HOCHSCHULE HAMM-LIPPSTADT

Praktische Informatik

Vorlesung 15

C++ Bibliotheken

Zuletzt haben wir gelernt…

- welche Vorteile C# und C++ jeweils haben.
- wie Klassen in C++ erstellt werden.
- welche die manuelle Speicherverwaltung in C++ funktioniert.
- wie man in C++ Objekte erzeugt.
- wie man Konstruktor und Destruktor benutzt.
- welche Eigenarten Zuweisungen in C++ haben und was der Copy-Konstruktor macht.
- wie Objekte in Form von Parametern übergeben werden.
- wie die Übersetzung eine C++ Projektes mit dem Präprozessor, dem Compiler und dem Linker funktioniert.
- wie man mit Makefiles den Übersetzungsprozess steuert.

Inhalt heute

- Bibliotheken in C++
- Die Standardbibliothek
- Container, Iteratoren, Algorithmen
- Die Klassen vector und map
- Grafische Oberflächen mit Qt Widgets
- Layout Container
- Signals und Slots

Bibliotheken

- Die Programmiersprache C++ bringt (wie andere Programmiersprachen auch) selbst nur wenig Funktionalität mit.
  - Einige einfache Datentypen (int, double, …)
  - Kontrollstrukturen (if, for, while, …)
  - Objektorientierte Konzepte (Klassen, Vererbung, …)

- Viele Funktionen und Datenstrukturen kommen erst aus Bibliotheken.
  - Die Klasse String, Zugriff auf Dateien
  - Container und Algorithmen (z.B. sortieren)
  - GUI Entwicklung

- Zur Erinnerung: In C++ eine Bibliothek zu benutzen bedeutet:
  - Eine Header-Datei einzubinden: #include …
  - Dem Linker mitzuteilen, wo die benutzen Funktionen zu finden sind.

Std, Qt, …

- Mit der Zeit haben sich aus der Vielzahl von Bibliotheken einige Standards etabliert.

- **Die C++ Standardbibliothek**
  - Wird bei der Installation des C++ Compilers mitgeliefert.
  - Klasse String, Ströme, Reguläre Ausdrücke, Datum und Uhrzeit, Multithreading, …
  - Datenstrukturen (Listen, Assoziative Arrays, …) und Algorithmen.

- **Qt**
  - Plattformunabhängige Entwicklung grafischer Benutzeroberflächen

- Boost, …

Namespace

- Damit die Namen von Klassen und Funktionen aus Bibliotheken nicht mit den Namen eigener Klassen kollidieren, wurden in C++ die sog. Namensbereiche (*engl. namespace*) eingeführt.
  - Namensbereiche organisieren die Klassen in logische Bereiche.
  - Das gleiche Konzept existiert auch in C#.

```
namespace mein_Namensbereich  {
     // hier sind nun Funktionen oder Klassen zu finden
}
```

- Will man Klassen aus einem bestimmten Namensbereich nutzen, so schreibt man den Namensbereich mit zwei Doppelpunkten vor den Namen der Klasse.
  - z.B. `std::string`

- Nutzt man häufig die Klassen aus einem bestimmten Namensbereich, so kann man diesen Namensbereich in einem Code-Block einführen.
  - Dadurch muss nicht mehr vor jedem Klassenname der Namensbereich aufgeführt werden.

```
using namespace std;
```

Die C++ Standardbibliothek

- Die C++ Standardbibliothek (STD) beinhaltet eine ganze Reihe von standardisierten Klassen und Funktionen.
  - Die Klassen der Standardbibliothek finden sich im Namensbereich `std`.

- Ungemein wichtig ist z.B. die Klasse `string`, die für C++ eine sehr bequeme Art der Zeichenkettenverwaltung darstellt.
  - `std::string zeichenkette("Text!");`

- Auch finden sich hier sog. Ein- und Ausgabeströme `cin` und `cout` mit deren Hilfe man Daten auf der Konsole ein oder ausgeben kann.

```
cout << "Dies ist ein Text!" << endl;
```

Beispiel

```
#include <iostream>
#include <string>

using namespace std;

int main(int argc, char** argv) {
    string name;
    cin >> name;

    string test("Hallo");
    test += " " + name;
    test.append("!");
    test.insert(5, ",");

    cout << test << endl;

    return 0;
}
```

<iostream> definiert die Ströme cin und cout.

Ein Zeichenkettenobjekt anlegen und von der Standardkonsole einlesen.

