Name:

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

Matrikelnummer:	*Punktzahl:*
Ergebnis:	

Freiversuch ☐ *F1* ☐ *F2* ☐ *F3* ☐

Klausur im WS 2010/11:

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.

Manchmal kann die Lösung direkt auf dem Aufgabenblatt notiert werden. Für die meisten Aufgaben ist aber ein Extrablatt (mit Namen und Matrikelnummer beschriften) zu verwenden.
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: 13 ((6 * 1 +2)+2+2+1) P.		
A2: 20 (1+2+2+1+6+1+2+1+2+2) P.		
A3: 24 (3*1+7+6+4+4) P.		
A4: 43 (8+3+6+8+11+7) P.		
Summe 100 P.		

Aufgabe 1 : Schleifen

ca. 13 ((6 * 1 +2)+2+2+1) Punkte

a.) 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=11; i < 25; ++i) {
   summe++;
}
```

Lösung:

```
loop1: 14 runs
```

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=11; i < 25; i++) {
   summe++;
}
```

Lösung:

```
loop2: 14 runs
```

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=25; i > 11; i--) {
   summe++;
}
```

Lösung:

```
loop3: 14 runs
```

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=72; i > 12; i = i - 6) {
   summe++;
}
```

Lösung:

```
loop4: 10 runs
```

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

```
int summe = 0;
int i=23;
while (i<58) {
   summe++;
   ++i;
}
```

Lösung:

```
loop5: 35 runs
```

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

```
int summe = 0;
int i=12;
while (i<112) {
   summe++;
   i++;
}
```

Lösung:

```
loop6: 100 runs
```

g.) Das Team hatte sich in der letzten Iteration (Dauer 30 Arbeitstage) Aufgaben im Umfang von 12 idealen Tagen vorgenommen. Erledigt wurden allerdings nur Aufgaben im Umfang von 10 idealen Tagen. Zusätzlich wurden ungeplante Aufgaben im Umfang von 6 idealen Tagen erledigt. Berechnen Sie den Load Factor (Rechenweg angeben).

h.) Die neue Iteration beginnt an einem Donnerstag und dauert 60 Arbeitstage. Wie viele Aufgaben (Angabe in idealen Tagen) sollte sich das Team vornehmen, wenn der soeben berechnete Load Factor der Iteration als Entscheidungsgrundlage verwendet wird? (Rechenweg angeben)

Lösung:

Load Factor : $\frac{10}{30} = 33,3\%$, damit kann man sich in der neuen Iteration Aufgaben im Umfang von ca. $(60 * \frac{10}{30} =)$ 20 Tagen vornehmen.

i.) Ihr Programm enthält einige Programmfragmente mehrfach. Was schlagen Sie vor, um diesen Missstand zu beheben (ein *richtiges* Wort reicht)?

Lösung:

Refactoring; akzeptabel ist auch noch: Aufteilung des Codes in Funktionen

Aufgabe 2 : Polymorphie

ca. 20 (1+2+2+1+6+1+2+1+2+2) Punkte
Für diese Aufgabe werden die folgenden Klassendeklarationen verwendet (Achtung: Nur einige Methoden sind als *virtuell* deklariert):

```
class Moebel { // Möbel
   int dummy; // not used
public:
   Moebel() {dummy=2;}
   ~Moebel() {datei << "-M ";}
    virtual string getType() const
      {return "Bruch,";}
   int getLegCnt() const
      {return 0;}
   };

class Couch: public Moebel { // Sofa
public:
   ~Couch() {datei << "-C ";}
    virtual string getType() const
      {return "Couch,";}
   int getLegCnt() const
      {return 4;}
   };
class Tisch: public Moebel { // Tisch
public:
   ~Tisch() {datei << "-T ";}
    virtual string getType() const
      {return "Tisch,";}
   int getLegCnt() const
      {return 3;}
   };
```

