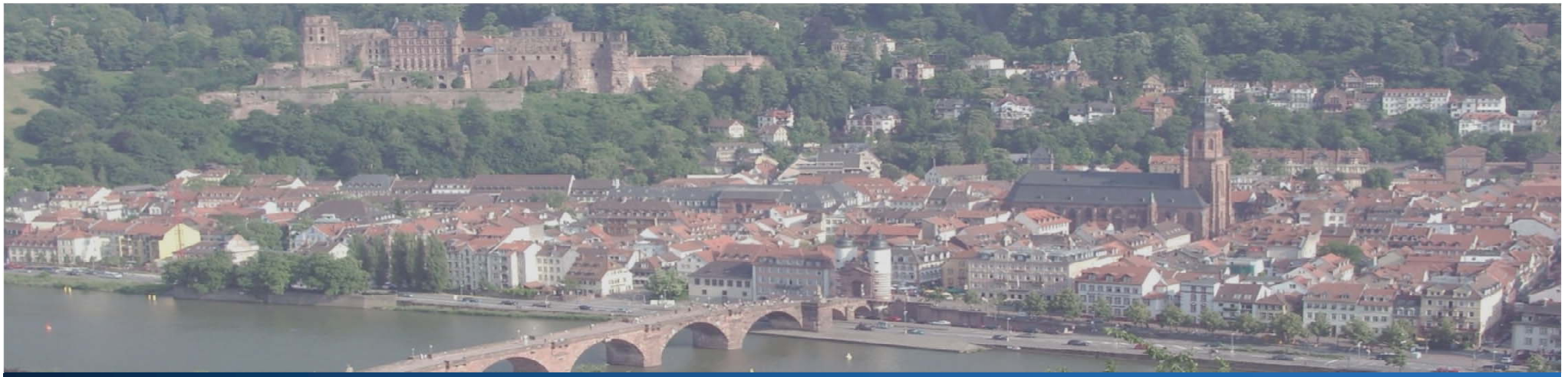
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



Vererbung

- Motivation: Polynome
- Implementation
- Öffentliche Vererbung
- Beispiel zu public/private und öffentlicher Vererbung
- Ist-ein-Beziehung
- Kosntruktoren, Destruktor und Zuweisungsoperatoren
- Auswertung
- Weitere Methoden
- Gleichheit
- Benutzung von *Polynomial*
- Diskussion
- Private Vererbung
- Methodenauswahl und virtuelle Funktionen

Motivation: Polynome

Definition: Ein Polynom $p : \mathbb{R} \to \mathbb{R}$ ist eine Funktion der Form

$$p(x) = \sum_{i=0}^{n} p_i x^i,$$

Wir betrachten hier nur den Fall reellwertiger Koeffizienten $p_i \in \mathbb{R}$ und verlangen $p_n \neq 0$. n heißt dann Grad des Polynoms.

Operationen:

- Konstruktion.
- Manipulation der Koeffizienten.
- Auswerten des Polynoms an einer Stelle x.

- Addition zweier Polynome

$$p(x) = \sum_{i=0}^{n} p_i x^i, \quad q(x) = \sum_{j=0}^{m} q_j x^j$$

$$r(x) = p(x) + q(x) = \sum_{i=0}^{\max(n,m)} \underbrace{(p_i^* + q_i^*)}_{r_i} x^i$$

$$p_i^* = \begin{cases} p_i & i \leq n \\ 0 & \text{sonst} \end{cases}, \quad q_i^* = \begin{cases} q_i & i \leq m \\ 0 & \text{sonst} \end{cases}.$$

- Multiplikation zweier Polynome

$$
\begin{aligned}
r(x) = p(x) * q(x) &= \left(\sum_{i=0}^{n} p_i x^i\right)\left(\sum_{j=0}^{m} q_j x^j\right) \\
&= \sum_{i=0}^{n}\sum_{j=0}^{m} p_i q_j x^{i+j} \\
&= \sum_{k=0}^{m+n} \underbrace{\left(\sum_{\{(i,j)|i+j=k\}} p_i q_j\right)}_{r_k} x^k
\end{aligned}
$$

Implementation

In großen Programmen möchte man Codeduplizierung vermeiden und möglichst viel Code wiederverwenden.

