Fachhochschule Köln
Cologne University of Applied Sciences

Angewandte Mathematik und Programmierung

Einführung in das Konzept der objektorientierten Anwendungen zu mathematischen Rechnens

SS2013

Fomuso Ekellem

Inhalt

- **Übung(Aufklärung)**
- **Polymorphismus (Untergliederung von Vererbung)**
  - Standardkonversionsionen bei abgeleiteten Klassen
  - Virtuelle Funktionen
  - „pure virtual“- Funktionen und abstrakte Klassen
  - Komplexes Pragrammbeispiel zum Einsatz von Vererbung
- **Templates**

Polymorphie

- Polymorphie bedeutet „Vielgestaltigkeit“. Der Begriff Polymorphie ist eng mit der Vererbung verbunden. Er bezeichnet die Tatsache, das eine Methode in verschiedenen abgeleiteten Klassen einer Hierarchie unterschiedlich implementiert werden kann.
- So ist es möglich, dass verschiedene Klassen gleichlautende Methoden enthalten, die aber nicht den gleichen Inhalt haben.
- Verschiedene Objekte werden gleich behandelt, reagieren aber unterschiedlich.
- Sie können gleich behandelt werden, weil sie zum Teil gleiche Schnittstellen besitzen (gleichlautende Datenelemente und Methoden).

Polymorphie

- Methoden, die in mehreren Klassen benötigt werden, werden in einer Basisklasse deklariert. In den abgeleiteten Klassen werden sie dann überschrieben, um eine klassenspezifische Ausführung zu erreichen.
- Methoden, die überschrieben werden sollen, werden in der Basisklasse mit dem Schlüsselwort virtual deklariert, um sicher zu stellen, dass beim Aufruf auch wirklich die überschriebenen Methoden aufgerufen werden und nicht die der Basisklasse.
- Es ist üblich auch die überschriebenen Methoden in den abgeleiteten Klassen mit virtual zu kennzeichnen.
- *Siehe Beispiel…*

Polymorphie und Methoden überschreiben

```
class Auto
{
public:
  virtual void marke() = 0;
};
```

```
class DeutschesAuto : public Auto{
public:
  virtual void marke(){cout << "VW" << endl;}
  void fahre250() {cout << "geht nicht" << endl;}
};
```

```
class Audi : public DeutschesAuto{
public:
virtual void marke(){ cout << "Audi" << endl;}
};
```

```
class Mercedes : public DeutchesAuto{
public:
virtual void marke(){ cout << "Mercesdes" << endl;}
void fahre250(){ cout << "sicher!" << endl;}
};
```

Identischer Methoden kopf heißt, dass sowohl Sichtbarkeit, Rückgabewert, Methodenname und die Datentypen der Methodenparameter gleich sein müssen.

Beim Auto-Beispiel wird die Methode marke von Auto in DeutschesAuto überschrieben, welche dann wiederum in den Klassen Audi und Mercedes überschrieben werden.

fahre250 ist zum ersten mal in Deutsches Auto definiert und wird in der Klasse Mercedes überschrieben, nicht aber in der Klasse Audi., und ist nicht virtuell.

Polymorphie

```
// Zuweisung von Objekten in Main
Mercedes m;
DeutschesAuto da;
da.marke(); // VW
da.fahre250(); // geht nicht
m.marke(); // Mercedes
m.fahre250(); // sicher!
```

```
// Zuweisung von Zeiger: in  Main
Mercedes m;
m.marke(); // Mercedes
m.fahre250(); // sicher!
DeutchesAuto *da = &m;
da->marke(); // Mercedes
da->fahre250(); // geht nicht
```

- Bei Polymorphismus spricht man auch oft von early binding und late binding
  - Early binding: bereits während des Kompilierens weiss man, welche Methode ausgeführt werden wird (statischer Typ): da.marke(); //VW
  - Late binding: erst zur Laufzeit kennt man den dynamischen Typ und dadurch die entsprechende Methode: da->marke(); //Mercedes
- Der Zusammenhang zwischen Polymorphismus und Methoden berschreiben ist ein wichtiger Aspekt von objektorientierten Programmiersprache
  - In Java wird immer die Methode des dynamischen Typs angewendet
  - In C++ wird das Verhalten durch den Programmierer bestimmt

