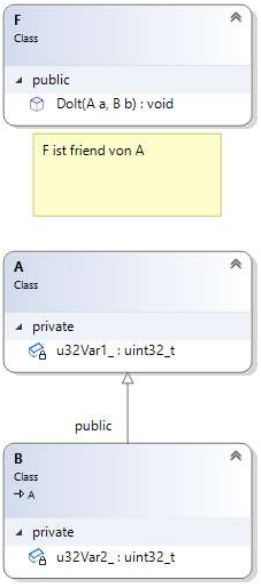
Kurseinheit 6: Vererbung 2

1. Was wird nicht vererbt?
2. Reihenfolge Kon- und Destruktoren
3. Virtuelle Funktionen
4. Dynamischer Polymorphismus
5. Mehrfachvererbung

1. Was wird nicht vererbt?

Friend-Eigenschaft




```
class F
{
public:
   void DoIt(A a, B b)
   {
   a.u32Var1_ = 1;

   b.u32Var1_ = 1;
   //b.u32Var2_ = 2;
   }
};
```

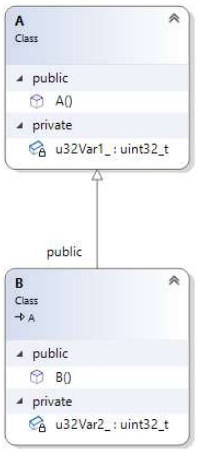
```
class A
{
private:
   uint32_t u32Var1_;

friend class F;
};
class B : public A
{
private:
    uint32_t u32Var2_;
};
```

Aus F aus kann nur auf u32Var1_ zugegriffen werden, da dieses von der friend-Klasse A vererbt wird (dto. bei friend-Funktionen).

Eine friend-Eigenschaft wird nicht automatisch an die abgeleitete Klasse weitergegeben. Nur auf den geerbten Attributen bleibt die friend-Beziehung erhalten.

1. Was wird nicht vererbt?

Konstruktoren




```
class A
{
public:
   A() noexcept(true)
   {
      u32Var1_ = 42U;
   }
private:
   uint32_t u32Var1_;
};
class B : public A
{
public:
   B() noexcept(true) : A()
   {
      u32Var2_ = 73U;
   }
private:
   uint32_t u32Var2_;
};
```

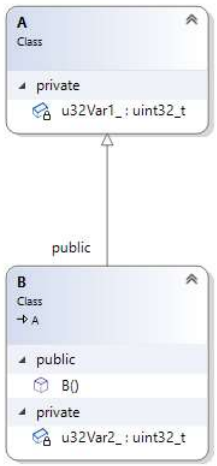
Konstruktoren werden nicht vererbt. Es ist eine **Arbeitsteilung** zu realisieren. Eltern- und Kindklassen initialisieren ihre „eigenen“ Attribute. Kindklasse ruft automatisch den parameterlosen Konstruktor (leerer Konstruktor oder Standardkonstruktor) der Elterklasse auf!

Automatischer **Default**-Aufruf des parameterlosen Konstruktors (**kann auch weggelassen werden**).

Unterschiede zu Java/C#!

1. Was wird nicht vererbt?

Standardkonstruktoren




```
class A
{
private:
   uint32_t u32Var1_;
};
class B : public A
{
public:
   B() noexcept(true)
   {
      u32Var2_ = 73U;
   }
private:
   uint32_t u32Var2_;
};
```

Falls eine Klasse keine eigenen Konstruktor hat, wird ein leerer Default-Konstruktor zur Verfügung gestellt.

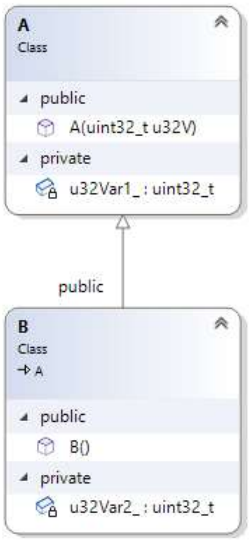
Zur Dokumentation könnte auch folgendes geschrieben werden:

```
class A
{
public:
   A() = default;
// more code
```

Falls die Elterklasse keinen Konstruktor explizit hat, wird der Default-Konstruktor implizit aufgerufen (parameterlos).

