HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Embedded Software Engineering 2
Implementing Virtual Functions

Prof. Reto Bonderer
HSR Hochschule für Technik Rapperswil
reto.bonderer@hsr.ch

April 2020

Dynamic Binding (Polymorphismus)

- Ist der mächtigste OO-Mechanismus (oft präziser als *run-time polymorphism* bezeichnet)
- Elementfunktionen, die dynamisch gebunden werden, muss bei der Deklaration das Schlüsselwort `virtual` vorangestellt werden (zwingend!).
  In der abgeleiteten Klasse muss die Funktion mit `override` gekennzeichnet werden (ab C++11).
- Polymorphismus wird häufig als ineffizient bezeichnet, oft jedoch zu unrecht (folgt noch)

Virtuelle Elementfunktionen

- Elementfunktionen müssen virtuell deklariert werden, wenn sie in einer Unterklasse bei gleicher Signatur und bei gleichem Rückgabetyp überschrieben werden

- Beim Aufruf einer virtuellen Elementfunktion gilt:
  - Aufruf direkt über das Objekt: es gilt die Implementation der Klasse, zu dem das Objekt gehört
  - Aufruf über Zeiger oder Referenz: es gilt die Implementation der Klasse des Objektes, auf das der Zeiger zeigt bzw. auf das die Referenz sich bezieht

- Eine in einer Basisklasse als `virtual` gekennzeichnete Memberfunktion definiert eine Schnittstelle für alle abgeleiteten Klassen

Regeln für virtuelle Memberfunktionen

- **Nicht virtuelle** Memberfunktionen einer Basisklasse sollen in Subklassen **nicht überschrieben** werden
- Ist vorauszusehen, dass Memberfunktionen in Subklassen überschrieben werden (sollen), so sollten die entsprechenden Memberfunktionen bereits in der Basisklasse als `virtual` deklariert sein.
- Wenn in einer Klasse virtuelle Memberfunktionen vorkommen, muss auch der Destruktor als `virtual` deklariert werden (sonst nicht!)
- Konstruktoren sind nie virtuell

Statischer vs. dynamischer Datentyp

```
class Article;
class Book : public Article {};
Article* pa;
pa = new Book;
```

- Der statische Datentyp bezeichnet den Datentyp bei der Deklaration
  Im Beispiel: **pa** ist ein Pointer auf **`Article`**

- Der dynamische Datentyp bezeichnet den effektiven Datentyp zur Laufzeit
  Im Beispiel: **pa** ist ein Pointer auf **Book**

Static vs. Dynamic Binding

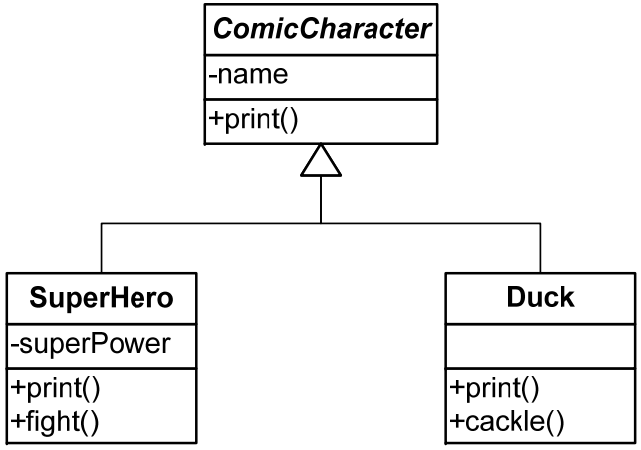
- Static binding (early binding, statische Bindung)
  - Bereits zur Compilezeit wird festgelegt, welcher (Elementfunktions-) Code ausgeführt wird (Normalfall)
  - Bei *statischen Aufrufen* erfolgt die Auswahl der richtigen Funktion bereits bei der Übersetzung (d.h. ohne Overhead).

- Dynamic binding (late binding, dynamische Bindung)
  - Erst zur Laufzeit wird in Abhängigkeit des Objekts festgelegt, welcher (Elementfunktions-) Code ausgeführt wird
  - Bei *dynamischen Aufrufen* (über Pointer oder Referenzen) erfolgt die Auswahl der richtigen Funktion zur Laufzeit aufgrund des tatsächlichen (dynamischen) Typs des Objekts. Dies ist mit etwas Overhead verbunden.

