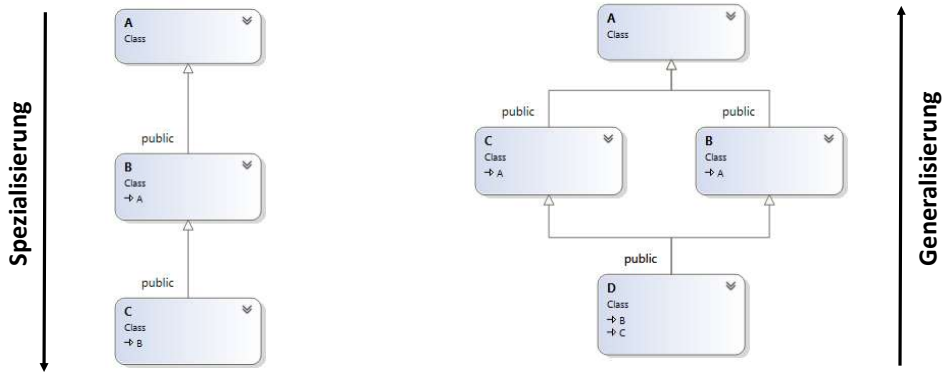
Kurseinheit 5: Vererbung 1

1. Grundprinzipien der Vererbung
2. Arten der Vererbung
3. Abbildung Vererbung im Speicher
4. Überdeckung
5. Friends
6. Typumwandlung

1. Grundprinzip der Vererbung

(Einfach-)Vererbung und Mehrfachvererbung

Klassen können Ihre Eigenschaften an weitere Klassen vererben. Wichtig: Die spezielle Notation in UML (Pfeilart beachten)! **Nur dies ist richtig:** △



Empfehlung: Maximal **sieben Hierarchien**, ansonsten werden diese unübersichtlich

Mehrfachvererbung ist in C#/Java nicht möglich. Dort werden Schnittstellen (Interfaces) eingesetzt, um diese Funktionalität abzubilden.

1. Grundprinzip der Vererbung

Attribute: private, public und protected!

Symbole für Sichtbarkeiten:

public

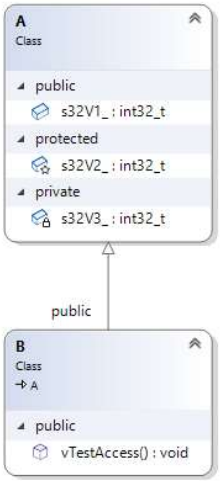
protected

private

Rechte Maustaste auf Class Diagramm: Group Members -> Group by Access

Rechte Maustaste auf Class Diagramm: Change Members Format -> Display Full Signature

Elterklasse



Kindklasse

Die Kindklasse erbt alle Eigenschaften der Elterklasse (Attribute und Methoden).

Allerdings hat die Kindklasse nur Zugriff auf die geerbten public- und protected Eigenschaften. Die Kindklasse kann **nicht** auf die geerbten private Eigenschaften zugreifen.

```
void B::vTestAccess(void)
{
s32V1_ = 42;
s32V2_ = 73;
//s32V3_ = 99;
}
```

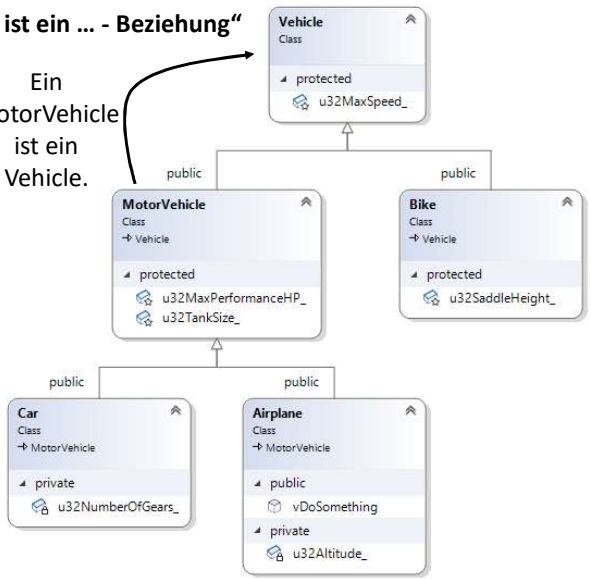
Die neue Sichtbarkeit protected ist somit nur bei Vererbung relevant.

