Inhalt

Einführung in C++, II

- Vererbung
- Polymorphie
- Abstrakte Klassen
- Mehrfachvererbung
- Templates

Verebung

C++ unterscheidet zwischen

public-, private- und protected-Vererbung

```
          class Sub : public    Basis {...}
oder      class Sub : private   Basis {...}
oder      class Sub : protected Basis {...}
```

Lesart:
public abgeleitet, private Basisklasse, …

defaults:
für `class` → `private`
für `struct` → `public`

Hierdurch wird definiert, welche Zugriffsrechte die vererbten Elemente in der subclass haben:

- *private Elemente* werden nicht vererbt
- public Vererbung:
  public und protected Elemente bleiben public / protected
- protected Vererbung:
  public und protected Elemente werden protected
- private Vererbung:
  public und protected Elemente werden private

Mögliche Verwendung der drei Vererbungsarten:

public Vererbung

- mit Abstand die häufigste Art der Vererbung
- dient der Modellierung von Ist-Beziehungen:

  *Baum **ist** eine Pflanze, Haus **ist** ein Gebäude, Segelboot **ist** ein Boot, ...*

  Das subclass Objekt **ist** auch ein Basisobjekt, denn alles was ein Basisobjekt kann, kann das Subobjekt ebenfalls.
- public abgeleitete subclasses sind Spezialisierungen der Basisklasse: Subobjekte sind spezifischer, die subclass fügt Elemente hinzu und erweitert so die Basisklasse um speziellere Methoden/Attribute
- Aus Anwendungssicht ist die subclass keine Einschränkung gegenüber der Basisklasse - die öffentliche Schnittstelle wird höchstens erweitert.

private Vererbung

- Alle Elemente der Basisklasse werden privat → die öffentliche Schnittstelle der subclass wird eingeschränkt. Innerhalb der subclass steht die öffentliche Schnittstelle der Basisklasse aber vollständig zur Verfügung. Aus äußerer Sicht hat die subclass Eigenschaften verloren - sie stellt keine Spezialisierung mehr dar.
- Durch private Vererbung wird eher eine Hat-Beziehung abgebildet:

  *Mensch **hat** Kopf, Team **hat** Mitarbeiter, Auto **hat** Räder, ...*

  Geerbte Elemente sind in der subclass gekapselt, sodass hier eine Alternative zu Aggregaten besteht.

protected Vererbung

- Ähnlich der privaten Vererbung, aber: Geerbtes ist in der subclass protected und kann weitervererbt werden.
  Die private Vererbung hingegen unterbricht die Vererbung der Basiselemente an weitere Generationen.

Konstruktoren, Initialisierung:

- Wird ein Subclassobjekt instanziiert, so wird zuerst der Konstruktor der Basisklasse aufgerufen.
  Wurde über mehrere Generationen vererbt, so wird zuerst der innerste Konstruktor gerufen.
- Besitzen die Basisklassen einen default-Konstruktor, wird dieser automatisch ausgeführt, falls nicht anders angegeben.
- Non-Default Konstruktoren müssen explizit in der Initialisierungsliste des Subclasskonstruktors aufgeführt werden.

Destruktoren

- Destruktoren sind immer „default-Destruktoren“, da sie keine Parameter haben - sie werden entlang der Vererbungshierarchie automatisch aufgerufen.
- Werden dynamisch Elemente angelegt, so muss dennoch ein spezieller Destruktor geschrieben werden, der den heap freigibt.
- Destruktoren werden in umgekehrter Reihenfolge der Konstruktoren aufgerufen, d.h. zuerst der Destruktor der subclass, dann der Basisdestruktor.

Zuweisung

Für alle Klassen gibt es einen default-Zuweisungsoperator, der Objekte einer gemeinsamen Klasse verarbeitet:

- Alle Attribute werden elementweise kopiert.
- Heapobjekte werden nicht berücksichtigt, sie müssen gesondert behandelt werden (Operator überladen, Copy Konstruktor, etc.)

Für Objekte abgeleiteter Klassen gilt:

- Bei der Zuweisung von Subclassobjekten wird für den Basisklassenanteil automatisch auch der Zuweisungsoperator der Basisklasse aufgerufen. Sind dort Heapobjekte enthalten, müssen sie durch Überladung in der Basisklasse behandelt werden.
- Da Subclassobjekte auch Basisklassenobjekte *sind*, ist die Zuweisung

  `basis_obj = sub_obj;` immer möglich

  Die spezifischen Attribute des **`sub_obj`** werden dabei natürlich nicht berücksichtigt.