1. Was wird nicht vererbt?

Standardkonstruktoren




```
class A
{
public:
   A(uint32_t u32V) noexcept(true)
   {
      u32Var1_ = u32V;
   }
private:
   uint32_t u32Var1_;
};
class B : public A
{
public:
   B() noexcept(true)
   {
      u32Var2_ = 42U;
   }
private:
   uint32_t u32Var2_;
};
```

```
B b;
```

Führt zu einem Compilerfehler!

'A': no appropriate default constructor available.

Warum?

_____

_____

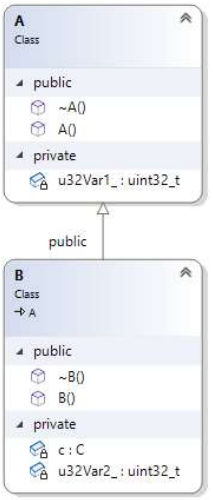
Abhilfe:

_____

_____

1. Was wird nicht vererbt?

Destruktoren



Sachlage im Gegensatz zu Konstruktoren einfacher. Es gibt immer nur einen Destruktor!

```
class B : public A
{
public:
   ~B() {}
// more Code
private:
   C c;
// more Code
};
```

Automatischer Aufruf!

```
class C
{
public:
   ~C() {}
//… more Code
};

class A
{
public:
   ~A() {}
//… more Code
};
```

Destruktoren werden nicht vererbt. Es ist eine **Arbeitsteilung** zu realisieren. Destruktoren der Kindklasse rufen automatisch die der Elternklassen und der Objektvariablen auf.

1. Was wird nicht vererbt?

Zuweisungsoperator

Jede Klasse verfügt über einen Default-Zuweisungsoperator (Operatorüberladen siehe später). Beispiel: **B erbt von A.**

```
B b1;
B b2;
//… more Code
b1 = b2;
```

C++ stellt die Weiterführung von C dar. Schon in C lassen sich Strukturvariablen zuweisen. In C++ können Objekte gleichen Typs (oder Subtyps) einander zugewiesen werden.

```
sB_t sb1;
sB_t sb2;
//… more Code
sb1 = sb2;
```

Fallunterscheidung: B hat …

Fall 1: Keinen eigenen Zuweisungsoperatoren:

Default-Zuweisungsoperator von A wird verwendet

Fall 2: Eigenen Zuweisungsoperator:

Dann muss darin der Aufruf des Zuweisungsoperators von A implementiert werden.

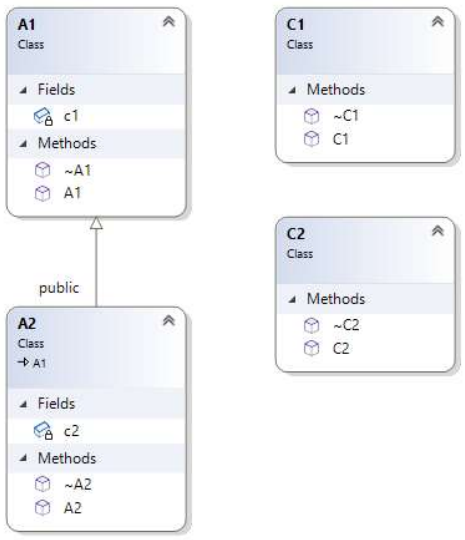
```
B& operator=(const B& rb2)
{
   A::operator=(rb2);
//… more Code
}
```

Funktioniert dank **Liskovschen Substitutionsprinzip**: Zuweisungsoperator von A erwartet eine Referenz auf ein A. Es kann aber auch eine Referenz auf Kindklasse (B) übergeben werden.

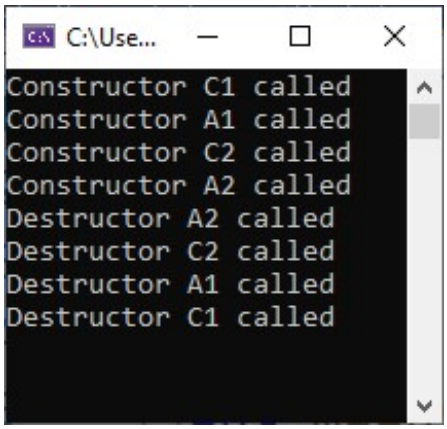
Zuweisungsoperatoren werden nicht vererbt. Es ist eine Arbeitsteilung zu realisieren.