Statischer Aufruf von virtuellen Elementfunktionen

```
Duck donald;
SuperHero luckyLuke;

// direkter (statischer) Aufruf
// über die entsprechenden Objekte
donald.print();     // Duck::print()
luckyLuke.print();  // SuperHero::print()
```

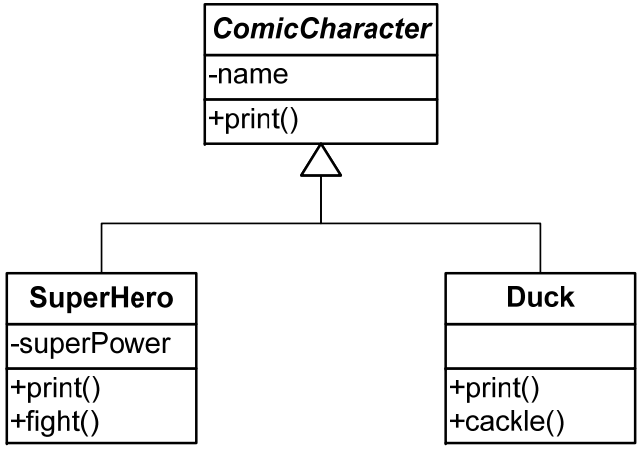


Dynamischer Aufruf von virtuellen Elementfunktionen

```
void printCC1(const ComicCharacter& c)
{
  c.print();   //dynamische Auflösung
}

void printCC2(const ComicCharacter* pc)
{
  pc->print(); //dynamische Auflösung
}

Duck donald;
SuperHero luckyLuke;
printCC1(donald);     // Duck::print()
printCC2(&luckyLuke); // SuperHero::print()
```

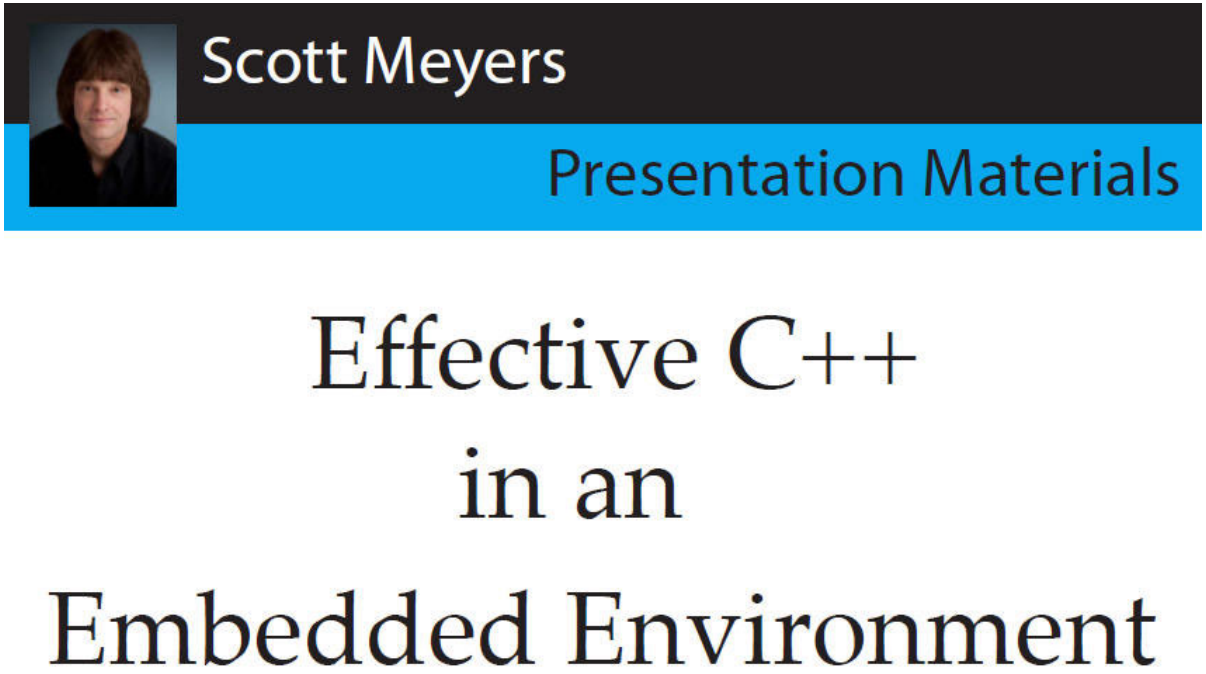


Polymorphe Klassen (Virtuelle Klassen)