Polymorphie und Methoden überschreiben

- Polymorphie ist i. A. nur dann sinnvoll, wenn zwischen den Objekten der Ober- und Unterklasse eine **IS-A-Beziehung** besteht.
- Methoden, die in mehreren Klassen benötigt werden, werden in einer Basisklasse deklariert. In den abgeleiteten Klassen werden sie dann überschrieben, um eine klassenspezifische Ausführung zu erreichen.
- Methoden, die überschrieben werden sollen, werden in der Basisklasse mit dem Schlüsselwort virtual deklariert, um sicher zu stellen, dass beim Aufruf auch wirklich die überschriebenen Methoden aufgerufen werden und nicht die der Basisklasse.
- Es ist üblich auch die überschriebenen Methoden in den abgeleiteten Klassen mit virtual zu kennzeichnen.
- ***Siehe Beispiele (Virtual und Polymorphie)***

Polymorphie und Methoden überschreiben

Überschreiben ist nicht Überladen

- Überladen von Methoden
  - Verschiedene Methoden in einer Klasse (diese können auch von einer Oberklasse geerbt werden) besitzen den gleichen Namen.
  - Die Signatur der Methoden muss eindeutig sein.
- Überschreiben von Methoden
  - Neuimplementierung einer Methode in einer Unterklasse. Die neue Methode besitzt insbesondere die gleiche Signatur (Methoden kopf) wie eine Methode der Oberklasse.

- **Wird eine Methode überschrieben, ist die Methode der Superklasse in einer Instanz der Subklasse nicht mehr sichtbar und es wird die „neue“ Version der Methode verwendet:**

Abstrakte Klassen und Methoden

- **Virtuelle Methoden** können zusätzlich auch noch **abstrakt** sein.
- Eine abstrakte Klasse in C++, ist eine Klasse, die mindestens eine abstrakte Methode enthält.
  - Von abstrakten Klassen können keine Objekte erzeugt werden.
  - Eine abstrakte Methode in C++ ist virtuell und dadurch gekennzeichnet, dass ihre Deklaration durch " = 0; „ abgeschlossen wird (rein Virtuellen Methoden).
- Eine abstrakte Methode kann jedoch außerhalb der Klasse wie üblich noch implementiert sein.

Fahrzeug.h

```
class fahrzeug
 {
   public:
   virtual void fahren() = 0;
 };
```

```
#include "fahrzeug.h"
#include <iostream>
class automobil : public fahrzeug
{
public:
        void fahren()
        {
        std::cout << "Brummbrumm" << std::endl;
        }
};
```

Abstrakte Klassen und Methoden

- Sinn abstrakter Klassen ist, dass ihre abstrakten Methoden in abgeleiteten Klassen reimplementiert werden müssen, um dort eine Instanzenbildung zuzulassen.
- Abstrakte Klassen verwendet man auch, um Schnittstellen in C++ zu definieren.

Schnittstellen (Bitte beachte)

- In Java dürfen Klassen nur eine Superklasse besitzen, das bedeutet, daß die von C++ bekannte **Mehrfachvererbung** nicht möglich ist. Statt dessen kennt Java die sogenannten Interfaces. Das sind reine Schnittstellen, die keinerlei Implementierungen enthalten.

Abstrakte Klassen und Methoden

- **C++**
  - Abstrakte Klassen besitzen mindestens eine abstrakte Methode.
  - Abstrakte Methoden können einen Rumpf besitzen.
- Java
  - Abstrakte Klassen werden durch das Schlüsselwort abstract gekennzeichnet.
  - Abstrakte Methoden besitzen keinen Rumpf.

- Empfehlung für abstrakte Klasse in C++
  - Der Destruktor sollte abstrakt deklariert werden.

