FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Teil 5 – Objektorientierung 3 | Polymorphie

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de

Objektorientierung 3

Inhalt

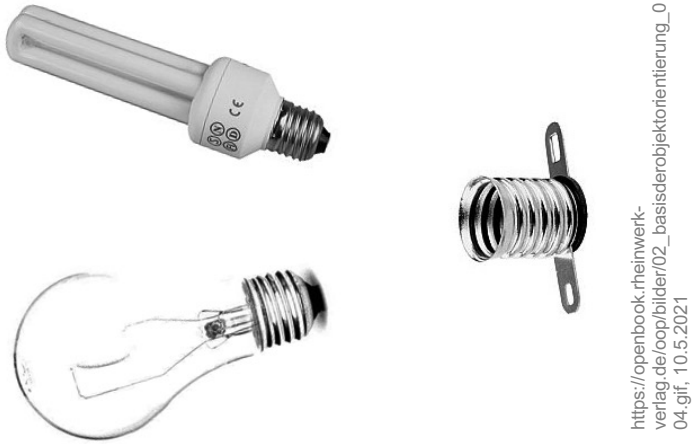
FH MÜNSTER
University of Applied Sciences

- Überladung (Statische Polymorphie)
  - Überladen von Methoden
  - Operator-Überladung in C++
- Polymorphie (Dynamische Polymorphie)
  - Überschreiben vs. Verdecken
  - Überschreiben von Methoden
  - die Virtuelle-Methoden-Tabelle
  - Konkrete, abstrakte und rein spezifizierende Klassen (Schnittstellen-Klassen)

Objektorientierung 3
Polymorphie: Begriffserklärung

- Polymorphie = "Vielgestaltigkeit"
- Bei der Polymorphie geht es vor allem darum, das es mehrere Funktionen oder Methoden mit demselben Namen gibt, die aber eine unterschiedliche Implementation haben.
- Polymorphie in der Objektorientierung:
  - verschiedene Objekte können bei Aufruf derselben Operation ein unterschiedliches Verhalten an den Tag legen.

➢ **Ziel: Bessere Lesbarkeit!**



https://openbook.rheinwerk-verlag.de/oop/bilder/02_basisderobjektorientierung_004.gif, 10.5.2021

Objektorientierung 3

Polymorphie: Übersicht

Statische

Funktionen und Methoden mit gleichem Namen, werden schon zur **Kompilierzeit** eindeutig einer Implementierung zugeordnet.

Dynamische

Funktionen und Methoden mit gleichem Namen werden erst zur **Laufzeit** einer Implementierung zugeordnet.

Polymorphie

Objektorientierung 3
Überladung

Überladung

Überladen von Methoden

FH MÜNSTER
University of Applied Sciences

- Die Überladung von Funktionen und Methoden nennt man auch statischen Polymorphismus.
- Man spricht von Überladung, wenn mehrere Funktionen denselben Namen haben, aber verschiedene Variablen(-Typen) erwarten.
- Die Funktion wird statisch zur Kompilierzeit anhand der Variablentypen ausgewählt.

➢ Dient dazu Programme überschaubarer und lesbarer zu machen.

Bei welchem der beiden Beispiele handelt es sich um Überladung?

Beispiel 1:

```
void print(const char* text);
void print(double number);
```

Beispiel 2:

```
void print(double number);
int print(double number);
```

Überladen von Operatoren in C++

Schlüsselwörter in diesem Kapitel

alignas	const	dynamic_cast	long	short	typedef
alignof	consteval	else	mutable	signed	typeid
asm	constexpr	enum	namespace	sizeof	typename
auto	constinit	explicit	new	static	union
bool	const_cast	export	noexcept	static_assert	unsigned
break	continue	extern	nullptr	static_cast	using
case	co_await	false	**operator**	struct	virtual
catch	co_return	float	private	switch	void
char	co_yield	for	protected	template	volatile
char8_t	decltype	friend	public	this	wchar_t
char16_t	default	goto	register	thread_local	while
char32_t	delete	if	reinterpret_cast	throw	
class	do	inline	requires	true	
concept	double	int	return	try	

Überladen von Operatoren in C++

Vorraussetzungen

- In C++ können Operatoren (+, -, * usw.) überladen werden, unter folgenden Bedingungen:
  - Die Überladung ist Teil einer Klasse.
  - oder die Überladung erfolgt mit einem Objekt als Variable.
- Standardoperationen, wie z.B. arithmetische Additionen: 3+4, können somit nicht Überladen werden.
- Die Reihenfolge der Operatoren kann nicht geändert werden.

Beispiel:

```
// Initialisierung von 2d
Vektoren
Vektor2d v1{3.9, 4.2};
Vektor2d v2{1.1, 0.8};
Vektor2d v3;

// Addition zweier Objekte der
// Klasse Vektor2d
v3 = v1 + v2;
```

Überladen von Operatoren in C++

Beispiel


```
class Vektor2d
{
  public:
    Vektor2d(double _x, double _y);
    Vektor2d() = default;

    Vektor2d operator+(const Vektor2d &other) const;
    void operator+=(const Vektor2d &other);

    // operator*()
    // operator*=() ...

  private:
    double x = 0;
    double y = 0;
};
```

Steht `const` am Ende einer Funktionsdeklaration einer Klasse, so ist es dieser Funktion nicht erlaubt Daten des Objektes zu verändern.

Deklaration

Überladen von Operatoren in C++

Beispiel


```
Vektor2d::Vektor2d(double _x, double _y):
x(_x), y(_y){};

Vektor2d Vektor2d::operator+(const Vektor2d &other) const
{
  Vektor2d result;
  result.x = this->x + other.x;
  result.y = this->y + other.y;
  return result;
}

void operator+=(const Vektor2d &other)
{
    this->x += other.x;
    this->y += other.y;
}
```

Steht `const` am Ende einer Funktionsdeklaration einer Klasse, so ist es dieser Funktion nicht erlaubt Daten des Objektes zu verändern.

