Name:

Prof. Dr.-Ing. Hartmut Helmke
Fachhochschule
Braunschweig/Wolfenbüttel
Fachbereich Informatik

Matrikelnummer:	*Punktzahl:*
Ergebnis:	

Freiversuch ☐ *F1* ☐ *F2* ☐ *F3* ☐

Klausur im SS 2009 :

☐ **Programmierkonzepte — Lösungen —**
☐ **Informatik III — Lösungen —**

Informatik B. Sc. ☐ *Technische Informatik B. Sc.* ☐

Hilfsmittel sind bis auf Computer, Handy etc. erlaubt !	Bitte Aufgabenblätter mit abgeben !
Austausch von Hilfsmitteln mit Kommilitonen ist **nicht** erlaubt !	

Die Lösungen sind auf separaten Blättern zu notieren.
Bitte notieren Sie auf **allen** Blättern Ihren Namen bzw. Ihre Matrikelnummer.
Auf eine absolut korrekte Anzahl der Blanks und Zeilenumbrüche braucht bei der Ausgabe nicht geachtet zu werden. Dafür werden keine Punkte abgezogen.

Hinweis: In den folgenden Programmen wird manchmal 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.

In vielen Fällen können Sie die Lösung direkt auf dem Aufgabenblatt notieren.
Gehen Sie davon aus, dass `double` 8 Bytes sowie `int` und Zeiger jeweils 4 Bytes im Speicher belegen.

Geplante Punktevergabe

Planen Sie pro Punkt etwas mehr als eine Minute Aufwand ein.

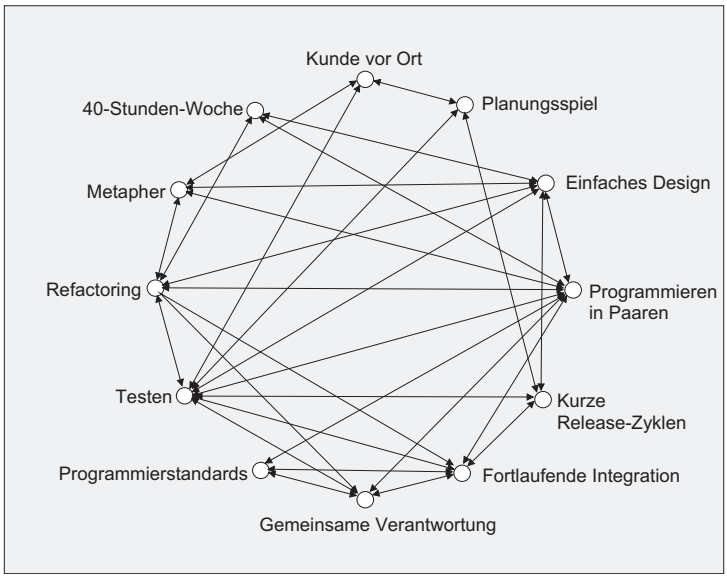
Punktziel	Im einzelnen	Pkte
A1: 14 (2+2+2+3+5) P.		
A2: 12 (4* (2+1)) P.		
A3: 34 (6+6+7+8+7) P.		
A4: 14 (2+ 3*2 + 3*2) P.		
A5: 12 (6*2) P.		
A6: 14 (3+6 + 3+2) P.		
Sonderpunkte Labor		XXXX
Sonderpunkte Übungen		XXXX
Summe		

Aufgabe 1 : Textfragen, Team

ca. 14 (2+2+2+3+5) Punkte

a.) Nennen Sie zwei Basistechniken von Extreme Programming.

b.) Die folgende Grafik zeigt, dass die Basistechniken des Extreme Programming sehr miteinander verzahnt sind (voneinander abhängen).



Beschreiben Sie kurz (ca. 2 Sätze), welche Bedeutung das Testen für die gemeinsame Verantwortung hat.

