Name:

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

Matrikelnummer:	*Punktzahl:*
Ergebnis:	

Freiversuch ☐ *F1* ☐ *F2* ☐ *F3* ☐

Klausur im SS 2011:

Programmierkonzepte — 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 !	

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 Programmfragmenten 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.

Meistens kann die Lösung direkt auf dem Aufgabenblatt notiert werden. Extrablätter bitte mit Namen und/oder Matrikelnummer versehen.
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
Laboraufgaben: 10 +10 P.		
A1: 20 P.		
A2: 35 P.		
A3: 30 P.		
A4: 15 P.		
Summe 100 P.		

Aufgabe 1 : Schleifen/STL

ca. 20 Punkte

a.) (2 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert liefert der Aufruf `cont.size()`?

```
vector<int> cont;
for (int i=1; i < 35; ++i) {
   cont.push_back(17);
}
datei << cont.size() << " runs\n";
```

Lösung:

```
34 runs
```

b.) (2 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert liefert der Aufruf `cont.size()`?

```
list <int> cont;
for (int i=1; i < 35; i++) {
   cont.push_back(17);
}
datei << cont.size() << " runs\n";
```

Lösung:

```
34 runs
```

c.) (2 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert hat die Variable `summe` nach der Schleife?

```
int summe = 0;
for (int i=16; i > 1; i--) {
   summe++;
}
datei << summe << " runs\n";
```

Lösung:

```
15 runs
```

d.) (2 P.) Wie oft wird die folgende Schleife durchlaufen, d.h. welchen Wert hat die Variable `summe` nach der Schleife?

```
int summe = 0;
for (int i=72; i > 12; i = i - 12) {
   summe++;
}
```

Lösung:

```
5 runs
```

e.) (2 P.) Welche Ausgabe wird durch das folgende Code-Fragment in `datei` ausgegeben?

```
vector<int> cont;
for (int i=4; i < 8; ++i) {
   cont.push_back(i);
}
for (unsigned int i=0; i < cont.size(); ++i) {
   datei << cont[i] << " ";
}
```

Lösung:

```
4 5 6 7
```

f.) (2+1 P.) Welche Ausgabe wird durch das folgende Code-Fragment in `datei` ausgegeben?

```
list <int> cont;
for (int i=4; i < 8; ++i) {
   cont.push_back(i);
}
list <int>::iterator iter = cont.begin();
while (iter != cont.end()) {
   datei << *iter << " ";
   ++iter;
}
```

Lösung:

```
4 5 6 7
```

Was könnte bzgl. der Verwendung von `const` im obigen Programmfragment verbessert werden?

Lösung:
`const_iterator` anstelle von `iterator` sollte verwendet werden, denn iter greift nur lesend auf die Container-Elemente zu.
Einen halben Zusatzpunkt, wenn angegeben, dass Konstanten statt den *Magic Numbers* für Schleifenstart, Schleifenende etc. zu verwenden sind.

g.) (3 P.) Welche Ausgabe wird durch das folgende Code-Fragment in `datei` ausgegeben?

```
list <int> cont;
for (int i=4; i < 18; ++i) {
   cont.push_back(i);
}
datei << *max_element(cont.begin(), cont.end());
datei << ", ";
datei << *find(cont.begin(), cont.end(), 11);
```

Lösung:

```
17, 11
```

h.) (4 P.)

Welche Ausgabe wird durch den Aufruf von `funk8` in `datei` ausgegeben?

```
void Print(int i) { datei << i << " "; }
void funk8() {
   vector<int> cont;
   for (int i=2; i < 5; ++i) {
      cont.push_back( i * i);
      cont.push_back( i );
   }
   for_each(cont.begin(), cont.end(), Print);
   datei << "\n";
   sort(++cont.begin(), cont.end()); // increasing
   for_each(cont.begin(), cont.end(), Print);
}
```

Lösung:

```
4 2 9 3 16 4
4 2 3 4 9 16
```

Aufgabe 2 : Instanzerzeugung

ca. 35 Punkte
Für diese Aufgabe wird die folgende Klassendefinition verwendet.

```
class Animal {
public:
   Animal(int le=14) {legs = le;
      datei << "+A " << legs << " ";}
   ~Animal() {datei << "-A " << legs << " ";}
private:
   int legs;
};
```

Veranschaulichen Sie jeweils grafisch die Stack- **und/-oder** Heap-Speicherbelegung in **allen** folgenden Programmfragmenten nach Ausführung der mit (// *1*) gekennzeichneten Programmzeilen.
Beachten Sie bitte auch, dass bei fast allen Aufgaben, die Ausgabe des Funktionsaufrufs in die Datei `datei` zu notieren ist.