Für den Koeffizientenvektor $p_0, \ldots, p_n$ wäre offensichtlich ein Feld der adäquate Datentyp. Wir wollen unser Feld `SimpleFloatArray` benutzen. Eine Möglichkeit:

Programm:

```
class Polynomial
{
private:
  SimpleFloatArray coefficients;
public:
    ...
};
```

Alternativ kann man Polynome als Exemplare von `SimpleFloatArray` mit zusätzlichen Eigenschaften ansehen, was im folgenden ausgeführt wird.

Öffentliche Vererbung

Programm: Definition der Klasse `Polynomial` mittels Vererbung:
(Polynomial.hh)

```
class Polynomial : public SimpleFloatArray
{
public:
  // konstruiere Polynom vom Grad n
  Polynomial( int n );

  // Default-Destruktor ist ok
  // Default-Copy-Konstruktor ist ok
  // Default-Zuweisung ist ok

  // Grad des Polynoms
  int degree();
```

```
  // Auswertung
  float eval( float x );

  // Addition von Polynomen
  Polynomial operator+( Polynomial q );

  // Multiplikation von Polynomen
  Polynomial operator*( Polynomial q );

  // Gleichheit
  bool operator==( Polynomial q );

  // drucke Polynom
  void print();
};
```

Syntax: Die Syntax der öffentlichen Vererbung lautet:

<ÖAbleitung> ::= class <Klasse2> : public <Klasse1> {
<Rumpf>
} ;

Bemerkung:

- Klasse 1 heißt Basisklasse, Klasse 2 heißt abgeleitete Klasse.
- Klasse 2 enthält ein Objekt von Klasse 1 als Unterobjekt.
- Alle öffentlichen Mitglieder der Basisklasse mit Ausnahme von Konstruktoren, Destruktor und Zuweisungsoperatoren sind auch öffentliche Mitglieder der abgeleiteten Klasse. Sie operieren auf dem Unterobjekt.
- Im Rumpf kann Klasse 2 weitere Mitglieder vereinbaren.

- Daher spricht man auch von einer Erweiterung einer Klasse durch öffentliche Ableitung.

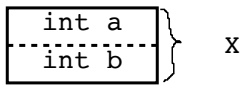
- Alle privaten Mitglieder der Basisklasse sind *keine* Mitglieder der Klasse 2. Damit haben auch Methoden der abgeleiteten Klasse *keinen* Zugriff auf private Mitglieder der Basisklasse.

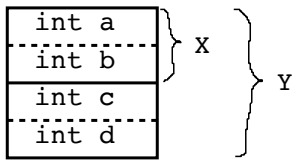
- Eine Klasse kann mehrere Basisklassen haben (Mehrfachvererbung), diesen Fall behandeln wir hier aber nicht.

Beispiel zu public/private und öffentlicher Vererbung

```
class X
{
public:
  int a;
  void A();
private:
  int b;
  void B();
};
```

```
class Y : public X
{
public:
  int c;
  void C();
private:
  int d;
  void D();
};
```





```
X x;

x.a = 5;      // OK
x.b = 10;     // Fehler

void X::A()
{
  B();        // OK
  b = 3;      // OK
}
```

```
Y y;
y.a = 1;  // OK
y.c = 2;  // OK
y.b = 4;  // Fehler
y.d = 8;  // Fehler

void Y::C() {
  d = 8;  // OK
  b = 4;  // Fehler
  A();    // OK
  B();    // Fehler
}
```

Ist-ein-Beziehung

Ein Objekt einer abgeleiteten Klasse enthält ein Objekt der Basisklasse als Unterobjekt.

Daher darf ein Objekt der abgeleiteten Klasse für ein Objekt der Basisklasse eingesetzt werden. Allerdings sind dann nur Methoden der Basisklasse für das Objekt aufrufbar.

