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 — Lösungen —

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	Im einzelnen	Pkte
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:

```
void Init(int anz, TYP** z){
   *z = (anz > 0) ? new TYP[anz] : nullptr;

   if (anz > 0){
      g_usedHeapSize += anz * sizeof(TYP);
      ++ g_initCallCnt;
      if (g_usedHeapSize > g_maxUsedHeap) {
         g_maxUsedHeap = g_usedHeapSize;
      }
   }
}
```

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:

```
void Freigeben(int anz, TYP** z){
   delete[] *z;
   *z = nullptr;
   g_usedHeapSize -= anz * sizeof(TYP);
}
```

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:

```
+M 200 +M 22 +M 4 -M 4 -M 22
```

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

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

Lösung:

```
 +M 0 +M 0 +M 0 Ende im Gelaende
-M 0 -M 0 -M 0
```

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:
`m2` wird als konstante Referenz übergeben. Die aufgerufene Methode `GetAt` ist allerdings nicht als `const` vereinbart. Dieses sollte geändert werden, denn `GetAt` verändert keine Attribute der Klasse `Matrix`.

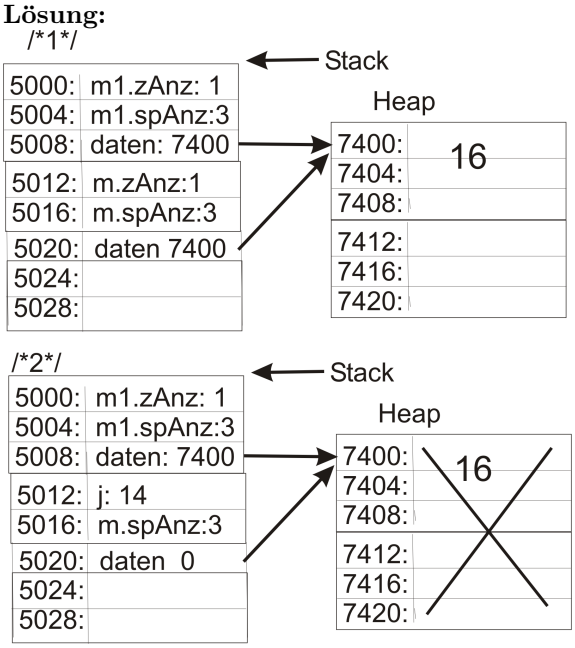
Je nach `TYP` kann die Nutzung des Vergleichsoperators `==` natürlich zu Rundungsproblemen führen, z.B. bei `TYP` gleich `double`. Dann liegt allerdings kein Syntaxfehler vor.

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.

Lösung:



Es gibt keinen Kopier-Konstruktor. Deshalb wird Matrix Byte für Byte kopiert. Somit zeigen zwei Zeiger auf den gleichen Speicherbereich im Heap. Sobald die Funktion `PrintMat` verlassen wird, wird im Destruktor durch Aufruf von `Freigeben` dieser Speicherbereich freigegeben. Beim Verlassen von `funk3` wird auch der Destruktor für `m1` aufgerufen und damit für den gleichen Speicherbereich abermals `delete[ ]` aufgerufen. Dieses führt zu undefiniertem Verhalten.
Die Lösung ist somit die Implementierung eines tief kopierenden Kopier-Konstruktors.

g (4)) Implementieren Sie den Kopier-Konstruktor für die Klasse `Matrx`.

Lösung:

```
Matrix::Matrix(const Matrix& m2) :
   zAnz(m2.zAnz), spAnz(m2.spAnz) {
   datei << " +MC " << zAnz*spAnz;
   Init(zAnz * spAnz, &daten);
   for (int i = 0; i < zAnz*spAnz; ++i){
      daten[i] = m2.daten[i];
   }
}
```

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

Lösung:
`right` ist immer ein bereits existierendes Objekt. `this` verweist aber gerade auf ein neues noch nicht existierendes Objekt, das im (Kopier-)Konstruktor gerade erzeugt wird. Deshalb können `&right` und `this` nicht auf die gleiche Instanz verweisen. Die Abfrage ist somit überflüssig. Sie würde immer `true` liefern.

i (2)) Warum darf der Parameter des Kopier-Konstruktors nicht als Werteparameter übergeben werden?