Lösung:
Durch Tests wird sichergestellt, dass das der Code vor und nach dem Refaktoring immer noch die gleiche Funktionalität hat, d.h. es müssen vorher und nachher alle Tests laufen. Ist das nicht der Fall, weiß man, dass das Refactoring nicht erfolgreich war. Entweder findet man die Fehler oder man setzt auf den Zustand vor dem Refactoring zurück.

c.) Beschreiben Sie kurz (ca. 2 Sätze), wie Refactoring und fortlaufende Integration zusammenhängen.

Lösung:
Eine Überarbeitung des Codes durch Refactoring sollte den anderen Teammitgliedern zeitnah zur Verfügung gestellt werden können. Wird fortlaufend integriert, ist dies gegeben. Andernfalls versuchen andere vielleicht auch noch die gleichen Verbesserungen durchzuführen.

d.) Nennen Sie zwei XP-Basistechniken, die den Truck Factor reduzieren und begründen Sie bei einer Basistechnik auch kurz Ihre Antwort. **Lösung:**
Gemeinsame Verantwortung, Programmieren in Paaren, (zum Teil auch: Programmierstandards, Kurze Releasezyklen, Einfaches Design)

e.) Spezifizieren Sie mit deutschen Worten (nicht in C++ implementieren) zwei **verschiedene** Tests für eine Funktion `sortElem`, die ein Array mit Integer-Werter aufsteigend sortieren soll.

Lösung:
Test1: Die Funktion wird mit dem Array mit den Werten `9, 8, 3, 5` aufgerufen. Es wird geprüft, ob sich `3, 5, 8, 9` ergibt.
Test2: Anschließend wird die Funktion mit gleichen Elementen aufgerufen, d.h. mit `3, 3, 2, 4, 11`. Es wird geprüft, ob sich `2, 3, 3, 4, 11` ergibt.

Aufgabe 2 : Stack-Heapspeicher

ca. 12 (4* (2+1)) Punkte

Veranschaulichen Sie im Folgenden jeweils grafisch die Stack- und ggf. die Heap-Speicherbelegung des folgenden Programmausschnitts zum Zeitpunkt /*2*/. Aus der Zeichnung sollte auch die Wirkungsweise der aufgerufenen Funktion hervorgehen, d.h. der Speicherinhalt zum Zeitpunkt /*1*/ erkennbar sein (eine Zeichnung genügt). Sie können auch mit durchgestrichenen Zahlen oder unterschiedlichen Farben arbeiten, um einen vorherigen Zustand zu kennzeichnen.

a.) Aufruf von Funktion `caller1`:

```
void funk1(int a) {
   a = 61; /*1*/
}
void caller1 () {
   int x=99;
   funk1(x);
    /*2*/
   datei << "caller1 " << x << "\n";
}
```

b.) Welche Ausgabe erzeugt der Aufruf der Funktion `caller1` in die Datei?

Lösung:

```
caller1  99
```

c.) Aufruf von Funktion `caller2`:

```
void funk2(int& a) {
   a = 61; /*1*/
}
void caller2 () {
   int x=99;
   funk2(x);
    /*2*/
   datei << "caller2 " << x << "\n";
}
```

d.) Welche Ausgabe erzeugt der Aufruf der Funktion `caller2` in die Datei?

Lösung:

```
caller2  61
```

e.) Aufruf von Funktion `caller3`:

```
void funk3(int* a) {
   a = new int [3];
   for (int j=0; j < 3; ++j) {
      a[j] = j * j;
   } /*1*/
}
void caller3 () {
   int x=99;
   int* p=&x;
   funk3(p);
    /*2*/
   datei << "caller3 " << *p << "\n";
}
```

