Objekt 2 wird erzeugt.
Block wird verlassen
Objekt 2 wird zerstört.
main wird verlassen
Objekt 1 wird zerstört.
Objekt 0 wird zerstört.

Der Destruktor statischer Objekte (static oder globale Objekte) wird nicht nur beim Verlassen eines Programms mit `return`, sondern auch bei Verlassen mit `exit()` aufgerufen. Im Gegensatz zum normalen Verlassen eines Blocks wird der Speicherplatz bei `exit()` jedoch nicht freigegeben.

3.6 Wie kommt man zu Klassen und Objekten? Ein Beispiel

Es kann hier keine allgemeine Methode gezeigt werden, wie man von einer Aufgabe zu Klassen und Objekten kommt. Es wird jedoch anhand eines Beispiels ein erster Eindruck vermittelt, wie der Weg von einer Problemstellung zum objektorientierten Programm aussehen kann.

Es geht hier um ein Programm, das zu einer gegebenen Personalnummer den Namen heraussucht[6]. Gegeben sei eine Datei *daten.txt* mit den Namen und den Personalnummern der Mitarbeiter. Dabei folgt auf eine Zeile mit dem Namen eine Zeile mit der Personalnummer. Das #-Zeichen ist die Endekennung. Der Inhalt der Datei ist:

```
Hans Nerd
06325927
Juliane Hacker
19236353
Michael Ueberflieger
73643563
#
```

Einige Analyse-Überlegungen

Um die Problemstellung zu verdeutlichen, wird sie aus verschiedenen Blickwinkeln betrachtet. Es handelt sich dabei nur um *Möglichkeiten*, nicht um den einzig wahren Lösungsansatz (den es nicht gibt).

1. In der Analyse geht es zunächst einmal darum, den typischen Anwendungsfall (englisch *use case*) *in der Sprache des (späteren Programm-) Anwenders* zu beschreiben. Ein ganz konkreter Anwendungsfall, Szenario genannt, ist ein weiteres Hilfsmittel zum Verständnis dessen, was das Programm tun soll.

[6] Ähnlichkeiten mit der Aufgabe 1.19 von Seite 106 sind weder zufällig noch unbeabsichtigt.

2. Im zweiten Schritt wird versucht, beteiligte Objekte, ihr Verhalten und ihr Zusammenwirken zu identifizieren. Dies ist nicht unbedingt einfach, weil spontan gefundene Beziehungen zwischen Objekten im Programm nicht immer die wesentliche Rolle spielen.

Anwendungsfall (use case)

Das Programm wird gestartet. Alle Namen und Personalnummern werden zur Kontrolle ausgegeben (weil es hier nur wenige sind). Anschließend erfragt das Programm eine Personalnummer und gibt daraufhin den zugehörigen Namen aus oder aber die Meldung, dass der Name nicht gefunden wurde. Die Abfrage soll beliebig oft möglich sein. Wird X oder x eingegeben, beendet sich das Programm.

Für einen konkreten Anwendungsfall (= Szenario) wird die oben dargestellte Datei *daten.txt* verwendet.

Szenario

Das Programm wird gestartet und gibt aus:

Hans Nerd 06325927
Juliane Hacker 19236353
Michael Ueberflieger 73643563

Anschließend erfragt das Programm eine Personalnummer. Die Person vor dem Bildschirm (Benutzer / User) gibt 19236353 ein. Das Programm gibt »Juliane Hacker« aus und fragt wieder nach einer Personalnummer. Jetzt wird 99999 eingegeben. Das Programm meldet »nicht gefunden!« und fragt wieder nach einer Personalnummer. Jetzt wird X eingegeben. Das Programm beendet sich.

Objekte und Operationen identifizieren

Im nächsten Schritt wird versucht, die beteiligten Objekte und damit ihre Klassen zu identifizieren und eine Beschreibung ihres Verhaltens zu finden.

In der nicht-objektorientierte Lösung zur Vorläuferaufgabe 1.19 werden alle Aktivitäten in `main()` abgehandelt. Das ist nicht vorteilhaft, weil die Funktionalität damit nicht einfach in ein anderes Programm transportiert werden kann. Deswegen bietet es sich an, die Aktivitäten in ein eigens dafür geschaffenes Objekt zu verlegen. Die Klasse dazu sei hier etwas hochtrabend `Personalverwaltung` genannt. Was müsste so ein Objekt tun?

1. Die Datei *daten.txt* lesen und die gelesenen Daten speichern. Der Einfachheit wird hier angenommen, dass keine andere Datei zur Wahl steht.
2. Die Daten auf dem Bildschirm *ausgeben*.
3. Einen *Dialog* mit dem Benutzer *führen*, in dem nach der Personalnummer gefragt wird.

