DAP2 Praktikum für ETIT und IKT, SoSe 2018, Langaufgabe L2

Fällig am 28.05. um 14:15

Es gelten die üblichen Programmierregeln in der gewohnten Härte.

Beachten Sie heute zusätzlich:

- Die unten abgedruckten Konstruktoren beinhalten noch keine Sicherungsmaßnahmen (z.B. bei fehlgeschlagener Beschaffung von dynamischen Speicher). Sie sind dafür verantwortlich, diese zu implementieren.
- Klassen dürfen einen `operator<<` in der Form

  ```
  friend ostream& operator<<(ostream &TheOstream, MeinKlassenName &me)
  ```

  bereitstellen, mit der eine elegante Ausgabemöglichkeit nach `cout` bereitgestellt werden kann.

Langaufgabe L2.1 (1 Punkt)

- Legen Sie eine Klasse `Point` durch Vererben von `vector<double>` an, die einen Punkt im Raum beschreibt. Die Koordinaten sind vom Typ `double`.
- Die Klasse `Point` soll mit den im nachfolgend gezeigten Codesegment gezeigten Konstruktoren ausgestattet sein. Hier, wie auch bei allen weiter unten gezeigten Klassen, ist eine Modifikation der Aufrufparameter der Konstruktoren oder eine Erweiterung um zusätzliche Konstruktoren nicht zulässig. Zudem sind, wie auch in den anderen Codesegmenten, für das Überprüfen der erfolgreichen Speicherzuweisung und anderer Sicherheitsmaßnahmen eventuell noch Ergänzungen innerhalb des Rumpfes der Konstruktoren notwendig.

```
#include <iostream>
#include <string>
#include <vector>
#include <assert.h>
#include <math.h>
#include <cstdlib>
#include <time.h>
#include <sstream>
#include <cstring>
#include <initializer_list> // Important!

using namespace std;

class Point : public vector<double> {
public:

      Point(initializer_list<double> args) {
            for(auto iter = args.begin(); iter != args.end(); ++iter)
                  push_back(*iter);
      }

      // Alle anderen Methoden Ihrer Wahl fehlen noch!
};
```

Beachtn Sie die Verwendung des Iterators. Diese Konstruktoren erlauben es die in C++11 mögliche Erzeugung von `Points` mit variabler Anzahl Koordinaten direkt im Konstruktor. Beispiele:

```
Point Eins=Point{1.1};        // Punkt im 1-dimensionalen-Raum
Point Zwei=Point{1.1,2.2};    // Punkt im 2-dimesionalen-Raum
```

- Punkte im Raum sollen im der Klasse `PointArray` gespeichert werden können. `PointArray` darf nicht unkontrolliert alle Methoden `vector<Point>` erben, da man ansonsten mit `vector<Point>::push_back` Punkte verschiedener Dimension unkontrolliert zusammen bauen kann. Sie dürfen eine neue `push_back` implementieren, die die unerlaubten Kombinationen mit durch das Werfen von Exeptions verhindert. Das nachfolgende Codesegment beinhaltet die zulässigen Konstruktoren von `PointArray`

```
class PointArray : private vector<Point> {
public:
      using vector<Point>::size;
      using vector<Point>::operator[];

      PointArray(initializer_list<Point> args) {
            for (auto iter = args.begin(); iter!=args.end();++iter)
                  push_back(*iter);

      // Alle anderen Methoden Ihrer Wahl fehlen noch!
};
```

Stellen Sie zudem durch das Erzeugen von Exeptions sicher, dass alle Punkte eines `PointArray` immer die gleiche Dimension aufweisen. Also:

```
PointArray Beide =PointArray{Zwei,Point{3.3;4.4} }; // Geht
PointArray Feinde=PointArray{Eins,Zwei}; // Verhindern!
```

Die Zugriffe auf die gespeicherten Informationen geschehen wie folgt:

```
Point ErsterPunkt = Beide[0];
double VierKommaVier = Beide[1][1];
```