1. Grundprinzip der Vererbung

Praktisches Beispiel (1)

„.. ist ein … - Beziehung“

Ein MotorVehicle ist ein Vehicle.



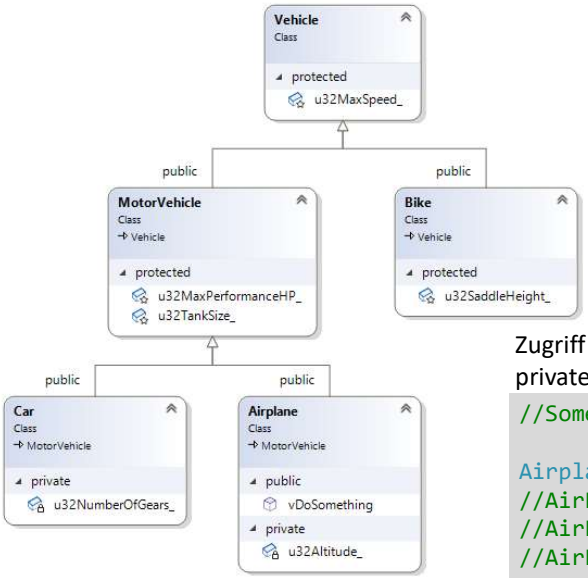
Elterklassen stellen teilweise nur Eigenschaften (Methoden und Attribute) zur Verfügung, können folglich auch unvollständig sein. Dann ist eine Instanziierung von Objekten auch nicht sinnvoll (Vehicle, MotorVehicle).

Wie dies verhindert werden kann, wird in der nächsten Kurseinheit beschrieben (Abstrakte Klasse).

```
sizeof(Vehicle):      ____
sizeof(MotorVehicle): ____
sizeof(Bike):         ____
sizeof(Car):          ____
sizeof(Airplane):     ____
```

1. Grundprinzip der Vererbung

Praktisches Beispiel (2)



Zugriff innerhalb der Klasse auf geerbte protected Attribute ist möglich.

```
void Airplane::
vDoSomething(void)
{
u32Altitude_ = 5000U;
u32TankSize_ = 10000U;
u32MaxSpeed_ = 700U;
}
```

Zugriff von außen auf protected (oder private) Attribute **nicht** möglich.

```
//Somewhere in main

Airplane AirForceOne;
//AirForceOne.u32Altitude_ = 5000U;
//AirForceOne.u32TankSize_ = 10000U;
//AirForceOne.u32MaxSpeed_ = 700U;
```

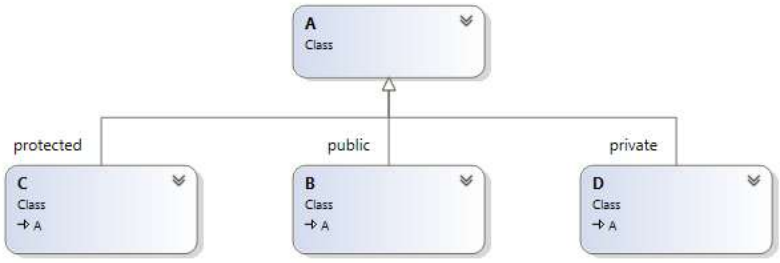

2. Arten der Vererbung

Drei Möglicheiten: public, protected, private

In C++ gibt es drei Arten der Vererbung! In Java/C# gibt es nur eine Art. Default ist bei class private und bei struct public.

```
class Airplane : public|protected|private MotorVehicle
```

Üblicherweise wird in C++ die public-Vererbung angewendet (praktische Relevanz).



