Name:

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

Matrikelnummer:	*Punktzahl:*
Ergebnis:	

Freiversuch ☐ *F1* ☐ *F2* ☐ *F3* ☐

Klausur im WS 2025/26:

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 **nicht** 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. Hauptsache es ist erkennbar, welche virtuellen Methoden aufgerufen werden bzw. wie viele Instanzen erzeugt und gelöscht werden.

Hinweis: In den folgenden Programmfragmenten wird die globale Variable `datei` verwendet. Hierfür kann der Einfachheit halber die Variable `cout` angenommen werden. Die Variable `datei` diente 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.

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, die für Ihre Notizen freigelassen wurden, gibt keine Hinweise darauf, wie umfangreich die erwarteten Ausgaben an der betreffenden Stelle sind.

Geplante Punktevergabe

Punktziel	Im einzelnen	Pkte
Übungen:	XXX	
A1: 26 P.		
A2: 14 P.		
A3: 40 P.		
Summe 80+ SP.		

Die Klasse `Point` hat folgende Schnittstelle:

```
class Point  { // instance needs 8 Bytes of memory
public:
   Point(int x, int y) :
      m_x(x), m_y(y) {++cntNow; ++cntAll;}
   Point():m_x(1), m_y(2){cntNow++;++cntAll;}
   Point(const Point& p) :
      m_x(p.m_x), m_y(p.m_y) {
      ++cntNow; ++cntAll;  }
    virtual ~Point() { --cntNow; }
    int GetXY() const { return m_x + m_y; }
   // static methods and members for counting
    static int GetNowCnt() { return cntNow; }
    static int GetAllCreated() { return cntAll; }
    static void PrintDiffs(int befCnt, int befEver);
    static void Reset() { cntAll = 0; cntAll = 0; }
private:
   //! number of currently existing instances
   //! i.e. constructor minus destructor calls
    static int cntNow;
 //! instances created since program start or reset
    static int cntAll;
    int m_x;
    int m_y;
};
```

Im Gegensatz zur Vorlesung erzeugen Konstruktoren und Destruktor keine Ausgaben nach `datei`, sondern es wird gezählt, wie viele Instanzen erzeugt und gelöscht werden. Die zugehörige Quellcode-Datei `Point.cxx` enthält:

```
int Point::cntNow = 0;
int Point::cntAll = 0;
void Point:: PrintDiffs(int befNow, int befAll) {
   datei<<"Mem Leaks: "<< cntNow - befNow <<"\n";
   datei<<"CrEver: " << cntAll - befAll << "\n";
}
```

Ferner haben wir die folgende Klassenhierarchie:

Basisklasse `GeoForm`:

```
class GeoForm {
public:
   GeoForm() : p_point(nullptr) {}
   GeoForm(int dim);
   GeoForm(const GeoForm& c);
    virtual ~GeoForm();
    int CalcLength() { return 0; }
    virtual int CalcArea() { return 0; }
    virtual int GetDim() const { return 0; }
private:
   Point* p_point;
};
```

mit den folgenden Konstruktor- und Destruktor-Definitionen:

```
inline GeoForm::GeoForm(int dim) {
   p_point = new Point[dim];
}
inline GeoForm::~GeoForm() {
   delete[] p_point;
}
```

```
inline GeoForm::GeoForm(const GeoForm& c) {
   if (c.GetDim() > 0) {
      p_point = new Point[c.GetDim()];
      for (int i = 0; i < c.GetDim(); ++i) {
         p_point[i] = c.p_point[i];
      }
   }
   else {
      p_point = nullptr;
   }
}
```

Abgeleitete Klasse `Square`:

```
class Square: public GeoForm {
public:
   Square(int s) : GeoForm(4) {
      m_side = s; }
    virtual ~Square() {};
    int CalcLength() {
      return 4 * m_side; }
    virtual int CalcArea() {
      return m_side * m_side; }
    virtual int GetDim()const { return 4; }
private:
   int m_side;
};
```

Abgeleitete Klasse `Circle`:

```
class Circle: public GeoForm {
public:
    Circle(double r) : GeoForm(1) {
      m_radius = r; }
    virtual ~Circle() {};
    int CalcLength() {
      return 2.0 * PI * m_radius ; }
    virtual int CalcArea() const {
      return m_radius * m_radius * PI;}
    virtual int GetDim() const { return 1; }
private:
   const double PI = 3.14;
   double m_radius;
};
```