Lösung:
Wenn der Parameter als Wert übergeben wird, wird beim Aufruf des Kopier-Konstruktors der Kopier- Konstruktor aufgerufen, was wiederum zu einem Aufruf des Kopier-Konstruktors führt, was wiederum zu einem Aufruf des Kopier-Konstruktors führt usw.

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?

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

Lösung:
Im Durchschnitt wird GetAt etwas über 10.000 mal im

Signal-Handler aufgerufen. Nehmen wir 11.000 Aufrufe an, so kämen wir auf 11.000 * 0,001 ms + 0,05 ms = 11,05 ms. Dieses ist deutlich kleiner als die geforderten 15 ms. Allerdings bedeutet echtzeitfähig, dass die Funktion unter allen Umständen schneller als 15 Millisekunden laufen muss. Das ist nicht garantiert, denn wenn `big` in jedem Schleifendurchlauf verändert wird, haben wir 20.001 Aufrufe von `GetAt` und somit eine Laufzeit von deutlich über 20 Millisekunden.
Die folgende Implementierung kommt garantiert mit genau 10.000 Aufrufen von `GetAt` aus:

```
void SetBiggestFast(int signum){
   TYP big = fabs(gp_matrix->GetAt(0, 0));
   for (int j = 1; j < MAX; ++j){
      TYP hlp = fabs(gp_matrix->GetAt(0, j));
         if (big < hlp) {
            big = hlp;
         }
      }
   for (int i = 1; i < MAX; ++i){
      for (int j = 0; j < MAX; ++j){
         TYP hlp = fabs(gp_matrix->GetAt(0, j));
         if (big < hlp) {
            big = hlp;
         }
      }
   }
   g_biggest = big;
}
```

Ein guter Compiler wird aber vermutlich diese Optimierung auch selber finden, aber es ist eben nicht garantiert.

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?
Lösung:

```
Vor Mult
 +MC 500 +MC 1000 +M 50
Vor return in op*
 +MC 50 -M 50 -M 1000 -M 500 op= -M 50
Nach Mult
```

Es werden somit 4 Matrix Objekte angelegt und auch vier wieder zerstört. Zwei Matrix-Objekten werden mit Kopier-Konstruktor bei der Übergabe der Faktoren an operator* erzeugt (Achtung: der rechte Operaand wird zuerst übergeben). Ein Matirx-Objekt wird im Operator angelegt (10*5) und ein Matrix-Objekt wird bei der Rückgabe aus der Funktion (10*5) erzeugt.

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)?

Lösung:

```
UsedHeap 400, Max 12800, Calls 5
```

In 1) wurde beschrieben, dass vier Konstruktoren aufgerufen werden. Im Zuweisungsoperator wird auch nochmals Heap-Speicher angefordert, sofern nicht eine effiziente Implementierung für den Zuweisungsoperator gewählt wurde, die abfragt, ob der bisher benötigte Speicherbereich genauso groß ist, wie der nun benötigte Speicherbereich, also z.B.

```
Matrix& Matrix::operator=(const Matrix& m2){
   datei << " op= " << zAnz*spAnz;
   if (m2.spAnz == spAnz && m2.zAnz == zAnz){
      for (int i = 0; i < zAnz*spAnz; ++i){
         daten[i] = m2.daten[i];
      }
   }
   else {
      TYP* hlp = nullptr;
      Init(m2.zAnz*m2.spAnz, &hlp);
      for (int i = 0; i < m2.zAnz*m2.spAnz; ++i){
         hlp[i] = m2.daten[i];
      }
      Freigeben(zAnz*spAnz, &daten);
      zAnz = m2.zAnz;
      spAnz = m2.spAnz;
      daten = hlp;
   }
   return *this;
}
```

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.