Beispiel:

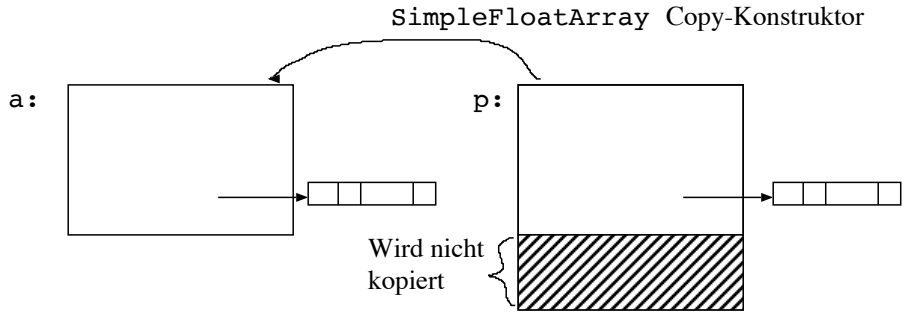
```
void g( SimpleFloatArray a ) { a[3] = 1.0; }

Polynomial p( 10 );
SimpleFloatArray b( 100, 0.0 );
g( p );   // (1) OK
p = b;    // (2) Fehler
b = p;    // (3) OK
```

Bemerkung:

- Im Fall (1) wird bei Aufruf von `g(p)` der Copy-Konstruktor des formalen Parameters `a`, also `SimpleFloatArray`, benutzt, um das `SimpleFloatArray`-Unterobjekt von `p` auf den formalen Parameter `a` vom Typ `SimpleFloatArray` zu kopieren.

- Falls `Polynomial` weitere Datenmitglieder hätte, so würde die Situation so aussehen:



In diesem Fall spricht man von *slicing*.

- Im Fall (2) soll einem Objekt der abgeleiteten Klasse ein Objekt der Basisklasse zugewiesen werden. Dies ist nicht erlaubt, da nicht klar ist, welchen Wert etwaige zusätzliche Datenmitglieder der abgeleiteten Klasse bekommen sollen.

- Fall (3) ist OK, der Zuweisungsoperator der Basisklasse wird aufgerufen und das Unterobjekt aus der abgeleiteten Klasse dem links stehenden Objekt der Basisklasse zugewiesen.

Konstruktoren, Destruktor und Zuweisungsoperatoren

Programm: **(PolynomialKons.cc)**

```
Polynomial::Polynomial( int n ) :
  SimpleFloatArray( n+1, 0.0 ) {}
```

Bemerkung:

- Die syntaktische Form entspricht der Initialisierung von Unterobjekten wie oben beschrieben.

- Die Implementierung des Copy-Konstruktors kann man sich sparen, da der Default-Copy-Konstruktor das Gewünschte leistet, dasselbe gilt für Zuweisungsoperator und Destruktor.

Auswertung

Programm: Auswertung mit Horner-Schema **(PolynomialEval.cc)**

```
// Auswertung
float Polynomial::eval( float x )
{
  float sum = 0.0;

  // Hornerschema
  for ( int i=maxIndex(); i>=0; i=i-1 )
    sum = sum * x + operator[](i);
  return sum;
}
```

$$p(x) = (\ldots(b_n x + b_{n-1})x + \cdots)x + b_0.$$

Bemerkung: Statt **operator**[](i) könnte man (∗**this**)[i] schreiben.

Weitere Methoden

Programm: **(PolynomialImp.cc)**

```
// Grad auswerten
int Polynomial::degree()
{
  return maxIndex();
}

// Addition von Polynomen
Polynomial Polynomial::operator+( Polynomial q )
{
  int nr = degree(); // mein Grad

  if ( q.degree() > nr) nr = q.degree();

  Polynomial r( nr ); // Ergebnispolynom
```

```
  for ( int i=0; i<=nr; i=i+1 )
  {
    if ( i <= degree() )
      r[i] = r[i] + (*this)[i]; // add me to r
    if ( i <= q.degree() )
      r[i] = r[i] + q[i];       // add q to r
  }

  return r;
}

// Multiplikation von Polynomen
Polynomial Polynomial::operator*( Polynomial q )
{
  Polynomial r( degree() + q.degree() ); // Ergebnispolynom

  for ( int i=0; i<=degree(); i=i+1 )
    for ( int j=0; j<=q.degree(); j=j+1 )
      r[i+j] = r[i+j] + (*this)[i] * q[j];
```

```
  return r;
}

// Drucken
void Polynomial::print()
{
  if ( degree() < 0 )
    std::cout << 0;
  else
    std::cout << (*this)[0];

  for ( int i=1; i<=maxIndex(); i=i+1 )
    std::cout << "+" << (*this)[i] << "*x^" << i;

  std::cout << std::endl;
}
```

