1

Es geht los!

Dieses Kapitel behandelt die folgenden Themen:

- Entstehung und Entwicklung der Programmiersprache C++
- Objektorientierte Programmierung – erste Grundlagen
- Wie schreibe ich ein Programm und bringe es zum Laufen?
- Einfache Datentypen und Operationen
- Ablauf innerhalb eines Programms steuern
- Erste Definition eigener Datentypen
- Standarddatentypen vector und string
- Einfache Ein- und Ausgabe

1.1 Historisches

C++ wurde etwa ab 1980 von Bjarne Stroustrup als die Programmiersprache »C with classes« (englisch *C mit Klassen*), die Objektorientierung stark unterstützt, auf der Basis der Programmiersprache C entwickelt. Später wurde die neue Sprache in C++ umbenannt. ++ ist ein Operator der Programmiersprache C, der den Wert einer Größe um 1 erhöht. Insofern spiegelt der Name die Eigenschaft »Nachfolger von C«. 1998 wurde C++ erstmals von der ISO (International Organization for Standardization) und der IEC (International Electrotechnical Commission) standardisiert. Diesem Standard haben sich nationale Standardisierungsgremien wie ANSI (USA) und DIN (Deutschland) angeschlossen. Die Anforderungen an C++ sind gewachsen, auch zeigte sich, dass manches fehlte und anderes überflüssig oder fehlerhaft war. Das C++-Standardkomitee hat kontinuierlich an der Verbesserung von C++ gearbeitet, sodass in Abständen neue Versionen

des Standards herausgegeben wurden. Die Kurznamen sind entsprechend den Jahreszahlen C++03, C++11, C++14 und C++17. C++17 wurde im Juli 2017 von der zuständigen ISO/IEC-Arbeitsgruppe JTC1/SC22/WG21 verabschiedet und bei der ISO zur Veröffentlichung eingereicht.

1.2 Objektorientierte Programmierung

Nach üblicher Auffassung heißt Programmieren, einem Rechner mitzuteilen, *was* er tun soll und *wie* es zu tun ist. Ein Programm ist ein in einer Programmiersprache formulierter Algorithmus oder anders ausgedrückt eine Folge von Anweisungen, die der Reihe nach auszuführen sind, ähnlich einem Kochrezept, geschrieben in einer besonderen Sprache, die der Rechner »versteht«. Der Schwerpunkt dieser Betrachtungsweise liegt auf den einzelnen Schritten oder Anweisungen an den Rechner, die zur Lösung einer Aufgabe abzuarbeiten sind.

Was fehlt hier beziehungsweise wird bei dieser Sicht eher stiefmütterlich behandelt? Der Rechner muss »wissen«, *womit* er etwas tun soll. Zum Beispiel soll er

- eine bestimmte Summe Geld von einem Konto auf ein anderes transferieren;
- eine Ampelanlage steuern;
- ein Rechteck auf dem Bildschirm zeichnen.

Häufig, wie in den ersten beiden Fällen, werden Objekte der realen Welt (Konten, Ampelanlage ...) *simuliert*, das heißt im Rechner abgebildet. Die abgebildeten Objekte haben eine *Identität*. Das *Was* und das *Womit* gehören stets zusammen. Beide sind also Eigenschaften eines Objekts und sollen daher nicht getrennt werden. Ein Konto kann schließlich nicht auf Gelb geschaltet werden, und eine Überweisung an eine Ampel ist nicht vorstellbar.

Ein *objektorientiertes Programm* kann man sich als Abbildung von Objekten der realen Welt in Software vorstellen. Die Abbildungen werden selbst wieder Objekte genannt. Klassen sind Beschreibungen von Objekten. Die *objektorientierte Programmierung* berücksichtigt besonders die Kapselung von Daten und den darauf ausführbaren Funktionen sowie die Wiederverwendbarkeit von Software und die Übertragung von Eigenschaften von Klassen auf andere Klassen, Vererbung genannt. Auf die einzelnen Begriffe wird noch eingegangen. Das Motiv hinter der objektorientierten Programmierung ist die rationelle und vor allem ingenieurmäßige Softwareentwicklung.

