

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

Matrikelnummer:		Punktzahl:	
Ergebnis:			
Freiversuch	<input type="checkbox"/>	F1	<input type="checkbox"/>
		F2	<input type="checkbox"/>
		F3	<input type="checkbox"/>

Klausur im WS 2024/25:

## 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 KommilitonInnen ist <b>nicht</b> erlaubt !
Die Kommunikation mit KommilitonInnen ist während der Klausur nicht erlaubt.
Anwesenheit von Handys, Smartphones etc. ist bei KlausurteilnehmerInnen 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 Leerzeichen 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` dient 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. Sie dürfen auch alle Lösungen auf Extrablättern notieren. Die Größe der Lücken gibt keine Hinweise darauf, wie umfangreich die erwarteten Ausgaben sind.

### Geplante Punktevergabe

Punktziel	Im einzelnen	Pkte
Übungen:	xxx	
A1: 32 P.		
A2: 15 P.		
A3: 20 P.		
A4: 13 P.		
Summe 80+ SP.		

**Aufgabe 1 : Objekterzeugung und -zerstörung**

ca. 32 Punkte

Die Schablonen-Klasse Punkt hat folgende Schnittstelle:

```
template <typename T, int N>
class Punkt
{
public:
    Punkt();
    Punkt(T data[N], int dummy);
    Punkt(const Punkt& cop);
    ~Punkt();
    double operator[](int ind) const {
        return p_koord[ind];
    }
```

```
private:
    void copyArr(T*, const T*);
    T* p_koord;
    int hlp; // Zur Unterscheidung der Instanzen
};
```

Die Implementierung der drei Konstruktoren und des Destruktors der Klasse ist:

```
template <typename T, int N>
Punkt<T, N>::Punkt(){
    hlp = -1;
    datei << "+P" << hlp << " ";
    p_koord = nullptr;
}
```

```
template <typename T, int N>
Punkt<T, N>::Punkt(T data[N], int dummy){
    hlp=dummy;
    datei << "+P" << hlp << " ";
    p_koord = new T[N];
    copyArr(p_koord, data);
}
```

```
template <typename T, int N>
Punkt<T, N>::~Punkt(){
    datei << "-P" << hlp << " ";
    delete[] p_koord;
    p_koord= nullptr;
}
```

```
template <typename T, int N>
Punkt<T, N>::Punkt(const Punkt& cop) {
    hlp = cop.hlp;
    datei << "+PC" << hlp << " ";
    if (cop.p_koord != nullptr) {
        p_koord = new T[N];
        copyArr(p_koord, cop.p_koord);
    }
    else {
        p_koord = nullptr;
    }
}
```

Die mehrfach benutzte Methode copyArr ist wie folgt implementiert:

```
template <typename T, int N>
void Punkt<T, N>::copyArr(T* dest, const T* src){
    for (int j = 0; j < N; ++j) {
        dest[j] = src[j];
    }
}
```

a.) (ca. 2 P.) Welche Ausgabe (nach datei) ergibt der Aufruf der folgenden Funktion f1 ?

```
void f1(){
    double data[] = { 4.0, 5.0, 3.0 };
    Punkt<double, 3> p1(data, 7);
}
```

**Lösung:**

+P7 -P7

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

```
void f2(){
    Punkt<double, 4> feld[2];
    double data[] = { 4.0, 5.0, 3.0, 7.0 };
    Punkt<double, 4> p1(data, 3);
    datei << " Ende ";
}
```

**Lösung:**

+P-1 +P-1 +P3 Ende -P3 -P-1 -P-1

c.) (ca. 4 P.) Welche Ausgabe (nach datei) ergibt der Aufruf der folgenden Funktion f3 ?

```
void f3(){
    Punkt<double, 3>* p;
    double data[] = { 4.0, 5.0, 3.0 };
    Punkt<double, 3>* p_p1 =
        new Punkt<double, 3>(data, -3);
    unique_ptr<Punkt<double, 3>> p_p2(
        new Punkt<double, 3>(data, 6));
    Punkt<double, 3>* p_feld[5];
    datei << " Ende\n";
}
```