Gleichheit

Gleichheit ist kein einfaches Konzept, wie man ja schon an Zahlen sieht: ist `0 == 0.0`? Oder `(int) 1000000000 == (short) 1000000000`? Gleichheit für selbstdefinierte Datentypen ist daher Sache des Programmierers:

Programm: **(PolynomialEqual.cc)**

```
bool Polynomial::operator==( Polynomial q )
{
  if ( q.degree() > degree() )
  {
    for ( int i=0; i<=degree(); i=i+1 )
      if ( (*this)[i] != q[i] ) return false;
    for ( int i=degree()+1; i<=q.degree(); i=i+1 )
      if ( q[i] != 0.0 ) return false;
  }
  else
```

```
    {
      for ( int i=0; i<=q.degree(); i=i+1 )
        if ( (*this)[i] != q[i] ) return false;
      for ( int i=q.degree()+1; i<=degree(); i=i+1 )
        if ( (*this)[i] != 0.0 ) return false;
    }

    return true;
}
```

Bemerkung: Im Gegensatz dazu ist Gleichheit von Zeigern immer definiert. Zwei Zeiger sind gleich, wenn sie auf *dasselbe* Objekt zeigen:

```
Polynomial p( 10 ), q( 10 );

Polynomial* z1 = &p;
Polynomial* z2 = &p;
Polynomial* z3 = &q;

if ( z1 == z2 ) ... // ist wahr
if ( z1 == z3 ) ... // ist falsch
```

Benutzung von `Polynomial`

Folgendes Beispiel definiert das Polynom

$$p = 1 + x$$

und druckt p, p^2 und p^3.

Programm: **(UsePolynomial.cc)**

```
#include <iostream>

// alles zum SimpleFloatArray
#include "SimpleFloatArray.hh"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
```

```
#include "SimpleFloatArrayAssign.cc"

// Das Polynom
#include "Polynomial.hh"
#include "PolynomialImp.cc"
#include "PolynomialKons.cc"
#include "PolynomialEqual.cc"
#include "PolynomialEval.cc"

int main()
{
  Polynomial p( 2 ), q( 10 );

  p[0] = 1.0;
  p[1] = 1.0;
  p.print();
```

```
    q = p * p;
    q.print();

    q = p * p * p;
    q.print();
  }
```

mit der Ausgabe:

```
1+1*x^1
1+2*x^1+1*x^2
1+3*x^1+3*x^2+1*x^3
```

Diskussion

- Diese Implementation hat die wesentliche Schwachstelle, dass der Grad bei führenden Nullen mathematisch nicht korrekt ist. Angenehmer wäre, wenn Konstanten den Grad 0 hätten, lineare Polynome den Grad 1 und der Koeffizientenvektor allgemein die Länge Grad+1 hätte.

- Die Abhilfe könnte darin bestehen, dafür zu sorgen, dass der Konstruktor nur Polynome mit korrektem Grad erzeugt. Allerdings können Polynome ja beliebig modifiziert werden, daher wäre eine andere Möglichkeit in der Methode `degree` den Grad jeweils aus den Koeffizienten zu bestimmen.

Wiederholung: Vererbung

Öffentliche Ableitung einer Klasse:

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

Öffentliche Mitglieder von A sind auch öffentliche Mitglieder von B. Private Mitglieder von A sind in B nicht sichtbar.

Objekt der Klasse B enthält Objekt der Klasse A als Unterobjekt.

Ein Objekt der Klasse B kann für ein Objekt der Klasse A eingesetzt werden, etwa bei einem Funktionsaufruf (Ist-ein Beziehung).

Konstruktor, Destruktor und Zuweisungsoperator werden nicht vererbt, es werden aber ggf. Default-Varianten erzeugt.

