Schlüsselwort `virtual`

- Polymorphie
- Virtuelle Methoden
- `virtual`

Polymorphie

- Jedes Objekt der abgeleiteten Klasse ist auch ein Objekt der Basisklasse
  - Vererbung impliziert immer ist-ein-Beziehung
- Jede Klasse definiert einen Datentyp
  - Objekte können mehrere Typen haben
  - Objekte abgeleiteter Klassen haben mindestens zwei Datentypen:
    - Typ der abgeleiteten Klasse
    - und Typ der Basisklasse
    - **BSP:** im letztem Beispiel ist Vektor vom Typ `Vector` und `Matrix`
- kann den jeweils passenden Typ verwenden
  - Diese Eigenschaft nennt man Polymorphie (*griech.* Vielgestaltigkeit)
- Das hat insbesondere Konsequenzen für Pointer!

Pointer und `virtual` 1/3

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   void print() {cout << "Basisklasse\n";}
 7 };
 8
 9 class Abgeleitet : public Basisklasse {
10 public:
11   void print() {cout << "Abgeleitet\n";}
12 };
13
14 int main() {
15   Abgeleitet a;
16   Abgeleitet* pA = &a;
17   Basisklasse* pB = &a;
18   pA->print();
19   pB->print();
20   return 0;
21 }
```

- Output:

  ```
  Abgeleitet
  Basisklasse
  ```

- Zeile 15: Objekt a vom Typ `Abgeleitet` ist auch vom Typ `Basisklasse`
- Pointer auf `Basisklasse` mit Adresse von `a` möglich
- Zeile 19 ruft `print` aus `Basisklasse` auf
  - i.a. soll `print` aus `Abgeleitet` verwendet werden

Pointer und `virtual` 2/3

```
 1 #include <iostream>
 2 using std::cout;
 3
 4 class Basisklasse {
 5 public:
 6   virtual void print() {cout << "Basisklasse\n";}
 7 };
 8
 9 class Abgeleitet : public Basisklasse {
10 public:
11   void print() {cout << "Abgeleitet\n";}
12 };
13
14 int main() {
15   Abgeleitet a;
16   Abgeleitet* pA = &a;
17   Basisklasse* pB = &a;
18   pA->print();
19   pB->print();
20   return 0;
21 }
```

- Output:

  ```
  Abgeleitet
  Abgeleitet
  ```

- Zeile 6: neues Schlüsselwort `virtual`
  - vor Signatur der Methode `print` (in Basisklasse!)
  - deklariert virtuelle Methode
  - zur Laufzeit wird korrekte Methode aufgerufen
    - **Varianten müssen gleiche Signatur haben**
  - Zeile 19 ruft nun redefinierte Methode `print` auf

Pointer und virtual 3/3

```
1 #include <iostream>
2 using std::cout;
3
4 class Basisklasse {
5 public:
6   virtual void print() {cout << "Basisklasse\n";}
7 };
8
9 class Abgeleitet1 : public Basisklasse {
10 public:
11   void print() {cout << "Nummer 1\n";}
12 };
13
14 class Abgeleitet2 : public Basisklasse {
15 public:
16   void print() {cout << "Nummer 2\n";}
17 };
18
19 int main() {
20   Basisklasse* var[2];
21   var[0] = new Abgeleitet1;
22   var[1] = new Abgeleitet2;
23
24   for (int j=0; j<2; ++j) {
25     var[j]->print();
26   }
27   return 0;
28 }
```

- Output:

  ```
  Nummer 1
  Nummer 2
  ```

- `var` ist Vektor mit Objekten verschiedener Typen!

Destruktor und virtual 1/2

```
1 #include <iostream>
2 using std::cout;
3
4 class Basisklasse {
5 public:
6   ~Basisklasse() {
7     cout << "~Basisklasse()\n";
8   }
9 };
10
11 class Abgeleitet : public Basisklasse {
12 public:
13   ~Abgeleitet() {
14     cout << "~Abgeleitet()\n";
15   }
16 };
17
18 int main() {
19   Basisklasse* var = new Abgeleitet;
20   delete var;
21   return 0;
22 }
```

- Output:

  ```
  ~Basisklasse()
  ```

- Destruktor von `Abgeleitet` wird nicht aufgerufen!
  - ggf. entsteht toter Speicher, falls `Abgeleitet` zusätzlichen dynamischen Speicher anlegt
- Destruktoren werden deshalb üblicherweise als virtual deklariert

Destruktor und virtual 2/2

```
1 #include <iostream>
2 using std::cout;
3
4 class Basisklasse {
5 public:
6   virtual ~Basisklasse() {
7     cout << "~Basisklasse()\n";
8   }
9 };
10
11 class Abgeleitet : public Basisklasse {
12 public:
13   ~Abgeleitet() {
14     cout << "~Abgeleitet()\n";
15   }
16 };
17
18 int main() {
19   Basisklasse* var = new Abgeleitet;
20   delete var;
21   return 0;
22 }
```

- Output:

  ```
  ~Abgeleitet()
  ~Basisklasse()
  ```

- `Destruktor von Abgeleitet wird aufgerufen`
  - `ruft implizit Destruktor von Basisklasse auf`

Virtuelle Methoden 1/2

```
1 #include <iostream>
2 using std::cout;
3
4 class Basisklasse {
5 public:
6   void ego() { cout << "Basisklasse\n"; }
7   void print() { cout << "Ich bin "; ego(); }
8 };
9
10 class Abgeleitet1: public Basisklasse {
11 public:
12   void ego() { cout << "Nummer 1\n"; }
13 };
14
15 class Abgeleitet2: public Basisklasse {};
16
17 int main() {
18   Basisklasse var0;
19   Abgeleitet1 var1;
20   Abgeleitet2 var2;
21   var0.print();
22   var1.print();
23   var2.print();
24   return 0;
25 }
```

- Output:

  ```
  Ich bin Basisklasse
  Ich bin Basisklasse
  Ich bin Basisklasse
  ```

- Obwohl `ego` redefiniert wird für `Abgeleitet1`, bindet `print` immer `ego` von `Basisklasse` ein

Virtuelle Methoden 2/2

```
 1  #include <iostream>
 2  using std::cout;
 3
 4  class Basisklasse {
 5  public:
 6    virtual void ego() { cout << "Basisklasse\n"; }
 7    void print() { cout << "Ich bin "; ego(); }
 8  };
 9
10  class Abgeleitet1: public Basisklasse {
11  public:
12    void ego() { cout << "Nummer 1\n"; }
13  };
14
15  class Abgeleitet2: public Basisklasse {};
16
17  int main() {
18    Basisklasse var0;
19    Abgeleitet1 var1;
20    Abgeleitet2 var2;
21    var0.print();
22    var1.print();
23    var2.print();
24    return 0;
25  }
```

- Output:

```
Ich bin Basisklasse
Ich bin Nummer 1
Ich bin Basisklasse
```

- `virtual` (Zeile 6) sorgt für korrekte Einbindung, falls diese für abgeleitete Klasse redefiniert ist

Abstrakte Klassen

- Manchmal werden Klassen nur zur Strukturierung / zum Vererben angelegt, aber Instanzierung ist nicht sinnvoll / nicht gewollt
  - d.h. es soll keine Objekte der Basisklasse geben
  - sog. **abstrakte Klassen**
  - dient nur als Schablone für abgeleitete Klassen
- abstrakte Klassen können nicht instanziert werden
  - Compiler liefert Fehlermeldung!
- In C++ ist eine Klasse abstrakt, falls eine Methode existiert der Form

  `virtual return-type method( ... ) = 0;`
  - Diese sog. **abstrakte Methode** muss in allen abgeleiteten Klassen implementiert werden
    - wird nicht in Basisklasse implementiert

Beispiel zu abstrakten Klassen

```
 1 #include<cmath>
 2
 3 class Figur {
 4   double schwerPunkt[2];
 5 public:
 6   virtual double getFlaeche() = 0;
 7 };
 8
 9 class Kreis : public Figur {
10   double radius;
11 public:
12   double getFlaeche() {
13     return radius*radius*3.14159;
14   }
15 };
16
17 class Dreieck : public Figur {
18   double a[2],b[2],c[2];
19 public:
20   double getFlaeche() {
21     return fabs(0.5*( (b[0]-a[0])*(c[1]-a[1])
22                      -(c[0]-a[0])*(b[1]-a[1])));
23   }
24 };
```

- Abstrakte Klasse `Figur`
  - durch abstrakte Methode `getFlaeche` (Zeile 6)
- abgeleitete Klassen `Kreis`, `Dreieick`
- `Kreis` und `Dreieck` redefinieren `getFlaeche`
  - alle abstrakten Meth. müssen redefiniert werden

Beispiel zu `virtual`: Matrizen

- für allgemeine Matrix $A \in \mathbb{R}^{m \times n}$
  - Vektoren $x \in \mathbb{R}^m \simeq \mathbb{R}^{m \times 1}$
  - quadratische Matrix $A \in \mathbb{R}^{n \times n}$
    - reguläre Matrix: $\det(A) \neq 0$
    - symmetrische Matrix: $A = A^T$
    - untere Dreiecksmatrix, $A_{jk} = 0$ für $k > j$
    - obere Dreiecksmatrix, $A_{jk} = 0$ für $k < j$
- symmetrischen Matrizen und Dreiecksmatrizen brauchen generisch weniger Speicher
  - $\frac{n(n+1)}{2}$ statt n^2
- muss Koeffizientenzugriff überladen
  - Koeffizientenzugriff in `Matrix` muss `virtual` sein, damit Methoden für `Matrix` z.B. auch für symmetrische Matrizen anwendbar

matrix.hpp 1/3

```
 1 #ifndef MATRIX
 2 #define MATRIX
 3 #include <cmath>
 4 #include <cassert>
 5 #include <iostream>
 6
 7 class Matrix {
 8 private:
 9   int m;
10   int n;
11   int storage;
12   double* coeff;
13
14 protected:
15   // methods such that subclasses can access data fields
16   void allocate(int m, int n, int storage, double init);
17   const double* getCoeff() const;
18   double* getCoeff();
19   int getStorage() const;
```

- abgeleitete Klassen, z.B. `SymmetricMatrix` können auf Datenfelder nicht zugreifen, da `hidden` nach Vererbung
  - muss Zugriffsfunktionen schaffen
  - `protected` stellt sicher, dass diese Methoden nur in den abgeleiteten Klassen verwendet werden können (aber nicht von Außen!)
- `SymmetricMatrix` hat weniger Speicher als `Matrix`
  - muss Allocation als Methode bereitstellen
    - $m \cdot n$ Speicherplätze für $A \in \mathbb{R}^{m \times n}$
    - nur $\frac{n(n+1)}{2} = \sum_{i=1}^{n} i$ für $A = A^T \in \mathbb{R}^{n \times n}$

matrix.hpp 2/3

```
21 public:
22   // constructors, destructor, assignment
23   Matrix();
24   Matrix(int m, int n, double init=0);
25   Matrix(const Matrix&);
26   ~Matrix();
27   Matrix& operator=(const Matrix&);
28
29   // return size of matrix
30   int size1() const;
31   int size2() const;
32
33   // read and write entries with matrix access A(j,k)
34   virtual const double& operator()(int j, int k) const;
35   virtual double& operator()(int j, int k);
```

- Destruktor nicht virtuell, da abgeleitete Klassen keinen dynamischen Speicher haben
- Koeffizienten-Zugriff muss `virtual` sein, da z.B. symmetrische Matrizen anders gespeichert
  - `virtual` nur in Klassendefinition, d.h. generisch im Header-File
- Funktionalität wird mittels Koeff.-Zugriff `A(j,k)` realisiert, z.B. `operator+`, `operator*`
  - kann alles auch für symm. Matrizen nutzen
  - nur 1x für Basisklasse implementieren
    - manchmal ist Redefinition sinnvoll für effizientere Lösung

matrix.hpp 3/3

