Informatik II

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2010

Inhalt

Klassen

- Komplexe Zahlen
- Friend-Funktionen
- Ein- und Ausgabe-Operatoren
- Mehrdimensionale Felder

Zusammenfassung

Vorlesung 6. Klassen, Teil 2

Klassen

- Komplexe Zahlen
- Friend-Funktionen
- Ein- und Ausgabe-Operatoren
- Mehrdimensionale Felder

Zusammenfassung

6. Klassen, Teil 2

Vorige Vorlesung

- Abstraktion und Objekte
- Klassendefinition und -deklaration
- Member-Funktionen
- Selbstreferenz und Operatoren
- Beispiele

Heutige Vorlesung

- Komplexe Zahlen als Klasse
- Friend-Funktionen
- Selbstdefinierte Ein- und Ausgabe-Operatoren
- Klassen mit dynamischen mehrdimensionalen Feldern

Lernziele dieser Vorlesung

- Kenntnis der Konstruktion vollständiger Klassen
- Verständnis des Zugriffs auf Elemente von Klassen aus anderen Klassen (friend-Funktionen)
- Kenntnis der Ein- und Ausgabe-Operatoren für selbstdefinierte Klassen
- Verständnis der dynamischen Speicherverwaltung in Klassen

Beispiel Komplexe Zahlen complex.h

```
class complex {
private:
  double re, im; // Real-, Imaginaerteil
public:
  complex () // Konstruktor
    {
      re = 0.0; im = 0.0;
    };
  void input ();
  void output ();
  complex operator + (complex z2);
  complex operator * (complex z2);
  complex operator / (complex z2);
  complex operator += (complex z2);
  complex operator = (complex z2);
};
```

Beispiel Komplexe Zahlen complex.cpp...

```
#include <iostream>
#include "complex.h"
using namespace std;
// Eingabe von komplexen Zahlen
void complex::input ()
{
  cin >> re >> im;
}

// Ausgabe von komplexen Zahlen
void complex::output ()
{
  cout << re << "␣+i*" << im;
}
```

...Beispiel Komplexe Zahlen complex.cpp...

```
// Operatorfunktion +, Addition komplexer Zahlen
complex complex::operator + (complex z2)
{
  complex erg;
  erg.re=re+z2.re;
  erg.im=im+z2.im;
  return erg;
}

// Operatorfunktion *, Multiplikation komplexer Zahlen
complex complex::operator * (complex z2)
{
  complex erg;
  erg.re=re*z2.re-im*z2.im;
  erg.im=re*z2.im+z2.re*im;
  return erg;
}
```

...Beispiel Komplexe Zahlen complex.cpp

```
// Operatorfunktion +=
complex complex::operator += (complex z2)
{
    re=re+z2.re;
    im=im+z2.im;
    return *this;
}

// Operatorfunktion =, Zuweisung komplexer Zahlen
complex complex::operator = (complex z2)
{
    re=z2.re;
    im=z2.im;
    return *this;
}
```

Beispiel Komplexe Zahlen complexmain.cpp

```
#include <iostream>
#include "complex.h"

int main ()
{
  complex a, b, c;

  cout << "komplexe␣Zahl?␣";
  a.input ();
  cout << "komplexe␣Zahl?␣";
  b.input ();

  a.output ();
  cout << "␣/␣";
  b.output ();
  cout << "␣=␣";
  c=a/b;
  c.output ();
  cout << endl;
  a.output ();
  cout << "␣+␣";
  b.output ();
  cout << "␣=␣";
  a+=b; // a hat jetzt den
        // wert von a+b
  a.output ();
  cout << endl;

  return(0);
}
```

Friend-Funktion...

friend *datentyp funktionsbezeicher*(*parameterdeklarationen*);

- Nicht-Member-Funktionen können auf private Daten einer Klasse nicht zugreifen.
- Der Zugriff auf private Daten kann in der Klassendefinition für bestimmte Nicht-Member-Funktionen zugelassen werden
- Durch Friend-Funktionen hat man für eine Klasse die Kontrolle darüber, welche externen Funktionen Zugriff auf die Daten einer Klasse haben
- Sinnvoll bei Operatoren, die Objekte verschiedener Klassen verarbeiten.

. . .Friend-Funktion. . .