Wiederverwendung heißt, Zeit und Geld zu sparen, indem bekannte Klassen wiederverwendet werden. Das Leitprinzip ist hier, das Rad nicht mehrfach neu zu erfinden! Unter anderem durch den Vererbungsmechanismus kann man Eigenschaften von bekannten Objekten ausnutzen. Zum Beispiel sei `Konto` eine bekannte Objektbeschreibung mit den »Eigenschaften« Inhaber, Kontonummer, Betrag, Dispo-Zinssatz und so weiter. In einem Programm für eine Bank kann nun eine Klasse `Waehrungskonto` entworfen werden, für die alle Eigenschaften von `Konto` übernommen (= geerbt) werden könnten. Zusätzlich wäre nur noch die Eigenschaft »Währung« hinzuzufügen.

Wie Computer können Objekte Anweisungen ausführen. Wir müssen ihnen nur »erzählen«, was sie tun sollen, indem wir ihnen eine *Aufforderung* oder *Anweisung* senden, die in einem Programm formuliert wird. Anstelle der Begriffe »Aufforderung« oder »Anweisung« wird manchmal *Botschaft* (englisch *message*) verwendet, was jedoch den Aufforderungscharakter nicht zur Geltung bringt. Eine gängige Notation (= Schreibweise) für solche Aufforderungen ist *Objektname.Anweisung(*gegebenenfalls *Daten)*. Beispiele:

```
dieAmpel.blinken(gelb);
dieAmpel.ausschalten();                                  // keine Daten notwendig!
dieAmpel.einschalten(gruen);
dasRechteck.zeichnen(position, hoehe, breite);           // Daten in cm
dasRechteck.verschieben(5.0);                            // Daten in cm
```

Die Beispiele geben schon einen Hinweis, dass die Objektorientierung uns ein Hilfsmittel zur Modellierung der realen Welt in die Hand gibt.

Klassen

Es wird zwischen der *Beschreibung* von Objekten und den *Objekten selbst* unterschieden. Die Beschreibung besteht aus Attributen und Operationen. Attribute bestehen aus einem Namen und Angaben zum Datenformat der Attributwerte. Eine Kontobeschreibung könnte so aussehen:

Attribute:

Inhaber: Folge von Buchstaben
Kontonummer: Zahl
Betrag: Zahl
Dispo-Zinssatz in %: Zahl

Operationen:

überweisen(Ziel-Kontonummer, Betrag)
abheben(Betrag)
einzahlen(Betrag)

Eine Aufforderung ist nichts anderes als der Aufruf einer Operation, die auch Methode genannt wird. Ein *tatsächliches* Konto k1 enthält *konkrete* Daten, also Attributwerte, deren Format mit dem der Beschreibung übereinstimmen muss. Die Tabelle zeigt k1 und ein weiteres Konto k2. Beide Konten haben *dieselben* Attribute, aber *verschiedene* Werte für die Attribute.

Tabelle 1.1: Attribute und Werte zweier Konten

Attribut	Wert für Konto k1	Wert für Konto k2
Inhaber	Roberts, Julia	Depp, Johnny
Kontonummer	12573001	54688490
Betrag	-200,30 €	1222,88 €
Dispo-Zinssatz	13,75 %	13,75 %

Julia will Johnny 1000 € überweisen. Bei einer Überweisung wird die IBAN (internationale Bankkontonummer) angegeben. Dem Objekt k1 wird also der Auftrag mit den benötigten Daten mitgeteilt: *k1.überweisen("DE25123456000054688490", 1000.00)*.

»DE25123456000054688490« ist die (hier fiktive) IBAN. Die letzten 10 Stellen der IBAN enthalten die Kontonummer, die von der empfangenden Bank extrahiert wird, um den Empfänger zu ermitteln. Anderes Beispiel: Johnny will 22 € abheben. Die Aufforderung wird an k2 gesendet: *k2.abheben(22)*.

Es scheint natürlich etwas merkwürdig, wenn einem Konto ein Auftrag gegeben wird. In der objektorientierten Programmierung werden Objekte als Handelnde aufgefasst, die auf Anforderung selbstständig einen Auftrag ausführen, entfernt vergleichbar einem Sachbearbeiter in einer Firma, der seine eigenen Daten verwaltet und mit anderen Sachbearbeitern kommuniziert, um eine Aufgabe zu lösen.

- Die *Beschreibung* eines tatsächlichen Objekts gibt seine innere *Datenstruktur* und die möglichen *Operationen* oder *Methoden* an, die auf die inneren Daten anwendbar sind.
- Zu *einer* Beschreibung kann es kein, ein oder beliebig viele Objekte geben.