Abgeleitete Klasse `Rectangle`:

```
class Rectangle: public GeoForm {
public:
   Rectangle(int len, int wid) : GeoForm(4) {
      m_len = len; m_width = wid; }
    virtual ~Rectangle() {};
    int CalcLength() {
       return 2* (m_len + m_width); }
    virtual int CalcArea() {
       return m_len * m_width; }
    virtual int GetDim()const { return 4; }
private:
   int m_len;
   int m_width;
};
```

Aufgabe 1 : Heap, Stack, Logging, Testen

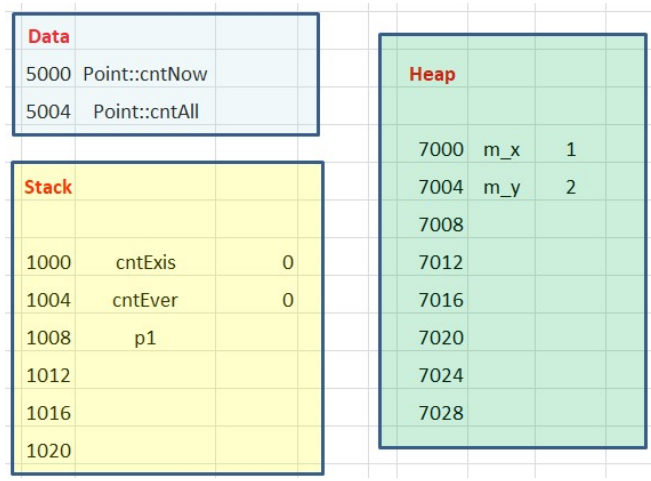
ca. 26 Punkte

In dieser Aufgabe geht es um das grundlegende Verständnis von Stack- und Heapspeicher sowie zum kleinen Teil um das Data-Segment. Im zweiten Teil geht es um ein Verständnis von Konstruktur und Destruktor und darum, dass C++ den Destruktor immer automatisch am Ende des Gültigkeitsbereichs von einer Variablen aufruft. Dieses kann man zur Freigabe von Heap-Speicher nutzen, aber wie in der Vorlesung am Beispiel der Klasse `FunctionLog` gezeigt, auch zur Protokollierung von Funktionsaufrufen oder wie hier zur Zählung der Instanzen einer Klasse. Im dritten Teil soll gezeigt werden, dass Sie verstanden haben, was ein Unit-Test für eine Methode ist. Hier muss die zu testende Funktion geeignet aufgerufen werden und dann das Ergebnis des Aufrufs mit den erwarteten Ergebnis verglichen werden.

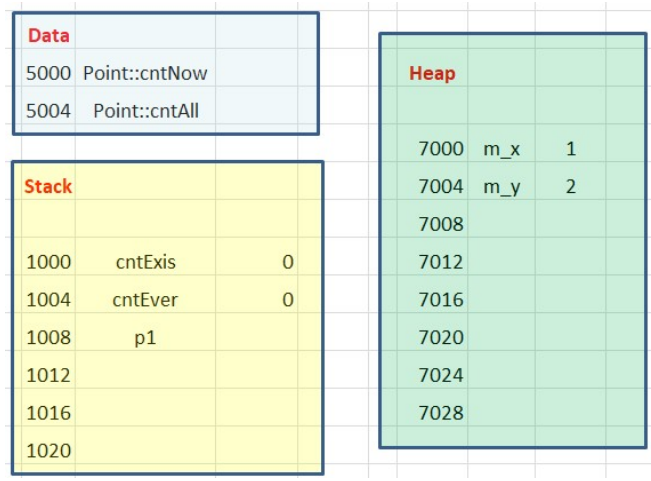
a.) (ca. 12 P.) Vervollständigen Sie die Skizzen der Speicherbelegung auf Daten-, Stack-, und Heapsegment bei Ausführung der folgenden Funktion `Point1` zu den Zeitpunkten /*1*/ und /2*/. `int` und Zeiger belegen jeweils 4 Byte. Verwenden Sie Pfeile zur Veranschaulichung, welcher Zeiger wohin zeigt.

```
void Point1() {
   int cntExis = Point::GetNowCnt();
   int cntEver = Point::GetAllCreated();
   Point* p1 = new Point();
   if (p1 != nullptr) {
      Point p2(4, 16); /*1*/
   }
   int k = 11;
   p1 = new Point(*p1); /*2*/
   delete p1;
   k = 12;
   Point:: PrintDiffs (cntExis, cntEver);
}
```