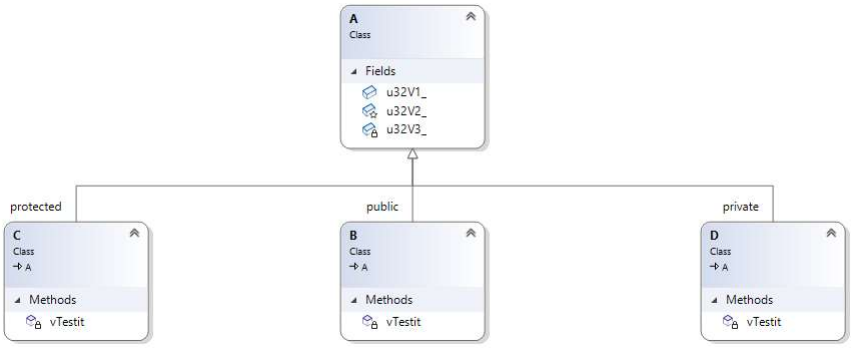
2. Arten der Vererbung

Sichtbarkeiten - Überblick

Vererbung	Element der Basisklasse	Element in der abgeleiteten Klasse	Zugriff in der abgeleiteten Klasse
public	public →	public	zugreifbar
	protected →	protected	zugreifbar
	private →	private	nicht zugreifbar
protected	public →	protected	zugreifbar
	protected →	protected	zugreifbar
	private →	private	nicht zugreifbar
private	public →	private	zugreifbar
	protected →	private	zugreifbar
	private →	private	nicht zugreifbar

2. Arten der Vererbung

Sichtbarkeiten - Beispiel




```
void C::vTestit(void)
{
   u32V1_ = 1U;
   u32V2_ = 2U;
   //u32V3_ = 3U;
}
```

```
void B::vTestit(void)
{
   u32V1_ = 1U;
   u32V2_ = 2U;
   //u32V3_ = 3U;
}
```

```
void C::vTestit(void)
{
   u32V1_ = 1U;
   u32V2_ = 2U;
   //u32V3_ = 3U;
}
```

2. Arten der Vererbung

Sichtbarkeiten – Was übersehen?

Aus Kurseinheit 2: Kopierkonstruktor

```
class Car
{
public:
   Car(uint32_t u32NewMaxSpeed);
   Car(void);
   Car(const Car& rCar);
   ~Car();

private:
   uint32_t u32MaxSpeed_;
   char* pac_;
};
```

```
Car::Car(const Car& rCar)
{
   u32MaxSpeed_ = rCar.u32MaxSpeed_;
   pac_ = rCar.pac_;
}
```

Der Kopierkonstruktor erhält eine Referenz auf ein Objekt der gleichen Klasse (hier Car):

Innerhalb dieser Methode kann auf ein private Attribut des anderen Objektes zugegriffen werden.

Folglich sind bei Sichtbarkeiten drei Fälle zu unterscheiden:

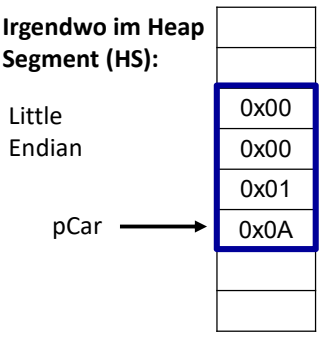
- Zugriff von außerhalb (nicht von einer Methode der gleichen Klasse)
- Zugriff innerhalb der Vererbungshierarchie
- Zugriff von einer Methode der gleichen Klasse auf ein Objekt dieser Klasse. Sonst würde der Kopierkonstruktor nicht funktionieren.

3. Abbildung Vererbung im Speicher

Speicher (noch ohne Vererbung)

```
class Car
{
public:
   Car() = default;
   void vSetMaxSpeed(uint32_t u32NewMaxSpeed);
   uint32_t u32GetMaxSpeed(void);
   static uint32_t u32GetNumberOfInstances(void);

protected:

private:
   uint32_t u32MaxSpeed_;
   static uint32_t u32NumberOfInstances_c;
};
```

```
Car* pCar = new Car{};
pCar->vSetMaxSpeed(0x10A);
```



