Grundkurs C++

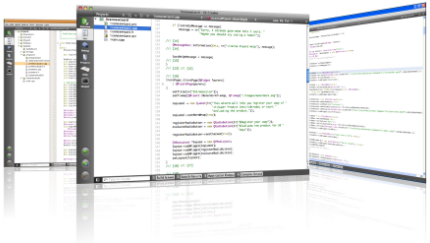
IDE
Klassenhierarchien

Martin Knopp, Martin Gottwald, Stefan Röhrl

18.04.2018

IDE

- Integrated Development Environment
- Wir empfehlen: Qt Creator (Bestandteil des Qt SDK)
- Download unter: https://download.qt.io/archive/qt/5.10/5.10.1/ (Für Windows bitte die `.exe`, nicht die `*-pdb-files-*.zip` herunterladen!)



- Bei Problemen:
  - Bitte nicht zögern, die Tutoren zu fragen!
  - Ausführliche Dokumentation unter http://doc.qt.io/

Namensbereiche (namespaces)

- Vermeidung von Mehrdeutigkeiten
- Klassen und Variablen werden zu logischen Paketen zusammengefasst
- Deklaration/Erweiterung an verschiedenen Stellen in mehreren Dateien möglich

```
namespace mynamespace{}
```

- Einbinden eines Namenbereichs ist global, innerhalb eines Namensbereichs, innerhalb einer Funktion möglich

```
using namespace mynamespace;
```

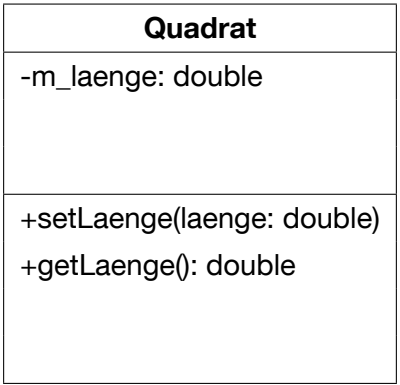
- Einbeziehen einzelner Bestandteile eines Namensbereichs mit Bereichszugriffsoperator möglich

```
mynamespace::fkt1();
```

- Einbinden eines einzelnen Elements:

```
using namespace mynamespace::fkt1();
```

Implementierung einer einfachen Klasse



```
// Quadrat.h
#ifndef QUADRAT_H
#define QUADRAT_H
class Quadrat {
    public:
        Quadrat(); // Konstruktor
        void setLaenge(double laenge); // Methode,
                                       // nur Deklaration
        double getLaenge() const;

    private:
        double m_laenge; // Attribut
};
#endif
```

```cpp
// Quadrat.cpp
#include "Quadrat.h"

Quadrat::Quadrat() {
}

// Implementierung der Funktion setLaenge
void Quadrat::setLaenge (double laenge) {
    m_laenge = laenge;
}

double Quadrat::getLaenge() const {
    return m_laenge;
}
```

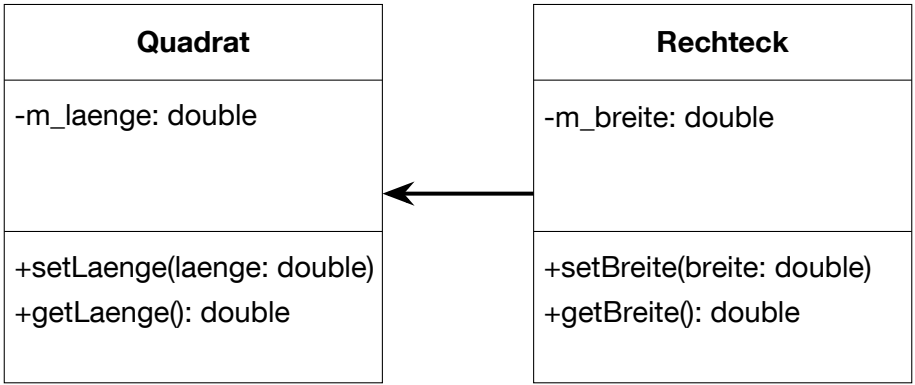
```
// main.cpp
#include "Quadrat.h"

int main () {
    Quadrat meinQuadrat;         // Quadratobjekt erstellen
    Quadrat* meinQuadratZeiger; // Zeiger auf Quadratobjekt
    double laenge;

    meinQuadratZeiger = &meinQuadrat;   // Zeigeradresse setzen
    meinQuadrat.setLaenge(25.0);        // Methode des Objekts aufrufen
    laenge = meinQuadratZeiger->getLaenge(); // Methode über
                                             // Zeiger aufrufen

    return 0;
}
```