Die Beschreibung eines Objekts in der objektorientierten Programmierung heißt *Klasse*. Die tatsächlichen Objekte heißen auch *Instanzen* einer Klasse.

Auf die inneren Daten eines Objekts nur mithilfe der vordefinierten Methoden zuzugreifen, dient der Sicherheit der Daten und ist ein allgemein anerkanntes Prinzip. Das Prinzip wird *Datenabstraktion* oder Geheimnisprinzip genannt. Der Softwareentwickler, der die Methoden konstruiert hat, weiß ja, wie die Daten konsistent (das heißt widerspruchsfrei) bleiben und welche Aktivitäten mit Datenänderungen verbunden sein müssen. Zum Beispiel muss eine Erhöhung des Kontostands mit einer Gutschrift oder einer Einzahlung verbunden sein. Außerdem wird jeder Buchungsvorgang protokolliert. Es darf nicht möglich sein, dass jemand anderes die Methoden umgeht und direkt und ohne Protokoll seinen eigenen Kontostand erhöht. Wenn Sie die Unterlagen eines Kollegen haben möchten, greifen Sie auch nicht einfach in seinen Schreibtisch, sondern Sie bitten ihn darum, dass er sie Ihnen gibt.

Die hier verwendete Definition einer Klasse als Beschreibung der Eigenschaften einer Menge von Objekten wird im Folgenden beibehalten. Gelegentlich findet man in der Literatur andere Definitionen, auf die hier nicht weiter eingegangen wird. Weitere Informationen zur Objektorientierung sind in Kapitel 3 und in der einschlägigen Literatur zu finden.

1.3 Werkzeuge zum Programmieren

Um Programme schreiben und ausführen zu können, brauchen Sie nicht viel: einen Computer mit einem Editor und einem C++-Compiler.

Der Editor

Ein Editor ist ein Programm, mit dem man Texte schreiben kann. Dabei darf er keine versteckten Sonderzeichen enthalten, weswegen LibreOffice oder Word nicht geeignet sind. Ein für Windows sehr gut geeigneter Texteditor ist *Notepad++*[1]. *Kwrite* läuft unter Linux

[1] *https://notepad-plus-plus.org/*

und Mac OS stellt *TextEdit* zur Verfügung, besser geeignet ist jedoch der Editor von *Xcode*, das Sie dazu jedoch installieren müssen. Um für den Anfang die Einarbeitung in eine Integrierte Entwicklungsumgebung zu sparen, können Sie stattdessen einen einfachen Texteditor benutzen und das Programm in der Konsole (=MS-DOS-Eingabeaufforderung oder PowerShell-Fenster, Linux/MacOS-Terminal)[2] mit den weiter unten gezeigten Kommandos compilieren.

Der Compiler

Compiler sind die Programme, die Ihren Programmtext in eine für den Computer verarbeitbare Form übersetzen. Von Menschen geschriebener und für Menschen lesbarer Programmtext kann nicht vom Computer »verstanden« werden. Das vom Compiler erzeugte Ergebnis der Übersetzung kann der Computer aber ausführen. Das Erlernen einer Programmiersprache ohne eigenes praktisches Ausprobieren ist kaum sinnvoll. Nutzen Sie daher die Dienste des Compilers möglichst bald anhand der Beispiele – wie, zeigt Ihnen der Abschnitt direkt nach der Vorstellung des ersten Programms. Falls Sie nicht schon einen C++-Compiler oder ein C++-Entwicklungssystem haben, um die Beispiele korrekt zu übersetzen, bietet sich als Alternative zu einem Editor die Benutzung der in Abschnitt 1.5 beschriebenen Entwicklungsumgebungen an. Ein viel verwendeter Compiler ist der GNU C++-Compiler [GCC]. Entwicklungsumgebung und Compiler sind kostenlos erhältlich. Ein Installationsprogramm für einen unter Windows lauffähigen Compiler finden Sie auf der Website zum Buch [CPP]. Hinweise zur Installation finden Sie in Abschnitt A.7 (Seite 957). Bei den meisten Linux-Systemen ist der GNU C++-Compiler enthalten oder kann nachträglich installiert werden. Auf einem Mac mit OS X sollten Sie die Entwicklungsumgebung Xcode mit dem Clang-Compiler[3] installieren (siehe Hinweise dazu ab Seite 961).

