Kapitel 11

Klassen

Bei der klassischen prozeduralen Programmierung, wie Sie sie bis zu diesem Kapitel vorwiegend verwendet haben, ist es so, dass Daten und Funktionen keine Einheit bildeten. Probleme traten hierbei häufiger bei falschem Zugriff auf (beispielsweise nicht initialisierte) Daten auf. Musste ein Programm geändert oder erweitert werden, war der (Zeit-)Aufwand häufig enorm, und weitere Fehler waren so gut wie sicher.

Bei der objektorientierten Programmierung hingegen bilden die Objekte eine echte und logische Einheit aus Daten und Funktionen (die hier auch *Methoden* bzw. *Elementfunktionen* genannt werden). Dabei werden die Daten gekapselt, um Fehler und unzulässige Zugriffe zu vermeiden. Sinn und Zweck ist es, einen Zugriff auf diese Daten nur noch über das Objekt und eine dazugehörende Methode zu ermöglichen.

Quellcodes herunterladen

Auch in diesem Kapitel werden nicht alle Listings in voller Länge wiedergegeben, Sie finden jedoch bei Bedarf sämtliche Quellcodes auf der Webseite *https://www.rheinwerk-verlag.de/3981* zum Download.

11.1 Klassen

Es wurde bereits bei den Strukturen kurz erwähnt, dass Klassen zunächst nichts anderes als einfache Strukturen (`struct`) sind, für die üblicherweise das Schlüsselwort `class` statt `struct` verwendet wird. Wenn Sie Abschnitt 10.2, »Erste eigene Datentypen mit Strukturen«, aufmerksam durchgelesen haben, wird Ihnen der Einstieg leichter fallen.

»struct« versus »class«

Der Unterschied zwischen `struct` und `class` ist folgender: Wenn Sie in Ihrer Klasse nicht das Schlüsselwort `public` oder `private` verwenden, sind bei einer mit `struct` definierten Klasse alle Elemente (Daten und Methoden) `public` (also öffentlich zugänglich) und beim Schlüsselwort `class` alle Daten `private` (nur zur Klasse gehörend). Auf `public` und `private` gehe ich noch gesondert ein. In C++ stellen die Strukturen allerdings nur einen Sonderfall einer Klasse dar, und in der Praxis werden Klassen hauptsächlich mit `class` programmiert.

11.1.1 Klassendefinition

Um das Klassenkonzept etwas deutlicher zu erklären, erstellen wir hier eine Klasse `Automat`. Bei dieser Klasse soll es sich um einen Geldautomaten handeln, an dem der Kunde Bargeld von seinem Giro- oder Kreditkartenkonto abheben kann. Ich möchte das Thema hier nicht verkomplizieren, weshalb wir uns bei dieser Klasse auf das Wesentliche beschränken.

Eine Klassendefinition sieht wie folgt aus:

```
class Automat {
   // Daten und Methoden
};
```

Wie bei der Struktur dürfen Sie bei der Klassendefinition nicht das Semikolon am Ende vergessen. Für den Klassennamen gelten dieselben Regeln wie bei den Bezeichnern (siehe Abschnitt 2.4.1, »Bezeichner«); allerdings sollte ein Klassenname eindeutig sein. Gewöhnlich verwendet man einen großen Anfangsbuchstaben (dies ist zwar kein Standard, aber gängige Praxis).

Als Nächstes fügen Sie die Elemente (auch *Members* genannt) zur Klasse hinzu. Mitglieder einer Klasse sind die Daten (auch als *Eigenschaften* oder *Attribute* bezeichnet) und die Methoden. Zunächst die Attribute (Daten) der Klasse:

```
class Automat {
  unsigned long geld;
  std::string standort;
};
```

Hier haben Sie die Klasse `Automat` mit den Daten `geld` und `standort` versehen.

Seit C++11 ist es möglich, die Daten einer Klasse direkt zu initialisieren. Bisher war dies nur bei statischen, konstanten Elementen integralen Typs möglich. Damit wird Ihnen eine Menge an Tipparbeit abgenommen, wenn Sie später Konstruktoren erstellen, und Sie hantieren hierbei garantiert nicht mehr mit nicht initialisierten Daten herum. Ein Beispiel:

```
class Automat {
  unsigned long geld {OUL};              // seit C++11
  std::string standort {"Unbekannt"};    // seit C++11
};
```

Bis jetzt entspricht unsere Klasse einer einfachen Struktur mit privaten Daten. Daher bekommt sie nun noch die Methoden:

```
class Automat {
  // Daten der Klasse "Automat"
  unsigned long geld{OUL};
  std::string standort{"Unbekannt"};
  // Methoden der Klasse "Automat"
  unsigned long get_geld();
  void set_geld(unsigned long g);
  const std::string& get_standort();
  void set_standort(const std::string& s);
  void print();
};
```

Hiermit hätten Sie eine (fast) fertige Klassendefinition mit zwei Attributen (Daten) und fünf Methoden.

Klassendeklaration oder Klassendefinition?

Auch wenn die Methoden in der Klasse noch nicht definiert, sondern nur deklariert wurden, spricht man bei `class Y{ ... };` von einer Klassendefinition und nicht von einer Klassendeklaration. Es ist eine Definition, weil sie einen neuen Datentyp definiert.

11.1.2 Methoden definieren

Die Methoden, die Sie in der Klasse deklariert haben, müssen Sie noch definieren, um eine Klassendefinition vollständig zu machen. Theoretisch könnten Sie eine Methode gleich direkt in der Klasse definieren, in der Praxis ist dies jedoch unüblich. Ein Beispiel:

```
class Automat {
  // Daten der Klasse "Automat"
  unsigned long geld{OL};
  std::string standort{"Unbekannt"};
  // Definition der Methode in der Klasse
  unsigned long get_geld() {
    // Anweisungen für die Methode
  }
  ...
};
```

Implizit »inline«

Methoden, die innerhalb einer Klasse definiert werden, werden implizit zu `inline`-Methoden (auch ohne das Schlüsselwort `inline`). Damit schlagen Sie dem Compiler vor, die Methode nicht über den Stack-Frame aufzurufen, sondern direkt im Code zu verwenden.

In der Praxis wird eine Methode gewöhnlich der Übersichtlichkeit halber außerhalb der Klassendefinition festgelegt. Allerdings genügt es nicht mehr, bei der Definition nur den Funktionsnamen anzugeben. Hierfür

müssen Sie den Klassennamen, gefolgt vom Scope-Operator (Bereichsoperator) :: und dem eigentlichen Funktionsnamen, verwenden. Die Syntax hierzu:

```
Typ Klassenname::funktionsname( parameter ) {
   // Anweisungen der Funktion
}
```

Hierdurch teilen Sie dem Compiler mit, dass die Funktion `funktionsname` eine Methode der Klasse `Klassenname` ist. Würden Sie den Klassennamen hierbei nicht verwenden, hätten Sie nur eine gewöhnliche Funktion definiert.

Bezogen auf die Klasse `Automat` und die Methode `set_geld()` sieht eine Definition außerhalb der Klasse wie folgt aus:

```
void Automat::set_geld(unsigned long g){
  geld = g;
}
```

Hierbei können Sie sehen, dass Sie auf die Daten einer Klasse innerhalb der Methode direkt ohne irgendwelche Scope-Operatoren oder Punkteoperatoren zugreifen können. Dies funktioniert auch, wenn Methoden einer Klasse andere Methoden derselben Klasse aufrufen. Somit kann man sagen, dass alle Daten und Methoden innerhalb einer Klasse miteinander harmonieren und Methoden ohne Umwege auf die Daten und andere Methoden zugreifen können.

Hierzu die komplette Implementierung des ersten Quellcodes *automat.cpp*:

```
00  // listings/011/automat1/automat.cpp
01  #include <iostream>
02  #include <string>
03  #include "automat.h"

04  unsigned long Automat::get_geld() {
05    return geld;
06  }
```



```
07  void Automat::set_geld(unsigned long g){
08    geld = g;
09  }

10  const std::string& Automat::get_standort() {
11    return standort;
12  }

13  void Automat::set_standort(const std::string& s){
14    standort = s;
15  }

16  void Automat::print() {
17    std::cout << '\n' << "Geldautomat : ";
18    std::cout << get_standort() << std::endl;
19    std::cout << "Geld (Euro) : ";
20    std::cout << get_geld() << std::endl;
21  }
```

Auf die Einzelheiten, wie den Zugriff auf die Methoden, gehe ich in Abschnitt 11.1.5 noch gesondert ein.

11.1.3 Zugriffskontrolle mit »public« und »private«

Mit unserer bisherigen Klassendefinition `Automat` könnten Sie zwar ein Objekt anlegen, aber Sie haben noch keinerlei Zugriff von außen auf die Klasse. Wie bereits erwähnt, sind alle Daten und Methoden innerhalb einer Klasse – im Gegensatz zu einer Struktur – zunächst `private`, wenn Sie nichts anders angeben. `private`-Daten und -Methoden können Sie somit nur innerhalb einer Klasse verwenden.