Implementierung

Überladen von Operatoren

Arten von Operatoren

- operator ***op***
- operator ***type***
- operator **new**
- operator **new []**
- operator **delete**
- operator **delete []**
- operator **""** ***suffix-identifier*** (seit C++11)

Überladen von Operatoren

Arten von Operatoren

- operator ***op*** :
  + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= && || ++ -- , ->* -> ( ) [ ]
  - ➢ https://en.cppreference.com/w/cpp/language/operator_arithmetic
- operator ***type***
- operator **new**
- operator **new** []
- operator **delete**
- operator **delete** []
- operator **""** ***suffix-identifier*** (seit C++11)

Objektorientierung 3
Polymorphie

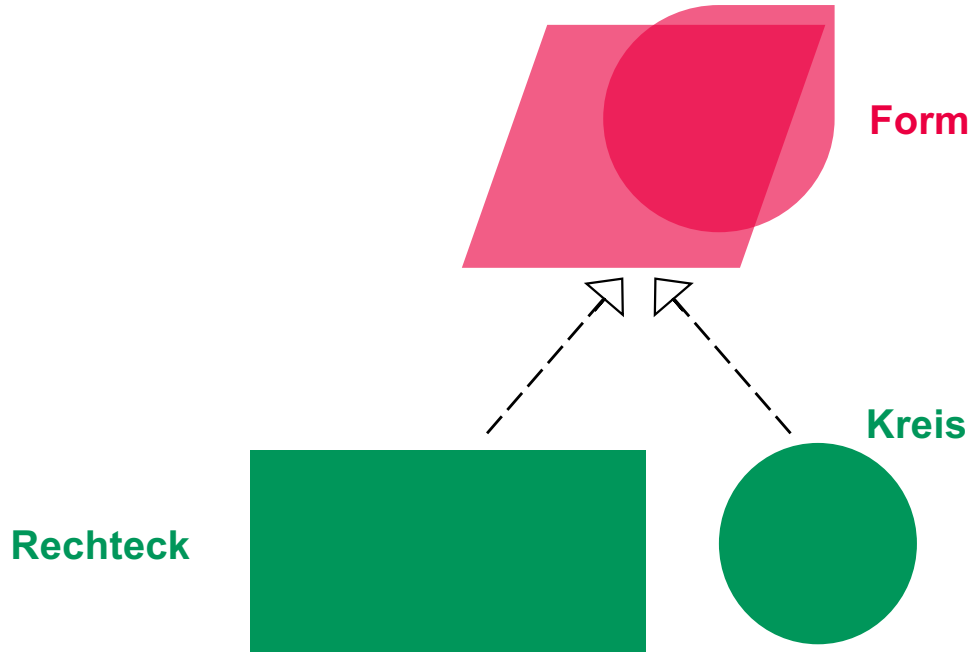
Polymorphie

Schlüsselwörter in diesem Kapitel

alignas	const	dynamic_cast	long	short	typedef
alignof	consteval	else	mutable	signed	**typeid**
asm	constexpr	enum	namespace	sizeof	typename
auto	constinit	explicit	new	static	union
bool	const_cast	export	noexcept	static_assert	unsigned
break	continue	extern	nullptr	static_cast	using
case	co_await	false	operator	struct	**virtual**
catch	co_return	float	private	switch	void
char	co_yield	for	protected	template	volatile
char8_t	decltype	friend	public	this	wchar_t
char16_t	default	goto	register	thread_local	while
char32_t	delete	if	reinterpret_cast	throw	
class	do	inline	requires	true	
concept	double	int	return	try	

Polymorphie
Beispiel

Polymorphie
Beispiel



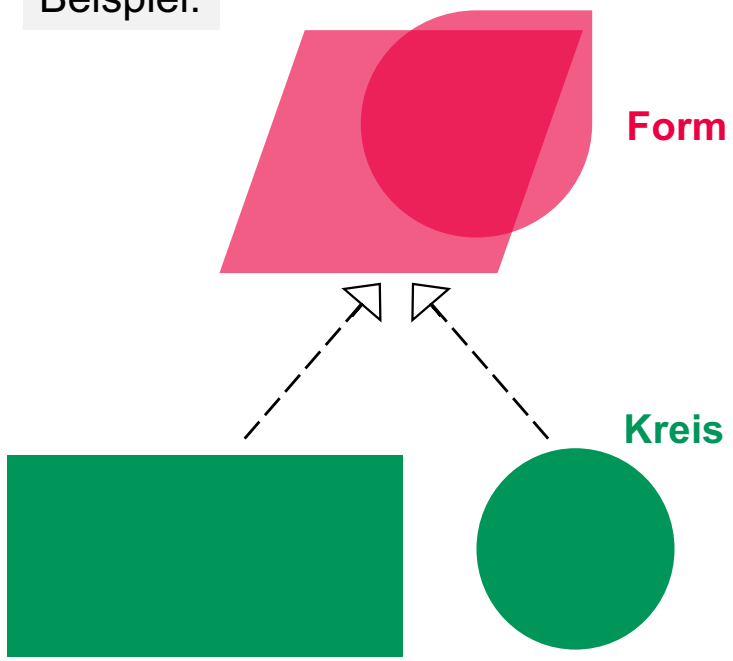
- Das Zeichnen von Rechtecken und Kreisen unterscheidet sich deutlich.
- Beide "Formen" haben aber gemeinsam, dass sie gezeichnet werden können

Polymorphie
Dynamische Polymorphie

- Voraussetzung: Verschiedene Objekte verwandter Klassen können ein und derselben Variablen zugeordnet werden.
  - Verwandte Klassen besitzen in der Vererbungshierarchie mind. eine gemeinsame Oberklasse bzw. sind die Oberklasse.
- Welche konkrete Methode das Programm zur Laufzeit (dynamisch) aufruft, hängt von der tatsächlichen Klasse des Objektes ab, welches der Variablen zugeordnet ist.