- Vererben Sie aus `PointArray` die Klasse `Simplex`. Ein `Simplex` der Dimension *n* hat immer *n+1* viele Punkte.

```
class Simplex : private PointArray {

public:
      using PointArray::size;
      using PointArray::operator[];

      Simplex(const PointArray &ThePointArray)
            :PointArray(ThePointArray)
            {;}

      // Alle anderen Methoden Ihrer Wahl fehlen noch!

};
```

Weitere Konstruktoren sind nicht zulässig. `Simplex` erbt privat Methoden von `PointArray`, damit man nicht z.B. durch `PointArray::push_back` nachträglich

den `Simplex` vergrößern kann. Stellen sie durch eine Exeption sicher, dass Dimension des `Simplex` zur Anzahl seiner Punkte passt. Also:

```
Simplex DerGeht(PointArray{Point{1,2},Point{5,6},Point{8,9}});
Simplex DerGehtNicht(PointArray{Point{1,2},Point{5,6}});
// Letzteres muss verhindert werden!
```

Ebenfalls darf nicht funktionieren (bitte testen!):

```
Simplex Probe = DerGeht;
Probe.push_back(Point{ 1, 2 } ); // Darf nicht funktionieren!
```

- Vererben Sie aus `Simplex` die Klasse `Triangle` , die drei Punkte im 2-dimensionalen Raum enthält.

```
class Triangle : public Simplex {

public:
      Triangle(const Simplex &TheSimplex):Simplex(TheSimplex){;}

      // Alle anderen Methoden Ihrer Wahl fehlen noch!
};
```

Weitere Konstruktoren sind nicht zulässig. Stellen sie durch eine Exeption sicher, dass die Dimension des `Triangle` zur Anzahl seiner Punkte passt. Also:

```
Triangle Dreieck(DerGeht);
Triangle ZweiEck(Beide); // Vrhindern!
PointArray DreiPunkteIm1DRaum =
      PointArray{Point{1},Point{2},Point{3} };

Triangle Eindimensional(DreiPunkteIm1DRaum);
      // Verhindern!
```

- Schreiben Sie eine Methode für `Point` mit

```
public double EuclidDistanceTo(const Point &Other)
```

welche den Euklidischen Abstand zu einem anderen Punkt `Other` berechnet.

- Schreiben Sie für die Klasse `Triangle` eine Methode

```
public double Girth()
```

die die Summe der Seitenlängen des Dreiecks zurückgibt.

- Schreiben Sie Programm `dr [ x1 y1 x2 y2 x3 y3 ]` das folgende Eigenschaften hat:
  - Wird es mit sechs Fließkommazahlen als Parameter aufgerufen, so soll es aus diesen drei Koordinaten ein Dreieck aufbauen.
  - Wird es ohne Parameter aufgerufen, so soll es bei jedem Aufruf ein neues zufälliges Dreieck aufbauen.
  - Bei anderen Eingaben erfolgt eine sinnvolle Fehlermeldung.

- Es soll jeweils den Umfang des Dreiecks (Länge aller drei Seitenlinien) ausgeben.

Beispiele:

```
>dr
Triangle ( (0.33,0.62)(0.68,0.57)(0.26,0.85) ) has Girth 1.08
>dr 1
Usage: dr [ x1 y1 x2 y2 x3 y3 ]
>dr 0 0 1 0 0 1
Triangle ( (0,0)(1,0)(0,1) ) has Girth 3.41421
```

Langaufgabe L2.2 (1 Punkt)