- Eine Klasse, welche mindestens eine virtuelle Funktion deklariert, heisst *virtuell* (*polymorph*)
- Virtuelle Klassen bewirken einen Mehraufwand für den Compiler und sind darum langsamer in der Ausführung
- Funktionen sollten nur dann als virtuell deklariert werden, wenn sie in einer abgeleiteten Klasse überschrieben werden (sollen)
- Konstruktoren sind nie virtuell
- Destruktoren virtueller Klassen müssen immer als virtuell deklariert werden, sonst wird nur der Destruktor der Basisklasse aufgerufen
- Nicht virtuelle Methoden dürfen nicht überschrieben werden

Effective C++ in an Embedded Environment

Die meisten der folgenden Informationen stammen aus einem Seminar von Scott Meyers



Implementing Virtual Functions

- Compilers are allowed to implement virtual functions in any way they like:
  - There is no mandatory “standard” implementation

- The description that follows is *mostly* true for most implementations:
  - I’ve skimmed over a few details
  - None of these details affects the fact that virtual functions are typically implemented *very* efficiently

Implementing Virtual Functions (cont'd)

Consider this class:

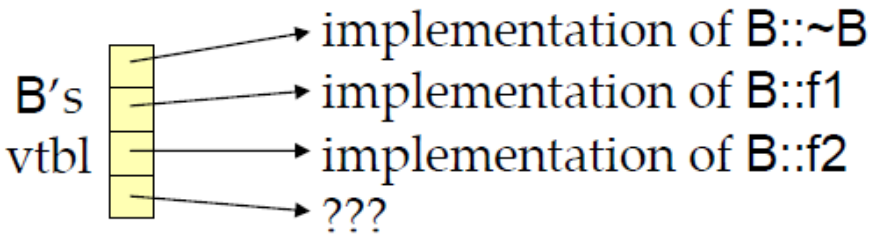
```
class B {
  public:
    B();
    virtual ~B();
    virtual void f1();
    virtual int f2(char c) const;
    virtual void f3(int x) = 0;
    void f4() const;
    ...
};
```

Compilers typically number the virtual functions in the order in which they're declared. In this example,

- The destructor is number 0
- f1 is number 1, f2 is number 2, f3 is number 3
- Nonvirtual functions (such as f4) get no number.

Implementing Virtual Functions (cont'd)

- A *vtbl* (“virtual table”) will be generated by the compiler for the class. It will look something like this:



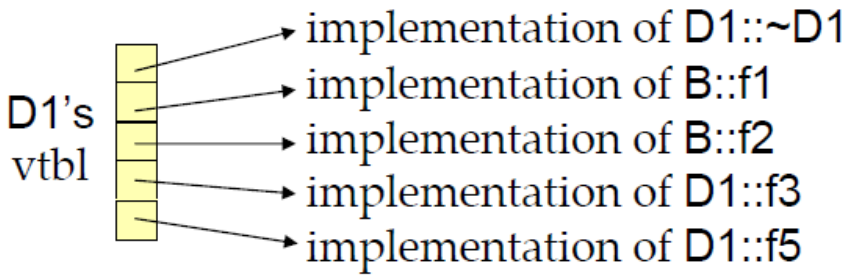
- The vtbl is an array of pointers to functions
- It points to virtual function implementations:
  - The *i*th element points to the virtual function numbered *i*
  - For pure virtual functions, what the entry is is undefined.
    - It’s often a function that issues an error and quits.
- Nonvirtual functions (including constructors) are omitted:
  - Nonvirtual functions are implemented like functions in C

Implementing Virtual Functions (cont'd)

Now consider a derived class:

```
class D1: public B {
  public:
    D1();                                  // nonvirtual
    void f3(int x) override;               // overrides base virtual
    virtual void f5(const std::string& s); // new virtual
    virtual ~D1();                         // overrides base virtual
    ...
};
```

It yields a vtbl like this:

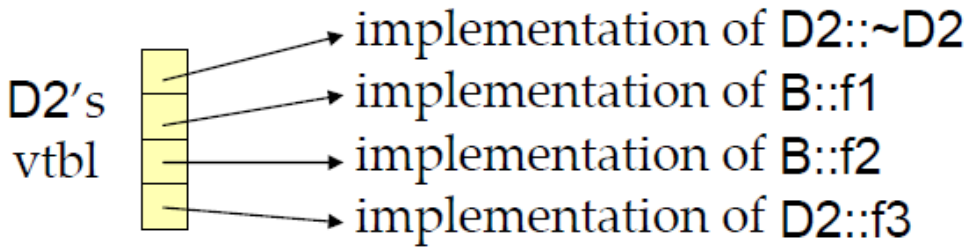


- Note how corresponding function implementations have corresponding indices in the vtbl.