Würden Sie jetzt ein Objekt der Klasse `Automat` erstellen und versuchen, eine Methode zu verwenden, würde der Compiler die Übersetzung verweigern, mit der Meldung, dass diese Methode `private` ist. Standardmäßig ist ein Zugriff ohne weitere Angaben auf Klassen von außen verboten. Der Grund solcher Rechte ist, dass der Zugriff auf die Daten einer Klasse nur noch kontrolliert erfolgen soll. So gehören beispiels-

weise die falsche Übergabe von Argumenten oder nicht initialisierte Variablen der Vergangenheit an (oder werden zumindest bei richtiger Anwendung eingedämmt).

Datenkapselung nicht aufweichen

Beachten Sie dabei bitte Folgendes: Der Schutz der Daten kann jederzeit umgangen werden. Der Schutz der Datenkapselung verhindert Änderungen der Daten, aber bewahrt nicht vor mutwilliger Aufweichung des Konzepts der Datenkapselung. C++ liefert Ihnen nur die Mittel; für eine saubere Implementierung von eigenen Klassentypen sind nach wie vor Sie als Programmierer verantwortlich.

Eine solche komplette Sperrung einer Klasse ergibt von außen wenig Sinn. Um auf einzelne Methoden einer Klasse von außen zuzugreifen, müssen Sie das Schlüsselwort `public` (das Gegenstück zu `private`) verwenden. Alle Elemente, die mit `public` gekennzeichnet sind, können Sie über das Objekt außerhalb der Klasse verwenden. Die Syntax dazu:

```
class Klassenname {
   // Auf Elemente kann nur innerhalb einer Klasse
   // zugegriffen werden
   private:
   typ daten1;
   typ daten2;
   ...

   // Zugriff von außen auf die Elemente möglich
   public:
   typ funktionsname1( parameter );
   typ funktionsname2( parameter );
   ...
};
```

Alle Elemente, die hinter `private` (der Doppelpunkt muss angegeben werden) stehen, sind nur innerhalb der Klasse sichtbar. Von außen kann darauf nicht zugegriffen werden. Diese Rechteerteilung gilt bis zum Ende der

Klasse – oder bis zum Schlüsselwort `public` (auch hier ist der Doppelpunkt dahinter wichtig). Ab dem Schlüsselwort `public` ist alles dahinter Geschriebene öffentlich außerhalb der Klasse zugänglich. Dabei spielt es keine Rolle, ob Sie zuerst das Klassenelement `public` oder `private` oder gar beide gemischt verwenden. Es ist auch nicht von Bedeutung, wie oft Sie `public` oder `private` in einer Klasse benutzen und ob Sie dabei Daten und/oder Methoden öffentlich oder privat zugänglich machen.

In der Praxis werden in der Regel als Entwurfsmuster die Daten einer Klasse als `private` verwendet und die Methoden als `public` (öffentlich) deklariert. Damit lässt sich der Zugriff auf die Daten über die Schnittstellen (Methoden) kontrollieren.

Mit diesem Wissen können Sie die Klassendefinition abschließen. Unsere Headerdatei *automat.h* sieht somit wie folgt aus:

```
00  // listings/011/automat1/automat.h
01  #ifndef _AUTOMAT_H_
02  #define _AUTOMAT_H_
03  #include <string>

04  class Automat {
05    private:
06      // Daten der Klasse "Automat"
07      unsigned long geld{0L};
08      std::string standort{"Unbekannt"};
09    public:
10      // Methoden der Klasse "Automat"
11      unsigned long get_geld();
12      void set_geld(unsigned long g);
13      const std::string& get_standort();
14      void set_standort(const std::string& s);
15      void print();
16  };
17  #endif
```

Die Bereiche hinter der geschweiften Klammer einer Klassendefinition, die mit keinem der Schlüsselwörter `private` oder `public` versehen wurden,

sind `private`. In der Headerdatei *automat.h* könnten Sie somit auf Zeile **(05)** komplett verzichten, weil die Daten an dieser Stelle ohnehin `private` sind. Der Bereich der Klassendefinition ab Zeile **(09)** ist öffentlich von außen zugänglich.

11.1.4 Zugriff auf die Daten innerhalb einer Klasse

Bevor Sie ein Objekt der Klasse `Automat` in einer `main()`-Funktion erzeugen und benutzen, möchte ich noch einen Abschnitt dem wichtigen Thema des Zugriffs auf die Daten einer Klasse widmen.

Wie Sie bereits erfahren haben, können Sie außerhalb der Klasse mit den Objekten einer Klasse nur auf die `public`-Elemente zugreifen. Die Daten sind in der Regel `private` und nicht von außen zu erreichen. Es ist daher gängige Praxis, die Daten einer Klasse nur über Methoden zu ermitteln und zu ändern.

Bezogen auf unsere Klasse `Automat` können Sie die Daten `geld` oder `standort` nur über Methoden (in dem Fall auch als *Zugriffsfunktionen* bezeichnet) setzen oder abfragen. Im Beispiel der Daten `geld`:

```
00  // listings/011/automat1/automat.cpp
...
01  unsigned long Automat::get_geld() {
02    return geld;
03  }

04  void Automat::set_geld(unsigned long g){
05    geld = g;
06  }
```

Da die Daten bei der Klassendefinition in demselben Gültigkeitsbereich definiert wurden, brauchen Sie hier keinen Punkte-, Pfeil- oder Scope-Operator zu verwenden, wie Sie es in Zeile **(02)** und **(05)** sehen können.

In der Praxis sind solche Methoden meistens nicht auf einfache Zuweisungen oder Wertrückgaben, wie im Beispiel abgedruckt, beschränkt. In der Regel sind hier noch Überprüfungen implementiert, ob beispielsweise ein gültiger Wert zugewiesen wurde.

11.1.5 Objekte erzeugen und benutzen

Jetzt haben Sie zwar eine Klasse `Automat` geschaffen, aber eine solche Klasse zu haben ist zunächst nur so viel wie die Vorstellung eines Bankautomaten im Kopf. Und allein Gedanken an einen Bankautomaten machen diesen noch nicht real.

Sie müssen ein Objekt (auch als *Instanz* einer Klasse bezeichnet) erzeugen. Klassen und Objekte? – Bevor Sie etwas durcheinanderbringen: Eine Klasse ist zunächst nichts anderes als der bloße Gedanke an einen Gegenstand (eine Skizze des Gegenstands, wenn Sie so wollen). Wenn wir von einem Objekt selbst reden, handelt es sich um einen realen Gegenstand, der sich vor Ihren Augen befindet (natürlich nicht wirklich vor Ihren Augen, aber im Hauptspeicher ist das Objekt real). Die Definition eines solchen Objekts verläuft so, wie Sie dies von den eingebauten Basistypen und den anderen fortgeschrittenen Typen wie den Strukturen her kennen:


```
Klassenname Objekt{};
```

Auf unsere Klasse `Automat` bezogen sieht dies wie folgt aus:

```
Automat device01{};
```

Sie können auch mehrere Objekte anlegen:

```
Automat device01{}, device02{};
```

Damit reservieren Sie gleich Speicher für die Daten der Objekte `device01` und `device02`. Im Beispiel sind dies die Attribute `geld` und `standort`.

Da wir in unserem Beispiel bei der Erstellung der Klasse `Automat` die Daten mit einem Standardwert (`geld=0L` und `standort="Unbekannt"`) versehen haben, was seit C++11 funktioniert, haben die beiden Elemente bereits einen Inhalt. Da `standort` eine vollständige Klasse vom Typ `std::string` ist, hat dieser Typ von Haus aus den Inhalt `""`. Hätten Sie die Klassendefinition ohne einen Standardwert für die Daten erstellt, wie dies vor C++11 der Fall gewesen ist, wäre mit einem leeren

```
Automat device01; // ohne Initialisierungsliste !!!
```

ohne eine leere Initialisierungsliste mit {} der Inhalt von geld nur ein zufälliger, nicht vorhersehbarer Wert. Auf das Thema der Initialisierung von Daten einer Klasse gehe im nächsten Abschnitt noch ausführlicher ein. Es ist nicht falsch, wenn Sie sich angewöhnen, die leere Initialisierungsliste mit {} zu verwenden.

Dies gilt für jedes Objekt. Es spricht nichts dagegen, ein Array von Objekten der Klasse Automat anzulegen:

```
#include <vector>
...
// 10 Objekte der Klasse "Automat"
std::vector <Automat> device_germany(10);
// Ein leeres Array der Klasse "Automat"
std::vector <Automat> device_austria{};
// ... auch die C-Arrays lassen sich hiermit verwenden
Automat Cdevice[10]={};
```

Jedes Objekt hat seinen eigenen Speicher für die Daten. Allerdings arbeiten alle Objekte mit denselben Methoden. Daher wird der Code für die Methoden sinnvollerweise nur einmalig im Speicher abgelegt.