**Lösung:**+P-3 +P6 Ende  
-P6

d.) (ca. 4 P.) Welche Ausgabe (nach datei) ergibt der Aufruf der folgenden Funktion f4 ?

```
void call_f4a (Punkt<double, 3> pkt) {
    datei << "call_f4a ";
}
void f4(){
    double data[] = { 1.0, 0.0, 0.0 };
    Punkt<double, 3> p1(data, 3);
    call_f4a (p1);
    datei << "Ende ";
}
```

**Lösung:**

+P3 +PC3 call\_f4a -P3 Ende -P3

Gegeben ist die Funktion f5:

```
void f5(bool b){
    typedef Punkt<double, 2> P2D;
    typedef shared_ptr<P2D> SP;
    double data[] = { 4.0, 5.0 };
    SP s1 = make_shared<P2D>(data, 3);
    s1 = make_shared<P2D>(data, 88);
    SP s4;
    datei << " Vor If\n"; /*1*/

```

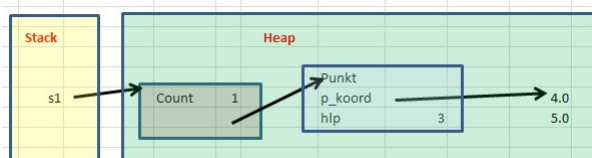
```
if (b) {
    datei << " Then ";
    SP s2 = make_shared<P2D>(data, 4);
    s1 = s2;
    SP s3(s1); /*2*/
}

```

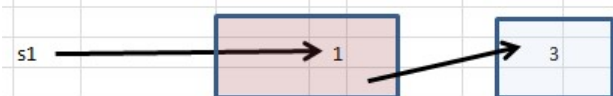
```
else {
    datei << " Else ";
    s1 = s4;
    SP s2 = make_shared<P2D>(data, 5);
    s4 = s2; /*3*/
}
datei << " Nach If\n"; /*4*/
s4 = make_shared<P2D>(data, 11);
s1 = s4;
datei << " Nach Ende\n"; /*5*/
} /*6*/

```

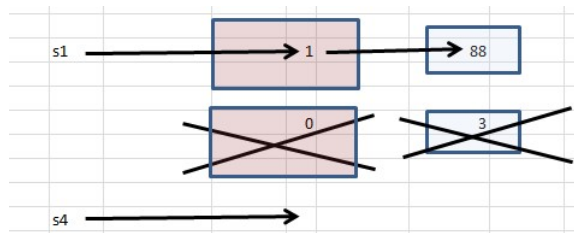
Die folgende Abbildung zeigt die Speicherbelegung auf Stack und Heap nach Belegung der Instanz s1.



Um Zeit zu sparen, vereinfachen Sie die Skizze:



Nach der Initialisierung von s4 durch SP s4; sieht die Skizze wie folgt aus.



Nun sind Sie aber dran. Lösungen bitte schrittweise auf diesem Arbeits- bzw. Extrablatt notieren. Die Funktion wird mit `b=true` aufgerufen.

Skizzieren Sie jeweils auf einem Extrablatt, wie hier gezeigt, auf welche Objekte die Instanzen von `shared_pointer` verweisen. Dabei soll erkennbar sein, wie viele Zeiger auf das Objekt verweisen.

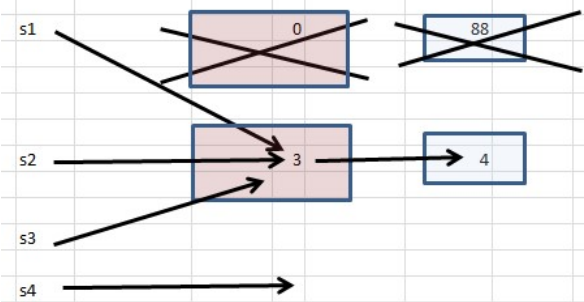
e.) (ca. 2 P.) Zu welcher Ausgabe (nach datei) führt die Ausführung bis `/*1*/`?

**Lösung:**

+P3 +P88 -P3 Vor If

f.) (ca. 2 P.) Skizze nach Ausführung bis `/*2*/` auf Extrablatt (Funktion wird mit `b=true` aufgerufen). In vorherigen Zeichnungen schon freigegebenen Heap-Speicher müssen Sie nicht nochmals zeichnen.