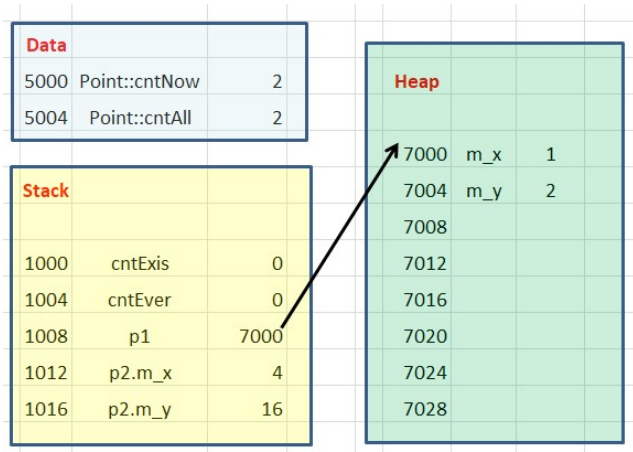
Speicherbelegung zum Zeitpunkt /* 1 */:



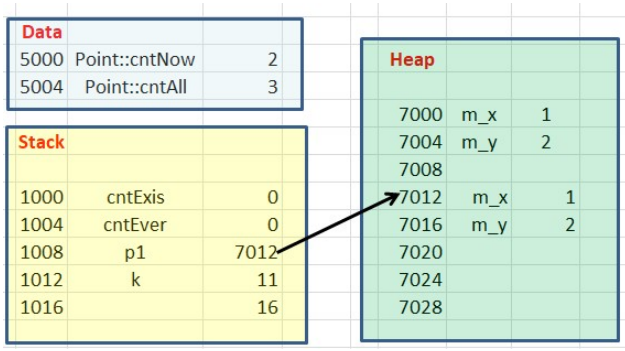
Speicherbelegung zum Zeitpunkt /* 2 */:



Lösung:
Die folgende Abbildung zeigt die Speicherbelegung auf den drei Segmenten zum Zeitpunkt /*1*/.



Und nun die Belegung zum Zeitpunkt /* 2 */:



b.) (ca. 8 P.) Der Aufruf der Funktion `Point1` führt zur folgenden Ausgabe in den Stream `datei`:

```
Mem Leaks: 1
CrEver: 3
```

denn es wurden 3 Instanzen von `Point` erzeugt, aber lediglich zwei wieder freigegeben.

Der Aufruf von `Point::GetNowCnt`, `Point::GetAllCreated` und am Ende von `Point::PrintDiffs` mit den richtigen Parametern ist auf die Dauer etwas umständlich und damit fehleranfällig. Implementieren Sie daher auf einem Extrablatt eine Klasse `PointCounter`, sodass die Überwachung der erzeugten Instanzen von `Point` vereinfacht wird. Die Funktion `Point1` soll wie folgt vereinfacht werden und trotzdem die gleiche Ausgabe nach `datei` erzeugen.

```
void Point1WithLogging() {
   PointCounter counter;
   Point* p1 = new Point();
   if (p1 != nullptr) {
      Point p2(4, 16); /*1*/
   }
   int k = 11;
   p1 = new Point(*p1); /*2*/
   delete p1;
   k = 12;
}
```

Hinweis: `LogTrace` bzw `FunctionLog` aus der Vorlesung. Die Ausgabe bei Aufruf von `Point1WithLogging` wäre dann wiederum:

```
Mem Leaks: 1
CrEver: 3
```

Lösung:

```
class PointCounter {
public:
   inline PointCounter();
   inline virtual ~PointCounter();
private:
   // created minus deleted beginning
   int m_cntSt;
   // all created instances beginning
   int m_everSt;
};
```

```
PointCounter::PointCounter() {
   m_cntSt = Point::GetNowCnt();
   m_everSt = Point::GetAllCreated();
}
PointCounter::~PointCounter(){
   Point:: PrintDiffs (m_cntSt, m_everSt);
}
```