2. Reihenfolge Kon- und Destruktoren

Beispiel

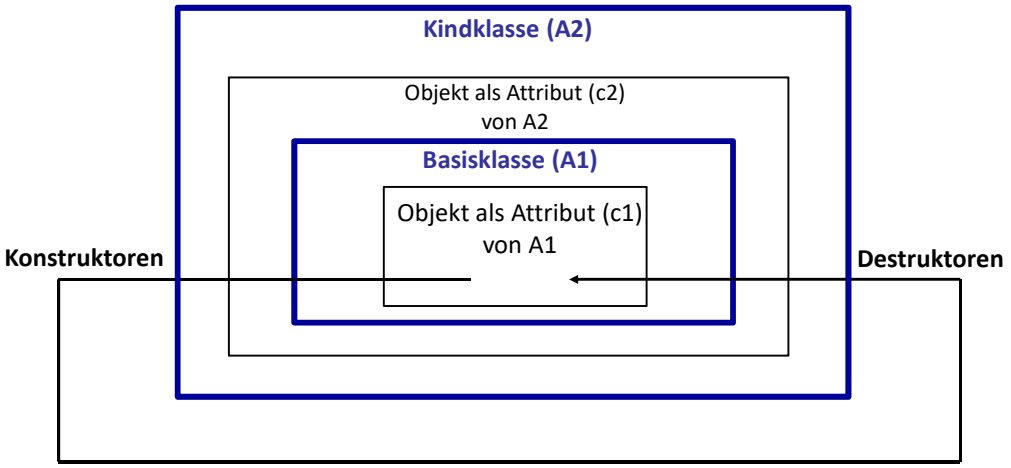



```
A2* pa2 = new A2();
delete pa2;
```



2. Reihenfolge Kon- und Destruktoren

Reihenfolge



3. Virtuelle Funktionen

Unterscheidung

Virtuelle Funktion

```
class V1
{
public:
   void virtual vDoIt(void)
   {

   }
};
```

```
V1* pV1 = new V1;
```

Objekte einer Klasse mit virtuellen Funktionen können instanziiert werden.

Rein virtuelle Funktion (pure virtual)

```
class V2
{
public:
void virtual vDoIt(void) = 0;
};
```

Kennzeichnung durch = 0

```
// V2* pV2 = new V2; // Error
```

Objekte einer Klasse mit **rein** virtuellen Funktionen können **nicht** instanziiert werden. Nur so können in C++ **abstrakte** Klassen oder **Interfaces** generiert werden.

Abstrakte Klasse

V2 ist abstrakt

```
class V2
{
public:
   void virtual vDoIt(void) = 0;
};
```

V3 ist abstrakt
pure virtual wird geerbt!

```
class V3 : public V2
{
public:

};
```

Kein Keyword virtual notwendig

V4 ist nicht abstrakt
void virtual test2()
wird überschrieben

```
class V4 : public V2
{
public:
   void vDoIt(void)
   {

   }
};
```

Von einer abstrakten Klasse können keine Objekte instanziiert werden. In C++ gibt es **kein Keyword abstract** wie in C#/Java. Eine Klasse wird abstrakt, wenn diese mindestens eine rein virtuelle Funktion enthält oder diese geerbt hat und nicht selbst implementiert.

Überschreiben

```
class V1
{
public:
   void virtual vDoIt(void)
   {
      std::cout << "V1 vDoIt" << std::endl;
   }
};

class V5 : public V1
{
public:
   void vDoIt(void) override
   {
      std::cout << "V2 vDoIt" << std::endl;
   }
};
```

Kein virtual in der geerbten Methode. override (ab C++11) dient zur Überprüfung, ob die Funktion überschrieben werden kann. Überschriebene Funktionen müssen **exakt** die gleiche Signatur aufweisen. Sicherer Code!

vDoIt **überschreibt** die geerbte Methode vDoIt. Wäre vDoIt in V1 nicht virtuell würde es sich um eine **Überdeckung** handeln.