Templates

- Programmiersprachen sollten elegante Mechanismen besitzen, um Redundanz beim Programmieren vermeiden zu können.
- Angenommen, wir wollen Listen-, Array, oder Matrizen-Klassen implementieren, die verschiedene Datentypen als Inhalt aufnehmen können(z.B. int, double, bool, usw…). Alle Operationen bleiben gleich:

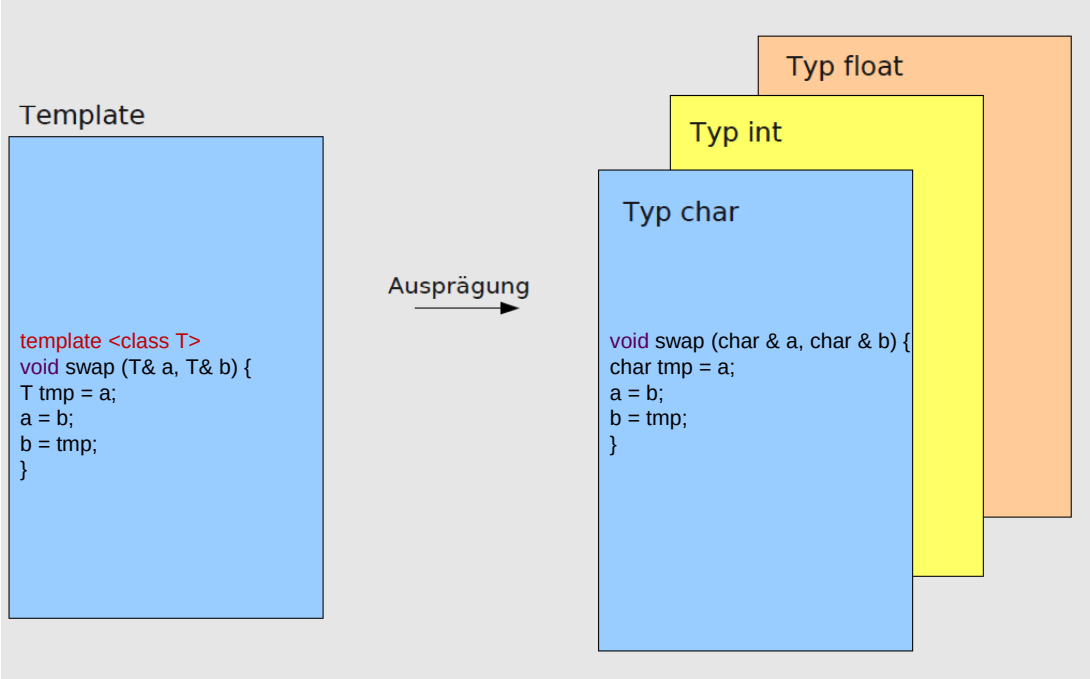
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \begin{bmatrix} 1.3 & 2.5 & 4.3 \\ 3.4 & 5.2 & 6.3 \\ 1.2 & 4.5 & 7.2 \end{bmatrix} \quad \begin{bmatrix} \text{true} & \text{false} & \text{true} \\ \text{true} & \text{true} & \text{true} \\ \text{false} & \text{true} & \text{false} \end{bmatrix}$$

- In C++ wird dieses Konzept “Template” genannt.
- Eine andere Bezeichnung (z.B. in Java 1.5) ist “Generic”.

Templates

- **Motivation:** Eine Klasse erstellen für verschiedene Datentypen
  - Datenrepräsentation: Matrix, Stacks, Listen, Bäume...
- Code für einen Datentyp sollte für kompatible Typen wieder verwendbar sein.
- C++ bietet die Möglichkeit, mit Hilfe von Templates (Schablonen, Vorlagen) eine parametrisierte Familie verwandter **Funktionen** oder **Klasse** zu definieren
  - Funktions-Templates legen die Anweisung einer Funktion fest, wobei statt eines konkreten Typs ein Parameter T gesetzt wird
  - Klassen-Templates legen die Definition einer Klasse fest, wobei statt eines Typs ein Parameter T eingesetzt wird.
- **Vorteile**
  - Ein Template muss nur einmal codiert werden. Einzelne Klassen oder Funktionen werden automatisch erzeugt.
  - Einheitliche Lösung für gleichartige Probleme, wobei man unabhängige Bestandteile frühzeitig testen kann