Zugriffsregelung und Vererbung



```
class Quadrat {
    public:
        void setLaenge(double laenge); // Methode, nur Deklaration
        double getLaenge () const {    // Methode mit Definition
            return m_laenge; }
    private:
        int m_laenge; // Attribut
};

class Rechteck : public Quadrat { // Vererbung der Klasse Quadrat
    public:
        void setBreite(double breite);
        double getBreite () const { return m_breite; }
    private:
        double m_breite;
};
```

Zugriffsregelungen bei Ableitung: public

```
AK : public BK {
   …
};
```

Basisklasse		abgeleitete Klasse
public	⟹	public
protected	⟹	protected
private	⟹	private*

*Zugriff kann nur über Methoden der Basisklasse erfolgen!

Zugriffsregelungen bei Ableitung: protected

```
AK : protected BK {
   …
};
```

Basisklasse		abgeleitete Klasse
public	⟹	protected
protected	⟹	protected
private	⟹	private*

*Zugriff kann nur über Methoden der Basisklasse erfolgen!

Zugriffsregelungen bei Ableitung: private

```
AK : private BK {
   …
};
```

```
AK : BK {
   …
};
```

Standard!

Basisklasse		abgeleitete Klasse
public	⟹	private
protected	⟹	private
private	⟹	private*

*Zugriff kann nur über Methoden der Basisklasse erfolgen!

new und delete

```
// main.cpp
#include "Quadrat.h"

int main () {
    Quadrat meinQuadrat; // Objekt auf Stack

    Quadrat* meinQuadratDyn; // Zeiger auf Quadrat-Objekt
    meinQuadratDyn = new Quadrat; // neues Objekt auf
                                  // dem Heap erstellen

    delete meinQuadratDyn; // Quadrat-Objekt löschen,
                           // Speicher freigeben
}
```

Konstruktor, Destruktor und this-Zeiger

```
class A {
    public:
        A (int c = 1000, int d = 2000) {
            this->c = c;
            this->d = d; } // Konstruktor
        ~A(); // Destruktor
        void getZustand();
    private:
        int c, d;
};

int main() {
    A neuesObjekt();
    A neuesObjekct2(200, 300);
}
```

Initialisierungslisten

```
class A {
    public:
        A (int k = 1000, int l = 2000) : c(k), d(l) {}
        void getZustand();

    private:
        int c;
        int d;
};
```

Konstruktor und Ableitung

```
class A {
    public:
        A (int k=1, int l=2) : c(k), d(l) { std::cout << "A!" << '\n'; }
    private:
        int c, d;
};
class B : A {
    public:
        B (int m=3) : e(m) { std::cout << "B!" << '\n'; }
    private:
        int e;
};
int main() {
    B b;
}
```

Konstruktor und Ableitung

```
class A {
    public:
        A (int k=1, int l=2) : c(k), d(l) { std::cout << "A!" << '\n'; }
    private:
        int c, d;
};
class B : A {
    public:
        B (int m=3) : e(m) { std::cout << "B!" << '\n'; }
    private:
        int e;
};
int main() {
    B b;
}
```

Ausgabe:
A!
B!

Abgeleitete Klasse ruft Standardkonstruktor der Basisklasse auf!

Aufrufreihenfolge von Konstruktoren

- Konstruktoren der Basisklassen(n)
- Konstruktoren der Elementobjekt-Klassen
- Konstruktoren der abgeleiteten Klasse

Initialisierungslisten II

```
class A {
    public:
        A (int k=1, int l=2) : c(k), d(l) {
            std::cout << "A!" << '\n'; }
    private:
        int c, d;
};

class B : A {
    public:
        B (int m=3, int n=6, int o=9) : A(n,o), e(m) {
            std::cout << "B!" << '\n'; }
    private:
        int e;
};
```

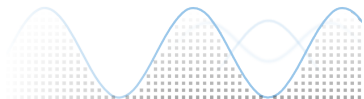
expliziter Aufruf des Konstruktors der Basisklasse
Reihenfolge bleibt unverändert!

