Exercise Week 12

GianAndrea Müller
mailto:muellegi@student.ethz

May 30, 2018

Exercise Week 12

GianAndrea Müller
mailto:muellegi@student.ethz

May 30, 2018

Time Schedule

- 10' Wiederholung Klassen
- 10' Vererbung
- 10' Polymorphie und Abstrakte Basisklassen
- 10' Konstruktoren
- 10' Has a - is a

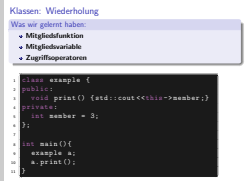
Time Schedule

- 10' Wiederholung Klassen
- 10' Vererbung
- 10' Polymorphie und Abstrakte Basisklassen
- 10' Konstruktoren
- 10' Has a - is a

Learning Objectives
• Verständnis: Vererbung
• Andwendung: Polymorphismus

Learning Objectives

- Verständnis: Vererbung
- Andwendung: Polymorphismus



- Eine **Mitgliedsfunktion** ist eine in der Klasse definierte Funktion, die direkten Zugriff auf oeffentliche und private Mitgliedsvariablen hat.
- Eine **Mitgliedsvariable** ist eine in der Klasse definierte Variable.
- Die Zugriffsoperatoren erlauben den Zugriff auf oeffentliche Mitglieder einer Instanz.

Klassen: Wiederholung

Was wir gelernt haben:

- **Mitgliedsfunktion**
- **Mitgliedsvariable**
- **Zugriffsoperatoren**

```
1  class example {
2  public:
3    void print() {std::cout<<this->member;}
4  private:
5    int member = 3;
6  };
7
8  int main(){
9    example a;
10   a.print();
11 }
```

Klassen: Wiederholung

Was wir gelernt haben:
- Zugriffsbereiche
- Zugriffsmethoden

- Es gibt den oeffentlichen und den privaten Zugriffsbereich. Oeffentlich heisst: Nicht verfuegbar von aussen, ueber Zugriffsoperatoren. Nur ueber Mitgliedsfunktionen erreichbar.
- Zugriffsmethoden erlauben ein klar definiertes Interface zu den privaten Mitgliedsvariablen. Sie koennen beispielsweise ueberpruefen ob ein gueltiger Wert zugewiesen wird.

Klassen: Wiederholung

Was wir gelernt haben:

- **Zugriffsbereiche**
- **Zugriffsmethoden**

```
1 class example {
2 private:
3   int variable;
4 public:
5   int get_variable();
6   void set_variable(int value);
7 };
```

Klassen: Wiederholung
Was wir gelernt haben:
Konstruktor
Kopierkonstruktor
Zuweisungsoperator
Destruktor
Regel der Drei

Klassen: Wiederholung

- Alle diese Mitgliedsfunktionen existieren standardmaessig fuer jede Klasse und muessen fuer dynamische Datenstrukturen angepasst werden.
- Der Konstruktor ist eine Mitgliedsfunktion, die den Namen der Klasse traegt. Sie wird IMMER wenn eine Instanz der Klasse generiert wird automatisch aufgerufen, kann ueberladen werden und wird, wenn eine entsprechende Ueberladung vorhanden ist auch fuer implizite Typenumwandlungen verwendet (automatisch).
- Der Kopierkonstruktor wird dann aufgerufen, wenn eine Instanz einer Klasse eine Kopie einer anderen Instanz derselben Klasse werden soll.
- Der Zuweisungsoperator muss ueberladen werden, falls eine Klasse dynamische Mitgliedsvariablen hatt, damit eine tiefe Kopie der Instanz erzeugt wird.
- Der Destruktor ist eine Mitgliedsfunktion, die den Namen der Klasse mit einem vorgeschalteten Tilde traegt. Sie wird immer automatisch aufgerufen, wenn eine Instanz ihren Gueltigkeitsbereich verlaesst und geloescht werden muss.