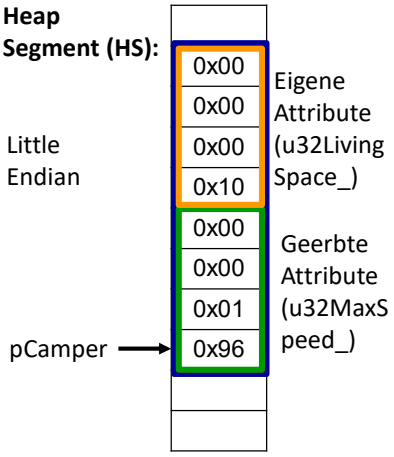
- Klassenmethoden und Methoden liegen im Code Segment (**CS**)
- Klassenvariablen liegen im Data Segment (**DS**)
- Attribute liegen je nach Allokierung im Heap Segment (**HS**), Stack Segment (**SS**) oder im Data Segment (**DS**)

3. Abbildung Vererbung im Speicher

Speicher (mit Vererbung)

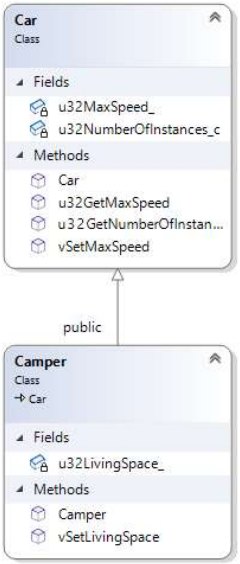
```
class Car
{
public:
   Car() = default;
   void vSetMaxSpeed(uint32_t u32NewMaxSpeed);
   uint32_t u32GetMaxSpeed(void);
   static uint32_t u32GetNumberOfInstances(void);
protected:

private:
   int32_t u32MaxSpeed_;
   static uint32_t u32NumberOfInstances_c;
};

class Camper : public Car
{
public:
   Camper() = default;
   void vSetLivingSpace(uint32_t u32NewLiving);
private:
   uint32_t u32LivingSpace_;
};
```

```
Camper* pCamper = new Camper{};
pCamper->vSetMaxSpeed(0x96);
pCamper->vSetLivingSpace(0x10);
```



3. Abbildung Vererbung im Speicher

Speicher (mit Vererbung)



Dynamische Allokation

1 x Camper
2 x Car

Statische Allokation

- **Global, modulglobal, statisch lokal**

  2 x Camper
  1 x Car

- **Lokal**

  1 x Camper
  1 x Car

Code Segment (**CS**)
- 1 x Car() (Konstruktor)
- 1 x Camper() (Konstruktor)
- Alle weiteren Default Kon- und Destruktoren
- 1 x u32GetMaxSpeed ()
- 1 x u32GetNumberOfInstances()
- 1 x vSetLivingSpace()

Data Segment (**DS**)
- 1 x u32NumberOfInstances_c
- 3 x u32MaxSpeed_
- 2 x u32LivingSpace_

Heap Segment (**HS**)
- 3 x u32MaxSpeed_
- 1 x u32LivingSpace_

Stack Segment (**SS**)
- 2 x u32MaxSpeed_
- 1 x u32LivingSpace_

4. Überdeckung

Prinzip

```
class A
{
public:
   int32_t s32V1;
   void vTestDoIt(void)
   {
      std::cout << "vTestDoIt of A" << std::endl;
   }
};

class B : public A
{
public:
   int32_t s32V1;
   void vTestDoIt(uint32_t u32V)
   {
      std::cout << "vTestDoIt of B" << std::endl;
   }
};
```

Exkurs / Wiederholung:
Implementierung in h-Datei ist **implizites Inlining**.
Sinnvoll bei kurzen Methoden.

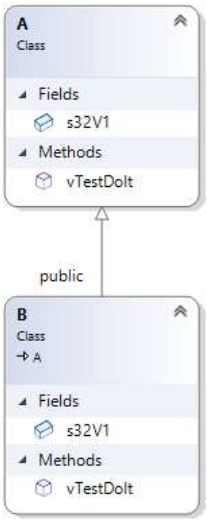
Attribute und Methoden können in abgeleiteten Klassen auch **überdeckt** (**Redefinition**) werden.