Einige Operationen auf der Zeichenkette.

Ausgabe auf der Konsole.

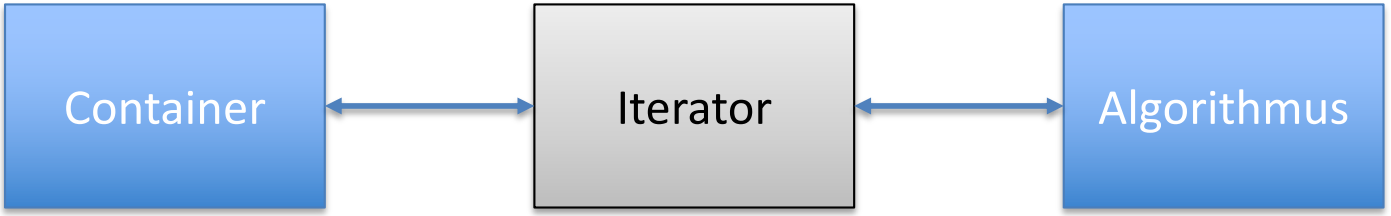
Container, Iteratoren, Algorithmen

- Die STD beinhaltet auch eine Vielzahl von unterschiedlichen Datenstrukturen.
  - Sequenzen: Listen, Warteschlangen, Stapel, Mengen, Assoziative Arrays, …
- Jede Datenstruktur speichert seine Elemente auf seine eigene Art.
  - Auch der Zugriff auf die Elemente gestaltet sich daher mitunter unterschiedlich.
- Würde man zu jedem diese *n* Container *m* **Algorithmen** zum Sortieren usw. anbieten, müsste man *n* * *m* Funktionen realisieren.
  - Das wäre ein großer Aufwand und nicht praktikabel.

Iteratoren

- Die STD löst dieses Problem mit Hilfe der **Iteratoren**.
  - Iteratoren sind **Objekte**, die wie **Zeiger** funktionieren,
  - Jede Datenstruktur bietet einen Iterator an.
  - Mit Hilfe des Iterators kann man über die Elemente des Container *iterieren*.

- Ein Iterator abstrahiert vom Zugriff auf einen Container.



- Mit Hilfe eines Iterators kann ein Algorithmus auf die Elemente des Containers zugreifen.
  - Der Zugriff erfolgt dabei, ohne die spezielle Arbeitsweise des Containers zu kennen.

Beispielcontainer: Vector

- Die Klasse `vector` ist einer der am häufigsten benutzen Container der STL.
  - Entspricht der Klasse List in C#.
  - Wenn keine besonderen Anforderungen an die Art des Zugriffs auf die Elemente vorliegen, ist der Vector die richtige Wahl.

- Ein Vector ist eine Sequenz, d.h. die Elemente im Container besitzen eine Reihenfolge.
  - Wie in einem Array können die Elemente über einen Index angesprochen werden (wahlfreier Zugriff).
  - Elemente können an beliebigen Stellen hinzugefügt und gelöscht werden.

Einige Elementfunktionen der Klasse `vector`

Elementfunktion	Bedeutung
size	Liefert die Anzahl der Elemente zurück.
push_back	Fügt ein Element an das Ende der Liste an.
pop_back	Liefert das letzte Element der Liste zurück und entfernt dieses.
[] oder at	Zugriff auf das Element an einer bestimmten Position.
front	Erstes Element der Liste.
back	Letztes Element
erase	Entfernt ein Element der Position nach, oder mehrere Elemente zwischen zwei Positionen in der Liste.
clear	Löscht alle Elemente aus der Liste.

Beispiel zur Klasse vector

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> ints;
    ints.push_back(1);
    ints.push_back(2);
    ints.push_back(3);
    ints.push_back(4);

