Name:

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

Matrikelnummer:	*Punktzahl:*
Ergebnis:	

Freiversuch ☐ *F1* ☐ *F2* ☐ *F3* ☐

Klausur im SS 2010:

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.

In vielen Fällen können Sie die Lösung direkt auf dem Aufgabenblatt notieren. Falls der Platz nicht ausreichen sollte, verweisen Sie **per Pfeil** auf die Extrablätter.
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: (1+3+2+2+ 2+1+3) P.		
A2: 9: (5*1 + 4 Gesamtverständnis) P.		
A3: 16: (6*2+ 3 + 1) P.		
A4: 17: (3*3+ 6 + 2) P.		
A5: 30: (2*2 + 7*1,5 +1+2*1,5 +1+2 +2+2,5+4) P.		
A6: 14: (2 * (2+2+3)) P.		
Sonderpunkte Übungen		XXXX
Summe 100 Punkte		

Aufgabe 1 : Textfragen, Team

ca. 14: (1+3+2+2+ 2+1+3) Punkte

a.) Wofür steht die Abkürzung „XP“ im Zusammenhang mit Software-Prozessmodellen?

Lösung:

e**X**treme **P**rogramming

b.) Sie und Ihr Team gehen nach XP vor. Ihre Aufgabe ist, heute eine doppelverkettete Liste auf Basis einer bereits vorhandenen einfach verketteten Liste zu implementieren. Hierzu spezifizieren Sie die Methoden. Anschließend beschreiben Sie diese in natürlicher Sprache und implementieren Tests für die verschiedenen Methoden der Klasse. Welche Schritte folgen nun noch **in diesem konkreten Beispiel**, bevor Sie den Code dem Rest des Teams durch Einchecken zur Verfügung stellen können?

Lösung:

- Wir refaktorisieren den Code der einfach verketteten Liste
- Wir lassen die Test Suite laufen und prüfen, ob alle Tests für die einfach verkettete Liste trotz Refaktorisierung noch laufen
- Wir implementieren die Funktion (doppelverkettete Liste)
- Wir lassen die Test Suite laufen und prüfen, ob alle Tests für die doppelt verkettete Liste nun funktioneren.
- Wir refaktorisieren unseren Code
- Wir sind fertig, wenn alle Tests erfüllt sind

Es sollte nicht nur aus dem Skript abgeschrieben worden sein, sondern wenigstens ein bisschen auf das spezielle Problem eingegangen worden sein.

c.) Welche Basistechnik von Extreme Programming ist ohne ein Versionsverwaltungssystem wie z.B. SVN nur mit ganz hohem Aufwand umsetzbar?

Lösung:

Gemeinsame Verantwortung, denn wie sollen ohne Versionsverwaltungssystem Änderungen am gleichen Code von verschiedenen Entwicklern zusammengeführt werden.

Auch „Fortlaufende Integration“ wird massiv durch ein Versionsverwaltungssystem unterstützt.

„Refactoring“ wird auch durch ein Versionverwaltungssystem erleichtert, aber hier ist es nicht ganz so zwingend. „Pair Programming“ geht aber auch ohne, wenn es nur um das gemeinsame Programmieren geht.

d.) Was bedeutet Refactoring?

Lösung:

Verbessern des Designs von Code, nachdem er geschrieben wurde, ohne dessen Verhalten zu ändern.

e.) Sie wollen die XP-Basistechnik „Zwei-Leute-ein-Bildschirm“ einsetzen und Ihren Chef davon überzeugen. Warum könnte Ihr Chef dagegen sein? Was antworten Sie ihm, um ihn von den Vorteilen von „Zwei-Leute-ein-Bildschirm“ zu überzeugen?

Lösung:

Vorbehalte des Chefs: Zwei Leute machen das gleiche, wenn jeder etwas anderes machen würde, wäre man doppelt so schnell fertig.