Beispiel – Compilerfehler durch override

```
class BaseClass
{
public:
   virtual void funcA();
   virtual void funcB() const;
   virtual void funcC(int = 0);
   void funcD();
};

class DerivedClass : public BaseClass
{
public:
   virtual void funcA() override;

   virtual void funcB() override;

   virtual void funcC( double = 0.0 ) override;

   void funcD() override;
};
```

Welches override erzeugt einen Compilerfehler?
… warum?

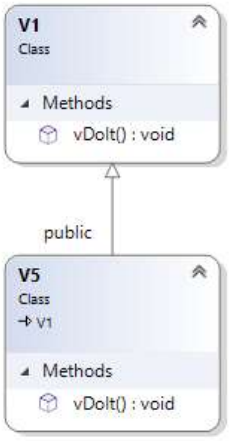
_____

_____

_____

_____

Wiederholung: Überladen, Überdecken und Überschreiben



Class Designer kann virtuelle Funktionen nicht darstellen.

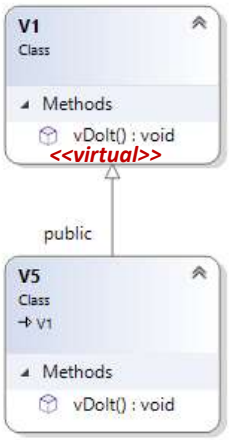
Überladen hat **nichts mit Vererbung** zu tun. Dieses Sprachfeature ist bereits im prozeduralen Teil von C++ enthalten. Realisiert wird dies durch Name Mangling.

Nur **Überdecken** und **Überschreiben** hängen mit Vererbung zu zusammen.

Überdecken	**Überschreiben**
Methode in Elternklasse nicht virtuell	Methode in Elternklasse ist virtuell
Methode in Kindklasse kann unterschiedliche Parameter haben.	Methode in Kindklasse hat gleiche Signatur. override dient zur Sicherheit

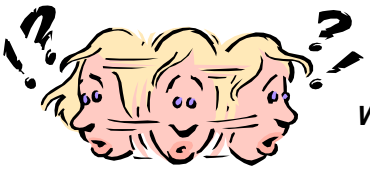
3. Virtuelle Funktionen

Prinzip




```
V1* pV1 = new V5;
//… Some Code
pV1->vDoIt();
```

Liskovsches Substitutionsprinzip



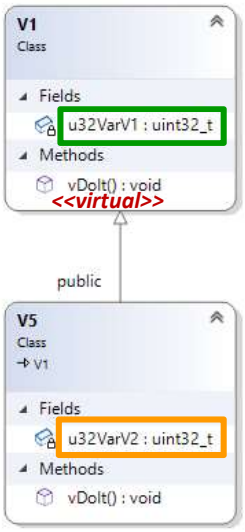
Herr C.C. Gnu

Welches vDoIt( )

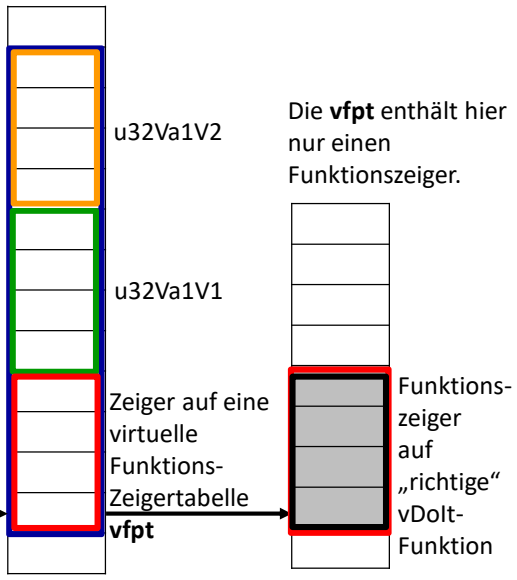
Compiler kann während der Übersetzung nur wissen, dass pv1 auf ein Objekt der Klasse V1 oder ein Objekt einer von V1 abgeleiteten Klasse zeigt. Compiler braucht zusätzliche Informationen für virtuelle Funktionen!