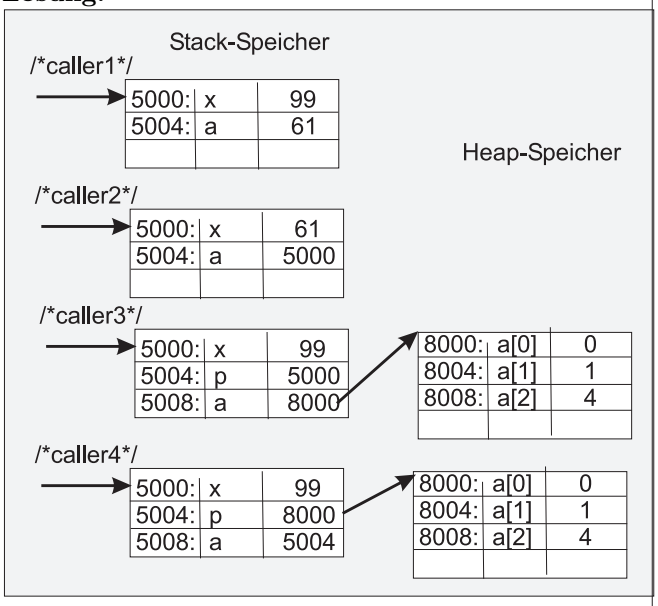
f.) Welche Ausgabe erzeugt der Aufruf der Funktion `caller3` in die Datei?
Lösung:

```
caller3 99
```

g.) Aufruf von Funktion `caller4`:

```
void funk4(int*& a) {
   a = new int [3];
   for (int j=0; j < 3; ++j) {
      a[j] = j * j;
   } /*1*/
}
void caller4 () {
   int x=99;
   int* p=&x;
   funk4(p);
    /*2*/
   datei << "caller4 " << *p << "\n";
}
```

Lösung:



h.) Welche Ausgabe erzeugt der Aufruf der Funktion `caller4` in die Datei?
Lösung:

```
caller4 0
```

Aufgabe 3 : Abstrakte Datentypen

ca. 34 (6+6+7+8+7) Punkte
Der folgende Code zeigt die Deklaration der beiden benutzerdefinierten Datentypen `Tier` und `Zoo`.

```
struct Tier {
   string farbe;
   double gewicht;
};


struct Zoo {
   Tier tiere[100];
};
```

Eine Anwendung zeigt die Funktion `initZoo`.

```
/** Das i-te Tier des Zoos
  emma wird mit "gelb" und 100.2 initialisiert .
*/
void initZoo(int i, Zoo& emma) {
   emma.tiere[i].farbe="gelb";
   emma.tiere[i].gewicht=100.2;
}
```

a.) Spezifizieren Sie zunächst mit Worten einen Test für die Funktion `initZoo`.
Lösung:

```
/** Es wird ein leerer Zoo angelegt.
Nach Aufruf von initZoo zur
 Initialisierung vom ersten und zehnten
Tier wird geprüft,
ob die Tiere die Farbe gelb und ein
Gewicht von 100.2 besitzen.
*/
```

b.) Implementieren Sie den soeben spezifizierten Test in C++.
Lösung:

```
/** Es wird geprüft, ob sich d1 und d2
um höchstens eps unterscheiden.
*/
const double eps = 0.000001;
bool doubleEqual(double d1, double d2) {
   return fabs(d1-d2) <= eps;
}
bool testInitZoo () {
   Zoo zoo1;
   initZoo(0, zoo1);
   initZoo(9, zoo1);
   return
      "gelb" == zoo1.tiere[0].farbe &&
      doubleEqual(zoo1.tiere[0].gewicht, 100.2)&&
      "gelb" == zoo1.tiere[9].farbe &&
      doubleEqual(zoo1.tiere[9].gewicht, 100.2);
}
```

c.) Passen Sie nun die Struktur `Tier` so an, dass Sie einen abstrakten Datentyp (Klasse) erhalten und zwar ohne öffentliche Attribute. Implementieren Sie alle erforderlichen get- und set-Zugriffsmethoden, sodass

Ihr Test und die Funktion `initZoo` angepasst werden könnten. Die Methoden dürfen inline (direkt im Header) implementiert werden. Vergessen Sie nicht, das Schlüsselwort `const` zu verwenden.