a.)
Lösung:
dynamic typ Welcher Typ (dynamischer Typ oder statischer Typ) bestimmt die Klasse, aus der eine virtuelle Methode aufgerufen wird?
Lösung:
dynamischer Typ
b.) Zu welcher Ausgabe führt der Aufruf von `f10`?
Achtung: Auch die Destruktoren erzeugen Ausgaben, aber die Konstruktoren nicht.

```
void f10() {
   Moebel m1;
   Couch c1;
   datei << m1.getLegCnt() <<",";
   datei << c1.getLegCnt() <<",";
   datei << m1.getType() <<",";
   datei << c1.getType() <<",";
   }
```

c.) Zu welcher Ausgabe führt der Aufruf von `f11`?

```
void f11() {
   Moebel* m1 = new Moebel();
   Moebel* c1 = new Couch();
   datei << m1->getLegCnt() <<",";
   datei << c1->getLegCnt() <<",";
   datei << m1->getType() <<",";
   datei << c1->getType() <<",";
   delete c1;
   delete m1;
   }
```

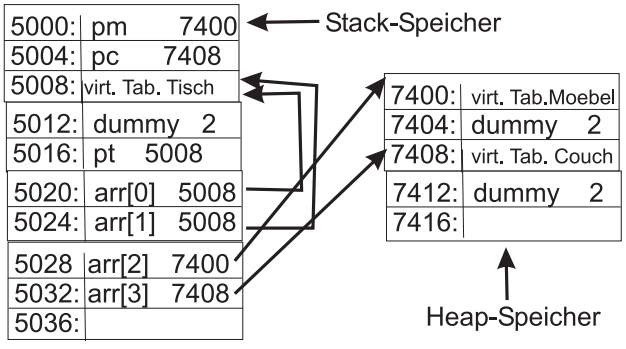
d.) Wie lautet der dynamische Typ der Variablen c1?

Lösung:
Zeiger auf Couch

e.) Veranschaulichen Sie grafisch die Stack- **und** Heap-Speicherbelegung des folgenden Programmfragments vor der `for`-Schleife (// *1*).

```
void f12() {
   Moebel* pm = new Moebel();
   Couch* pc = new Couch();
   Tisch t;
   Tisch* pt = &t;

   Moebel* arr[]={pt, &t, pm, pc};
   // *1*  Stack and Heap-Memory-Contents ???
   for (int i=0; i<4; ++i) {
      datei << arr[i]->getLegCnt()<<",";
      datei << arr[i]->getType() <<"\n";
   }
   delete pm;
   delete pc;
   // delete pt;
   }
```

Lösung:



f.) Wie lautet der statische Typ der Variablen pt?

Lösung:
Zeiger auf Tisch

g.) Zu welcher Ausgabe führt der Aufruf von `f12`?

Lösung:

```
f10
0,4,Bruch,,Couch,,-C -M -M
f11
0,0,Bruch,,Couch,,-M -M
f12
0,Tisch,
0,Tisch,
0,Bruch,
0,Couch,
-M -C -M -T -M
```

h.) Warum ist die auskommentierte Anweisung `delete pt;` nicht erlaubt?

Lösung:
pt verweist auf ein Objekt, das auf dem Stackspeicher erzeugt wurde.

i.) Aufgabe nicht ganz einfach (und wenig Punkte):

Warum führt die Ausführung des folgenden Programmfragments

```
class X {
private:
   int dummy;
public:
    virtual int getCnt() const {return 8;}
};

class Y {
private:
   int dummy;
public:
   int getCnt() const {return 4;}
};

datei << "sizeof(X) = " << sizeof(X) << endl;
datei << "sizeof(Y) = " << sizeof(Y) << endl;
```

zur Ausgabe?

```
sizeof(X) = 8
sizeof(Y) = 4
```

`sizeof` liefert die Größe in Bytes die eine Variable oder ein Typ im Speicher belegt.

Bedenken Sie bei Ihrer Lösung, wie der Compiler zur Laufzeit entscheiden könnte, die richtige virtuelle Methode aufzurufen.

