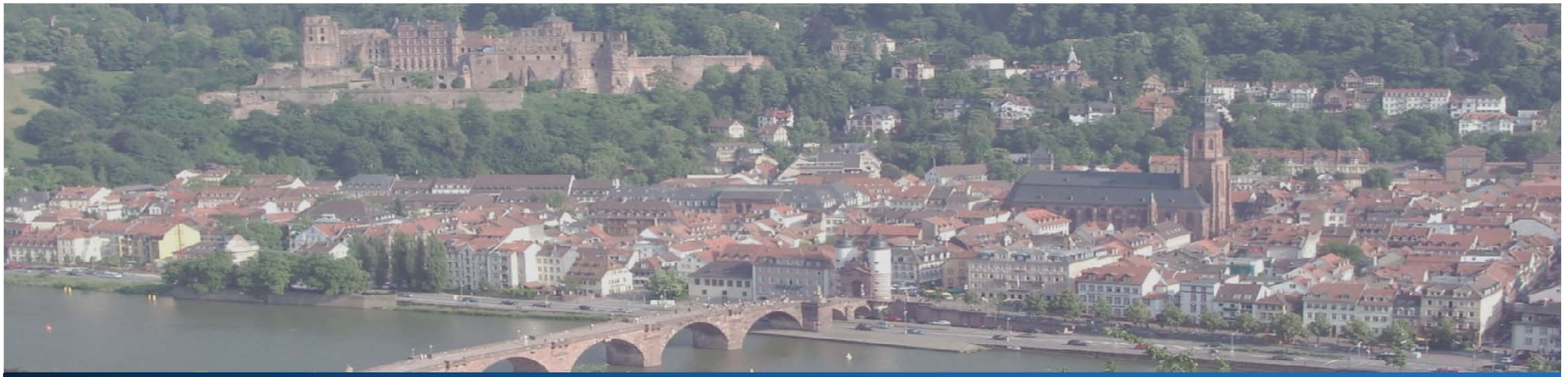
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group

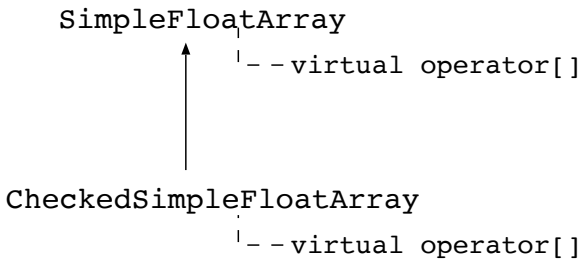


Abstrakte Klassen

- Motivation
- Schnittstellenbasisklassen
- Beispiel: geometrische
- Beispiel: Funktoren
- Beispiel: Exotische Felder
- Zusammenfassung

Motivation

Hatten:



Beobachtung: Beide Klassen besitzen dieselben Methoden und unterscheiden sich nur in der Implementierung von `operator[]`. Wir könnten ebenso `SimpleFloatArray` von `CheckedSimpleFloatArray` ableiten. Das Klassendiagramm drückt diese Symmetrie aber nicht aus.

Grund: `SimpleFloatArray` stellt sowohl die Definition der Schnittstelle eines ADT *Feld* dar, als auch eine Implementierung dieser Schnittstelle. Es ist aber sinnvoll, diese beiden Aspekte zu trennen.

Schnittstellenbasisklassen

Idee: Definiere eine möglichst allgemeine Klasse `FloatArray`, von der sowohl `SimpleFloatArray` als auch `CheckedSimpleFloatArray` abgeleitet werden.

Bemerkung: Oft will und kann man für (virtuelle) Methoden in einer solchen Basisklasse keine Implementierung angeben. In C++ kennzeichnet man sie dann mit dem Zusatz `= 0` am Ende. Solche Funktionen bezeichnet man als rein virtuelle (*engl.: pure virtual*) Funktionen.

Beispiel: **(FloatArray.hh)**

```
class FloatArray
{
public:
   virtual ~FloatArray() {};
   virtual float& operator[]( int i ) = 0;
   virtual int numIndices() = 0;
   virtual int minIndex() = 0;
   virtual int maxIndex() = 0;
   virtual bool isMember( int i ) = 0;
};
```

Bezeichnung: Klassen, die mindestens eine rein virtuelle Funktion enthalten, nennt man abstrakt. Das Gegenteil ist eine konkrete Klasse.

Bemerkung:

- Man kann keine Objekte von abstrakten Klassen instanzieren. Aus diesem Grund haben abstrakte Klassen auch keine Konstruktoren.