Private Vererbung

Wenn man nur die Implementierung von `SimpleFloatArray` nutzen will, ohne die Methoden öffentlich zu machen, so kann man dies durch private Vererbung erreichen:

Programm:

```
class Polynomial : private SimpleFloatArray
{
public:
   ...
};
```

Bemerkung: Hier müsste man dann die Schnittstelle zum Zugriff auf die Koeffizienten des Polynoms selbst Programmieren, oder zumindest die Initialisierung mittels

```
Polynomial::Polynomial( SimpleFloatArray& coeffs ) { ... }
```

Eigenschaften der privaten Vererbung

Bemerkung: Private Vererbung bedeutet:

- Ein Objekt der abgeleiteten Klasse enthält ein Objekt der Basisklasse als Unterobjekt.
- Alle *öffentlichen* Mitglieder der Basisklasse werden *private* Mitglieder der abgeleiteten Klasse.
- Alle privaten Mitglieder der Basisklasse sind keine Mitglieder der abgeleiteten Klasse.
- Ein Objekt der abgeleiteten Klasse kann *nicht* für ein Objekt der Basisklasse eingesetzt werden!

Zusammenfassung

Wir haben somit drei verschiedene Möglichkeiten kennengelernt, um die Klasse `SimpleFloatArray` für `Polynomial` zu nutzen:

1. Als privates Datenmitglied

2. Mittels öffentlicher Vererbung

3. Mittels privater Vererbung

Bemerkung: Je nach Situation ist die eine oder andere Variante angemessener. Hier hängt viel vom guten Geschmack des Programmierers ab. In diesem speziellen Fall würde ich persönlich Möglichkeit 1 bevorzugen.

Methodenauswahl und virtuelle Funktionen

Motivation: Feld mit Bereichsprüfung

Problem: Die für die Klasse `SimpleFloatArray` implementierte Methode `operator[]` prüft nicht, ob der Index im erlaubten Bereich liegt. Zumindest in der Entwicklungsphase eines Programmes wäre es aber nützlich, ein Feld mit Indexüberprüfung zu haben.

Abhilfe: Ableitung einer Klasse `CheckedSimpleFloatArray`, bei der sich `operator[]` anders verhält.

Programm: Klassendefinition **(CheckedSimpleFloatArray.hh)**:

```
class CheckedSimpleFloatArray :
  public SimpleFloatArray
{
public:
  CheckedSimpleFloatArray( int s, float f );

  // Default-Versionen von copy Konstruktor,
    Zuweisungsoperator
  // und Destruktor sind OK

  // Indizierter Zugriff mit Indexprüfung
  float& operator[]( int i );
};
```

Methodendefinition **(CheckedSimpleFloatArrayImp.cc)**:

```
CheckedSimpleFloatArray::
  CheckedSimpleFloatArray( int s, float f ) :
  SimpleFloatArray( s, f )
{}

float& CheckedSimpleFloatArray::operator[]( int i )
{
  assert( i>=minIndex() && i<=maxIndex() );
  return SimpleFloatArray::operator[](i);
}
```

Verwendung **(UseCheckedSimpleFloatArray.cc)**:

```
#include <iostream>
#include <cassert>

#include "SimpleFloatArray.hh"
#include "SimpleFloatArrayImp.cc"
#include "SimpleFloatArrayIndex.cc"
#include "SimpleFloatArrayCopyCons.cc"
#include "SimpleFloatArrayAssign.cc"

#include "CheckedSimpleFloatArray.hh"
#include "CheckedSimpleFloatArrayImp.cc"

void g( SimpleFloatArray& a )
{
  std::cout << "beginn in g: " << std::endl;
  std::cout << "access: " << a[1] << "  " << a[10] << std::endl;
  std::cout << "ende in g: " << std::endl;
```

```
}

int main()
{
  CheckedSimpleFloatArray a( 10, 0 );
  g( a );
  std::cout << "beginn in main: " << std::endl;
  std::cout << "zugriff in main: " << a[10] << std::endl;
  std::cout << "ende in main: " << std::endl;
}
```

mit der Ausgabe:

```
beginn in g:
access: 0  1.85018e-40
ende in g:
beginn in main:
UseCheckedSimpleFloatArray: CheckedSimpleFloatArrayImp.cc:8:
  float& CheckedSimpleFloatArray::operator[](int):
  Assertion 'i>=minIndex() && i<=maxIndex()' failed.
```

```
Aborted (core dumped)
```

