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++

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	Sonderpunkte	erreicht
Übungen:	XXX	
A1: 26 P.		
A2: 14 P.		
A3: 40 P.		
Summe 80+ SP.		

V.1

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;
};
```

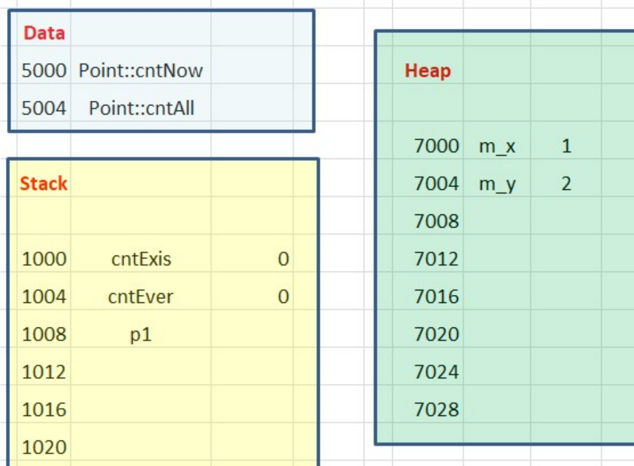
Aufgabe 1 : Heap, Stack, Logging, Testen

ca. 26 Punkte

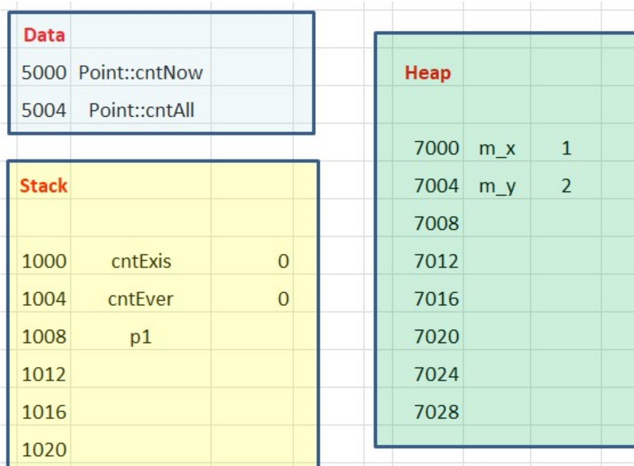
a.) (ca. 12 P.) Vervollständigen Sie die Skizzen der Speicherbelegung auf Daten-, Stack-, und Heapsegment bei Ausführung der folgenden Funktion `Point1` zu den Zeitpunkten /*1*/ und /*2*/. `int` und Zeiger belegen jeweils 4 Byte. Verwenden Sie Pfeile zur Veranschaulichung, welcher Zeiger wohin zeigt.

```
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 */:



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
```

——— Bitte Extrablatt verwenden.——————

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.
——— Bitte Extrablatt verwenden.——————

Aufgabe 2 : Smart Pointer

ca. 14 Punkte

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.

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();
}
```

——— Bitte Extrablatt verwenden.———————

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();
}
```

——— Bitte Extrablatt verwenden.———————

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);
}
```

```
 --- Bitte hier vervollstaendigen ---
Mem Leaks:

CrEver:
-
```

Aufgabe 3 : Polymorphie und Instanzerzeugung

ca. 40 Punkte

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`.

```
 --- Bitte hier vervollstaendigen ---
l=      a=

l=      a=

l=      a=

Mem Leaks:

CrEver:

-
```

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];
}
```

```
 --- Bitte hier vervollstaendigen ---
Mem Leaks:

CrEver:
-
```

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.

```
--- Bitte hier vervollstaendigen ---
before g[4]
before PrintValue
l=      a=      (wie oft wird Schleife durchlaufen?)





before PrintRef
l=      a=      (wie oft wird Schleife durchlaufen?)





After PrintRef
Mem Leaks:

CrEver:
-
```

```
--- Bitte hier vervollstaendigen ---
before g[4]
before PrintValue
l=      a=      (wie oft wird Schleife durchlaufen?)





before PrintRef
l=      a=      (wie oft wird Schleife durchlaufen?)





After PrintRef
Mem Leaks:

CrEver:
-
```

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.

```cpp
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
```