Noch effizienter ist die Attributbelegung mit der Doppelpunkt-Schreibweise:

```
PointCounter::PointCounter() :
   m_cntSt(Point::GetNowCnt()),
   m_everSt(Point::GetAllCreated())
{}
```

c.) (ca. 6 P.) Sie haben sicherlich schon bemerkt, dass für die Methode `CalcArea` der Klasse `Circle` kein dynamisches Binden erfolgt, denn die Schnittstelle ist verschieden von der Deklaration `GeoForm::CalcArea`. Dieses könnte unabsichtlich erfolgt sein. Implementieren Sie auf einem Extrablatt **einen** Test der prüft, ob dynamisches Binden für `CalcArea` für Instanzen von `Circle` erfolgt oder eben nicht.

Lösung:

```
/* Es wird ein Zeiger mit statischem Typ GeoForm
   angelegt und einem Circle zugewiesen. Es wird
   dann über den Zeiger die virtuelle Methode
   CalcArea aufgerufen. Es wird geprüft, ob die
   Methode CalcArea von Circle aufgerufen wird,
   d.h. eine Fläche ungleich 0 ermittelt wird.
   Wenn dieses der Fall ist, wird true geliefert.
*/
bool CircleTest() {
   GeoForm* pc = new Circle(4);
   return pc->CalcArea() > 0;
}
```

Man sollte hier darauf achten, dass man Prüfung auf exakte Werte `double` nicht mit == prüft, sondern prüft, ob die absolute Differenz kleiner als ein Epsilon ist.

Aufgabe 2 : Smart Pointer

ca. 14 Punkte

In dieser Aufgabe geht es nur teilweise um `unique_ptr`. Im ersten Teil soll allerdings erkannt werden, dass man einen konstanten Parameter einer Funktion in der Funktion nicht verändern darf. Im zweiten Teil geht es um die Syntax der Übergabe einer Instanz an einen Werteparameter. Im Bezug auf `unique_ptr` kommt allerdings die Herausforderung hinzu, dass erkannt werden muss, dass man `unique_ptr` nicht als Werteparameter an Funktionen übergeben darf. Man muss die Variable zunächst mit der `move`-Funktion in einen R-Value umwandeln. Der umgewandelte `unique_ptr` wird dadurch ungültig.

Im dritten Teil geht es um die Syntax der Übergabe einer Instanz an einen Referenzparameter. Man muss zeigen, dass man erkannt hat, dass alles was in der Funktion mit dem Referenzparameter passiert, auch mit dem Original in der aufrufenden Funktion erfolgt. Dieses erfolgt hier nicht am Beispiel einer `int`-Variablen, sondern am Beispiel einer Instanz der Klasse `unique_ptr` .

Der vierte Teil beschäftigt sich dann wirklich mit `unique_ptr`. Es wird überprüft, ob verstanden ist, dass `unique_ptr` am Ende ihres Gültigkeitsbereichs im Gegensatz zu normalen C-Pointern den Heap-Speicher ohne explizites Aufruf von `delete` wieder freigeben.

Gegeben sind drei Funktionen, die jeweils `unique_ptr` als Eingangsparameter haben:

```
void Ref(unique_ptr<Point>& ap) { // as ref
   datei << "Ref " << ap->GetXY() << "\n";
   Point* p = new Point(3, 4);
   ap = unique_ptr<Point>(p);
}
```

```
void Val(unique_ptr<Point> ap) { // as value
   datei << "Val " << ap->GetXY() << "\n";
   Point* p = new Point(3, 4);
   ap = unique_ptr<Point>(p);
}
void RefC(const unique_ptr<Point>& ap) {// const ref
   datei << "RefC " << ap->GetXY() << "\n";//2
   Point* p = new Point(3, 4.0);     //3
   ap = unique_ptr<Point>(p); //4
}  //5
```

a.) (ca. 2 P.) In welcher Zeile enthält die Funktion `RefC` einen Syntaxfehler? Begründen Sie Ihre Entscheidung auf einem Extrablatt.