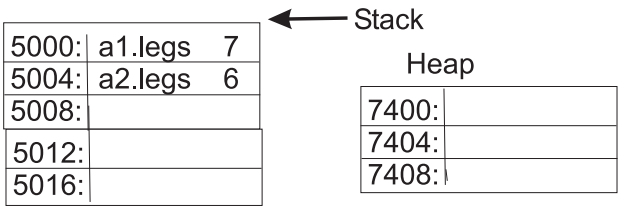
a.) (5 P.) Welche Ausgabe ergibt der Aufruf von `f10`?

```
void f10() {
   Animal a1(7);
   Animal a2(6); //*1*
   }
```

Lösung:

```
f10
+A 7 +A 6 -A 6 -A 7
```

Speicherbelegung:
Lösung:



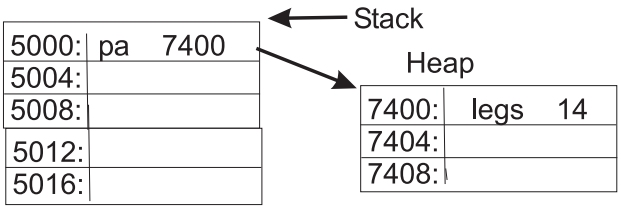
b.) (5 P.) Welche Ausgabe ergibt der Aufruf von `f11`?

```
void f11() {
   Animal* pa = new Animal(); //*1*
   }
```

Lösung:

```
f11
+A 14
```

Speicherbelegung:
Lösung:



c.) (7 P.) Erweitern Sie die Klasse `Animal`, sodass sie eine minimale Standardschnittstelle erhält (Hinweis: der Klasse fehlen noch zwei *Dinge*).
Implementieren Sie die zwei fehlenden Teile.
Lösung:

```
Animal(const Animal& a2){
   legs=a2.legs;
}
Animal& operator=(const Animal& a2){
   legs=a2.legs;
   return *this;
}
```

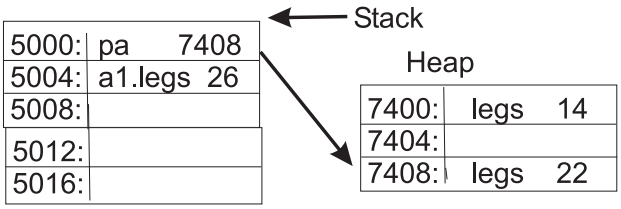
d.) (6 P.) Welche Ausgabe ergibt der Aufruf von `f12`?

```
void f12() {
   Animal* pa = new Animal();
   if (pa != NULL)
   {
      datei << "If ";
      pa = new Animal(22);
      Animal a1(26);          //*1*
      datei << "EndIf ";
   }
   delete pa;
   datei << " EndFunc ";
}
```

Lösung:

```
f12
+A 14 If +A 22 +A 26 EndIf -A 26 -A 22 EndFunc
```

Speicherbelegung:
Lösung:



Es wird willkürlich angenommen, dass das erste `new`

die Adresse 7000 und das zweite 7408 liefert. Das Freibleiben von 7404 ist hier willkürlich.

e.) (6 P.) Welche Ausgabe ergibt der Aufruf von `f13`?

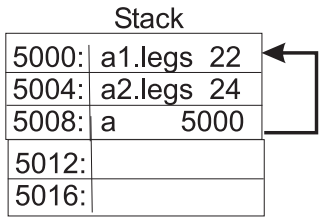
```
Animal makeAnimal(const Animal& a) {
   datei << " EndMake ";  // *1*
   return a;
}
void f13() {
   Animal a1(22);
   Animal a2(24);
   datei << "Call ";
   a2 = makeAnimal(a1);
   datei << " EndFunc ";
}
```

Lösung:

```
f13
+A 22 +A 24 Call EndMake -A 22
EndFunc -A 22 -A 22
```

Speicherbelegung:

Lösung:



f.) (6 P.) Welche Ausgabe ergibt der Aufruf von `f14`?