Templates


Template
template <class T>
void swap (T& a, T& b) {
T tmp = a;
a = b;
b = tmp;
}
Ausprägung
Typ float
Typ int
Typ char
void swap (char & a, char & b) {
char tmp = a;
a = b;
b = tmp;
}

Templates

Funktions-Templates

- **Beispiel**: swap()-Funktion
- Verhalten ist für jeden Typ gleich (elementar und andere)
- **Ineffizient**: schreibe für jeden Typ eine eigene überladene Funktion
- **Lösung**: definiere eine Schablone für die Funktion

swap() für Typ <int>

```
void swap (int& a, int& b) {
int tmp = a;
a = b;
b = tmp;
}
```

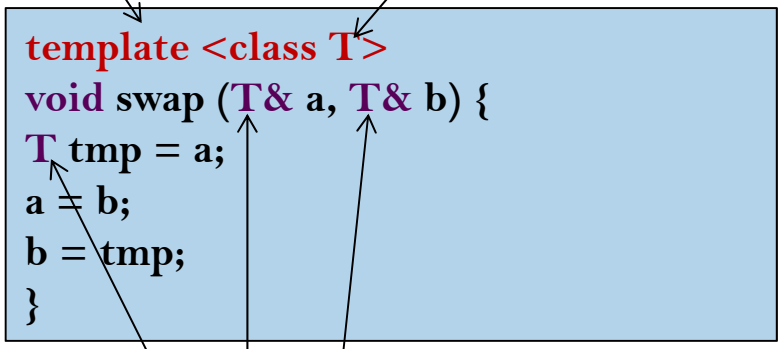
swap() für beliebigen Typ <T>

```
template <class T>
void swap (T& a, T& b) {
T tmp = a;
a = b;
b = tmp;
}
```

Templates

Die template <...> Zeile sagt dem Compiler: „die folgende Deklaration o. Definition ist als Schablone zu verwenden“.

Ein Template erwartet als Argumente Platzhalter für Typen: Template-Parameter



Die Platzhalter innerhalb des Template werden später (noch vor der Übersetzung in Maschinensprache) durch spezifische Typen ersetzt

Templates

- Templates sollen den Aufwand für den Entwickler reduzieren:
  - Templates werden vom Kompiler nach Bedarf in eine normale Funktion/Klasse gewandelt.
  - Wird swap für 2 int-Werte verwendet, so wird eine weitere Funktion erzeugt.
  - Wird swap für 2 Auto-Werte verwendet, so wird eine weitere Funktion erzeugt.
  - Beim Kompilieren findet auch die Typprüfung statt.
- **Syntax für explizite Spezialisierung**

```
double d1 = 4.0, d2 = 2.0;
swap<double>(d1, d2);
```

- **Explizit bedeutet:** der Programmierer spezifiziert ausdrücklich, mit welchem Typ die Templateparameter zu ersetzen sind.

Templates

- **Implizite Spezialisierung**

```
double d1 = 4.0, d2 = 2.0;
//Alles klar: T = double
swap(d1, d2);
```

```
double d1 = 4.0;
float f1 = 10.0f;
swap(d1, f1);
```

Das geht nicht ! Compiler meldet:
no matching function for call to
´swap(double&, float&)´

```
swap<double>(d1, f1);
```

Implizite Typkonvertierung von <f1>

```
swap(d1, double(f1));
```

Explizite Typkonvertierung von <f1>

Templates

- In vielen Anwendungen ist es nicht nur notwendig eine Funktion, sondern eine ganze Klasse mit einem Typ zu parametrisieren. Z.B.:
  - Implementierung von elementaren Datenstrukturen wie Stack, Queue, binäre Suchbäume,usw.
  - Stack Beispiel

```
template <class T>
class Stack {
        private:
        T* inhalt;              // Datenbereich des Stacks
        int index, size;        // Aktueller Index, Grösse des Stacks
        public:
        Stack(int s): index(-1), size(s) { // Constructor
        inhalt = new T[size];  // Stack als Array implementiert
        }
        void push(T item) {     // Ein Element auf den Stack "pushen"
        if (index < (size - 1)) {
        inhalt[++index] = item;
        }
        }
        T top() const {         // Ein Element vom Stack lesen
        if (index >= 0) {return inhalt[index];}
        }
        void pop() {            // Ein Element auf dem Stack löschen
        if (index >= 0) {--index;}
        }
};
```

