Konstruktoren und Destruktoren

AUFGABE 2.1 (Kuscheltiere)

Ziel dieser Übung ist es, anhand eines ausführlichen Beispiels zu veranschaulichen, wie man einerseits Konstruktoren und Destruktoren schreibt und andererseits deren Funktionsweise aufzeigt.

Dafür schlage ich vor, mit Kuscheltieren zu spielen. Diese ausgestopften Tiere werden durch die Art des dargestellten Tieres, den Namen des ausgestopften Tieres sowie seinen Verkaufspreis gekennzeichnet.

(a) Beginnen Sie mit Ihrem Wissen aus der vergangenen Woche, eine erste Grundlage für die Modellierung von Kuscheltieren (Attribute, Accessoren, Modifikatoren) zu schreiben. Sie können davon ausgehen, dass sich die Attribute `tierart` und `name` sich nicht mehr ändern, wenn ein Kuscheltier einer bestimmten Art hergestellt wird (es wird nicht in eine anderes umgewandelt). Für `preis` ist dies nicht der Fall. Stellen Sie für dieses eine Attribut eine Methode bereit, die es ermöglicht, seinen Wert zu ändern.

(b) Statten Sie nun das Programm mit einer externen `etikett`-Funktion aus. Nehmen Sie ein Kuscheltier als Argument und führen die Anzeige des Etiketts darauf durch. Wer einwenden würde, dass die Anzeige des Etiketts eher eine Methode der Kuscheltier-Klasse sein sollte, hat vollkommen recht! Aber nur dieses Mal nehmen Sie eine „externe“ Funktion.

(c) Lassen Sie uns auch ein minimalistisches main() hinzufügen, das ein Kuscheltier instanziiert und die Anzeige seines Etiketts anfordert. Was fällt Ihnen auf?

Hinweise:

Der vom Compiler bereitgestellte Standardkonstruktor wird zwar ganz am Anfang des Programms ausgeführt, aber man stellt fest, dass weder Name noch Art angezeigt wird und vor allem, dass der Preis ein Fantasiepreis ist. Dies geschieht, weil beim Schreiben des Programms ein eklatanter Fehler gemacht wurde. Der Inhalt einiger Variablen wurde verwendet, ohne dass diese zuvor initialisiert worden waren. Der bereitgestellte Standardkonstruktor vom Compiler initialisiert nämlich die Größen des Grundtyps (int, double, ...) nicht.
Der Sinn von Konstruktoren besteht gerade darin, eine elegante Möglichkeit zu bieten, diese Art von Versäumnissen zu vermeiden. Man kann feststellen, dass die Attribute, die über einen Standardkonstruktor verfügen (in diesem Fall die beiden Strings), trotz dieses Versehens initialisiert wurden (hier mit dem leeren String). Diese Konstruktoren wurden tatsächlich aufgerufen, so wie der Standardkonstruktor von Kuscheltier aufgerufen wird, wenn man schreibt:

```
Kuscheltier bobo;
```

Die Angabe eines bestimmten Werts für Attribute im Standardkonstruktor (insbesondere für Attribute, die selbst keine Standardkonstruktoren zulassen) ist nicht möglich. Standardkonstruktoren wären eine Möglichkeit, die in diesem Fall jedoch

unbefriedigend ist. Es ist zwingend erforderlich, diese mit einem relevanten Wert zu initialisieren.
Die Lösung besteht darin, dafür einen expliziten Konstruktor zu verwenden und nicht den Standardkonstruktor (der in einem solchen Beispiel kaum eine Rolle spielt). Die Argumente dieses Konstruktors sind notwendigerweise der Name, den das Kuscheltier erhalten soll, seine Art sowie ein Startwert für den Preis (der als einziges Element später geändert werden kann).

Bei der Programmausführung wird folgendes angezeigt:

```
[Ein Kuscheltier wurde hergestellt.]
Etikett:
Hallo, mein Name ist Bobo
Ich bin ein Baer und koste 14,95 Euro.
```

(d) Lassen Sie uns nun einen expliziten Kopierkonstruktor hinzufügen (und ersetzen daher den automatisch vom Compiler generierten), was zu Folgendem führt: Anstatt eine perfekte Kopie zu erstellen, ändert dieser Konstruktor den Namen des Kuscheltiers, dass er kopieren muss ein bisschen. Dies ermöglicht es, beim Anbringen des Etiketts die Kopien von den Originalen zu unterscheiden, zum Beispiel:

```
[Ein Kuscheltier wurde hergestellt.]
Etikett:
[Ein Kuscheltier wurde kopiert.]
Hallo, mein Name ist Bobo-Kopie
Ich bin ein Baer und koste 14,95 Euro.
```

