Name:

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

Matrikelnummer:	*Punktzahl:*
Ergebnis:	

Freiversuch ☐ *F1* ☐ *F2* ☐ *F3* ☐

Klausur im WS 2023/24:

Die verschiedenen Programmierparadigmen von C++ – Lösungen –

Hilfsmittel wie Bücher und Skripte und eigene Notizen sind erlaubt.
Die Nutzung eines Computers z.B. mit einer Programmierumgebung oder einem Compiler ist gar nicht erlaubt!
Austausch von Hilfsmitteln mit Kommilitonen ist **nicht** erlaubt !
Die Kommunikation mit Kommilitonen ist während der Klausur nicht erlaubt.
Anwesenheit von Handys, Smartphones etc. ist bei Klausurteilnehmern im Hörsaal nicht erlaubt.
Sie sind vor Beginn der Klausur am Dozentenpult abzugeben!

Bitte notieren Sie auf **allen** Blättern, die in die Bewertung eingehen sollen, Ihren Namen und Ihre Matrikelnummer.
Auf eine korrekte Anzahl der Blanks und Zeilenumbrüche braucht bei der Ausgabe nicht geachtet zu werden. Dafür werden keine Punkte vergeben.

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 bei der Klausurerstellung lediglich dazu, automatisch eine Lösungsdatei zu erstellen.
Wir befinden uns jeweils im Namensraum `std`, d.h., ein `using namespace std;` dürfen Sie in jeder Codedatei annehmen. Außerdem dürfen Sie annehmen, dass für alle Code-Fragmente die erforderlichen *include*-Anweisungen für C++-Header-Dateien erfolgt sind. Syntaxfehler sind allenfalls unabsichtlich in den Programmfragmenten enthalten.

Für einige Aufgabenteile ist ein Extrablatt zu verwenden; bitte mit Namen und Matrikelnummer beschriften. Die Größe der Lücken gibt keine Hinweise darauf, wie viel Ausgaben erfolgen. Sie dürfen auch alle Lösungen auf Extrablättern notieren.

Geplante Punktevergabe

Punktziel	Im einzelnen	Pkte
Übungen:	XXX	
A1: 34 P.		
A2: 46 P.		
Summe 80+ SP.		

Aufgabe 1 : Objekterzeugung und -zerstörung

ca. 34 Punkte

Die Klasse `Mitarbeiter` hat folgende Schnittstelle:

```
class Mitarbeiter {
public:
    Mitarbeiter();
    Mitarbeiter(int id);
   ~Mitarbeiter();
   void SetWert(int w) { m_id = w; }

   // Kopier-Funktionalität
    Mitarbeiter(const Mitarbeiter& s2);
    Mitarbeiter& operator=(const Mitarbeiter& s2);
   // Verschiebe-Funktionalität
    Mitarbeiter(Mitarbeiter&& s2);
   void operator=(Mitarbeiter&& s2);
private:
   int m_id;
};
```

Die Implementierung der beiden Konstruktoren und des Destruktors ist:

```
Mitarbeiter::Mitarbeiter() {
   m_id = -1;
   datei << " +M " << m_id;
}
Mitarbeiter::Mitarbeiter(int id)  {
   m_id = id;
   datei << " +M " << m_id;
}
Mitarbeiter::~Mitarbeiter() {
   datei << " -M " << m_id;
}
```