**Lösung:**



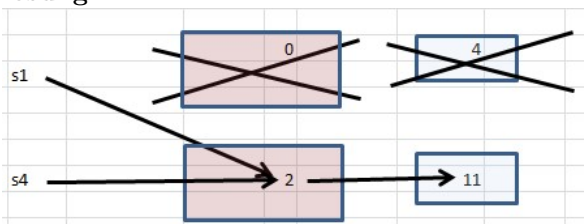
g.) (ca. 2 P.) Ausgabe (nach datei) zwischen `/*1*/` und `/*4*/`?

**Lösung:**

Then +P4 -P88 Nach If

h.) (ca. 2 P.) Skizze bei Ausführung bis `/*5*/` (ab `/*4*/`) auf Extrablatt. Variablen, die nicht mehr auf dem Stack existieren, bitte nicht mehr zeichnen:

**Lösung:**



i.) (ca. 2 P.) Welche Ausgabe nach datei ergibt sich zwischen `/*4*/` und `/*5*/` ?

**Lösung:**

+P11 -P4 Nach Ende

j.) (ca. 2 P.) Ausgabe (nach `datei`) nach `/*5*/?`

Lösung:

```
-P11
```

k.) (ca. 3 P.) Die folgende Funktion ermittelt den Abstand zwischen zwei Punkten.

```
/* Berechnet die Distanz zwischen 2 Punkten */
template <typename T, int N>
double dist(Punkt<T, N> p1, Punkt<T, N> p2){
    double d = 0.0;
    for (int j = 0; j < N; ++j) {
        // [] liefert j-te Koordinate
        auto diff = p1[j] - p2[j];
        d += diff * diff;
    }
    return sqrt(d);
}
```

Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `f6` ?

```
void f6() {
    double data[] = { 1.0, 2.0, 4.0 };
    Punkt<double, 3> p1(data, 3);
    // dist ist 0.
    datei << "dist: " << dist(p1, p1) << "\n";
}
```

Lösung:

```
+P3 dist: +PC3 +PC3 -P3 -P3 0
-P3
```

l.) (ca. 2 P.) Verbessern Sie die aufgerufene Schablonenfunktion `dist`, sodass beim Aufruf weniger Instanzen von `Punkt` erzeugt und zerstört werden. Identischen Code dürfen Sie abkürzen (z.B. durch *Rest wie im Original*).

Bitte Extrablatt verwenden.

Lösung:

```
template <typename T, int N>
double distFast(const Punkt<T, N>& p1,
               const Punkt<T, N>& p2) {
    double d = 0.0;
    for (int j = 0; j < N; ++j) {
        // [] liefert j-te Koordinate
        auto diff = p1[j] - p2[j];
        d += diff * diff;
    }
    return sqrt(d);
}
```

Die Ausgabe wäre dann:

```
+P3 dist: 0
-P3
```

m.) (ca. 2 P.) Warum muss in der Funktion `dist` in der Zeile

```
auto diff = p1[j] - p2[j];
```

der Index-Operator verwendet werden? Hätte man nicht einfach auch

```
auto diff = p1.p_koord[j] - p2.p_koord[j];
```

verwenden können?

Bitte Extrablatt verwenden.

Lösung:

Das Attribut `p_koord` gehört nicht zur Schnittstelle. Es ist im privatem Bereich der Klasse `Punkt`. `abstand` ist keine Methode der Klasse und darf **nicht** auf private Attribute der Klasse zugreifen. Der Index-Operator gehört dagegen zur Klassenschnittstelle. Er ist im `public`-Bereich deklariert. Darauf darf die Funktion `abstand` zugreifen.

