Lösung Klausur 4 infT

Programmieren in C++

Aufgabe 1 (Operator Overloading und Komposition)
Gegeben sind die Dateien *main.cpp*, *Complex.cpp*, *Fraction.cpp* und *FracComplex.cpp*, die sich kompilieren und rudimentär ausführen lassen.

a) (5 Pkte.)
Ergänzen Sie die Klasse **`Complex`** mit den nötigen überladenen Operatoren, so dass die in *main.cpp* auskommentierten Zeilen ebenfalls kompiliert und ausgeführt werden können. Der erwartete erweiterte Output sieht wie folgt aus:

```
c13 = c6          = 5 - 2*i
c14 = c6          = 5 - 2*i
c15 = -3 + 2*i    = -3 + 2*i
c16 = -21*i       = -21*i
c17 = -i + 11     = 11 - i
c18 = i * 3 - 4 = -4 + 3*i
```

Punkte:	
- (unärer Operator)	1 P
-, +, * (binäre Operatoren)	3 P
i als komplexe Konstante	1 P

Hinweise: Realisieren Sie die Operatoren für Reflexivität. Beachten Sie **`const`**ness.

b) (5 Pkte.)
Realisieren Sie in der Datei *Fraction.cpp* die Klasse **`Fraction`**. Diese Klasse repräsentiert gebrochene Zahlen. Folgender Output wird erwartet:

```
f1(0, 7)               = 0
f2(3, 1)               = 3
f3(3, -1)              = -3
f4(1, -1)              = -1
f5(-1, 2)              = -1/2
f6(1, -2)              = -1/2
f7 = 0                 = 0
f8 = Fraction(-1)/5    = -1/5
f9 = -1/Fraction(5)    = -1/5
f10 = (Fraction)-1/5 = -1/5
f11 = -1/5             = 0
f12(5)                 = 5
f13 = -1/f12           = -1/5
f14 = f12/f13          = -25
```

Punkte:	
Zähler und Nenner	1 P
Konstruktor	1 P
/ -Operator	1 P
<< -Operator	2 P

Hinweise:
Stellen Sie an Funktionalität nur zur Verfügung, was Sie für obigen Output auch tatsächlich brauchen. Die Situation mit einem Nenner gleich 0 wird hier nicht betrachtet. Beachten Sie wiederum **`const`**ness.

c) (7 Pkte.)
Realisieren Sie in der Datei *FracComplex.cpp* die Klasse **`FracComplex`**. Diese Klasse repräsentiert wiederum komplexe Zahlen, allerdings sind Real- und Imaginärteil nun vom Typ **`Fraction`**.

Punkte:	
Konstruktor	1 P
<< Operator	1 P
==, !=, >, < in Fraction	4 P
abs in Fraction	1 P

Folgender Output wird erwartet:

```
fc1(Fraction(1, 5), Fraction(-3, 8)) = 1/5 - 3/8*i
fc2(Fraction(15), Fraction(-3))      = 15 - 3*i
fc3(15, -3)                          = 15 - 3*i
```

Hinweise:
Konstruktor und **`operator<<`** der Klasse **`FracComplex`** sind analog aufgebaut zur Klasse **`Complex`**. Es lohnt sich, zuerst diese zu implementieren. Auf Grund erhaltener Fehlermeldungen nehmen Sie jetzt nötige Ergänzungen in der Klasse **`Fraction`** vor.

Aufgabe 2 (Vererbung und Polymorphismus)
Gegeben ist die Datei *geomFig.cpp* und die erwartete Ausgabe (vgl. unten).

a) (1 Pkt.)
Korrigieren Sie den Konstruktor in **`Circle`**.

b) (1 Pkt.)
Schreiben Sie die fehlende Definition der Funktion **`print()`** der Klasse **`Circle`**. Die Funktion gibt den Mittelpunkt und den Radius eines **`Circle`** aus.

c) (4 Pkte.)
Leiten Sie von der Klasse **`Shape`** die Klasse **`Rectangle`** und von dieser die Klasse **`Square`** ab. Der Konstruktor für **`Rectangle`** erhalte als Argumente den Koordinatenbezugspunkt in der linken oberen Ecke (positive x-Achse nach rechts, positive y-Achse nach unten) sowie Höhe und Breite. Nutzen Sie geerbte Instanzvariablen. Definieren Sie einen angepassten Konstruktor für **`Square`**. Definieren Sie für beide Klassen eine **`print`**-Funktion.

L:

```
class Rectangle : public Shape {
   double height, width;
public:
  Rectangle(double x, double y, double h, double w)
  : Shape(x,y), height(h), width(w) {
    cout << "Rectangle() called" << endl;
  }

  ~Rectangle()  { cout << "~Rectangle() called" << endl; }

   double getHeight() const { return height; }

   void print() const {
     cout <<         "Rectangle: Ursprung: ";
     Shape::print();
     cout << endl << "           Hoehe: " << height << ", Breite: " << width
<< endl;
  }
};


class Square : public Rectangle {
 public:
   Square( double x, double y, double side) : Rectangle(x, y, side, side) {
    cout << "Square() called" << endl;
  }

  ~Square()  { cout << "~Square() called" << endl; }

   void print() const {
     cout <<         "Square: Ursprung: ";
     Shape::print();
     cout << endl << "        Seite: " << Rectangle::getHeight() << endl;
  }
};
```

1

1

1

1

d) (5 Pkte.)
Realisieren Sie ein **`main`** mit einem Array, das drei Zeiger auf **`Shape`**s aufweist. Die Zeiger sollen auf einen **`Circle`**, einen **`Rectangle`** und einen **`Square`** zeigen, die alle auf dem Heap angelegt wurden.
Iterieren Sie über das Array und senden Sie via Zeiger die Funktion **`print`** an die **`Shape`**s (ohne eigene Casts!). Nehmen Sie ggf. nötige Änderungen an **`print`** vor. Zerstören Sie am Schluss alle Objekte. vom höchsten zum niedrigsten Index. Nehmen Sie auch hier nötige Ergänzungen in den Klassen vor.

Erwartete Ausgabe:

```
Shape() called
Circle() called
Shape() called
Rectangle() called
Shape() called
Rectangle() called
Square() called

Circle: Mittelpunkt: (1, 2)
        Radius: 10

Rectangle: Ursprung: (10, 10)
           Hoehe: 2, Breite: 6

Square: Ursprung: (3, 3)
        Seite: 7

~Square() called
~Rectangle() called
~Shape() called
~Rectangle() called
~Shape() called
~Circle() called
~Shape() called
```

L:

```
int main() {
  Shape *shapes[] = {
    new Circle(1, 2, 10), new Rectangle(10, 10, 2, 6), new Square(3, 3, 7)
  };
  cout << endl;

  for (int i = 0; i < sizeof shapes / sizeof(Shape *); ++i) {
    shapes[i]->print();
    cout << endl;
  }

  for (int i = sizeof shapes / sizeof(Shape *) - 1; i >= 0 ; --i) {
    delete(shapes[i]);
    shapes[i] = 0;
  }
}
```

Punkte:	
Array und Initialisierung	1 P
virtual print	1 P
virtual Destructors	1 P
Ausgabe Konstr./Destr.	1 P
Cleanup	1 P