Lösung:
Der Heap-Speicherbereich reicht nicht aus, wie die folgende Ausgabe zeigt:

```
UsedHeap 400, Max 12800, Calls 5
```

In den Konstruktoren werden (500 + 1000 + 50 + 50) 1600 Instanzen von `TYP` angefordert. Zusätzlich werden im Zuweisungsoperator nochmals 50 Instanzen von `TYP` angefordert. Diese werden aber erst angefordert, wenn die meisten Instanzen bereits wieder freigegeben sind. Es werden somit maximal 1.600*8 Byte = 12.800 Byte Heap-Speicher benötigt. Dieses ist mehr als die zur Verfügung stehende maximale Anzahl von `11.000` Bytes.

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)?

Lösung:
Am einfachsten ist es, wenn die Parameter an den Operator als `const`-Referenzparameter und nicht alss Werteparameter übergeben werden.

```
   friend Matrix operator*(
      const Matrix& m1, const Matrix& m2);
```

```
TEST2: testMultOpPrint
 +M 1000 +M 500 +M 0
Vor Mult
 +M 50
Vor return in op*
 +MC 50 -M 50 op= 50 -M 50
Nach Mult
UsedHeap 0, Max 800, Calls 3
 -M 50 -M 500 -M 1000
```

Im Operator wird eine Instanz von `Matrix` mit 50 Instanzen von `TYP` erzeugt. Bei der Rückgabe wird nochmals eine Kopie erzeugt (manche Compiler optimieren diese Instanz sogar weg). Dann werden im Zuweisungsopoerator nochmals 50 Instanzen von `TYP` erzeugt. Es werden somit maximal 800 Byte auf dem Heapspeicher gleichzeitig benötigt und es gibt drei Aufrufe von `Init` zur Heap-Speicheranforderung mit `new` bzw. `malloc`.

Mit der bereits gezeigten Optimierung

```
Matrix& Matrix::operator=(const Matrix& m2){
   datei << " op= " << zAnz*spAnz;
   if (m2.spAnz == spAnz && m2.zAnz == zAnz){
      for (int i = 0; i < zAnz*spAnz; ++i){
         daten[i] = m2.daten[i];
      }
   }
   else {
      TYP* hlp = nullptr;
      Init(m2.zAnz*m2.spAnz, &hlp);
      for (int i = 0; i < m2.zAnz*m2.spAnz; ++i){
         hlp[i] = m2.daten[i];
      }
      Freigeben(zAnz*spAnz, &daten);
      zAnz = m2.zAnz;
      spAnz = m2.spAnz;
      daten = hlp;
   }
   return *this;
}
```

im Zuweisungsoperator würden weitere 50 Instanzen eingespart, d.h. es gäbe nur 2 Aufrufe von `Init`. Trotzdem würden weiterhin maximal 800 Byte Heapspeicher benötigt.

Ein weitere Optimierung ist die Implementierung eines Verschiebe-Konstruktoren und eines Verschiebs-Zuweisungsoperators:

```
Matrix::Matrix(Matrix&& m2) :
   zAnz(m2.zAnz), spAnz(m2.spAnz) {
   datei << " +MMo " << zAnz*spAnz;
   daten = m2.daten;
   m2.daten = nullptr;
   m2.spAnz = 0;
   m2.zAnz = 0;
}

Matrix& Matrix::operator=(Matrix&& m2){
   datei << " op=&& " << zAnz*spAnz;
   zAnz = m2.zAnz;
   spAnz = m2.spAnz;
   Freigeben(spAnz*zAnz, &daten);
   daten = m2.daten;
   m2.daten = nullptr;
   m2.spAnz = 0;
   m2.zAnz = 0;
   return *this;
}
```

```
TEST3: testMultOpPrint
 +M 1000 +M 500 +M 0
Vor Mult
 +M 50
Vor return in op*
 +MMo 50 -M 0 op=&& 0 -M 0
Nach Mult
UsedHeap 0, Max 400, Calls 1
 -M 50 -M 500 -M 1000
```