➢ Der Aufruf der Methode erfolgt somit polymorph.

Beispiel:



Polymorphie
Vorraussetzungen

FH MÜNSTER
University of Applied Sciences

Späte Bindung

- Objektorientierte Programmiersprachen sind in der Lage erst zur Laufzeit zu entscheiden welche Methode benutzt werden soll.
- Diese Fähigkeit wird Späte Bindung genannt.

➢ Der Mechanismus der Späten Bindung ist die Voraussetzung für dynamische Polymorphie.

Vererbung

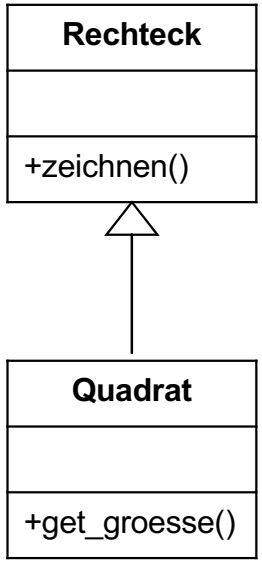
- Dynamische Polymorphie hängt stark mit
  - dem Konzept der Vererbung der Spezifikation (Schnittstelle)
  - und dem Prinzip der Ersetzbarkeit zusammen.

➢ Dynamische Polymorphie bietet die technische Voraussetzung beides effektiv zu nutzen.

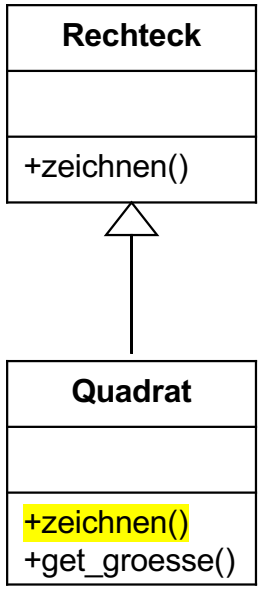
Polymorphie
Übernahme vs. Redefinition

FH MÜNSTER
University of Applied Sciences

Übernahme der Methode aus Oberklasse



Redefinition:



Polymorphie
durch Klassenumwandlung

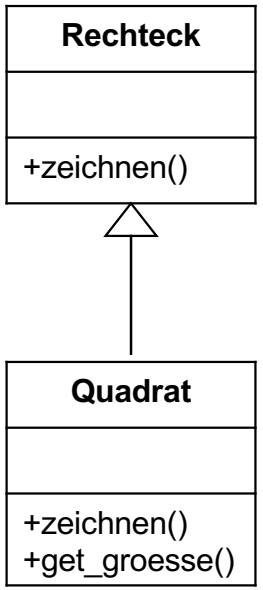
Handelt es sich hierbei um Polymorphie?

```
Quadrat quadrat {2};
Rechteck rechteck {3,4};
Rechteck* objekte[2] {&quadrat, &rechteck};

quadrat.zeichnen();
rechteck.zeichnen();

for(int i=0; i<2; i++)
{
  objekte[i]->zeichnen();
}
```

Beispiel:



Polymorphie
durch Klassenumwandlung

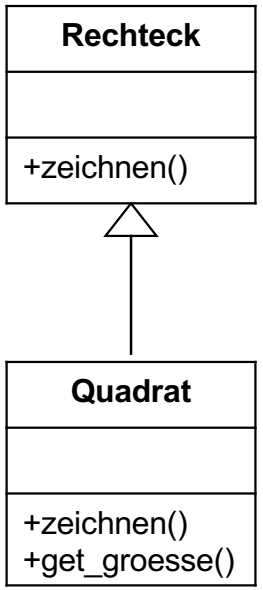
Ja, wenn die Methode `zeichnen()` zur Laufzeit Quadrat zugeordnet wird.

```
Quadrat quadrat {2};
Rechteck rechteck {3,4};
Rechteck* objekte[2] {&quadrat, &rechteck};

quadrat.zeichnen();
rechteck.zeichnen();

for(int i=0; i<2; i++)
{
  objekte[i]->zeichnen();
}
```

Beispiel:



Polymorphie

durch Klassenumwandlung

```
class Rechteck {
   // öffentlicher Teil
   public:
       // Konstruktor initialisiert breite und hoehe
       // b=Breite, h=Höhe
       Rechteck(unsigned int b, unsigned int h);
       // Methode zum Zeichnen des Quaders
       void zeichnen();
   // privater Teil
   private:
      unsigned int breite;
      unsigned int hoehe;
};
```

```
Rechteck::zeichnen(){
  cout << "--" << endl;
  cout << "Ein " << this->breite << "x";
  cout << this->hoehe << " Rechteck." << endl;
}
```

Redefinition der Methode `zeichnen()`

```
class Quadrat : public Rechteck {
   // öffentlicher Teil
   public:
       // Konstruktor initialisiert groesse
       // g=Größe
        Quadrat(unsigned int g);
        void zeichnen();
   ..
};
```

```
Quadrat::zeichnen(){
  /* Ein expliziter Aufruf einer Funktion der
   Oberklasse erfolgt durch Oberklasse:: */
  Rechteck::zeichnen();
  cout << "Ist ein Quadrat." << endl;
}
```

Polymorphie

durch Klassenumwandlung

```
Quadrat quadrat {2};
Rechteck rechteck {3,4};
Rechteck* objekte[2] {&quadrat, &rechteck};

quadrat.zeichnen();
rechteck.zeichnen();

for(int i=0; i<2; i++)
{
  objekte[i]->zeichnen();
}
```

Ausgabe:

```
--
Ein 2x2 Rechteck.
Ist ein Quadrat.
--
Ein 3x4 Rechteck.
--
Ein 2x2 Rechteck.
--
Ein 3x4 Rechteck.
```

```
class Rechteck {
  ...
       void zeichnen();
  ...
};
```

```
class Quadrat : public Rechteck {
  ...
       void zeichnen();
  ...
};
```