Eine Implementierung der Kopier- und Verschiebeoperatoren zeigen die beiden folgenden Listings:

```
Mitarbeiter::Mitarbeiter(const Mitarbeiter& cop) {
   m_id = cop.m_id;
   datei << " +MCop " << m_id;
}
Mitarbeiter&
Mitarbeiter::operator=(const Mitarbeiter& cop) {
   datei<<" op= old "<<m_id<< " new "<<cop.m_id;
   m_id = cop.m_id;
   return *this;
}
```

```
Mitarbeiter::Mitarbeiter(Mitarbeiter&& cop) {
   m_id = cop.m_id;
   datei << " +MMov " << m_id;
   cop.m_id = -2;
}
void Mitarbeiter::operator=(Mitarbeiter&& cop) {
   datei<<" move= this "<<m_id<<" cop "<<cop.m_id;
   m_id = cop.m_id;
   cop.m_id = -3;
}
```

a.) (ca. 4 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f1` ?

```
void f1() {
   Mitarbeiter s1();
   Mitarbeiter s2(4);
   Mitarbeiter s3;
   datei << " Ende" << endl;/*1*/
}
```

Lösung:

```
+M 4 +M -1 Ende
-M -1 -M 4
```

Die Zeile mit `s1()` deklariert lediglich eine Funktion, die eine Instanz des betreffenden Typs zurückliefert. Die Zeile `s3()` ruft den Default-Konstruktor auf, der selbstverständlich auch das Attribut initialisiert.
`s3` wurde als letztes erzeugt und wird damit als erstes wieder zerstört. Es werden somit zwei Destruktoren aufgerufen.

b.) (ca. 3 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f2` ?

```
void f2() {
   Mitarbeiter* p1 = new Mitarbeiter(4);
   unique_ptr<Mitarbeiter> p2(new Mitarbeiter(8));
   datei << " Ende" << endl;/*1*/
}
```

Lösung:

```
+M 4 +M 8 Ende
-M 8
```

`p1` erzeugt eine Instanz auf dem Heap. Es wird kein `delete` für diese Instanz aufgerufen. Es wird somit dafür kein Destruktor aufgerufen. `p2` erzeugt auch eine Instanz auf dem Heap. Bei Beendigung der Funktion wird allerdings der Destruktor von `unique_ptr` aufgerufen. Dieser ruft delete auf und damit wird die Instanz durch Destruktor-Aufruf zerstört.

c.) (ca. 7 P.) Gegeben ist die Funktion `f3`:

```
void f3() {
   Mitarbeiter m1(5); /*1*/
   datei << "\n";
   Mitarbeiter m2(m1); /*2*/
   datei << "\n";
   Mitarbeiter m3(move(m1));
   datei << "\n"; /*3*/
}
```

Lösungen schrittweise auf diesem Arbeitsblatt notieren.
Zu welcher Ausgabe führt ihr Aufruf bis /*1*/?
Lösung:

```
+M 5
```

Ausgabe zwischen zwischen /*1*/ und /*2*/?
Lösung:

```
+MCop 5
```

Ausgabe zwischen zwischen /*2*/ und /*3*/?
Lösung:

```
+MMov 5
```

Ausgabe nach /*3*/?
Lösung:

```
-M 5 -M 5 -M -2
```

m2 wird durch den Kopienkonstruktor erzeugt. m3 würde normalerweise auch durch den Kopierkonstruktor erzeugt, allerdings wird das Argument m1 durch expliziten Aufruf von move in eine temporäres Objekt umgewandelt. Das führt zum Aufruf des Verschiebe-Konstruktors. Der Verschiebe-Konstruktor ändert sein Argument. Deshalb unterscheidet sich die Ausgabe beim Aufruf des letzten Destruktors. Drei Instanzen wurden hier erzeugt und werden auch wieder zerstört.

d.) (ca. 8 P.) Gegeben ist die Funktion `f4`:

```
void funkR(Mitarbeiter& m) {
   datei << " funkR ";
   m.SetWert(51);
}
void funkW(Mitarbeiter m) {
   datei << " funkW";
   m.SetWert(31);
}
void f4() {
   Mitarbeiter m1(5);
   Mitarbeiter m2(50); /*1*/
   datei << "\n";
   funkR(m1);
   datei << "\n";/*2*/
   funkW(m2);
   datei << "\n"; /*3*/
}
```

