Name:

Hon. Prof. Dr.-Ing. Hartmut Helmke
Ostfalia
Hochschule für angewandte Wissenschaften
Fakultät für Informatik

Matrikelnummer:	*Punktzahl:*
Ergebnis:	

Klausur im WS 2016/17:

Embedded Toolchain

Bitte Aufgabenblätter mit abgeben !
Austausch von Hilfsmitteln mit Kommilitonen ist **nicht** erlaubt !
Anwesenheit von Handys, Smartphones etc. bei Teilnehmern im Hörsaal nicht erlaubt.
Sie sind vor Beginn am Dozentenpult abzugeben!

Bitte notieren Sie auf **allen** Blättern Ihren Namen bzw. Ihre Matrikelnummer.
Hinweis: In den folgenden Programmfragmenten wird die globale Variable *datei* verwendet. Hierfür kann der Einfachheit halber die Variable *cout* angenommen werden. Die Variable *datei* diente lediglich bei der Klausurerstellung dem Zweck der Ausgabeumlenkung.

Meistens kann die Lösung direkt auf dem Aufgabenblatt notiert werden. Extrablätter bitte mit Aufgabennummer, Namen und/oder Matrikelnummer versehen.
Aufrufe von `malloc` oder `new` sollen jeweils den nächsten zusammenhängenden ausreichend großen freien Speicherbereich ab den Adressen beginnend bei 7400 liefern. Die Speicherbelegung soll hier auf Stack und Heap jeweils von den tieferen zu den höheren Adressen verlaufen.
Auf eine absolut korrekte Anzahl der Blanks und Zeilenumbrüche braucht bei der Ausgabe nicht geachtet zu werden. Dafür werden keine Punkte abgezogen.

Geplante Punktevergabe

Punktziel	Sonderpunkte	erreicht
Tests: (max. 30)		
A1: ca. 35 P.		
A2: ca. 10 P.		
A3: ca. 20 P.		
A4: ca. 15 P.		
Summe ca. 80 P.		

Aufgabe 1 : Instanzen

ca. 35 Punkte

Gegeben sei die folgende Deklaration der Klasse `Matrix`.

```
class Matrix {
public:
   Matrix(int z=0, int sp=0) :
      zAnz(z), spAnz(sp) {
      datei << " +M " << z*sp;
      Init(z*sp, &daten);
   }
   ~Matrix()       {
      datei << " -M " << zAnz*spAnz;
      Freigeben(zAnz*spAnz, &daten);
   }
   TYP GetAt(int z, int sp){
      return daten[z * spAnz + sp];
   }
   void SetAt(int z, int sp, TYP w){
      daten[z * spAnz + sp] = w;
   }
private:
   int zAnz;
   int spAnz;
   TYP* daten;
};
```

Für TYP gilt:

```
typedef double TYP;
```

a (4)) Implementieren Sie zunächst die im Konstruktor und Destruktor verwendeten Funktionen `Init` und `Freigeben` entsprechend der folgenden Spezifikationen:

```
// Init reserviert Heap-Speicher fuer anz Elemente
// vom Typ TYP. Nach Aufruf
// der Funktion zeigt der Referenzparameter z
// auf diesen Speicherbereich im Heap.
// Falls anz <=0 ist, wird z mit nullptr belegt.
```

(*— Lösung für Init hier notieren ——*)

b (3))

```
// Der Inhalt von z verweist auf anz Instanzen vom Typ
// TYP. Dieser Heap-Speicherbereich wird hier wieder
// freigegeben. Anschliessend enthaelt der uebergebene
// Zeiger den nullptr. anz wird aber
// ansonsten in der Funktion nicht benoetigt.
```

(*— Lösung für Freigeben hier notieren ——*)

c (4)) Welche Ausgabe in `datei` liefert der `funk1`-Aufruf?

```
void funk1(){
   Matrix* pc = new Matrix(40, 5);
   Matrix m1(11, 2);
   Matrix m2(4, 1);
}
```

(*— Lösung hier notieren ——*)

d (3)) Welche Ausgabe in `datei` liefert der `funk2`-Aufruf?

```
void funk2(){
   Matrix arr[3];
   datei << " Ende im Gelaende ";
}
```

(*— Lösung hier notieren ——*)

e (3)) An welcher Stelle enthält die Funktion `funk4` (aufgrund der Schnittstelle von `Matrix`) einen Syntaxfehler und wie sollte dieser am besten korrigiert werden?

```
bool funk4(const Matrix& m2){
   if (m2.GetAt(0, 0) == m2.GetAt(1, 1)) {
      return true;
   }
   return false;
}
```