Sollte eher vermieden werden.
Könnte aber ungewollt oft passieren:

- Attribute a, b, …
- Generische Methoden wie Print, Store, Output, GetValue, …

4. Überdeckung

Attribute



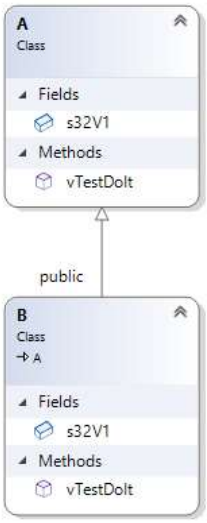

```
B b;
std::cout << sizeof(b) << std::endl;
std::cout << std::hex;
std::cout << &b << std::endl;

//Access to own s32V1
b.s32V1 = 42;

//Access to covered s32V1
//Only possible for protected/public
b.A::s32V1 = -1;
```

s32V1 gibt es zweimal (sizeof)

Hinweis
Beim Memoryfenster sind kleine Adressen oben.
Bei der normalen Speicherdarstellung sind höhere Adressen oben.

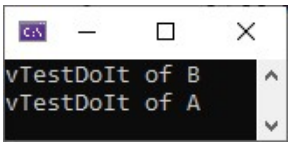
4. Überdeckung

Methoden



Die Methode vTestDoIt der abgeleiteten Klasse B **überdeckt** nun die geerbte Methode. **Anzahl und Datentyp der formalen Parameter sowohl der Rückgabewert müssen nicht übereinstimmen.**

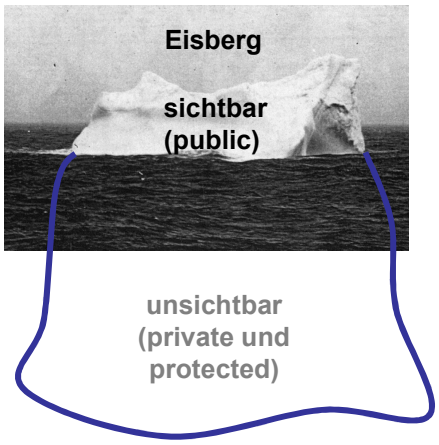
```
//Access to own vTestDoIt
b.vTestDoIt(42U);

//Access to covered vTestDoIt
//Only possible for protected/public
b.A::vTestDoIt();
```



Die Funktion vTestDoIt gibt es zweimal im Code Segment.

5. Friends

Prinzip: Aufhebung der Kapselung



Prinzip der **Kapselung** soll in Ausnahmefällen aufgehoben werden:

- Friend-Funktionen
- Friend-Klassen

Statt großzügig public zu verwenden, wird private/protected weiterhin verwendet und nur im Einzelfall wird mittels dem Schlüsselwort friend Zugriff auf private Elemente gewährt.

Sparsam damit umgehen – für Sonderfälle verwenden! Ohne friend müssten aufwendige public Getter-Funktionen implementiert werden, um auf private/protected Attribute zuzugreifen.

Friend-Funktionen

```
class Car
{
public:
   Car() = default;
   void vSetMaxSpeed(uint32_t u32NewMaxSpeed);
   uint32_t u32GetMaxSpeed(void);
   static uint32_t u32GetNumberOfInstances(void);

private:
   uint32_t u32MaxSpeed_;
   static uint32_t u32NumberOfInstances_c;

friend void vBreakIn(Car& rcCar);
};
```

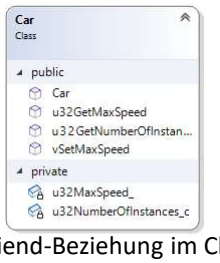
Kein Bezug zu private

```
void vBreakIn(Car& rcCar)
{
   rcCar.u32MaxSpeed_ = 0U;
}
```

vBreakIn kann auf alle Methoden und Eigenschaften von Car zugreifen.