Vorteile von XP: Die Qualität steigt, man muss weniger Zeit für die Fehlersuche aufwenden, sodass insgesamt die Arbeitsgeschwindigkeit steigt. Truck Faktor sinkt.

f.) Sie vervierfachen die Zeit eines Projekts, d.h. z.B. eine Dauer von acht statt zwei Monaten. Welche Auswirkungen auf die Variable Umfang könnte dieses haben?

Lösung:

In der vierfachen Zeit kann man vermutlich viermal so viele Funktionen implementieren, d.h. die Variable Umfang vervierfacht sich auch. Es wird somit davon ausgegangen, dass die Qualität gleich bleibt und mehr Personal zur Verfügung steht, was somit auch die Kosten in etwa linear erhöht. Lässt man die Kosten konstant und die Mitarbeiter arbeiten nur einen Teil der Zeit, würde der Umfang vermutlich in etwa konstant bleiben.

g.) Von wem stammt das *Gesetz* „Adding men to a late project makes it later, not earlier“[1]

Erklären Sie, warum das so ist. Sollte man deshalb grundsätzlich ein Projekt besser mit drei Leuten als mit 200 durchführen?

Lösung:

Brooks Gesetz von Frederick Phillips Brooks, Jr.

Die neuen Leute müssen eingearbeitet werden, was erst mal nochmals Zeit der vorhandenen Belegschaft kostet.

Startet man jedoch ein Projekt zu Beginn mit vielen Mitarbeiten, so sinkt der Zeitbedarf zwar nicht proportional mit der Anzahl der Mitarbeiter, aber 10 Leute werden in der Regel schneller fertig sein als 5, aber eben nicht doppelt so schnell.

Aufgabe 2 : Polymorphie

ca. 9: (5*1 + 4 Gesamtverständnis) Punkte

Gegeben sei die folgende Deklaration der Klasse `Hose`

```
class Hose {
public:
   Hose(int k=0) {knoepfe = k;}
   ~Hose()       {;}

    virtual  int knopfAnzahl() const { return knoepfe;}
    string  farbe() {return "gruen";}

private:
   int knoepfe;
};
```

und die abgeleitete Klasse `Jeans`:

[1] Einem bereits verspätetem Projekt Leute hinzuzufügen, verspätet es noch mehr.

```
class Jeans: public Hose {
public:
   Jeans(): Hose(400) {}

    virtual int knopfAnzahl() const { return 4;}
    string farbe() {return "blau";}
};
```

a.) Welche Ausgabe in `datei` liefert der Aufruf von `abgeleitet1`?

```
void abgeleitet1 (){
   Hose* j1 = new Jeans;
   datei << j1->knopfAnzahl();
   datei << " " << j1->farbe();
}
```

Lösung:

```
abgeleitet1 : 4 gruen
```

b.) Welche Ausgabe in `datei` liefert der Aufruf von `abgeleitet2`?

```
void help(Hose* h2) {
   datei << h2->knopfAnzahl();
   datei << " " << h2->farbe();
}
void abgeleitet2 (){
   Hose* h1 = new Jeans;
   help(h1);
}
```

Lösung:

```
abgeleitet2 : 4 gruen
```

c.) Welche Ausgabe in `datei` liefert der Aufruf von `abgeleitet3`?

```
void helpWert(Hose wert) {
   datei << wert.knopfAnzahl();
   datei << " " << wert.farbe();
}
void abgeleitet3 (){
   Hose* h1 = new Jeans();
   helpWert(*h1);
}
```

Lösung:

```
abgeleitet3 : 400 gruen
```

d.) Welche Ausgabe in `datei` liefert der Aufruf von `abgeleitet4`?

```
void abgeleitet4 (){
   Jeans* jp1 = new Jeans();
   datei << jp1->knopfAnzahl();
   datei << " " << jp1->farbe();
   delete jp1;
}
```

Lösung:

```
abgeleitet4 : 4 blau
```

e.) Welche Ausgabe in `datei` liefert der Aufruf von `abgeleitet5`?

```
void abgeleitet5 (){
   Jeans j2;
   datei << j2.knopfAnzahl();
   datei << " " << j2.farbe();
}
```