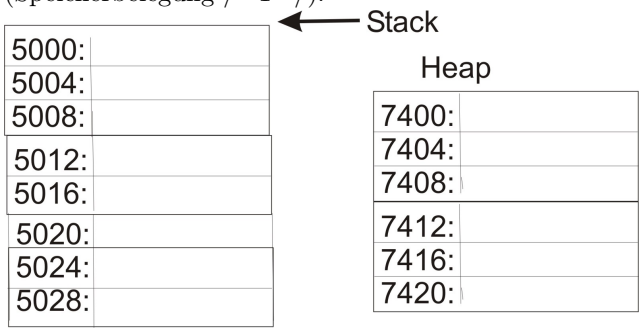
(*— Lösung hier notieren ——*)

f (10)) Veranschaulichen Sie die Speicherbelegung des Aufrufs der Funktion `PrintMat` durch `funk3` zu den Zeitpunkten /*1*/ und /*2*/ in zwei verschiedenen Grafiken. Erklären Sie daran, warum das Verhalten der Funktion `funk3` zumindest undefiniert ist. Hilfe: Sie haben keinen Kopier-Konstruktor!

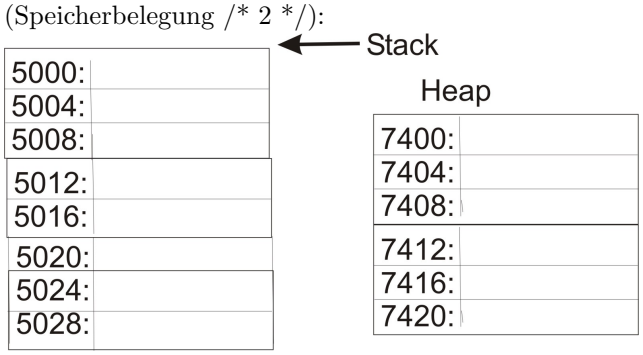
```
void PrintMat(Matrix m) {
   m.SetAt(0, 0, 16); /*1*/
   datei << "m[0,0] ist" << m.GetAt(0, 0);
}
void funk3(){
   Matrix m1(1, 3);
   PrintMat(m1);
   int j = 14; /*2*/
   datei << j;
}
```

Gehen Sie davon aus, dass `double` 8 Bytes `int` 4 Byte und Zeiger jeweils 4 Bytes im Speicher belegen.

(Speicherbelegung /* 1 */):



(Speicherbelegung /* 2 */):



(*— Erklärung ——*)

g (4)) Implementieren Sie den Kopier-Konstruktor für die Klasse `Matrx`.
(*— Lösung hier notieren ——*)

h (2)) Warum muss man im Kopier-Konstruktor nicht vor Eigenzuweisung schützen (this != &right)?
(*— Lösung hier notieren ——*)

i (2)) Warum darf der Parameter des Kopier-Konstruktors nicht als Werteparameter übergeben werden?
(*— Lösung hier notieren ——*)

Aufgabe 2 : Embedded

ca. 10 Punkte

Gegeben ist der folgende Signal-Handler.

```
#include <cmath>
#define MAX 100
Matrix* gp_matrix = nullptr;
TYP g_biggest;
/* gp_matrix verweist auf eine 100 * 100 Matrix,
die extern einmal angelegt und immer wieder
beschrieben wird. Der Signal (Interrupt)-Handler
SetBiggest bestimmt den Wert des
groessten Elements (absolut gesehen) und legt
diesen in der globalen Variablen g_biggest ab.
*/
void SetBiggest(int signum){
   TYP big = fabs(gp_matrix->GetAt(0, 0));
   for (int i = 0; i < MAX; ++i){
      for (int j = 0; j < MAX; ++j){
         if (big < fabs(gp_matrix->GetAt(i, j)) ) {
            big = fabs(gp_matrix->GetAt(i, j));
         }
      }
   }
   g_biggest = big;
}
```

Gehen Sie im Folgenden davon aus, dass *nur* der Aufruf der Methode `GetAt` einen signifikanten Zeitanteil benötigt, da die Matrix in einem speziellen (langsamen) Speicherbereich abgelegt wurde. Pro Aufruf benötigt diese Methode jeweils eine Mikrosekunde, d.h. $\frac{1}{1000}$ Millisekunde. Die Ausführungszeit des gesamten Rests (die Schleifen an sich, `if`-Statements, fabs, Zuweisung an `big` und `g_biggest` etc.) dürfen Sie großzügig mit einer Nanosekunde abschätzen. Für die ca. (100*100=) 10.000 Aufrufe *des Rest* werden somit insgesamt weniger als 0,05 Millisekunden benötigt.

a (6)) Die Anforderung ist, dass die Funktion innerhalb von 15 Millisekunden ein Ergebnis liefern muss. Ist diese Funktion im Sinne der Definition aus der Vorlesung echtzeitfähig?
(*— Lösungen hier notieren ——*)

b (4)) Wie könnte man den Signalhandler im Sinne von Laufzeiteffizienz weiter verbessern?