```
37   // read and write storage vector A[ell]
38   const double& operator[](int ell) const;
39   double& operator[](int ell);
40
41   // compute norm
42   double norm() const;
43
44   // print matrix
45   void print() const;
46 };
47
48 // matrix-matrix sum and product
49 const Matrix operator+(const Matrix&, const Matrix&);
50 const Matrix operator*(const Matrix&, const Matrix&);
51
52 #endif
```

- Operator `[ ]` für effiziente Implementierung
  - z.B. Addition bei gleichem Matrix-Typ
  - d.h. bei gleicher interner Speicherung
    - muss nur Speichervektoren addieren
- Implementierung von `norm`, `print`, `operator+`, `operator*` mittels Koeffizienten-Zugriff `A(j,k)`
  - direkt für abgeleitete Klassen anwendbar

matrix.cpp 1/5

```
 1 #include "matrix.hpp"
 2 using std::cout;
 3
 4 void Matrix::allocate(int m, int n, int storage,
 5                                              double init) {
 6   assert(m>=0);
 7   assert(n>=0);
 8   assert(storage>=0 && storage<=m*n);
 9   this->m = m;
10   this->n = n;
11   this->storage = storage;
12   if (storage>0) {
13     coeff = new double[storage];
14     for (int ell=0; ell<storage; ++ell) {
15       coeff[ell] = init;
16     }
17   }
18   else {
19     coeff = (double*) 0;
20   }
21 }
22
23 const double* Matrix::getCoeff() const {
24   return coeff;
25 }
26
27 double* Matrix::getCoeff() {
28   return coeff;
29 }
30
31 int Matrix::getStorage() const {
32   return storage;
33 }
```

matrix.cpp 2/5

```
35 Matrix::Matrix() {
36   m = 0;
37   n = 0;
38   coeff = (double*) 0;
39   cout << "constructor, empty\n";
40 }
41
42 Matrix::Matrix(int m, int n, double init) {
43   allocate(m,n,m*n,init);
44   cout << "constructor, " << m << " x " << n << "\n";
45 }
46
47 Matrix::Matrix(const Matrix& rhs) {
48   m = rhs.size1();
49   n = rhs.size2();
50   storage = m*n;
51   if (storage > 0) {
52     coeff = new double[storage];
53     for (int j=0; j<m; ++j) {
54       for (int k=0; k<n; ++k) {
55         (*this)(j,k) = rhs(j,k);
56       }
57     }
58   }
59   else {
60     coeff = (double*) 0;
61   }
62   cout << "copy constructor, " << m << " x " << n << "\n";
63 }
64
65 Matrix::~Matrix() {
66   if (storage > 0) {
67     delete[] coeff;
68   }
69   cout << "destructor, " << m << " x " << n << "\n";
70 }
```


matrix.cpp 3/5

```
72 Matrix& Matrix::operator=(const Matrix& rhs) {
73   if (storage > 0) {
74     delete[] coeff;
75   }
76   m = rhs.size1();
77   n = rhs.size2();
78   storage = m*n;
79   if (storage > 0) {
80     coeff = new double[storage];
81     for (int j=0; j<m; ++j) {
82       for (int k=0; k<n; ++k) {
83         (*this)(j,k) = rhs(j,k);
84       }
85     }
86   }
87   else {
88     coeff = (double*) 0;
89   }
90   cout << "deep copy, " << m << " x " << n << "\n";
91   return *this;
92 }
93
94 int Matrix::size1() const {
95   return m;
96 }
97
98 int Matrix::size2() const {
99   return n;
100 }
```


matrix.cpp 4/5

```
102 const double& Matrix::operator()(int j, int k) const {
103   assert(j>=0 && j<m);
104   assert(k>=0 && k<n);
105   return coeff[j+k*m];
106 }
107
108 double& Matrix::operator()(int j, int k) {
109   assert(j>=0 && j<m);
110   assert(k>=0 && k<n);
111   return coeff[j+k*m];
112 }
113
114 const double& Matrix::operator[](int ell) const {
115   assert( ell>=0 && ell<storage );
116   return coeff[ell];
117 }
118
119 double& Matrix::operator[](int ell) {
120   assert( ell>=0 && ell<storage );
121   return coeff[ell];
122 }
123
124 double Matrix::norm() const {
125   double norm = 0;
126   for (int j=0; j<m; ++j) {
127     for (int k=0; k<n; ++k) {
128       norm = norm + (*this)(j,k) * (*this)(j,k);
129     }
130   }
131   return sqrt(norm);
132 }
133
134 void Matrix::print() const {
135   for (int j=0; j<m; j++) {
136     for (int k=0; k<n; k++) {
137       cout << " " << (*this)(j,k);
138     }
139     cout << "\n";
140   }
141 }
```


matrix.cpp 5/5

```
143 const Matrix operator+(const Matrix& A, const Matrix& B) {
144   int m = A.size1();
145   int n = A.size2();
146   assert(m == B.size1() );
147   assert(n == B.size2() );
148   Matrix sum(m,n);
149   for (int j=0; j<m; ++j) {
150     for (int k=0; k<n; ++k) {
151       sum(j,k) = A(j,k) + B(j,k);
152     }
153   }
154   return sum;
155 }
156
157 const Matrix operator*(const Matrix& A, const Matrix& B) {
158   int m = A.size1();
159   int n = A.size2();
160   int p = B.size2();
161   double sum = 0;
162   assert(n == B.size1() );
163   Matrix product(m,p);
164   for (int i=0; i<m; ++i) {
165     for (int k=0; k<p; ++k) {
166       sum = 0;
167       for (int j=0; j<n; ++j) {
168         sum = sum + A(i,j)*B(j,k);
169       }
170       product(i,k) = sum;
171     }
172   }
173   return product;
174 }
```

Bemerkungen

- da Matrizen spaltenweise gespeichert sind, sollte man eigentlich bei der Reihenfolge der Schleifen beachten, z.B.