1.4 Das erste Programm

Sie lernen hier die Entwicklung eines ganz einfachen Programms kennen. Dabei wird Ihnen zunächst das Programm vorgestellt und wenige Seiten weiter erfahren Sie, wie Sie es eingeben und zum Laufen bringen können. Der erste Schritt besteht in der Formulierung der Aufgabe. Sie lautet: »Lies zwei Zahlen a und b von der Tastatur ein. Berechne die Summe beider Zahlen und zeige das Ergebnis auf dem Bildschirm an.« Die Aufgabe ist so einfach, wie sie sich anhört! Im zweiten Schritt wird die Aufgabe in die Teilaufgaben »Eingabe«, »Berechnung« und »Ausgabe« zerlegt:

Listing 1.1: Das erste Programm!

```
int main() {                // Noch tut dieses Programm nichts!
    // Lies zwei Zahlen ein
```

[2] Statt des Wortungetüms Eingabeaufforderungsfenster oder des Begriffs Terminal werde ich von nun an in der Regel das Wort *Konsole* verwenden.

[3] *http://clang.llvm.org/get_started.html*

```
    /* Berechne die Summe beider
       Zahlen   */
    // Zeige das Ergebnis auf dem Bildschirm an
}
```

Hier sehen Sie schon ein einfaches C++-Programm. Es bedeuten:

`int`	ganze Zahl zur Rückgabe
`main`	Schlüsselwort für Hauptprogramm
`()`	Innerhalb dieser Klammern können dem Hauptprogramm Informationen mitgegeben werden.
`{ }`	Block
`/* ... */`	Kommentar, der über mehrere Zeilen gehen kann
`// ...`	Kommentar bis Zeilenende

Ein durch { und } begrenzter *Block* enthält die Anweisungen an den Rechner. Der *Compiler* übersetzt den Programmtext in eine rechnerverständliche Form. Im obigen Programm sind lediglich *Kommentare* enthalten und noch keine Anweisungen an den Computer, sodass unser Programm (noch) nichts tut.

Kommentare werden einschließlich der Kennungen vom Compiler vollständig ignoriert. Ein Kommentar, der mit `/*` beginnt, ist mit der ersten `*/`-Zeichenkombination beendet, auch wenn er sich über mehrere Zeilen erstreckt. Ein mit `//` beginnender Kommentar endet am Ende der Zeile. Auch wenn Kommentare vom Compiler ignoriert werden, sind sie doch sinnvoll für den menschlichen Leser eines Programms, um ihm die Anweisungen zu erläutern, zum Beispiel für den Programmierer, der Ihr Nachfolger wird, weil Sie befördert worden sind oder die Firma verlassen haben. Kommentare sind auch wichtig für den Autor eines Programms, der nach einem halben Jahr nicht mehr weiß, warum er gerade diese oder jene komplizierte Anweisung geschrieben hat. Sie sehen:

Ein Programm ist »nur« ein Text!

- Der Text hat eine Struktur entsprechend den C++-Sprachregeln: Es gibt Wörter wie hier das Schlüsselwort `main` (in C++ werden alle Schlüsselwörter kleingeschrieben). Es gibt weiterhin Zeilen, Satzzeichen und Kommentare.
- Die Bedeutung des Textes wird durch die Zeilenstruktur nicht beeinflusst. Mit `\` und folgendem ENTER ist eine Worttrennung am Zeilenende möglich. Das Zeichen `\` wird »Backslash« genannt. Mit dem Symbol ENTER ist hier und im Folgenden die Betätigung der großen Taste ↵ rechts auf der Tastatur gemeint.
- Groß- und Kleinschreibung werden unterschieden! `main()` ist nicht dasselbe wie `Main()`.

Weil die Zeilenstruktur für den Rechner keine Rolle spielt, kann der Programmtext nach Gesichtspunkten der Lesbarkeit gestaltet werden. Im dritten Schritt müssen »nur« noch die Inhalte der Kommentare als C++-Anweisungen formuliert werden. Dabei bleiben die Kommentare zur Dokumentation stehen, wie im Beispielprogramm unten zu sehen ist.