Polymorphie
durch Klassenumwandlung

Noch keine dynamische Polymorphie!

```
Quadrat quadrat {2};
Rechteck rechteck {3,4};
Rechteck* objekte[2] {&quadrat, &rechteck};

quadrat.zeichnen();
rechteck.zeichnen();

for(int i=0; i<2; i++)
{
  objekte[i]->zeichnen();
}
```

Ausgabe:

```
--
Ein 2x2 Rechteck.
Ist ein Quadrat.
--
Ein 3x4 Rechteck.
--
Ein 2x2 Rechteck.
--
Ein 3x4 Rechteck.
```

```
class Rechteck {
  ...
       void zeichnen();
  ...
};
```

```
class Quadrat : public Rechteck {
  ...
       void zeichnen();
  ...
};
```

Polymorphie
durch Klassenumwandlung

Dynamische Polymorphie!

```
Quadrat quadrat {2};
Rechteck rechteck {3,4};
Rechteck* objekte[2] {&quadrat, &rechteck};

quadrat.zeichnen();
rechteck.zeichnen();

for(int i=0; i<2; i++)
{
  objekte[i]->zeichnen();
}
```

Ausgabe:

```
--
Ein 2x2 Rechteck.
Ist ein Quadrat.
--
Ein 3x4 Rechteck.
--
Ein 2x2 Rechteck.
Ist ein Quadrat.
--
Ein 3x4 Rechteck.
```

```
class Rechteck {
  ...
      virtual void zeichnen();
  ...
};
```

```
class Quadrat : public Rechteck {
  ...
      void zeichnen();
  ...
};
```

Polymorphie
Virtuelle Methode

- Das Schlüsselwort **`virtual`** vor Funktionen bewirkt, dass immer die Methode des Objektes aufgerufen wird, z.B. ist das Objekt ein Kreis, wird auch ein Kreis gezeichnet.

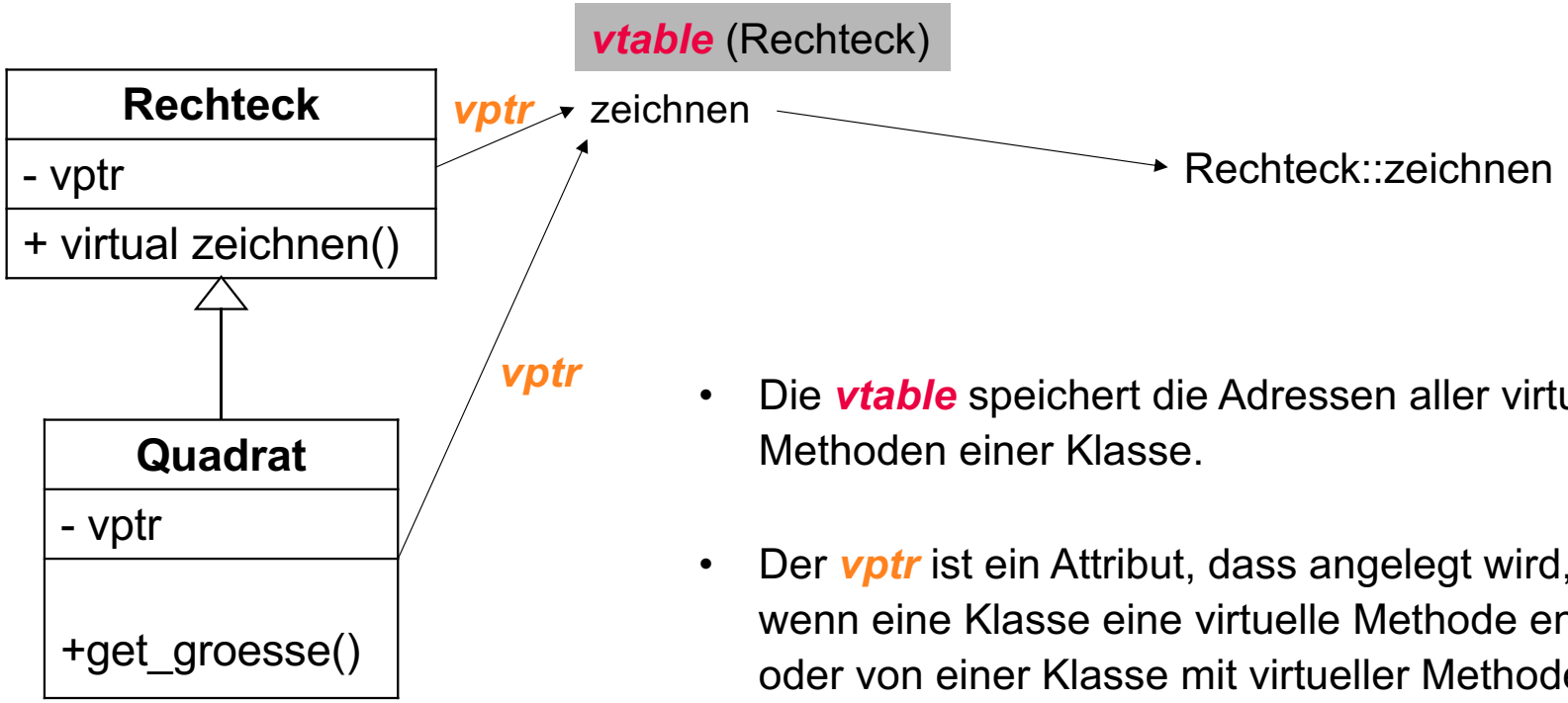
Virtuelle Methode

```
virtual void foo ();
```

- Hinweis: **`virtual`** wird lediglich vor die Deklaration einer Funktion geschrieben nicht aber vor deren Implementierung. Es sei denn Deklaration und Implementierung erfolgt in einem Schritt.