Matrix und Vektor

- Zwei Klassen: Matrix und Vektor
- Multiplikation von Matrix und Vektor

```
Class Matrix;
class Vektor {
  float v[4]; // 4-dimensional
  // hier ohne Formalparameter
  friend Vektor operator * (const Matrix &m,
                            const Vektor &v);
};
class Matrix {
  Vektor v[4]; // Matrix = 4 Vektoren
  friend Vektor operator * (const Matrix &m,
                            const Vektor &v);
};
```

...Friend-Funktion

Matrix und Vektor

```
Vektor operator * (const Matrix &m, const Vektor &v)
{
  Vektor r; // Ergebnis
  for (int i=0; i<4; i++) { // r[i]=m[i]*v
    r.v[i] = 0;
    for (int j=0; j<4; j++)
      r.v[i] += m-v[i].v[j]*v.v[j];
  }
  return (r);
}
```

Benutzerdefinierte Ein- und Ausgabe-Operatoren

- Für Klassen können spezielle Ein- und Ausgabe-Operatoren definiert werden.
- Analog zu den vordefinierten Operatoren >> und << für Standard-Datentypen zu den Objekten `cin` und `cout` werden die Operatoren >> und << für benutzerdefinierte Datenobjekte durch Operator-Overloading definiert.
- `cin` und `cout` sind Objekte der Klassen `istream` und `ostream`.

Definitionsschema für Ein- und Ausgabe-Operatoren

- Damit Anweisungen der Form `cin >> x` und `cout << x` möglich werden, muss nach folgendem Schema überladen werden.
-

```
ostream &operator<<(ostream &os, const X &x)
{
  // Anweisungen der Form: os << x.daten;
  return os;
}

istream &operator>>(istream &is, X &x)
{
  // Anweisungen der Form: is >> x.daten;
  return is;
}
```

- Wird auf private Daten zugegriffen, müssen `operator<<()` und `operator>>()` `friend`-Funktionen sein.

Beispiel komplexe Zahlen complex.h

```
using namespace std;
class complex {
private:
  double re, im; // Real-, Imaginaerteil
public:
  complex () // Konstruktor
  {
    re = 0.0; im = 0.0;
  };
  friend istream &operator >> (istream &is, complex &z);
  friend ostream &operator << (ostream &os,
                               const complex &z);
  friend complex operator + (complex z1, complex z2);
  friend complex operator * (complex z1, complex z2);
  friend complex operator / (complex z1, complex z2);
  complex operator += (complex z2);
  complex operator = (complex z2);
};
```

Beispiel komplexe Zahlen complex.cpp...

```
// Datei complex.cc
#include <iostream>
#include "complex.h"
using namespace std;
// Einagbe von komplexen Zahlen
istream &operator>>(istream &is, complex &z)
{
  char c[3];
  is >> z.re >> c[0] >> c[1] >> c[2];
  if (!(c[0] == '+' & &c[1] == 'i' & &c[2] == '*')) {
    cerr << "Input␣error␣complex" << endl;
    exit(1);
  } else {
    is >> z.im;
  }
  return is;
}
```