Lösung:

```
class Tier {
   string farbe;
   double gewicht;
public:
   void setFarbe(string f) {
      farbe=f;
   }
   string getFarbe() const {
      return farbe;
   }
   void setGew(double g) {
      gewicht=g;
   }
   double getGew() const {
      return gewicht;
   }
};
```

d.) Passen Sie nun noch die Struktur `Zoo` an. Die Methoden dürfen wiederum inline (direkt im Header) implementiert werden. Hier können Sie sich auf die Methoden beschränken, sodass anschließend der Test für die Funktion `initZoo` angepasst werden kann.

Am besten, Sie beginnen deshalb mit der folgenden Aufgabe zur Anpassung der Benutzung der Struktur `Zoo`.

Lösung:

```
class Zoo {
   Tier tiere[100];
public:
   void setFarbe(string f, int index) {
      tiere[index].setFarbe(f);
   }
   string getFarbe(int index) const {
      return tiere[index].getFarbe();
   }
   void setGew(double gew, int index) {
      tiere[index].setGew(gew);
   }
   double getGew(int index) const {
      return tiere[index].getGew();
   }
};
```

Noch eleganter wird der folgende Code, wenn man eine alternative Implementierung mit Operatoren etc. verwendet:

```
class Tier {
   string farbe;
   double gewicht;
public:
   Tier(string f="", double g=0.0):
      farbe(f), gewicht(g) {;}
   Tier(const Tier& ob):
      farbe(ob.farbe), gewicht(ob.gewicht) {;}
   Tier& operator=(const Tier& ob){
      farbe=ob.farbe;
      gewicht=ob.gewicht;
      return *this;
   }
   string getFarbe() const {
      return farbe;
   }
   double getGew() const {
      return gewicht;
   }
};

bool operator==(const Tier& o1, const Tier& o2) {
   return o1.getFarbe() == o2.getFarbe() &&
      doubleEqual(o1.getGew(), o2.getGew());
}
```

```
class Zoo {
   Tier tiere[100];
public:
   Zoo() {;}
   void setTier(const Tier ob, int j){
      tiere[j]=ob;
   }
   const Tier& getTier(int j) const {
      return tiere[j];
   }
};
```

e.) Passen Sie nun auch den Test für die Funktion `initZoo` an, indem Sie hier nur noch die öffentlichen Methoden von `Tier` und `Zoo` aufrufen.

Lösung:

```
/** Es wird ein leerer Zoo angelegt.
Nach Aufruf von initZoo zur
 Initialisierung vom ersten und zehnten
Produkt wird geprüft,
ob die Produkte die Farbe gelb und ein
Gewicht von 100.2 besitzen.
*/
bool testInitZoo() {
   Zoo zoo1;
   initZoo(0,zoo1);
   initZoo(9,zoo1);
   return
      "gelb" == zoo1.getFarbe(0) &&
      doubleEqual(zoo1.getGew(0), 100.2)&&
      "gelb" == zoo1.getFarbe(9) &&
      doubleEqual(zoo1.getGew(9), 100.2);
}
```

Oder alternativ:

```
/** Es wird ein leerer Zoo angelegt.
Nach Aufruf von initZoo zur
 Initialisierung vom ersten und zehnten
Tier wird geprüft,
ob die Tiere die Farbe gelb und ein
Gewicht von 100.2 besitzen.
*/
bool testInitZoo() {
   Zoo zoo1;
   initZoo(0,zoo1);
   initZoo(9,zoo1);
   Tier loewe("gelb", 100.2);
   return (zoo1.getTier(0) == loewe &&
      zoo1.getTier(9) == loewe);
}
```

Aufgabe 4 : Schleifen

ca. 14 (2+ 3*2 + 3*2) Punkte

a.) Geben Sie jeweils ein Beispiel für eine `for`-Schleife mit **halb-offenen** Intervallen und ein Beispiel für eine `for`-Schleife ohne **halb-offene** Intervalle (nur abgeschlossene Intervalle) an. **Lösung:**

```
for (int i=2; i < 24; ++i) {
```

Ohne halboffene Intervalle:
$for(int\ i = 3; i <= 24; ++i)$

b.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert hat die Variable `summe` am Ende der Schleife?

```
int summe = 0;
for (int i=2; i < 24; ++i) {
   summe++;
}
```