Lösungen schrittweise auf diesem Arbeitsblatt notieren:
Zu welcher Ausgabe führt ihr Aufruf bis /*1*/?
Lösung:

```
+M 5 +M 50
```

Ausgabe zwischen zwischen /*1*/ und /*2*/?
Lösung:

```
funkR
```

Ausgabe zwischen zwischen /*2*/ und /*3*/?
Lösung:

```
+MCop 50 funkW -M 31
```

Ausgabe nach /*3*/?
Lösung:

```
-M 50 -M 51
```

Bei Aufruf von `funkW` bzw. `funkWert` wird eine Kopie durch Aufruf des Kopien-Konstruktors erzeugt. Die Kopie wird in der Funktion geändert. Beim Verlassen der Funktion wird diese Kopie wieder zerstört. Allerdings wurde die Instanz in der Funktion geändert, sodass sich die ausgegebene Zahl geändert hat.
Der Aufruf von `funkR` bzw. `funkRef` erzeugt keine Kopie. Es wird ein anderer Name für das Original erzeugt. Es wird damit kein Konstruktor aufgerufen. Alle Änderungen in der Funktion am Argument ändern damit in Wirklichkeit das Original. Beim Verlassen der Funktion wird somit auch kein Destruktor aufgerufen. Der Aufruf des Destruktors am Ende von f4 für das übergebene Original führt somit zu einer anderen Zahlenausgabe als im Konstruktor.

e.) (ca. 6 P.) Gegeben ist die Funktion `f5`.

```
void f5(int arg) {
   Mitarbeiter m1(Mitarbeiter(8));
   datei << "\n";
   int a = arg * 4;     /*1*/
   if (a > 0) {
      Mitarbeiter m2 = m1;
      m2.SetWert(6);
   }
   else {
      Mitarbeiter m3[2];
      m3[0] = m1;
   }              /*2*/
   datei << "\n";
}
```

Ihr Aufruf mit dem **Wert 4** führt zur folgenden Ausgabe:

```
+M 8
+MCop 8 -M 6
-M 8
```

Erklären Sie auf einem Extrablatt für jede der Zeilen, wodurch die jeweilige Ausgabe zustande kommt.

Lösung:

Zunächst wird die Instanz von `Mitarbeiter m1` erzeugt. Hierbei optimiert der Compiler. Es wird **nicht** zunächst eine temporäre Instanz von `Mitarbeiter` erzeugt und dann mit dieser **nicht** der Kopierkonstruktor aufgerufen, sondern es wird gleich der Umwandlungskonstruktor zur Umwandlung von `int` in eine Instanz von `Mitarbeiter` aufgerufen.

Die Bedingung wird zu `true` ausgewertet. Der Then-Teil wird betreten. Es wird eine neue Instanz m2 von `Mitarbeiter` auf dem Stack mit Verweis auf den Heap erzeugt. Da eine neue Instanz erzeugt wird, wird der Kopierkonstruktor und nicht der Zuweisungsoperator aufgerufen. Man hätte auch `Mitarbeiter m2(m1);` schreiben können, um deutlicher zu machen, dass der Kopierkonstruktor aufgerufen wird. Das Attribut wird von `8` auf `6` geändert. Am Ende der Then-Verbundanweisung wird m2 wieder zerstört. Die Destruktor gibt `-M 6` aus, da das Attribut zuvor angepasst wurde.

Der Else-Teil wird in diesem Fall nicht betreten. Daher werden auch keine 2 Instanzen auf dem Stack erzeugt und auch nicht wieder zerstört.

Am Ende der Funktion wird noch der Destruktor für m1 aufgerufen. Das Attribut hat immer noch den Wert `8`.

f.) (ca. 6 P.)

Zu welcher Ausgabe führt der Aufruf von `f5` mit dem Wert **minus `5`** zwischen /*1*/ und /*2*/?