- Die Umkehrung

  `sub_obj = basis_obj;` ist nur möglich,

  wenn die subclass einen entsprechenden Operator bereitstellt.

Redefinition

Haben Elemente in der abgeleiteten Klasse den gleichen Bezeichner wie Elemente der Basisklasse, so nennt man dies
➔ Redefinition (Überdeckung).

Dabei ist zu beachten:

- Subclasselemente (Methoden und Attribute) verdecken die Elemente der Basisklasse (analog zur Verdeckung von Variablen in lokalen Blöcken).
- Mit dem Bereichsoperator **::** kann auf die verdeckten Elemente zugegriffen werden.
- Redefinition ist kein Overloading:
  - Overloading findet im selben Gültigkeitsbereich statt, z.B. innerhalb *einer* Klasse.
  - Überladene Funktionen werden anhand der Signatur unterschieden, **ein redefiniertes Element verdeckt jedoch alle gleichnamigen Elemente der gesamten Klassenhierarche**, unabhängig von ihrer Signatur.
- Sind Elemente in der Klassenhierarchie mehrfach redefiniert, so wird in der Hierarchie 'aufwärts' gesucht und das erste passende Element gewählt.

Beispiel zur Redefinition

```
class Basis
{
public:
   void moin()   { cout << "Moin "; }
   void fritz()  { cout << "Fritz\n"; }
};

class Sub : public Basis
{
public:
   void moin()   { cout << "MOIN "; }
   void fritz(int dummy) { cout << "FRITZ\n"; }

   void moinFritz()
   {
      Basis::moin();
      Basis::fritz();
   }
};
```

```
int main()
{
   Sub obj;

   obj.moin();
   // nicht moeglich:
   // obj.fritz();
   obj.fritz(1);
   obj.moinFritz();
}
```

Ausgabe:
MOIN FRITZ
Moin Fritz

Polymorphie

Der Begriff Polymorphie (Vielgestaltigkeit) wird unterschiedlich verwendet. Hier:
Polymorphie liegt vor, wenn über die Vielgestaltigkeit erst zur Laufzeit entschieden wird.

- *Overloading*, *Redefinition* und *Templates* werden zur Übersetzungszeit aufgelöst, d.h. ein Funktionsaufruf wird früh an die endgültige Funktionsdefinition gebunden
  ➔ *early binding, static binding*
- Ein polymorpher Aufruf wird zur Laufzeit, an die tatsächliche Funktion gebunden
  ➔ *late binding, dynamic binding*
  Diese Bindung kann im weiteren Verlauf des Programms geändert werden.

Späte Bindung wird in C++ mit virtuellen Methoden realisiert.

Basisklassenzeiger (statische Bindung)

Zeiger des Typs "Zeiger auf Basisklasse" (*Basisklassenzeiger*) haben in C++ zentrale Bedeutung für polymorphe Programme:

- Für Basisklassenzeiger wird implizites upcasting ausgeführt, d.h. ein solcher Zeiger kann auch auf Objekte aus abgeleiteten Klassen zeigen
  ➔ der Basisklassenzeiger ist zu seinen subclass-Objekten kompatibel
- Liegt statische Bindung vor, sind über diesen Zeiger aber nur die in dem subclass-Objekt enthaltenen Basisklassenmember ansprechbar.

➔ **Bei statischer Bindung entscheidet der Typ des Zeigers**, welche Member tatsächlich angeprochen werden ("Basiszeiger spricht Basiselemente an").

Die Aussagen dieser Seiten gelten analog auch für *Basisklassenreferenzen*.

Basisklassenzeiger (dynamische Bindung)

- Virtuelle Methoden werden dynamisch gebunden.
- Sie werden mit dem specifier **`virtual`** deklariert.
- Virtuelle Methoden werden vererbt.
- Abgeleitete Klassen können virtuelle Methoden ***'überschreiben'***. Man sagt auch: eine subclass implementiert eine *spezifische Version*.
- Andes als beim overloading müssen Signatur und Ergebnistyp der 'neuen Version' in der subclass identisch zur überschriebenen Methode sein (sonst liegt Redefinition vor).
- Der specifier **`virtual`** muss nicht wiederholt werden, es gilt: *„einmal virtuell, immer virtuell“*

