C/C++ Programmierung
Übungsblatt 9

Florian Oetke

1 Klassenhierarchie für mathematische Funktionen (Teil 1)

> Auf dem nächsten Übungsblatt gibt es eine Aufgabe, die auf der Lösung dieser Aufgabe aufbaut!

Schreiben Sie eine Basisklasse `Function` mit einer rein virtuellen Methode `calculate`, welche einen `float` Parameter und gibt das Ergebnis der Funktion für die Eingabe als `float` zurückgibt.

Darüber hinaus sollen Sie die Special-Member-Functions, wie in der Vorlesung demonstriert, definieren damit sichergestellt ist, dass die Destruktoren der abgeleiteten Klassen immer aufgerufen werden und Slicing (Z.17) vermieden wird.

Leiten Sie von dieser Klasse anschließend zwei konkrete Klassen `Linear_function` ($f(x) = ax + b$) und `Quadratic_function`| ($f(x) = ax^2 + bx + c$) ab, die im Konstruktor entsprechend zwei bzw. drei Faktoren erhalten/speichern und die virtuelle Member-Funktion der Basisklasse passend implementieren.

Als Basis für die Implementierung können Sie den im Handout bereitgestellten Programmcode verwenden, der bei einer korrekten Lösung z.B. die folgende Ausgabe erzeugen sollte:

```
1.
    x= 2 : 5
    x=-2 : 1
2.
    x= 2 : 7
    x=-2 : -1
```

2 Implementieren einer Klassenhierarchie

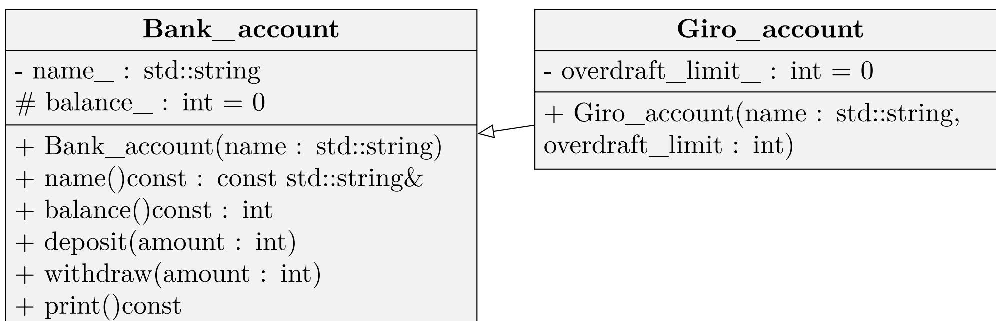
Implementieren Sie die beiden Klassen, welche durch das folgende UML-Diagramm beschrieben werden, sodass die main-Funktion aus dem Handout korrekt funktioniert.

Beim `Bank_account` dürfen keine Abhebungen `withdraw()` ausgeführt werden, durch die der Kontostand negativ werden würde. Statt dessen soll in diesen Fällen die im Handout definierte `Not_enough_money` Exception geworfen werden.

Der `Giro_account` erlaubt dagegen auch ein überziehen des Kontos, solange die Überziehung nicht größer als das festgelegte Limit wird.

Die `print()` Member-Funktion soll, wie in den Beispielausgaben im Handout, den Namen des Accounts und den aktuellen Kontostand ausgeben.

Entscheiden Sie selbst welche Member-Funktion für diese Aufgabe mit `virtual` markiert werden müssen und beachten Sie auch die Hinweise aus der Vorlesung was die besprochenen Probleme (z.B. Slicing) angeht.



3 Shape-Beispielprojekt erweitern

Im Handout finden Sie eine Kopie des Beispielprojekts aus der letzten Vorlesung, dessen `main`-Funktion um einige zusätzliche Befehle erweitert wurde.

Schreiben Sie eine `Triangle`-Klasse, bestehend aus den Längen der drei Seiten, die von der `Shape`-Klasse ableitet und mit der erweiterten `main`-Funktion korrekt funktioniert. Beachten Sie hierbei auch, dass sie die neue Klasse im `Shape_visitor` und im `Cmd_renderer` entsprechend berücksichtigen müssen.

4 Dynamische vs. statische Bindung

Im folgenden Programm wurden absichtlich einige Guidelines aus der Vorlesung missachtet. Schauen Sie sich das Programm an und geben Sie, in einer Tabelle oder als Textdatei analog zu a), für jede der markierten Zeilen (a-n) an:

- Welche Deklaration beim Kompilieren gefunden wird
- Welche Definition über die dynamische Bindung zur Laufzeit tatsächlich aufgerufen wird oder "keine", falls keine dynamische Bindung stattfindet
- Bei der `draw`-Funktion: Was der Wert des Parameters ist

Beschreiben Sie *außerdem* kurz in eigenen Worten den Unterschied zwischen dynamischer und statischer Bindung und in welchen Fällen sie jeweils angewendet werden.

```
1  #include <iostream>
2  #include <stdexcept>
3  #include <memory>
4  #include <vector>
5
6  class Game_state {
7    public:
8      virtual ~Game_state() = default;
9      virtual std::string name()const {return "";}
10     virtual void draw(float delta_time = 0.f) {}
11 };
12
13 class Menu_screen : public Game_state {
14   public:
15     std::string name()const { return "Menu_screen"; }
16 };
17
18 class Main_menu_screen : public Menu_screen {
19   public:
20     std::string name() { return "Main_menu_screen"; }
21     void draw(float delta_time=-1.f) {}
22 };
23
24 class My_exception {
25   public:
26     virtual ~My_exception() = default;
27     virtual std::string what()const {return "¯\\_(ツ)_/¯";}
28 };
29 class Invalid_argument : public My_exception {
30   public:
31     std::string what()const {return "Invalid_argument";}
32 };
33
34
```

```
35 void by_ref_1(Main_menu_screen& s) {
36     s.draw(); // a) Main_menu_screen::draw, Main_menu_screen::draw, -1.f
37     s.name(); // b)
38 }
39 void by_ref_2(Menu_screen& s) {
40     s.draw(); // c)
41     s.name(); // d)
42 }
43 void by_ref_3(Game_state& s) {
44     s.draw(); // e)
45     s.name(); // f)
46 }
47
48 void by_value_1(Main_menu_screen s) {
49     s.draw(); // g)
50     s.name(); // h)
51 }
52 void by_value_2(Menu_screen s) {
53     s.draw(); // i)
54     s.name(); // j)
55 }
56 void by_value_3(Game_state s) {
57     s.draw(); // k)
58     s.name(); // l)
59 }
60
61 int main() {
62     Main_menu_screen menu = Main_menu_screen{};
63
64     by_ref_1(menu);
65     by_ref_2(menu);
66     by_ref_3(menu);
67
68     by_value_1(menu);
69     by_value_2(menu);
70     by_value_3(menu);
71
72     try {
73         throw Invalid_argument{};
74     } catch(const My_exception& e) {
75         e.what(); // m)
76     }
77
78     try {
79         throw Invalid_argument{};
80     } catch(const My_exception e) {
81         e.what(); // n)
82     }
83 }
```