Lösung:

```
abgeleitet5 : 4 blau
```

Aufgabe 3 : Werte- und Referenzsemantik

ca. 16: (6*2+ 3 + 1) Punkte

Gegeben sei die folgende Deklaration der Klasse `Hose`.

```
class Hose {
public:
   Hose(int k=0) {knoepfe = k;}
   ~Hose()       {;}

    virtual int knopfAnzahl() const { return knoepfe;}
    string farbe() {return "gruen";}

private:
   int knoepfe;
};
```

a.) Welche Ausgabe in `datei` liefert der Aufruf von `callHose1`?

```
void FunkHose1(Hose h1) {
   Hose hx(44);
   h1 = hx; /* 1 */
}
void callHose1() {
   Hose hc(21);
   datei << hc.knopfAnzahl() << " ";
   FunkHose1(hc);
   datei << hc.knopfAnzahl() << " ";
}
```

Lösung:

```
callHose1:21 21
```

b.) Veranschaulichen Sie im obigen Programmfragment die Speicherbelegung im Stackspeicher unmittelbar nachdem die mit /* 1 */ gekennzeichnete Anweisung ausgeführt wurde.

Lösung:

5000:	hc	21
5004:	h1	44
5008:	hx	44
5012:		
5016:		

Stack-Speicher

c.) Welche Ausgabe in `datei` liefert der Aufruf von `callHose2`?

```
void FunkHose2(Hose& h1) {
   Hose hx(44);
   h1 = hx; /* 1 */
}
void callHose2() {
   Hose hc(21);
   datei << hc.knopfAnzahl() << " ";
   FunkHose2(hc);
   datei << hc.knopfAnzahl() << " ";
}
```

Lösung:

```
callHose2:21 44
```

d.) Veranschaulichen Sie die Speicherbelegung im Stackspeicher unmittelbar nachdem die mit /* 1 */ gekennzeichnete Anweisung ausgeführt wurde.

Lösung:

5000:	fc	~~21~~ / 44
5004:	f1	5000
5008:	fx	44
5012:		
5016:		

Stack-Speicher

e.) Welche Ausgabe in `datei` liefert der Aufruf von `callHose3`?

```
void FunkHose3(Hose* h1) {
   Hose hx(44);
   *h1 = hx; /* 1 */
}
void callHose3() {
   Hose hc(21);
   datei << hc.knopfAnzahl() << " ";
   FunkHose3(&hc);
   datei << hc.knopfAnzahl() << " ";
}
```

Lösung:

```
callHose3:21 44
```

f.) Veranschaulichen Sie die Speicherbelegung im Stackspeicher unmittelbar nachdem die mit /* 1 */ gekennzeichnete Anweisung ausgeführt wurde.

Lösung:

5000:	fc	~~21~~ / 44
5004:	f1	5000
5008:	fx	44
5012:		
5016:		

Stack-Speicher

g.) Beschreiben Sie kurz (2 Sätze reichen vermutlich) die Vorteile der Strategie `Test First`.

Lösung:
Mit dem Test liegt auch eine genaue Spezifikation der zu implementierenden Funktion vor. Sobald die Implementierung abgeschlossen ist, kann man sofort testen (sofortiges Feedback, ob Implementierung erfolgreich war oder nicht. Außerdem kann man die Tests im Falle einer Änderung immer wieder (automatisch) ausführen.
2 Gründen reichen, beim 3. Grund Sonderpunkt **h.)** Hätte die Methode `farbe` auch als konstante Methode vereinbart werden dürfen? Warum?

Lösung:
Ja, denn sie ändert genau wie `kernAnzahl` kein Attribut der Klasse.