3. Virtuelle Funktionen

Realisierung



Objekt V5 im Speicher



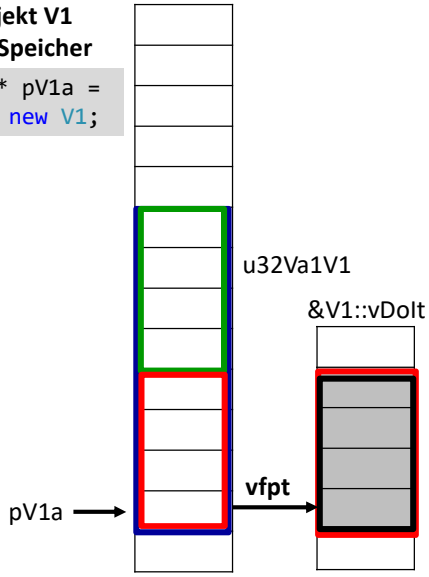

```
V1* pV1 = new V5;
```

3. Virtuelle Funktionen

Abbildung im Speicher

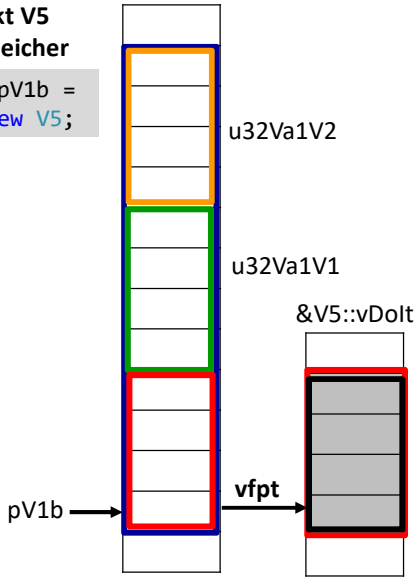
Objekt V1 im Speicher

```
V1* pV1a =
   new V1;
```



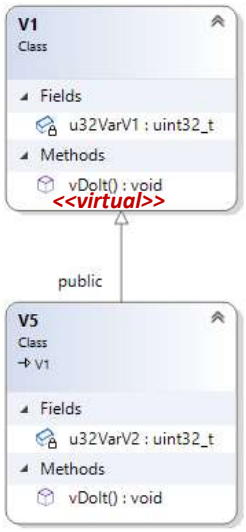
Objekt V5 im Speicher

```
V1* pV1b =
   new V5;
```

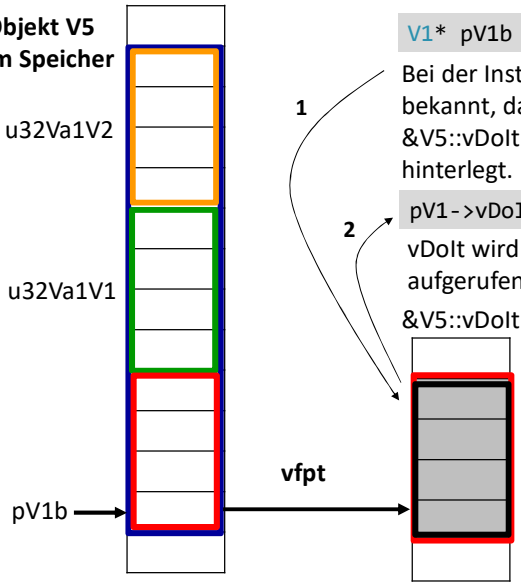


3. Virtuelle Funktionen

vfpt – Anlegen und Zugriff



Objekt V5 im Speicher




```
V1* pV1b = new V5;
```

1 Bei der Instanziierung ist bekannt, dass es ein V5 ist. &V5::vDoIt wird in vfpt hinterlegt.

```
pV1->vDoIt();
```

2 vDoIt wird über die vfpt aufgerufen

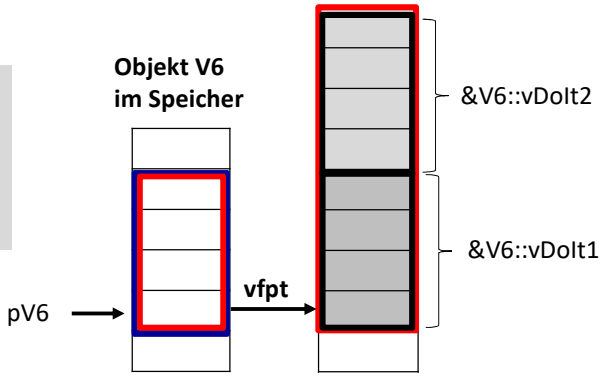
Dynamic oder Late Bindung

Wird ein Funktionsaufruf erst zur Laufzeit eines Programms der tatsächlichen Funktion zugeordnet (hier über Funktionszeiger) so spricht man von **Dynamic Binding** oder **Late Binding**! Dieses geschieht mittels virtueller Funktionen!
Bei nicht virtuellen Funktionen wird der Aufruf bei der Übersetzung „fest verdrahtet“. Man spricht daher auch von **Early Binding** oder **Static Binding**.

