Johannes Gutenberg-Universität Mainz
Institut für Informatik
Einführung in die Softwareentwicklung
Prof. Dr. Michael Wand
Dr. Christian Ali Mehmeti-Göpel
Visual Computing Group

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Einführung in die Softwareentwicklung

ÜBUNGSBLATT 06: Low-Level Programmierung in C++

23. Juli 2025

Über dieses Übungsblatt

Auf diesem Übungsblatt machen wir etwas ganz anderes. Wir schauen uns an, wie man mit Hilfe der Programmiersprache C++ schnellen Code schreiben kann. Dies ist insbesondere in der Kombination mit Python nützlich, da hier die Ausführungsgeschwindigkeit oft für kritische Teile von Systemen („die inneren Schleifen“) nicht ausreicht.

Optional ist eine Vergleichsimplementation in Python mit Laufzeitmessung. Auch eine Integration der C++-Bibliothek in Python-Code via „`pybind11`“ können Sie optional ausprobieren (beides ohne Bewertung, letzteres ist eher empfohlen für Studierende mit Vorkenntnissen).

Abgabe: (Abgabe 13. SW)

Für dieses Übungsblatt ist nur eine Woche Bearbeitungszeit vorgesehen (die Punktzahl ist entsprechend geringer). Das Übungsblatt muss entsprechend bis zum **29. Juni 2025, 23:59h** in LMS abgegeben werden. Laden Sie dazu den Quellcode aller Aufgaben hoch. Die Ergebnisse müssen in den Übungen zwischen dem 30. Juni-4. Juli 2025 vorgestellt werden.

Aufgabe 1: Lineare Algebra in C++ (20+5+15+10 = 50 Punkte)

In dieser Aufgabe sollen Klassen für Matrizen (Dimension frei einstellbar) programmiert werden, sowie eine Matrix-Matrix Multiplikation. Am Ende messen wir, wie effizient unser Code ist.

Hinweis: Falls Sie noch wenig Erfahrung mit dem Programmieren in C++ haben, beachten Sie die „Anleitung zur Strukturierung von C++-Programmen" am Ende des Übungsblattes

a) Schreiben Sie eine Klasse „Matrix", die eine quadratische $n \times n$ Matrix, bestehend aus doppeltgenauen Fließkommazahlen (Typ „double") speichert. Hierbei handelt es sich (im Wesentlichen) um ein zweidimensionales Array. Quadratisch bedeutet dabei, dass Breite und Höhe der Matrix (des 2D-Arrays) gleich sind, nämlich dem Wert n entsprechen, der eine ganze Zahl mit Wert 1 oder größer annehmen kann. Die Zahl n soll für jede einzelne Instanz (jedes Objekt) der Matrixklasse einzeln einstellbar sein.

Die Klasse sollte folgendes leisten:

- Erzeugen von Instanzen mit gewünschter Größe `n` mittels geeignetem Konstruktor.
- Lesezugriff auf Einträge m_{xy} der Matrix mit einer geeignete Getter-Methode, z.B. einer Member-Funktion „**`double get(int x, int y)`**". Optional können Sie auch überladene Operatoren wie in den Vorlesungsfolien angerissen verwenden.
- Schreibzugriff auf Einträge m_{xy} der Matrix mit einer geeignete Setter-Methode, z.B. einer Member-Funktion „**`void set(int x, int y, double value)`**". Optional können Sie auch überladene Operatoren wie in den Vorlesungsfolien angerissen verwenden.
- Zugriff auf die Größe n der Matrix via z.B. einer Member-Funktion „**`int getDimension()`**". Es ist nicht nötig, eine Änderung der Größe zu unterstützen/erlauben.
- Speicher sollte (soweit nötig, s.u.) richtig angelegt und freigegeben werden.

Tipp: Arrays in C++ erstellt man am einfachsten mit der generischen Klasse `vector<…>`, hier also mit `vector<double>`. Diese Arrays sind immer eindimensional, aber mit Indizes `x*Breite + y` kann man zweidimensionale Arrays mit vorgegebener Breite und Höhe leicht simulieren. Nimmt man einen Wert (nicht als Zeiger) vom Typ `vector<…>` als Member in eine Klasse oder einen „Struct" auf, so wird auch der Speicher automatisch freigegeben; es ist also im Destruktor nichts mehr zu tun.