Listing 1.2: Summe zweier Zahlen berechnen

```
// cppbuch/k1/summe.cpp
// Hinweis: Alle Programmbeispiele sind von der Internet-Seite zum Buch herunterladbar
// (http://www.cppbuch.de/).
```

```
// Die erste Zeile in den Programmbeispielen gibt den zugehörigen Dateinamen an.
#include<iostream>
using namespace std;

int main() {
   int summe;
   int summand1;
   int summand2;
    // Lies zwei Zahlen ein
    cout << "␣Zwei␣ganze␣Zahlen␣eingeben:";
    cin >> summand1  >> summand2;
    /* Berechne die Summe beider Zahlen */
    summe = summand1 + summand2;

    // Zeige das Ergebnis auf dem Bildschirm an
    cout << "Summe=" << summe << '\n';
    return 0;
}
```

Es sind einige neue Worte dazugekommen, die hier kurz erklärt werden. Machen Sie sich keine Sorgen, wenn Sie nicht alles auf Anhieb verstehen! Alles wird im Verlauf des Buchs wieder aufgegriffen und vertieft. Wie das Programm zum Laufen gebracht wird, erfahren Sie nur wenige Seiten danach.

`#include<iostream>` Einbindung der Ein-/Ausgabefunktionen. Diese Zeile muss in jedem Programm stehen, das Eingaben von der Tastatur erwartet oder Ausgaben auf den Bildschirm bringt. Sie können sich vorstellen, dass der Compiler beim Übersetzen des Programms an dieser Stelle erst alle zur Ein- und Ausgabe notwendigen Informationen liest. Details folgen in Abschnitt 2.3.5.

`using namespace std;` Der Namensraum `std` wird benutzt. Schreiben Sie es einfach in jedes Programm an diese Stelle und haben Sie Geduld: Eine genauere Erklärung folgt später (Seiten 64 und 144).

`int main()` `main()` ist das Hauptprogramm (es gibt auch Unterprogramme). Der zu `main()` gehörende Programmcode wird durch die geschweiften Klammern `{` und `}` eingeschlossen. Ein mit `{` und `}` begrenzter Bereich heißt *Block*. Mit `int` ist gemeint, dass das Programm `main()` nach Beendigung eine Zahl vom Typ `int` (= ganze Zahl) an das Betriebssystem zurückgibt. Dazu dient die unten beschriebene `return`-Anweisung. Normalerweise – das heißt bei ordnungsgemäßem Programmablauf – wird die Zahl 0 zurückgegeben. Andere Zahlen können über das Betriebssystem einem folgenden Programm einen Fehler signalisieren.

`int summe;`
`int summand1;`
`int summand2;` *Deklaration* von Objekten: Mitteilung an den Compiler, der entsprechend Speicherplatz bereitstellt und ab jetzt die Namen `summe`, `summand1` und `summand2` innerhalb des Blocks `{ }` kennt. Es gibt verschiedene Zahlentypen in C++. Mit `int` sind ganze Zahlen gemeint: `summe`, `summand1`, `summand2` sind ganze Zahlen.

`;`	Ein Semikolon beendet jede Deklaration und jede Anweisung (aber keine Verbundanweisung, siehe weiter unten).
`cout`	Ausgabe: `cout` (Abkürzung für *character out* oder *console out*) ist die Standardausgabe. Der Doppelpfeil deutet an, dass alles, was rechts davon steht, zur Ausgabe `cout` gesendet wird, zum Beispiel `cout << summand1;`. Wenn mehrere Dinge ausgegeben werden sollen, sind sie durch << zu trennen.
`cin`	Eingabe: Der Doppelpfeil zeigt hier in Richtung des Objekts, das ja von der Tastatur einen neuen Wert aufnehmen soll. Die Information fließt von der Eingabe `cin` zum Objekt `summand1` beziehungsweise zum Objekt `summand2`.
`=`	Zuweisung: Der Variablen auf der linken Seite des Gleichheitszeichens wird das Ergebnis des Ausdrucks auf der rechten Seite zugewiesen.
`"Text"`	beliebige Zeichenkette, die die Anführungszeichen selbst nicht enthalten darf, weil sie als Anfangs- beziehungsweise Endemarkierung einer Zeichenfolge dienen. Wenn die Zeichenfolge die Anführungszeichen enthalten soll, sind diese als `\"` zu schreiben: `cout << "\"C++\" ist der Nachfolger von \"C\"!";` erzeugt die Bildschirmausgabe *"C++" ist der Nachfolger von "C"!*.
`'\n'`	die Ausgabe des Zeichens `\n` bewirkt eine neue Zeile.
`return 0;`	Unser Programm läuft einwandfrei, es gibt daher 0 zurück. Diese Anweisung darf im `main()`-Programm fehlen, dann wird automatisch 0 zurückgegeben.