➔ **Bei dynamischer Bindung entscheidet der Typ des Objekts**

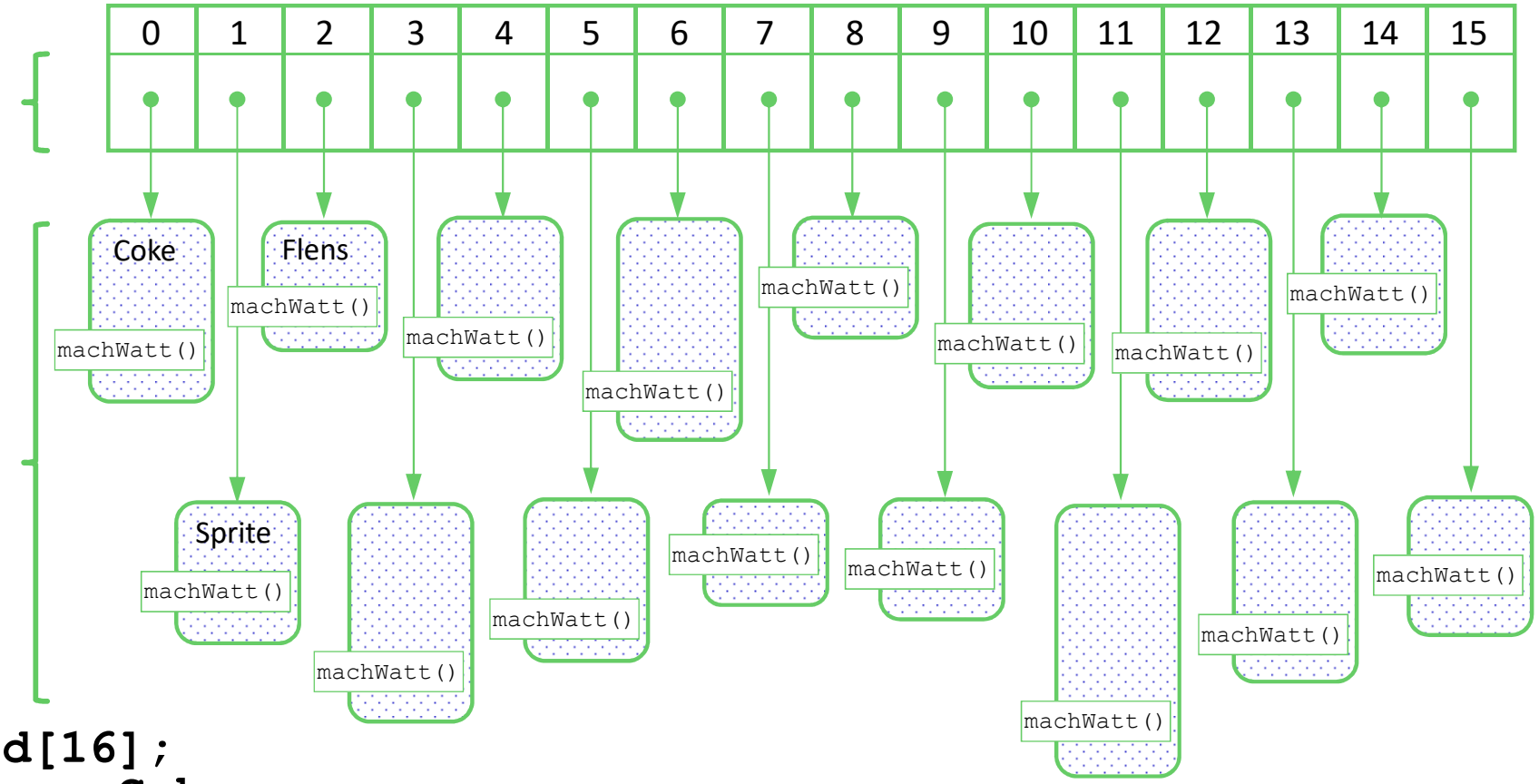
Ruft man eine (überschriebene) virtuelle Methode über den Basisklassenzeiger oder die Basisklassenreferenz auf, so kommt immer die Methode desjenigen Objekts zu Ausführung, auf das aktuell gezeigt wird.

Zweck der virtuellen Methoden (Polymorphie)

- Polymorphes Verhalten von Objekten einer Familie:
  ➔ Ein gleichlautender Methodenaufruf kann dann für alle Objekte einer Klassenfamilie ausgeführt werden, wobei jedes Objekt seine Version ausführt.
- Methoden aus Klassenbibliotheken können mit eigenen Versionen überschreiben werden:
  Eine eigene subclass der Bibliothek wird gebildet, eine virtuelle Methode wird dort überschrieben.
  ➔ Über den Basisklassenzeiger können *bereits vorhandene* Bibliotheksmethoden jetzt die neue Version aufrufen.

  Hinweis: Der Quellcode der Bibliothek ist hierzu nicht erforderlich.
- Basisklassenzeiger dienen als Funktionsparameter:
  ➔ Die mit diesem Zeiger aufgerufenen Methoden werden polymorph ausgeführt.