Aufgabe 4 : Testen

ca. 17: (3*3+ 6 + 2) Punkte

a.) Im Folgenden ist die Spezifikation eines Tests für die Funktion `erzeugeHose` angegeben (Klasse `Hose` wie vorherige Aufgabe).

```
/* Die Funktion prüft, ob die Funktion erzeugeHose
 korrekt arbeitet, d.h. ob sie zwei neue Instanzen von
Hose mit Werten wert1 und wert2 auf dem Heap erzeugt,
sodass z1 bzw. z2 anschließend darauf verweisen.
Der Test ruft erzeugeHose mit verschiedenen
Werten auf. Nur wenn alle Prüfungen erfolgreich sind,
wird true geliefert. */
```

Die Implementierung des Tests liegt auch bereits vor.

```
bool testErzeugeHose(){
   bool retValue = true;
   Hose* z1=NULL; Hose* z2=NULL;
   int w1 = 22; int w2 = 123;
   // Speicherbelegung in der Funktion (siehe später)
   erzeugeHose(z1, z2, w1, w2);
   retValue = retValue &&
      z1->knopfAnzahl()==w1 &&
      z2->knopfAnzahl()==w2;
   w1 = -22; w2 = -123;
   erzeugeHose(z1, z2, w1, w2);
   retValue = retValue &&
      z1->knopfAnzahl()==w1 &&
      z2->knopfAnzahl()==w2;
   w1 = 400; w2 = w1;
   erzeugeHose(z1, z2, w1, w2);
   retValue = retValue &&
      z1->knopfAnzahl()==w1 &&
      z2->knopfAnzahl()==w2;
   return retValue;
}
```

Ihre Aufgabe ist nun zunächst die Schnittstelle (Prototyp oder auch Funktionsdeklaration genannt) dieser Funktion `erzeugeHose` anzugeben.

Lösung:

```
void erzeugeHose(
   Hose*& pf1, Hose*& pf2,
   int wert1, int wert2);
```

b.) Bevor Sie mit der Implementierung der Funktionalität beginnen, erklären Sie **an diesem Beispiel,** was die Technik `Think, Red-Bar, Green-Bar, Refactor` bedeutet.

Lösung:
Zunächst überlegen wir (Think), was und wie etwas zu tun ist, z.B. benötigen wir die Zeiger als Referenzen. Dann implementieren wir die Funktion ganz einfach, d.h. zu einfach, sodass das Programm zwar kompilierbar ist, aber die Tests noch scheitern. In diesem Fall könnte die Funktion einfach gar nichts machen (leere Funktion). Nun führen wir die Tests aus. Sie scheitern (Red Bar).

Nun wird die Funktionalität korrekt implementiert, d.h. die Test laufen nun erfolgreich durch (Greenz Bar).

Nun verbessern wir ggf. noch das Design, ohne noch die Funktionalität zu verändern (Refactor).

c.) Implementieren Sie die Funktion `erzeugeHose`, sodass der Test `testErzeugeHose` erfolgreich ausgeführt wird [2].

Lösung:

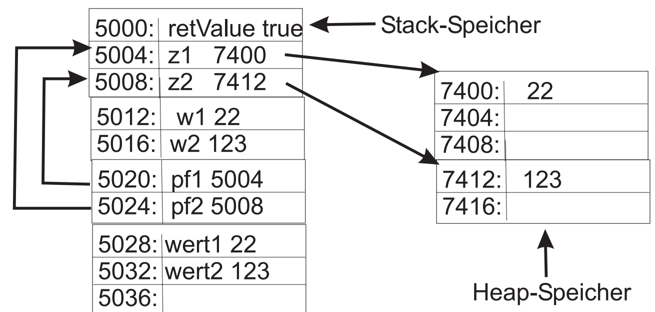
```
void erzeugeHose
   (
   Hose*& ph1,
   Hose*& ph2,
   int wert1,
   int wert2
   )
{
   ph1 = new Hose(wert1);
   ph2 = new Hose(wert2);
}
```

d.) Veranschaulichen Sie die Speicherbelegung (Stack- und Heapspeicher) beim ersten Aufruf von `erzeugeHose` am Ende der Funktion `erzeugeHose`.

Hier noch einmal das relevante Codesegment:

```
   Hose* z1=NULL; Hose* z2=NULL;
   int w1 = 22; int w2 = 123;
   // Speicherbelegung in der Funktion (siehe später)
   erzeugeHose(z1, z2, w1, w2);
```

Lösung:



e.) Begründen Sie, warum es in der vorliegenden Implementierung des Tests `testErzeugeHose` Speicherlecks gibt. Wie viele Instanzen von `Hose` werden insgesamt im Test nicht freigegeben?