Friend-Klassen

```
class BreakIn; //Forward Declaration
class Car
{
public:
Car() = default;
void vSetMaxSpeed(uint32_t u32NewMaxSpeed);
uint32_t u32GetMaxSpeed(void);
static uint32_t u32GetNumberOfInstances(void);

private:
uint32_t u32MaxSpeed_;
static uint32_t u32NumberOfInstances_c;

friend class BreakIn;
};
```



Friend-Beziehung im Class Designer nicht dargestellt.


```
void BreakIn::DoIt(Car& rcCar)
{
    rcCar.u32MaxSpeed_ = 0U;
}
```

Alle Methoden von BreakIn können auf alle Methoden und Eigenschaften von Car zugreifen.

Liskovsches Substitutionsprinzip

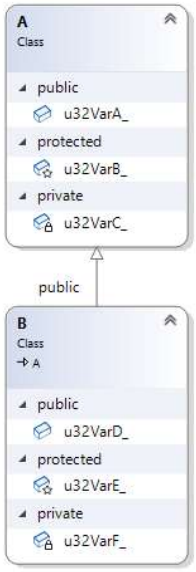
Das **Liskovsche Substitutionsprinzip** (**LSP**) oder **Ersetzbarkeitsprinzip** ist ein Kriterium in der objektorientierten Programmierung, das die Bedingungen zur Modellierung eines Datentyps für seinen Untertyp angibt. Es besagt, dass ein Programm, das Objekte einer Basisklasse *T* verwendet, auch mit Objekten der davon abgeleiteten Klasse *S* korrekt funktionieren muss, ohne dabei das Programm zu verändern.
Das Liskovsche Substitutionsprinzip wurde erstmals 1987 von Barbara Liskov auf einer Konferenz *Data abstraction and hierarchy* vorgestellt und wurde 1993 von Barbara Liskov und Jeannette Wing formuliert.

Sei q(x) eine beweisbare Eigenschaft von Objekten x des Typs T.
Dann soll q(y) für Objekte y des Typs S wahr sein, wobei S ein Untertyp von T ist.“

https://de.wikipedia.org/wiki/Liskovsches_Substitutionsprinzip

Die programmiersprachlichen Mittel zur Umsetzung dieses Prinzips sind **Polymorphismus** und **Dynamisches Binden** (siehe später).

Prinzip



Überall dort, wo

- ein Objekt der Basisklasse
- ein Zeiger auf ein Objekt der Basisklasse
- eine Referenz auf ein Objekt der Basisklasse

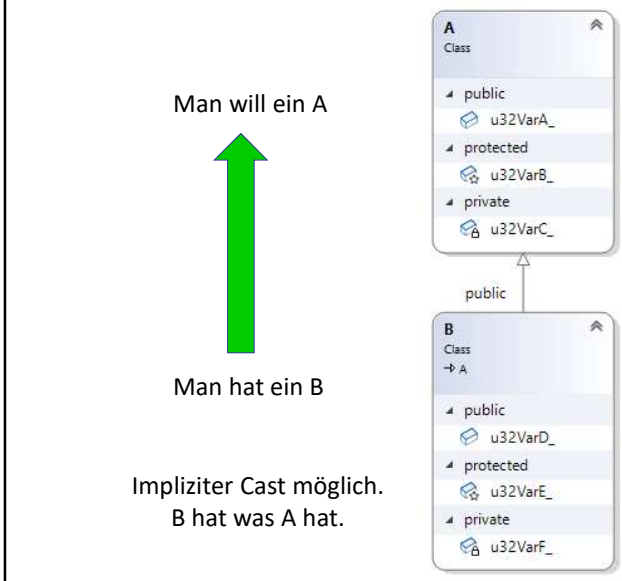
verlangt wird, kann auch ein

- ein Objekt einer abgeleiteten Klasse
- ein Zeiger auf ein Objekt einer abgeleiteten Klasse
- eine Referenz auf ein Objekt der abgeleiteten Klasse

übergeben werden.

Der cast geschieht dabei implizit!