Polymorphie

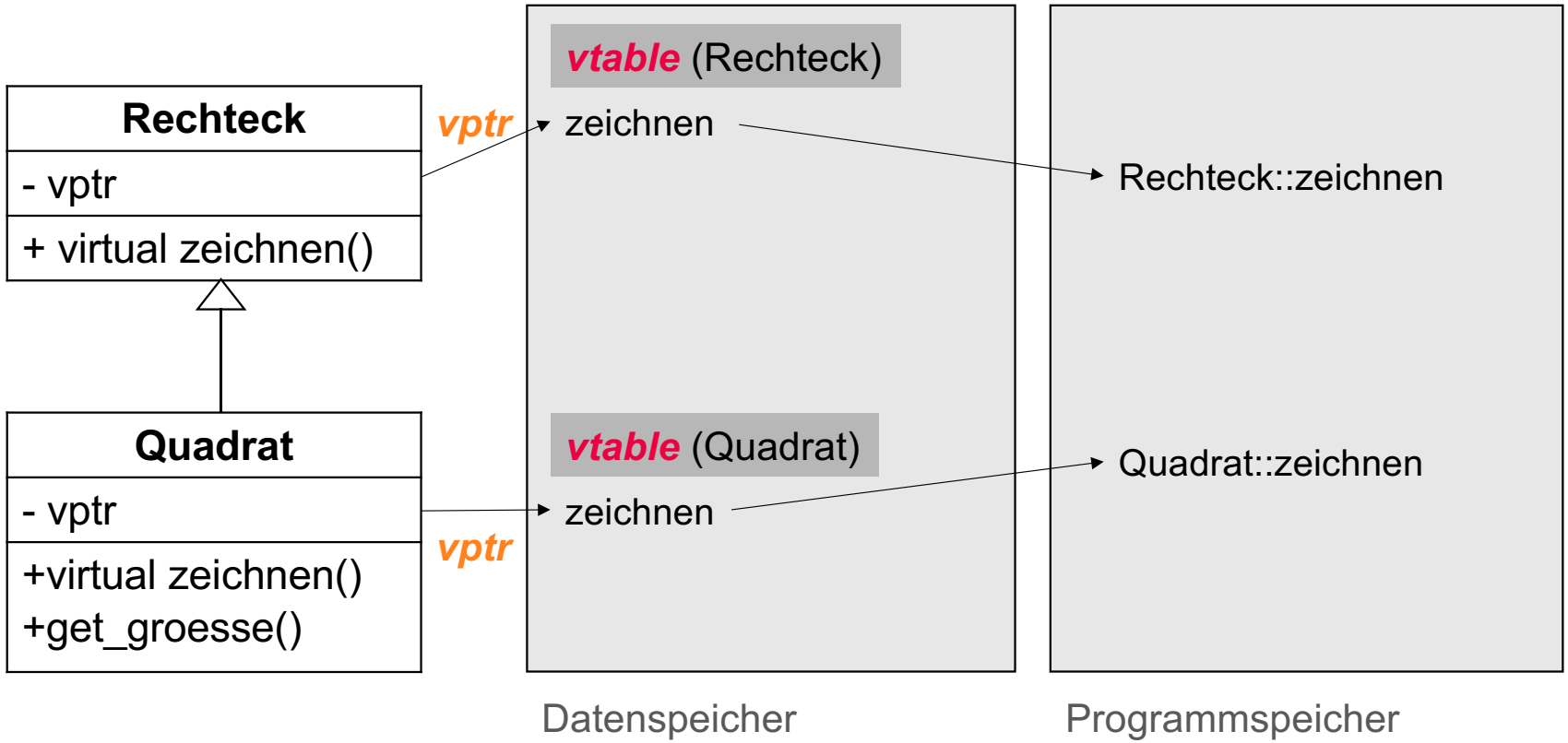
Die Virtuelle-Methoden-Tabelle (vtable)



- Die ***vtable*** speichert die Adressen aller virtuellen Methoden einer Klasse.
- Der ***vptr*** ist ein Attribut, dass angelegt wird, wenn eine Klasse eine virtuelle Methode enthält oder von einer Klasse mit virtueller Methode erbt. Der ***vptr*** enthält die Adresse der ***vtable***.

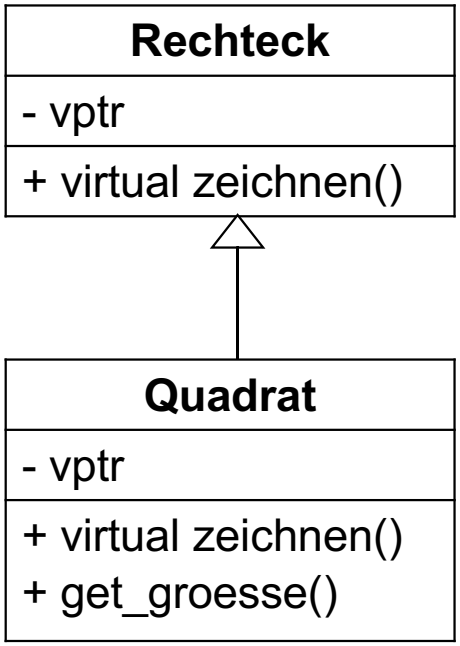
Polymorphie

Die Virtuelle-Methoden-Tabelle (vtable)



Polymorphie

Exkurs: Typinformationen von polymorphen Objekten



Wie lautet die Ausgabe?

```
Quadrat quadrat;
Rechteck& rechteck = quadrat;

cout << typeid(rechteck).name() << endl;
```

Polymorphie
Virtueller Destruktor

- Enthält eine Klasse eine virtuelle Funktion, wird üblicherweise der Destruktor ebenfalls als virtuell deklariert.
- Da die virtuellen Funktionen der Klasse objektbezogen aufgerufen werden, sollte dies später beim Aufräumen der Klasse im Destruktor ebenfalls geschehen.
- Die Deklaration des virtuellen Destruktors erfolgt mit:

```
virtual ~Klassenname ();
```

- Hinweis: Das Schlüsselwort `virtual` kann nicht auf Konstruktoren angewendet werden.