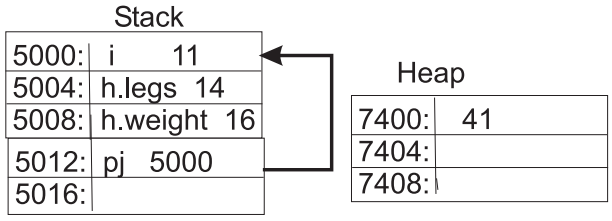
```
class Horse: public Animal {
public:
   Horse(int we=44) {weight = we;
      datei << "+H " << we << " ";}
   ~Horse() {datei << "-H " << weight;}
private:
   int weight;
};
void f14() {
   int i=11;
   Horse h1(16);
   int* pj= new int(41); // no array
   pj= &i;   // *1*
}
```

Lösung:

```
f14
+A 14 +H 16 -H 16-A 14
```

Speicherbelegung:

Lösung:



Aufgabe 3 : Testbasierte Software-Entwicklung

ca. 30 Punkte

Für diese Aufgabe wird die folgende Klassendefinition verwendet.

```
class Animal1 {
public:
   Animal1(int le=14) {legs = le;}
   ~Animal1() { }
   void setLegs(int le){legs=le;}
   int getLegs(){return legs;}
private:
   int legs;
};
```

a.) (2 P.) Welche der Methoden kann als `const` vereinbart werden? Erklären Sie.

Lösung:
Die get-Methode sollte unbedingt als `const` deklariert werden, da sie keine Attribute der Klasse verändert. Die set-Methode ändert jedoch ein Klassenattribut und kann deshalb nicht `const` vereinbart werden.

b.) (7 P.) Die folgende Beschreibung einer Funktion ist gegeben.

```
Die folgende Funktion erzeugt n Animal1 auf
dem Heap. Das erste wird mit 0 Beinen (legs)
 initialisiert , das zweite mit einem,
das dritte mit 2 usw.
Ein Zeiger auf die Animal1 auf dem Heap wird
zurückgegeben.
Für n < 1, wird der Null-Zeiger geliefert .
```

```
Animal1* createAnimals(int n){
```

Beschreiben Sie einen Test für die obige Funktion. Bedenken Sie dabei, dass ein **einziger** Aufruf der zu testenden Funktion vielleicht nicht ausreichend ist, um einigermaßen sicher zu sein, dass die getestete Funktion keine Fehler enthält.
Beschreiben Sie den Test in deutschen Worten, z.B.: Wir rufen die Funktion mit ... auf und prüfen, dass .. und dann rufen wir ... auf und prüfen ...

Lösung:

```
Die Funktion ruft createAnimals mit
den Werten -1, 0, 1 und 4 auf und es
wird geprüft, ob 0,0, 1 bzw. 4 Tiere mit der
entsprechenden Anzahl von Beinen erzeugt werden.
```

c.) (11 P.) Implementieren Sie nun noch den Test in C++. (Das Ergebnis eines Tests ist immer ein Boole'scher Wert.) Vielleicht benötigen Sie auch eine weitere Funktion, die Sie im Test (mehrfach) aufrufen, um doppelten Code zu vermeiden. Vermeiden Sie Speicherlecks.

Lösung:

```
/**
 If no > 0, this function checks
weather the array a containts at position i
(0 <= i < no) an animal with i legs.

Sofern no > 0 ist, prüft die Funktion, ob
das Array a an der Stelle i (0 <= i < no)
ein Tier mit i Beinen enthält.
*/
bool testAnimals(Animal1 a[], int no) {
   if (no < 1) {return true;}
   for (int i=0; i<no; ++i) {
      if (a[i].getLegs() != i ) {
         return false;
      }
   }
   return true;
}
```

Lösung:

```
bool testCreateAnimals() {
   datei << "testCreateAnimals\n";
   bool retValue=true;
   Animal1* pa = createAnimals(-1);
   retValue = testAnimals(pa, 0) && retValue;
   delete[] pa;

   pa = createAnimals(0);
   retValue = testAnimals(pa, 0) && retValue;
   delete[] pa;

   pa = createAnimals(1);
   retValue = testAnimals(pa, 1) && retValue;
   delete[] pa;

   pa = createAnimals(4);
   retValue = testAnimals(pa, 4) && retValue;
   delete[] pa;

   return retValue;
}
```

d.) (7 P.) Implementieren Sie nun noch die Funktion selbst.