Wir haben nur noch einen Aufruf des Konstruktors in operator* selbst. Der Rest wird durch Verschiebung erledigt. Wir haben somit nur einen Aufruf von `Init` und benötigen maximal 400 Byte im Heap-Speicher.

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:
Die Boole'sche Variable kann in einer Interrupt-Service-Routine verändert werden.

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

Lösung:
Wird die Boole'sche Variable in einer Interrupt-Service-Routine verändert, so darf der Compiler die Speicherposition der Variablen nicht optimieren und z.B. in einem Register ablegen. Dann würde in der Tat eine Todschleife vorliegen. Durch das Schlüsselwort `volatile` wird diese Optimierung verhindert.

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 Aufgabenstellung 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
```

Lösung:

Die vier Matrizen M1, M2, M3, M4 liegen schon vor. Es werden nun noch drei weitere Matrizen gebraucht, z.B. M1*M2, M3*M4 sowie für das Gesamtprodukt (M1*M2)*(M3*M4). Diese drei Ergebnissenmatrizen legen wir vorab an und übergeben sie dann nur noch an `ABS_ApplicationFast` als Referenzparameter. Entsprechend gehen wir bei der globalen Funktion `Mult` vor und ersparen uns somit völlig die Erzeugung von `Matrix`-Instanzen in der Scheife. Die maximale weitere Heapspeichergröße ist 0 Byte. 0 mal wird `Init` und 0 mal wird `Freigeben` innerhalb der `while`-Schleife aufgerufen.

Der Aufruf `Mult` statt `operator*` führrt natürlich auch zu Einspareffekten. Allerdings darf man `Mult` nicht nur zwei verschiedene Parameter übergeben, d.h. zu versuchen, einen der beiden konstanten Eingangsparamter als Ergebnis zu verwenden. Durch die Schleifenkonstruktion würde hierdurch schon der Eingangsparameter verändert werden. Es nützt auch nichts, wenn in `ABS_ApplicationFast` in der Schleife oder in `StartABS1` noch Instanzen von Matrix angelegt werden. Hierfür muss ja auf jeden Fall Heap-Speicher (und zwar in jedem Schleifendurchlauf) angefordert und wieder freigegeben werden.

```
bool StartABS1(
   const Matrix& m1, const Matrix& m2,
   const Matrix& m3, const Matrix& m4,
   Matrix& hlp1, Matrix& hlp2, Matrix& res)
{
   Mult(m1, m2, hlp1);
   Mult(m3, m4, hlp2);
   Mult(hlp1, hlp2, res);
   return res.SummeDiagonale() > 0;
}
```

```
void ABS_ApplicationFast(){
   Matrix m1(m1zAnz, m1spAnz);
   Matrix m2(m1spAnz, m2spAnz);
   Matrix m3(m2spAnz, m3spAnz);
   Matrix m4(m3spAnz, m4spAnz);
   Matrix dum1(m1zAnz, m2spAnz);
   Matrix dum2(m2spAnz, m4spAnz);
   Matrix res(m1zAnz, 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, dum1, dum2, res)){
         datei << "\nUse ABS now";
      }
   }
   datei << "\n";
}
```

dum1 hat eine Dimension von 3*10. dum2 hat eine Dimension von 10*3 und die Ergebnismatrix von 3*3. `Mult` ließe sich verbessern, indem zur Summation eine Hilfsvariable `tmp` verwendet wird und erst zum Schluss einmalig die *t*euere Methode `SetAt` aufgerufen wird.