Polymorphie
Methoden Spezifizierer `override`

- Mit `override` Schreibfehler reduzieren

```
class Form {
  // ....
  virtual double flaeche() const { return 0.0;}
};
```

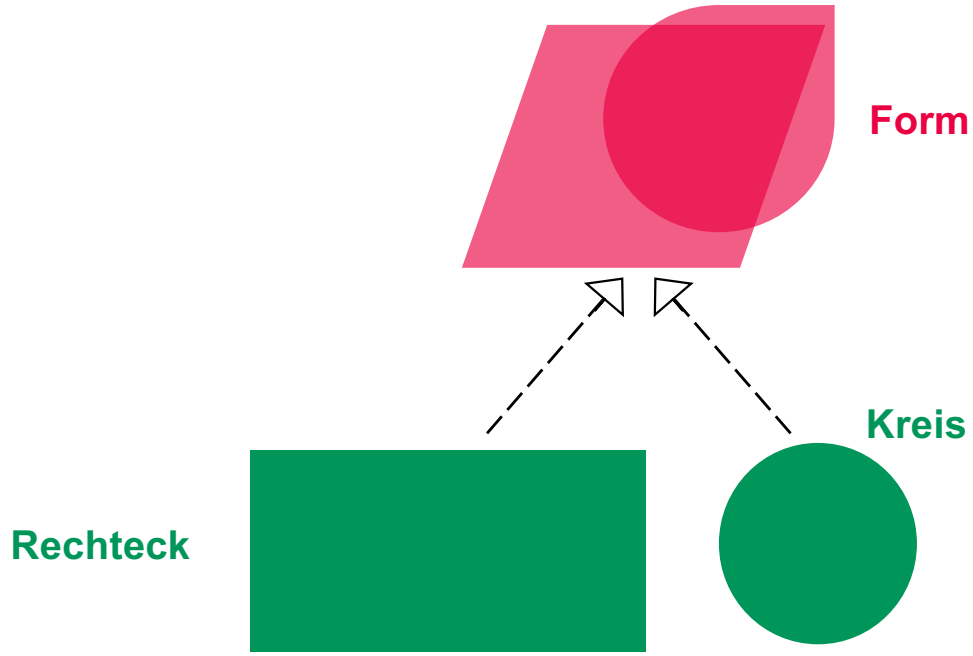
```
class Rechteck : public Form {
  // ....
  virtual double flaeche() const override
  {
    return static_cast<double>(hoehe) * breite;
  }
};
```

Tipp: Benutzen Sie bei allen virtuellen Methoden, die eine Methode der Oberklasse überschreiben, den Spezifizierer `override`.

Identifier in C++ sind keine Schlüsselwörter und können daher auch Variablen oder Funktionsnamen sein.

Vererbung der Spezifikation

Konkrete, abstrakte und rein abstrakte Klassen



Abstrakte Methoden

Rein virtuelle Methoden

- Abstrakte Methoden sind virtuelle Methoden, die nicht implementiert werden:

Abstrakte Methode

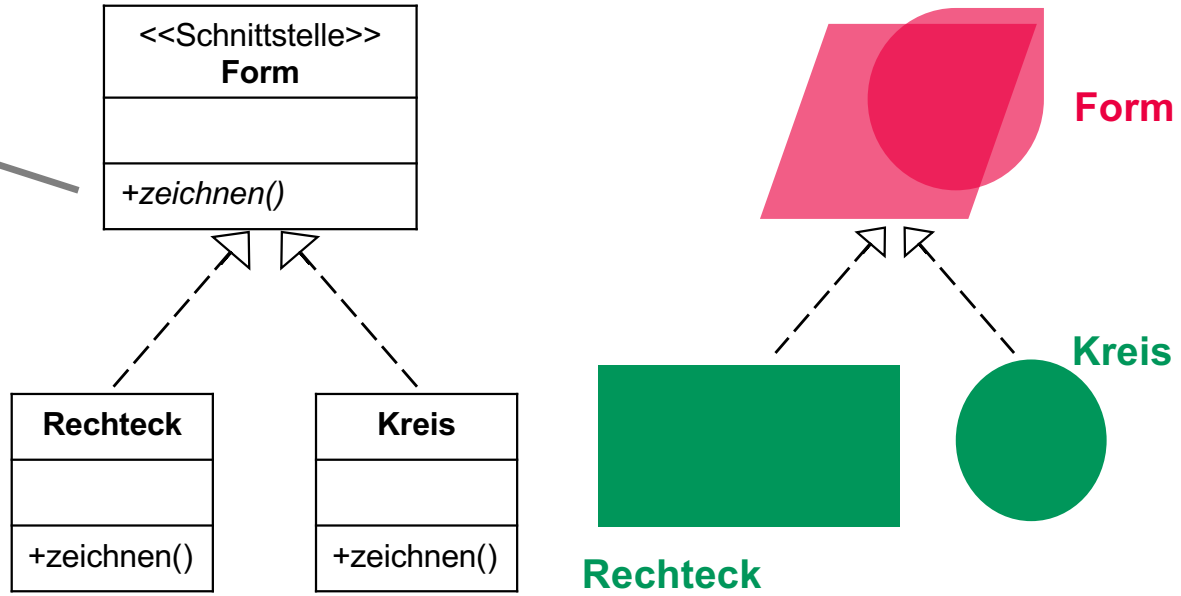
```
virtual void foo () = 0;
```

- Je nach Zusammensetzung von abstrakten und konkret implementierten Funktionen ergeben sich drei Formen von Klassen:
  - **Konkrete Klasse:** Alle Methoden sind implementiert.
  - **Abstrakte Klasse:** Es gibt mindestens eine abstrakte Methode.
  - **Rein spezifizierende Klasse (Schnittstelle):** Alle Methoden sind abstrakte Methoden.