Wann funktioniert ein impliziter Cast



Man will ein A

Man hat ein B

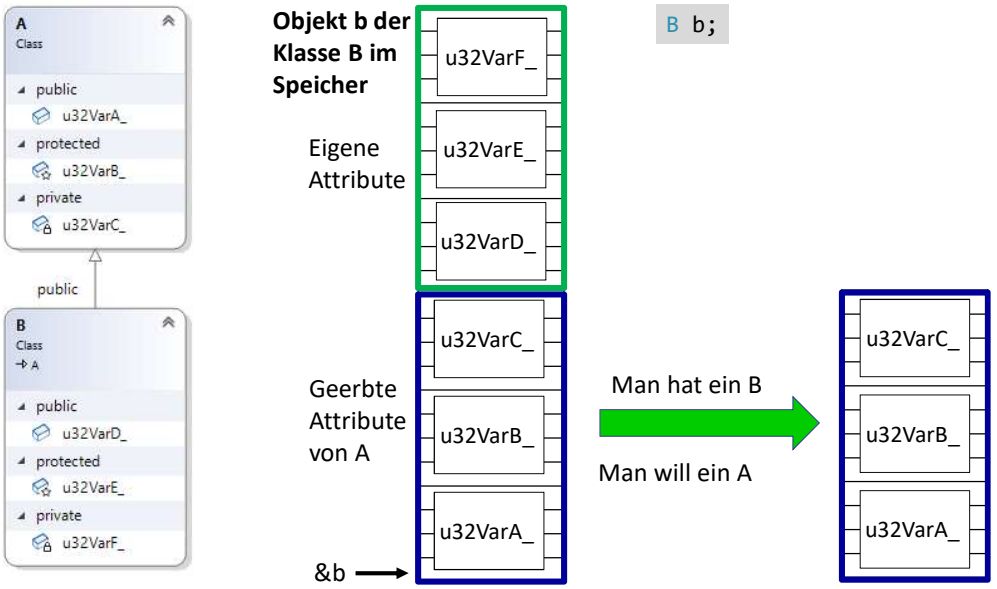
Impliziter Cast möglich.
B hat was A hat.

Man hat ein A

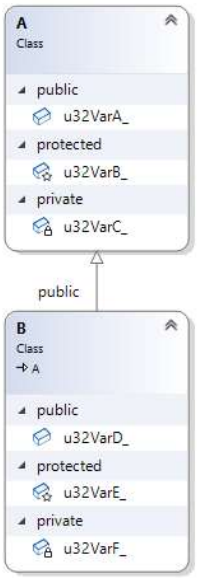
Man will ein B

Impliziter Cast **nicht** möglich.
A hat **nicht** alles was B hat.

Warum funktioniert ein impliziter Cast? - Memory



Beispiele - Übergabeparameter




```
A a;
B b;
A* pa = new A;
B* pb = new B;
A& ra = a;
B& rb = b;
```

```
void vf1(A);
void vf2(B);
void vf3(A*);
void vf4(B*);
void vf5(A&);
void vf6(B&);
```

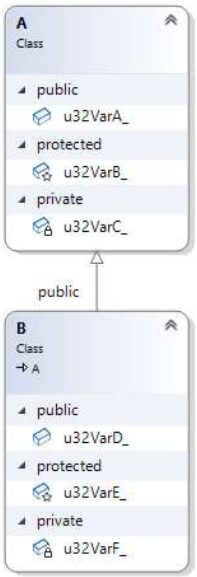
Möglich (T, C) oder nicht (F)?

vf1(a)		vf3(pa)		vf5(ra)	
vf1(b)		vf3(pb)		vf5(rb)	
vf2(a)		vf4(pa)		vf6(ra)	
vf2(b)		vf4(pb)		vf6(rb)	

T: trivial (gleicher Typ) **C**: Cast möglich **F**: Fehler

Nur bei public Vererbung ist impliziter Cast möglich.