Tipp: Kurzreferenz zu `vector<…>`:

```
vector<int> test = vector<int>(42); // Vektor mit 42 ints anlegen (Werte nicht initialisiert)
test.resize(23); // Länge auf 23 ändern (ggf. Abschneiden)
cout << test.size(); // Funktion liefert Länge zurück (hier 23); „cout <<" ist nur Konsolenausgabe
test[7] = 9; // Schreibzugriff
cout << test[7]; // Lesezugriff

// Der Destruktor von vector gibt Speicher automatisch freizugeben
// Dies geschieht am Ende von {…} und auch dann, wenn der vector selbst in einer
// anderen Klasse/struct enthalten ist (natürlich nicht bei Zeigern vector<>*)
```

Sie können natürlich auch mit „alten" C/C++-Arrays (rohe Zeiger auf Speicher und `new[]`/`delete[]`) arbeiten (weniger Komfortabel, aber vielleicht einfacher zu durchschauen).

b) Schreiben Sie eine Funktion (oder Member-Funktion), die Matrizen auf der Konsole ausgibt. Die Formatierung ist dabei nicht so wichtig, man sollte das Ergebnis aber (in den Übugen) gut lesen können. Hinweis: Einen Ausdruck „`expr`“ gibt man in C++ via `cout << expr;` aus. `"\n"` bedeutet Zeilenumbruch (wie in Python).

c) Schreiben Sie eine Funktion, die es erlaubt zwei Matrizen miteinander zu multiplizieren. Dabei ist die Matrix-Matrix-Multiplikation wie folgt definiert:

- Gegeben sind zwei $n \times n$ Matrizen `A` und `B`. Sollten die Größen nicht übereinstimmen, so sollte Ihre Funktion eine Fehlermeldung auf der Konsole ausgeben und die weitere Berechnung abbrechen.
- Wenn man die Einträge in Spalte `x` und Zeile `y` in einer Matrix `A` als `A[x][y]` bezeichnet, dann sollte für das Produkt der Matrizen `P = A*B` gelten, das

  ```
  P[x][y] = A[1][y]*B[x][1] + A[2][y]*B[x][2] + … + A[n][y]*B[x][n]
  ```

  Man multipliziert also für den Eintrag an der Stelle x,y die Einträge der y-ten Zeile von A mit der x-ten Spalte von B.

Testen Sie Ihre Routine mit den folgenden Testmatrizen:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{pmatrix} = \begin{pmatrix} 84 & 90 & 96 \\ 201 & 216 & 231 \\ 318 & 342 & 366 \end{pmatrix}$$

Sollten Sie hier andere Werte erhalten, stimmt etwas nicht…

d) Schreiben Sie eine Funktion, die zwei große Matrix mit 1024 x 1024 Einträgen erstellt und diese auf feste Werte setzt (z.B. alles auf null/eins oder auf Zufallswerte via geeigneter Zufallsfunktion aus der Standardbilbiothek). Messen Sie die Laufzeit für die Matrix-Matrix-Multiplikation (einfach direkt mit einer Stoppuhr per Hand; ggf. können Sie den Code mehrfach mit einer Schleife wiederholen lassen, damit es langsam genug wird). Wieviele „Megaflops“ (Fließkommaoperationen pro Sekunde) erreicht Ihre Implementation? Vergleichen Sie dies mit den theoretischen Werten für Ihre CPU.

Aufgabe 2: Lineare Algebra in Python (freiwillig, unbewertet)

Wiederholen Sie die Implementation in Python und vergleichen Sie die Laufzeiten. Wieviel schneller (hoffentlich :-)) ist Ihre C++-Implementation? Vergleichen Sie auch mit der Matrix-Matrix-Multiplikation aus dem bekannten „NumPy“ Packet – dieses (ebenfalls in C/C++ geschriebene) Paket ist vermutlich nochmal deutlich schneller (falls nicht: Gratulation!).