Schnittstellen Klassen
UML

FH MÜNSTER
University of Applied Sciences

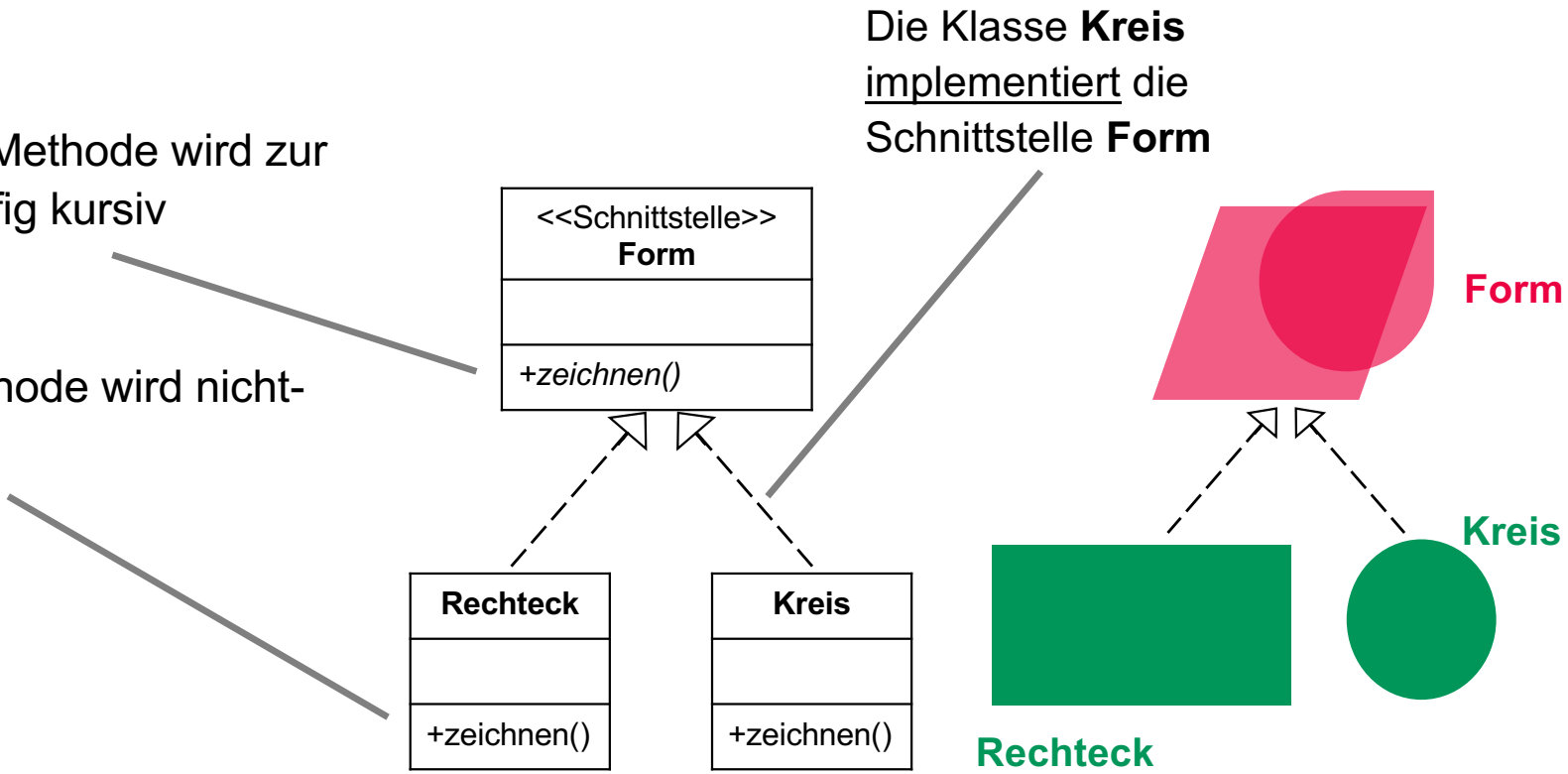
- Spezifikation einer Methode wird zur Verdeutlichung häufig kursiv geschrieben.
- Implementierte Methode wird nicht-kursiv geschrieben.



Schnittstellen Klassen

UML

- Spezifikation einer Methode wird zur Verdeutlichung häufig kursiv geschrieben.
- Implementierte Methode wird nicht-kursiv geschrieben.



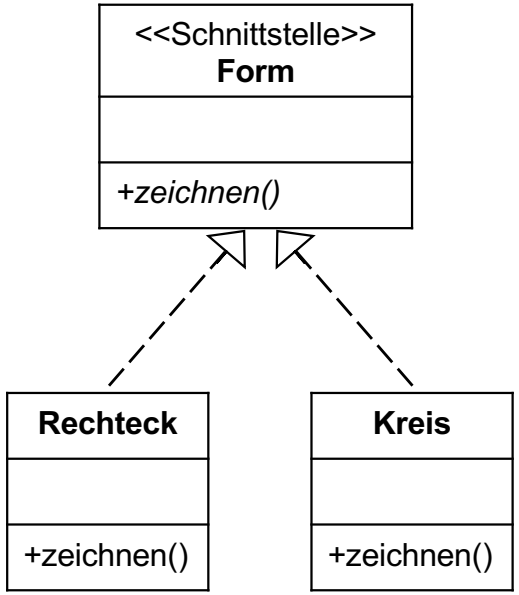
Schnittstellen Klassen

sind rein abstrakte Klasse

- Eine Schnittstelle enthält nur die Spezifikation eines Objektes, nicht die Implementierung.
- Deklaration einer Schnittstelle in C++:

```
class Form {
   // öffentlicher Teil
   public:
       // Abstrakte Methode
       virtual void zeichnen() = 0;
};
```

- Schnittstellen besitzen in C++ nur abstrakte Methoden (**`public`** oder **`protected`**).