Zugriff mit dem Punkteoperator

Wenn Sie ein Objekt definiert haben, können Sie den Punkteoperator verwenden, um auf die public-Elemente direkt zuzugreifen (wie schon bei den Strukturen mit struct). Im Grunde können Sie auf die Klassenelemente nur mit einem Objekt der Klasse zugreifen. Dieser Zugriff auf ein Element erfolgt durch die Bezeichnung des Objekts, gefolgt vom Punkteoperator und vom Bezeichner des Klassenelements. Die Syntax lautet:

```
// Objekt erzeugen
Klassenname Objekt{};
// Zugriff auf public-Daten der Klasse (nicht zu empfehlen)
Objekt.Daten = 1234;
// Zugriff auf public-Methoden der Klasse
Objekt.Funktion( parameter );
```


In unserem Beispiel Automat können Sie daher einen Bankautomat folgendermaßen erzeugen und den Inhalt ausgeben:

```
//  Objekt der Klasse "Automat" erzeugen
Automat device{};
//  Daten des Objektes ändern
device.set_geld(20000);
device.set_standort("Augsburg, Fuggerweg 345");
// Inhalt des Objektes ausgeben
device.print(10000, "Bonn, Hauptstrasse 123");
```

Im Beispiel wurde die Methode print() zur Ausgabe der Daten der Klasse Automat verwendet:

```
00  // listings/011/automat1/automat.cpp
...
16  void Automat::print() {
17    std::cout << '\n' << "Geldautomat : ";
18    std::cout << get_standort() << std::endl;
19    std::cout << "Geld (Euro) : ";
20    std::cout << get_geld() << std::endl;
21  }
```

Die Methode print() der Klasse Automat wiederum ruft zwei Methoden auf (Zeile **(18)** und **(20)**), durch die der Inhalt der entsprechenden Daten des Objekts zurückgegeben wird.

Wenn Sie versuchen, direkt auf die Daten der Klasse Automat zuzugreifen, wird der Compiler dies mit einer Fehlermeldung quittieren. Folgender Codeausschnitt ist nicht möglich (bzw. sollte niemals möglich sein):

```
Automat device{};
// !!! Fehler, geld ist ein privat-Element !!!
std::cout << device.geld << std::endl;
// !!! Fehler, standort ist ein privat-Element !!!
std::cout << device.standort << std::endl;
// !!! Fehler, Zugriff auf ein privat-Element !!!
device.geld = 12345L;
```

Ein Zugriff auf ein mit `private` gekennzeichnetes Element ist weder lesend noch schreibend möglich. Die einzige theoretische Möglichkeit wäre es daher, das mit `private` gekennzeichnete Element mit `public` zu kennzeichnen. Allerdings würden Sie hiermit das OOP-Prinzip aufweichen. Wenn Sie einmal in die Verlegenheit kommen sollten, Daten notgedrungen als `public` zu kennzeichnen, haben Sie einen Designfehler in der Planung gemacht und sollten Ihren Code überdenken und überarbeiten.

```
...
class Automat {
  // So etwas ist möglich, sollte aber unbedingt vermieden
  // werden. Damit wird das Prinzip des Klassendesigns
  // aufgeweicht !!!
  public:
    // !!! Daten der Klasse "Automat" sind jetzt public !!!
    unsigned long geld{0L};
    string standort{"Unbekannt"};
...
};
```

Indirekter Zugriff mit dem Pfeiloperator

Wie bei den Zeigern auf Strukturen können Sie mit Zeigern auf Objekte von Klassen arbeiten. Der indirekte Zugriff wird mit dem Objektzeiger realisiert, gefolgt vom Pfeiloperator und vom Bezeichner eines Elements. Sie müssen vor der Verwendung des Objektzeigers zunächst auf eine gültige Adresse verweisen, bevor Sie ihn verwenden.

In der Praxis wird dies gerne zum dynamischen Erzeugen von Objekten zur Laufzeit verwendet. Wenn Sie eine dynamische Speicherreservierung benötigen, sollten Sie, wenn möglich, die neuen Smart Pointer von C++11 verwenden. Im konkreten Fall würde ich Ihnen daher zu `std::unique_ptr` raten. Der Vorteil damit ist: Sie müssen sich nicht mehr um die Freigabe des Speichers kümmern.

Bezogen auf den Bankautomaten könnten Sie folgendermaßen zur Laufzeit dynamisch Speicher für ein neues Objekt anfordern und indirekt mit dem Pfeiloperator darauf zugreifen:


```
#include <memory> // für std::unique_ptr
...
// Speicher für ein neues Objekt anfordern
std::unique_ptr<Automat> device_ptr(new Automat{});
device_ptr->set_standort("Mainz, Hauptstrasse 1");
device_ptr->print();
...
```

Verzichten Sie auf rohe Zeiger und »new«

Wenn es irgendwie geht, sollten Sie bei Objekten auf die dynamische Speicherreservierung mit rohen Zeigern ganz verzichten und stattdessen auf eine der vielen Containerklassen wie beispielsweise `std::vector` zurückgreifen. Zum einen machen Sie sich damit das Leben erheblich einfacher, und zum anderen hat die Vergangenheit gezeigt, dass die Verwendung von rohen Zeigern, kombiniert mit `new` und `delete`, zu vielen Fehlern (Stichwort: *Memory Leaks*) geführt hat. Wenn Sie trotzdem eine dynamische Speicherreservierung benötigen, sollten Sie, wenn möglich, die neuen Smart Pointer von C++11 verwenden.

Hierzu, wie angekündigt, die `main()`-Funktion zur Headerdatei *automat.h* und zur Quelldatei *automat.cpp*, um die erste einfache Verwendung von Objekten mit der Klasse `Automat` in der Praxis zu demonstrieren:

```
00  // listings/011/automat1/main.cpp
01  #include <iostream>
02  #include <vector>
03  #include <string>
04  #include <memory>
05  #include "automat.h"

06  int main(void) {
07    std::vector <Automat> Vdevice(2);
08    Vdevice[0].set_geld(20000);
09    Vdevice[0].set_standort("Augsburg, Fuggerweg 345");
10    Vdevice[1].set_geld(25000);
11    Vdevice[1].set_standort("Berlin, Berliner Allee 567");
```

```
12    // Ausgabe aller Elemente
13    for( auto &elem : Vdevice ) {
14      elem.print();
15    }
16    // Speicher dynamisch zur Laufzeit reservieren
17    std::unique_ptr<Automat> device_ptr(new Automat{});
18    device_ptr->set_standort("Mainz, Hauptstrasse 1");
19    device_ptr->print();

20    Automat device{};
21    device.print();
22    return 0;
23  }
```

Das Beispiel zeigt in der Praxis in mehreren Variationen, die ich in den Abschnitten zuvor beschrieben habe, wie Sie ein Objekt der Klasse `Automat` erzeugen und verwenden können.

11.2 Konstruktoren

Bis jetzt wurden die Daten der Klasse `Automat` gleich direkt mit den Standardwerten versehen, die Sie bei der Definition des Klassentyps angegeben haben:

```
class Automat {
  private:
    unsigned long geld{0L};
    std::string standort{"Unbekannt"};
...
};
```

Diese Form, die Mitglieder einer Klasse mit Daten zu versehen, ist recht praktisch, weil so nicht mit undefinierten Werten gearbeitet wird.

Folgenden Versuch, ein Objekt zu erzeugen, würde der Compiler mit einer Fehlermeldung quittieren:


```
// !!! Der Compiler kann hiermit nichts anfangen !!!
Automat device02{"Freiburg, Bahnhofweg 2"};
```

Klassen bieten hierfür einen speziellen Mechanismus an, der dafür verantwortlich ist, dass ein Objekt bei der Definition automatisch mit einer Initialisierungsfunktion mit Werten für die Daten belegt wird. Damit ist garantiert, dass Sie mit gültigen Werten arbeiten. Die Rede ist von den *Konstruktoren* (im Englischen auch kurz *ctor* genannt), die prinzipiell den Methoden einer Klasse recht ähnlich sind.

Doch in drei Bereichen unterscheiden sich Konstruktoren von Klassenmethoden:

- Der Name des Konstruktors ist derselbe wie der Name der Klasse.
- Der Konstruktor besitzt keinen Rückgabewert (auch nicht `void`).
- Vor dem Block {} des Konstruktors können (und sollten) die Daten der Klasse mit : `data1{wert}`, `data2{wert}`, ... initialisiert werden.

11.2.1 Konstruktoren deklarieren

Damit der Konstruktor zur Erzeugung von Objekten von außen zur Verfügung steht, erfolgt die Deklaration gewöhnlich im `public`-Bereich der Klasse. Hierzu müssen Sie zunächst die Konstruktoren in der Klassendefinition `Automat` deklarieren. Die einzelnen Konstruktoren (im Beispiel sind es vier) unterscheiden sich durch ihre Parameter:

```
// listings/011/automat2/automat.h
#ifndef _AUTOMAT_H_
#define _AUTOMAT_H_
#include <string>

class Automat {
  private:
    // Daten der Klasse "Automat"
    unsigned long geld{0L};
    std::string standort{"Unbekannt"};
  public:
    // Deklaration der Konstruktoren
```

```
    Automat();
    Automat(unsigned long, std::string);
    Automat(unsigned long);
    Automat(std::string);
    // Methoden der Klasse "Automat"
    unsigned long get_geld();
    ...
};
#endif
```

Ja, Sie haben es richtig erkannt, die Konstruktoren arbeiten nach demselben Prinzip der **Funktionsüberladung**, wie Sie dies in Abschnitt 8.1.8 kennengelernt haben. Daher ist es wichtig, wenn Sie mehrere Konstruktoren verwenden wollen, dass sich diese durch ihre Parameter (genauer: Signatur) unterscheiden. Dadurch lassen sich die Daten der Objekte auf unterschiedliche Arten initialisieren.