```
for (int j=0; j<m; ++j) {
  for (int k=0; k<n; ++k) {
    (*this)(j,k) = rhs(j,k);
  }
}
```

  besser ersetzen durch

```
for (int k=0; k<n; ++k) {
  for (int j=0; j<m; ++j) {
    (*this)(j,k) = rhs(j,k);
  }
}
```

- Speicherzugriff ist dann schneller,
  - es wird nicht im Speicher herumgesprungen
- weitere Funktionen zu `Matrix` sind denkbar
  - Vorzeichen
  - Skalaramultiplikation
  - Matrizen subtrahieren
- weitere Methoden zu `Matrix` sind denkbar
  - Matrix transponieren

squareMatrix.hpp

```
1 #ifndef SQUAREMATRIX
2 #define SQUAREMATRIX
3 #include "matrix.hpp"
4 #include <cassert>
5 #include <iostream>
6
7 class SquareMatrix : public Matrix {
8 public:
9   // constructor, type cast, and destructor
10  SquareMatrix();
11  SquareMatrix(int n,double init=0);
12  SquareMatrix(const Matrix&);
13  ~SquareMatrix();
14
15  // further members
16  int size() const;
17 };
18
19 #endif
```

- Jede quadratische Matrix $A \in \mathbb{R}^{n\times n}$ ist insb. eine allgemeine Matrix $A \in \mathbb{R}^{m\times n}$
  - zusätzliche Funktion: z.B. $\det(A)$ berechnen
  - hier wird nur `SquareMatrix` von `Matrix` abgeleitet
  - keine zusätzliche Funktionalität, nur
    - Standardkonstruktor und Konstruktor
    - Type Cast `Matrix` auf `SquareMatrix`
    - Destruktor
    - `size` als Vereinfachung von `size1`, `size2`

squareMatrix.cpp

```
1 #include "squareMatrix.hpp"
2 using std::cout;
3
4 SquareMatrix::SquareMatrix() {
5   cout << "constructor, empty square matrix\n";
6 }
7
8 SquareMatrix::SquareMatrix(int n, double init) :
9                                         Matrix(n,n,init) {
10  cout << "constr., square matrix, size " << n << "\n";
11 };
12
13 SquareMatrix::SquareMatrix(const Matrix& rhs) :
14                         Matrix(rhs.size1(),rhs.size2()) {
15  assert(size1() == size2());
16  for (int j=0; j<size(); ++j) {
17    for (int k=0; k<size(); ++k) {
18      (*this)(j,k) = rhs(j,k);
19    }
20  }
21  cout << "type cast Matrix -> SquareMatrix\n";
22 }
23
24 SquareMatrix::~SquareMatrix() {
25  cout << "destructor, square matrix\n";
26 }
27
28 int SquareMatrix::size() const {
29  return size1();
30 }
```

- Type Cast garantiert, dass `rhs` $\in \mathbb{R}^{m\times n}$ mit $m = n$
  - d.h. Konversion auf `SquareMatrix` ohne Verlust
- theoretisch auch Cast durch Abschneiden sinnvoll
  - hier aber anders!

Demo zu squareMatrix 1/3

```
1 #include "matrix.hpp"
2 #include "squareMatrix.hpp"
3
4 using std::cout;
5
6 int main() {
7   int n = 3;
8
9   cout << "*** init A\n";
10  SquareMatrix A(n);
11  for (int ell=0; ell<n*n; ++ell) {
12    A[ell] = ell;
13  }
14  cout << "A =\n";
15  A.print();
16
17  cout << "*** init B\n";
18  Matrix B = A;
19
20  cout << "*** init C\n";
21  SquareMatrix C = A + B;
22  cout << "C=\n";
23  C.print();
24
25  cout << "*** terminate\n";
26  return 0;
27 }
```

- erwartetes Resultat:
  - $A = \begin{pmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{pmatrix} = B,$
  - $C = A + B = \begin{pmatrix} 0 & 6 & 12 \\ 2 & 8 & 14 \\ 4 & 10 & 16 \end{pmatrix},$

Demo zu squareMatrix 2/3

```
 1 #include "matrix.hpp"
 2 #include "squareMatrix.hpp"
 3
 4 using std::cout;
 5
 6 int main() {
 7   int n = 3;
 8
 9   cout << "*** init A\n";
10   SquareMatrix A(n);
11   for (int ell=0; ell<n*n; ++ell) {
12     A[ell] = ell;
13   }
14   cout << "A =\n";
15   A.print();
16
17   cout << "*** init B\n";
18   Matrix B = A;
```

- Output:

```
*** init A
constructor, 3 x 3
constr., square matrix, size 3
A =
 0 3 6
 1 4 7
 2 5 8
*** init B
copy constructor, 3 x 3
```

- man sieht spaltenweise Speicherung von A

Demo zu squareMatrix 3/3

```
20   cout << "*** init C\n";
21   SquareMatrix C = A + B;
22   cout << "C=\n";
23   C.print();
24
25   cout << "*** terminate\n";
26   return 0;
```

- Output:

```
*** init C
constructor, 3 x 3
constructor, 3 x 3
type cast Matrix -> SquareMatrix
destructor, 3 x 3
C=
 0 6 12
 2 8 14
 4 10 16
*** terminate
destructor, square matrix
destructor, 3 x 3
destructor, 3 x 3
destructor, square matrix
destructor, 3 x 3
```


lowerTriangularMatrix.hpp

```
 1 #ifndef LOWERTRIANGULARMATRIX
 2 #define LOWERTRIANGULARMATRIX
 3 #include "squareMatrix.hpp"
 4 #include <cassert>
 5 #include <iostream>
 6
 7 class LowerTriangularMatrix : public SquareMatrix {
 8 private:
 9   double zero;
10   double const_zero;
11
12 public:
13   // constructor, type cast, and destructor
14   LowerTriangularMatrix(int n, double init=0);
15   LowerTriangularMatrix(const Matrix&);
16   ~LowerTriangularMatrix();
17   // read and write entries with matrix access A(j,k)
18   virtual const double& operator()(int j, int k) const;
19   virtual double& operator()(int j, int k);
20 };
21
22 #endif
```

- eine Matrix $A \in \mathbb{R}^{n\times n}$ ist untere Dreiecksmatrix, falls $A_{jk} = 0$ für $k > j$
  - d.h. $A = \begin{pmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$ für $n = 3$
- d.h. untere Dreiecksmatrix ist insb. quadr. Matrix
- muss nur $\frac{n(n+1)}{2} = \sum_{i=1}^{n} i$ Einträge speichern
- zeilenweise Speicherung: $A_{jk} = a_\ell$ mit $\ell = \frac{j(j+1)}{2} + k$
  - muss Matrix-Zugriff ( ) redefinieren

lowerTriangularMatrix.cpp 1/2

```
 1 #include "lowerTriangularMatrix.hpp"
 2 using std::cout;
 3
 4 LowerTriangularMatrix::LowerTriangularMatrix(int n,
 5                                               double init) {
 6   zero = 0;
 7   const_zero = 0;
 8   allocate(n, n, n*(n+1)/2, init);
 9   cout << "constr., lower tri. matrix, size " << n << "\n";
10 }
11
12 LowerTriangularMatrix::LowerTriangularMatrix(
13                                       const Matrix& rhs) {
14   zero = 0;
15   const_zero = 0;
16   int n = rhs.size1();
17   assert (n == rhs.size2());
18   allocate(n, n, n*(n+1)/2, 0);
19   for (int j=0; j<n; ++j) {
20     for (int k=0; k<=j; ++k) {
21       (*this)(j,k) = rhs(j,k);
22     }
23     for (int k=j+1; k<n; ++k) {
24       assert( rhs(j,k) == 0);
25     }
26   }
27 }
28
29 LowerTriangularMatrix::~LowerTriangularMatrix() {
30   cout << "destr., lower triangular matrix\n";
31 }
```

- `private` Member `zero`, `const_zero` haben Wert 0
  - dienen für Zugriff auf $A_{jk} = 0$ für $k > j$
- Type Cast kontrolliert, dass `rhs` $\in \mathbb{R}^{n\times n}$ eine untere Dreiecksmatrix ist
- beachte unterschiedliche ( ) in Zeile 21

lowerTriangularMatrix.cpp 2/2

```
33 const double& LowerTriangularMatrix::operator()(
34                                     int j, int k) const {
35   assert( j>=0 && j<size() );
36   assert( k>=0 && k<size() );
37   if ( j < k ) {
38     return const_zero;
39   }
40   else {
41     const double* coeff = getCoeff();
42     return coeff[j*(j+1)/2+k];
43   }
44 }
45
46 double& LowerTriangularMatrix::operator()(int j, int k) {
47   assert( j>=0 && j<size() );
48   assert( k>=0 && k<size() );
49   if ( j < k ) {
50     zero = 0;
51     return zero;
52   }
53   else {
54     double* coeff = getCoeff();
55     return coeff[j*(j+1)/2+k];
56   }
57 }
```

- Jedes Objekt der Klasse `LowerTriangularMatrix` ist auch Objekt der Klassen `SquareMatrix` und `Matrix`
- Redefinition von Matrix-Zugriff `A(j,k)`
- Garantiere $A_{jk} = 0$ für $k > j$, damit Methoden aus `Matrix` genutzt werden können
  - `const_zero` hat stets Wert 0 (durch Konstruktor)
    - Benutzer kann nicht schreibend zugreifen
  - `zero` wird explizit immer auf 0 gesetzt
    - Benutzer könnte auch schreibend zugreifen

Demo zu lowerTriangularMatrix 1/6

```
 1 #include "matrix.hpp"
 2 #include "squareMatrix.hpp"
 3 #include "lowerTriangularMatrix.hpp"
 4
 5 using std::cout;
 6
 7 int main() {
 8   int n = 3;
 9
10   cout << "*** init A\n";
11   SquareMatrix A(n,1);
12   cout << "A =\n";
13   A.print();
14
15   cout << "*** init B\n";
16   LowerTriangularMatrix B(n);
17   for (int ell=0; ell<n*(n+1)/2; ++ell) {
18     B[ell] = 2;
19   }
20   B(0,n-1) = 10; //*** hat keinen Effekt!
21   cout << "B =\n";
22   B.print();
23
24   cout << "*** init C\n";
25   Matrix C = A + B;
26   cout << "C =\n";
27   C.print();
28
29   cout << "*** init D\n";
30   LowerTriangularMatrix D(n);
31   for (int ell=0; ell<n*(n+1)/2; ++ell) {
32     D[ell] = ell;
33   }
34   D = D + B;
35   cout << "D =\n";
36   D.print();
37
38   cout << "*** terminate\n";
39   return 0;
40 }
```


Demo zu lowerTriangularMatrix 2/6