Beispiel "Feld mit Basisklassenzeigern"




```
Basis* feld[16];
feld[0] = new Coke;
feld[1] = new Sprite;
   ...

for(int i = 0; i <= 15; i++)
   feld[i]->machWatt();
```

machWatt() ist in der Basisklasse **virtual** deklariert

ein polymorpher Aufruf für alle Objekte

Abstrakte Klassen

- Die Basisklasse einer 'polymorphen Klassenhierarchie' hat vor allem die Aufgabe, *virtuelle Methoden öffentlich bereitzustellen*.
- In den meisten Fällen ist an dieser Stelle aber noch nicht bekannt, welche Aufgaben diese Methoden im Detail erfüllen sollen, d.h. der Rumpf ist i.A. unbekannt.

  Eine virtuelle Methode mit noch undefiniertem Rumpf heißt **rein virtuell** und wird bei der Deklaration mit `= 0` gekennzeichnet, z.B:

```
virtual void machWatt() = 0;
```

Regeln

- Enthält eine Klasse mindestens eine rein virtuelle Methode, so ist die Klasse **abstrakt.**
- Aus abstrakten Klassen können keine Instanzen erzeugt werden.
- Alle abgeleiteten Klassen erben die rein virtuellen Methoden und werden selbst abstrakt, sofern sie nicht **alle** rein virtuellen Methoden konkret implementieren (d.h. einen Rumpf bereitstellen).

Virtuelle Destruktoren

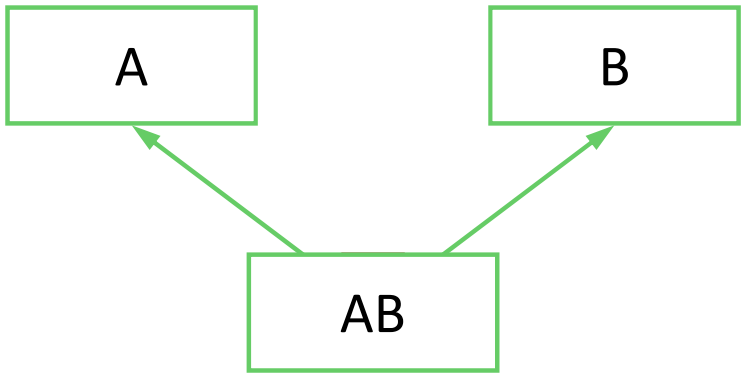
- Wird ein Objekt mit dem Basisklassenzeiger oder einer Basisklassenreferenz verwaltet, so wird - wie bei anderen Methoden auch - nur der Basisklassenanteil angesprochen. Bei statische Bindung gilt dann auch für Destruktoren:
  ➔ nur der Basisklassendestruktor wird ausgeführt

- Sollen in diesem Fall die Destruktoren aus subclasses jemals zur Ausführung kommen, muss in der Basisklasse der Destruktor virtuell deklariert werden.
- Da normalerweise in der Basisklasse noch nicht bekannt ist, ob in einer subclass ein Destruktor implementiert werden muss, sollte eine Basisklasse möglichst immer den Destruktor virtuell deklarieren.

Mehrfachvererbung

Mehrfachvererbung (multiple inheritance) erlaubt, eine Klasse aus mehr als einer direkten Basisklasse abzuleiten:

```
class AB : public A, public B
{
        ...
};
```



➔ Die Klasse AB erbt aus beiden Klassen, sie verfügt über die Elemente aus A und B.

einfache Mehrdeutigkeiten

Durch identisch bezeichnete Elemente treten Mehrdeutigkeiten auf, die mit dem Bereichsoperator :: aufzulösen sind.

Sei in **A** und **B** je eine Methode **machWatt()** definiert, so können diese für ein Objekt **ab** der Klasse **AB** wie folgt aufgerufen werden:

```
ab.A::machWatt();
ab.B::machWatt();
```

Der Aufruf `ab.machWatt();` würde nicht kompiliert werden.

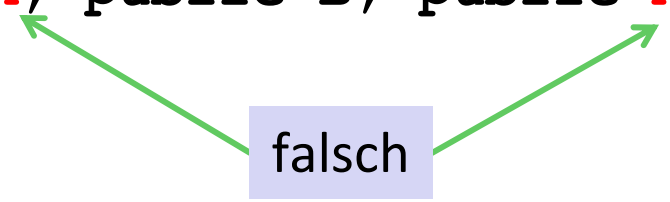
Alle eindeutig bezeichneten Elemente können direkt verwendet werden, denn **ab** ist ein Objekt der Klasse `A` und der Klasse `B`.

