

Einführung in die Programmierung

Wintersemester 2012/13

Prof. Dr. Günter Rudolph

Lehrstuhl für Algorithm Engineering

Fakultät für Informatik

TU Dortmund

Inhalt

- Matrixmultiplikation (Schablonen / Ausnahmen)
- Klassenhierarchien

MATRIX Multiplikation (für quadratische Matrizen)

```
class Matrix {  
private:  
    double **fElem;  →  
    unsigned int fDim;  
public:  
    Matrix(unsigned int aDim);  
    Matrix(const Matrix& aMat);  
  
    unsigned int Dim() { return fDim; }  
    void Set(unsigned int aRow, unsigned int aCol, double aVal);  
    double Get(unsigned int aRow, unsigned int aCol);  
    void Mult(const Matrix& aMat);  
    void Print();  
  
    ~Matrix();  
};
```

Konstruktor

Aufgaben: Speicher allokieren & Datenstruktur initialisieren

```
Matrix::Matrix(unsigned int aDim) : fDim(aDim) {  
    if (fDim == 0) throw "matrix of dimension 0";  
    fElem = new double* [fDim];  
    if (fElem == nullptr) throw "memory exceeded";  
    for (unsigned int i = 0; i < fDim; i++) {  
        fElem[i] = new double[fDim];  
        if (fElem[i] == nullptr) throw "memory exceeded";  
    }  
    for (unsigned int i = 0; i < fDim; i++)  
        for (unsigned int j = 0; j < fDim; j++)  
            fElem[i][j] = 0;  
}
```

Speicherallokation

initiale
Belegung

Kopierkonstruktor

```
Matrix::Matrix(const Matrix& aMat) : fDim(aMat.fDim) {  
    if (fDim == 0) throw "matrix of dimension 0";  
    fElem = new double* [fDim];  
    if (fElem == nullptr) throw "memory exceeded";  
    for (unsigned int i = 0; i < fDim; i++) {  
        fElem[i] = new double[fDim];  
        if (fElem[i] == nullptr) throw "memory exceeded";  
    }  
    for (unsigned int i = 0; i < fDim; i++)  
        for (unsigned int j = 0; j < fDim; j++)  
            fElem[i][j] = aMat.fElem[i][j];  
}
```

Speicherallokation

kopieren

Destruktor

```
Matrix::~~Matrix() {  
    for (unsigned int i = 0; i < fDim; i++)  
        delete[] fElem[i];  
    delete[] fElem;  
}
```

MATRIX Multiplikation

Methoden

```
void Matrix::Set(unsigned int aRow, unsigned int aCol,
                 double aVal)
{
    if (aRow >= fDim || aCol >= fDim) throw "index overflow";
    fElem[aRow][aCol] = aVal;
}

double Matrix::Get(unsigned int aRow, unsigned int aCol) {
    if (aRow >= fDim || aCol >= fDim) throw "index overflow";
    return fElem[aRow][aCol];
}
```

Indexbereichsprüfung!

MATRIX Multiplikation

A und B sind quadratische Matrizen der Dimension n.

C = A · B ist dann:

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

```
void Matrix::Mult(const Matrix& aMat) {
    if (aMat.fDim != fDim) throw "incompatible dimensions";
    Matrix tMat(*this);
    for (unsigned int i = 0; i < fDim; i++)
        for (unsigned int j = 0; j < fDim; j++) {
            double sum = 0.0;
            for (unsigned int k = 0; k < fDim; k++)
                sum += tMat.Get(i, k) * aMat.fElem[k][j];
            fElem[i][j] = sum;
        }
}
```

MATRIX Multiplikation

Nächste Schritte

1. Ausnahmen vom Typ `char*` ersetzen durch unterscheidbare Fehlerklassen:
z.B. `class MatrixError` als Basisklasse,
ableiten: `MatrixIndexOverflow`, `MatrixDimensionMismatch`, ...
2. Umwandeln in Schablone: `template <class T> class Matrix { ...`
also nicht nur Multiplikation für `double`,
sondern auch `int`, `Complex`, `Rational`, ...

Code nach Fehlermeldungen durchforsten ...

Konstruktoren:

`throw "matrix of dimension 0";` \longrightarrow `MatrixZeroDimension`
`throw "memory exceeded";` \longrightarrow `MatrixMemoryExceeded`

Get / Set:

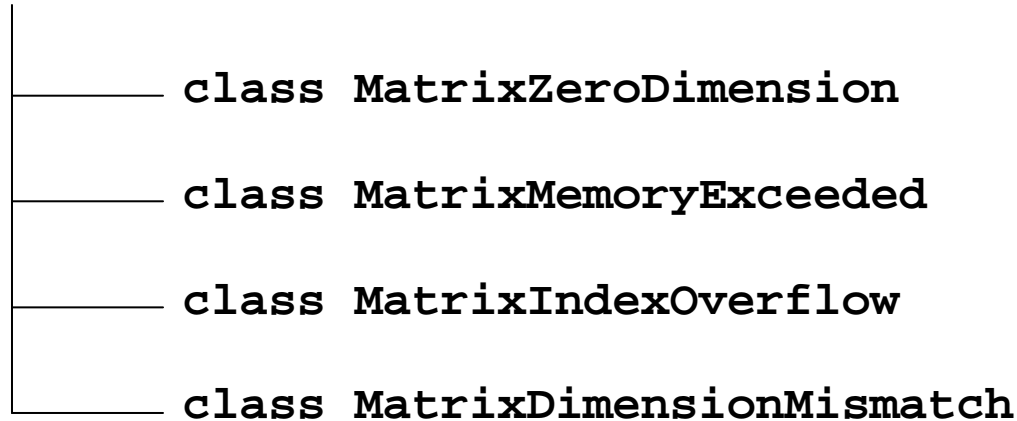
`throw "index overflow";` \longrightarrow `MatrixIndexOverflow`

