Praxisorientierte Einführung in C++
Aufgabenblatt 7

Christof Elbrechter
celbrech@techfak.uni-bielefeld.de

24. April 2014

Aufgabe1: Aggregation und einfache Vererbung

Grundlage für diese Aufgabe soll folgende Typhierarchie darstellen:

```
enum Ability { Swim, Fly, Walk, LayEggs };
```

```
                        Animal:
                        int legs
                        string name
                        std::vector<Ability> abilities
                        bool isA(string) const
                        bool can(Ability) const

    Mammel:             Insect:             Fish:               Bird:
    bool carnivore      bool canSting       bool eatsFish       bool raptor
    bool liveStock      bool hasPoison      float maxSpeed
                        int wings

Dog:        Cow:        Ant:     Bee:       Shark:  Carp:       Eagle:  Blackbird:
```

a. (8 Punkte) Implementieren Sie die Vererbungshierarchie auf Basis einfacher `public` Vererbung. Dabei sollen alle spezielleren Typen auch die Variablen der jeweiligen Oberklasse mit-initialisieren. Hierbei sollen Merkmale, die für einen Typen fest stehen, automatisch gesetzt werden. Z.B. hat der `Insect`-Konstruktor kein Argument `legs`, sondern setzt `legs` immer automatisch auf 6. Die Initialisierung der Variablen soll – falls möglich – komplett in der Initialisierungsliste erfolgen. Dabei müssen immer auch Oberklassenkonstruktoren aufgerufen werden. Eine Sonderstellung soll die Variable `name` haben. Diese soll immer eine hierarchische Representation eines Speziestypen enthalten. Für den Hund könnte der Name z.B. "Animal.Mammel.Dog" lauten. Die `Animal::isA` Funktion soll `name` verwenden, um festzustellen, ob ein Tier einer bestimmten Spezies angehört. Hierbei soll z.B. auch `Dog(...).isA("Mammel")` `true` ergeben.

Aufgabe2: Friends und Operatoren

Nehmen Sie folgende Klasse als Ausgangspunkt (Nennen Sie den Header `vector_3d.h`):

```
// vector_3d.h
#ifndef VECTOR_3D_H
#define VECTOR_3D_H

namespace vector {

class Vector3D {
        float m_data[3];

        public:
                Vector3D(float x1 = 0, float x2 = 0, float x3 = 0)
                {
                        m_data[0] = x1;
                        m_data[1] = x2;
                        m_data[2] = x3;
                }

                inline float &operator[](unsigned int index) {
                        return m_data[index];
                }

                inline const float &operator[](unsigned int index) const {
                        return m_data[index];
                }
};

} // namespace
#endif
```

Implementieren Sie alle Operatoren der folgenden Aufgaben **nicht inline**. D.h. deklarieren Sie die Operatoren (wie üblich) im Header file und liefern Sie die Implementationen einer Quelltextdatei. Nennen Sie diese `vector_3d.cpp`.

a. (5 Punkte) Implementieren Sie `operator*` dreimal (als Funktionen, nicht als Methoden der Klasse `Vector3D`) im Namensraum `vector`, und zwar mit den folgenden Signaturen:

```
inline float operator*(const Vector3D &v, const Vector3D &w);
```

Diese Variante soll das Skalarprodukt von `v` und `w` berechnen. Zur Erinnerung: Das Skalarprodukt zweier Vektoren, v und w aus R^3, sieht so aus: $(v, w) = \sum_{i=1}^{3} v_i w_i$

```
inline Vector3D operator*(float f, const Vector3D& v);
```

```
inline Vector3D operator*(const Vector3D& v, float f);
```

Diese beiden Varianten sollen die Komponenten des Vectors `v` mit f multiplizieren und das Ergebnis als neues Objekt zurückgeben. Deklarieren Sie alle drei Operatoren als `friend` der Klasse `Vector3D`. Führen Sie alle Zugriffe auf `m_data` direkt, und nicht über den `Vector3D::operator[]`, aus.

b. (2 Punkte) Schreiben Sie ein Programm (in einer Datei namens `vector_3d_main.cpp`), welches die drei Operatoren aus der vorherigen Aufgabe benutzt, um die Vektoren $(6, 5, 4)$ und $(3, 2, 1)$ um den Faktor 22 zu skalieren, und anschliessend das Skalarprodukt der resultierenden Vektoren zu berechnen.

c. (3 Punkte) Implementieren Sie zusätzlich den Operator

```
inline float operator*(const Vector3D &v, const Vector3D &w);
```

noch einmal als Methode. Wie muss die Signatur angepasst werden? Gibt es Probleme, wenn der zugehörige Operator auch als globale Funktion zur Verfügung steht? Wie können evtl. Ambiguitäten aufgelöst werden?

Aufgabe3: Enums

a. (2 Punkte) Implementieren Sie einen Aufzählungstypen `Trool` welcher den Typen `bool` um eine dritte mögliche Wertausprägung erweitert. Die möglichen Werte sollen `Yes`, `No` und `Perhaps` heissen.

b. (3 Punkte) Implementiern Sie die Stream-Operatoren

```
std::ostream &operator<<(std::ostream&,const Trool &)
```

und

```
std::istream &operator>>(std::istream&,Trool&)
```

. Die serialisierten Bezeichner sollen dafür gleich den Bezeichnern im Code sein.

c. (3 Punkte) Implementieren Sie die logischen Operatoren `||`, `&&` und `!` unter Berücksichtigung folgender Regeln:

- Nur `Yes` **und** `Yes` bleibt `Yes`
- `No` **und** irgendwas bleibt `No`
- Alles andere bleibt `Perhaps` wenn es mit **und** verknüpft wird
- Nur `No` **oder** `No` bleibt `No`
- `Yes` **oder** irgentwas bleibt `Yes`
- Alles andere bleibt `Perhaps` wenn es mit **oder** verknüpft wird
- **Nicht** `Yes` wird `Perhaps`
- **Nicht** `No` wird `Perhaps`
- **Nicht** `Perhaps` wird `Yes` oder `No` :-)

d. (3 Punkte) **Textaufgabe:** Sie fragen ihren Lebenspartner (offensichtlich ein großer Fan der `Trool`-Logik), ob er/sie mit Ihnen ins Kino gehen möchte. Die Antwort ist:

```
!(Perhaps || No && (!Perhaps || No) && Yes || !Perhaps && (Yes || No))
```

Müssen Sie nun allein ins Kino oder nicht? Falls Sie sich nicht sicher sind, können Sie nun sehr leicht ein Programm schreiben, welches Ihnen die Antwort berechnet.