Lösung:

```
void RefC(const unique_ptr<Point>& ap) {// const ref
   datei << "RefC " << ap->GetXY() << " ";
   Point* p = new Point(3, 4);
   // ap ist const und darf deshalb nicht links vom
   // Zuweisungsoperator operator= stehen
   //ap = unique_ptr<Point>(p);
}
```

Zeile 3 wird eine Warnung ergeben, denn 3.0 ist ein `double` und die Methode erwartet einen `int`. Die Frage der Aufgabe bezieht sich allerdings auf Syntaxfehler und nicht auf mögliche Warnungen.

b.) (ca. 4 P.) Rufen Sie im folgenden Beispielcode die Funktion `Val` auf, d.h. ersetzen Sie die Fragezeichen entsprechend. Erklären Sie auf einem Extrablatt, warum es an der gekennzeichneten Stelle nach dem Aufruf ein undefiniertes Programmverhalten gibt.

```
void funcVal() {
   unique_ptr<Point> up(new Point());
   datei << up->GetXY();
   // korrekter Aufruf von Val mit up
   // Val(????  up);
   // warum ist folgende Ausgabe undefiniert?
   datei << up->GetXY();
}
```

Lösung:

```
void funcValCorr() {
   unique_ptr<Point> up(new Point());
   datei << up->GetXY();
   Val(move(up));
   // Durch die Umwandlung von up in einen
   // R-Value durch move wird up verändert.
   // Es enthält nun einen nullptr.
   datei << up->GetXY();
}
```

c.) (ca. 4 P.) Rufen Sie im folgenden Beispielcode die Funktion `Ref` auf, d.h. ersetzen Sie die Fragezeichen entsprechend. Zu welcher Ausgabe nach `datei` führt der Aufruf der Funktion `funcRef`? Beachten Sie, dass in `Ref` selbst auch eine Ausgabe erfolgt.

```
void funcRef() {
   unique_ptr<Point> up(new Point(4, 5));
   datei << up->GetXY();
   // korrekter Aufruf von Val mit up
   Ref(??? up);
   datei << up->GetXY();
}
```

Lösung:

```
void funcRefRun() {
   unique_ptr<Point> up(new Point(4,5));
   datei << up->GetXY() << "\n";
   Ref(up);
   datei << up->GetXY();
}
```

Ausgabe ist:

```
9
Ref 9
7
```

d.) (ca. 4 P.) Zu welcher Ausgabe nach `datei` führt der Aufruf von `UseUniquePtr`? Es geht also ausschließlich darum, wie viele Instanzen von `Point` erzeugt und wieder freigegeben werden. Die Klasse `PointCounter` wurde von Ihnen bereits in der Aufgabe 1 implementiert. Sofern dieses noch nicht erfolgt ist, hier ein Hinweis. Die Instanz der Klasse gibt am Ende eines Blocks aus, wie viele Instanzen von `Point` angelegt, aber nicht wieder freigegeben wurden und wie viele Instanzen von `Point` insgesamt in dem Block (hier Funktion) erzeugt wurden.

```
void UseUniquePtr() {
   PointCounter counter;
   unique_ptr< Circle > pc = make_unique<Circle>(5.0);
   unique_ptr<Square> ps(new Square(3));
   GeoForm* pgf = new Circle(7.1);
   pgf = new Rectangle(44, 1);
}
```

Lösung:

```
Mem Leaks: 5
CrEver: 10
```

Es wird ein `Circle` erzeugt. Dabei wird eine Instanz von `Point` erzeugt und da `pc` ein `unique_ptr` ist, wird es über den Destruktor des `unique_ptr` auch wieder freigegeben. Entsprechendes gilt für die Instanz von `Square`. Hier werden 4 Instanzen von `Point` erzeugt und wieder freigegeben. Die beiden Instanzen von `Circle` und `Rectangle`, die im Anschluss erzeugt werden, werden nicht mehr freigegeben. Hierbei wird eine bzw. vier Instanzen von `Point` erzeugt. Insgesamt werden also 10 Instanzen von `Point` erzeugt, aber fünf werden nicht wieder freigegeben.