Templates

- **Gebrauch der Template-Klasse Stack**

```
int main() {
Stack<int> intStack(100);                 // Stack für 100 int
Stack<double> doubleStack(250);           // Stack für 250 double
Stack<rect> rectStack(50);                // Stack für 50 rect
intStack.push(7);
doubleStack.push(3.14);
rectStack.push(rect(2,5));
}
```

- Bei der Allozierung eines Stacks muss explizit der Typ angegeben werden(<int>, <rect>, ...)
- Auch hier gilt: die entsprechende Klasse wird vom Compiler bei Bedarf aus der Template-Klasse generiert.

Templates

- Nochmals die Template-Klasse Stack, diesmal aber mit Trennung von Deklaration und Definition:

Header-Datei: stack.h

```
template <class T>
class Stack {
        private:
          T* inhalt;
          int index;
          int size;
        public:
          Stack(int s);
          void push(Type item);
          T top() const;
          void pop();
};
```

Definition: stack.cpp

```
#include "stack.h"
using namespace std;
// Achtung: nicht Stack::Stack(int s)!
template <class T>
Stack<T>::Stack(int s):index(-1),size(s)
{
inhalt = new T[size];
}
template <class T>
void Stack<T>::push(T item) {
if(index<size) {inhalt[++index]=item;}
}
template <class T>
T Stack<T>::top() const {
if (index>=0) {return inhalt[index];}
}
template <class T>
void Stack<Type>::pop() {
if (index >= 0) {index--;}
}
```

Templates

- **Gebrauch!!!**

```
#include "stack.cpp"
using namespace std;
int main() {
Stack<int> intStack(100);
Stack<double> doubleStack(250);
intStack.push(7);
doubleStack.push(3.14);
}
```

- Man beachte: stack.cpp, nicht stack.h!!!
- Bei Templates muss die komplette Implementierung eingebunden werden, weil der Compiler die Klasse bei Bedarf erzeugt (Spezifikation allein reicht also nicht)
- Dies gilt auch für Template Funktionen; Die Funktionsdeklaration allein reicht nicht.

Templates

- Die Angabe des Datentyps beim Gebrauch eines Templates bestimmt, wofür eine Instanz des Templates gebraucht werden kann:

```
Stack<Person> personStack(100);          // Nur für Typ Person verwendbar
```

- **Ausnahme:** auch Subklassen von Person können auf diesem Stack abgelegt werden.
- Dabei werden die Objekte aber in Person konvertiert...
  - wodurch die in Subklassen spezifizierte Funktionalität (und damit auch Polymorphismus) verloren geht.
- **Besser:** man verwendet einen Zeiger auf die Basisklasse:

- **Vorteile:** Polymorphismus funktioniert; Subklassen von Person können auf dem Stack

```
Stack<Person*> personStack(100);         // Für alle verwendbar
```

Templates

- **Templates von Templates:** Templates können als Parameter für andere Templates dienen:

```
complex<float> c1, c2;
swap<complex<float> >(c1, c2);
```

- complex<T> ist eine Klasse der C++ Standard Bibl. für Komplexe Zahlen (#include <complex>)
- **Achtung:** Bei geschachtelten Templates muss man aufeinander folgende '>' durch Leerzeichen trennen, da sie sonst als shift-Operator missinterpretiert werden.

Templates

- Ein Template existiert nicht als Typ oder Objekt.
- Erst bei Instanziierung eines Template entsteht eine neuer Typ (Funktion/Klasse) bei dem die Template-Parameter durch konkrete Typen ersetzt wurden.
- **Ohne Templates:**
  - werden Interface und Implementierung in separaten Dateien untergebracht (.h, .cpp)
  - stellt der Linker die Verbindung zwischen dem Aufruf einer deklarierten Methode und dem zusätzlichen Maschinen-Code her.
- **Mit Templates:**
  - wird Code erst bei Bedarf generiert. Bedarf heißt, z.B. durch Bindung an spezifische Template-Parameter
  - kann nur der vom Template generierte Code übersetzt und gelinkt werden.