Hat eine Klasse mehrere virtuelle Funktionen, so muss für jede ein Funktionszeiger zur Verfügung gestellt werden.

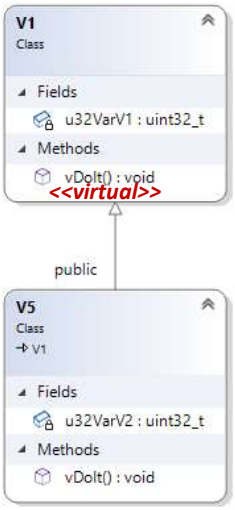
```
class V6
{
public:
   void virtual vDoIt1(void);
   void virtual vDoIt2(void);
};
```

```
V6* pV6 = new V6;
```



Umgehung der Virtualität

Late Binding (Polymorphie-Eigenschaft) kann nur genutzt werden, falls der Aufruf über einen Zeiger oder eine Referenz geschieht.




```
V1* pV1 = new V5;

V5 v5;

v5.vDoIt();


pV1->V1::vDoIt();


pV1->vDoIt();
```

Aufruf geschieht nicht über vfpt, es handelt sich um Early Binding.

Aufruf geschieht nicht über vfpt, mittels des Bereichszugriffsoperators wird über Early Binding die Funktion vDoIt von V1 aufgerufen.

Nur hier Late Binding!

3. Virtuelle Funktionen

Konstruktoren / Destruktoren

Konstruktoren können **nicht** virtual sind. Destruktoren dürfen und **müssen manchmal virtual** sein.

```
class V1
{
};
class V6 : public V1
{
private:
   uint32_t *pau32;
public:
   V6()
   {
      pau32 = new uint32_t[10];
   }
   ~V6( )
   {
      delete [] pau32;
   }
};
```

```
V1 * pv1 = new V6;
```

1. Default-Konstruktor von V1 wird aufgerufen
2. Konstruktor von V6 wird aufgerufen

```
delete pv1;
```

Hier wird nur der Default-Destruktor von V1 aufgerufen. → **Memory Leak**

Vermeidung des Problems: V1 muss einen virtuellen Destruktor haben.

```
virtual ~V1( ) { }
```

Ab jetzt: Destruktor **immer** virtual machen. Sonst vergisst man es später bei Vererbung.

4. Dynamischer Polymorphismus

Definition

„Vielgestaltigkeit“

Ähnliche Objekte verhalten sich im Detail unterschiedlich.
(Beispiel: Klasse C, Gaspedal wird gedrückt)

Zwei Arten von Polymorphismus

Statischer Polymorphismus

Überladen von Funktionen/Methoden
(Name Mangling in C++)

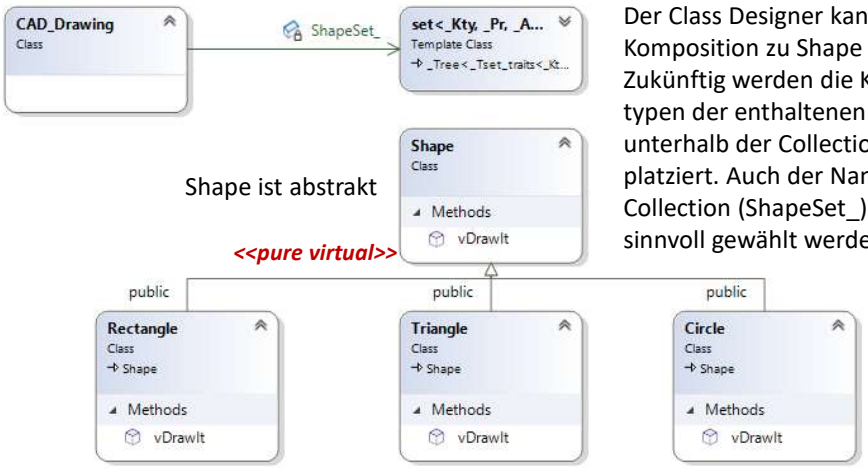
Dynamischer Polymorphismus

Überschreiben von Funktionen
(Virtuelle Methoden, Abstrakte Klassen, Vererbung)