11.2.2 Konstruktoren definieren

Nachdem Sie die Deklaration der Konstruktoren in die Headerdatei geschrieben haben, müssen Sie diese Konstruktoren, wie auch die Klassenmethoden, definieren. Die Definition ähnelt ebenfalls den Klassenmethoden, mit dem Unterschied, dass hierbei zweimal der Klassenname, gefolgt von den Parametern, verwendet wird:

```
Klassenname::Klassenname( )
: data1{wert}, data2{wert}, dataN{wert}  {
  // Konstruktionskörper mit Anweisungen
}
```

Wenn Sie hier eine Initialisierung in Form von `Klassenname name{};` vornehmen, wird dieser Konstruktor ausgeführt. Bevor hierbei der Konstruktorkörper ausgeführt wird, werden zunächst noch alle Variablen mit den angegebenen Werten über einen *Elementinitialisierer* initialisiert. Eine solche Liste von Elementinitialisierern wird mit einem Doppelpunkt noch vor dem Funktionskörper eingeleitet, und die einzelnen Elemente werden mit einem Komma getrennt.

Bezogen auf unsere Definition der Klasse `Automat` sieht die Definition der Konstruktoren im Quellcode *automat.cpp* wie folgt aus:

```
// listings/011/automat2/automat.cpp
#include <iostream>
#include <string>
#include "automat.h"

// Definition der Konstruktoren

Automat::Automat()
: geld{0L}, standort{""} { }

Automat::Automat(unsigned long g, std::string s)
: geld{g}, standort{s} { }

Automat::Automat(unsigned long g)
: geld{g}, standort{""} { }

Automat::Automat(std::string s)
: geld{0L}, standort{s} { }

unsigned long Automat::get_geld() {
  return geld;
}

// Definition der Methoden
...
```

Je nachdem, welches Objekt instanziiert wird, wird der entsprechende Konstruktor aufgerufen und/oder das Objekt mit Daten oder Nullwerten initialisiert. Im Beispiel werden diese Werte über Elementinitialisierer mit Werten versehen. Hierzu nochmals ein solcher Konstruktor bei der Definition mit Elementinitialisierern in etwas übersichtlicherer Form:

```
Automat::Automat(unsigned long g, std::string s)
: geld{g},      // Initialisierungswert für geld
```

```
  standort{s}   // Initialisierungswert für standort
{ }             // Leerer Funktionskörper
```

Dank dieser Konstruktoren können Sie folgendermaßen Objekte erzeugen und direkt mit Zugriffsfunktionen verwenden, ohne befürchten zu müssen, mit nicht initialisierten Werten zu arbeiten:

```
// listings/011/automat2/main.cpp
...
  // Automat()
  Automat device01{};
  device01.print();
  // Automat(std::string)
  Automat device02{"Freiburg, Bahnhofweg 2"};
  device02.print();
  // Automat(unsigned long, std::string)
  Automat device03{20000, "Fulda, Hauptstrasse 1"};
  device03.print();
  // Automat(unsigned long)
  Automat device04{10000};
  device04.print();
  // Automat(unsigned long, std::string) - Alter Stil
  Automat device05("Augsburg, Berliner Allee 1");
  device05.print();
  // Automat(unsigned long) - Alter Stil vor C++11
  Automat device06(12345);
  device06.print();
...
```

Schnell ist man als Programmierer versucht, seiner Klasse möglichst viel Flexibilität zu verleihen und daher möglichst viele Konstruktoren zu implementieren, um alle Möglichkeiten abzudecken. An manchen Stellen sollten Sie überlegen, ob Sie nicht lieber statt einer Serie von Konstruktoren den Einsatz von Standardparametern (Default-Argumenten) in Erwägung ziehen sollten.

Neben den Initialisierungen von Variablen werden Konstruktoren auch für andere Dinge verwendet. Gängige Beispiele sind das Reservieren von Speicher, das Öffnen von Dateien oder das Vorbereiten von Daten.

Folgendes lässt sich übrigens ebenfalls realisieren, wenn ein entsprechender Konstruktor vorhanden ist:

```
// Automat(unsigned long)
Automat device07 = 25000;
device07.print();
```

Hiermit wird der Konstruktor `Automat(unsigned long)` aufgerufen.

11.2.3 Implizite Konvertierungen verhindern – »explicit«

Wie bereits zuvor gezeigt, kann ein Konstruktor mit nur einem Parameter per Zuweisung ein Objekt erzeugen. Solche Konstruktoren werden als *Konvertierungskonstruktoren* bezeichnet. Damit ist es möglich, mit folgenden Zeilen ein Objekt anzulegen, obwohl überhaupt kein Konstruktor für `Automat(double)` vorhanden ist:

```
01  Automat device01(123.123);    // Ok
02  Automat device02{345.345};    // Narrowing -> C++11
03  Automat device03 = 456.456;   // Ok
```

Zeile (**01**) erstellt ein neues Objekt mithilfe einer expliziten Konvertierung. In Zeile (**02**) wird eine einheitliche Initialisierung verwendet, und daher wird eine Warn- oder Fehlermeldung (abhängig vom Compiler und von der Einstellung) zu einem Narrowing ausgegeben (siehe Abschnitt 4.5.2, »Automatische Typumwandlung beschränken«) – ein weiterer Vorteil der einheitlichen Initialisierung. Bei der Verwendung des Zuweisungsoperators in Zeile (**03**) wird hingegen eine implizite Konvertierung durchgeführt und ein neues Objekt erzeugt.

Eine solche implizite Konvertierung wie in Zeile (**03**) ist allerdings nicht immer erwünscht. Wollen Sie keine solche implizite Konvertierung mit nur einem Parameter zulassen, müssen Sie dies mit dem Schlüsselwort `explicit` einfordern. Hierbei müssen Sie lediglich bei der Deklaration das Schlüsselwort `explicit` wie folgt voranstellen:

```
// listings/011/automat2/explicit/automat.h
...
  public:
    explicit Automat(unsigned long);
...
```

Jetzt schlägt das Schlüsselwort `explicit` zu, wenn Sie in Zeile (**03**) eine implizite Konvertierung durchführen wollen:

```
// listings/011/automat2/explicit/main.cpp
...
01  Automat device01(123.123);    // OK
02  Automat device02{345.345};    // Narrowing -> C++11
03  Automat device03 = 456.456;   // Fehler !!!
```

Wegen des Schlüsselwortes `explicit` kommen Sie in Zeile (**03**) nicht darum herum, eine explizite Konvertierung bei Bedarf durchzuführen.

11.2.4 Optimierung 1: Klassenelemente gleich direkt initialisieren

Neben dem Vorteil, nicht mehr mit nicht initialisierten Daten hantieren zu müssen, sparen Sie sich mit dem *direkten Initialisieren* von Klassenelementen (Daten) seit C++11 eine Menge an Tipparbeit, weil Sie beispielsweise auf die Initialisierungen beim Standardkonstruktor verzichten können.

```
00  // listings/011/automat2/member-default/automat.h
...
01  class Automat {
02    private:
03      unsigned long geld{0};
04      string standort{""};
05    public:
06      Automat();
07      Automat(unsigned long, string);
08      Automat(unsigned long);
09      Automat(string);
...
10  };
```


Hier werden die Klassenelemente `geld` und `standort` in Zeile (**03**) und (**04**) direkt initialisiert, was Ihnen bei den Konstruktoren einiges an Tipparbeit erspart:

```
00  // listings/011/automat2/member-default/automat.cpp
...
01  // Definition der Konstruktoren

02  Automat::Automat() { }

03  Automat::Automat(unsigned long g, std::string s)
04    : geld{g}, standort{s} { }

05  Automat::Automat(unsigned long g)
06    : geld{g} { }

07  Automat::Automat(std::string s)
08    : standort{s} { }
...
```

Beim Standardkonstruktor in Zeile (**02**) können Sie dank dieser direkten Initialisierung auf beide Elementinitialisierer verzichten. Beim Konstruktor `Automat(unsigned long)` (Zeile (**05**) bis (**06**)) können Sie auf die Initialisierung von `standort` und beim Konstruktor `Automat(std::string)` (Zeile (**07**) bis (**08**)) auf die Initialisierung von `geld` verzichten, weil diese Werte bereits per Default für jedes neue Objekt initialisiert wurden.