Lösung:
Die Funktion `erzeugeHose` erzeugt pro Aufruf zwei Instanzen von `Hose`, die nicht wieder durch `delete` freigegeben werden. Es fehlen somit im Beispieltest 6 `delete`-Aufrufe.

Aufgabe 5 : Konstruktor/Destruktor und Stack-/ Heapspeicherbelegung

ca. 30: (2*2 + 7*1,5 +1+2*1,5 +1+2 +2+2,5+4) Punkte
Gegeben sei die folgende Deklaration der Klasse `Hose`.

```
class Hose {
public:
   Hose(int k=0){knoepfe = k; datei<<"+H "<<knoepfe;}
   ~Hose()     {datei << " -H " << knoepfe;}

    virtual int knopfAnzahl() const { return knoepfe;}
    string farbe() {return "gruen";}

private:
   int knoepfe;
};
```

Es geht im Folgenden jeweils darum, welche Instanzen der Klasse `Hose` wann erzeugt und gelöscht werden, d.h. wann werden welche Konstruktoren und Destruktoren aufgerufen. Vergessen Sie nicht den String `Finito` bei der Ausgabe!
Leerzeichen und Leerzeilen werden bei der Bewertung nicht beachtet.

a.) Welche Ausgabe in `datei` liefert der `funk1`-Aufruf?

```
void funk1() {
   Hose* ph1 = new Hose(4);
   Hose* ph2 = new Hose(2);
   datei << " Finito ";
}
```

Lösung:

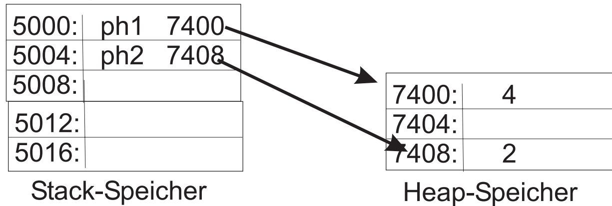
```
funk1:+H 4+H 2 Finito
```

b.) Veranschaulichen Sie die Speicherbelegung der obigen Funktion im Stack- und Heapspeicher unmittelbar nach Ausgabe des Strings `Finito` . Eine Instanz der

[2] Wenn Sie in der vorherigen Teilaufgabe bereits die Schnittstelle angegeben haben, reicht hier der Funktionsrumpf, d.h. alles zwischen der öffnenden und schließenden geschweiften Klammer.

Klasse `Hose` belegt 4 Byte im Speicher. Zeiger belegen ebenfalls 4 Byte.

Lösung:



In der Lösung wurde willkürlich angenommen, dass die beiden new-Aufrufe keine aufeinander folgenden Speicherstellen zurückgeben. Genauso wahrscheinlich ist es aber auch, dass der erste Zeiger auf 7400 und der zweite direkt auf Adresse 7404 verweist.

c.) Welche Ausgabe in `datei` liefert der `funk2`-Aufruf?

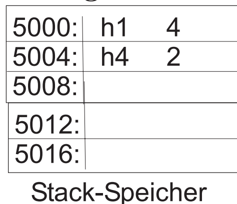
```
void funk2() {
   Hose h1(4);
   Hose h4(2);
   datei << " Finito ";
}
```

Lösung:

```
funk2:+H 4+H 2 Finito -H 2 -H 4
```

d.) Veranschaulichen Sie die Speicherbelegung der obigen Funktion im Stackspeicher unmittelbar nach Ausgabe des Strings `Finito` .

Lösung:



e.) Welche Ausgabe in `datei` liefert der `funk3`-Aufruf?