`<iostream>` ist ein Header. Dieser aus dem Englischen stammende Begriff (head = dt. Kopf) drückt aus, dass Zeilen dieser Art am Anfang eines Programmtextes stehen. Der Begriff wird im Folgenden verwendet, weil es zurzeit keine gängige deutsche Entsprechung gibt. Einen Header mit einem Dateinamen gleichzusetzen, ist meistens richtig, nach dem C++-Standard aber nicht zwingend.

`summand1`, `summand2` und `summe` sind veränderliche Daten und heißen Variablen. Sie sind Objekte eines vordefinierten Grunddatentyps für ganze Zahlen (`int`), mit denen die üblichen Ganzzahloperationen wie `+`, `-` und `=` durchgeführt werden können. Der Begriff »Variable« wird für ein veränderliches Objekt gebraucht. Für Variablen gilt:

- Objekte müssen deklariert werden. `int summe;` ist eine Deklaration, wobei `int` der *Datentyp* des Objekts `summe` ist, der die Eigenschaften beschreibt. Entsprechendes gilt für `summand1` und `summand2`. Die Objektnamen sind frei wählbar im Rahmen der unten angegebenen Konventionen. Unter *Deklaration* wird verstanden, dass der Name dem Compiler bekannt gemacht wird. Wenn dieser Name danach im Programm versehentlich falsch geschrieben wird, z. B. `sume = summand1 + summand2;` im Programm auf Seite 35, kennt der Compiler den falschen Namen `sume` nicht und gibt eine Fehlermeldung aus. Damit dienen Deklarationen der Programmsicherheit.
- Objektnamen bezeichnen Bereiche im Speicher des Computers, deren Inhalte verändert werden können. Die Namen sind symbolische Adressen, unter denen der Wert

gefunden wird. Über den Namen kann dann auf den aktuellen Wert zugegriffen werden (siehe Abbildung 1.1).

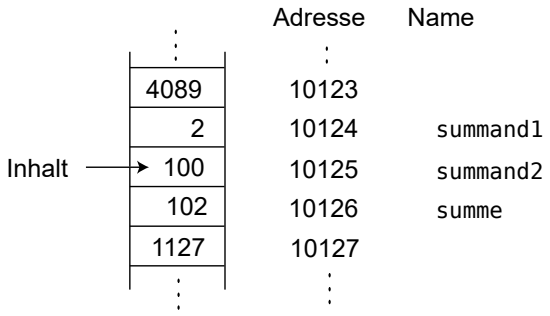


Abbildung 1.1: Speicherbereiche mit Adressen

Der Speicherplatz wird vom Compiler reserviert. Man spricht dann von der *Definition* der Objekte. Definition und Deklaration werden unterschieden, weil es auch Deklarationen ohne gleichzeitige Definition gibt, doch davon später. Zunächst sind die Deklarationen zugleich Definitionen. Abbildung 1.2 zeigt den Ablauf der Erzeugung eines lauffähigen Programms. Ein Programm ist ein Text, von Menschenhand geschrieben (über Programmgeneratoren soll hier nicht gesprochen werden) und dem Rechner unverständlich. Um dieses Programm auszuführen, muss es erst vom Compiler in eine für den Computer verständliche Form übersetzt werden. Der Compiler ist selbst ein Programm, das bereits in maschinenverständlicher Form vorliegt und speziell für diese Übersetzung zuständig ist. Nach Eingabe des Programmtextes mit dem Editor können Sie den Compiler starten.