- Schreiben Sie ein Programm `ConvexHull`, das aus einer vorgegeben Menge von Punkten im zweidimensionalen Raum die dazugehörige konvexe Hülle berechnet. Verwenden Sie als Ausgangspunkt dazu die einfache iterative Methode, die in der Vorlesung als `SimpleConvexHull` eingeführt wurde.
- Wird `ConvexHull` nur einem Integer-Wert `n` aufgerufen wird, erzeugt es `n` zufällige zweidimensionale Punkte vom Typ `Point`. Die erzeugten Punkte sollen sich von Programmaufruf zu Programmaufruf unterscheiden.
- Wird es mit einer geraden Anzahl von Zahlen aufgerufen, so werden hieraus Punkte vom Typ `Point` erzeugt und in `AllPoints` vom Typ `PointArray` abgelegt. Dabei werden jeweils zwei Zahlen der Eingabe für einen Punkt in der Reihenfolge x_1, y_1, x_2, y_2, … verwendet.
- Unsinnige oder fehlerhafte Eingaben werden dem Benutzer gemeldet und das Programm wird beendet.
- Danach berechnet das Programm die konvexe Hülle mit Hilfe der von Ihnen zu implementierenden Funktion

  ```
  PointArray CalculateHull(const PointArray &AllPoints)
  ```

  Die zurückgegebenen Punkte beinhalten die konvexe Hülle aufsteigend im Uhrzeigersinn mit der minimalen Anzahl von Punkten, die die konvexe Hülle beschreiben und das ohne Dubletten oder anderen Unschönheiten. Dazu muss die aus der Vorlesung bekannte Methode entsprechend ergänzt werden.
- Es kann für Sie sinnvoll sein, eine Klasse `Line` zu definieren, die genau zwei zweidimensionale Punkte vom Typ `Point` enthält. Ein (unvollständiges) Gerüst findet sich dazu hier:

```
class Line : private PointArray {

public:
      using PointArray::operator[];

      Line(const Point &P1, const Point &P2) :PointArray({P1,P2}){;}

      double Length() { return ((*this)[0] - (*this)[1]); }

      bool ThisfPointLeftOfMe(const Point &To) {

      // Nutzt Hessesche Normalenform der Geradengleichung aus. Siehe
      // http://de.wikipedia.org/wiki/Hessesche_Normalform#Abstand
```

```
            Point S0=Point{(*this)[0][0]-To[0],(*this)[0][1]-To[1]};
            Point S1=Point{(*this)[1][0]-To[0],(*this)[1][1]-To[1]};

            double ax = S1[0] - S0[0];
            double ay = S1[1] - S0[1];
            double d = ay*S0[0] - ax*S0[1];
            return d <= 0.0;
      }

};
```

Als kleinen Bonus enthält diese Klasse bereits die Methode `ThisfPointLeftOfMe` die man eventuell braucht. Bitte denken Sie daran, unzulässige Linien in den Konstruktoren durch eine Exeption abzuwehren.

- Das Programm gibt danach die konvexe Hülle aus. Ihr Programm sollte sich wie folgt verhalten:

```
>ConvexHull
Usage: ConvexHull  n | { x1 y1 x2 y2 x3 y3 ... }

>ConvexHull 1 2 3
Usage: ConvexHull  n | { x1 y1 x2 y2 x3 y3 ... }

>ConvexHull -1
Parameter n must be positive Integer.

>ConvexHull -1 2
Set of 1 Points is:
 ( (-1,2) )

Hull needs at least 2 Points to build

>ConvexHull -1 -1 1 1 0 0 -1 1 1 -1
Set of 5 Points is:
(-1,-1)
(1,1)
(0,0)
(-1,1)
(1,-1)

Convex Hull is build clockwise form the 4 following Points:
(-1,-1)
(-1,1)
(1,1)
(1,-1)

> ConvexHull 7
Set of 7 Points is:
(0.204634,-0.879945)
(-0.858626,-0.392731)
(0.0141722,-0.275384)
(-0.217833,-0.559116)
(0.700964,-0.089918)
(0.746785,0.56155)
(0.405626,0.859462)

Convex Hull is build clockwise form the 5 following Points:
(-0.858626,-0.392731)
(0.405626,0.859462)
(0.746785,0.56155)
(0.700964,-0.089918)
(0.204634,-0.879945)
```