➔ **ab** enthält ein Subobjekt aus **A** und ein Subobjekt aus **B**.

Mehrfach indirekte Basisklassen

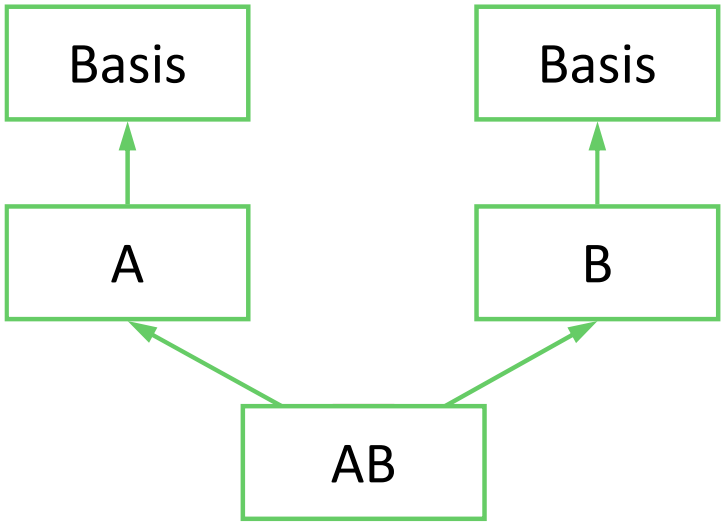
Es ist nicht möglich, dass eine Klasse direkt mehrfach als Basisklasse verwendet wird:

```
class AB : public A, public B, public A
{
  ...
};
```



Indirekt ist dies möglich, in den meisten Fällen aber nicht erwünscht:

- **`Basis`** ist mehrfach indirekte Basisklasse
- alle **`AB`** Objekte erben indirekt zweifach aus **`Basis`**. Sie enthalten alle **`Basis`**-Elemente **doppelt.**



➔ Auch diese Mehrdeutigkeiten müssen mit Angabe der Klasse über die weitervererbt wurde (**`A`** bzw. **`B`**) und dem Bereichsoperator aufgelöst werden.

Beispiel indirekte Basisklasse

```
class Basis
{
protected:
   string str;
public:
   void zeigDich() { cout <<
      "zeigDich aus Basis\n"; }
};

class A : public Basis
{
public:
   void machWatt() { cout <<
      "machWatt aus A\n"; }
};

class B : public Basis
{
public:
   void machWatt() { cout <<
      "machWatt aus B\n"; }
};
```

```
class AB : public A, public B
{
public:
   AB()
   {
      A::str = "Asterix und ";
      B::str = "Obelix\n";
   }
   void zeigWatt()
   { cout << "Ich habe zwei strings: "
          << A::str + B::str; }
};

int main()
{
   AB ab;

//   ab.zeigDich();  // mehrdeutig
   ab.A::zeigDich();

//   ab.machWatt();  // mehrdeutig
   ab.B::machWatt();

   ab.zeigWatt();
}
```

```
zeigDich aus Basis
machWatt aus B
Ich habe zwei strings: Asterix und Obelix
```

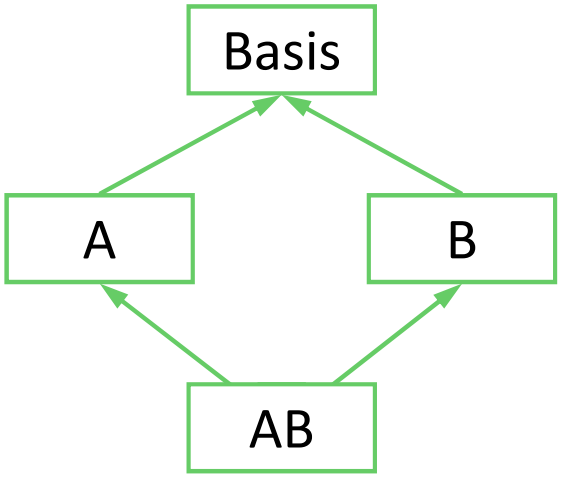
Virtuelle Basisklassen

Virtuelle Basisklassen lösen das Problem der mehrfach indirekt vererbten Klassen (die Bezeichnung *virtuell* ist hier unglücklich).