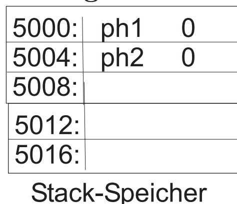
```
void funk3() {
   Hose* ph1=NULL;
   Hose* ph2=ph1;
   datei << " Finito ";
}
```

Lösung:

```
funk3: Finito
```

f.) Veranschaulichen Sie die Speicherbelegung der obigen Funktion im Stackspeicher unmittelbar nach Ausgabe des Strings `Finito` .

Lösung:



g.) Welche Ausgabe in `datei` liefert der `funk4`-Aufruf?

```
void funk4() {
   Hose* ph = new Hose(2);

   // entspricht  Hose h2(*ph);
   Hose h2 = *ph; //
   delete ph;
   datei << " Finito ";
}
```

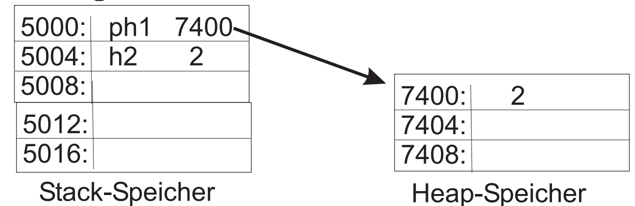
Lösung:

```
funk4:+H 2 -H 2 Finito -H 2
```

`Frucht f2 = *pf;` bzw. `Hose h2 = *ph;` ruft den Kopierkonstruktor auf. Dieses ist der automatisch erzeugte Kopierkonstruktor, der damit keine Ausgabe erzeugt. Nun wird das Objekt, auf das `pf` bzw. `ph` verweist, wieder mit `delete` zerstört, was zur Ausgabe führt. Am Ende der Funktion schließlich wird das Objekt, das mit dem Kopierkonstruktor erzeugt wurde und das auf dem Stack liegt, wieder automatisch durch Destruktoraufruf zerstört (zweite Ausgabe mit minus).

h.) Veranschaulichen Sie die Speicherbelegung der obigen Funktion im Stack- und Heapspeicher unmittelbar **vor** dem `delete`-Aufruf.

Lösung:



i.) Welche Ausgabe in `datei` liefert der `funk5`-Aufruf?

```
void funk5() {
   Hose hosen[3];
   datei << " Finito ";
}
```

Lösung:

```
funk5:+H 0+H 0+H 0 Finito -H 0 -H 0 -H 0
```

j.) Im Gegensatz zur Klasse `Hose` können von der folgenden Klasse `Tier` keine Arrays erzeugt werden (Syntaxfehler).Warum?

```
class Tier {
public:
   Tier(int f) {fuesse = f;}
   ~Tier()     {fuesse=2;}
private:
   int fuesse;
};
```

Lösung:
Die Klasse `Tier` besitzt im Gegensatz zur Klasse `Hose` keinen Standardkonstruktor, d.h. keinen Konstruktor ohne Argumente.

k.) Die folgende Funktion ist aber zumindest syntaktisch richtig. Warum? Können etwa doch Instanzen der

Klasse `Tier` erzeugt werden?

```
void funk5a() {
   Tier* tiere[3];
}
```

Lösung:
Nein, es können weiterhin keine Instanzen von der Klasse `Tier` erzeugt werden. `feld` ist hier ein Array von Zeigern auf Instanzen der Klasse `Tier`, aber es ist kein Array von Instanzen der Klasse `Tier` selbst. Zeiger benötigen keinen expliziten Standardkonstruktor zu ihrer Erzeugung.

l.) Welche Ausgabe in `datei` liefert der `funk6`-Aufruf?

```
void funk6() {
   Hose* hosen[3];
   datei << " Finito ";
}
```

Lösung:

```
funk6: Finito
```

m.) Welche Ausgabe in `datei` liefert der `funk7`-Aufruf?

```
void funk7() {
   Hose* hosen[3];
   hosen[1] = new Hose(4);
   datei << " Finito ";
}
```

Lösung:

```
funk7:+H 4 Finito
```

n.)
Zur Unterstützung der Heap-Freigabe ist die Klasse `AutoPtr` wie angegeben definiert.

```
class AutoPtr{
public:
   AutoPtr(Hose* p) {ph = p;}
   ~AutoPtr() {delete ph;}
private:
   Hose* ph;
   };
```