```
 1 #include "matrix.hpp"
 2 #include "squareMatrix.hpp"
 3 #include "lowerTriangularMatrix.hpp"
 4
 5 using std::cout;
 6
 7 int main() {
 8   int n = 3;
 9
10   cout << "*** init A\n";
11   SquareMatrix A(n,1);
12   cout << "A =\n";
13   A.print();
```

- Output:

```
*** init A
constructor, 3 x 3
constr., square matrix, size 3
A =
 1 1 1
 1 1 1
 1 1 1
```


Demo zu lowerTriangularMatrix 3/6

```
15   cout << "*** init B\n";
16   LowerTriangularMatrix B(n);
17   for (int ell=0; ell<n*(n+1)/2; ++ell) {
18     B[ell] = 2;
19   }
20   B(0,n-1) = 10; //*** hat keinen Effekt!
21   cout << "B =\n";
22   B.print();
```

- Output:

```
*** init B
constructor, empty
constructor, empty square matrix
constr., lower tri. matrix, size 3
B=
 2 0 0
 2 2 0
 2 2 2
```

- Zeile 20 hat keinen Effekt, da in temporäre Variable `zero` geschrieben wird

Demo zu lowerTriangularMatrix 4/6

```
24   cout << "*** init C\n";
25   Matrix C = A + B;
26   cout << "C =\n";
27   C.print();
```

- Output:

```
*** init C
constructor, 3 x 3
C=
 3 1 1
 3 3 1
 3 3 3
```


Demo zu lowerTriangularMatrix 5/6

```
29   cout << "*** init D\n";
30   LowerTriangularMatrix D(n);
31   for (int ell=0; ell<n*(n+1)/2; ++ell) {
32     D[ell] = ell;
33   }
34   D = D + B;
35   cout << "D =\n";
36   D.print();
```

- Output:

```
*** init D
constructor, empty
constructor, empty square matrix
constr., lower tri. matrix, size 3
constructor, 3 x 3
constructor, empty
constructor, empty square matrix
deep copy, 3 x 3
destr., lower triangular matrix
destructor, square matrix
destructor, 3 x 3
destructor, 3 x 3
D =
2 0 0
3 4 0
5 6 7
```


Demo zu lowerTriangularMatrix 6/6

```
38   cout << "*** terminate\n";
39   return 0;
40 }
```

- Output:

```
*** terminate
destr., lower triangular matrix
destructor, square matrix
destructor, 3 x 3
destructor, 3 x 3
destr., lower triangular matrix
destructor, square matrix
destructor, 3 x 3
destructor, square matrix
destructor, 3 x 3
```


symmetricMatrix.hpp

```
1 #ifndef SYMMETRICMATRIX
2 #define SYMMETRICMATRIX
3 #include "squareMatrix.hpp"
4 #include <cassert>
5 #include <iostream>
6
7 class SymmetricMatrix : public SquareMatrix {
8 public:
9   // constructor, type cast, and destructor
10  SymmetricMatrix(int n, double init=0);
11  SymmetricMatrix(const Matrix&);
12  ~SymmetricMatrix();
13  // read and write entries with matrix access A(j,k)
14  virtual const double& operator()(int j, int k) const;
15  virtual double& operator()(int j, int k);
16 };
17
18 #endif
```

- eine Matrix $A = A^T \in \mathbb{R}^{n \times n}$ ist symmetrisch, falls $A_{jk} = A_{kj}$ für $j, k = 0, \ldots, n-1$
  - d.h. $A = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$ für $n = 3$
- d.h. symmetrische Matrix ist insb. quadr. Matrix
- muss nur $\frac{n(n+1)}{2} = \sum_{i=1}^{n} i$ Einträge speichern
- für $j \geq k$ speichere $A_{jk} = a_\ell$ mit $\ell = \frac{j(j+1)}{2} + k$
- für $j < k$ nutze $A_{jk} = A_{kj} = a_\ell$ mit $\ell = \frac{k(k+1)}{2} + j$
- muss Matrix-Zugriff ( ) redefinieren

symmetricMatrix.cpp 1/2

```
 1 #include "symmetricMatrix.hpp"
 2 using std::cout;
 3
 4 SymmetricMatrix::SymmetricMatrix(int n, double init) {
 5   allocate(n, n, n*(n+1)/2, init);
 6   cout << "constructor, symm. matrix, size " << n << "\n";
 7 }
 8
 9 SymmetricMatrix::SymmetricMatrix(const Matrix& rhs) {
10   int n = rhs.size1();
11   assert( n == rhs.size2() );
12   allocate(n, n, n*(n+1)/2, 0);
13   for (int j=0; j<n; ++j) {
14     for (int k=0; k<=j; ++k) {
15       (*this)(j,k) = rhs(j,k);
16     }
17     for (int k=j+1; k<n; ++k) {
18       assert( rhs(j,k) == rhs(k,j) );
19     }
20 }
21
22 SymmetricMatrix::~SymmetricMatrix() {
23   cout << "destructor, symmetric matrix\n";
24 }
```

- Type Cast kontrolliert, dass `rhs` $\in \mathbb{R}^{n\times n}$ symm. ist
- beachte unterschiedliche `( )` in Zeile 18

symmetricMatrix.cpp 2/2

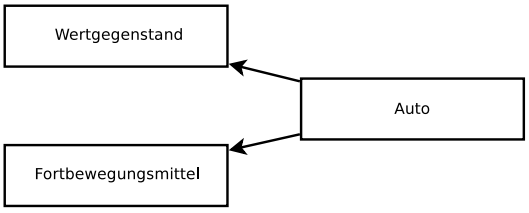
```
26 const double& SymmetricMatrix::operator()(int j, int k)
27                                                     const {
28   assert( j>=0 && j<size() );
29   assert( k>=0 && k<size() );
30   const double* coeff = getCoeff();
31   if ( j>=k ) {
32     return coeff[j*(j+1)/2+k];
33   }
34   else {
35     return coeff[k*(k+1)/2+j];
36   }
37 }
38
39 double& SymmetricMatrix::operator()(int j, int k) {
40   assert( j>=0 && j<size() );
41   assert( k>=0 && k<size() );
42   double* coeff = getCoeff();
43   if ( j>=k ) {
44     return coeff[j*(j+1)/2+k];
45   }
46   else {
47     return coeff[k*(k+1)/2+j];
48   }
49 }
```

- Jedes Objekt der Klasse `SymmetricMatrix` ist auch Objekt der Klassen `SquareMatrix` und `Matrix`
- Redefinition von Matrix-Zugriff `A(j,k)`
  - analog zu `LowerTriangularMatrix`

Mehrfachvererbung

Mehrfachvererbung

- C++ erlaubt Vererbung mit multiplen Basisklassen



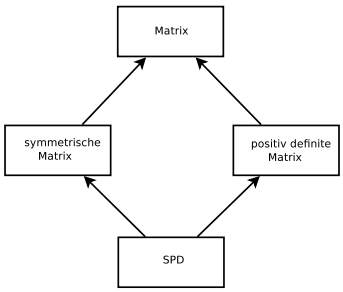
- Syntax:

```
class Auto : public Wertgegenstand, public Fortbew{...}
```

- Vertieft Konzept der Objektorientierung
  - erhöht Wiederverwendbarkeit von Code
- Problem: Mehrdeutigkeiten (nächste Folie)

Diamantvererbung 1/2

- Es könnte eine gemeinsame Oberklasse geben



- Führt zu Mehrdeutigkeit
  - Felder und Methoden sind mehrfach vorhanden
  - Unklar worauf zugegriffen werden soll
  - Speicherverschwendung
  - Schlimmstenfalls: Objekte inkonsistent
- SPD = symmetrisch positiv definite Matrix
  - **symmetrisch:** $A = A^T \in \mathbb{R}^{n \times n}$
  - **positiv definit:** $Ax \cdot x > 0$ für alle $x \in \mathbb{R}^n \backslash \{0\}$
    - äquivalent: alle Eigenwerte sind strikt positiv

Diamantvererbung 2/2

- Klasse `Matrix` hat Member `int n`
- beide abgeleiteten Klassen erben `int n`
- SPD erbt von zwei Klassen
  - SPD hat `int n` doppelt
- Lösung 1: Zugriff mittels vollem Namen
  - `SymmetricMatrix::n` bzw. `PositiveMatrix::n`
- unschön, da Speicher dennoch doppelt
  - unübersichtlich
  - fehleranfällig
  - schlimmstenfalls sogar verschiedene Werte
- Lösung 2: virtuelle Basisklassen
  - `class SymmetricMatrix : virtual public Matrix`
  - `class PositiveMatrix : virtual public Matrix`
- virtuelle Basisklasse wird nur einmal eingebunden

Templates

- Was sind Templates?
- Funktionentemplates
- Klassentemplates
- `template`

Generische Programmierung

- Wieso Umstieg auf höhere Programmiersprache?
  - Mehr Funktionalität (Wiederverwendbarkeit/Wartbarkeit)
  - haben wir bei Vererbung ausgenutzt
- Ziele:
  - möglichst wenig Code selbst schreiben
  - Gemeinsamkeiten wiederverwenden
  - nur Modifikationen implementieren
- Oftmals ähnlicher Code für verschiedene Dinge
- Vererbung bietet sich oft nicht an
  - es liegt nicht immer Ist-Ein-Beziehung vor
- Idee: Code unabhängig vom Datentyp entwickeln
- Führt auf generische Programmierung

Beispiel: Maximum / Quadrieren

- Ziel: Maximum berechnen / quadrieren

```
 1  int max(int a, int b) {
 2    if (a < b)
 3      return b;
 4    else
 5      return a;
 6  }
 7
 8  double max(double a, double b) {
 9    if (a < b)
10      return b;
11    else
12      return a;
13  }
14
15  int square(int a) {
16    return a*a;
17  }
18
19  double square(double a) {
20    return a*a;
21  }
```

- Gleicher Code für viele Probleme
  - Vererbung bietet sich hier nicht an
- Lösung: Templates

Funktionstemplate 1/2

- Funktionalität unabhängig vom Datentyp:

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  template <typename T>
 5  T square(const T& t) {
 6    return t*t;
 7  }
 8
 9  int main() {
10    cout << square<double>(4.7) << endl;
11    cout << square(4.7) << endl;
12  }
```

- Ausgabe:
```
22.09
22.09
```
- t ist Platzhalter für Daten vom Typ T
  - Für alle DT die Multiplikation $*$ bereitstellen z.B. Quadrate von Matrizen
- Bei Aufruf DT in spitze Klammern (Z. 10)
- kann auch weggelassen werden (Z. 11)
- `template <typename T> rtype funName(varlist)`
  - Einleitung durch Schlüsselwort `template`
  - Templateparameter in spitzen Klammern
    - allgemeiner Datentyp: `typename`
    - alternative Syntax: `class`
    - T ist Name des Parameters (beliebiger Name)
  - `rtype` ist Typ des Rückgabewerts von `funName`
  - Rückgabe u Eingabe können vom Typ T sein
    - auch Referenz oder Pointer auf T möglich

Funktionstemplate 2/2

- Was passiert eigentlich bei folgendem Code?

```
 1  #include <iostream>
 2  using namespace std;
 3
 4  template <typename T>
 5  T square(const T& t) {
 6    return t*t;
 7  }
 8
 9  int main() {
10    int x = 2;
11    double y = 4.7;
12    cout << square(x) << endl;
13    cout << square(y) << endl;
14  }
```