Lösung:

```
+M -1 +M -1 op= old -1 new 8 -M -1 -M 8
```

Aufgabe 2 : Komplexe Objekterzeugung und -zerstörung

ca. 46 Punkte

Die Klasse `Hochschule` hat folgende Schnittstelle:

```
class Hochschule {
public:
   Hochschule();
   Hochschule(int a_anz);
   Hochschule(const Hochschule& s2);
   ~Hochschule();
   Hochschule& operator=(const Hochschule& s2);
   int GetAnz() const { return anz;}
private:
   Mitarbeiter dekan;
   Mitarbeiter* p_stud;
   int anz;
};
```

Die Implementierung der Konstruktoren und des Destruktors ist:

```
Hochschule::Hochschule() : dekan(114) {
   p_stud = nullptr;
   anz = 0;
   datei << "\n"; /* H1 */
   datei << " +H " << anz; /* H2 */
}
Hochschule::Hochschule(int a_anz) {
   datei << "\n"; /* H1 */
   dekan = Mitarbeiter(11);
   anz = a_anz; /* H2 */
   datei << "\n+H " << anz;
   if (anz > 0) {
      p_stud = new Mitarbeiter[a_anz]; /* H3 */
   }
   else {
      anz = 0;
      p_stud = nullptr;
   }
}
Hochschule::~Hochschule() {
   datei << " -H " << anz;
   delete[] p_stud;
}
```

Bei der Implementierung von Zuweisungsoperator und Kopierkonstruktor handelt es sich um die Default-Implementierung mit flacher Kopie.

a.) (ca. 5 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `g1` ?

```
void g1() {
   Hochschule wolfenbuettel;
   datei << "\nEnde "; /*1*/
}
```

Lösung:

```
+M 114
+H 0
Ende -H 0 -M 114
```

Sobald einer der Konstruktoren von `Hochschule` betreten wird, sind bereits alle Attribute der Instanz dieser Klasse initialisiert, d.h. es wurde bereits implizit oder explizit der Destruktor der Klasse `Mitarbeiter` aufgerufen. Ein expliziter Aufruf erfolgt durch Aufruf des Konstruktors nach dem Doppelkopf. Ist nichts nach dem Doppelpunkt angegeben, erfolgt der implizite Aufruf des Default-Konstruktors von der Klasse `Mitarbeiter` . Wäre kein Default-Konstruktor vorhanden, würde dieses natürlich ein Syntaxfehler sein.

b.) (ca. 10 P.) Gegeben ist die Funktion `g2`.

```
void g2() {
   Hochschule wolfsburg(2);
   datei << "\nEnde ";/*1*/
}
```

Lösungen schrittweise auf diesem Arbeitsblatt notieren.
Zu welcher Ausgabe führt ihr Aufruf bis /*H1*/?
Lösung:

```
+M -1
```

Ausgabe zwischen zwischen /*H1*/ und /*H2*/?
Lösung:

```
+M 11 move= this -1 cop 11 -M -3
```

Ausgabe zwischen zwischen /*H2*/ und /*H3*/?
Lösung:

```
+H 2 +M -1 +M -1
```

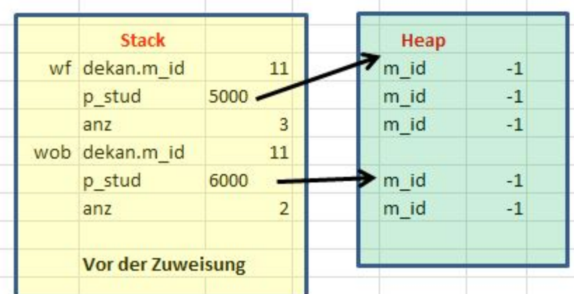
Ausgabe nach /*H3*/?
Lösung:

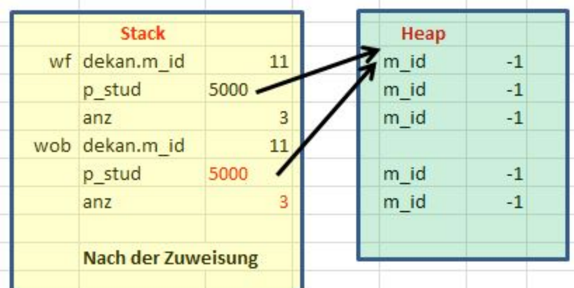
```
Ende -H 2 -M -1 -M -1 -M 11
```

c.) (ca. 9 P.) Erklären Sie, warum die Funktion `g4` durch die fehlende bzw. falsche Implementierung des Zuweisungsoperators zu einem undefinierten Programmverhalten führt. Skizzieren Sie hierzu die Speicherbelegung auf Stack- und Heapspeicher am Ende der Funktion `g4` bei /*1*/ auf einem Extra-Blatt.

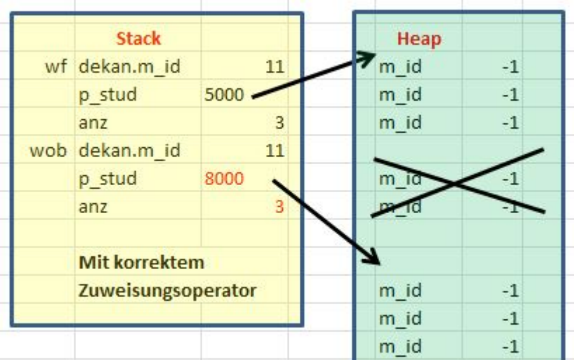
```
void g4() {
   Hochschule wolfenbuettel(3);
   Hochschule wolfsburg(2);
   datei << "\n vor copy\n";
   wolfsburg = wolfenbuettel;
   datei << " Ende" << endl;/*1*/
}
```

Lösung:
Die folgende Abbildung zeigt die Speicherbelegung am Ende der Funktion g4.





Beide Zeiger, d.h. sowohl `wob.p_stud` als auch `wf.p_stud` verweisen nach der Zuweisung auf den gleichen Speicherbereich im Heap. Dadurch wird im Destruktor zweimal der gleiche Heap-Speicherbereich freigegeben. Dies führt zu undefiniertem Programmverhalten. Zudem ergibt sich auch ein Speicherleck.
Die folgende Abbildung zeigt die Speicherbelegung nach der Zuweisung, wenn ein korrekt implementierter Zuweisungsoperator vorhanden ist:



d.) (ca. 13 P.) Implementieren Sie nun einen korrekten Zuweisungsoperator. Notieren Sie Ihre Lösung auf einem Extra-Blatt. Ihre Implementierung wird sehr wahrscheinlich mindestens eine Verzweigung (`if`-Anweisung) benötigen.

Lösung:

```
void CreateHeapAndCopy(
    Mitarbeiter*& p_stud, const Mitarbeiter* p_copyStud,
    int& r_anzDest, int anzSrc) {
    if (p_copyStud != nullptr && anzSrc > 0) {
        p_stud = new Mitarbeiter[anzSrc];
        r_anzDest = anzSrc;
        for (int i = 0; i < anzSrc; ++i) {
            p_stud[i] = p_copyStud[i];
        }
    }
    else {
        p_stud = nullptr;
        r_anzDest = 0;
    }
}
```

```
Hochschule&
Hochschule::operator=(const Hochschule& cop) {
    Mitarbeiter* hlp = nullptr;
    CreateHeapAndCopy(hlp, cop.p_stud,anz,cop.anz);
    delete[] p_stud;
    p_stud = hlp;
    dekan = cop.dekan;
    return *this;
}
```

