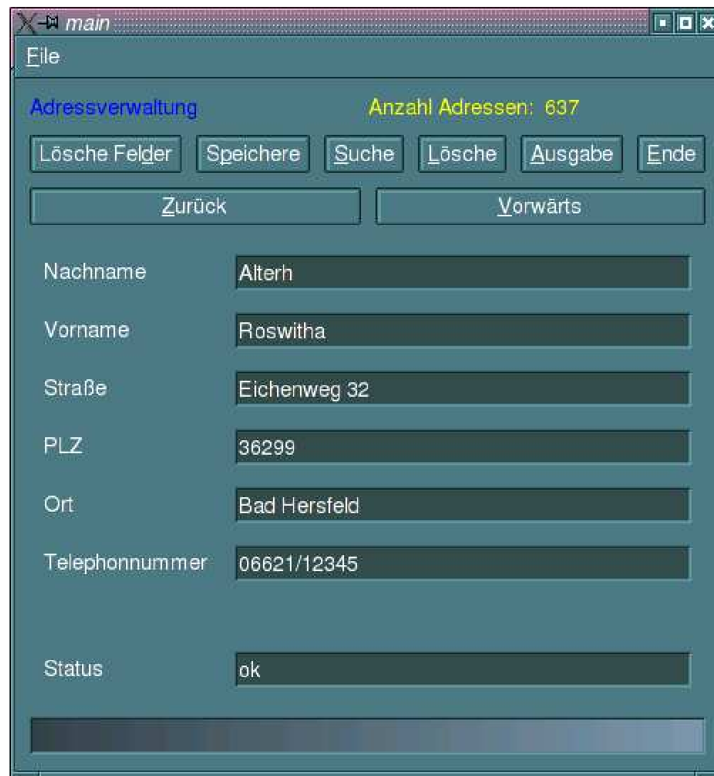
Um das alles übersetzt zu bekommen empfiehlt sich die Steuerung des Compilierens mit `make`. In einen `Makefile` werden die Sourcen, die Headerabhängigkeiten und der `moc` (Meta Object Compiler) von Qt notiert:

```
CFLAGS = -I/usr/lib/qt3/include -Wno-deprecated
LDFLAGS = -L/usr/lib/qt3/lib -lqt-mt
CC = c++
EXE = main
SRC = gui_main.c gui.c db.c db.moc.c gui.moc.c adressen.c
OBJ = $(SRC:.c=.o)

$(EXE): $(OBJ)
	$(CC) $(CFLAGS) $(LDFLAGS) -o $(EXE) $(OBJ)

gui_main.o: gui.h
gui.o: gui.h adressen.h db.h
db.o: adressen.h db.h
adressen.o: adressen.h

.SUFFIXES: .h .moc.c .moc.o
.h.moc.c:
	/usr/lib/qt3/bin/moc $< -o `basename $< .h`.moc.c

.moc.c.moc.o:
	$(CC) -c $(CFLAGS) `basename $< .h`.moc.c

clean:
	rm -f $(OBJ)
```

Nach den Flags für den Compiler und Binder (Pfade für Headerfiles und Bibliotheken) werden die Quellen notiert. Die Hauptabhängigkeit betrifft den startbaren exe-File (main). In einer besonderen (neuen) Regel wird der `moc` beschrieben, der aus den Headerfiles mit Klassen, die Slots definieren, einen zusätzlichen Code generiert, der dann dazugebunden werden muß.

Und so sieht das Resultat aus:

Die hier gezeigten Listing-Fragmente sollen nur dem Verständnis und der Erklärung der Zusammenhänge dienen. Im beigefügten Archiv befinden sich die vollständigen Quellen (Implementierung der einzelnen Methoden usw.).



Literatur:

Breymann, C++, Hanser, 3-446-21723-1
Stockmayer, Tips und Tricks zu UNIX, C und C++, Hanser, 3-446-18202-0
Lehner, KDE- und Qt-Programmierung, Addison-Wesley, 3-8273-1753-3
Qt: www.trolltech.com
UML: www.togethersoft.com

zur Person:

Christoph Stockmayer ist seit 20 Jahren freiberuflicher Trainer in den Gebieten Programmierung, C/C++/Java/Perl, OOA/OOD und im gesamten UNIX/Linux-Sektor. Er ist SuSE Certified Linux-Trainer und Lehrbeauftragter an der FH Nürnberg. Er betreut außerdem Programmier- und UNIX/Linux-Projekte.