Aufgabe 3 : Polymorphie und Instanzerzeugung

ca. 40 Punkte

In dieser Aufgabe geht es darum zu verstehen, wann Instanzen einer Klasse erzeugt und wieder freigegeben werden. Außerdem gibt es um Vererbung sowie um Polymorphie.

Im ersten Teil wird eine Instanz einer Unterklasse als Referenz an eine Variable der Oberklasse übergeben. Hierbei wird keine Kopie erzeugt. Außerdem ist der dynamische Typ der Referenz immer noch vom Typ der Unterklasse. Für virtuelle Funktion erfolgt deshalb dynamisches Binden, sofern diese die gleiche Signatur wie die entsprechenden Methode in der Basisklasse aufweisen.

Im zweiten Teil ist zu erkennen, dass Zeiger keine Instanzen einer Klasse sind, d.h. bei der Definitionen von Zeigern wird niemals ein Konstruktor der Klasse selbst aufgerufen.

Der dritte Aufgabenteil ist die komplexeste Aufgabe der Klausur. Hier sollen verschiedenen Wissenselemente kombiniert werden. Ist der Unterschied zwischen Zeiger und Werten bekannt. Ist verstanden, wie oft eine Schleife durchlaufen wird. Ist verstanden, von welchem dynamischen Typ eine Instanz einer Variablen ist. Hiermit kann ermittelt werden, welche konkrete Methode einer als virtuell vereinbarten Methode aufgerufen wird. Wurde verstanden, dass bei der Übergabe von Instanzen einer Variablen an einen Werteparameter eine Kopie erzeugt wird und damit der Kopierkonstruktor der Klasse aufgerufen wird.

a.) (ca. 5,5 P.) Zu welcher Ausgabe nach `datei` führt der Aufruf der folgenden Funktion `Poly1`?

Achtung: Damit Sie nicht unnötig Zeit mit Kopfrechnen verbringen, sind Umfang und Fläche jeweils angegeben. Ob die Methoden aber wirklich aufgerufen werden oder ob nicht ggf. die Methoden der Basisklassen aufgerufen werden, müssen Sie schon selber entscheiden. Achten Sie auch auf `double` bzw. int. Die Klasse `PointCounter` wurde von Ihnen bereits in der Aufgabe 1 implementiert. Sofern dieses noch nicht erfolgt ist, hier ein Hinweis. Die Instanz der Klasse gibt am Ende eines Blocks aus, wie viele Instanzen von `Point` angelegt, aber nicht wieder freigegeben wurden und wie viele Instanzen von `Point` insgesamt in dem Block (hier Funktion `Poly1`) erzeugt wurden.

```
void Poly1() {
   PointCounter counter;
   Circle c1(3.0);
   Square s1(3);
   Rectangle r1(4, 1);
   // Umf : 18.84, Flaeche: 28.26
   PrintRef(c1);
   // Umf : 12, Flaeche: 9
   PrintRef(s1);
   // Umf : 10, Flaeche: 4
   PrintRef(r1);
}
```

Die Funktion `PrintRef` ist wie folgt definiert:

```
inline void PrintRef(GeoForm& g) {
   datei << "l=" << g.CalcLength()
      << " a=" << g.CalcArea() << "\n";
}
```

Beachten Sie bei Ihrer Antwort zwei Dinge: Welche der Methoden `CalcLength` und `CalcArea` werden aufgerufen und wie viel Instanzen von `Point` werden bei der Ausführung von `Poly1` erzeugt? Letzteres wird durch die Instanz von `PointCounter` protokolliert.Die abgeleiteten Instanzen von `GeoForm` erzeugen (meist) Instanzen von `Punkt`.

Lösung:

```
l=0 a=0
l=0 a=9
l=0 a=4
Mem Leaks: 0
CrEver: 9
```

b.) (ca. 3 P.) Zu welcher Ausgabe nach `datei` führt der Aufruf der Funktion `Poly2`?

```
void Poly2() {
   PointCounter counter;
   Square* ps[5];
   GeoForm* pg[3];
   pg[2] = ps[1];
}
```