- Sehr wohl kann man aber Zeiger und Referenzen dieses Typs haben, die dann aber auf Objekte abgeleiteter Klassen zeigen.

- Eine abstrakte Klasse, die der Definition einer Schnittstelle dient, bezeichnen wir nach Barton/Nackman als Schnittstellenbasisklasse (*interface base class*).

- Schnittstellenbasisklassen enthalten üblicherweise keine Datenmitglieder und alle Methoden sind rein virtuell.

- Die Implementierung dieser Schnittstelle erfolgt in abgeleiteten Klassen.

Bemerkung: (Virtueller Destruktor) Eine Schnittstellenbasisklasse sollte einen virtuellen Destruktor

```
virtual ~FloatArray();
```

mit einer Dummy-Implementierung

```
FloatArray::~FloatArray() {}
```

besitzen, damit man dynamisch erzeugte Objekte abgeleiter Klassen durch die Schnittstelle der Basisklasse löschen kann. Beispiel:

```
void g( FloatArray* p )
{
  delete p;
}
```

Bemerkung: Der Destruktor darf nicht rein virtuell sein, da der Destruktor abgeleiteter Klassen einen Destruktor der Basisklasse aufrufen will.

Beispiel: geometrische Formen

Aufgabe: Wir wollen mit zweidimensionalen geometrischen Formen arbeiten. Dies sind von einer Kurve umschlossene Flächen wie Kreis, Rechteck, Dreieck, ...

Programm: Eine mögliche C++-Implementierung wäre folgende **(shape.cc)**:

```
#include <iostream>
#include <cmath>

const double pi = 3.1415926536;

class Shape
{
public:
  virtual ~Shape() {};
  virtual double area() = 0;
  virtual double diameter() = 0;
  virtual double circumference() = 0;
};
```

```
// works on every shape
double circumference_to_area( Shape& shape )
{
  return shape.circumference() / shape.area();
}

class Circle : public Shape
{
public:
  Circle( double r ) { radius = r; }
  virtual double area()
  {
    return pi * radius * radius;
  }
  virtual double diameter()
  {
    return 2 * radius;
  }
```

```
  virtual double circumference()
  {
    return 2 * pi * radius;
  }
private:
  double radius;
};

class Rectangle : public Shape
{
public:
  Rectangle( double aa, double bb )
  {
    a = aa; b = bb;
  }
  virtual double area() { return a*b; }
  virtual double diameter()
  {
    return sqrt( a*a + b*b );
```

```
  }
  virtual double circumference()
  {
    return 2 * (a + b);
  }
private:
  double a, b;
};

int main()
{
  Rectangle unit_square( 1.0, 1.0 );
  Circle unit_circle( 1.0 );
  Circle unit_area_circle( 1.0 / sqrt( pi ) );

  std::cout << "Das Verhaeltnis von Umfang zu Flaeche betraegt\n";
  std::cout << "Einheitsquadrat:      "
            << circumference_to_area( unit_square )
            << std::endl;
```

```
    std::cout << "Kreis mit Flaeche 1: "
              << circumference_to_area( unit_area_circle )
              << std::endl;
    std::cout << "Einheitskreis:       "
              << circumference_to_area( unit_circle )
              << std::endl;
    return 0;
}
```

Ergebnis: Wir erhalten als Ausgabe des Programms:

```
Das Verhaeltnis von Umfang zu Flaeche betraegt
Einheitsquadrat:     4
Kreis mit Flaeche 1: 3.54491
Einheitskreis:       2
```

Beispiel: Funktoren

$\rightarrow$ Klasse der Objekte, die sich wie Funktionen verhalten.

Hatten: Definition einer Inkrementierer-Klasse in `Inkrementierer.cc`. Nachteile waren:

- Möchten beliebige Funktion auf die Listenelemente anwenden können.
- Syntax `ink.eval(...)` nicht optimal.

Dies wollen wir nun mit Hilfe einer Schnittstellenbasisklasse und der Verwendung von `operator()` verbessern.

Programm: **(Funktor.cc)**

```
#include <iostream>

class Function
{
public:
  virtual ~Function() {};
  virtual int operator()( int ) = 0;
};

class Inkrementierer : public Function
{
public:
  Inkrementierer( int n ) { inkrement = n; }
  virtual int operator()( int n ) { return n + inkrement; }
private:
  int inkrement;
};
```

```
void schleife( Function& func )
{
  for ( int i=1; i<10; i++ )
    std::cout << func( i ) << " ";
  std::cout << std::endl;
}

class Quadrat : public Function
{
public:
  virtual int operator()( int n ) { return n * n; }
};

int main()
{
  Inkrementierer ink( 10 );
  Quadrat quadrat;
  schleife( ink );
```

```
    schleife( quadrat );
}
```

Bemerkung: Unangenehm ist jetzt eigentlich nur noch, dass der Argument- und Rückgabetyp der Methode auf **int** festgelegt ist. Dies wird bald durch Schablonen (*Templates*) behoben werden.