Erläuterung
Beachten Sie, dass wir `main()` nicht ändern mussten, um eine Kopie zu erhalten! Es ist das Etikett einer Kopie des Bobo-Kuscheltiers, das angezeigt wird. Der Grund ist: Die Label-Funktion nimmt ein `Kuscheltier` als Parameter, durch eine Wertübergabe; diese Funktion arbeitet daher mit einer (lokalen) Kopie des Kuscheltiers, der als Argument für den Aufruf verwendet wird. Genau um diesen Punkt zu veranschaulichen, wollten wir eine externe Funktion.
Um dies zu vermeiden, ist es notwendig, Parameter als Referenz zu übergeben, und zwar in diesem Fall durch eine konstante Referenz, da Label das als Argument erhaltene Kuscheltier nicht ändern muss: `void etikett(const Kuscheltier& k) { ... }` Wenn das Programm nun erneut ausgeführt wird, sehen wir, dass der Kopierkonstruktor nicht mehr aufgerufen wird.

(e) Fügen Sie nun einen expliziten Destruktor hinzu (wiederum den automatisch generierten ersetzen) und die Nachrichten der vorherigen Konstruktoren ein wenig ändern, sodass sie den Namen des betroffenen Kuscheltiers anzeigen. Fügen Sie zudem ein paar weitere Kuscheltiere zur `main` hinzu. Welches Objekt wird wann kopiert und wann gelöscht? Warum?

```
Kuscheltier* pingu; // noch kein Objekt, nur ein Pointer
{
      Kuscheltier ssss("Cobra", "Ssss", 10.00);
      pingu = new Kuscheltier("Pinguin", "Pingu", 20.00);
}
Kuscheltier pingu_klon(*pingu);
etikett_lesen(pingu_klon);
delete pingu;
```

AUFGABE 2.2 (Mittagessen)

Definieren Sie eine Klasse `Vorspeise` so, dass folgende `main()` die untenstehende Ausgabe liefert:

```
int main() {
   Vorspeise amuse_bouche;
   cout << "Gerne!" << endl;
   amuse_bouche.zugabe();
   cout << "Nein, danke." << endl;
   return 0;
}
```

Ausgabe:

```
Es ist Zeit fuer das Mittagessen. Vorspeise?
Gerne!
Nachschlag?
Nein, danke.
Dann kommt die Hauptspeise!
```

AUFGABE 2.3 (Bank) - **Abgabe**

Das (sehr schlechte) Programm `bank1.cpp`, dessen Code auf StudIP bereitgestellt wird, enthält ein Bankprogramm, das in Form von Funktionen modularisiert ist. Verwandeln Sie es in ein objektorientiertes Programm namens `bank2.cc`, indem Sie folgenden Schritten folgen:

a) Studieren Sie, wie das Programm funktioniert. Die Bank hat 2 Kunden. Jeder Kunde hat ein Privatkonto und ein Sparkonto mit unterschiedlichen Guthaben. Der Zinssatz eines Sparkontos ist höher als der eines Privatkontos. Die Daten jedes Kunden (Name, Stadt und Salden) werden vor und nach dem Kontoabschluss angezeigt.

b) Überlegen Sie sich, welche Objekte Sie in Ihrem Programm verwenden möchten, und fügen Sie die entsprechenden Klassen hinzu. Dies können Objekte jeglicher Art sein (Kunde, Haus, Ticket, Konto, Bankbeziehung etc.). Denken Sie daran, dass Modularisierung keine exakte Wissenschaft ist. Jede/r ProgrammiererInn entscheidet, welche Klassen er/sie sinnvoll findet und welche ihr/m als das beste Realitätsmodell erscheinen. Dies ist oft der schwierigste Schritt in einem Programmierprojekt.

c) Übertragen Sie Code über Objekte an Klassen. Verwenden Sie das Schlüsselwort private, um Attribute und Methoden zu kapseln, die außerhalb der Klasse nicht verwendet werden. Jede Methode sollte kurz und ohne zu viele detaillierte Anweisungen sein.

Bezeichner (Namen von Variablen, Attributen und Methoden) sollten aussagekräftig sein.