Beispiele - Zuweisungen




```
A a;
B b;
A* pa = new A;
B* pb = new B;
A& ra = a;
B& rb = b;
```

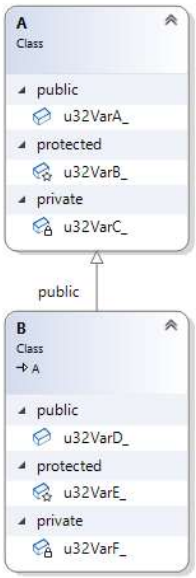
Möglich (T, C) oder nicht (F)?

a = b		pa = &a		ra = b	
b = a		pb = &b		rb = a	
pa = pb		pa = &b		pa=&ra	
pb = pa		pb = &a		pa=&rb	

T: trivial (gleicher Typ) **C**: Cast möglich **F**: Fehler

Nur bei public Vererbung ist impliziter Cast möglich.

Überdeckungen (implizite Casts)

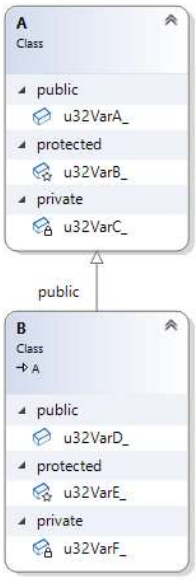



```
B b1;
B b2;
A* paX = &b1; // Impliziter Cast
B* pbX = &b2;
A& raX = b1;  // Impliziter Cast
B& rbX = b2;
```

```
//std::cout << paX->u32VarD_ << std::endl; Error
std::cout << pbX->u32VarD_ << std::endl;
//std::cout << raX.u32VarD_ << std::endl; Error
std::cout << rbX.u32VarD_ << std::endl;
```

Mit einem Zeiger, einer Referenz oder einer Objektvariablen kann man nur auf die Elemente zugreifen, die durch den jeweiligen Typ definiert sind, auch wenn sich „dahinter“ ein abgeleitetes Objekt befindet.
Ausnahme (wichtig, siehe später): Virtuelle Funktionen!!!

Überdeckungen (explizite Casts)




```
B b1;
A* paX = &b1; // Impliziter Cast
```

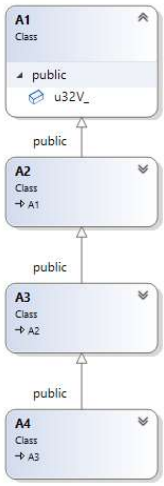
```
//std::cout << paX->u32VarD_ << std::endl; Error
```

```
std::cout << (static_cast<B*>(paX))->u32VarD_ <<
std::endl; // Expliziter Cast
```

Expliziter Cast ist immer möglich. Sind die Typen nicht kompatibel, führt dies zu Laufzeitfehlern (Speicher wird überschrieben, etc.)! Risiko wird an den Programmierer delegiert!

6. Typumwandlung

Hierarchien

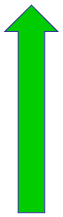


Man will ein A1

Man hat ein A2

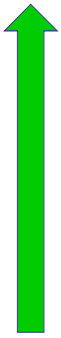
```
A1 a1;
A2 a2;
A3 a3;
A4 a4;

a1 = a2;
a1 = a3;
a1 = a4;
```

Man will ein A1



Man hat ein A3

Man will ein A1

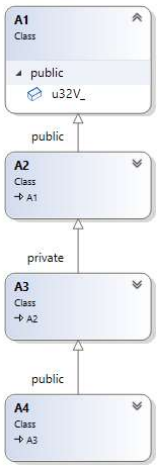


Man hat ein A4

Implizite Casts sind über mehrere Vererbungshierarchien möglich. Eine public-Vererbung ist allerdings hier auch wieder die Voraussetzung!

6. Typumwandlung

Hierarchien



Man will ein A1

Man hat ein A2

Unterbrechung

Man will ein A3

Man hat ein A4

```
a1 = a2;
//a1 = a3; Fehler
//a1 = a4; Fehler

//a2 = a3; Fehler
//a2 = a4; Fehler

a3 = a4;
```

Eine private oder protected Vererbung unterbricht die Kette der möglichen Casts! So ist a2.u32V = 1U möglich, es ist a3.u32V = 1U oder ist a4.u32V = 1U nicht mehr möglich, da die Variable u32V_ in den Objekten a3 und a4 jetzt private ist.