11.2.5 Optimierung 2: Konstruktoren delegieren

Noch mehr Tipparbeit können Sie sich ersparen, wenn Sie nur einen zentralen Konstruktor definieren, der von allen anderen Konstruktoren aufgerufen wird. Die Rede ist hierbei von einer **Delegation von Konstruktoren**. Der folgende Codeausschnitt soll dieses neue Feature etwas deutlicher zeigen:

```
00  // listings/011/automat2/delegation/automat.h
...
01  class Automat {
```

```
02    private:
03      // Daten der Klasse "Automat"
04      unsigned long geld{0L};
05      std::string standort{""};
06    public:
07      // Deklaration der Konstruktoren
08      Automat():Automat(0,""){};
09      Automat(unsigned long g):Automat(g,""){};
10      Automat(std::string s):Automat(0L,s){};
11      Automat(std::string s,unsigned long g):Automat(g,s){};
12      Automat(unsigned long, std::string);
...
13  };
```

In Zeile (**08**) finden Sie die erste Delegation von `Automat()` nach `Automat(unsigned long, string)`. Diese Delegation wird aufgerufen, wenn Sie ein Objekt folgendermaßen erzeugen:

```
Automat device01{};  // Automat()->Automat(0, "")
```

Auch `Automat(unsigned long)` und `Automat(std::string)` wurden in Zeile (**09**) und (**10**) nach `Automat(unsigned long, string)` delegiert:

```
Automat d02{11}; // Automat(unsigned long)->Automat(11, "")
Automat d03{"Fulda"}; // Automat(string)->Automat(0L, "Fulda")
```

Hier wurde in Zeile **(11)** nach `Automat(unigned long, std::string)` auch eine Delegation für `Automat(std::string, unsigned long)` eingeführt, womit Sie ein Objekt in beliebiger Reihenfolge aufrufen können:

```
Automat device07{"Frankfurt, Am Main 1", 250000};
Automat device08{250000, "Frankfurt, Am Main 1"};
```

Wenn Sie die Delegationen von Konstruktoren hier im Code verfolgt haben, dürfte Ihnen aufgefallen sein, dass alle Konstruktoren auf den Konstruktor `Automat(unsigned long, std::string)` (Zeile (**12**)) delegiert wurden. Daher müssen Sie gegebenenfalls lediglich Code für diesen einen Konstruktor schreiben:


```
00  // listings/011/automat2/delegation/automat.cpp
...
01  Automat::Automat(unsigned long g, std::string s)
02    : geld{g}, standort{s} { }
...
```

Dank der Delegation von Konstruktoren ist nur noch Code für einen Konstruktor erforderlich, wie ich ihn hier in Zeile (**01**) bis (**02**) definiert habe.

Wichtig in diesem Zusammenhang ist auch, dass eine solche Delegation eines Konstruktors nur innerhalb des Elementinitialisierers, direkt nach dem Doppelpunkt (:), funktioniert. Im Konstruktorkörper sollte diese Delegation nicht stehen, weil sonst ein neues temporäres Objekt erzeugt würde. Und das wollen Sie sicher nicht wirklich erzielen.

11.2.6 Standardkonstruktor (Default-Konstruktor)

Konstruktoren ohne Parameter heißen *Standardkonstruktoren* (*Default-Konstruktoren*). Sie werden aufgerufen, wenn bei der Definition eines Objekts keine weiteren Initialisierungswerte angegeben wurden, beispielsweise:

```
// Verwendet Standardkonstruktor
Klassenname Objekt;
Klassenname Objekt {};  // C++11
```

In der Praxis schreibt man diesen Standardkonstruktor gewöhnlich so, dass alle Eigenschaften einer Klasse mit Standardwerten versehen werden, ausgenommen, Sie verwenden die direkte Initialisierung von Klassenelementen, die mit C++11 eingeführt wurde. Im Beispiel zuvor sahen die Deklaration und die Definition des Standardkonstruktors (abgesehen von der Delegation) wie folgt aus:

```
// automat.h
...
// Deklaration des Standardkonstruktors
Automat();
```

```
// automat.cpp
...
// Definition des Standardkonstruktors
Automat::Automat()
: geld{0L}, standort{""} { }
```

Dieser Konstruktor wurde verwendet, wenn ein Objekt folgendermaßen erzeugt wurde:

```
// Beide Male wird der Standardkonstruktor aufgerufen
Automat device01;
Automat device02{};  // C++11
unique_ptr<Automat> device03{ new Automat{} }; // C++11
```

Sofern Sie keinen Standardkonstruktor verwenden (was Sie in der Praxis allerdings immer machen sollten), stellt Ihnen der Compiler automatisch einen solchen zur Verfügung. Dieser Standardkonstruktor (ebenfalls von außen erreichbar – als `public`) initialisiert allerdings **nicht** die Daten der Klasse mit gültigen Werten.

Beachten Sie dabei, dass der Compiler Ihnen, sobald Sie mindestens einen Konstruktor deklariert und definiert haben, keinen Standardkonstruktor mehr zur Verfügung stellt und Sie sich selbst darum kümmern müssen. Entfernten Sie den Standardkonstruktor im Beispiel `Automat`, würde folgende Objektdefinition zu einem Compilerfehler führen, weil der Compiler keinen passenden Konstruktor dafür finden könnte:

```
Automat device04{};
```

Dies hätte allerdings den Vorteil, dass Sie gezwungen wären, bei der Erzeugung eines neuen Objekts eine Initialisierungsliste mit anzugeben.

11.2.7 Kopierkonstruktor

Ein Konstruktor, den der Compiler ebenfalls von Haus aus zur Verfügung stellt, ist der Kopierkonstruktor. Gesetzt den Fall, zwei Objekte seien von derselben Klasse `Automat`, dann würde eine Initialisierung von `a1=a2` oder `a1{a2}` ein elementweises Kopieren der Daten von `a2` nach `a1` bedeuten, zum Beispiel:


```
Automat device01{12000, "Bonn, Rheinwerkallee"};
// Kopie von device01 nach device02
Automat device02 = device01;
// Kopie von device 02 nach device03
Automat device03{device02};
```

Die Verwendung des Gleichheitszeichens `=` ist hier bei der Initialisierung ein Sonderfall und optional. Das heißt in diesem konkreten Fall, dass mit `=` hier tatsächlich der Kopierkonstruktor aufgerufen wird und nicht der Zuweisungsoperator. Somit ist in diesem Beispiel `Automat device03{device02}` identisch mit `Automat device03 = device02`.

Falls das Kopieren nicht zum gewünschten Verhalten führt – was beispielsweise dann der Fall sein kann, wenn eines der Attribute ein Zeiger ist, der dynamisch zur Laufzeit Speicherplatz anfordert –, können Sie einen eigenen Kopierkonstruktor definieren. Ein solcher Kopierkonstruktor hat folgende Syntax:


```
Klassenname ( const Klassenname & );
```

Die Deklaration des Kopierkonstruktors in der Headerdatei *automat.h* als `public()`-Methode der Klasse `Automat` sähe wie folgt aus:

```
// listings/011/automat2/copyCtor/automat.h
...
   public:
   ...
   // Kopierkonstruktor
   Automat( const Automat & );
   ...
```

Hierzu die Definition des Kopierkonstruktors in *automat.cpp*:

```
00  // listings/011/automat2/copyCtor/automat.cpp
...
// Definition Kopierkonstruktor
01  Automat::Automat(const Automat &a)
02    : geld{a.geld}, standort{a.standort} { }
...
```

Damit das neue Objekt dieselben Daten wie das Original bekommt, müssen Sie hier die privaten Daten Objekt für Objekt kopieren, wie Sie es in Zeile (02) mit dem Elementinitialisierer sehen können. Wie bereits erwähnt, können wir uns diesen Kopierkonstruktor hier sparen, weil keine speziellen Klassenelemente wie beispielsweise ein roher Zeiger vorhanden waren und somit der Standard-Kopierkonstruktor hier ausreicht.

11.2.8 Verschiebekonstruktor (Move-Konstruktor)

Es gibt Objekte, die Sie nicht kopieren sollten. Das einfachste Beispiel wären Objekte mit geöffneten Dateien oder Sperren. Würden Sie solche Objekte kopieren und würde jedes dieser Objekte in eine Datei schreiben, diese sperren oder eine Sperre aufheben, käme es mit Sicherheit irgendwann zum Datensalat. Für solche Zwecke steht Ihnen seit C++ eine Verschiebe-Semantik (Move-Semantik) zur Verfügung. Auch sind solche Verschiebeoperationen viel performancesparender als Kopieroperationen. Das Thema selbst erläutere ich erst in Abschnitt A.1.3, »RValue (neue Move-Semantik)«, detaillierter. Hier in diesem Abschnitt konzentrieren wir uns zunächst ganz darauf, eine solche Verschiebe-Semantik in Klassen einzubauen.

»Kopieren und Einfügen« versus »Ausschneiden und Einfügen«

Vereinfacht dargestellt können Sie sich den Kopierkonstruktor als einen »Kopieren und Einfügen«-Mechanismus und den Verschiebekonstruktor als »Ausschneiden und Einfügen«-Mechanismus vorstellen.

Der Verschiebekonstruktor ist ein Verwandter des Kopierkonstruktors, bei dem das Argument mit `&&` gekennzeichnet wird. Ein `&&`-Argument stellt somit ein temporäres Objekt dar, das in Kürze gelöscht wird. Im Gegensatz zum Kopierkonstruktor darf der Parameter nicht mehr `const` sein, weil zur Erzeugung eines neuen Objekts die Daten aus dem Objekt »gestohlen« werden sollen, sodass das alte Objekt keine Daten mehr enthält oder zumindest »leer« ist.

Ein solcher Verschiebekonstruktor hat folgende Syntax:

```
Klassenname ( Klassenname && );
```


Die Deklaration des Verschiebekonstruktors in der Headerdatei *automat.h* als `public()`-Methode der Klasse `Automat` sähe wie folgt aus:

```
00  // listings/011/automat2/moveCtor/automat.h
...
01    public:
...
02      // Kopieren verbieten
03      Automat( const Automat & ) = delete;
04      // Move-Konstruktor
05      Automat( Automat && )
...
```

Bei manchen Beispielen ist es sinnvoll, das Kopieren von Objekten komplett zu verbieten. Dies können Sie (optional) realisieren, indem Sie, wie hier in Zeile (03) geschehen, dem Kopierkonstruktor `delete` zuweisen. Diese Möglichkeit kann seit C++11 verwendet werden. In Zeile (05) sehen Sie die Deklaration des Verschiebekonstruktors.