Was wir gelernt haben:

- **Konstruktor**
- **Kopierkonstruktor**
- **Zuweisungsoperator**
- **Destruktor**

Regel der Drei

Klassen: Wiederholung
Was wir gelernt haben:
Operatoren
Erkenntnisse
Für neue Datenstrukturen brauchen wir neue Operatorüberladungen.
Operatoren als Mitgliedsfunktionen
Operatoren als globale Überladungen
Implizite Typenumwandlung

Klassen: Wiederholung

- Die Operatoren, die fuer einfache Datentypen standardmaessig verfuegbar sind, muessen fuer neu eingefuehrte Datentypen ueberladen werden, damit sie wieder Nutzbar sind.
- Dabei hat man die Moeglichkeit Operatoren als Mitgliedsfunktionen zu definieren, damit der Zugriff auf private Mitglieder gegeben ist.
- Um automatische implizite Typenkonversion zu erlauben, koennen Operatoren auch global ueberladen werden (nicht als Mitgliedsfunktion). Dann wird der operator, wenn er eine Kombination von Argumenten mit verschiedenen Typen hat, versuchen diese entsprechend zu konvertieren.
- Ein gutes Beispiel ist die Definition des $+=$ Operators als Mitgliedsfunktion, damit der Zugriff auf Mitglieder gegeben ist. Dann kann der $+$ Operator global definiert werden, und innerhalb auf die Definition von $+=$ zurueckgegriffen werden. So nutzt man effektiv die Vorteile beider Implementationen und muss weiterhin bei einer Anpassung der Mitglieder nur einen der beiden anpassen.

Was wir gelernt haben:

- **Operatoren**

Erkenntnisse

- Für neue Datenstrukturen brauchen wir neue Operatorüberladungen.
- Operatoren als Mitgliedsfunktionen
- Operatoren als globale Überladungen
- Implizite Typenumwandlung

Vererbung: Grundlagen

Vererbung: Grundlagen

- Vererbung erlaubt alle Mitglieder einer Basisklasse in einer abgeleiteten Klasse zu uebernehmen und weitere hinzuzufuegen. So kann einmal geschriebener code mehrfach verwendet werden. Weiter kann eine ganze Familie von Klassen die ihre grundlegenden Eigenschaften identisch Teilen. Die selben grundlegenden Eigenschaften (Basisklasse) koennen dann auf fuer den ganzen Vererbungsbaum auf einen Schlag angepasst werden.

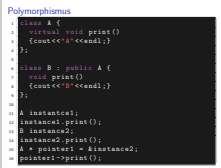
```
1  class A{
2    ... //Basisklasse
3  };
4
5  class B: public A{
6    ... //Abgeleitete Klasse
7  };
8
9  class C: public B{
10   ...
11 };
```

Vererbung: Zugriffskontrolle

Vererbung \ Mitglied	public	protected	private
public	public	protected	n/a
protected	protected	protected	n/a
private	private	private	n/a

Vererbung: Zugriffskontrolle

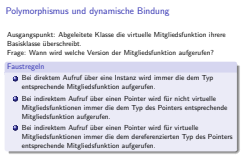
Vererbung \ Mitglied	`public`	`protected`	`private`
`public`	`public`	`protected`	n/a
`protected`	`protected`	`protected`	n/a
`private`	`private`	`private`	n/a



- virtual bewirkt, dass die nachfolgende Methode, wenn sie in einer Subklasse ueberschreiben wird bei Laufzeit dynamisch auswaehlt, welcher Typ die aufrufende Instanz hat.
- Das heisst ein Pointer der Basisklasse kann sowohl auf eine Instanz der Basisklasse, wie auch auf eine Instanz einer davon abgleiteten Klasse zeigen. Fuer nicht virtuelle Mitgliedsfunktionen wird dann ueber den Typ des Pointers bestimmt welche Mitgliedsfunktion ausgefuehrt wird. Fuer virtuelle Mitgliedsfunktionen wird dynamisch ausgewertet auf welche Instanz der Pointer zeigt, und dann deren Version der Mitgliedsfunktion ausgefuehrt.