...Beispiel komplexe Zahlen complex.cpp

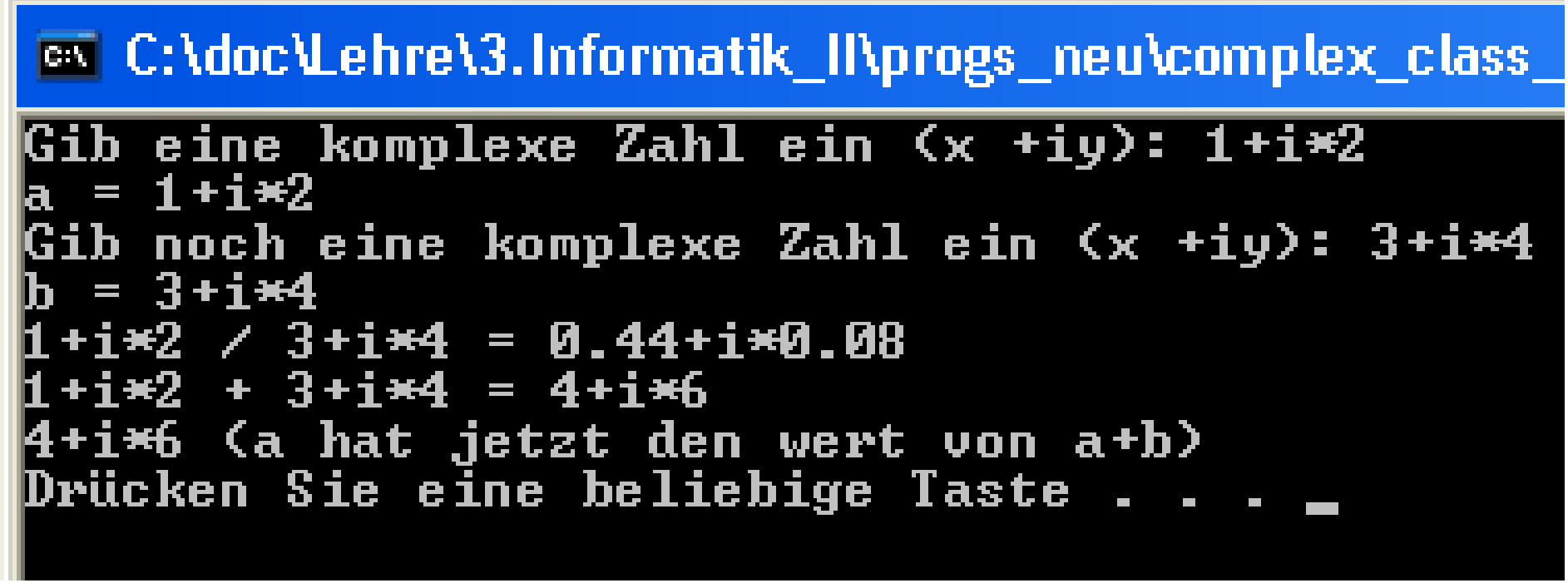
```
// Ausgabe von komplexen Zahlen
ostream &operator<<(ostream &os, const complex &z)
{
  os << z.re << "+i*" << z.im;
  return os;
}
```

Komplexe Zahlen (Benutzung)...

```
#include <iostream>
#include "cplcl.h"
using namespace std;
int main()
{
  complex a, b, c;
  cout << "Gib␣eine␣komplexe␣Zahl␣ein␣(x␣+i*y):␣";
  cin >> a;
  cout << "a␣=␣" << a << endl;
  cout << "Gib␣noch␣eine␣komplexe␣Zahl␣ein␣(x␣+i*y):␣";
  cin >> b;
  cout << "b␣=␣" << b << endl;
  cout << a << "␣/␣" << b << "␣=␣" << a/b << endl;
  cout << a << "␣+␣" << b << "␣=␣" << a+b << endl;
  a+=b; // a hat jetzt den wert von a+b
  cout << a << "␣(a␣hat␣jetzt␣den␣wert␣von␣a+b)" << endl;
  return (0);
}
```