Lösung:
Eine Klasse mit virtuellen Methoden enthält immer auch einen Zeiger auf die Tabelle, die die Adressen der virtuellen Methoden enthält. Der Zeiger benötigt zusätzlich 4 Byte.

j.) Wie wird nun wohl die Ausgabe des folgenden Programmfragments in `datei` sein?

```
class V {
private:
   int dummy; int dummy2;
public:
    virtual int getCnt() const {return 8;}
    virtual int getCnt2() const {return 12;}
};

class W {
private:
   int dummy;  int dummy2;
public:
   int getCnt() const {return 4;}
   int getCnt2() const {return 12;}
};

datei << "sizeof(V) = " << sizeof(V) << endl;
datei << "sizeof(W) = " << sizeof(W) << endl;
```

Lösung:

```
sizeof(V) = 12
sizeof(W) = 8
```

Aufgabe 3 : Objekte flach kopieren

ca. 24 (3*1+7+6+4+4) Punkte
Für diese Aufgabe wird die folgenden Klassendeklaration verwendet:

```
class Moebel { // Furniture
public:
   Moebel(int le=0, double w=0.0)
      {legs=le; weight=w;}
private:
   int     legs;
   double weight;
   friend bool operator==(
         const Moebel&m1,const Moebel&m2){
      return (m1.legs==m2.legs) &&
         (fabs(m1.weight - m2.weight) < 0.0000001);
   }
};
```

a.) Warum wurden die Argumente von `operator==` als `const` vereinbart?
Lösung:
Eingangsparameter, die nicht verändert werden.
b.) Warum werden die Argumente von `operator==` als Referenzparameter übergeben?
Lösung:
2 unnötige Aufrufe des Kopierkonstruktors werden gespart.
c.) Warum wurde `fabs` verwendet, um zu prüfen, ob die 2 `double` Attribute gleich sind?
Lösung:
Aufgrund von Rundungsproblemen sollten double-Variablen nie mit == verglichen werden.
d.) Erweitern Sie die Klasse, sodass Sie eine minimale Standardschnittstelle erhält (Hinweis: Der Klasse fehlen noch drei *Dinge*).
Implementieren Sie die drei fehlenden Teile.
Lösung:

```
Moebel::~Moebel(){
}
Moebel::Moebel(const Moebel& f2)
{
    legs = f2.legs;
   weight = f2.weight;
   ++instOfMoebel;
}
Moebel& Moebel::operator=(const Moebel& f2)
{
    legs= f2.legs;
   weight = f2.weight;
   return *this;
}
```

Im Kopierkonstruktor kann das Incrementieren des statischen Attributs ignoriert werden. Dies wird für eine andere Aufgabe erst benötigt.
e.) Beschreiben Sie einen vollständigen Test für den Zuweisungsoperator zunächst mit Worten (*Wir erzeugen ... verschiedene/gleiche Objekte vom Typ* `Moebel` *und machen dann ... und prüfen dann ... und machen dann ... und prüfen dann ...*)

Prüfen Sie zumindest in diesem Test,

a.) ob die normale Zuweisung funktioniert,

b.) ob eine Eigenzuweisung erfolgreich ist (f1=f1),

c.) ob eine Kettenzuweisung erfolgreich ist (f1=f2=f3).

Lösung:

```
/**
Der folgende Test erzeugt drei verschiedene Instanzen und
prüft, ob alle ungleich sind.
Anschließend wird eine der anderen per Zuweisungsoperator
zugewiesen.
Nun müssen zwei gleich sein und die dritte verschieden.
Es erfolgt eine Eigenzuweisung und Prüfung auf
Gleichheit von zweien und Ungleichheit mit der dritten
Es erfolgt eine Kettenzuweisung, sodass alle
drei Instanzen gleich sein müssen.
*/
```