**typedef's für folgende Aufgaben:**

Zur Vereinfachung der Schreibweise werden nun einige `typedef`'s eingeführt. Sie helfen Ihnen im Folgenden ggf. auch, wenn Sie selber Code implementieren.

```
// Vereinfachung der Schreibweisen
// P1D bedeutet im Folgenden: Punkt<double, 1>
typedef Punkt<double, 1> P1D;
// P2D bedeutet Punkt<double, 2> usw.
typedef Punkt<double, 2> P2D;
// P3D bedeutet Punkt<double, 3> usw.
typedef Punkt<double, 3> P3D;
typedef Punkt<double, 4> P4D;
typedef Punkt<double, 5> P5D;
```

```
// Entsprechend fuer int
typedef Punkt<int, 1> P1I;
typedef Punkt<int, 2> P2I;
typedef Punkt<int, 3> P3I;
typedef Punkt<int, 4> P4I;
typedef Punkt<int, 5> P5I;
```

**Aufgabe 2 : Testen**

ca. 15 Punkte

a.) (ca. 7 P.) Wie bereits zuvor gesehen, enthält der Kopierkonstruktor der Schablonenklasse `Punkt` eine if-Abfrage, die auch erforderlich ist:

```
template <typename T, int N>
Punkt<T, N>::Punkt(const Punkt& cop) {
    hlp = cop.hlp;
    datei << "+PC" << hlp << " ";
    if (cop.p_koord != nullptr) {
        p_koord = new T[N];
        copyArr(p_koord, cop.p_koord);
    }
    else {
        p_koord = nullptr;
    }
}
```

Implementieren Sie einen Test, der scheitern würde, wenn die if-Abfrage (und der else-Teil) fehlen würde. Es genügt, wenn der Test dann ein undefiniertes Verhalten zeigt, z.B. abstürzt. Im Erfolgsfall wie hier beim vorhandenen if liefert der Test natürlich `true` zurück.

—— Bitte Extrablatt verwenden. ——

**Lösung:**

```
bool test1() {
    /* Pruefung, ob der Kopierkonstruktor nicht abstuerzt,
    wenn erste Objekt nullptr enthaelt */
    P3D p1;
    try {
        P3D p2(p1);
    }
    catch (...) {
        return false;
    }
    return true;
}
```

b.) (ca. 8 P.) Implementieren Sie nun zwei unterschiedliche Tests für die Funktion `dist`, die auch wirklich unterschiedliche Aspekte testen. Um Ihnen das Umblättern zu ersparen, ist der Code hier nochmals angegeben:

```
/* Berechnet die Distanz zwischen 2 Punkten */
template <typename T, int N>
double dist(Punkt<T, N> p1, Punkt<T, N> p2){
    double d = 0.0;
    for (int j = 0; j < N; ++j) {
        // [] liefert j-te Koordinate
        auto diff = p1[j] - p2[j];
        d += diff * diff;
    }
    return sqrt(d);
}
```

Schreiben Sie einen Satz (nicht so viel Text), warum Sie meinen, dass die beiden Tests unterschiedliche Aspekte testen.

—— Bitte Extrablatt verwenden. ——

**Lösung:**

```
bool test2() {
    /* Pruefung, ob zwischen zwei verschiedenen
    Punkten 5 als Abstand ermittelt wird */
    double data1[] = { 4.0, 5.0};
    P2D p1(data1, 7);
    double data2[] = { 7.0, 9.0 };
    P2D p2(data2, 5);

    double abs = dist(p1, p2);
    return 5 == abs;
}
```

**Lösung:**

```
bool test3() {
    /* Pruefung, ob zwischen zwei gleichen
    Punkten 0 als Abstand ermittelt wird */
    double data1[] = { 4.0, 5.0};
    Punkt<double, 2> p1(data1, 7);

    double abs = dist(p1, p1);
    return 0 == abs;
}
```

**Lösung:**

```
bool test4() {
    /* Pruefung, ob auch Punkte der Dimension 5
    funktionieren */
    double data1[] = { 4.0, 5.0, 3.0, 2.0, 1.0 };
    Punkt<double, 5> p1(data1, 7);
    double data2[] = { 4.0, 7.0, 3.0, 2.0, 1.0 };
    Punkt<double, 5> p2(data2, 5);

    double abs = dist(p1, p2);
    return 2 == abs;
}
```

**Lösung:**

```
bool test5() {
    /* Pruefung, ob auch Punkte der Dimension 1
    funktionieren */
    double data1[] = { 4.0};
    Punkt<double, 1> p1(data1, 7);
    double data2[] = { 14.0};
    Punkt<double, 1> p2(data2, 5);

    double abs = dist(p1, p2);
    return 10 == abs;
}
```