(*— Lösungen hier notieren ——*)

Aufgabe 3 : Speicher einsparen

ca. 20 Punkte

Das folgende Listing zeigt nochmals den bisherigen Stand der Implementierung der Klasse `Matrix`:

```
class Matrix {
private:
   int zAnz;
   int spAnz;
   TYP* daten;
public:
   Matrix(int z=0, int sp=0) :zAnz(z), spAnz(sp){
      datei << " +M " << z*sp;
      Init(zAnz * spAnz, &daten);
   }
   ~Matrix()      {
      datei << " -M " << zAnz*spAnz;
      Freigeben(zAnz * spAnz, &daten);
   }
   TYP GetAt(int z, int sp){
      return daten[z * spAnz + sp];
   }
   void SetAt(int z, int sp, TYP w){
      daten[z * spAnz + sp] = w;
   }
```

Wir erweitern die Klassen-Schnittstelle nun um Kopier-Konstruktor, Zuweisungsoperator, sowie als `friend`-vereinbarte Funktionen zur Matrizenmultiplikation:

```
   friend Matrix operator*(Matrix m1, Matrix m2);
   friend void Mult(const Matrix& m1,
      const Matrix& m2, Matrix& res);
   Matrix(const Matrix& m2);
   Matrix& operator=(const Matrix& m2);
```

Die Implementierung des Multiplikations-Operators sieht wie folgt aus:

```
Matrix operator*(Matrix m1, Matrix m2){
   Matrix res(m1.zAnz, m2.spAnz);
   for (int i = 0; i < m1.zAnz; i++) {
      for (int j = 0; j < m2.spAnz; j++) {
         res.SetAt(i, j, static_cast<TYP>(0));
         for (int k = 0; k < m1.spAnz; k++) {
            res.SetAt(i, j, res.GetAt(i, j) +
               m1.GetAt(i, k) * m2.GetAt(k, j));
         }
      }
   }
   datei << "\nVor return in op*\n";
   return res;
}
```

Diese Implementierung ist in vielerlei Hinsicht verbesserungsfähig. Das wollen wir an folgendem Code-Fragment verdeutlichen:

```
void testMultOp(){
   Matrix m1(10, 100);
   Matrix m2(100, 5);
   Matrix res(0, 0);
   datei << "\nVor Mult\n";
   res = m1 * m2;
   datei << "\nNach Mult\n";
}
```

Die Ausgabe von `+M -M` etc. ist im Folgenden nicht so wichtig. Sie kann jedoch bei der Beantwortung der folgenden Fragen helfen. Die Antworten auf die Fragen sind das Wichtige! Es geht im Folgenden ausschließlich um den Code, der zwischen den Ausgaben `Vor Mult` und `Nach Mult` ausgeführt wird.

a (5)) Wieviele Matrix-Objekte werden (zwischen der Ausgabe `Vor Mult` und `Nach Mult`) erzeugt und wie viele zerstört?
(*— Anzahl plus Begründung ——*)

b (3)) Wie oft wird neuer Heap-Speicher angefordert (Wir gehen davon aus, dass die nicht dargestellte Implementierung von Kopier-Konstruktor und Zuweisungsoperator den Heap-Speicher auch korrekt unter Nutzung von `Init` und `Freigeben` verwalten)?
(*— Anzahl plus Begründung ——*)

c (4)) Nach der Ausgabe von `Vor Mult` stehen Ihrem Prozess noch `11.000` Bytes auf dem Heap-Speicher zur Verfügung. Reicht dieser Platz aus, wenn Sie davon ausgehen, dass nicht benötigter Heap-Speicherplatz sofort wieder (im Destruktor und Zuweisungsoperator) freigegeben wird? **Achtung:** Eine Instanz von `TYP` belegt `8` Bytes.
(*— Anzahl plus Begründung ——*)

d (8)) Welche Möglichkeiten zur Anpassung der Schnittstelle der Klasse `Matrix` gibt es, sodass die Ausführung der (unveränderten) Funktion `testMultOp` sehr viel weniger Heapspeicher benötigt und auch die maximal benötigte Byteanzahl an Heap-Speicher deutlich kleiner wird (Diskutieren Sie die Verbesserungsmöglichkeiten an, Sie müssen Sie nicht implementieren)?