Wiederholung: Virtuelle Funktionen

```
class A { public: virtual void f() {} };

class B : public A { public: virtual void f() {} };

void g( A& a ) { a.f(); }
```

Abgeleitete Klasse definiert Methode mit gleichem Namen und Argumenten wie in der Basisklasse.

In g darf Objekt der Klasse A oder B eingesetzt werden.

Mit **virtual** wird in der Funktion g die Methodenauswahl am konkreten Objekt vorgenommen.

Ohne **virtual** wird in der Funktion g die Methodenauswahl anhand des Typs A vorgenommen.

Achtung: Ohne Referenz wird in jedem Fall eine Kopie erstellt und die Methode der Klasse A verwendet.

Abstrakte Klassen haben mindestens eine rein virtuelle Methode.

Schnittstellenbasisklassen dienen zur Definition einer Schnittstelle und haben in der Regel nur rein virtuelle Methoden und keine Datenmitglieder.

Beispiel: Exotische Felder

Programm: Wir definieren folgende (schon gezeigte) Schnittstellenbasisklasse **(FloatArray.hh)**:

```
class FloatArray
{
public:
  virtual ~FloatArray() {};
  virtual float& operator[]( int i ) = 0;
  virtual int numIndices() = 0;
  virtual int minIndex() = 0;
  virtual int maxIndex() = 0;
  virtual bool isMember( int i ) = 0;
};
```

Von dieser kann man leicht `SimpleFloatArray` ableiten. Außerdem passt die Schnittstelle auf weitere Feldtypen, was wir im Folgenden zeigen wollen.

Dynamisches Feld

Wir wollen jetzt ein Feld mit variabel großer, aber zusammenhängender Indexmenge $I = \{o, o+1, \ldots, o+n-1\}$ mit $o, n \in \mathbb{Z}$ und $n \geq 0$ definieren. Wir gehen dazu folgendermaßen vor:

- Der Konstruktor fängt mit einem Feld der Länge 0 an ($o = n = 0$).
- `operator[]` prüft, ob $i \in I$ gilt; wenn nein, so wird der Indexbereich erweitert, ein entsprechendes Feld allokiert, die Werte aus dem alten Feld werden in das neue kopiert und das alte danach freigegeben.

Programm: (DFA.cc)

```
class DynamicFloatArray : public FloatArray
{
public:
  DynamicFloatArray() { n = 0; o = 0; p = new float[1]; }
  virtual ~DynamicFloatArray() { delete[] p; }
  virtual float& operator[]( int i );
  virtual int numIndices() { return n; }
  virtual int minIndex() { return o; }
  virtual int maxIndex() { return o + n - 1; }
  virtual bool isMember( int i ) { return (i >= o) && (i < o+n); }
private:
  int n;        // Anzahl Elemente
  int o;        // Ursprung der Indexmenge
  float* p;     // Zeiger auf built-in array
};

float& DynamicFloatArray::operator[]( int i )
```

```
{
  if ( i < o || i >= o+n )
  {     // resize
    int new_o, new_n;
    if ( i < o ) {
      new_o = i;
      new_n = n + o - i;
    }
    else
    {
      new_o = o;
      new_n = i - o + 1;
    }
    float* q = new float[new_n];
    for ( int j=0; j<new_n; j=j+1 ) q[j] = 0.0;
    for ( int j=0; j<n; j=j+1 )
      q[j + o - new_o] = p[j];
    delete[] p;
    p = q;
```

```
      n = new_n;
      o = new_o;
    }
    return p[i - o];
  }
```

Bemerkung:

- Im Konstruktor wird der Einfachheit halber bereits ein **float** allokiert.
- Der Copy-Konstruktor und Zuweisungsoperator müssten auch implementiert werden (um zu vermeiden, dass der Zeiger kopiert wird).