Polymorphismus

```
1  class A {
2    virtual void print()
3    {cout<<"A"<<endl;}
4  };
5
6  class B : public A {
7    void print()
8    {cout<<"B"<<endl;}
9  };
10
11 A instantce1;
12 instance1.print();
13 B instance2;
14 instance2.print();
15 A * pointer1 = &instance2;
16 pointer1->print();
```



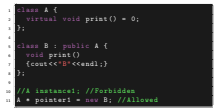
Polymorphismus und dynamische Bindung

Ausgangspunkt: Abgeleitete Klasse die virtuelle Mitgliedsfunktion ihrere Basisklasse überschreibt.
Frage: Wann wird welche Version der Mitgliedsfunktion aufgerufen?

Faustregeln

1. Bei direktem Aufruf über eine Instanz wird immer die dem Typ entsprechende Mitgliedsfunktion aufgerufen.
2. Bei indirektem Aufruf über einen Pointer wird für nicht virtuelle Mitgliedsfunktionen immer die dem Typ des Pointers entsprechende Mitgliedsfunktion aufgerufen.
3. Bei indirektem Aufruf über einen Pointer wird für virtuelle Mitgliedsfunktionen immer die dem dereferenzierten Typ des Pointers entsprechende Mitgliedsfunktion aufgerufen.

Abstrakte Basisklasse

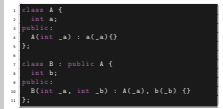


Abstrakte Basisklasse

- Wenn eine virtuelle Mitgliedsfunktion = 0 gesetzt wird, wird sie zur rein virtuellen Methode und die inhabende Klasse wird zur abstrakten Basisklasse.
- Das heisst, das die Klasse nicht mehr instantiiert werden kann, sondern nur als Basis fuer abgeleitete Klassen fungiert.

```
1  class A {
2    virtual void print() = 0;
3  };
4
5  class B : public A {
6    void print()
7    {cout<<"B"<<endl;}
8  };
9
10 //A instance1; //Forbidden
11 A * pointer1 = new B; //Allowed
```

Konstruktoren



- Da die privaten Mitgliedsvariablen der Basisklasse fuer die abgeleiteten Klassen nicht mehr verfuegbar sind, muss im Konstruktor der abgeleiteten Klasse der Konstruktor der Basisklasse aufgerufen werden.

Konstruktoren

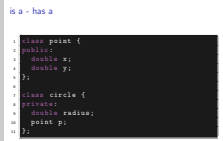
```
1  class A {
2    int a;
3  public:
4    A(int _a) : a(_a){}
5  };
6
7  class B : public A {
8    int b;
9  public:
10   B(int _a, int _b) : A(_a), b(_b) {}
11 };
```

is a - has a

- Die Klasse circle erbt von der Klasse Punkt und **ist** somit ein Punkt.
- Dies ist die erste Moeglichkeit der Implementation einer Kreis Klasse die Zugriff auf die Variablen eines Punkts hat.

is a - has a

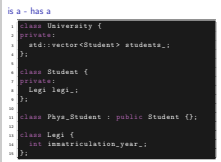
```
1  class point {
2  public:
3    double x;
4    double y;
5  };
6
7  class circle : private point {
8  private:
9    double radius;
10 };
```



is a - has a

- Die Klasse circle erbt nicht von point sondern **hat** einen Punkt als Mitglied.
- Der Unterschied beider Implementierungen ist minimal, eigentlich unterscheidet sich nur der Zugriff auf die Mitgliedsvariablen des Punkts.
- Sprachlich unterscheidet man von Fall 1 **“ist”** zu Fall 2 **“hat”** einen Punkt.

```
1  class point {
2  public:
3    double x;
4    double y;
5  };
6
7  class circle {
8  private:
9    double radius;
10   point p;
11 };
```



is a - has a

```
1  class University {
2  private:
3    std::vector<Student> students_;
4  };
5
6  class Student {
7  private:
8    Legi legi_;
9  };
10
11 class Phys_Student : public Student {};
12
13 class Legi {
14   int immatriculation_year_;
15 };
```

Prüfungsvorbereitung
Alte Prüfungen D-ITET
Alte Prüfungen D-PHYS

Prüfungsvorbereitung

Alte Prüfungen D-ITET

Alte Prüfungen D-PHYS