Kennzeichnet man bei der Deklaration der Klassen **`A`** und **`B`** die Basisklasse **`Basis`** als **`virtual`**, so entsteht ein anderes Vererbungsdiagramm:

```
class A : virtual public Basis { ... };
class B : virtual public Basis { ... };
```

- Die **`Basis`**-Elemente sind jetzt nur noch einfach vorhanden.
- **`zeigWatt()`** kann nun auf **`str`** direkt zugreifen, oder auch wie vorher mit **`A::str`** oder **`B::str`**



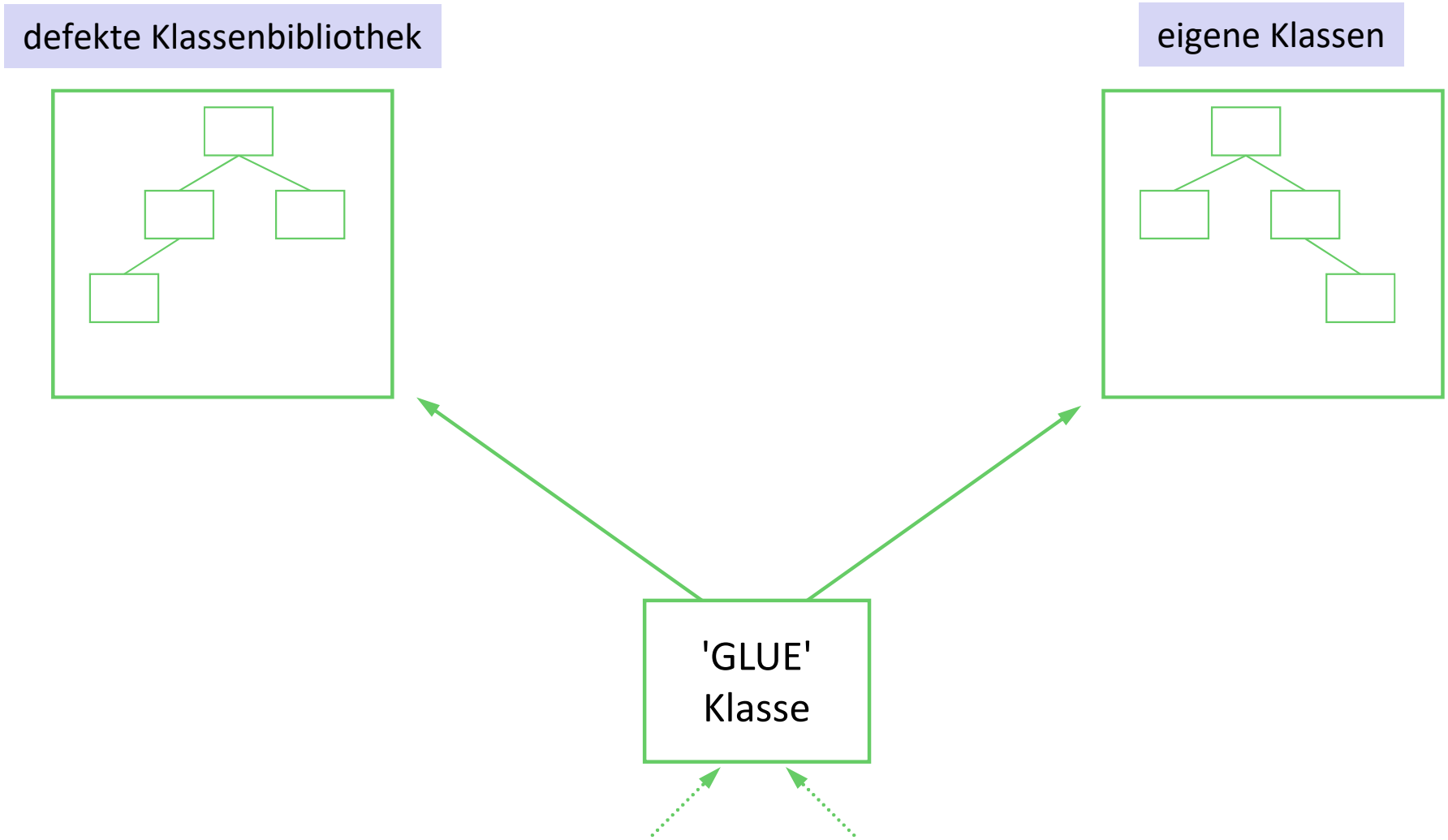
Zweck der Mehrfachvererbung

- Die Notwendigkeit der Mehrfachvererbung ist umstritten.
- Mehrfachvererbung erhöht die Komplexität eines Programms.
- Die meisten Probleme lassen sich auch ohne Mehrfach-vererbung lösen:

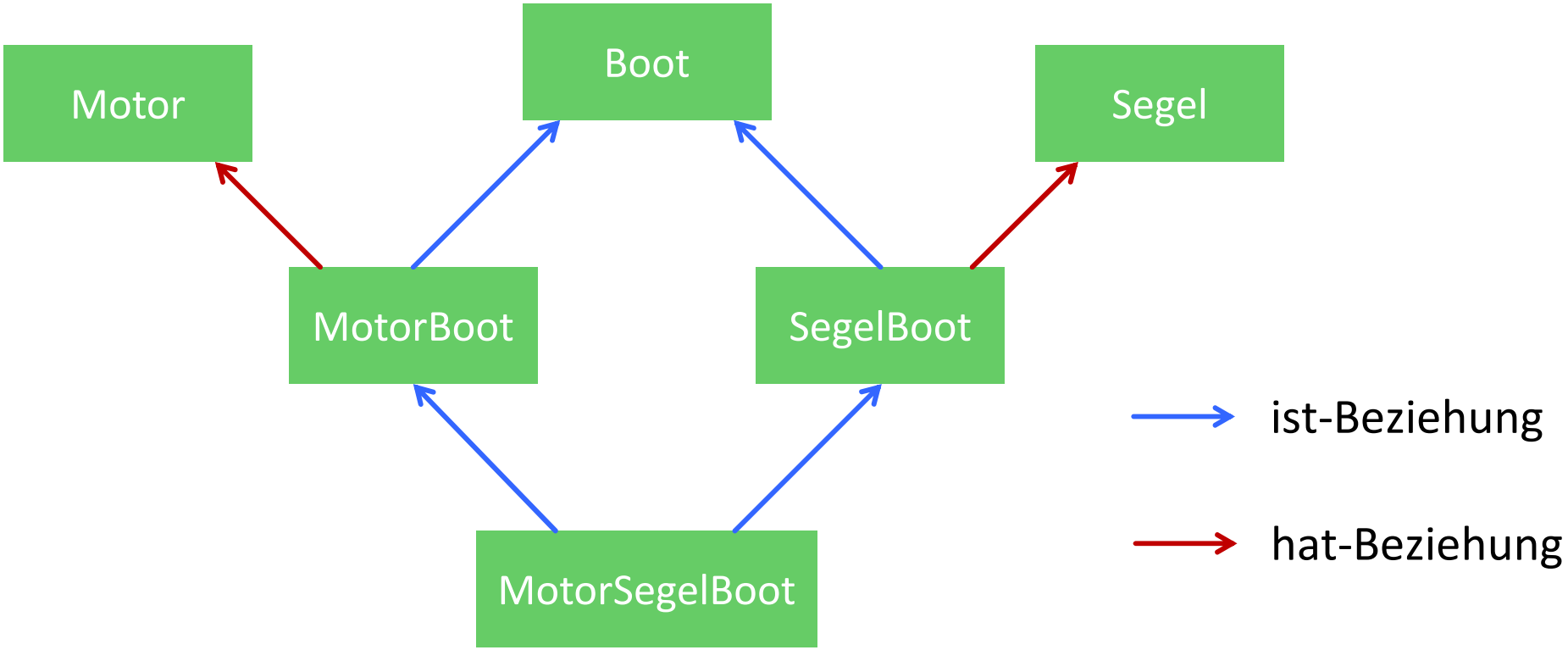
  "Sie können sich MV als eine Eigenschaft der Programmiersprache 'von geringerer Bedeutung' vorstellen, die in Ihre täglichen Entscheidungen hinsichtlich des Entwurfs nicht einfließen sollte." (Bruce Eckel)

- Häufiger als in anderen OOP-Sprachen entsteht in C++ der Bedarf an MV dadurch, dass C++ Klassenhierarchien i.A. keine gemeinsame Basis haben.
- Eine anerkannt sinnvolle Verwendung der MV ist die Reparatur von Schnittstellen, z.B.:
  - erworbene Klassenbibliothek kann/soll nicht modifiziert werden
  - Methoden fehlen
  - Methoden oder Destruktor nicht virtuell deklariert
  - ...

Reparatur von Schnittstellen mit MV



Beispiel Mehrfachvererbung




```
class MotorBoot : virtual public Boot, protected Motor { ... }
class SegelBoot : virtual public Boot, protected Segel { ... }

class MotorSegelBoot: public MotorBoot, public SegelBoot { ... }
```

Templates

- Templates (*Schablonen*) gehören nicht zu den objektorientierten Konstrukten. Sie wurden in C++ erst spät, nach dem Vorbild der Sprache Ada, eingeführt.
- Templates erlauben es, z.B. einen Algorithmus oder einen abstrakten Datentyp generisch zu implementieren, genauer mit generischen Typen.
- Statt etwa einen Stack für `int` zu programmieren, arbeitet man mit einem generischen template Typ, der erst später konkret eingesetzt wird, wenn der Stack verwendet wird.
- Schablonen eignen sich insbesondere zur Implementation von ***Containern*** (siehe auch STL, *Standard Template Library*).
- C++ kennt:
  - template functions und
  - template classes