Jetzt noch die Definition des Verschiebekonstruktors in *automat.cpp*:

```
00  // listings/011/automat2/moveCtor/automat.cpp
...
01  Automat::Automat(Automat &&a)
02  : geld{a.geld}, standort{a.standort} {
03    a.geld=0L;
04    a.standort="";
05  }
...
```

Die Daten für das neue Objekt übergeben wir direkt per Elementinitialisierer (Zeile (02)). Die ursprünglichen Daten »räumen wir auf« oder löschen sie, wie hier in Zeile (03) und (04). Jetzt haben Sie eine einfache Klasse erstellt, deren Instanzen Sie nicht mehr kopieren, sondern nur noch verschieben können.

Verschieben und Kopieren

Wenn Sie zusätzlich kopieren wollen, müssen Sie den Kopierkonstruktor ebenfalls noch selbst definieren.

Um den Inhalt der Objekte zu verschieben, packen Sie das Objekt mit dem zu verschiebenden Inhalt in die Standardfunktion `std::move()`:

```
00  // listings/011/automat2/moveCtor/main.cpp
...
01  Automat device01{10000, "Fulda, Hauptstrasse 1"};
02  Automat device02{std::move(device01)};
03  Automat device03 = std::move(device02);
...
```

In Zeile (**01**) wird ein neues Objekt mit dem Bezeichner `device01` erstellt, dessen Daten in Zeile (**02**) nach `device02` verschoben werden. `device01` ist jetzt »leer«. Das Gleiche machen wir in Zeile (**03**), in der wir den Inhalt von `device02` nach `device03` verschieben, womit `device02` ebenfalls ohne Inhalt ist.

Standard-Verschiebekonstruktor

In diesem Kontext ist es wichtig zu wissen, dass der Compiler einen Standard-Verschiebekonstruktor generiert. Somit hätten Sie in diesem Beispiel auf die Implementierung des Verschiebekonstruktors verzichten können. Aber auch hier gilt, dass Sie beispielsweise bei kritischen Daten wie zum Beispiel dynamischen Elementen (flachen Kopien) einer Klasse diese Spezialfunktionen selbst implementieren müssen.

11.3 Destruktoren

Das Gegenstück zum Konstruktor, der für das Initialisieren von Objekten zuständig ist, wird *Destruktor* (im Englischen kurz als *dtor* bezeichnet)

genannt. Der Destruktor wird genauso wie der Konstruktor automatisch aufgerufen – allerdings mit dem Unterschied, dass der Destruktor nicht bei der Definition aufgerufen wird, sondern beim »Zerstören« des Objekts. Der Destruktor wird gewöhnlich für Aufräumarbeiten wie beispielsweise das Freigeben von dynamisch reserviertem Speicherplatz, das Aufheben von Sperren, Dateifreigaben und andere Rückgaben von bestimmten Ressourcen verwendet. Man spricht vom *Abbauen* eines Objekts.

11.3.1 Lebensdauer eines Objekts

Die Lebensdauer einer Klasseninstanz entspricht in etwa der von gewöhnlichen Variablen. Sie gilt ab dem Moment, in dem sie definiert wird. Dabei wird auch direkt der Konstruktor aufgerufen. Diese Instanz können Sie so lange verwenden, bis die Programmausführung deren Gültigkeitsbereich verlässt, was meistens im definierten Bereich bzw. Anweisungsblock der Instanz die schließende geschweifte Klammer ist. An dieser Stelle wird in der Regel der Destruktor aufgerufen, um die Instanz aus dem Speicher zu entfernen. Wenn somit der Gültigkeitsbereich der Klasseninstanz verlassen wurde, kann diese nicht mehr verwendet werden.

11.3.2 Wann wird ein Destruktor erforderlich?

Bevor Sie erfahren, wie Sie einen eigenen Destruktor erstellen können, möchte ich hier noch kurz erklären, wann überhaupt ein Destruktor nötig wird. In dem Beispiel, wie Sie es mit der Klasse `Automat` bisher verwendet haben, ist ein Destruktor nicht nötig, weil hier Datenfelder der Klasse automatisch aufgeräumt werden, wenn die Klasseninstanz den Gültigkeitsbereich verlässt. Wenn eine Klasse einfache Datentypen wie `int`, `double` usw. enthält, werden die Daten automatisch aufgeräumt. Das Gleiche gilt auch für die Typen der Standardbibliothek wie `std::string`, `std::vector` usw., für die kein gesonderter Destruktor erforderlich ist.

Ein spezieller Destruktor wird erst nötig, wenn Sie rohe Zeiger oder andere C-Datentypen als Ressource in den Datenfeldern verwenden. Solche Ressourcen werden nicht automatisch entfernt.

RAII (Ressourcenbelegung ist Initialisierung)

Resource Acquisition Is Initialization, meistens kurz *RAII* genannt, ist eine Programmiertechnik zur Verwaltung von Ressourcen, die gewöhnlich mit einem Konstruktor angefordert werden und durch die Freigabe an den Destruktor gebunden sind. Sie schreiben damit praktisch den Destruktor zum Konstruktor. Wie bereits erwähnt, ist ein spezieller Konstruktor erst erforderlich, wenn eine Ressource wie beispielsweise ein roher Zeiger verwendet wird, da solche Ressourcen nicht automatisch freigegeben werden. Der Grund dafür, dass Sie bei der Verwendung von Typen der Standardbibliothek wie `std::string` oder `std::vector` als Datenfeld das Konzept nicht beachten müssen, liegt darin, dass diese Typen das RAII-Konzept bereits umgesetzt haben – sprich: Hier wurde alles bereits sorgfältig in den Konstruktoren und Destruktoren der Klassen verpackt.

11.3.3 Destruktor deklarieren

Der Destruktor besteht wie der Konstruktor lediglich aus dem Klassennamen – allerdings mit einem vorangestellten ~ (Komplement-Zeichen), beispielsweise:

```
// Deklaration eines Destruktors
~Klassenname( );
```

Ebenso gilt für den Destruktor, dass er keinen Rückgabewert enthält, und zusätzlich (im Gegensatz zum Konstruktor) besitzt er niemals einen Parameter. Der Destruktor steht üblicherweise im `public`-Bereich einer Klasse.

In der Klasse `Automat` deklarieren wir den Destruktor im `public`-Bereich. Hierzu die Headerdatei *automat.h*:

```
// listings/011/automat3/automat.h
...
class Automat {
  private:
    // Daten der Klasse "Automat"
    unsigned long geld{0L};
    std::string standort{""};
  public:
    // Deklaration der Konstruktoren
    Automat():Automat(0,""){};
    Automat(unsigned long g):Automat(g,""){};
    Automat(std::string s):Automat(0L,s){};
    Automat(std::string s, unsigned long g):Automat(g,s){};
    Automat(unsigned long, std::string);
    // Deklaration des Destruktors
    ~Automat();
    // Methoden der Klasse "Automat"
...
};
```


11.3.4 Destruktor definieren

Sofern Sie keinen Destruktor deklarieren und definieren, erzeugt der Compiler auch hierbei eine Standardversion davon. Ein Destruktor zerstört die Eigenschaften eines Objekts in umgekehrter Reihenfolge der Erzeugung, was der Reihenfolge der Datenelemente in der Klassendefinition entspricht. Dies gilt auch, wenn das Attribut selbst ein Objekt ist – nur wird dann dessen Destruktor (implizit oder falls vorhanden explizit) verwendet.

Wenn Sie hingegen einen Destruktor selbst explizit definieren, gehen Sie ähnlich wie beim Konstruktor vor. Die Syntax lautet:

```
// Definition eines Destruktors
Klassenname::~Klassenname( ) {
   // Anweisungen
}
```

Bezogen auf unsere Klasse `Automat` kann dieser Destruktor daher wie folgt aussehen:

```
// listings/011/automat3/automat.cpp
...
~Automat::Automat() {
  std::cout << get_standort() << " entfernt\n";
}
...
```

Der Destruktor dient in der Praxis – anders als im Beispiel – nicht dazu, einen Text auf dem Bildschirm auszugeben, sondern er kümmert sich um diverse Aufräumarbeiten. Ganz besonders darum, um Ressourcen, die im Konstruktor angefordert wurden, im Destruktor wieder freizugeben (Stichwort: *RAII*). Im hier vorliegenden Beispiel war allerdings ohnehin kein Destruktor erforderlich. Die Textausgabe in diesem Beispiel diente nur zur Demonstration, damit Sie im anschließenden Beispiel besser verstehen, wann der Destruktor aufgerufen wird.