Implementing Virtual Functions (cont'd)

A second derived class would be treated similarly:

```
class D2: public B {
  public:
    D2();
    void f3(int x) override;
    ...
};
```



- D2's destructor is automatically generated by the compiler.

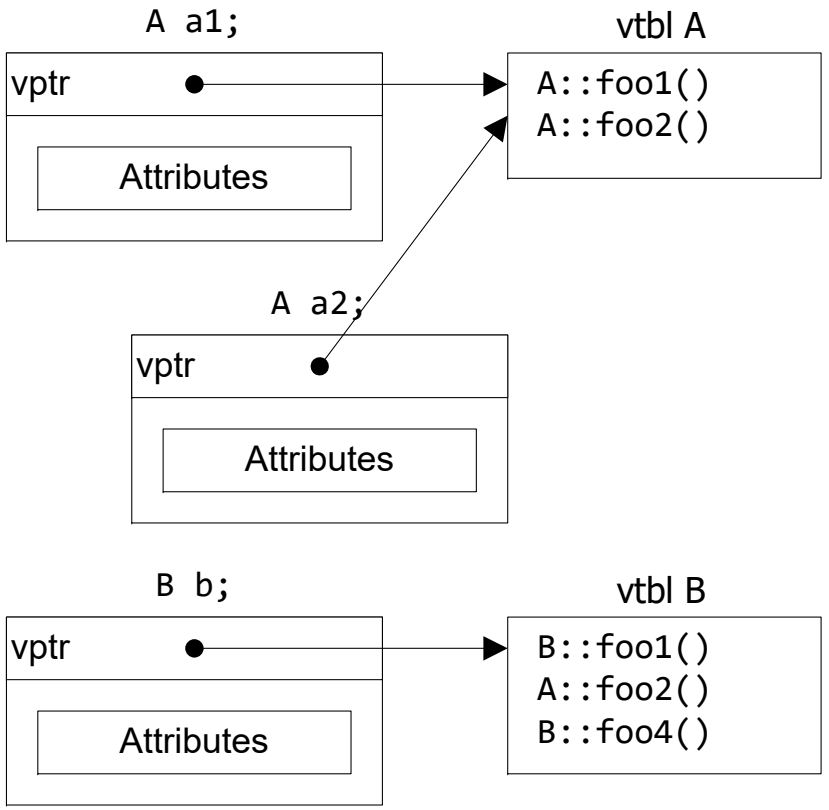
Repräsentation polymorpher Objekte im Speicher

- In der *Virtual Function Table* (*vtbl*) vermerkt das System der Reihe nach die Adressen der für eine Klasse gültigen **virtuellen** Elementfunktionen
- Das System legt für jede polymorphe **Klasse** (diese hat mindestens eine virtuelle Methode) eine *vtbl* an
- Jedes **Objekt** einer polymorphen Klasse enthält einen *Virtual Table Pointer (vptr)*, welcher auf die *vtbl* der entsprechenden Klasse zeigt