Das function template Beispiel

ältere C++ Versionen verwenden hier **`class`** statt **`typename`**

```
template <typename irgendeinTyp>
void tausche( irgendeinTyp& x, irgendeinTyp& y )
{
    irgendeinTyp hilf = x;
    x = y;
    y = hilf;
}
```

- In der ersten Zeile wird der Bezeichner **`irgendeinTyp`** als template Typname für diese function deklariert.
- Die function **`tausche`** ist nun für alle Typen anwendbar ('instanziierbar'), für welche die in der function benutzten Operationen definiert sind.
  ➔ hier alle Typen, für die der Operator '**=**' definiert ist

Mögliche Aufrufe der **`tausche`** function

```
double  a = 1.0;          double  b = 5.0;
string s1 = "mene mu";    string s2 = "ene";

tausche(  a,  b );
tausche( s1, s2 );
```

➔ Trifft der Compiler auf einen **`tausche`** -Aufruf, generiert er aus dem template die passende konkrete function, sofern dies möglich ist.

Implizite Konvertierungen werden nicht vorgenommen:

```
int i = 42;
double  a = 24.0;
tausche( i, a );
```

`i` oder **`a`** müsste konvertiert werden, da **`tausche`** nur Parameter gleichen Typs akzeptiert

➔ wird nicht kompiliert, da gleichzeitig weitere überladene functions **`tausche`** existieren könnten und keine eindeutige Zuordnung mehr gegeben wäre.

Template **Spezialisierungen**

- Ist eine template function mit einer non-template function überladen (d.h. für einen konkreten Typ), so nennt man dies eine Spezialisierung.
- Eine `int`-Spezialisierung für `tausche` könnte sein:

```
template <>
void tausche( int& x, int& y) { ... }
```

Spezialisierung nach ANSI C++

Dies ist eigentlich eine ganz 'normale' function, die aber vom Compiler bevorzugt ausgewählt wird - die template function würde nämlich auch 'passen'.

mehrere template-Parameter:

Die Anzahl der generischen Parameter ist nicht beschränkt, folgendes ist möglich:

```
template <typename meinTyp, typename deinTyp>
deinTyp pipapo( meinTyp x, deinTyp y, const deinTyp& z)
{ ... }
```

Hinweis: die Aufteilung in Deklarations- und Definitionsdatei wird für templates meist nicht unterstützt.
➔ Alles in die `*.h` Datei oder am Ende der `*.h` Datei ein `#include "*.cpp"` einfügen.

`class` templates

- Analog zu function templates, werden ein oder mehrere template Typen für den Geltungsbereich einer Klassendeklaration angegeben:

```
template <typename AiTy,typename MiTy, typename KtTy>
class AiMiKt
{
  ...
};
```

- Sollen Methoden außerhalb der Klassendeklaration definiert werden, so müssen die template-Typen bei jeder Methodendefinition deklariert werden.
  Außerdem wird hinter dem Klassenbezeichner die Liste der template Typen in spitzen Klammern angegeben, denn es könnte z.B. eine gleichnamige non-template **`class`** existieren.

Beispiel template class

```
template <typename Ttyp>
class Quatsch
{
     Ttyp a;
public:
     Quatsch( Ttyp A ): a(A) {}
     void zeigDich();
};

template <typename Ttyp>
void Quatsch<Ttyp>::zeigDich()
{
   cout << "a: " << a << endl;
}
```

Hier nur der Prototyp

vollständiger Name der Methode

Bei der Instanziierung wird der gewünschte konkrete Typ in spitzen Klammern angegeben:

```
Quatsch<char>   q1( 'x' );
Quatsch<string> q2( "Ping" );

q1.zeigDich();
q2.zeigDich();
```

sonstige Parameter

In der Liste der template Parameter sind auch Parameter konkreter Typen erlaubt (z.B. **`int`**). Die Werte werden beim Instanziieren in den spitzen Klammern angegeben.
Zu beachten ist:

- Fließkommatypen sind nicht erlaubt
- in der instanziierten Klasse sind diese Parameter konstant

Beispiel:

```
template <typename T, int konkretKeinFloatKonstant>
class Quatsch2
{
   T Feld[konkretKeinFloatKonstant + 42];
   void machWatt(){ ++konkretKeinFloatKonstant; }
  ...
};
```

nicht möglich, konstant