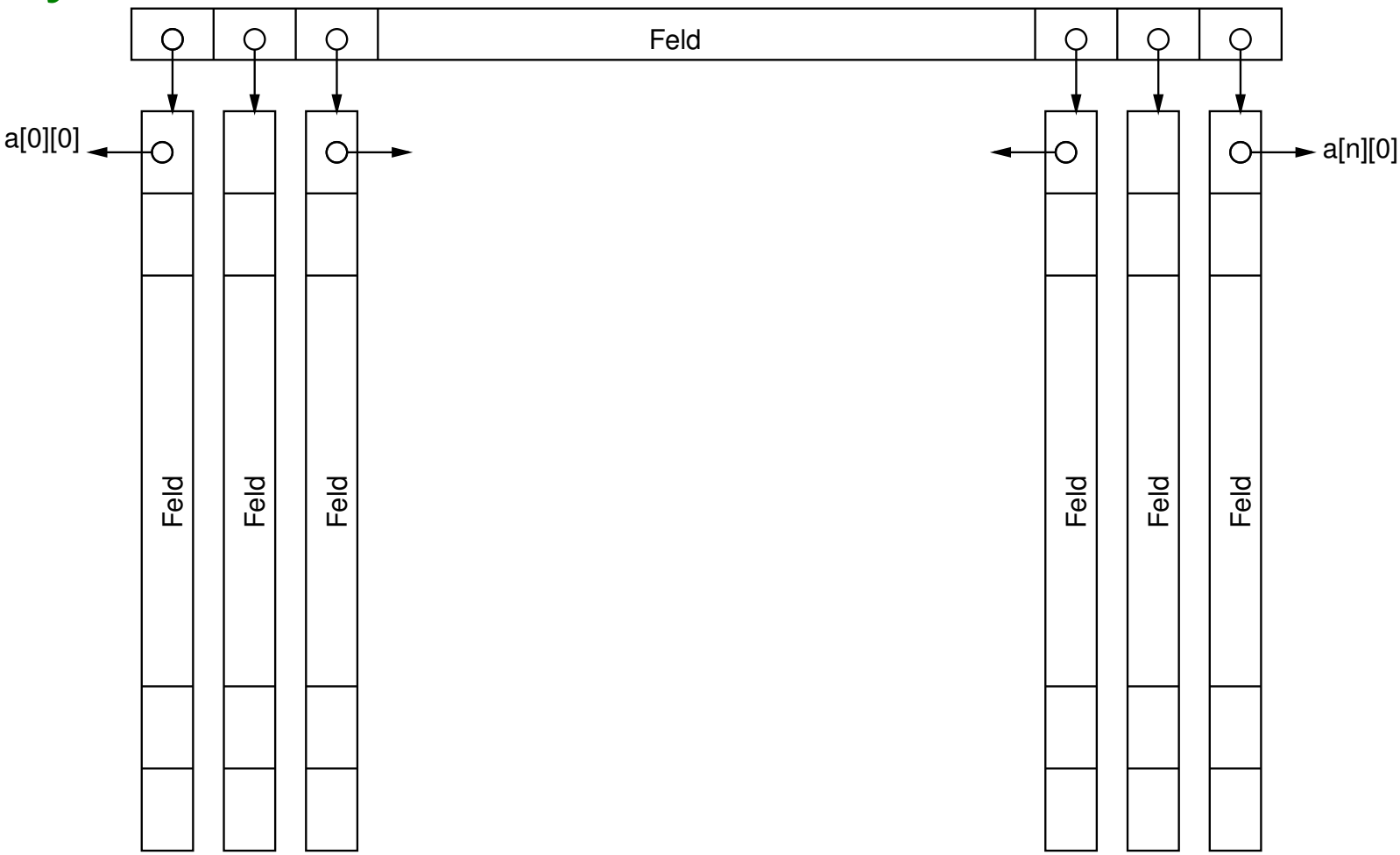
. . .Komplexe Zahlen (Benutzung)

Programmausgabe



Dynamische mehrdimensionale Felder...

Dynamische Datenstruktur Matrix, `double **elem`



...Dynamische mehrdimensionale Felder...

- In Klassenkonstruktoren mit `new` eine Datenstruktur erzeugen
- Beispiel: Matrix

```
const int DIM = 3;
class matrix
{
private:
  const int zeilen, spalten;
  double **elem;
public:
  matrix();
  matrix(int dim);
  matrix(int zeilen, int spalten);
};
int main ()
{
  matrix a; // eine 3x3-Matrix
  matrix b (5); // eine 5x5-Matrix
  matrix c (3, 4); // eine 3x4-Matrix
}
```

...Dynamische mehrdimensionale Felder

- Erzeugung eines Feldes von Zeigern auf ein Feld, z. B. Basistyp `double **elem`
- Im Konstruktor

```
double **elem;
elem = new double*[zeilen];
for (i = 0; i < zeilen; ++i) {
  elem[i] = new double[spalten];
}
```

- Im Destruktor

```
for (i = 0; i < zeilen; ++i) {
  delete [] elem[i];
}
delete [] elem;
```

Dynamische mehrdimensionale Felder (Feld-Operator)

- Eleganter Zugriff auf Feldelemente durch Überschreiben des Operators ()
-
```
double &operator () (int i, int j)
{
  // Matrix-Index von 1 bis zeilen bzw. spalten,
  // Feld von 0 bis zeilen - 1 bzw. spalten - 1
  return(elem[i - 1][j - 1]);
}
```
- Überschreiben ermöglicht Zugriff auf Feldelemente mit Ausdruck `m(3, 4)`, z. B.

```
cout << m(3, 4); // m.elem[2][3]
```

Dynamische mehrdimensionale Felder (gesamte Klasse)

```
const int DIM = 3;
class matrix
{
private:
  void init ();
  const int zeilen, spalten;
  double **elem;
public:
  matrix ();
  matrix (int dim);
  matrix (int zeilen, int spalten);
  ~matrix ();
  matrix (const matrix &old_matrix);
  matrix &operator = (const matrix &m);
  double &operator () (int i, int j)};
  void input ();
  void print ();
};
```