Aufgabe 3: Für Fortgeschrittene – Integration von C++ in Python (freiwillig, unbewertet)

Nutzen Sie (z.B.) das Python-Packet „`pybind11`“ um die Matrix-Klasse, die in C++ implementiert wurde, in Python als Python-Objekt verfügbar zu machen. Ein solcher Ansatz ist ungemein nützlich, um kritischen Code für Python-Projekte ausreichend schnell zu machen.

Informationen zum Paket finden Sie unter: https://pybind11.readthedocs.io

Anhang: Hinweise zur Struktur von C++-Programmen

Wenn Sie dieses Übungsblatt möglichst einfach lösen möchten, können Sie einfach alles (die Matrix-Klasse, mit Code für Memberfunktionen direkt in der Klasse, sowie weitere Funktionen) in eine `*.cpp`-Datei (z.B. Blatt06.cpp) schreiben. Das Ganze sähe dann in etwa so aus:

Datei `Blatt06.cpp`:

```
#include <iostream>  // für Konsolenausgabe via cout
#include <vector>    // falls benötigt
#include <string>    // falls benötigt (hier wahrscheinlich nicht)
using namespace std; // damit man nicht immer std::vector schreiben muss

struct MyClass { // struct = class mit default “public”
   int myField;
   vector<string> myArray;
   MyClass(int blubb) {
      myField = blubb;
   }
   void print_blah() {
      cout << myField << "\n"; // mit Zeilenumbruch
   }
};

void test_func() {
   cout << "Test\n";
}

int main() { // Das Hauptprogramm tut nicht viel sinnvolles...
   MyClass instance(42); // Achtung C++-Compiler sind „one-pass“
   test_func();          // Man kann nur auf Symbole zugreifen, die vorher
   return 0;             // im Text (auch via #include) deklariert wurden.
}
```

Übersetzen z.B. via `g++ Blatt06.cpp -o Blatt06.exe` (GNU C++-Compiler)

Wenn man „vernünftige“ C++-Programme schreibt, teilt man den Code aber in Module mit Schnittstellen und Implementation auf. Dazu würde man für jedes Modul (z.B. jede Klasse) eine eigene „Header“-Datei mit Endung `*.h` und eine Implementationsdatei mit Endung `*.cpp` anlegen. Die CPP-Datei bindet dabei den eigenen Header via „`#include`“ ein (mit Anführungszeichen statt spitzen Klammern; die sind für Standardbibliotheken). Am Ende überesetzt man alle *.cpp Dateien getrennt und linkt sie zusammen (am einfachsten mit einer IDE wie QtCreator umzusetzen).

Im Folgenden nochmal das gleiche Beispiel in „richtiger“ Aufteilung (für dieses Aufgabenblatt optional!):

Datei `MyClass.h`: *Nur Schnittstellen*

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

struct MyClass {  // struct = class mit default “public”
   int myField;
   vector<string> myArray;

   MyClass(int blubb);  // Keine Implementation!
   void print_blah();   // Keine Implementation!
};

void test_func(); // Keine Implementation!
```

Datei `MyClass.cpp`: *Implementation dazu*

```
#include "MyClass.h"

MyClass::MyClass(int blubb) { // Implementation einer Methode von MyClass
   myField = blubb;
}

void MyClass::print_blah() { // Implementation einer Methode von MyClass
   cout << myField << "\n";
};

void test_func() { // Implementation einer globalen Funktion ohne Klasse
   cout << "Test\n";
}
```

Datei `Blatt06.cpp`: *Implementation des Hauptprogramms (ohne Schnittstellen, da nur Hauptprogram – hier ist, anders als bei allen anderen Modulen, kein Header üblich)*

```
#include "MyClass.h"

int main() {
   MyClass instance(42);
   test_func();
   return 0;
}
```

Alles übersetzen z.B. via `g++ MyClass.cpp Blatt06.cpp -o Blatt06.exe` (GNU C++-Compiler)