Lösung:

```
Mem Leaks: 0
CrEver: 0
```

c.) (ca. 20 P.) Welche Ausgabe (nach `datei`) ergibt der Aufruf der folgenden Funktion `Poly3` ?

```
void Poly3() {
   Point::Reset(); // Zähler auf 0 setzen
   PointCounter counter;
   Square s(3); // Umf : 12, Flaeche: 9
   Rectangle r(4, 1); // Umf : 10, Flaeche: 4
   Square* ps = new Square(7); // Umf : 28, Fl.: 49
   // Umf : 4, Flaeche: 1
   Rectangle* pr = new Rectangle(1, 1);
   datei << "before g[4] "
      << Point::GetAllCreated() << "\n";/*0*/
   GeoForm* g[4] = { &s, &r, ps, pr }; /*1*/
   datei << "before PrintValue "
      << Point::GetAllCreated() << "\n";
   for (int j = 0; j < 3; j++) {
      PrintValue(*(g[j]));       /*2*/
   }
   datei << "before PrintRef "
      << Point::GetAllCreated() << "\n";
   for (int j = 0; j < 3; ++j) {
      PrintRef(*(g[j]));         /*3*/
   }
   datei << "After PrintRef "
      << Point::GetAllCreated() << "\n";
}// counter erzeugt hier Ausgabe nach Datei
```

`PrintRef` kennen Sie schon von oben. `PrintValue` ist wie folgt definiert:

```
inline void PrintValue(GeoForm g) {
   datei << "l=" << g.CalcLength()
      << " a=" << g.CalcArea() << "\n";
}
```

Es geht bei den Ausgaben mit `Point::GetAllCreated` also darum, herauszufinden wie viele weitere Instanzen von `Punkt` jeweils erzeugt werden. Außerdem erzeugen die `PrintRef`- und `PrintValue`-Aufrufe in den for-Schleifen noch Ausgaben nach `datei`. Kommentieren Sie Ihre Überlegungen, damit ich Ihre Ideen und mögliche Fehler nachvollziehen kann. Einige Texte sind schon vorgedruckt. Die Anzahl der Aufrufe von `PrintRef`- und `PrintValue` müssen Sie allerdings selber ermitteln.

Lösung:

```
before g[4]  16
before PrintValue  16
l=0 a=0
l=0 a=0
l=0 a=0
before PrintRef  28
l=0 a=9
l=0 a=4
l=0 a=49
After PrintRef  28
Mem Leaks: 8
CrEver: 28
```

Es werden jeweils zwei Instanzen von `Square` und `Rectangle` erzeugt, jeweils eine auf dem Stack und eine auf dem Heap. Hierbei wurden 16 Instanzen von `Point` erzeugt. Die Zeiger auf diese Instanzen werden im Array `g4` abgelegt. Hierbei werden nur Adresse kopiert. Deshalb ist die Ausgabe von `Point::GetAllCreated` im Anschluss die gleiche. Die erste j-Schleife wird dreimal mit den Werten von 1 bis 3 durchlaufen. Die Übergabe an `PrintValue` erfolgt per Wert, d.h. es wird der Kopierkonstruktor von `GeoForm` aufgerufen. Der Kopierkonstruktur ruft allerdings die virtuelle Methode `GetDim` auf. Da das Argument `c` vom Kopierkonstruktor als Referenz übergeben wurde, findet Polymorphie statt, d.h. es wird `GetDim` von der ursprünglichen Klasse aufgerufen. Daher werden hier nochmals 3mal 4 Instanzen von `Point` erzeugt. Deshalb ist der nächste Aufruf von `Point::GetAllCreated` um 12 höher. in der Schleife wird 3mal `GeoForm::CalcLength` und 3mal `GeoForm::CalcArea` aufgerufen. Das erfolgt unabhängig von Polymorphie.denn dynamischer und statischer Typ sind gleich `GeoForm`.
Jetzt sind wir schon bei 28 Instanzen. Beim Verlassen der Methode wird `GeoForm` wieder zerstört und im Destruktor `GeoForm` wird auch `Point` wieder freigegeben, d.h. 12 mal wurde der Destruktor von `Point` auf gerufen.
Im der zweiten j-Schleife wird `PrintRef` aufgerufen. `GeoForm` wird als Referenz übergeben, d.h. es wird kein Kopierkonstruktor aufgerufen. Die Anzahl der erzeugten Instanzen von `Point` erhöht sich also nicht. `Point::GetAllCreated` hat nach dieser Schleife die gleiche Ausgabe wie zuvor. Die j-Schleife wird natürlich auch dreimal durchlaufen und zwar wiederum von 1 bis 3. Hier erfolgt nun dynamischen Binden für alle virtuellen Methoden, d.h. für `PrintArea`.
Am Ende der Funktion werden noch die beiden Instanzen von `Square` und `Rectangle` auf dem Stack wieder freigegeben. Die Instanzen auf dem Heap bleiben bestehen und damit auch 8 Instanzen von `Point`. Wir haben also ein Memory Leak von 8 Instanzen von `Point`.