Dynamische mehrdimensionale Felder (Initialisierung)

```
void matrix::init () // Ausnullen
{
  int i, j;
  for (i = 0; i < zeilen; ++i) {
    for (j = 0; j < spalten; ++j) {
      elem[i][j] = 0.0;
    }
  }
}
```

Dynamische mehrdimensionale Felder (Standard-Konstruktor)

```
const int DIM = 3;
// ...
matrix::matrix() : zeilen(DIM), spalten(DIM)
{
  int i;
  elem = new double*[DIM];
  for (i = 0; i < DIM; ++i) {
    elem[i] = new double[DIM];
  }
  init ();
}
```

Dynamische mehrdimensionale Felder ($m \times m$-Konstruktor)

```
matrix::matrix(int dim) : zeilen(dim), spalten(dim)
{
  int i;
  elem = new double*[dim];
  for (i = 0; i < dim; ++i) {
    elem[i] = new double[dim];
  }
  init ();
}
```

Dynamische mehrdimensionale Felder ($m \times n$-Konstruktor)

```
matrix::matrix (int z, int s) :
  zeilen(z), spalten(s)
{
  int i;
  elem = new double*[zeilen];
  for (i = 0; i < zeilen; ++i) {
    elem[i] = new double[spalten];
  }
}
```

Dynamische Speicherverwaltung

- Bei dynamischer Speicherverwaltung sind die automatisch erzeugten Copy-Konstruktoren, Destruktoren und Zuweisungsoperatoren meist unbrauchbar
- Bei dynamischer Speicherverwaltung sind die relevanten Member-Variablen Zeiger
- Der Copy-Konstruktor erzeugt einen neuen Zeiger, der auf die Daten der zu kopierenden Instanz zeigt! Änderungen in der Kopie bewirken Änderungen im Original
- Bei Feldern gibt der Destruktor nur den Speicher für das erste Element frei
- Der Zuweisungsoperator weist nur die Zeiger zu

Dynamische mehrdimensionale Felder (Copy-Konstruktor)

```
matrix::matrix(const matrix &old_matrix) :
  zeilen(old_matrix.zeilen), spalten(old_matrix.spalten)
{
  int i, j;
  elem = new double*[zeilen];
  for (i = 0; i < zeilen; ++i) {
    elem[i] = new double[spalten];
  }
  for (i = 0; i < zeilen; ++i) {
    for (j = 0; j < spalten; ++j) {
      elem[i][j] = old_matrix.elem[i][j];
    }
  }
}
```

Dynamische mehrdimensionale Felder (Destruktor)

```
matrix::~matrix ()
{
  int i;
  for (i = 0; i < zeilen; ++i) {
    delete [] elem[i];
  }
  delete [] elem;
}
```