4. Dynamischer Polymorphismus

Standardbeispiel: CAD Zeichnung

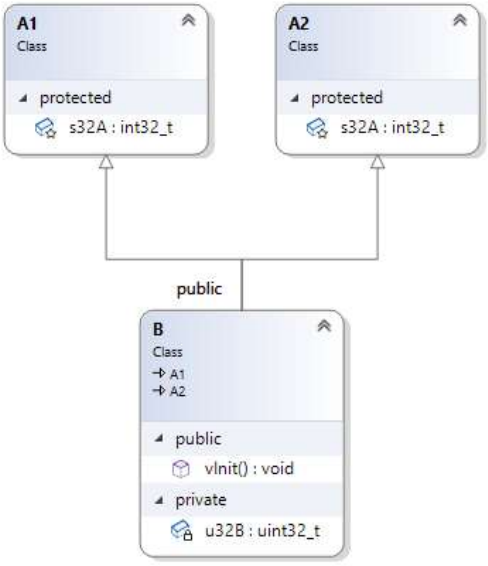
Set ist ein/e Collection/Container aus der Standard Template Library (STL), die ein Element nur einmal enthalten kann (siehe später). Ein CAD_Drawing enthält mehrere Shapes.



Der Class Designer kann keine Komposition zu Shape herstellen. Zukünftig werden die Klassentypen der enthaltenen Objekte unterhalb der Collectionklasse platziert. Auch der Name der Collection (ShapeSet_) sollte sinnvoll gewählt werden.

5. Mehrfachvererbung

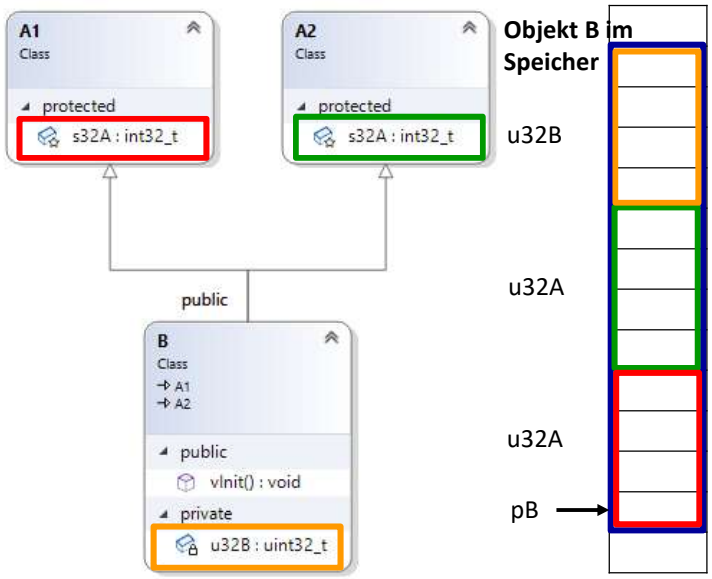
Prinzip




```
class A1
{
protected:
   int32_t s32A;
};
class A2
{
protected:
   int32_t s32A;
};
class B : public A1, public A2
{
public:
   void vInit(void)
   {
   //s32A = 42;
   }
private:
   uint32_t u32B;
};
```

5. Mehrfachvererbung

Speicher




```
B* pB = new B;
```

Mehrdeutigkeiten:
- Attribute
- Methoden

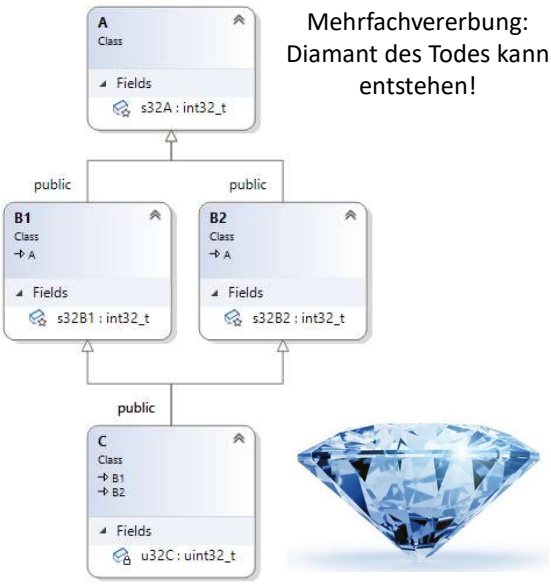
Auch bei unterschiedlichen Datentypen ist dies möglich.