Schnittstellen Klassen

Beispiel: Implementierung

- Implementierung (Ableitung von) einer Schnittstelle

```
// Deklaration
class Kreis: public Form {
   // öffentlicher Teil
   public:
        // Konstruktor
        Kreis(unsigned int r);
        // Methode zum Zeichnen des Kreises
        void zeichnen();
   // privater Teil
   private:
      unsigned int radius;
};
```

```
// Implementierung
Kreis::Kreis(unsigned int r) : radius{r} {};

void Kreis::zeichnen()
{
  cout << "--" << endl;
  cout << "Ein Kreis mit Radius ";
  cout << this->radius << endl;
}
```

Schnittstellen Klassen

Beispiel: Verwendung

```
Kreis kreis{1};
Rechteck rechteck1{3, 4};
Rechteck rechteck2{2, 3};


// Eine Liste verschiedener Formen
Form *objekte[3]{&kreis, &rechteck1, &rechteck2};


// Die Liste wird abgearbeitet
for (int i = 0; i < 3; i++)
{
  objekte[i]->zeichnen();
  cout << typeid(*objekte[i]).name() << endl;
}
```

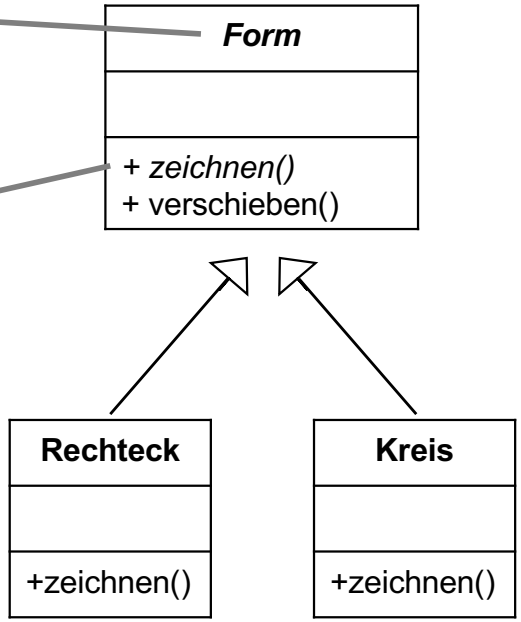
Mit Hilfe des Schlüsselwortes **`typeid`** kann dynamisch überprüft werden, welcher Typ sich hinter dem Objekt verbirgt.

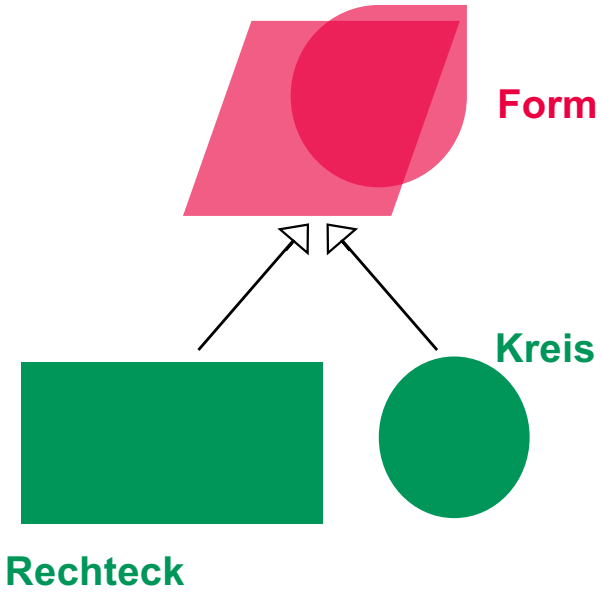
`typeid` zur Unterscheidung von Objekten funktioniert nur richtig in Verbindung mit polymorphen Objekten, d.h. Oberklasse muss mind. eine virtuelle Funktion besitzen! Dabei muss die Oberklasse nicht zwingend eine Schnittstelle sein!

Abstrakte Klassen

UML

- Eine abstrakte Klasse (kursiv) ist eine Mischform aus konkreter und rein spezifizierender Klasse (Schnittstelle)
- Sie enthält sowohl abstrakte (kursiv) als auch implementierte Methoden.



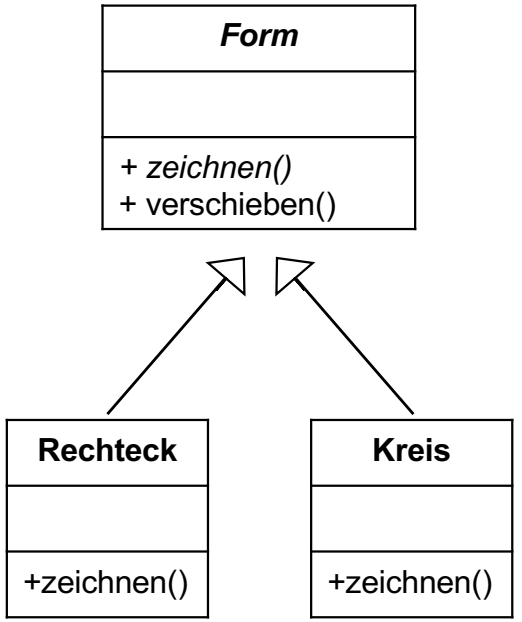


Abstrakte Klassen

Deklaration

```
class Form {
   // öffentlicher Teil
   public:
        // Abstrakte Methode
        virtual void zeichnen() = 0;
        // Konkrete Methode
        // (diese Methode kann, muss aber nicht virtuell sein)
        virtual void verschieben()
        {
           /* Quelltext zum Verschieben von Formen
             ... */
        }
};
```

In C++ gibt es im Gegensatz zu anderen Programmiersprachen keine expliziten Schlüsselworte für abstrakte oder Schnittstellen Klassen. Der Typ der Klasse folgt implizit aus der Implementierung!



FH MÜNSTER
University of Applied Sciences

Programmieren in C++

Prof. Dr. Kathrin Ungru
Fachbereich Elektrotechnik und Informatik

kathrin.ungru@fh-muenster.de