    for (size_t i=0; i<ints.size(); i++) {
        cout << ints[i] << endl;
    }

    ints.clear();
    cout << ints.size() << endl;
}
```

Klasse vector einbinden.

Instanz des Containers für int erzeugen.

Einige Elemente hinzufügen.

Auf Elemente über Index zugreifen.

Inhalte des Containers löschen.

Iterator

- Ein **Iterator** wird dazu genutzt, um auf die Elemente eines Containers zuzugreifen.
  - Ein Iterator ist ein Objekt einer Klasse, der sich wie ein Zeiger verhält.
  - Damit das Objekt wie ein Zeiger aussieht, werden in der zugehörigen Klassen Operatoren überladen, z.B. *, ++ und --.

- Die STL definiert eine Basisklasse, die das Standardverhalten eines Iterators festlegt.
  - Ein Iterator kann mindestens …
  - inkrementiert werden (++) und zeigt dann auf das nächste Element des Containers.
  - dereferenziert werden, um auf das aktuelle Element des Containers zugreifen zu können.

- Jeder spezielle Iterator eines Containers kann die Fähigkeiten um weitere Elementfunktionen und Operatoren erweitern.

Iterator erzeugen

- Jeder Container der STL stellt Methoden bereit, einen Iterator zu liefern.

- Der Vector besitzt hier gleich mehrere Möglichkeiten:

Elementfunktion	Bedeutung
begin	Liefert den Iterator ab dem Anfang der Liste.
end	Liefert den Iterator **nach** dem Ende der Liste.
rbegin	Liefert einen invertierten Iterator ab dem Anfang.
rend	Liefert einen invertierten Iterator nach dem Ende.

Elemente eines Vectors über einen Iterator

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<double> doubles;
    doubles.push_back(0.5);
    doubles.push_back(2.5);
    doubles.push_back(3.5);
    doubles.push_back(4.5);

    vector<double>::iterator i = doubles.begin();
    while (i != doubles.end()) {
        cout << *i << endl;
        i++;
    }
}
```

Instanz des Containers für double erzeugen.

Einige Elemente hinzufügen.

Einen Iterator vom Anfang des Containers beschaffen.

Den Iterator wie einen Zeiger benutzen.

Algorithmen

- Die STL definiert eine Vielzahl von Algorithmen, die auf Datenstrukturen operieren.
  - Suche und Tests
  - Tauschen, Füllen, Rotieren, Kopieren
  - Partitionen und Zusammenführen
  - Sortieren, Min, Max

- Ein guter Überblick über die Fähigkeiten bietet http://www.cplusplus.com/reference/algorithm/

Sortieren eines Vectors

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
    srand(static_cast<int>(time(NULL)));
    vector<int> ints;

    for (int i=0; i<1000; i++)
        ints.push_back(rand() % 1000);

    sort(ints.begin(), ints.end());

    for (int i=0; i<ints.size(); i++)
        cout << ints[i] << endl;
}
```

Vector mit Zufallszahlen füllen.

Vector sortieren.

Übersicht über Algorithmen

Algorithmus	Bedeutung
sort	Sortiert die Elemente eines Containers.
count	Zählt das Vorkommen bestimmter Elemente.
merge	Vereinigt zwei Container.
all_of	Prüft, ob alle Elemente in einem Container eine Bedingung erfüllen.
any_of	Prüft, ob irgendein Element in einem Container eine Bedingung erfüllt.
for_each	Führt eine Funktion auf allen Elementen des Containers aus.
transform	Transformiert die Elemente eines Containers.
remove_if	Entfernt Elemente, wenn sie einer Bedingung entsprechen.

Beispiele für Algorithmen

```
bool test(int i) {
    return i > 990;
}

void ausgeben(int i) {
    cout << i << " ";
}

int main()
{
    srand(static_cast<int>(time(NULL)));
    vector<int> ints;
    for (int i=0; i<1000; i++)
        ints.push_back(rand() % 1000);

    int count_of_50 = count(ints.begin(), ints.end(), 50);
    cout << count_of_50 << endl;

    bool any_greater_990 = any_of(ints.begin(), ints.end(), &test);
    cout << any_greater_990 << endl;

    for_each(ints.begin(), ints.end(), &ausgeben);
}
```

Zählen, wie oft die 50 im vector vorkommt.

Prüfen, ob irgendeine Zahl im Vector die Bedingung test erfüllt.

Mit allen Elementen des Vectors die Funktion ausgeben aufrufen.

Callbacks

- Im letzten Beispiel haben wir gesehen, dass wir manchen Algorithmen der STL eine Funktion als Parameter übergeben müssen.
  - Es muss z.B. bei `for_each` mitgeteilt werden, was mit jedem einzelnen Element geschehen soll.
  - Dabei wird ein Zeiger auf eine Funktion benutzt.
  - Dies wird als Rückruffunktion (engl. Callback) bezeichnet.

- Beispiel:

```
for_each(ints.begin(), ints.end(), &ausgeben);
```

- Der `for_each` Anweisung wird ein Zeiger auf die Funktion `ausgeben` übergeben.
  - Die Funktion `ausgeben` wird in diesem Falle für jedes Element des Containers aufgerufen.

Map

- Wie wir gesehen haben, realisiert die Klasse `vector` eine geordnete Liste von Elementen mit wahlfreiem Zugriff.

- Die Klasse `map` realisiert einen gänzlich anderen Container.
  - Der Container `map` speichert Schlüssel-Wert-Paare.
  - Über die Schlüssel kann in ~ O(1) auf die Werte zugegriffen werden.

- Schlüssel müssen eindeutig sein und dürfen nur ein Mal in der Map vorhanden sein.
  - Die Map ist automatisch nach Schlüssel sortiert.

- Der Datentyp für Schlüssel und Werte kann beim Erzeugen der Map mit Hilfe von Templates angegeben werden.
  - map<string, int>
  - map<int, double>
  - …

Beispiel einer Map

```
int main()
{
    map<string, double> gehalt;
    gehalt["Peter"] = 1700;
    gehalt["Hans"] = 600;
    gehalt["Fritz"] = 2500;

    cout << gehalt["Hans"] << endl;
    gehalt.erase(gehalt.find("Fritz"));

    map<string, double>::iterator it = gehalt.begin();
    while (it != gehalt.end()) {
        cout << it->first << ": " << it->second << endl;
        it++;
    }
}
```

Einige Schlüssel-Wert-Paare hinzufügen.

Den Wert zum Schlüssel "Hans" ausgeben.

Den Eintrag mit dem Schlüssel "Fritz" löschen.

Mit Hilfe eines Iterators über die Elemente iterieren.

Qt

- Qt ist eine externe Bibliothek, um mit C++ Anwendungen (mit GUI) zu entwickeln.
  - Qt existiert für viele Plattformen (Windows, Mac, Linux, …).
  - Der gleiche Programmcode kann für unterschiedliche Plattformen übersetzt werden.
  - Qt wird häufig im Umfeld kleiner eingebetteter Systeme benutzt, die über eine GUI verfügen (Medizin, Automotive, …)
- Die Bibliothek wird durch ein eigenes Unternehmen (QT Company) entwickelt.
  - Ehemals war Qt im Besitz von Nokia, …
- Die Bibliothek muss zunächst zusätzlich installiert werden.
  - Aktuelle Version: Qt 5.x.

QT Widgets

- QT ist vollständig objektorientiert.
  - Die Funktionalität ist durchgängig logisch auf Klassen abgebildet.
  - Die Namen aller Klassen fangen in Qt mit Q an.
  - Es muss kein Namespace angegeben werden.
  - Jede Klasse wird in einer eigenen Header-Datei definiert, die über #include eingebunden werden muss.

- Die Programmierung ist technisch ähnlich zu Windows Forms.
  - Um ein Fenster mit Interaktionselementen zu versehen, werden Objekte im Programmcode erzeugt und konfiguriert.
  - z.B. die Klasse `QWindow`, `QLabel`, `QPushButton`, `QTextBox`, `QCheckBox`, …

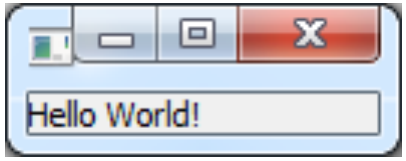
- Es gibt keine Trennung von Design und Code (kein XAML).
  - Datenbindung ist nur rudimentär realisiert.
  - Man muss deutlich mehr „von Hand“ programmieren.

Hello World mit Qt

```
#include <QApplication>
#include <QLabel>

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QLabel label("Hello World!");
    label.show();

    return app.exec();
}
```

Ein Objekt der Klasse QApplication wird erstellt.

Ein Objekt der Klasse QLabel wird erstellt.



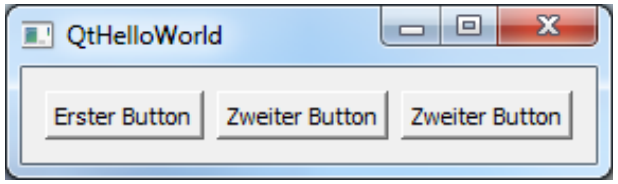
Layout Manager in Qt

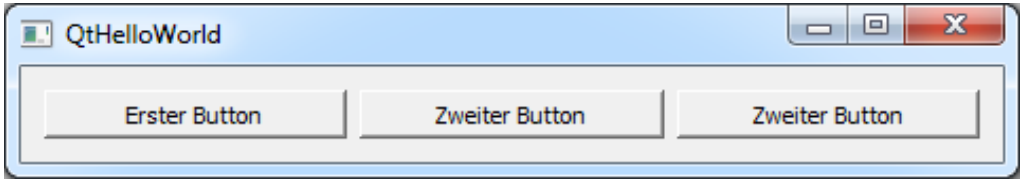
- Auch in Qt existieren sog. Layout Manager.
  - Wie in WPF werden diese gebraucht, um eine Benutzeroberfläche aus mehrere Interaktionselementen aufzubauen.
  - Jeder Layout Manager richtet die Interaktionselemente nach bestimmten Kriterien aus.

- Wichtige Klassen:
  - QHBoxLayout → Horizontales Layout
  - QVBoxLayout → Vertikales Layout
  - QGridLayout → Layout als Tabelle
  - QFormLayout → Anordnung wie in einem Formular

Beispiel