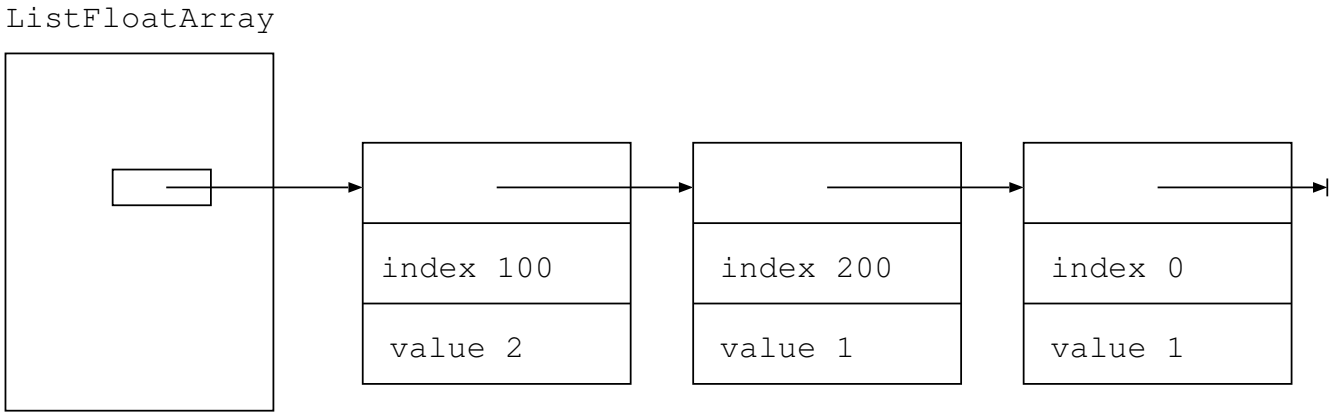
Listenbasiertes Feld

Problem: Wenn man `DynamicFloatArray` zur Darstellung von Polynomen verwendet, so werden Polynome mit vielen Nullkoeffizienten, z. B.

$$p(x) = x^{100} + 1 \text{ oder } q(x) = p^2(x) = x^{200} + 2x^{100} + 1$$

sehr ineffizient verwaltet.

Abhilfe: Speichere die Elemente des Feldes als einfach verkettete *Liste* von Index–Wert–Paaren:



Programm: (LFA.cc)

```
class ListFloatArray :
  public FloatArray
{
public:
  ListFloatArray();                    // leeres Feld
  virtual ~ListFloatArray();           // ersetzt ~FloatArray

  virtual float& operator[]( int i );
  virtual int numIndices();
  virtual int minIndex();
  virtual int maxIndex();
  virtual bool isMember( int i );
private:
  struct FloatListElem
  { // lokale Struktur
    FloatListElem* next;
    int index;
```

```
    float value;
  };
  FloatListElem* insert( int i, float v );
  FloatListElem* find( int i );

  int n;              // Anzahl Elemente
  FloatListElem* p; // Listenanfang
};

// private Hilfsfunktionen
ListFloatArray::FloatListElem*
  ListFloatArray::insert( int i, float v )
{
  FloatListElem* q = new FloatListElem;

  q->index = i;
  q->value = v;
  q->next = p;
  p = q;
```

```
  n = n + 1;
  return q;
}

ListFloatArray::FloatListElem*
  ListFloatArray::find( int i )
{
  for ( FloatListElem* q=p; q!=0; q=q->next )
    if ( q->index == i )
      return q;
  return 0;
}

// Konstruktor
ListFloatArray::ListFloatArray()
{
  n = 0; // alles leer
  p = 0;
}
```

```
// Destruktor
ListFloatArray::~ListFloatArray()
{
  while ( p != 0 )
  {
    FloatListElem* q;
    q = p;              // q ist erstes
    p = q->next;        // entferne q aus Liste
    delete q;
  }
}

float& ListFloatArray::operator[]( int i )
{
  FloatListElem* r = find( i );
  if ( r == 0 )
    r = insert( i, 0.0 ); // index einfuegen
  return r->value;
```

```
}

int ListFloatArray::numIndices()
{
  return n;
}

int ListFloatArray::minIndex()
{
  if ( p == 0 ) return 0;
  int min = p->index;
  for ( FloatListElem* q=p->next; q!=0; q=q->next )
    if ( q->index < min ) min = q->index;
  return min;
}

int ListFloatArray::maxIndex()
{
  if ( p == 0 ) return 0;
```

```
  int max = p->index;
  for ( FloatListElem* q=p->next; q!=0; q=q->next )
    if ( q->index > max ) max = q->index;
  return max;
}

bool ListFloatArray::isMember( int i )
{
  return ( find( i ) != 0 );
}
```

Bemerkung:

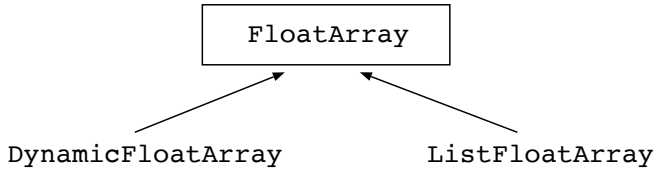
- Das Programm verwendet eine unsortierte Liste. Man hätte auch eine sortierte Liste verwenden können (Vorteile?).

