

[Exercise Week 12](#)

GianAndrea Müller
<mailto:muellegi@student.ethz>

May 28, 2018

Exercise Week 12

GianAndrea Müller
<mailto:muellegi@student.ethz>

May 28, 2018

└ Time Schedule

Time Schedule

- 10' Wiederholung Klassen
- 10' Vererbung
- 10' Polymorphie und Abstrakte Basisklassen
- 10' Konstruktoren

Time Schedule

- 10' Wiederholung Klassen
- 10' Vererbung
- 10' Polymorphie und Abstrakte Basisklassen
- 10' Konstruktoren

└ Learning Objectives

Learning Objectives

- Verständnis: Vererbung
- Anwendung: Polymorphismus

Learning Objectives

- Verständnis: Vererbung
- Anwendung: Polymorphismus

└ Klassen: Wiederholung

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:

- Mitgliedsfunktion
- Mitgliedsvariable
- Zugriffsoperatoren

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

```
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

Klassen: Wiederholung

Was wir gelernt haben:

- Zugriffsbereiche
- Zugriffsmethoden

Klassen: Wiederholung

Was wir gelernt haben:

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

- Es gibt den oeffentlichen und den privaten Zugriffsbereich. Oeffentlich heisst: Nicht verfuegbar von aussen, ueber Zugriffsooperatoren. 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

Klassen: Wiederholung

Was wir gelernt haben:

- Konstruktor
- Kopierkonstruktor
- Zuweisungsoperator
- Destruktor

Klassen: Wiederholung

Was wir gelernt haben:

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

- 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.

└ Klassen: Wiederholung

Klassen: Wiederholung

Was wir gelernt haben:

• Operatoren

Klassen: Wiederholung

Was wir gelernt haben:

• Operatoren

- 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.

└ 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.

Vererbung: Grundlagen

```
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: Grundlagen

```
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: 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

└ Polymorphismus

```

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 instance1;
12 instance1.print();
13 B instance2;
14 instance2.print();
15 A * pointer1 = &instance2;
16 pointer1->print();

```

- virtual bewirkt, dass die nachfolgende Methode, wenn sie in einer Subklasse ueberschrieben wird bei Laufzeit dynamisch ausgewaehlt, 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 abgeleiteten 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 instance1;
12 instance1.print();
13 B instance2;
14 instance2.print();
15 A * pointer1 = &instance2;
16 pointer1->print();

```

└ Polymorphismus und dynamische Bindung

Polymorphismus und dynamische Bindung

Ausgangspunkt: Abgeleitete Klasse die virtuelle Mitgliedsfunktion ihrer Basisklasse überschreibt.

Frage: Wann wird welche Version der Mitgliedsfunktion aufgerufen?

Faustregeln

- ❶ Bei direktem Aufruf über eine Instanz wird immer die dem Typ entsprechende Mitgliedsfunktion aufgerufen.
- ❷ Bei indirektem Aufruf über einen Pointer wird für nicht virtuelle Mitgliedsfunktionen immer die dem Typ des Pointers entsprechende Mitgliedsfunktion aufgerufen.
- ❸ Bei indirektem Aufruf über einen Pointer wird für virtuelle Mitgliedsfunktionen immer die dem dereferenzierten Typ des Pointers entsprechende Mitgliedsfunktion aufgerufen.

Polymorphismus und dynamische Bindung

Ausgangspunkt: Abgeleitete Klasse die virtuelle Mitgliedsfunktion ihrer Basisklasse überschreibt.

Frage: Wann wird welche Version der Mitgliedsfunktion aufgerufen?

Faustregeln

- ❶ Bei direktem Aufruf über eine Instanz wird immer die dem Typ entsprechende Mitgliedsfunktion aufgerufen.
- ❷ Bei indirektem Aufruf über einen Pointer wird für nicht virtuelle Mitgliedsfunktionen immer die dem Typ des Pointers entsprechende Mitgliedsfunktion aufgerufen.
- ❸ Bei indirektem Aufruf über einen Pointer wird für virtuelle Mitgliedsfunktionen immer die dem dereferenzierten Typ des Pointers entsprechende Mitgliedsfunktion aufgerufen.

└ Abstrakte Basisklasse

Abstrakte Basisklasse

```

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

```

Abstrakte Basisklasse

```

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

```

- 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.

└ 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 };
```

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 };
```