Bemerkung:

- In der Funktion `main` funktioniert die Bereichsprüfung dann wie erwartet.
- In der Funktion g wird hingegen keine Bereichsprüfung durchgeführt, auch wenn sie mit einem Objekt vom Typ `CheckedSimpleFloatArray` aufgerufen wird!
- Warum ist das so?
- In der Funktion g betrachtet der Compiler die übergebene Referenz als Objekt vom Typ SimpleFloatArray und dies hat keine Bereichsprüfung.
- Der Funktionsaufruf mit dem Objekt der öffentlich abgeleiteten Klasse ändert daran nichts! Dies ist eine Konsequenz der Realisierung der Ist-ein-Beziehung.

- Die Auswahl der Methode hängt vom angegeben Typ ab und nicht vom konkreten Objekt welches übergeben wird.

- Meistens ist dies aber nicht das gewünschte Verhalten (vgl. dazu auch die späteren Beispiele).

Virtuelle Funktionen

Idee: Gib dem Compiler genügend Information, so dass er schon bei der Übersetzung von `SimpleFloatArray`-Methoden ein flexibles Verhalten von `[]` möglich macht. In C++ geschieht dies, indem man Methoden in der Basisklasse als virtuell (*virtual*) kennzeichnet.

Programm:

```
class SimpleFloatArray
{
public:
  ...
  virtual float& operator[]( int i );
  ...
private:
  ...
};
```

Beobachtung: Mit dieser Änderung funktioniert die Bereichsprüfung auch in der Funktion `g` in `UseCheckedSimpleFloatArray.cc`: wird sie mit einer Referenz auf ein `CheckedSimpleFloatArray`-Objekt aufgerufen, so wird der Bereichstest durchgeführt, bei Aufruf mit einer Referenz auf ein `SimpleFloatArray`-Objekt aber nicht.

Bemerkung:

- Die Einführung einer virtuellen Funktion erfordert also Änderungen in bereits existierendem Code, nämlich der Definition der Basisklasse!

- Die Implementierung der Methoden bleibt jedoch unverändert.

Implementation: Diese Auswahl der Methode in Abhängigkeit vom tatsächlichen Typ des Objekts kann man dadurch erreichen, dass jedes Objekt entweder Typinformation oder einen Zeiger auf eine Tabelle mit den für seine Klasse virtuell definierten Funktionen mitführt.

Bemerkung:

- Wird eine als virtuell markierte Methode in einer abgeleiteten Klasse neu implementiert, so wird die Methode der abgeleiteten Klasse verwendet, wenn das Objekt für ein Basisklassenobjekt eingesetzt wird.

- Die Definition der Methode in der abgeleiteten Klasse muss genau mit der Definition in der Basisklasse übereinstimmen, ansonsten wird überladen!

- Das Schlüsselwort **virtual** muss in der abgeleiteten Klasse nicht wiederholt werden, es ist aber guter Stil dies zu tun.

- Die Eigenschaften virtueller Funktionen lassen sich nur nutzen, wenn auf das

Objekt über Referenzen oder Zeiger zugegriffen wird! Bei einem Aufruf (*call-by-value*) von

```
void g( SimpleFloatArray a )
{
  cout << a[1] << "  " << a[11] << endl;
}
```

erzeugt der Copy-Konstruktor ein Objekt `a` vom Typ `SimpleFloatArray` (Slicing!) und innerhalb von `g()` wird entsprechend dessen `operator[]` verwendet.

- Virtuelle Funktionen stellen wieder eine Form des Polymorphismus dar („eine Schnittstelle — viele Methoden“).

- Der Zugriff auf eine Methode über die Tabelle virtueller Funktionen ist deutlich ineffizienter, was für Objektorientierung auf niedriger Ebene eine Rolle spielen kann.

- In vielen objektorientierten Sprachen (z. B. Smalltalk, Objective C, Common Lisp/CLOS) verhalten sich alle Methoden „virtuell“.

- In der Programmiersprache Java ist das virtuelle Verhalten der Normalfall, das Default-Verhalten von C++-Methoden kann man aber durch Hinzufügen des Schlüsselworts `static` erreichen.