Hierzu daher noch das Beispiel, das Ihnen den Destruktor in der Praxis zeigt:

```
00  // listings/011/automat3/main.cpp
01  #include <iostream>
02  #include <vector>
03  #include <string>
04  #include <memory>  // unique_ptr
05  #include "automat.h"

06  Automat device01{"Global: Bonn"};

07  int main(void) {
08    Automat* device02 = new Automat{"main(new): Mering"};
09    static Automat device03{"main(static): Erfurt"};
10    delete device02;
11    device02=nullptr;
12    Automat device04{"main(): Berlin"};
13    {
14      Automat device05{"main(lokal): Kiel"};
15      std::unique_ptr<Automat> device_ptr{ new Automat{
```



```
                "main(lokal): Augsburg"}  };
16    }
17    return 0;
18  }
```

Das Programm bei der Ausführung:

```
main(new): Mering entfernt
main(lokal): Augsburg entfernt
main(lokal): Kiel entfernt
main(): Berlin entfernt
main(static): Erfurt entfernt
Global: Bonn entfernt
```

An der Ausgabe des Beispiels erkennen Sie, dass der Destruktor explizit aufgerufen wird, wenn sich der Gültigkeitsbereich eines Objekts beendet. Somit gilt für den Gültigkeitsbereich von Objekten dasselbe wie bei den Basisdatentypen:

- Ist ein Objekt lokal deklariert und gehört es nicht zur Speicherklasse `static`, wird es am Ende des entsprechenden Anweisungsblocks zerstört (im Beispiel die Objekte in Zeile **(12)** und **(14)**). Dasselbe gilt für den in C++11 eingeführten klugen Zeiger `unique_ptr` (Zeile (**15**)), der den Speicherbereich am Ende seines Gültigkeitsbereichs (hier Zeile (**16**)) wieder freigibt.
- Ist ein Objekt global oder `static` deklariert, wird es am Ende des Programms zerstört (im Beispiel die Objekte in Zeile **(06)** und **(09)**).
- Dynamische Objekte, die Sie mit dem klassischen `new` reservieren, werden an Ort und Stelle gelöscht, wenn der Speicher mit `delete` explizit freigegeben wird (wie im Beispiel in Zeile **(08)** und **(10)** geschehen).

Achtung bei Ausnahme im Konstruktor

Wird eine Ausnahme im Konstruktor ausgelöst und wird diese nicht abgefangen, erfolgt kein Aufruf des Destruktors, weil das Objekt als nicht erzeugt gilt. Ausnahmen werden noch gesondert in Kapitel 16 behandelt.

11.4 Methoden

In diesem Abschnitt gehe ich auf typische Themen zu den Methoden (oft auch als *Elementfunktionen* oder *Member-Funktionen* bezeichnet) ein – deshalb sollten Sie die Abschnitte zu den Klassen bereits verinnerlicht haben.

11.4.1 »inline«-Methoden

Wie Sie bereits bei den Funktionen erfahren haben, stellt der Aufruf einer solchen einen nicht unerheblichen Aufwand dar (Stichwort: *Stack[-Frame]*). Hierbei wird die Rücksprungadresse auf den Stack gelegt, und die Funktion wird angesprungen. Ebenso werden lokale Parameter angelegt. Am Ende der Funktion müssen lokale Daten wieder freigegeben werden, und das Programm springt zurück zum Ausgangspunkt. Um diesen Aufwand zu vermeiden, steht Ihnen das Schlüsselwort `inline` als Spezifizierer zur Verfügung. Mit diesem Schlüsselwort verlangen Sie vom Compiler, dass der Block der Funktion an die Stelle des Funktionsaufrufs kopiert. Dadurch entfällt der gerade erwähnte Aufwand, und das Programm kann so um einiges schneller laufen.

Das Einkopieren spart zwar manchen Funktionsaufruf, kann aber wiederum den Code unnötig aufblähen, weil hierbei jedes Mal der komplette Rumpf kopiert wird. Damit könnte der Pufferspeicher des Prozessors unnötig gefüllt werden, und das Programm läuft trotz der geringeren Aufrufkosten langsamer. Daher wird empfohlen, diesen `inline`-Spezifizierer nur für kleine Funktionen zu verwenden.

Ähnlich ist dies bei den Methoden – die im Grunde auch nur Funktionen (für Klassen) sind. Auch bei den Methoden steht Ihnen die Möglichkeit zur Verfügung, sie als `inline` zu definieren.

Kluger Compiler

Beachten Sie hier bitte, dass moderne Compiler ohne explizite oder implizite Verwendung von `inline` die Methoden automatisch als *inline* deklarieren können. Letztendlich entscheidet nämlich der Compiler selbst über eine tatsächliche `inline`-Verwendung.

Zur Verwendung von `inline` stehen Ihnen bei den Klassen zwei Möglichkeiten zur Verfügung: explizit und implizit.

Implizit »inline«

Definieren Sie kleinere Methoden gleich innerhalb der Klassendefinitionen, werden diese implizit zu `inline`-Methoden – auch wenn Sie das Schlüsselwort `inline` nicht mit angeben. Sie können diese Methoden trotzdem, zur besseren Lesbarkeit, extra mit `inline` definieren. Nehmen wir als Beispiel die Klasse `Automat`:

```
00 // listings/011/automat4/automat.h
01 #ifndef _AUTOMAT_H_
02 #define _AUTOMAT_H_
03 #include <string>

04 class Automat {
05   private:
06     // Daten der Klasse "Automat"
07     unsigned long geld{0L};
08     std::string standort{""};
09   public:
10     // Deklaration der Konstruktoren
11     Automat():Automat(0,""){};
12     Automat(unsigned long g):Automat(g,""){};
13     Automat(std::string s):Automat(0L,s){};
14     Automat(std::string s,unsigned long g):Automat(g,s){};
15     Automat(unsigned long, std::string);
16     // Methoden der Klasse "Automat"
17     unsigned long get_geld() {
18       return geld;
19     }
20     void set_geld(unsigned long g){
21       geld = g;
22     }
23     const std::string& get_standort() {
24       return standort;
25     }
```

```
26      void set_standort(const std::string& s){
27        standort = s;
28      }
29      void print() {
30        std::cout << '\n' << "Geldautomat : ";
31        std::cout << get_standort() << std::endl;
32        std::cout << "Geld (Euro) : ";
33        std::cout << get_geld() << std::endl;
34      }
35  };
36  #endif
```

Da die Methoden in der Klassendefinition festgelegt wurden, gelten `get_geld()`, `set_geld()`, `get_standort()`, `set_standort()` und `init()` aus Zeile **(17)** bis **(34)** implizit als `inline`.

Dies bedeutet, dass Sie die Definitionen der Quelldatei *automat.cpp* entfernen müssen.

Explizit »inline«

Methoden können Sie auch explizit als `inline` definieren, was durchaus die gängigere Methode darstellt. Sie setzen hier bei der Definition lediglich das Schlüsselwort `inline` vor die Klassenmethode.

Auf unser Beispiel der Klasse `Automat` bezogen müssten Sie hierzu in der Datei *automat.cpp* vor die entsprechende Methode das Schlüsselwort `inline` schreiben (natürlich setzt dies voraus, dass Sie in der Headerdatei *automat.h* in der Klassendefinition noch keine Methode, wie eben im Abschnitt »Implizit »inline«‹‹« gesehen, definiert haben). Hier das Listing dazu (gekürzte Fassung):

```
// listings/011/automat5/automat.cpp
#include <iostream>
#include <string>
#include "automat.h"
...
inline unsigned long Automat::get_geld() {
  return geld;
}
```



```
inline void Automat::set_geld(unsigned long g){
  geld = g;
}

inline const std::string& Automat::get_standort() {
  return standort;
}

inline void Automat::set_standort(const std::string& s){
  standort = s;
}
...
```

Wann »inline« verwenden?

Da letztendlich ohnehin der Compiler entscheidet, ob er Ihren `inline`-Vorschlag annimmt, stellt sich die Frage, wann denn solche `inline`-Methoden bzw. -Funktionen überhaupt sinnvoll sind. In der Praxis hängt das zunächst vom Anwendungsfall ab. Aber wenn Ihr Programm mehr Performance benötigt, könnten Sie es mit `inline` probieren. Wie bereits erwähnt, kann damit allerdings auch das Programm unnötig größer gemacht und dadurch wiederum ausgebremst werden. Zudem ist es nicht leicht, zu erkennen, ob ein Programm mit `inline` überhaupt schneller läuft. Daher werden Sie hier um weitere Tests Ihres Programms nicht herumkommen.