### Aufgabe 3 : Tiefe und flache Kopie

ca. 20 Punkte

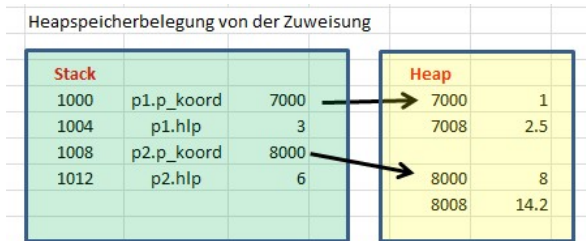
a.) (ca. 9 P.) Der Aufruf der Funktion `f7_Crash` führt zu undefiniertem Verhalten. Erklären Sie warum. Verwenden Sie dazu eine Skizze oder mehrere Skizzen, die die Speicherbelegung auf dem Stack und auf dem Heap skizzieren.

```
void f7_Crash() {
    double data1[] = { 1.0, 2.5 };
    P2D p1(data1, 3);
    double data2[] = { 8.0, 14.2 };
    P2D p2(data2, 6);
    p1 = p2;
}
```

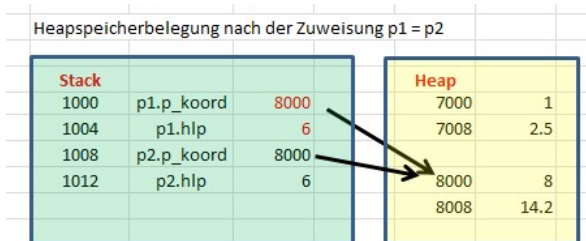
Bitte Extrablatt verwenden.

### Lösung:

Die folgende Abbildung zeigt die Speicherbelegung auf Stack und Heap vor der Zuweisung `p1=p2`;



Und nun die Belegung nach der Zuweisung:



Beide Zeiger zeigen auf die gleiche Speicheradresse im Heapspeicher. Zunächst wird der Speicher im Destruktor von p2 durch Aufruf von `delete[]` freigegeben. Der abermalige Aufruf von `delete[]` im Destruktor von p1 für die gleiche Speicheradresse führt zu undefiniertem Verhalten. Der Zuweisungsoperator darf somit nicht nur flach kopieren, sondern muss tief kopieren, wie der folgenden Aufgabenteil zeigt.

b.) (ca. 11 P.) Verbessern Sie daher die Implementierung der Schablonen-Klasse `Punkt`, sodass es keinen Absturz bzw. undefiniertes Verhalten mehr gibt. Tipp: Sie müssen den Zuweisungsoperator der Klasse implementieren.

Um Ihnen etwas Schreibarbeit zu ersparen, sind die ersten Zeilen der Implementierung bereits angegeben.

```
template <typename T, int N>
Punkt<T, N>& Punkt<T, N>::operator=
(const Punkt<T, N>& cop) {
    hlp = cop.hlp;
    datei << "Pop=" << hlp << " ";
```

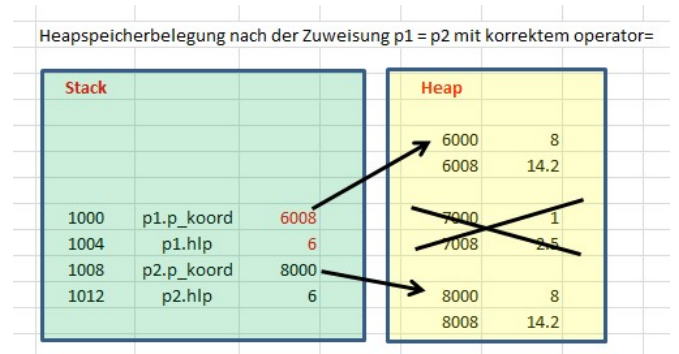
Setzen Sie die Implementierung auf einem Extra-Blatt fort. Überlegen Sie sich eine effiziente Implementierung, die unnötiges Anfordern und Freigeben von Heap-Speicher vermeidet. Rufen Sie die bereits vorgestellte Methode `copyArr` von `Punkt` auf:

```
template <typename T, int N>
void Punkt<T, N>::copyArr(T* dest, const T* src){
    for (int j = 0; j < N; ++j) {
        dest[j] = src[j];
    }
}
```

Bitte Extrablatt verwenden.