```
#include <QApplication>
#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QWidget window;
    QVBoxLayout *layout = new QVBoxLayout();

    QPushButton *button1 = new QPushButton("Erster Button");
    layout->addWidget(button1);

    QPushButton *button2 = new QPushButton("Zweiter Button");
    layout->addWidget(button2);

    QPushButton *button3 = new QPushButton("Dritter Button");
    layout->addWidget(button3);

    window.setLayout(layout);
    window.show();

    return app.exec();
}
```





Dialoge als Klasse

- Auch in Qt werden Dialoge/Fenster als eigene Klasse erstellt.
  - Man leitet dann i.d.R. von Klassen wie `QWidget`, `QWindow` oder `QDialog` ab.

Header Datei MyDialog.h

```
class MyDialog : public QWidget
{
Q_OBJECT

public:
    MyDialog();
};
```

Wichtig! Das Makro Q_OBJECT in die Klasse einfügen. Wird durch Qt benötigt!

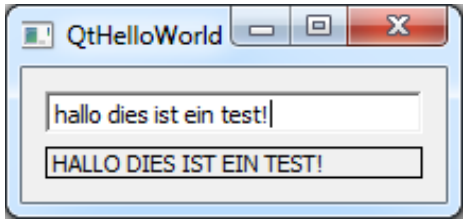
Signals und Slots

- In C++ existiert kein integriertes Event-System.
  - Es gibt keine Delegaten, wie in C#.
  - Stattdessen können aber Zeiger auf Funktionen verwendet werden (Achtung: nicht typsicher!).
- Auch existiert das Schlüsselwort `event` nicht.
  - Man kann aber z.B. das Beobachter-Muster realisieren.
- In Qt wurde ein eigenes Event-System implementiert.
  - Die sog. **Signals und Slots**.
- Ein Signal emittieren ein Ereignis (Event).
  - Ein Slot reagiert auf Ereignisse (Event-Handler).

Beispiel

- Sobald in einem Textfeld etwas eingeben wird, soll der Inhalt in Großbuchstaben in einem Label angezeigt werden.

```
class Dialog : public QWidget
{
   Q_OBJECT

public:
  Dialog();

private:
   QLineEdit *edit;
   QLabel *label;

private slots:
   void doSomething(const QString& text);
};
```



Es wurde ein eigener Slot doSomething() definiert, der ein Objekt vom Typ QString übergeben bekommen soll.

Beispiel

```
Dialog::Dialog() {
    QVBoxLayout *layout = new QVBoxLayout(this);

    edit = new QLineEdit();
    layout->addWidget(edit);

    label = new QLabel();
    label->setStyleSheet("border:1px solid black;");
    layout->addWidget(label);

    QObject::connect(edit, SIGNAL(textChanged(const QString&)),
        this, SLOT(doSomething(const QString&)));
}

void Dialog::doSomething(const QString& text) {
    label->setText(text.toUpper());
}
```

Das Signal des Textfeldes wird mit dem neuen Slot doSomething() des eigenen Objektes verbunden.

Der Text des Textfeldes wird zuerst in Großbuchstaben umgewandelt und dann in das Bezeichnungsfeld geschrieben.

Wir haben heute gelernt…

- welche Bibliotheken in C++ genutzt werden.
- Was die Standardbibliothek in C++ bietet.
- Wie man mit Containern, Iteratoren, und Algorithmen der Standardbibliothek umgeht.
- Was die Klassen vector und map leisten.
- Wie man grafische Oberflächen mit Qt Widgets entwickelt.
- Wie man Layout Container in Qt einsetzt.
- Was Signals und Slots in Qt sind.