d.) (ca. 12 P.) Wie ändert sich die Ausgabe, wenn die Zeile /* 1 */

```
GeoForm* g[4] = { &s, &r, ps, pr };
```

ersetzt wird durch:

```
GeoForm g[4] = { s, r, *ps, *pr };?
```

Welche Ausgabe erzeugt also der Aufruf von `Poly3b` nach `datei`? Geben Sie die Ausgaben nochmals vollständig an. Bei den Begründungen genügt es, wenn Sie die Unterschiede erklären.

```
void Poly3b() {
   Point::Reset(); // Zähler auf 0 setzen
   PointCounter counter;
   Square s(3); // Umf : 12, Flaeche: 9
   Rectangle r(4, 1); // Umf : 10, Flaeche: 4
   Square* ps = new Square(7); // Umf : 28, Fl.: 49
   // Umf : 4, Flaeche: 1
   Rectangle* pr = new Rectangle(1, 1);
   datei << "before g[4] "
      << Point::GetAllCreated() << "\n";
   GeoForm g[4] = { s, r, *ps, *pr }; /*1*/
   datei << "before PrintValue "
      << Point::GetAllCreated() << "\n";
   for (int j = 0; j < 3; j++) {
      PrintValue(g[j]);       /*2*/
   }
   datei << "before PrintRef "
      << Point::GetAllCreated() << "\n";
   for (int j = 0; j < 3; ++j) {
      PrintRef(g[j]);         /*3*/
   }
   datei << "After PrintRef "
      << Point::GetAllCreated() << "\n";
}// counter erzeugt hier Ausgabe nach Datei
```

Lösung:

```
before g[4] 16
before PrintValue 32
l=0 a=0
l=0 a=0
l=0 a=0
before PrintRef 32
l=0 a=0
l=0 a=0
l=0 a=0
After PrintRef 32
Mem Leaks: 8
CrEver: 32
```

Genau wie vorher werden jeweils zwei Instanzen von `Square` und `Rectangle` erzeugt, jeweils eine auf dem Stack und eine auf dem Heap. Hierbei wurden also wie bisher16 Instanzen von `Point` erzeugt. In der neuen Zeilen mit g[4] werden nun keine Zeiger erzeugt, sondern Instanzen von `GeoForm`. Es wird der Kopierkonstruktor von `GeoForm` aufgerufen. Der Kopierkonstruktur ruft allerdings die virtuelle Methode `GetDim` auf. Da das Argument `c` vom Kopierkonstruktor als Referenz übergeben wurde, findet Polymorphie statt, d.h. es wird `GetDim` von der ursprünglichen Klasse aufgerufen. Daher werden hier nochmals 4mal 4 Instanzen von `Point` erzeugt. Deshalb ist der nächste Aufruf von `Point::GetAllCreated` um 16 höher.
Beide Schleifen laufen wiederum jeweils 3mal. Beim Aufruf von `PrintValue` wird zwar der Kopierkonstruktor von `GeoForm` aufgerufen. Beim Aufruf liegen aber bereits Instanzen von `GeoForm` vor, sodass `GetDim` hier 0 zurückliefert und somit keine neuen Instanzen von Punkt erzeugt werden.
In beiden Schleifen liegen nur noch Instanzen von `GeoForm` vor. Es ist somit gleichgültig, ob nun Polymorphie erfolgt oder nicht. Nur `GeoForm::CalcLength` und `GeoForm::CalcArea` werden aufgerufen, und es wird jeweils 0 ausgegeben.
Am Ende wurden 32 Instanzen von `Point erzeugt`. Die 8 auf dem Heap werden nicht zerstört.