### Lösung:

Beim tiefen Kopieren im Zuweisungsoperator zeigen `p1.p_koord` und `p2.p_koord` nicht mehr auf die gleichen Heap-Speicheradressen, sondern auf verschiedene Heap-Speicherbereiche mit gleichem Inhalt verweisen:



```
template <typename T, int N>
Punkt<T, N>& Punkt<T, N>::operator=
(const Punkt<T, N>& cop) {
    hlp = cop.hlp;
    datei << "Pop=" << hlp << " ";
    if (cop.p_koord != nullptr) {
        T* tmp = new T[N];
        copyArr(tmp, cop.p_koord);
        delete[] p_koord;
        p_koord = tmp;
    }
    else {
        delete[] p_koord;
        p_koord = nullptr;
    }
    return *this;
}
```

Wir erkennen aber auch, dass der bisherige Speicherbereich, auf den `this->p_koord` verweist, genau so groß ist wie der neue Speicherbereich. Es reicht somit eigentlich aus, wenn nur die Inhalte kopiert werden, wie die folgende Abbildung zeigt:



Die gewünschte effiziente Implementierung ist also:

```
template <typename T, int N>
Punkt<T, N>& Punkt<T, N>::operator=
(const Punkt<T, N>& cop) {
    hlp = cop.hlp;
    datei << "Pop=" << hlp << " ";
    if (cop.p_koord != nullptr) {
        if (p_koord == nullptr) {
            p_koord = new T[N];
        }
        copyArr(p_koord, cop.p_koord);
    }
    else {
        delete[] p_koord;
        p_koord = nullptr;
    }
    return *this;
}
```

#### Aufgabe 4 : Zusammengesetzte Objekte

ca. 13 Punkte

Wie bereits zuvor gezeigt, verwenden wir zur Vereinfachung der Schreibweise folgenden `typedef`:

```
// P2D bedeutet Punkt<double, 2> usw.
typedef Punkt<double, 2> P2D;
```

Die Klasse `Gerade` selber hat folgende Schnittstelle:

```
class Gerade{
public:
    Gerade(const P2D& a_p1, const P2D a_p2):
        m_p1(a_p1), m_p2(a_p2){ datei << "+G ";}
    Gerade(const Gerade& cop);
    ~Gerade() {datei << "-G ";}
private:
    P2D m_p1;
    P2D m_p2;
};
```

Die Implementierung des Kopier-Konstruktors ist:

```
Gerade::Gerade(const Gerade& cop){
    datei << "+CG ";
    m_p1 = cop.m_p1;
    m_p2 = cop.m_p2;
}
```

Gegeben ist die Funktion `funkt1`:

```
void funk1() {
    double data1[] = { 4.0, 5.0 };
    Punkt<double, 2> p1(data1, 7);
    double data2[] = { 1.0, 1.0 };
    Punkt<double, 2> p2(data1, 5);
    datei << "Bis hier Ausgabe ignorieren\n";
    /*1*/
    Gerade g1(p1, p2);
    datei << endl; /*2*/
    Gerade g2(g1);
    datei << endl; /*3*/
    datei << "Ab hier wieder ignorieren\n";
}
```

Lösungen bitte schrittweise auf diesem Arbeits- oder Extrablatt notieren. Gehen Sie davon aus, dass der Zuweisungsoperator der Klasse `Punkt` nun korrekt implementiert ist und beim Aufruf den String „+Pop=“ gefolgt von der Ausgabe des Attributs `hlp` nach `datei` ausgibt. Die Implementierung des Zuweisungsoperators von `Punkt` enthält zu Beginn somit die beiden Zeilen:

```
hlp = cop.hlp;
datei << "Pop=" << hlp << " ";
```

a.) (ca. 4 P.) Zu welcher Ausgabe nach `datei` führt die Ausführung zwischen `/*1*/` und `/*2*/`?