Welche Ausgabe in `datei` liefert der `funk8`-Aufruf?

```
void funk8() {
   AutoPtr aph1(new Hose(11));
   AutoPtr aph2(new Hose(12));
   datei << " Finito ";
}
```

Lösung:

```
funk8:+H 11+H 12 Finito -H 12 -H 11
```

o.) Welche Ausgabe in `datei` könnte der `funk9`-Aufruf liefern?

```
void funk9() {
   AutoPtr aph1(new Hose(11));
   AutoPtr aph2 = aph1;
   datei << " Finito ";
}
```

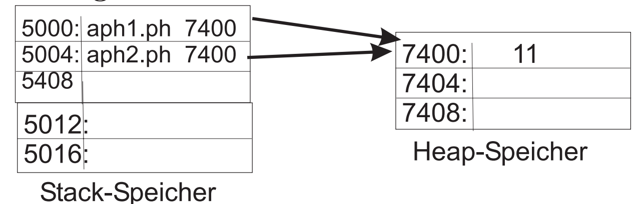
Lösung:

```
nicht definiert : z.B. funk9:+H 11 Finito
-H 11 -H 11
```

Der Aufruf `AutoPtr apf2 = apf1;` bzw. `AutoPtr aph2 = aph1;` ruft keinen Zuweisungsoperator auf, sondern einen Kopierkonstruktor. Der Code entspricht ja: `AutoPtr apf2(apf1);` bzw. `AutoPtr aph2(aph1);`

p.) Veranschaulichen Sie die Speicherbelegung der obigen Funktion im Stack- und Heapspeicher unmittelbar nach Ausgabe des Strings `Finito`.

Lösung:



q.) Welchen Fehler enthält die Klasse `AutoPtr`, der zu einem undefinierten Verhalten des Funktionsaufrufs von `funk9` führt?
Korrigieren Sie diesen Fehler, d.h. erweitern/-verändern Sie die Klasse `AutoPtr`, sodass zumindest `funk9` ein definiertes Verhalten zeigt.

Lösung:
Die Attribute `apf1.pf` und `apf2.pf` zeigen beide auf die gleiche Adresse im Heap-Speicher. Bei Aufruf der Destruktoren für `apf1` bzw. `apf2` wird somit zwei Mal der gleiche Speicherbereich freigegeben.

Die Lösung besteht in der Vereinbarung eines eigenen Kopierkonstruktors (und eines eigenen Zuweisungsoperators, der hier aber noch nicht erforderlich ist). Eine Möglichkeit ist, dass nur jeweils eine Instanz von `AutoPtr` für einen Heap-Speicherbereich zuständig ist. Dann würde im Kopierkonstruktor einer `AutoPtr` die Kontrolle übernehmen und der andere auf 0 gesetzt.

```
class AutoPtr2{
public:
   AutoPtr2(Hose* p) {ph = p;}
   AutoPtr2(AutoPtr2& ah2) {
      ph = ah2.ph; ah2.ph = NULL;
   }
   ~AutoPtr2() {delete ph;}

private:
   Hose* ph;
   };

void funk9b() {
   AutoPtr2 aph1(new Hose(11));
   AutoPtr2 aph2 = aph1;
   datei << " Finito ";
}
```

Beachten Sie, dass das Argument im Kopierkonstruktor `af2` nicht konstant sein kann, da es verändert wird (Attribut `pf` wird auf NULL gesetzt).

Die Ausgabe dieses Programms wäre:

```
funk9b:+H 11 Finito -H 11
```

Eine zweite Möglichkeit besteht im Anlegen einer tiefen Kopie von Frucht im Kopierkonstrutor:

```
class AutoPtr3{
public:
   AutoPtr3(Hose* p) {ph = p;}
   AutoPtr3(const AutoPtr3& ah2) {
      ph = new Hose(*(ah2.ph));
   }
   ~AutoPtr3() {delete ph;}

private:
   Hose* ph;
   };

void funk9c() {
   AutoPtr3 aph1(new Hose(11));
   AutoPtr3 aph2 = aph1;
   datei << " Finito ";
}
```

Beachten Sie, dass das Argument im Kopierkonstruktor `af2` nun wieder konstant sein sollte, denn es wird nicht verändert.