Verdeckung

```
class A {
    protected:
        int c, d;
    public:
        A() : c(1),d(2) {}
        void getZustand(){ std::cout << "A: " << c << " " << d << "\n"; }
};

class B : A {
    private:
        int d;
    public:
        B(): d(3) {}
        void getZustand(){ std::cout << "B: " << c << " " << d << "\n"; }
};

int main() {
    B b;
    b.getZustand();
}
```



Verdeckung

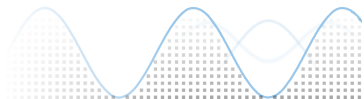
```
class A {
    protected:
        int c, d;
    public:
        A() : c(1),d(2) {}
        void getZustand(){ std::cout << "A: " << c << " " << d << "\n"; }
};

class B : A {
    private:
        int d;
    public:
        B(): d(3) {}
        void getZustand(){ std::cout << "B: " << c << " " << d << "\n"; }
};

int main() {
    B b;
    b.getZustand();
}
```

Ausgabe:

```
B: 1 3
```



Bereichszugriffsoperator

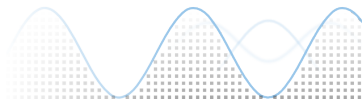
```
class A {
    protected:
        int c, d;
    public:
        A() : c(1),d(2) {}
        void getZustand(){ std::cout << "A: " << c << " " << d << "\n"; }
};

class B : A {
    private:
        int d;
    public:
        B(): d(3) {}
        void getZustand(){ std::cout << "B: " << c << " " << A::d << "\n"; }
};

int main() {
    B b;
    b.getZustand();
}
```

Ausgabe:

```
B: 1 2
```



Mehrfachvererbung

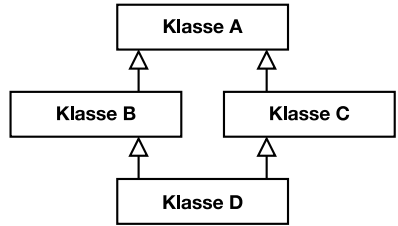
```
class A {
    public:
        A(int x=1000, int y=2000): i(x), j(y) { cout << "A (constr.): i=" << i << ", j=" << j << "\n"; }
        void getZustand() { cout << "A: i=" << i << ", j=" << j << "\n"; }
    private:
        int i, j;
};

class B : public A {
    public:
        B(int x=101, int y=102, int z=103): A(x, y), k(z) {
            cout << "B (constr.): "; getZustand(); }
    private:
        int k;
};

class C : public A {
    public:
        C(int x=201, int y=202, int z=203): A(x, y), l(z) {
            cout << "C (constr.): "; getZustand(); }
    private:
        int l;
};

class D : public B, public C {
    public:
        D(int x=301, int y=302, int z=303): B(x, y), C(x+10,y+10), m(z) {}
    private:
        int m;
};

int main() { D instanz; }
```



Attribute der Klasse A 2× in D, unveränderte Reihenfolge der Konstruktoraufrufe

Mehrfachvererbung

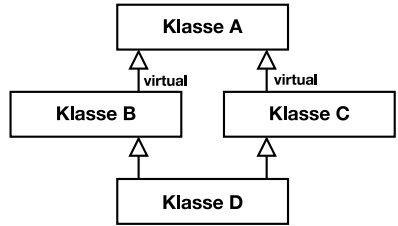
```
class A {
    public:
        A(int x=1000, int y=2000): i(x), j(y) { cout << "A (constr.): i=" << i << ", j=" << j << "\n"; }
        void getZustand() { cout << "A: i=" << i << ", j=" << j << "\n"; }
    private:
        int i, j;
};

class B : virtual public A {
    public:
        B(int x=101, int y=102, int z=103): A(x, y), k(z) {
            cout << "B (constr.): "; getZustand(); }
    private:
        int k;
};

class C : virtual public A {
    public:
        C(int x=201, int y=202, int z=203): A(x, y), l(z) {
            cout << "C (constr.): "; getZustand(); }
    private:
        int l;
};

class D : public B, public C {
    public:
        D(int x=301, int y=302, int z=303): B(x, y), C(x+10,y+10), m(z) {}
    private:
        int m;
};

int main() { D instanz; }
```



Attribute der Klasse A nur noch einmal in D!