Lösung:

```
Animal1* createAnimals(int n){
   if (n < 1) {return 0;}
   Animal1* animals = new Animal1[n];
   for (int i=0; i<n; ++i) {
      animals[i].setLegs(i);
   }
   return animals;
}
```

e.) (3 P.) Die Deklaration der zu testenden Funktion ist eine echte Funktion

```
Animal1* createAnimals(int n){
```

Ändern Sie die Schnittstelle der Funktion, sodass eine Anweisungsfunktion gleicher Funktionalität deklariert wird. Anweisungsfunktionen sind als `void` vereinbart. Nur die Funktionsdeklaration wird hier benötigt.

Lösung:

```
void createAnimals(int no, Animal1*& a);
```

Aufgabe 4 : Fehlersuche

ca. 15 Punkte

Sie sind Entwickler in einem Team, das testbasierte Software-Entwicklung durchführt.

Sie stellen fest, dass der folgende Test `test`, der die Funktion `funcToTest` testen soll, scheitert. Gestern lief der Test allerdings noch erfolgreich.

```
bool test() {
   vector<Object> seq;
   /* ... */
   return funcToTest(seq) <= 284;
}
```

Außerdem stellen Sie fest, dass es für die Funktionen, die von `funcToTest` direkt oder indirekt aufgerufen werden, überhaupt keine eigenen Tests gibt.

Wie könnten Sie vorgehen, um die Ursache des Fehlers zu lokalisieren und schließlich zu beheben? Sie wollen natürlich dafür sorgen, dass zukünftig die Wahrscheinlichkeit für das *Einschleichen* von Fehlern in die Funktion `funcToTest` verringert wird.

Sie sollen hier die Lösung zur Lokalisierung von Fehlern mittels testbasierter Software-Entwicklung allgemein beschreiben. Sie dürfen sich dabei auch auf die folgenden Codefragmente zur Implementierung der Funktion `funcToTest` beziehen.

```
int f2(vector<Object>& seq, int no){
   if (no == 0) {
      return f3(seq[0]);
   }
   else {
      return f4(seq[no], seq[no-1]);
   }
}
```

```
int f1(vector<Object>& seq){
   int retValue = 0;
      /* ... */
   for (unsigned int i=0; i < seq.size(); ++i){
      /* ... */
      retValue += f2(seq, i);
      /* ... */
   }
   /* ... */
   return retValue;
}
```

```
int funcToTest(vector<Object>& seq){
   int retValue=0;
   /* ... */
   while (bed(seq, retValue)){
      retValue = f1(seq);
   }
   /* ... */
   return retValue;
}
```

Sie sollen nicht Fehler im Beispielcode suchen, sondern allgemein beschreiben, wie Sie **systematisch** vorgehen können, um Fehler zu finden und die Qualität der Software zu erhöhen. Deshalb brauchen Sie auch nicht auf die wenig aussagekräftigen Namen der Funktionen oder die fehlende Dokumentation hinzuweisen. Sie sollen hier auch keinen Code schreiben.
In Ihrem Text sollten u.a. die folgende Aspekte angesprochen werden:

- Debugging, Haltepunkt,
- test-first und die Vorteile,
- Red-Bar-Green-Bar (was ist das überhaupt?),
- Tests,
- Erwartungen,
- Abbruch der Fehlersuche.

Lösung:

Gemäß testbasierte Software-Entwicklung sollte für jede Funktion **z**uerst ein Test erstellt werden, bevor mit der Codierung der Funktion selbst begonnen wird (test-first). Da wir nicht wissen, in welcher der vielen direkt und indirekt von `funcToTest` aufgerufenen Funktionen sich der Fehler befindet, werden nun für jede Funktion ohne ausreichende Tests zusätzliche Tests implementiert. Normalerweise wären diese Tests vor Implementierung der Funktion selbst erstellt worden. Es wäre zunächst eine Dummy-Funktion erstellt worden, damit die Tests überhaupt ausgeführt werden können (Red-Bar-Green-Bar-Ansatz).
Vorteil des Test-First-Verfahrens ist, dass man Tests in demselben Maße entwickelt, wie die Funktionalität – so dass Fehler schneller lokalisiert werden können, als wenn dazu Haltepunkte und Debugger benutzt werden.
Wir werden zunächst genügend Tests für die Funktionen erstellen, die selbst keine anderen Funktion aufrufen oder bereits durch automatische Tests überprüft wurden. Wenn hier alle Tests erfolgreich laufen, beginnen wir mit der Erstellung von Tests für die Funktionen, die die bereits getesteten Funktionen nutzen. Irgendwann werden dann (hoffentlich) alle Tests (auch `test`) erfolgreich laufen.
Immer wenn ein Test für eine Funktion scheitert, z.B. der `testF4` für die Funktion `f4`, gehen wir zunächst davon aus, dass sich der Fehler in dieser zu testenden Funktion (oder im Test selbst) befindet. Wir setzen nun Haltepunkte in diese zu testende Funktion oder führen Bildschirmausgaben in die Funktion ein (oder noch besser, wir verwenden die Logging-Funktionalität von `LogTrace`), um Variablenwerte zu prüfen. Wir vergleichen unsere Erwartungen mit den tatsächlichen Variablenwerten. Gibt es Abweichungen, müssen wir entweder unsere Erwartungen anpassen oder wir haben den Fehler lokalisiert. Der Fehler muss sich vor oder während der Ausführung dieser Codezeile ereignet haben. Entweder in der gerade getesteten Funktion oder aber in einer von ihr aufgerufenen Funktion. Letzeres hatten wir ja eigentlich ausgeschlossen, weil wir die aufgerufenen Funktionen schon getestet hatten, aber Tests können immer nur die Anwesenheit von Fehlern, nie die Abwesenheit von Fehlern zeigen (außer in sehr, sehr einfachen Fällen).
Wir beheben den Fehler nun. Nun sollte zumindest dieser Test erfolgreich durchlaufen. Wir setzen mit der Erstellung weiterer Tests und ggf. mit dem entsprechendem Debugging fort, bis schließlich auch der Test `test` für die Funktion `funcToTest` erfolgreich ausgeführt wird.
Andere mögliche Antwort:
Man kann den Fehler identifizieren, indem man einen Haltepunkt in `test` setzt, mit dem Debugger dann Zeile für Zeile durchgeht und die erwarteten Ergebnisse (im Kopf) mit den beobachteten Ergebnissen (im Debugger) vergleicht, bis eine Abweichung vorliegt. Diese könnte die Fehlerursache darstellen. Dieses Vorgehen wäre aber bei jedem neuen Fehler komplett zu wiederholen. Da alle Funktionen in Betracht kommen, ist dieser Vorgang aufwendig, viele Tätigkeiten würden doppelt ausgeführt.
Beim sog. Test-First Ansatz schreibt man erst einen Test, bevor man dann die Funktionalität implementiert. Jeder neue Test muss folglich zu Beginn einen Fehlschlag signalisieren (er soll ja neue, noch zu implementierende Funktionalität abprüfen, die bisher noch nicht vorhanden ist) (*Red-Bar*), nach der korrekten Implementierung dann aber einen Erfolg (fehlerfreier Durchlauf) (*Green-Bar*). In typischen Testumgebungen werden diese Zustände durch einen roten und grünen Balken angedeutet (rot: Fehlschlag, grün: Erfolg). Im vorliegenden Fall ist offensichtlich nicht nach Test-First vorgegangen worden. Vorteil des Test-First-Verfahrens ist, dass man Tests in demselben Maße entwickelt, wie Funktionalität, so dass Fehler schneller lokalisiert werden können, als wenn dazu Haltepunkte und Debugger benutzt werden.
Also würde man in Anlehnung an den test-first Ansatz für zukünftige Überprüfungen Tests anlegen, die die einzelnen Funktionen f3, f4, ... getrennt testen. Jede Funktion für sich ist weniger komplex und daher einfacher zu testen, als die *große* Funktion `funcToTest`. Dazu würde man sich anhand der (fehlenden) Dokumentation/Spezifikation der Funktionen Paare von korrekten Ein- und Ausgaben überlegen und per Test prüfen, ob die Funktion zur gegebenen Eingabe auch die entsprechende Ausgabe liefert. Schlägt z.B. der Test für Funktion f4 fehl, ergibt sich daraus ein potentieller

Fehlerort für den Fehlschlag von `funcToTest`.