Beispiel für *vtbl*

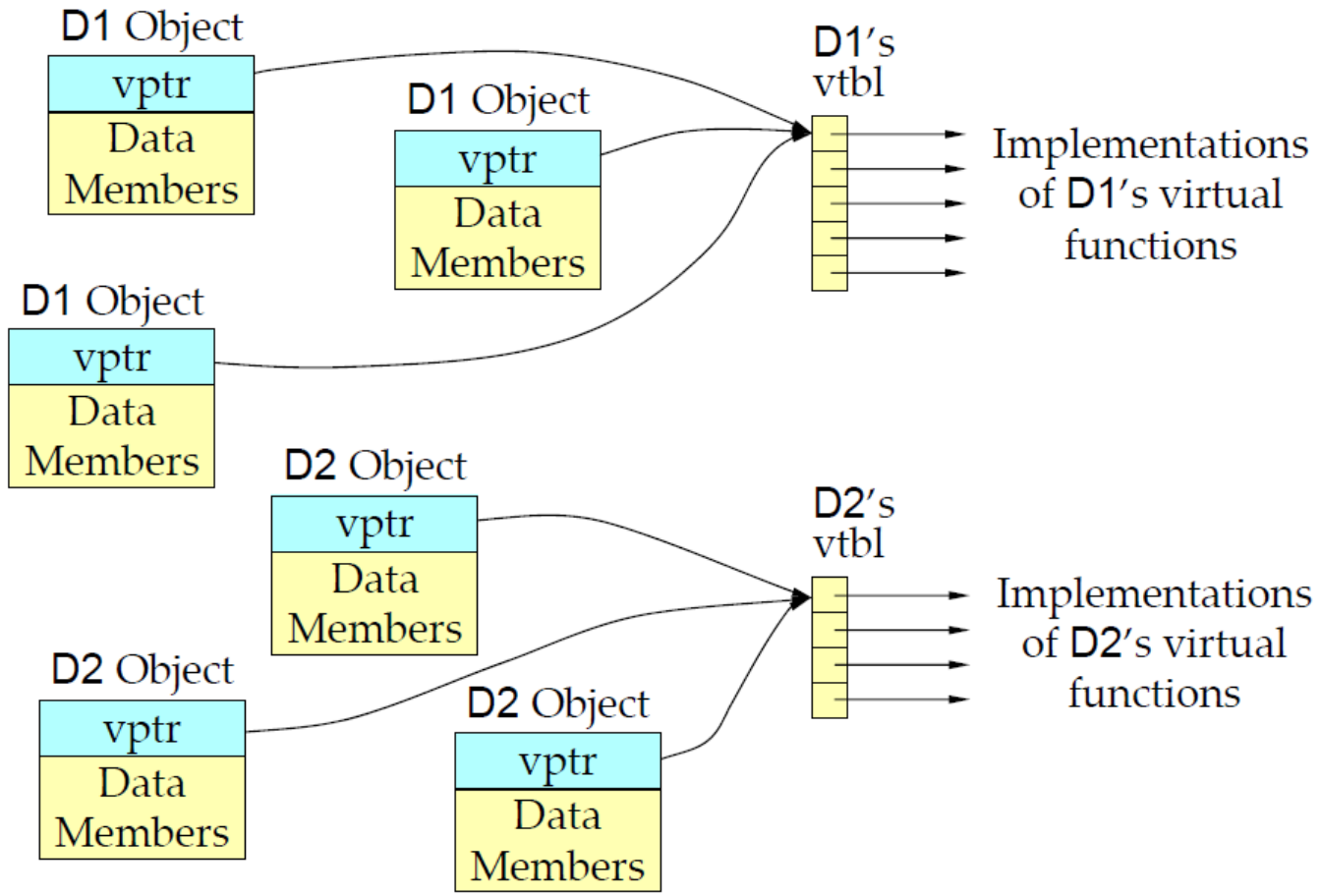
```
class A
{
  public:
    virtual void foo1();
    virtual void foo2();
    void foo3();
};

class B : public A
{
  public:
    void foo1() override;
    virtual void foo4();
};
```



Implementing Virtual Functions (cont'd)

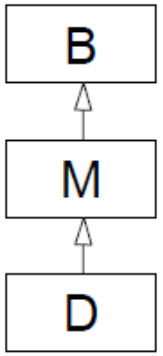
Vptrs point to vtbls:



Implementing Virtual Functions (cont'd)

vptrs are set by code compilers insert into constructors and destructors.

- In a hierarchy, each class's constructor sets the vptr to point to that class's vtbl
- Ditto for the destructors in a hierarchy.



Compilers are permitted to optimize away unnecessary vptr assignments.

- E.g., vptr setup for a D object could look like this:

```
D obj;
```

Set vptr to B's vtbl; // may be optimized away

Set vptr to M's vtbl; // may be optimized away

Set vptr to D's vtbl;

...

Set vptr to M's vtbl; // may be optimized away

Set vptr to B's vtbl; // may be optimized away

Implementing Virtual Functions (cont'd)

Consider this C++ source code:

```
void makeACall(B* pB)
{
  pB->f1();
}
```

The call to f1 yields code equivalent to this:

```
(*pB->vptr[1])(pB); // call the function pointed to by vtbl entry 1 in the vtbl
                    // pointed to by pB->vptr; pB is passed as the “this” pointer
```

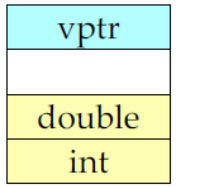
One implication:

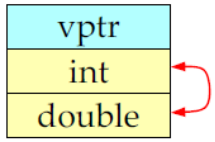
- When a virtual function changes, every caller must recompile!
  - e.g., if the function’s order in the class changes i.e., its compiler-assigned number.
  - e.g., if the function’s signature changes.