Diese drei Punkte und die Kenntnis der Datei führen zu entsprechenden Schlussfolgerungen. Dabei sind im ersten Schritt die Substantive (Hauptworte) als Kandidaten für Klassen zu sehen und Verben (Tätigkeitsworte) als Methoden. Passivkonstruktionen sollen dabei

vorher stets in Aktivkonstruktionen verwandelt werden, d.h. *ausgeben* ist besser als *die Ausgabe erfolgt*.

1. Eine Wahl der Datei ist hier nicht vorgesehen. Ein Objekt der Klasse `Personalverwaltung` soll daher schon beim Anlegen die Datei einlesen und die Daten speichern. Das übernimmt am besten der Konstruktor, dem der Dateiname übergeben wird.
   Die gelesenen Daten gehören zu Personen. Jede *Person* hat einen Namen und eine Personalnummer. Es bietet sich an, Name und Personalnummer in einer Klasse `Person` zu kapseln. Aus Gründen der Einfachheit sollen Vor- und Nachname nicht getrennt gehalten werden; ein Name genügt.
   Die Personalnummer soll nicht als `int` vorliegen, sondern als `string`, damit nicht führende Nullen (siehe Datei oben) beim Einlesen verschluckt werden oder zu einer Interpretation als Oktalzahl führen. Außerdem könnte es Nummernsysteme mit Buchstaben und Zahlen geben.
   Die Klasse `Personalverwaltung` soll die Daten speichern. Dafür bietet sich ein `vector<Person>` als Attribut an.
2. Das Tätigkeitswort *ausgeben* legt nahe, eine gleichnamige Methode `ausgeben()` vorzusehen. In der Methode werden Name und Personalnummer einer Person ausgegeben. Es muss also entsprechende Methoden in der Klasse `Person` geben, etwa `getName()` und `getPersonalnummer()`. Diese Methoden würden innerhalb der Funktion `ausgeben()` aufgerufen werden.
3. *Dialog führen* legt nahe, eine Methode `dialogfuehren()` oder kurz `dialog()` vorzusehen.

Weil nur ein erster Eindruck vermittelt werden soll und die Problemstellung einfach ist, wird auf eine vollständige objektorientierte Analyse (OOA) und ein entsprechendes Design (OOD) verzichtet und auf die Literatur verwiesen, die die OOA/D-Thematik ausführlich behandelt, zum Beispiel [Oe]. Hier konzentrieren wir uns gleich auf eine Lösung mit C++. Ein mögliches `main()`-Programm könnte wie folgt aussehen:

Listing 3.17: main-Programm zur Personalverwaltung

```
// cppbuch/k3/personalverwaltung/main.cpp
#include "personalverwaltung.h"
#include <iostream>
using namespace std;

int main() {
   Personalverwaltung personalverwaltung("daten.txt");  // Konstruktor
   cout << "Gelesene Namen und Personalnummern:\n";
   personalverwaltung.ausgeben();

   personalverwaltung.dialog();
   cout << "Programmende\n";
}
```

Die Klasse `Person` ist einfach zu entwerfen:

Listing 3.18: Klasse `Person`

```
// cppbuch/k3/personalverwaltung/person.h
#ifndef PERSON_H
```

```
#define PERSON_H
#include <string>

class Person {
public:
   Person(const std::string& name_, const std::string& personalnummer_)
      : name {name_}, personalnummer {personalnummer_} {
   }

   const std::string& getName() const {
      return name;
   }

   const std::string& getPersonalnummer() const {
      return personalnummer;
   }
private:
   std::string name;
   std::string personalnummer;
};
#endif
```

Auch die Klasse `Personalverwaltung` ist nach den obigen Ausführungen nicht schwierig, wenn man sich zunächst auf die Prototypen der Methoden beschränkt:

Listing 3.19: Klasse `Personalverwaltung`

```
// cppbuch/k3/personalverwaltung/personalverwaltung.h
#ifndef PERSONALVERWALTUNG_H
#define PERSONALVERWALTUNG_H
#include <vector>
#include "person.h"

class Personalverwaltung {
public:
   Personalverwaltung(const std::string& dateiname);

   void ausgeben() const;

   void dialog() const;
private:
   std::vector<Person> personal;
};
#endif
```

Für die Implementierung der Methoden der Klasse `Personalverwaltung` muss man sich mehr Gedanken machen. Das überlasse ich Ihnen (siehe die nächste Aufgabe)! Die Lösung dürfte aber nicht schwer sein, wenn Sie die Aufgabe 1.19 von Seite 106 gelöst oder deren Lösung nachgesehen haben.