Mult:

`throw "incompatible dimensions";` \longrightarrow `MatrixDimensionMismatch`

Klassenhierarchie für Ausnahmen

```
class MatrixError
```



Minimale Lösung in C++:

```
class MatrixError {};  
  
class MatrixZeroDimension : public MatrixError() {};  
class MatrixMemoryExceeded : public MatrixError() {};  
class MatrixIndexOverflow : public MatrixError() {};  
class MatrixDimensionMismatch : public MatrixError() {};
```

Schablone MATRIX Multiplikation (für quadratische Matrizen)

```
typedef unsigned int uint;

template <class T> class Matrix {
private:
    T **fElem;
    uint fDim;

public:
    Matrix(uint aDim);
    Matrix(const Matrix& aMat);

    uint Dim() { return fDim; }
    void Set(uint aRow, uint aCol, T aVal);
    T Get(uint aRow, uint aCol);
    void Mult(const Matrix& aMat);
    void Print();

    ~Matrix();
};
```

ursprünglichen Typ
der Nutzdaten
ersetzen durch
generischen Typ **T**

Schablone MATRIX Multiplikation mit vernünftiger Ausnahmebehandlung

Konstruktor

```
template <class T>
Matrix::Matrix(uint aDim) : fDim(aDim) {
    if (fDim == 0) throw MatrixZeroDimension();
    fElem = new T* [fDim];
    if (fElem == nullptr) throw MatrixMemoryExceeded();
    for (uint i = 0; i < fDim; i++) {
        fElem[i] = new T[fDim];
        if (fElem[i] == nullptr)
            throw MatrixMemoryExceeded();
    }
    for (uint i = 0; i < fDim; i++)
        for (uint j = 0; j < fDim; j++)
            fElem[i][j] = 0;
}
```

Speicherallokation

initiale
Belegung

Schablone MATRIX Multiplikation mit vernünftiger Ausnahmebehandlung

Kopierkonstruktor

```
template <class T>
Matrix::Matrix(const Matrix& aMat) : fDim(aMat.fDim) {
    if (fDim == 0) throw MatrixZeroDimension();
    fElem = new T* [fDim];
    if (fElem == nullptr) throw MatrixMemoryExceeded();
    for (uint i = 0; i < fDim; i++) {
        fElem[i] = new T[fDim];
        if (fElem[i] == nullptr)
            throw MatrixMemoryExceeded();
    }
    for (uint i = 0; i < fDim; i++)
        for (uint j = 0; j < fDim; j++)
            fElem[i][j] = aMat.fElem[i][j];
}
```

Speicherallokation

kopieren

Destruktor

```
template <class T> Matrix::~~Matrix() {
    for (uint i = 0; i < fDim; i++) delete[] fElem[i];
    delete[] fElem;
}
```

Methoden

```
template <class T>
void Matrix::Set(uint aRow, uint aCol, T aVal) {
    if (aRow >= fDim || aCol >= fDim)
        throw MatrixIndexOverflow();
    fElem[aRow][aCol] = aVal;
}
template class <T>
T Matrix::Get(uint aRow, uint aCol) {
    if (aRow >= fDim || aCol >= fDim)
        throw MatrixIndexOverflow();
    return fElem[aRow][aCol];
}
```

Schablone MATRIX Multiplikation mit vernünftiger Ausnahmebehandlung

```
template <class T>
void Matrix::Mult(const Matrix& aMat) {
    if (aMat.fDim != fDim)
        throw MatrixIncompatibleDimensions();

    Matrix tMat(*this);
    for (uint i = 0; i < fDim; i++)
        for (uint j = 0; j < fDim; j++) {
            T sum = 0.0;
            for (uint k = 0; k < fDim; k++)
                sum += tMat.Get(i, k) * aMat.fElem[k][j];
            fElem[i][j] = sum;
        }
}
```

```
class A {  
private:  
    int a;  
public:  
    A(int a);  
    int Get_a();  
    void Show(bool cr = true);  
};
```



```
class B : public A {  
private:  
    int b;  
public:  
    B(int a, int b);  
    int Get_b();  
    void Show(bool cr = true);  
};
```



```
class C : public B {  
private:  
    int c;  
public:  
    C(int a, int b, int c);  
    int Get_c();  
    void Show(bool cr = true);  
};
```

Klassenhierarchie

Erzwingen der Verwendung
des Konstruktors der Oberklasse:

Attribute durch **private** schützen,
Zugriff nur durch **public** Methoden!


```
A::A(int aa) : a(aa) {}
int A::Get_a() { return a; }
void A::Show(bool cr) {
    cout << a << ' ';
    if (cr) cout << endl;
}
B::B(int aa, int bb) : A(aa), b(bb) {}
int B::Get_b() { return b; }
void B::Show(bool cr) {
    A::Show(false);
    cout << b << ' ';
    if (cr) cout << endl;
}
C::C(int aa, int bb, int cc) : B(aa, bb), c(cc) {}
int C::Get_c() { return c; }
void C::Show(bool cr) {
    B::Show(false);
    cout << c << ' ';
    if (cr) cout << endl;
}
```

a(aa)



Test

```
int main(void) {  
  
    A a(1);  
    B b(1,2);  
    C c(1,2,3);  
  
    a.Show();  
    b.Show();  
    c.Show();  
  
    C c2(2*c.Get_a(), 2*c.Get_b(), 2*c.Get_c());  
    c2.Show();  
  
    return 0;  
}
```

Ausgabe



```
C:\ d:\Code\...  
1  
1 2  
1 2 3  
2 4 6  
-
```