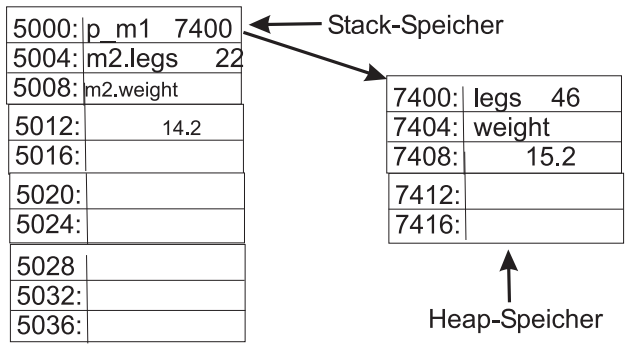
f.) Implementieren Sie nun noch den Test in C++. (Die meisten Punkte gibt es aber auf die vorherige Beschreibung). Das Ergebnis eines Tests ist immer ein Boole'scher Wert.
Lösung:

```
bool testAssOpMoebel() {
   Moebel f1(16, 17.2);
   Moebel f2(14, 18.2);
   Moebel f3(120, 19.4);
   bool retValue = !(f1==f2);
   retValue = !(f1==f3) && retValue;
   retValue = !(f2==f3) && retValue;

   // Use Assignment-Operator
   f1 = f2;
   retValue = retValue &&
      (f1 == f2) && !(f3 == f1);

   // Self Assignment Test
   f2=f2;
   retValue = retValue &&
      (f1 == f2) && !(f3 == f2);

   // Chain Assignment Test
   f1 = f2 = f3;
   retValue = retValue &&
      (f1 == f2) && (f3 == f2);
   return retValue;
}
```

g.) Veranschaulichen Sie graphisch die Stack- und Heap-Speicherbelegung des folgenden Programmfragments.

```
void funk1(){
   Moebel* p_m1 = new Moebel(46, 15.2);
   Moebel m2(22, 14.2);
}
```

Lösung:



Aufgabe 4 : Objekte tief kopieren

ca. 43 (8+3+6+8+11+7) Punkte

Für diese Aufgabe verwenden wir die folgende Klassendeklaration als Ausgangspunkt:

```
/** Container, zum Speichern von Möbeln als Werte
Container which stores Furniture as values
*/
class Raum {
public:
   Raum(int n) {p_moebel = new Moebel[n];
                max = n;cnt=0;}
   ~Raum() {delete[] p_moebel;}
   // insert f into container (no error checking)
   void insertMoebel(const Moebel& f) {
      p_moebel[cnt] = f;
      ++cnt;
   }
   /* ..*/
```

```
private:
   int max; // maximal number of Moebels
   int cnt;  // number of Moebels stored
   Moebel* p_moebel;
};
```

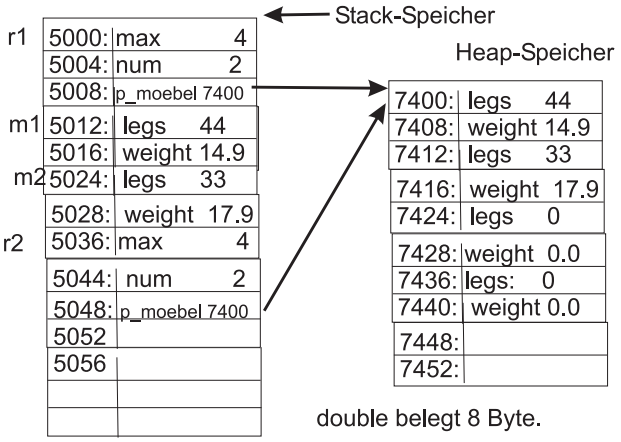
Klasse `Moebel` wie in der vorherigen Aufgabe:

```
class Moebel { // Furniture
public:
   Moebel(int le=0, double w=0.0)
      {legs=le; weight=w;}
private:
   int     legs;
   double weight;
   friend bool operator==(
         const Moebel&m1,const Moebel&m2){
      return (m1.legs==m2.legs) &&
         (fabs(m1.weight - m2.weight) < 0.0000001);
   }
};
```

a.) Veranschaulichen Sie graphisch die Stack- und Heap-Speicherbelegung des folgenden Programmfragments (Zeitpunkt `// memory`). Sie dürfen dazu die folgende Vorlage verwenden.

```
void funk2(){
   Raum r1(4);
   Moebel m1(44, 14.9);
   Moebel m2(33, 17.9);

   r1.insertMoebel(m1);
   r1.insertMoebel(m2);

   // Copy-Constructor
   Raum r2(r1); // memory ?
}
```

Lösung:



b.) Warum führt der Aufruf der obigen Funktion zu einem Programmabsturz bzw. zu einem undefinierten Verhalten?