Lösung:

```
loop1: 22 Durchlaeufe
```

c.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert hat die Variable `summe` am Ende der Schleife?

```
int summe = 0;
for (int i=3; i < 31; i++) {
   summe++;
}
```

Lösung:

```
loop2: 28 Durchlaeufe
```

d.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert hat die Variable `summe` am Ende der Schleife?

```
int summe = 0;
for (int i=18; i > -33; i--) {
   summe++;
}
```

Lösung:

```
loop3: 51 Durchlaeufe
```

e.) Wie oft wird die folgende Schleife (in Abhängigkeit von N und K) durchlaufen, d.h. welchen Wert hat die Variable `summe` am Ende der Schleife?

```
int summe = 0;
for (int i=N+4; i < K-3; i++) {
   summe++;
}
```

Lösung:

```
loop4: 20 Durchlaeufe (K-3-(N+4))
```

f.) Warum ist es für den Software-Entwickler von Vorteil, wenn er versucht, **immer** halb-offenen Intervalle als Abbruchbedingung einer for-Schleife zu verwenden (im Gegensatz dazu: manchmal werden halb-offene Intervalle verwendet, ein anderes Mal werden abgeschlossene Intervalle verwendet)?

Lösung:
Er kann dann immer die gleiche Formel verwenden, um die Anzahl der Schleifendurchläufe zu berechnen:
$for(int\ i = Anfang; i < Ende; ++i)$ bzw.
$for(int\ i = Ende; i > Anfang; --i)$
Es sind immer: (Ende minus Anfang) Schleifendurchläufe. Man muss nicht mal plus oder minus Eins rechnen.

g.) Sie wollen eine Funktion implementieren, die die N-te Primzahl ermittelt (N im Bereich von 10 bis ca. 1.000.000). Sie dürfen davon ausgehen, dass es eine Boolesche Funktion `istPrimzahl(int zahl)` gibt, auf die Sie zurückgreifen können. Welchen Schleifentyp (do, for, while) verwenden Sie, d.h. welche Schleife ist hier die geeignetste? Die Funktion sollen Sie nicht wirklich implementieren, sondern nur Ihre Antwort begründen!

Lösung:
Die do-Schleife, denn die Schleife läuft mindestens einmal (sogar mehr als 10mal) durch. Allerdings ist nicht genau bekannt, bei welcher Natürlichen Zahl die Schleife abbricht, da man ja keine Tabelle mit den ersten N Primzahlen ablegen will. Dann bräuchte man gar keine Schleife.

Aufgabe 5 : Erzeugen und Zerstören von Objekten

ca. 12 (6*2) Punkte

Gegeben seien die folgenden Klassen.

```
class Fahrzeug {
public:
   Fahrzeug() {
      datei << "+F " ;
   }
   Fahrzeug(const Fahrzeug& f) {
      datei << "+FCopy ";
   }
   ~Fahrzeug() {
      datei << "-F " ;
   }
};
/***********************************/
class Schiff: public Fahrzeug {
public:
   Schiff(): Fahrzeug() {
      datei << "+S ";
   }
   ~Schiff() {
      datei << "-S ";
   }
};
```

a.) Was gibt der Aufruf der Funktion `test1` in `datei` aus?

```
void test1() {
   Schiff s;
   Fahrzeug f;
   datei << " Ende ";
}
```

Lösung:

```
+F +S +F Ende -F -S -F
```

b.) Was gibt der Aufruf der Funktion `test2` in `datei` aus?

```
void test2() {
   Schiff* s = new Schiff();
   Fahrzeug f;
   datei << " Ende ";
}
```

Lösung:

```
+F +S +F Ende -F
```

c.) Was gibt der Aufruf der Funktion `test3` in `datei` aus?

```
void test3() {
   Fahrzeug f;
   Fahrzeug f2(f);
   datei << " Ende ";
}
```