- Compiler erkennt dass Fkt `square` einmal für Typ `int` und einmal für Typ `double` benötigt wird
- Compiler erzeugt ("programmiert") und kompiliert anhand von dieser Information, zwei (!) Funktionen mit der Signatur
  - `double square(double)`
  - `int square(int)`
- D.h. Funktion `square` wird automatisch anhand des Templates (= Vorlage) generiert
  - also nur für die Typen die wirklich benötigt

Klassentemplate 1/3

- Auch allgemeine Klassen möglich
  - haben wir bereits verwendet (`vector`)
- Automatische Speicherverwaltung bei Pointern:

```
 1  template <typename T>
 2  class simple_ptr {
 3  public:
 4    simple_ptr(T* ptr);
 5    ~simple_ptr();
 6    T& operator*();
 7    T* operator->();
 8  private:
 9    //Die Klasse soll nicht kopierbar sein.
10    simple_ptr(const simple_ptr&);
11    simple_ptr& operator=(const simple_ptr&);
12
13    T* m_ptr;
14  };
```

- Idee: Speicher automatisch freigeben
  - verhindert Speicherlecks
- Für Pointer von beliebigem Typ
  - Das nennt man Smartpointer
- Um zu verhindern, dass Obj der Kl kopiert werden kann, schreibt man Kopierkonstruktor und Zuweisungsoperator in den `private` Bereich

Klassentemplate 2/3

- Implementierung von Klassenmethoden
  - Beispiel: Konstruktor

```
1  template <typename T>
2  simple_ptr<T>::simple_ptr(T* ptr) : m_ptr(ptr)
3  {}
```

- Syntax:
  - voranstellen von `template <typename T>`
  - Wichtig: `<T>` nach Klassenname
    (Methode für Klasse `simple_ptr<T>`)
- tatsächliche Implementierung wie gehabt

Klassentemplate 3/3

- Restliche Klassenmethoden

```
 1  template <typename T>
 2  simple_ptr<T>::~simple_ptr() {
 3    if(m_ptr)
 4      delete m_ptr;
 5    m_ptr = NULL;
 6  }
 7
 8  template <typename T>
 9  T& simple_ptr<T>::operator*() {
10    return *m_ptr;
11  }
12
13  template <typename T>
14  T* simple_ptr<T>::operator->() {
15    return m_ptr;
16  }
17
18  int main() {
19    simple_ptr<string>
20        some_stringptr(new string("Hallo"));
21    cout << *some_stringptr << endl;
22  }
```

- Ausgabe: `Hallo`
- Destruktor gibt Speicher wieder frei
  (Garbage Collection)
- Operatoren `*` und `->` überladen
  - für Funktionalität von Pointer

Private Methoden

- Probleme bei Kopien von Smartpointern:

```
 1  int main() {
 2    simple_ptr<string> p(new string("blub"));
 3    p->size();
 4    {
 5      simple_ptr<string> q = p;
 6      q->size();
 7      // Destruktor von q wird aufgerufen
 8    }
 9    p->size();
10  }
```

- Pointer wird kopiert (Z. 5)
  - Hier: Zuweisung in (Z. 5) nicht möglich
    (Methode ist `private`)
- Speicher wird freigegeben (Z. 8)
- Problem: Speicher von p auch freigegeben
- Sonst Lösung: Kopien zählen
  - Speicher nur freigeben wenn kein Zugriff mehr
    (wird hier nicht vertieft)

Vektor-Template 1/2

```
 1  #include <string>
 2  #include <vector>
 3  using namespace std;
 4
 5  struct Eintrag {
 6    string name;
 7  };
 8
 9  int main() {
10    vector<Eintrag> telbuch(2);
11    telbuch[0].name = "Peter Pan";
12    telbuch[1].name = "Wolverine";
13  }
```

- Zeile 10: Vektor der Länge 2 wird angelegt.
  Einträge im Vektor sind vom Datentyp `Eintrag`
- `Vektoren` sind C++ Standardcontainer
  - man kann beliebige Datentypen verwenden
  - dienen zum Verwalten von Datenmengen
- Verwendung: `vector<type> name(size);`
  - Achtung, nicht verwechseln:
    `vector<Eintrag> buch(1000);` *1000 Einträge*
    `vector<Eintrag> buecher[1000];` *1000 Vektoren*
- Zugriff auf j-tes Element wie bei Arrays
  - `name[j]` (Z. 12–13)

Vektor-Template 2/2

```
 1  #include <iostream>
 2  #include <string>
 3  #include <vector>
 4  using namespace std;
 5
 6  struct Eintrag {
 7    string name;
 8  };
 9
10  int main() {
11    vector<Eintrag> telbuch(1);
12    telbuch[0].name = "Peter Pan";
13    cout << "size: " << telbuch.size() << endl;
14    telbuch.resize(telbuch.size()+4);
15    cout << "size: " << telbuch.size() << endl;
16  }
```

- Ausgabe: `size: 1`
  `size: 5`
- Speicher dyn veränderbar mittels `resize(newsize)`
- Weitere hilfreiche Funktionen:
  - `size()` Gibt Länge des Vek zurück
  - `push_back(val)` Fügt `val` am Ende ein (vgl Liste!)
  - `pop_back()` Gibt letztes Ele des Vek zurück und löscht es aus Vek
  - uvm.

Weitere Standardcontainer

- `list` (verkettete Listen)
- `queue`
- `stack`
- `deque`
- `set`
- `multiset`
- `map`
- `multimap`

- Weitere C++ Bibliotheken
  - Boost Library: Große Sammlung an Bib.
  - Teile davon bald in C++ Standardbib.
  - `http://www.boost.org`
  - `http://http://www.highscore.de/cpp/boost/`