Lösung:
Da nur der Default-Kopierkonstruktor implementiert ist, kopiert dieser Byte für Byte, sodass sowohl das Attribut `p_moebel` von `b1` als auch von `b2` auf die gleiche Heap-Speicherstelle verweisen. Für beide Instanzen werden die Destruktoren aufgerufen, die durch Aufruf von `delete[]` den Heap-Speicher wieder freigegeben. Es wird somit zweimal der gleiche Heap-Speicher freigegeben.

c.) Korrigieren Sie das Problem durch Implementierung eines Kopierkonstruktors.

Lösung:

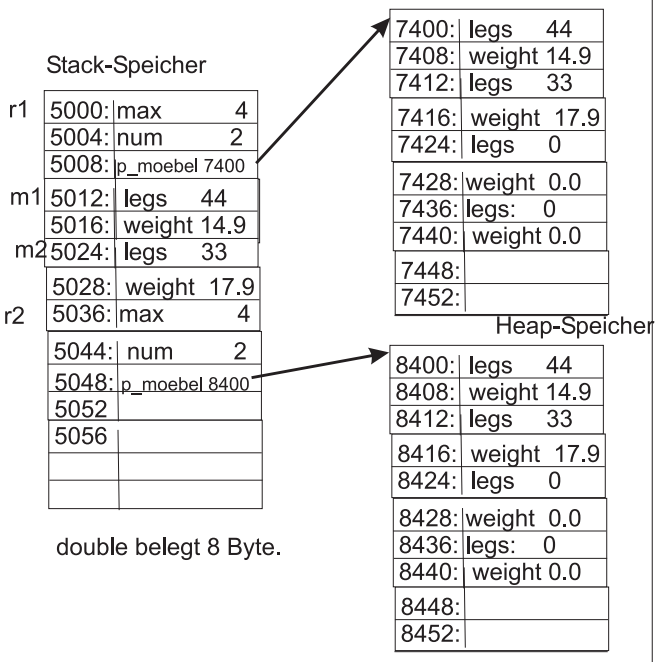
```
Raum::Raum(const Raum& r2)
{
   max = r2.max;
   cnt = r2.cnt;
   p_moebel = new Moebel[max];
   for (int i=0; i<cnt; ++i) {
      p_moebel[i] = r2.p_moebel[i];
   }
   ++instOfRaum;
}
```

Im Kopierkonstruktor kann das Incrementieren des statischen Attributs ignoriert werden. Dies wird für eine andere Aufgabe erst benötigt.

d.) Veranschaulichen Sie graphisch die sich nun ergebende Stack- und Heapspeicherbelegung des obigen Programmfragments (wiederum Zeitpunkt `// memory`).

Gehen Sie davon aus, dass die Klasse einen korrekten Kopierkonstruktor und Zuweisungsoperator besitzt.

Lösung:



e.)

Wir erweitern nun die öffentliche Klassenschnittschnittstelle von `Raum` um einige Methoden:

```
   Moebel moebelNoVal(int i) const {
      return p_moebel[i];}
   const Moebel& moebelNoRef(int i) const {
      return p_moebel[i];}
   int contSize() const {return cnt;}
```

Gegeben sei die folgende Implementierung zum Zusammenfügen von zwei Instanzen der Container-Klasse. Sie ist sehr ineffizient.

```
// r1 mit r2 angehängt wird zurückgegeben
// r1 with r2 appended is returned
Raum Add(Raum r1, Raum r2) {  // 1

   Raum tmp(r1.contSize() + r2.contSize()); //2
   for (int i=0; i<r1.contSize(); ++i) { // 3
      tmp.insertMoebel(r1.moebelNoVal(i)); // 4
   }// for                                // 5
   for (int i=0; i<r2.contSize(); ++i) { // 6
      tmp.insertMoebel(r2.moebelNoVal(i)); // 7
   }// for                                // 8
   return tmp;                           // 9
}                                        // 10
```