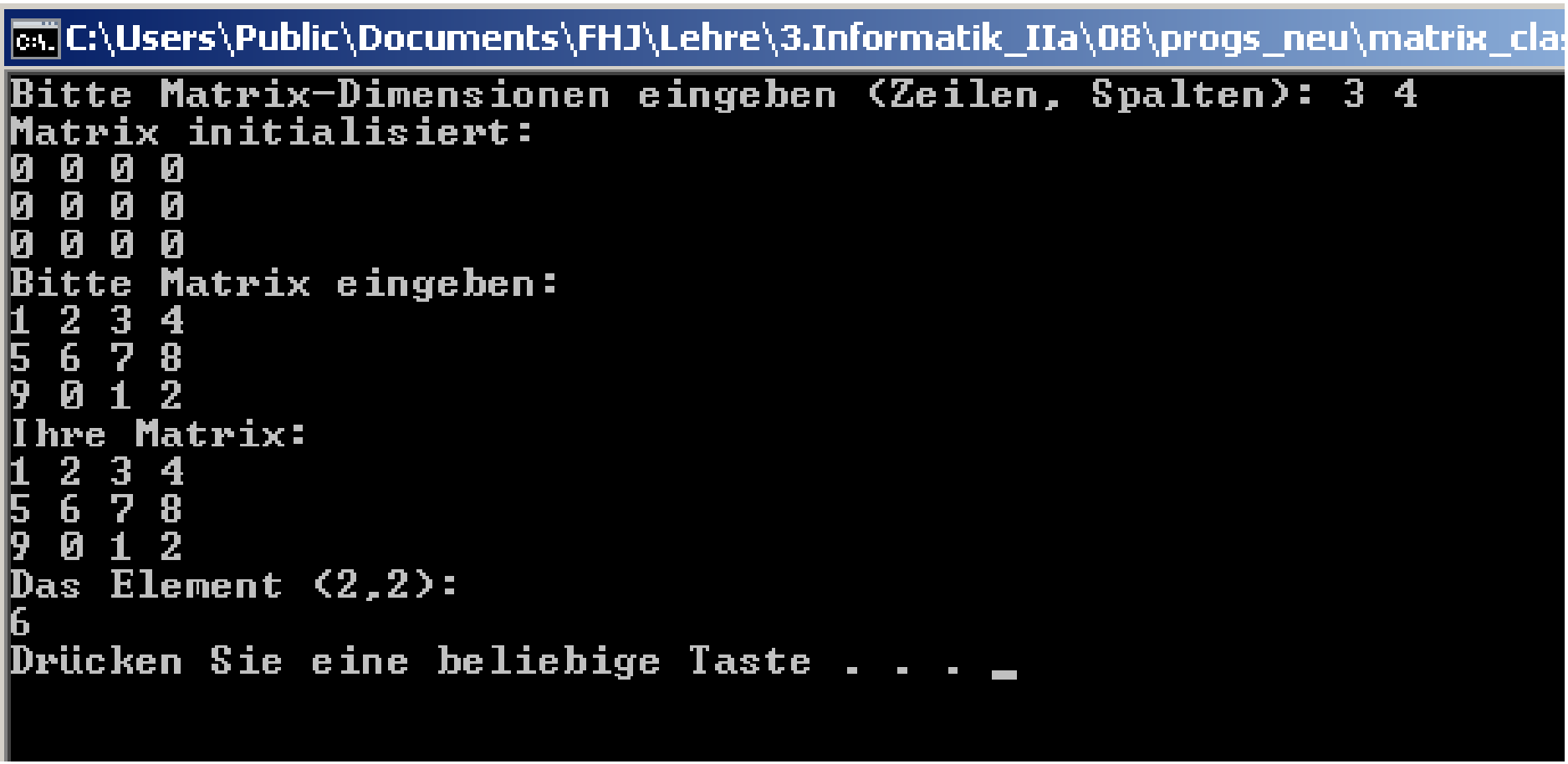
Dynamische mehrdimensionale Felder (Zuweisung)

```
matrix matrix::matrix operator = (const matrix &org_matrix)
{
  if ((zeilen != org_matrix.zeilen)
      || (spalten != org_matrix.spalten)) {
    cerr << "Matrizen haben ";
    cerr << "verschiedenes Format. ";
    cerr << "Keine Zuweisung" << endl;
    return(*this);
  }
  for (int i = 0; i < zeilen; ++i) {
    for (int j = 0; j < spalten; ++j) {
      data[i][j] = org_matrix.data[i][j];
    }
  }
  return(*this);
}
```

Dynamische mehrdimensionale Felder (Benutzung)

```
int main ()
{
  int zeilen, spalten;
  cout << "Bitte␣Matrix-Dimensionen␣eingeben␣"
       << "(Zeilen,␣Spalten):␣";
  cin >> zeilen >> spalten;
  matrix a (zeilen, spalten);
  cout << "Matrix␣initialisiert:␣" << endl;
  a.print ();
  cout << "Bitte␣Matrix␣eingeben:␣" << endl;
  a.input ();
  cout << "Ihre␣Matrix:␣" << endl;
  a.print ();
  cout << "Das␣Element␣(2,2):␣" << endl;
  cout << a(2, 2) << endl;
  return (0);
}
```

Programmausgabe



Klassen, Teil 2

- Für selbstdefinierte Klassen können spezielle Ein- und Ausgabe-Operatoren (>>, <<) definiert werden.
- Für Ein- und Ausgabe-Operatoren müssen friend-Funktionen definiert werden, falls auf private Daten zugegriffen werden soll.
- Dynamische Datenstrukturen in Klassen werden am Besten in den Konstruktoren der Klasse verwaltet.