11.4.2 Konstante Methoden (»nur-lesen-erlaubt«)

Methoden, die nur lesend auf die Daten zugreifen, werden in der Praxis speziell gekennzeichnet. Damit ist es möglich, sie mit `const`-Objekten aufzurufen. Eine Methode deklarieren und definieren Sie als *nur-lesen-erlaubt*, indem Sie an den Funktionskopf das Schlüsselwort `const` anhängen:

```
// Deklaration in der Klasse
typ methode( parameter ) const;
```

```
...

// Definition der Methode
typ methode( parameter ) const {
   // Anweisungen
}
```

In unserer Klasse `Automat` sind beispielsweise die `get()`-Methoden geeignet, eine Methode als *nur-lesen-erlaubt* zu deklarieren und zu definieren. Hierzu müssen Sie im Beispiel nur an den entsprechenden Stellen das Schlüsselwort `const` anhängen. In der Headerdatei *automat.h* wären dies die folgenden zwei Methoden (gekürzte Fassung):

```
// listings/011/automat6/automat.h
...
class Automat {
...
    // Methoden der Klasse "Automat"
    unsigned long get_geld() const;
    void set_geld(unsigned long g);
    const std::string& get_standort() const;
    void set_standort(const std::string& s);
    void print();
};
```

Neben der Deklaration in der Klassendefinition müssen Sie auch die Definitionen der Methoden mit dem Schlüsselwort `const` signieren. Hierzu passen Sie im Quellcode *automat.cpp* ebenfalls nur die entsprechenden zwei Methoden `Automat::get_geld()` und `Automat::get_standort()` an (gleichfalls gekürzt):

```
// listings/011/automat6/automat.cpp
...
unsigned long Automat::get_geld() const {
  return geld;
}
...
```



```
const std::string& Automat::get_standort() const {
  return standort;
}
...
```

Der Vorteil solcher konstanten Methoden besteht darin, dass aus diesen Methoden keine anderen Funktionen aufgerufen werden können, die die Daten eines Objekts womöglich versehentlich überschreiben. Beispielsweise würde der Compiler folgenden Codeabschnitt bemängeln und das Programm nicht übersetzen:

```
unsigned long Automat::get_geld() const {
  // Nicht erlaubt, da als read-only mit const signiert
  set_geld(0);   // !!! Fehler !!!
  return geld;
}
```


11.4.3 »this«-Zeiger

Sicherlich haben Sie sich bereits gefragt, wie es möglich sein kann, mit den Methoden auf die Daten eines bestimmten Objekts zuzugreifen, obwohl niemals eine explizite Angabe zum entsprechenden Objekt gemacht wurde. Als Beispiel verwenden wir die Klasse `Automat`. Nehmen wir an, Sie wollen folgendes Objekt ausgeben:

```
Automat device{123456, "Mering, Bahnhofweg 1"};
...
std::cout << device.get_standort() << std::endl;
```

Hierbei wird die Methode `get_standort()` aufgerufen:

```
std::string Automat::get_standort() const {
  return standort;
}
```

Bei dieser Methode ist weit und breit keine Angabe in Sicht, mit welchem Objekt diese Methode eigentlich arbeitet. Die Antwort lautet, dass beim

Aufruf als nicht sichtbares Argument die Adresse des aktuellen Objekts mit übergeben wird. Diese Adresse steht in der Methode mit dem konstanten Zeiger `this` zur Verfügung. Die Syntax der Deklaration dieses `this`-Zeigers sieht intern folgendermaßen aus:

```
Klassenname* const this = &Objekt;
```

Auf das Beispiel `Automat device` bezogen stellt sich dies folgendermaßen dar:

```
Automat* const this = &device;
```

Somit ist das Objekt `device` das Objekt, mit dem die Methode aufgerufen wird.

Dieser implizite Parameter `this` steht somit für die Speicheradresse der Instanz, mit der Sie die Methode aufrufen. Verwenden Sie somit `this` innerhalb von Klassenmethoden, erfolgt der Zugriff auf die einzelnen Mitglieder einer Klasse mit folgendem Ausdruck:

```
this->Element
```

In der Tat ist das genau das, was der Compiler implizit daraus machen würde, wenn Sie nur den folgenden Ausdruck verwenden würden:

```
Element
```

Somit entspricht – bezogen auf die Klasse `Automat` und die Methode `get_standort()` – die Definition:

```
std::string Automat::get_standort() const {
  return standort;
}
```

der folgenden Definition der Methode:

```
string Automat::get_standort() const {
  return this->standort;
}
```


In der Praxis kann dieser explizite `this`-Zeiger verwendet werden, um bei Bedarf die Bezeichner von lokalen Variablen einer Methode von den Daten (Klassenvariablen) mit gleichem Namen zu unterscheiden (wenn ein gleicher Name verwendet wurde). Allerdings wird in der Praxis der `this`-Zeiger als Ganzes benutzt, wenn zum Beispiel aus einer Methode ein Objekt als Kopie oder Referenz zurückgegeben werden soll (`return *this`). Darauf gehe ich in Abschnitt 12.2.3, »Das Zielobjekt mit dem ›this‹-Zeiger«, noch näher ein.

11.5 Kontrollfragen und Aufgaben

1. Mit welchen Schlüsselwörtern erstellen Sie in C++ eine Klasse, und worin unterscheiden sie sich?
2. Wie erfolgt der Zugriff auf die Daten in einer Klasse?
3. Was ist ein Konstruktor?
4. Was ist ein Destruktor?
5. Erklären Sie kurz, was der `this`-Zeiger ist.
6. Die folgende Headerdatei enthält einen Designfehler. Welchen?

```
// listings/006/aufgabe001.h
#ifndef _TEST_H_
#define _TEST_H_

class XYZ {
  public:
    int ival;
    XYZ(int i) : ival{i} { }
    XYZ() : ival{0} { }
    int get_ival() const { return ival; };
    void set_ival(int i) { ival = i; }
};
#endif
```

7. Die Klassendefinition der Headerdatei *aufgabe002.h* und die Definition der Methode des Quellcodes *aufgabe002.cpp* lassen sich nicht übersetzen. Wo sind die Fehler? Zunächst die Headerdatei *aufgabe002.h*:

```
00 // listings/006/aufgabe002.h
01 #ifndef _TEST_H_
02 #define _TEST_H_

03 class XYZ {
04   private:
05     int ival;
06   public:
07     XYZ(int i) : ival{i} { }
08     XYZ() : ival{0} { }
09     int get_ival() const { return ival; };
10     void set_ival(int i) { ival = i; }
11     void manipVal(int i=0) const;
12 };
13 #endif
```

Und hier der Quellcode *aufgabe002.cpp*:

```
00 // listings/011/aufgaben/aufgabe002.cpp
01 #include "aufgabe002.h"

02 void manipVal(int i) const {
03   set_ival(i);
04 }
```

B.10 Lösungen zu Kapitel 11

1. Eine Klasse können Sie neben dem üblichen Schlüsselwort `class` auch mit dem Schlüsselwort `struct` erstellen. Beim Schlüsselwort `struct` sind alle Mitglieder der Klasse von Haus aus von außen öffentlich zugänglich. Beim Schlüsselwort `class` hingegen sind alle Elemente einer Klasse zunächst privat und nicht von außen erreichbar.
2. Der Zugriff auf Daten einer Klasse, die gewöhnlich `private` sind – also außerhalb der Klasse nicht erreichbar –, erfolgt über Methoden der Klasse. Damit allerdings die Methoden einer Klasse verwendet werden können, müssen sie von außen zugänglich sein. Mit dem Schlüsselwort `public` sorgen Sie dafür, dass die Methoden außerhalb der Klasse verwendbar sind.
3. Der Konstruktor ist eine Funktion, die ausschließlich dem Zweck dient, die Daten eines Objekts mit gültigen Werten zu initialisieren, um nicht mit Objekten zu arbeiten, deren Werte undefiniert sind. Ein solcher Konstruktor wird ausgeführt, wenn das Objekt erzeugt wird, und zeichnet sich dadurch aus, dass er denselben Namen hat wie die dazugehörende Klasse (`X::X()`).
4. Der Destruktor ist das Gegenstück zum Konstruktor und wird in der Regel automatisch aufgerufen, wenn ein Objekt zerstört oder freigegeben werden soll. Einen Destruktor können Sie auch selbst definieren, wenn Sie beim Zerstören von Objekten bestimmte Ressourcen (Freispeicher, Dateien, Sperren usw.) freigeben müssen. Die Deklaration und die Definition eines Destruktors entsprechen denen des Konstruktors, mit dem Unterschied, dass Sie hier noch das Komplement-Zeichen voranstellen (`~X::X()`).
5. Der `this`-Zeiger ist ein Zeiger auf das Objekt, für das die Methode aufgerufen wurde, quasi ein Zeiger auf die eigene Klasse.
6. Hier wurden die Daten (in dem Fall zwar nur der Wert `ival`) mit `public` signiert und sind somit öffentlich zugänglich, was man beim Klassendesign mit der Datenkapselung unbedingt vermeiden will. In der Praxis sollten Sie die Daten mit `private` kennzeichnen. Die folgende Klassendefinition behebt diesen Designfehler:


```
// listings/006/loesung001.h
#ifndef _TEST_H_
#define _TEST_H_

class XYZ {
  private:
    int ival;
  public:
    XYZ(int i) : ival{i} { }
    XYZ() : ival{0} { }
    int get_ival() const { return ival; };
    void set_ival(int i) { ival = i; }
};
#endif
```

7. In der Klassendefinition der Headerdatei *aufgabe002.h* muss in Zeile **(12)** das Schlüsselwort `const` entfernt werden, ebenso wie in Zeile **(02)** beim Quellcode *aufgabe002.cpp*, weil sonst kein Zugriff über die Methode in Zeile **(03)** möglich ist. Des Weiteren fehlen bei der Methodendefinition in Zeile **(02)** der Klassenname und der Bereichsoperator (`XYZ::`), sodass es sich nur um eine gewöhnliche Funktionsdefinition handelt. Hier der überarbeitete Code der Quelldatei *aufgabe002.cpp*:

```
// listings/011/loesungen/loesung002.cpp
#include "loesung002.h"

// Klassenname und Bereichsoperator hinzugefügt
void XYZ::initVal(int i)  {   // const entfernt
  set_ival(i);
}
```

B.11 Lösungen zu Kapitel 12

1. Der `this`-Zeiger kann innerhalb einer (nicht statischen) Methode verwendet werden und stellt einen Zeiger auf das Objekt dar, für das die Methode aufgerufen wurde. Der Zeiger wird intern immer mit an eine