e.) (ca. 9 P.) Implementieren Sie nun verschiedene Unit-Tests, die zeigen, dass Sie Ihren neuen Zuweisungsoperator auf korrekte Funktionalität prüfen. Nutzen Sie die Methode `GetAnz` sinnvoll. Ihre Tests dürfen abstürzen, wenn die fehlerhafte Default-Implementierung des Zuweisungsoperator vorliegen würde. Sofern Ihre Implementierung des Zuweisungsoperators Verzweigungen benötigt, implementieren Sie verschiedene Tests, sodass zumindest **alle** Pfade und Pfadkombinationen im Zuweisungsoperator durchlaufen werden. Beachten Sie: Tests liefern im Erfolgsfall `true` zurück, sonst `false` und verschiedene Tests sind unabhängig voneinander ausführbar.

Lösung:
Im Zuweisungsoperator erfolgte eine Unterscheidung, ob das Zeiger-Attribut der zu kopierenden Instanz vom nullptr verschieden ist oder nicht. Daher sollte ein Tests eine Zuweisung enthalten, bei der eine Instanz mit Zeiger-Attribut gleich nullptr einer anderen Instanz zugewiesen wird. Einmal sollte ein Tests vorhanden sein, der eine Instanz mit Zeiger verschieden vom nullptr zuweist.
Idealerweise gibt es auch einen Tests, der auf Eigenzuweisung prüft.
Weitere mögliche Tests sind in den folgenden Listings dargestellt.

```
/* Wir erzeugen zwei Instanzen von Hochschule
* Beide sind mit normalen Konstruktor
* erzeugt. Es wird geprüft, ob nach Zuweisung
* beide die gleiche Anzahl haben und Funktion
* nicht abstürzt.
*/
bool testBothValue() {
    Hochschule wf(3);
    Hochschule wob(2);
    wob = wf;
    return (3 == wob.GetAnz() && 3 == wf.GetAnz());
}
```

```
/* Wir erzeugen zwei Instanzen von Hochschule.
* Die erste ist mit Default-Konstruktor erzeugt,
* die zweite hat 3 Studenten. Erste wird zweite
* zugewiesen. Es wird geprüft, ob nach Zuweisung
* beide 0 Studenten haben und nicht abstürzt.
*/
bool testFirstDef() {
    Hochschule wf;
    Hochschule wob(3);
    wob = wf;
    return (0 == wob.GetAnz() && 0 == wf.GetAnz());
}
```

```
/* Wir erzeugen zwei Instanzen von Hochschule.
* Die zweite ist mit Default-Konstruktor erzeugt,
* die erste hat 3 Studenten. Erste wird zweite
* zugewiesen. Es wird geprüft, ob nach Zuweisung
* beide 3 Studenten haben und nicht abstürzt.
*/
bool testSecondDef() {
    Hochschule wf(3);
    Hochschule wob;
    wob = wf;
    return (3 == wob.GetAnz() && 3 == wf.GetAnz());
}
```

```
/* Wir erzeugen zwei Instanzen von Hochschule.
 * Beide werden mit Default-Konstruktor erzeugte.
 * Erste wird zweite
 * zugewiesen.  Es wird geprüft, ob nach Zuweisung
 * beide 0 Studenten haben und Test nicht abstürzt.
 */
bool testBothDef() {
   Hochschule wf;
   Hochschule wob;
   wob = wf;
   return (0 == wob.GetAnz() && 0 == wf.GetAnz());
}
```

```
/* Wir prüfen, ob Eigenzuweisung einer Hochschule,
 * die mit Default-Konstruktor erzeugt wurde,
 *  funktioniert .
 */
bool testOwnAssignmentDef() {
   Hochschule wf;
   wf = wf;
   return 0 == wf.GetAnz();
}
```

```
/* Wir prüfen, ob Eigenzuweisung einer Hochschule
 * die mit normalem-Konstruktor erzeugt wurde,
 *  funktioniert .
 */
bool testOwnAssignmentNormal() {
   Hochschule wf(13);
   wf = wf;
   return 13 == wf.GetAnz();
}
```