Ausgabe:

```
Daten vor Abschluss der Konten:
Kunde Pedro aus Berlin
   Privatkonto: 1000 Euro
   Sparkonto: 2000 Euro
Kunde Alexandra aus Bonn
   Girokonto: 3000 Euro
   Sparkonto: 4000 Euro
Daten nach Abschluss der Konten:
Kunde Pedro aus Berlin
   Privatkonto: 1010 Euro
   Sparkonto: 2040 Euro
Kunde Alexandra aus Bonn
   Privatkonto: 3030 Euro
   Sparkonto: 4080 Euro
```

d) Sicher ist Ihnen aufgefallen, dass die Darstellung des bisherigen Programms nicht zwischen männlichen und weiblichen Kunden unterscheidet. Dies lässt sich in der objektorientierten Version des Programms leicht beheben, indem Sie beispielsweise eine männliche boolesche Instanzvariable zur Kunden-Klasse hinzufügen (falls vorhanden) und ihren Wert in der Anzeigemethode testen. Ändern Sie Ihr Programm so, dass es "Kundin" anstelle von "Kunde" anzeigt.

Ausgabe:

```
Daten vor Abschluss der Konten:
Kunde Pedro aus Berlin
   Privatkonto: 1000 Euro
   Sparkonto: 2000 Euro
Kundin Alexandra aus Bonn
   Girokonto: 3000 Euro
   Sparkonto: 4000 Euro
Daten nach Abschluss der Konten:
Kunde Pedro aus Berlin
   Privatkonto: 1010 Euro
   Sparkonto: 2040 Euro
Kundin Alexandra aus Bonn
   Privatkonto: 3030 Euro
   Sparkonto: 4080 Euro
```

AUFGABE 2.3 (Supermarkt) - **Abgabe**

Ein Supermarkt bittet Sie, ihm dabei zu helfen, die an seinen Kassen erfassten Gesamteinkäufe anzuzeigen. Vervollständigen Sie dafür das auf StudIP bereitgestellte Programm unter `supermarkt.cc`.

Hier sind die Entitäten, die benötigt werden, um die Funktionsweise des Supermarkts zu modellieren:

- die verkauften `Artikel`: gekennzeichnet durch ihre Namen (eine Zeichenfolge), ihren Einheitspreis (ein double) und einen booleschen Wert, der angibt, ob der Artikel Teil einer Aktion ist oder nicht;
- `Einkauf`: Ein Einkauf ist gekennzeichnet durch den gekauften Artikel und die gekaufte Menge dieses Artikels;
- `Einkaufswagen`: gekennzeichnet durch alle darin enthaltenen Einkäufe;
- `Kasse`: zuständig für das Scannen und Erfassen des Inhalts der Einkaufswagen. Eine Kasse ist durch eine Kassennummer (eine ganze Zahl) und die Gesamtzahl der gescannten Einkäufe (ein double) gekennzeichnet.

(a) Das auf StudIP bereitgestellte Hauptprogramm soll die Gesamtmenge jeder Box am Ende eines bestimmten Tages anzeigen. Beginnen Sie damit, es sich anzuschauen.
(b) Fügen Sie dann die fehlenden Datenstrukturen und Methoden hinzu. Diese Entitäten müssen mit dem bereitgestellten Hauptprogramm getestet werden können.
(c) Deklarieren Sie in dieser Datei die Klassen, die zum Modellieren des Supermarkts benötigt werden, wie oben vorgeschlagen.
(d) Es wird empfohlen, einen Vektor für die Einkäufe zu verwenden, um den Inhalt des Einkaufswagens zu modellieren. Achten Sie genau auf die Kapselung (z. B. müssen Attribute privat sein).
(e) Die Methode, die in der Klasse `Einkauf` implementiert werden muss ist: `anzeigen()`, die die Eigenschaften des Artikels anzeigt (seinen Namen, seinen Einheitspreis, die gekaufte Menge und den Endpreis). Wenn der betreffende Artikel im Angebot ist, muss außerdem der Text „(Aktion)“ angezeigt werden. Ergänzen Sie ggf. weitere nötige Methoden.

Hier ist die Ausgabe für `anzeigen()`:

```
Kleine-Milch : 1,5 x 6 = 9 Euro (Aktion)
```

(f) Für die Klasse `Einkaufswagen` implementieren Sie `fuellen(..)` wie im Startprogramm angegeben. Überlegen Sie sich wie Sie den Inhalt des Einkaufwagens speichern (der im Anschluss gescannt wird). Ergänzen Sie ggf. weitere nötige Methoden.