Konstruktoraufrufe bei virtuellen geerbten Klassen

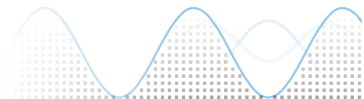
Reihenfolge der Konstruktoraufrufe:

- Konstruktoren der virtuellen (geerbten) Basisklasse(n)
- Konstruktoren der nicht-virtuellen Basisklasse(n)
- Konstruktoren der Elementobjekt-Klassen
- Konstruktoren der abgeleiteten Klasse

Der Konstruktor der „am weitesten abgeleiteten Klasse“ übernimmt die Initialisierung des virtuell geerbten Teiles der Klasse. Die anderen Aufrufe des Basisklassenkonstruktors der virtuell geerbten Klasse werden ignoriert.

Zeiger auf die Basisklasse

```
class Auto {
    protected:
        int ps, zylinder;
    public:
        Auto(int lei, int toepf): ps(lei),zylinder(toepf) {}
        void bremsen() { std::cout << "BREMSE..." << "\n"; }
};
class Cabrio: public Auto {
        int offen;
    public:
        Cabrio(int lei, int toepf): Auto(lei,toepf), offen(0) {}
        void dachAuf() {
            std::cout << "AUF!" << "\n";
            offen = 1; }
        void bremsen() { std::cout << "BREMSE Cabrio..." << "\n"; }
};
int main() {
    Auto *z4 = new Cabrio(215,6);
    z4->bremsen();
    ((Cabrio*)z4)->bremsen();
    ((Cabrio*)z4)->dachAuf(); }
```



Zeiger auf die Basisklasse

```
class Auto {
    protected:
        int ps, zylinder;
    public:
        Auto(int lei, int toepf): ps(lei),zylinder(toepf) {}
        void bremsen() { std::cout << "BREMSE..." << "\n"; }
};
class Cabrio: public Auto {
        int offen;
    public:
        Cabrio(int lei, int toepf): Auto(lei,toepf), offen(0) {}
        void dachAuf() {
            std::cout << "AUF!" << "\n";
            offen = 1; }
        void bremsen() { std::cout << "BREMSE Cabrio..." << "\n"; }
};
int main() {
    Auto *z4 = new Cabrio(215,6);
    z4->bremsen();
    ((Cabrio*)z4)->bremsen();
    ((Cabrio*)z4)->dachAuf(); }
```

Ausgabe:

```
BREMSE...
BREMSE Cabrio...
AUF!
```

Zugriff durch Basisklassenzeiger

```
int main() {
    Auto *parkpl[3] = { new Cabrio(215,6),
        new Lkw(600,16),
        new Jeep(400,12) };

    Auto *auswahl = parkpl[i%3];
}
```

Jede Spezialisierung von Auto (Cabrio, Lkw, Jeep) habe eine eigene Methode bremsen()

Ist der Zeiger vom Basistyp Auto, wird die Methode von Auto aufgerufen, nicht die der Spezialisierungen.

Polymorphismus

```
class Auto { ...
    virtual void bremsen() { std::cout << "BREMSE..." << "\n"; }
};
class Cabrio : public Auto { ...
    void bremsen() { std::cout << "BREMSE Cabrio..." << "\n"; }
};
class Lkw : public Auto { ...
    void bremsen() { std::cout << "BREMSE LKW..." << "\n"; }
};
```

Zeigt der Zeiger `auswahl` vom Typ `Auto*` auf ein abgeleitetes Objekt, so bestimmt nun beim Aufruf `auswahl->bremsen()` der Typ des Objekts, welche Variante von `bremsen()` zum Einsatz kommt.
Dies geschieht zur Laufzeit („late binding“).

Destruktoren sollten immer virtuell angelegt werden!

Polymorphismus II – abstrakte Klassen

```
class Auto { ...
    virtual void bremsen()=0;
};
class Cabrio : public Auto { ...
    void bremsen() { std::cout << "BREMSE Cabrio..." << "\n"; }
};
class Lkw : public Auto { ...
    void bremsen() { std::cout << "BREMSE LKW..." << "\n"; }
};
```

- Die Klasse Auto kann nicht instanziiert werden
- Abstrakte Klassen können nicht als Typ für Funktionsparameter oder Rückgabewerte verwendet werden
- Abgeleitete Klassen müssen die rein virtuellen Methoden neu definieren, um instanziiert werden zu können.