Der folgende Test ruft diese ineffiziente Funktion auf.

```
/** 2 Räume werden mit den gleichen Möbeln
 gefülllt . Anschließend wird die Funktion
Add aufgerufen. Raum res sollte dann die
doppelte Anzahl an Möbeln enthalten. Dies
wird geprüft und true oder false geliefert .

2 rooms are created and filled with furniture .
Then we call add.
Room res should contains twice the number
of furniture , which is checked by this test .
*/
bool testAdd() {
   const int SIZE=500;
   Raum ra1(SIZE);
   Raum ra2(SIZE);
   Moebel f1(44, 18.2);
   for (int i=0; i<SIZE; ++i) {
      ra1.insertMoebel(f1);
      ra2.insertMoebel(f1);
   }
   Raum res(1);

   // how many objects of Raum and Moebel
   // are created by the following call
   res = Add(ra1, ra2);                  // 11
                                         // 12
   return (2 * SIZE) == res.contSize();
}
```

Wie viele Instanzen der Klasse `Raum` und wie viele Instanzen der Klasse `Moebel` werden in obigen Test vom folgenden Programmfragment erzeugt?

```
   res = Add(ra1, ra2);                  // 11
```

Sie dürfen davon ausgehen, dass für beide Klassen korrekt implementierte Kopierkonstruktoren und Zuweisungsoperatoren vorhanden sind.

Wichtiger als die exakte Anzahl ist auf jeden Fall, dass Sie erklären, in welcher Zeile des Codes wie viele Instanzen wovon erzeugt werden (Verwenden Sie die im Code angegebenen Zeilennummern zur Erklärung).

Lösung:

```
Created rooms: 4
Created furniture : 5000
```

```
   res = Add(ra1, ra2);                  // 11
Raum Add(Raum r1, Raum r2) {  // 1
call by value:
  Kopierkonstruktor Raum erzeugt für r1   1 Raum
    und                                   SIZE Moebel
  Kopierkonstruktor Raum erzeugt für r2   1 Raum
    und                                   SIZE Moebel

   Raum tmp(r1.contSize() + r2.contSize()); //2
  Kopierkonstruktor Raum erzeugt für tmp 1 Raum
    und                                 2 * SIZE Moebel

      tmp.insertMoebel(r1.moebelNoVal(i)); // 4
          insertMoebel: call by Reference   0 Moebel
          moebelNoVal: return of value
                                          SIZE Moebel

      tmp.insertMoebel(r2.moebelNoVal(i)); // 7
          insertMoebel: call by Reference   0 Moebel
          moebelNoVal: return of value
                                          SIZE Moebel
   return tmp;                             // 9
    Kopierkonstruktor von Raum für return-Wert
      als Wert
                                          1 Raum
                                  2 * SIZE Moebel

   res = Add(ra1, ra2);                  // 11
     res hat nur Größe 1
     Im Zuweisungsoperator muss Platz für
       genügend Moebel geschaffen werden
                                  2 * SIZE Moebel
**********************************************
                       4          Raum
                  10 * SIZE       Moebel
```

f.) Verbessern Sie die Implementierung der Funktion `Add` und ggf. den Aufruf, sodass deutlich weniger Instanzen erzeugt werden. Sie dürfen Schnittstelle und Implementierung der Funktion `Add` ändern. Allerdings dürfen die Schnittstellen von `Raum` and `Moebel` und deren Implementierungen nicht verändert werden.

Hinweis: Vewenden Sie häufiger Referenz- anstatt Wertesemantik.

Lösung:

```
Created rooms: 0
Created furniture : 0
```

ist möglich durch:

```
    Raum res(SIZE+SIZE);
```

```
    AddFast(ra1, ra2, res);
```

mit:

```
// r1 mit r2 angehängt wird in res zurückgegeben
// r1 with r2 appended is returned in res
void AddFast(const Raum& r1,
      const Raum& r2, Raum& res) {
   for (int i=0; i<r1.contSize(); ++i) {
      res.insertMoebel(r1.moebelNoRef(i));
   }// for
   for (int i=0; i<r2.contSize(); ++i) {
      res.insertMoebel(r2.moebelNoRef(i));
   }// for
}
```

Es wird das Resultat gleich groß genug erzeugt, sodass das Ergebnis von `FastAdd` direkt in `res` eingetragen werden kann. Die Argumente werden jeweils als Referenzen geliefert. In der Methode `insert*` wird allerdings ein Objekt überschrieben, aber nicht neu erzeugt, denn es ist auf dem Heap schon vorhanden.