Für die Klasse `Kasse` implementieren Sie folgende Methoden:

- `anzeigen()` welche die Kassennummer und den Wert seines Gesamtbetrags in Form des folgenden Beispiels anzeigt:

  ```
  Kasse 1 hat heute 121,15 Euro einkassiert.
  ```

  Wobei 1 die Kassennummer und 121,15 der Gesamtbetrag ist. Gehen Sie davon aus, dass dieser Gesamtbetrag als Attribut gespeichert ist (und von der `scannen(..)`–Methode weiter unten aktualisiert wird).

- `Scannen(..)`: Diese Methode, (siehe Nutzung im Hauptprogramm), ermöglicht es der Kasse, den dem Inhalt des Einkaufswagens entsprechenden Beleg anzuzeigen. Diese Methode sollte auch den Gesamtbetrag der Kasse aktualisieren, indem die Anzahl der

Einkäufe aus dem Einkaufswagen hinzugefügt wird. Die Anzeige der Quittung muss gemäß der untenstehenden Ausgabe erfolgen und die zuvor implementierte `anzeigen()`-Methode verwenden.

Beispiel Ausgabeanzeige des Programms

Nach Abschluss des Programms sollte die Ausführung des Hauptprogramms wie folgt aussehen:

```
Blumenkohl extra : 3.5 x 2 = 7 Euro
C++ in drei Tagen : 48.5 x 1 = 48.5 Euro
Omamas Gebñck : 3.2 x 4 = 12.8 Euro
Kleine-Milch : 2.5 x 6 = 7.5 Euro (Aktion)
Sardinen : 6.5 x 2 = 13 Euro
-----
Gesamtsumme Einkauf : 88.8 Euro
========================================
Die Leider der Sophie : 16.5 x 1 = 8.25 Euro (Aktion)
Der Cremige 100% : 5.8 x 1 = 5.8 Euro
Erbsen TK : 4.35 x 2 = 8.7 Euro
Williamsbirne : 4.8 x 2 = 9.6 Euro
-----
Gesamtsumme Einkauf : 32.35 Euro
========================================
100% Arabica : 6.9 x 2 = 6.9 Euro (Aktion)
Vollkornbrot : 6.9 x 1 = 6.9 Euro
Der Cremige 100% : 5.8 x 2 = 11.6 Euro
-----
Gesamtsumme Einkauf : 25.4 Euro
========================================
Gesamt Tag :
Kasse 1 : 121.15 Euro
Kasse 2 : 25.4 Euro
```

LÖSUNG AUFGABE 2.1:

(a) `Kuscheltier`-Klasse erstellen

```
using namespace std;
#include <iostream>
#include <string>

class Kuscheltier {
private:
    string tierart;
    string name;
    double preis;
public:
    string getTierart() const { return tierart; }
    string getName() const { return name; }
    double getPreis() const { return preis; }
    void setPreis(double valeur) { preis = valeur; }
};
```

(b) Externe Funktion `etikett_lesen` erstellen

```
void etikett_lesen(Kuscheltier k) {
    cout << "Hallo, mein Name ist  " << k.getName() << endl
         << "Ich bin ein  " << k.getTierart() << " und koste "
         << k.getPreis() << " Euro." << endl;
}
```

(c) Main ergänzen und eigenen Konstruktor ergänzen

```
int main() {
    Kuscheltier bobo; // Hier wird der Standardkonstruktor genutzt
    cout << "Etikett:" << endl;
    etikett_lesen(bobo);
    return 0;
}

double preis = 0.0;
```

oder

```
    //eigener Konstruktor
    Kuscheltier(string tierart, string name, double kaufpreis)
        : tierart(tierart), name(name), preis(kaufpreis)
    {
        cout << "[Ein Kuscheltier wurde hergestellt.]" << endl;
    }
```

(d) Kopierkonstruktor ergänzen

```
    //Kopierkonstruktor
    Kuscheltier(Kuscheltier const& k)
        : tierart(k.tierart), name(k.name + "-Kopie"), preis(k.preis)
    {
        cout << "[Ein Kuscheltier wurde kopiert.]" << endl;
```

```
    }
(e) Destruktor hinzufügen

    ~Kuscheltier()
    {
          cout << "[Das Kuscheltier " << name << " ist kaputt.]" << endl;
    }
```