Zugriff dann über Scope-Operator :: möglich

```
void B::vInit(void)
{
   A1::s32A = 42;
}
```


5. Mehrfachvererbung

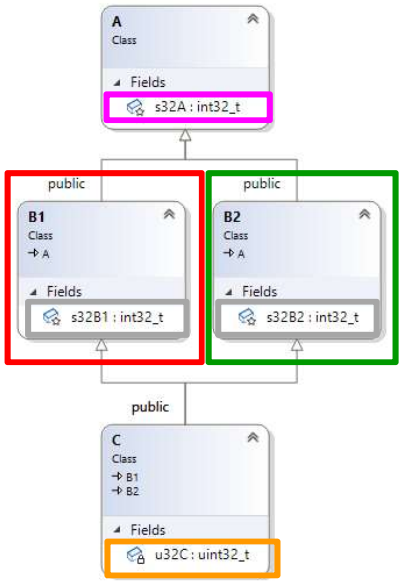
Diamond of Death

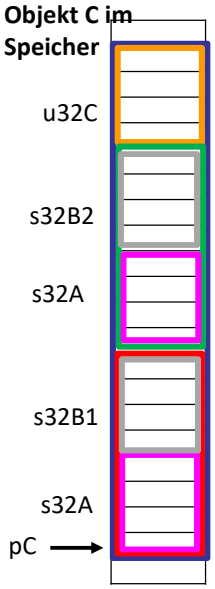


Mehrfachvererbung: Diamant des Todes kann entstehen!

```
class A
{
protected:
   int32_t s32A;
};
class B1 : public A
{
protected:
   int32_t s32B1;
};
class B2 : public A
{
protected:
   int32_t s32B2;
};
class C : public B1, public B2
{
private:
   uint32_t u32C;
};
```

Diamond of Death





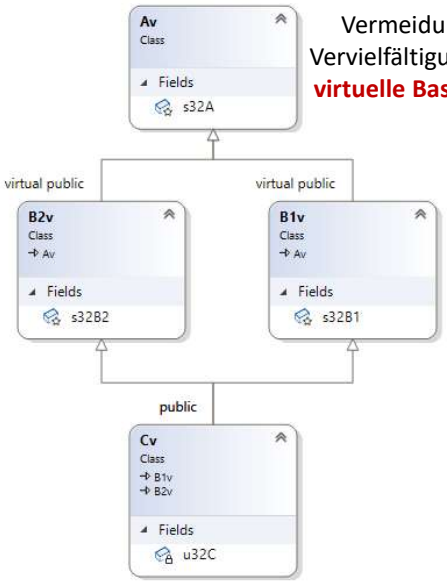

```
C* pC = new C;
```

Das geerbte Element s32A liegt nun doppelt im Speicher.
Problem der **Vervielfältigung.**

Zugriff kann wieder über Scope-Operator erfolgen.

```
void C::vInit(void)
{
   B1::s32A = 42;
   B2::s32A = 73;
}
```


Vermeidung der Vervielfältigung beim Diamond of Death



Vermeidung der Vervielfältigung durch **virtuelle Basisklasse**!

```
class A
{
protected:
   int32_t s32A;
};
class B1 : virtual public A
{
protected:
   int32_t s32B1;
};
class B2 : public virtual A
{
protected:
   int32_t s32B2;
};
class C : public B1, public B2
{
private:
   uint32_t u32C;
};
```

Vermeidung der Vervielfältigung beim Diamond of Death

Übungen:

1. Wie könnten Sie feststellen, dass s32A nur einmal in einem Objekt der Klasse Cv vorhanden ist (keine Vervielfältigung)?

2. Wo würde sich dann s32A im Speicher befinden?