Überladen von Funktionen

```
int wurzel (int i);
float wurzel (float f);
double wurzel (double f);

void main () {
    int ia, ib;
    float fa, fb;
    ia = wurzel (ib);
    fa = wurzel (fb);
}
```

- Funktionen müssen sich in Anzahl und/oder Typ der Parameter unterscheiden, d.h. sie müssen eine andere Signatur aufweisen.
- Unterscheidung in Rückgabewert oder in den Namen der Parameter reicht nicht aus.

Überladen von Funktionen II

- Bei Vererbung beachten: gleichnamige Methode in der abgeleiteten Klasse verdeckt alle gleichnamigen Methoden in der Basisklasse, unabhängig von der Signatur
- impliziter Aufruf von Basisklassenkonstruktor nur möglich, wenn ein Standardkonstruktor (Aufruf ohne Parameter) existiert
- virtuelle Methoden: Virtualität nur dann, wenn die Methode in der Basisklasse und der abgeleiteten Klasse dieselbe Signatur hat; ansonsten einfache Verdeckung

Überladen von Funktionen

```
class Quadrat {
    public:
        Quadrat ( double laenge );
        Quadrat ( const Quadrat &q );
        double m_laenge;
        char *bezeichnung;
};

Quadrat::Quadrat ( const Quadrat &q ) {
    m_laenge = q.m_laenge;
    // Kopieren des Dateinamens
    bezeichnung = new char[strlen(q.bezeichnung) +1];
    strcpy ( bezeichnung, q.bezeichnung );
}
```

Überladen von Operatoren

```
class MeinString {
    public:
        char *inhalt;
        int laenge;
        void operator= ( const MeinString &zweiterString );
        int operator< ( const MeinString &zweiterString ) const;
    ...
};

void MeinString::operator= ( const MeinString &zweiterString ) {
    laenge = zweiterString.laenge;
    delete [] inhalt;
    inhalt = new char [ laenge+1 ];
    strcpy ( inhalt, zweiterString.inhalt );
}
```

Templates: Funktionstemplates

```
template <class T>
T max (T a1, T a2) {
    if (a1 < a2)
        return a2;
    else
        return a1;
}
```

- kann auf beliebigen Datentypen/Klassen arbeiten
- nur eine Implementierung notwendig
- mehrere Platzhalter sind möglich (getrennt durch Kommata)
- konkrete Variante für bestimmten Datentyp wird erst erstellt, wenn die Funktion damit verwendet wird
- Trennung von Definition und Deklaration von daher nicht möglich

Templates: Funktionstemplates

```
template <class T>
T max (T a1, T a2) {
    if (a1 < a2)
        return a2;
    else
        return a1;
}
```

```
int main () {
    std::cout << max(3, 5);
    std::cout << max<int>(3, 5);
}
```

- kann auf beliebigen Datentypen/Klassen arbeiten
- nur eine Implementierung notwendig
- mehrere Platzhalter sind möglich (getrennt durch Kommata)
- konkrete Variante für bestimmten Datentyp wird erst erstellt, wenn die Funktion damit verwendet wird
- Trennung von Definition und Deklaration von daher nicht möglich

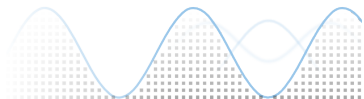
Templates: Klassentemplates

```
#ifndef CPPKURSTEM_H
#define CPPKURSTEM_H

namespace cppKurs {
    template <class T> class tupleClass {
        private:
            T array[2];
        public:
            void setVals(T a, T b);
            T show();
    };

    template <class T> void tupleClass<T>::setVals(T a, T b) {
        array[0] = a;
        array[1] = b;
    }

    template <class T> T tupleClass<T>::show() {
        std::cout << array[0] << ", " << array[1] << std::endl;
        return array[0] + array[1];
    }
}
#endif
```



Hausaufgabe

- heute Abend im Moodle-Kurs
- Abgabetermin: 25. April 2018, 15:00 Uhr