Lösung:

```
+F +FCopy Ende -F -F
```

d.) Was gibt der Aufruf der Funktion `test4` in `datei` aus?

```
void hlp4(Fahrzeug& f) {
   datei << " hlp ";
}
void test4() {
   Fahrzeug f;
   hlp4(f);
   datei << " Ende ";
}
```

Lösung:

```
+F  hlp  Ende -F
```

e.) Was gibt der Aufruf der Funktion `test5` in `datei` aus?

```
void hlp5(Fahrzeug f) {
   datei << " hlp ";
}
void test5() {
   Fahrzeug f;
   hlp5(f);
   datei << " Ende ";
}
```

Lösung:

```
+F +FCopy hlp -F Ende -F
```

f.) Was gibt der Aufruf der Funktion `test6` in `datei` aus?

```
Fahrzeug hlp6() {
   Fahrzeug x;
   datei << " hlp ";
   return x;
}
void test6() {
   Fahrzeug f;
   f = hlp6();
   datei << " Ende ";
}
```

Lösung:

```
+F +F hlp +FCopy -F -F Ende -F
```

Aufgabe 6 : Polymorphie

ca. 14 (3+6 + 3+2) Punkte

Gegeben seien die folgenden Klassen.

```
ofstream datei("poly.txt", ios::out);

class Fahrzeug {
public:
   Fahrzeug(int r): raeder(r) {}
    virtual int getRaeder() const {return raeder;}
    string getId() const {return "Fahrzeug";}
private:
   int raeder; // Anzahl
};
/***********************************/
class Auto: public Fahrzeug {
public:
   Auto(): Fahrzeug(8) {}
    virtual int getRaeder() const {return 4;}
    string getId() const {return "Auto";}
};
/***********************************/
class Schiff: public Fahrzeug {
public:
    Schiff(): Fahrzeug(-1) {hoehe = 40;}
    virtual int getRaeder() const {return 0;}
    string getId() const {return "Schiff";}
private:
   int hoehe;
};
```

a.) Was gibt der Aufruf der Funktion `polytest1` in `datei` aus?

```
void polytest1() {
    Schiff s;
   Fahrzeug f(-1);
   datei << s.getId()<<" " << s.getRaeder() << "\n";
   datei << f.getId()<<" " << f.getRaeder() << "\n";
}
```

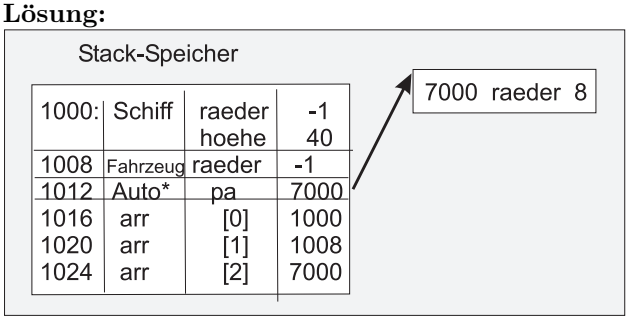
Lösung:

```
Schiff 0
Fahrzeug -1
```

b.) Zeichnen Sie Stack- und Heapspeicherbelegung der Variablen zum Zeitpunkt /*1*/ bei Aufruf der Funktion `polytest2`.

```
void polytest2() {
    Schiff s;
   Fahrzeug f(-1);
   Auto* pa = new Auto();
   // Array mit 3 Zeigern auf Fahrzeuge
   Fahrzeug* arr[3] = {&s, &f, pa};
   for (int j=0; j<3;++j) {
      datei << arr[j]->getId() << " "
         << arr[j]->getRaeder() << "\n";
   }
   /* 1 */
}
```

Lösung:



c.) Was gibt der Aufruf der Funktion `polytest2` in `datei` aus?

Lösung:

```
Fahrzeug 0
Fahrzeug -1
Fahrzeug 4
```

d.) Wie ist der dynamische und statische Typ der Variablen `arr[0]` und `arr[1]`?

Lösung:

`arr[0]`: dyn. Schiff*, stat. Fahrzeug* `arr[1]`: dyn. Fahrzeug*, stat. Fahrzeug*;