- Für die Index–Wert–Paare wird innerhalb der Klassendefinition der zusammengesetzte Datentyp `FloatListElem` definiert.

- Die privaten Methoden dienen der Manipulation der Liste und werden in der Implementierung der öffentlichen Methoden verwendet. Merke: Innerhalb einer Klasse können wiederum Klassen definiert werden!

Anwendung

Wir haben somit folgendes Klassendiagramm:



Da sowohl `DynamicFloatArray` als auch `ListFloatArray` die durch `FloatArray` definierte Schnittstelle erfüllen, kann man nun Methoden für `FloatArray` schreiben, die auf beiden abgeleiteten Klassen funktionieren.

Als Beispiel betrachten wir folgendes Programm, welches `FloatArray` wieder zur Polynom-Multiplikation verwendet (der Einfachheit halber ohne es in eine Klasse `Polynomial` zu packen).

Programm: (UseFloatArray.cc)

```
#include <iostream>

#include "FloatArray.hh"
#include "DFA.cc"
#include "LFA.cc"

void polyshow( FloatArray& f )
{
  for ( int i=f.minIndex(); i<=f.maxIndex(); i=i+1 )
    if ( f.isMember( i ) && f[i] != 0.0 )
      std::cout << "+" << f[i] << "*x^" << i;
  std::cout << std::endl;
}

void polymul( FloatArray& a, FloatArray& b, FloatArray& c )
{
  // Loesche a
```

```
  for ( int i=a.minIndex(); i<=a.maxIndex(); i=i+1 )
    if ( a.isMember( i ) )
      a[i] = 0.0;

  // a = b*c
  for ( int i=b.minIndex(); i<=b.maxIndex(); i=i+1 )
    if ( b.isMember( i ) )
      for ( int j=c.minIndex(); j<=c.maxIndex(); j=j+1 )
        if ( c.isMember( j ) )
          a[i+j] = a[i+j] + b[i] * c[j];
}

int main()
{
  // funktioniert mit einer der folgenden Zeilen:
  // DynamicFloatArray f, g;
  ListFloatArray f, g;

  f[0] = 1.0; f[100] = 1.0;
```

```
    polymul( g, f, f );
    polymul( f, g, g );
    polymul( g, f, f );
    polymul( f, g, g ); // f = (1 + x^100)^16

    polyshow( f );
}
```

Ausgabe:

```
+1*x^0+16*x^1000+120*x^2000+560*x^3000+1820*x^4000+4368*x^5000+8008*x^6000
+11440*x^7000+12870*x^8000+11440*x^9000+8008*x^10000+4368*x^11000
+1820*x^12000+560*x^13000+120*x^14000+16*x^15000+1*x^16000
```

Bemerkung:

- Man kann nun sehr „spät“, nämlich erst in der `main`-Funktion entscheiden, mit welcher Art Felder man tatsächlich arbeiten will.
- Je nachdem, wie vollbesetzt der Koeffizientenvektor ist, ist entweder `DynamicFloatArray` oder `ListFloatArray` günstiger.
- Schlecht ist noch die Weise, in der allgemeine Schleifen über das Feld implementiert werden. Die Anwendung auf `ListFloatArray` ist sehr ineffektiv! Eine Abhilfe werden wir bald kennenlernen (Iteratoren).

Zusammenfassung

In diesem Abschnitt haben wir gezeigt, wie man mit Hilfe von Schnittstellenbasisklassen eine Trennung von

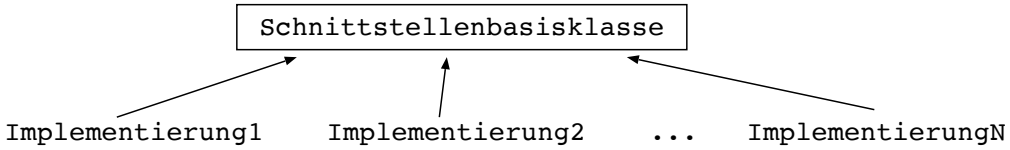
- Schnittstellendefinition und
- Implementierung

erreicht.

Dies gelingt durch

- rein virtuelle Funktionen in Verbindung mit
- Vererbung.

Typischerweise erhält man Klassendiagramme der Form:



Man *erzeugt* Objekte konkreter (abgeleiteter) Klassen und *benutzt* diese Objekte durch die Schnittstellenbasisklasse:

Create objects, use interfaces!