(*— Extrablatt verwenden oder hier notieren ——*)

Aufgabe 4 : Parameter und Matrizenmultiplikation

ca. 15 Punkte

Das folgende Listing zeigt eine Anwendung einer Matrizenmultiplikation:

```
volatile bool gb_ABS_isOn = true;
void ABS_Application(){
   // 4 Matrizen mit Heapspeicher initialisieren
   Matrix m1(m1zAnz, m1spAnz);
   Matrix m2(m1spAnz, m2spAnz);
   Matrix m3(m2spAnz, m3spAnz);
   Matrix m4(m3spAnz, m4spAnz);
   while (gb_ABS_isOn){
      // Matrizen immer wieder mit Werten fuellen
      GetMatrixValue(m1, m2, m3, m4);
      // Inhalt des Matrizen-Produkts pruefen
      if (StartABS1(m1, m2, m3, m4)){
         datei << "\nUse ABS now";
      }
   }
}
```

Es werden zunächst 4 Matrizen erzeugt In einer Schleife werden die Matrizen immer wieder in `GetMatrixValue` mit neuen Werten belegt. Hierbei wird kein Heapspeicher neu belegt oder freigegeben. In `StartABS1` werden die 4 Matrizen miteinander multipliziert und dann wird geprüft, ob die Summe der Elemente in der Hauptdiagonalen der resultierenden quadratischen Produktmatrix größer null ist. Die Anwendung bricht ab, sobald das ABS durch Setzen der Variablen `gb_ABS_isOn` auf `false` abgeschaltet wird.

a (2)) Weder in `GetMatrixValue` noch in `StartABS1` noch in den darin aufgerufenen Funktionen wird die globale Variable `gb_ABS_isOn` verändert. Trotzdem liegt keine Todschleife vor. Finden Sie hierfür eine Erklärung.

(*— Lösung hier notieren ——*)

b (2)) Warum muss `gb_ABS_isOn` deshalb als `volatile` gekennzeichnet werden?

(*— Lösung hier notieren ——*)

c (11)) Wir sehen nun die Implementierung von `SummeDiagonale`

```
// Summe der Hauptdiagonalenelemente
TYP Matrix::SummeDiagonale()const {
   TYP sum = static_cast<TYP>(0);
   if (zAnz == spAnz){
      for (int i = 0; i < zAnz; ++i){
         sum += GetAt(i, i);
      }
   }
   return sum;
}
```

und von der Methode `StartABS1`:

```
bool StartABS1(const Matrix& m1,
   const Matrix& m2,
   const Matrix& m3, const Matrix& m4) {
   Matrix res = m1 * m2 * m3 * m4;
   return res.SummeDiagonale() > 0;
}
```

Die 4 Eingangsmatrizen werden miteinander multipliziert. Die Schnittstelle von `Matrix` soll im Folgenden gegenüber Ihren Verbesserungen aus der vorherigen Aufgabe nicht verändert werden, aber die Funktion `StartABS1` muss noch weiter verbessert werden. Hierzu darf sowohl ihre Schnittstelle als auch ihr Funktionsrumpf noch angepasst werden, sodass wir eine noch effizientere Nutzung des Heapspeichers erhalten.
Wir nehmen folgende Matrizendimensionen an (wobei die Verbesserung unabhängig davon erfolgen kann):

```
// Matrix m1: 3*200
const int m1zAnz = 3;
const int m1spAnz = 200;
// Matrix m2: 200*10
const int m2spAnz = 10;
// Matrix m3: 10*20
const int m3spAnz = 20;
// Matrix m4: 20*3
const int m4spAnz = 3;
// m1*m2*m3*m4: 3 * 3
```

Ohne unsere Optimierung aus der vorherigen Aufga-

benstellung würden pro Durchlauf durch die `while`-Schleife 10 Aufrufe von `Init`, d.h. 10 Anforderungen von Heap-Speicher, und entsprechend viele Freigaben erfolgen.
Außerdem würden maximal `23.360` Bytes auf dem Heap belegt.
Mit unseren (hoffentlich auch Ihren) Optimierungen aus der vorherigen Aufgabenstellung kämen wir auf 3 Aufrufe von `Init` und `Freigeben` sowie auf die zusätzliche maximale Heap-Speicheranforderung von `792` Bytes.
Leider ist das immer noch zu viel. Verbessern Sie daher die Schnittstelle und die Implementierung von `StartABS1` weiter. Passen Sie dann auch die aufrufende Funktion `ABS_Application` an. Sie dürfen hierbei abkürzen, solange immer noch Ihre Verbesserung deutlich erkennbar bleibt.
Hinweis: Nutzen Sie statt des Operators `operator*` die friend-Funktion `Mult` von Matrix:

```
void Mult(const Matrix& m1,
   const Matrix& m2, Matrix& res) {
   for (int i = 0; i < m1.zAnz; i++) {
      for (int j = 0; j < m2.spAnz; j++) {
         res.SetAt(i, j, static_cast<TYP>(0));
         for (int k = 0; k < m1.spAnz; k++) {
            res.SetAt(i, j, res.GetAt(i, j) +
               m1.GetAt(i, k) * m2.GetAt(k, j));
         }
      }
   }
   datei << "\nVor return in Mult";
} // Mult
```

(*— Extrablatt verwenden und hier notieren ——*)