**Lösung:**

```
+PC5 +PC7 +PC5 +G -P5
```

Beim Aufruf des sonstigen Konstruktors von `Gerade` wird das letzte Argument als Wert und nicht als Referenz übergeben. Sicherlich ist dieses nicht sinnvoll, aber so ist der Code nun einmal. Es wird daher der Kopierkonstruktor der Klasse `Punkt` aufgerufen. Danach werden die Instanzen von `Punkt`, also die Attribute von `Gerade` nach dem Doppelpunkt durch expliziten Aufruf des Kopierkonstruktors initialisiert. `Gerade` setzt sich aus Attributen zusammen. Es werden immer erst die Konstruktoren der Attribute aufgerufen. Hier ist es explizit erfolgt. Erst danach wird der Konstruktor von `Gerade` ausgeführt, d.h. erst dann erfolgt die Ausgabe `+G`. Nachdem der Kopierkonstruktor verlassen wird, wird noch der Destruktor für die per Wert übergebene Instanz von `Punkt` aufgerufen.

Das letzte Argument wird im sonstigen Konstruktor als *call-by-value* übergeben. Hierdurch wird der Kopier-Konstruktor von `Punkt` aufgerufen. Beim Verlassen des Konstruktor wird diese Instanz von `Punkt` dann wieder zerstört. Dadurch werden hier drei Kopier-Konstruktoren von `Punkt` und ein Destruktor aufgerufen plus die Ausgabe im Konstruktor von `Gerade`. Sofern dieses erkannt wurde, gab einen Sonderpunkt.

b.) (ca. 4 P.) Zu welcher Ausgabe nach `datei` führt die Ausführung zwischen `/*2*/` und `/*3*/`?

**Lösung:**

```
+P-1 +P-1 +CG Pop=7 Pop=5
```

Im Kopierkonstruktor gibt es die Initialisierung mit der Doppelpunktschreibweise nicht. `Gerade` setzt sich aus Attributen zusammen. Es werden immer erst die Konstruktoren der Attribute aufgerufen. Hier erfolgt es allerdings implizit, d.h. der Default-Konstruktor von `Punkt` wird entsprechend oft aufgerufen. Erst danach wird der Kopier-Konstruktor von `Gerade` ausgeführt, d.h. erst dann erfolgt die Ausgabe `+CG`. Nun werden die per Default-Konstruktor initialisierten Instanzen von `Punkt` noch mit dem richtigen Wert durch Aufruf des Zuweisungsoperators von `Punkt` initialisiert. Es werden hierbei allerdings keine neuen Instanzen von `Punkt` erzeugt. Deshalb wird beim Verlassen des Kopierkonstruktors von `Gerade` auch kein Destruktor von `Punkt` aufgerufen. Die doppelte Initialisierung der Instanzen von `Punkt` ist jedenfalls überflüssig und nicht effizient. Sie kann durch die Doppelpunktschreibweise, siehe nächste Teilaufgabe, vermieden werden.

c.) (ca. 3 P.) Sie sollten bei der Ausführung zwischen `/*2*/` und `/*3*/` erkannt haben, dass beim Aufruf des Kopier-Konstruktors der Klasse `Gerade` unnötig viele Instanzen der Klasse `Punkt` erzeugt und zerstört werden. Verbessern Sie deshalb die Implementierung des Kopier-Konstruktors.

—— Bitte Extrablatt verwenden. ——

#### Lösung:

Die Instanzen von `Punkt` müssen vor Betreten des Kopier-Konstruktors initialisiert sein, d.h. die Doppelpunkt-Initialisierung ist zu verwenden. Andernfalls werden die beiden Instanzen von `Punkt` durch den Default-Konstruktor von `Punkt` initialisiert. Diese ist nutzlos, weil im Konstruktor von `Gerade` selber nochmals eine Initialisierung erfolgt. Diese kann dann unterbleiben:

```
Gerade::Gerade(const Gerade& cop):  
    m_p1(cop.m_p1), m_p2(cop.m_p2) {  
    datei << "+CG ";  
}
```

d.) (ca. 2 P.) Zu welcher Ausgabe nach `datei` führt die Ausführung zwischen `/*2*/` und `/*3*/` nach Ihrer Verbesserung?

#### Lösung:

```
+PC7 +PC5 +CG
```