Implementing Virtual Functions (cont'd)

Size penalties:

- vptr makes each object larger
  - Alignment restrictions could force padding
    - Reordering data members often eliminates problem
- Per-class vtbl increases each application's data space





Speed penalties:

- Call through vtbl slower than direct call, but usually only by a few instructions
- Inlining usually impossible:
  - This is often inherent in a virtual call

But compared to C alternatives:

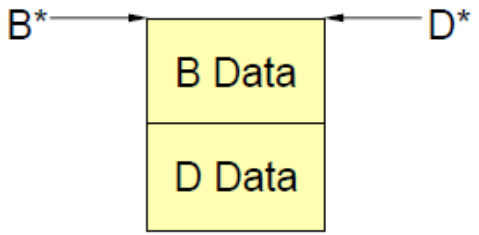
- ***Faster and smaller* than if/then/else or switch-based techniques**
- **Guaranteed to be right**

Object Addresses under Multiple Inheritance (MI)

Under SI, we can generally think of object layouts and addresses like this:

```
class B { ... };
class D: public B { ... };
```

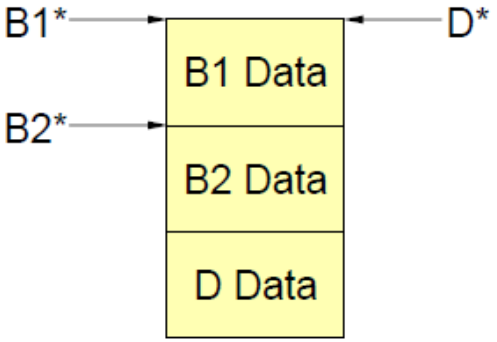
- An exception (with some compilers) is when D has virtual functions, but B doesn't.



Under MI, it looks more like this:

```
class B1 { ... };
class B2 { ... };
class D: public B1, public B2 { ... };
```

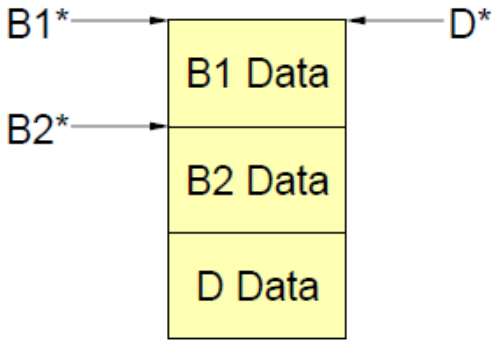
- D objects have multiple addresses:
  - One for B1* and D* pointers.
  - Another for B2* pointers.



Object Addresses under Multiple Inheritance (MI) (cont'd)

There is a good reason for this:

```
void f(B1* pb1); // expects pb1 to point
                 // to the top of a B1
void g(B2* pb2); // expects pb2 to point
                 // to the top of a B2
```



Some calls thus require *offset adjustments*:

```
D* pd = new D;    // no adjustment needed
f(pd);            // no adjustment needed
g(pd);            // requires D* ⇒ B2* adjustment
B2* pb2 = pd;     // requires D* ⇒ B2* adjustment
```

Proper adjustments require proper type information:

```
if (pb2 == pd) … // test succeeds (pd converted to B2*)
if ((void*)pb2 == (void*)pd) … // test fails (why?)
```

Wieso ist `virtual` notwendig?

- In Java sind alle (nicht `static`, nicht `final`) Methoden dynamisch gebunden, in C++ nur die Elementfunktionen, die `virtual` deklariert sind.
  - Die dynamische Bindung erfordert einen Overhead für Laufzeitunterstützung und Interpretation, dieser wird in Java akzeptiert.
  - C++ versucht, möglichst effizient zu sein. Der Programmierer soll sehen, wenn etwas Effizienz kostet.

- Objekte mit `virtual` Memberfunktionen benötigen mehr Overhead als solche ohne

- Aufruf von `virtual` Memberfunktionen erfolgt indirekt durch Nachschlagen in vtbl, die zur konkreten Klasse gehört.