Die Ausgabe dieses Programms wäre:

```
funk9c:+H 11 Finito -H 11 -H 11
```

Aufgabe 6 : Kopieren und Zuweisen

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

Gegeben sei die folgende Deklaration der Klasse `Hose`.

```
class Hose {
public:
   Hose(int k=0) {knoepfe = k;}
   ~Hose()      {;}

    virtual int knopfAnzahl() const { return knoepfe;}
    string farbe() {return "gruen";}

private:
   int knoepfe;
};
```

a.) Implementieren Sie für diese Klasse den Zuweisungsoperator.

b.) Spezifizieren Sie (mit deutschen Worten) einen Test, der die Funktionalität Ihrer Implementierung zeigt/nachweist.

c.) Implementieren Sie nun diesen soeben spezifizierten Test.

d.) Implementieren Sie für diese Klasse den Kopier-Konstruktor.

e.) Spezifizieren Sie (mit deutschen Worten) einen Test, der die Funktionalität Ihrer Implementierung zeigt/nachweist.

f.) Implementieren Sie nun diesen soeben spezifizierten Test.

Lösung:
Kopierkonstruktor und Zuweisungsoperator:

```
Hose2::Hose2(const Hose2& h2) {
   knoepfe = h2.knoepfe;
}
Hose2& Hose2::operator=(const Hose2& h2) {
   // Test auf Eigenzuweisung kann entfallen
   // if (this != &h2) ..
   knoepfe = h2.knoepfe;
   return *this;
}
```

Spezifikation des Tests für den Kopierkonstruktor:

```
/** Es wird eine Hose erzeugt. Mit dem
 Kopierkonstruktor wird eine Kopie erzeugt.
 Anschließend wird geprüft, ob beide Hosen
 gleich sind.
*/
```

Implementierung des Tests für den Kopierkonstruktor:

```
bool testCreateHose() {
   Hose2 h1(14);
   Hose2 h2(h1);
   return h1.knopfAnzahl() == h2.knopfAnzahl();
}
```

Spezifikation des Tests für den Zuweisungsoperator:

```
/** Es werden zwei verschiedene Instanzen von
Hose erzeugt. Anschließend werden diese
mit Zuweisungsoperator einander zugewiesen.
Es wird überprüft, ob beide gleich sind und
den Erwartungen entsprechen.
Anschließend erfolgt noch ein Test, auf
Eigenzuweisung und ob die Kettenzuweisung
 funktioniert .
*/
```

Einfacher Test genügt, alles andere mit Sonderpunkten, auch wenn die Werte geprüft werden und nicht nur geprüft wird, ob getKerne() jeweils das gleiche liefert. Dann könnte aber eigentlich die Zuweisung ja noch falsch herum implementiert worden sein, d.h. das Argument verändert worden sein.

Implementierung des Tests für den Zuweisungsoperator:

```
bool testAssignHose() {
   bool retValue = true;

   Hose2 h1(24);
   Hose2 h2(44);
   retValue = h1.knopfAnzahl() != h2.knopfAnzahl();

   h1 = h2;
   retValue = retValue &&
      h1.knopfAnzahl() == h2.knopfAnzahl() &&
      h1.knopfAnzahl() == 44;

   // Test auf Eigenzuweisung
   Hose2 h3(22);
   h3 = h3;
   retValue = retValue &&
      h3.knopfAnzahl() == 22;

   // Test der Kettenzuweisung
   h1 = h2 = h3;
   retValue = retValue &&
      h3.knopfAnzahl() == h2.knopfAnzahl() &&
      h3.knopfAnzahl() == h1.knopfAnzahl() &&
      h3.knopfAnzahl() == 22;
   return retValue;
}
```