Ein Programmtext wird auch »Quelltext« (englisch *source code*) genannt. Der Compiler erzeugt aus dem Quelltext den Objektcode, der noch nicht ausführbar ist. Hinter den einfachen Anweisungen `cin >> ...` und `cout << ...` verbergen sich eine Reihe von Aktivitäten wie die Abfrage der Tastatur und die Ansteuerung des Bildschirms, die nicht speziell programmiert werden müssen, weil sie schon in vorübersetzter Form in Bibliotheksdateien vorliegen. Die Aufrufe dieser Aktivitäten im Programm müssen mit den dafür vorgesehenen Algorithmen in den Bibliotheksdateien zusammengebunden werden, eine Aufgabe, die der *Linker* übernimmt, auch *Binder* genannt. Der Linker bindet Ihren Objektcode mit dem Objektcode der Bibliotheksdateien zusammen und erzeugt daraus ein ausführbares Programm, das nun gestartet werden kann. Der Aufruf des Programms bewirkt, dass der *Lader*, eine Funktion des Betriebssystems, das Programm in den Rechnerspeicher lädt und startet. Diese Schritte werden stets ausgeführt, auch wenn sie in den Programmentwicklungsumgebungen verborgen ablaufen. Bibliotheksmodule können auch während der Programmausführung geladen werden (nicht im Bild dargestellt). Weitere Details werden in Abschnitt 2.3 erläutert.

Wie bekomme ich ein Programm zum Laufen?

Nachdem Sie den Programmtext mit einem Editor geschrieben haben, kann er übersetzt (compiliert) und ausgeführt werden. Integrierte Entwicklungsumgebungen (englisch *Integrated Development Environment, IDE*) haben einen speziell auf Programmierzwecke zugeschnittenen Editor, der darüber hinaus auf Tastendruck oder Mausklick die Überset-

1.8.5 Wiederholungen

Häufig muss die gleiche Teilaufgabe oft wiederholt werden. Denken Sie nur an die Summation von Tabellenspalten in der Buchführung oder an das Suchen einer bestimmten Textstelle in einem Buch. In C++ gibt es zur Wiederholung von Anweisungen drei verschiedene Arten von Schleifen. In einer Schleife wird nach Abarbeitung einer Teilaufgabe (zum Beispiel Addition einer Zahl) wieder an den Anfang zurückgekehrt, um die gleiche Aufgabe noch einmal auszuführen (Addition der nächsten Zahl). Durch bestimmte Bedingungen gesteuert, zum Beispiel Ende der Tabelle, bricht irgendwann die Schleife ab.

Schleifen mit while

Abbildung 1.7 zeigt die Syntax von `while`-Schleifen. *AnweisungOderBlock* ist wie auf Seite 67 definiert. Die Bedeutung einer `while`-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis ungleich 0 oder `true` liefert, wird die Anweisung bzw. der Block ausgeführt. Die Bedingung wird auf jeden Fall zuerst geprüft. Wenn die Bedingung von vornherein unwahr ist, wird die Anweisung gar nicht erst ausgeführt (siehe Abbildung 1.8). Die Anweisung oder der Block innerhalb der Schleife heißt *Schleifenkörper*. Schleifen können wie `if`-Anweisungen beliebig geschachtelt werden.

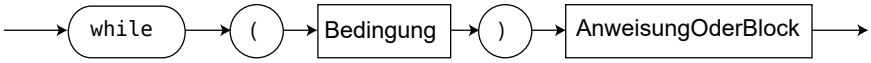
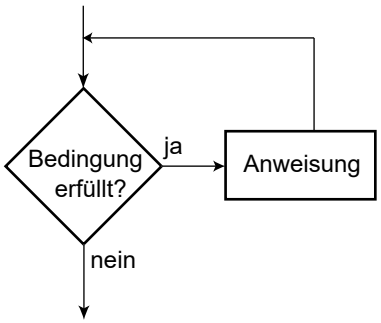


Abbildung 1.7: Syntaxdiagramm einer `while`-Schleife

```
while(Bedingung1)       // geschachtelte Schleifen, ohne und mit geschweiften Klammern
   while(Bedingung2) {
        .....
        while(Bedingung3) {
            .....
        }
   }
```




```
while (Bedingung) {
  Anweisung
}
```

Abbildung 1.8: Flussdiagramm für eine `while`-Anweisung

Beispiele

- Unendliche Schleife:

```
while(true)
   Anweisung
```

- Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```
while(false)
   Anweisung
```

- Summation der Zahlen 1 bis 99:

```
int sum = 0;
int n = 1;
int grenze = 99;
while(n <= grenze) {
   sum += n++;
}
```

- Berechnung des größten gemeinsamen Teilers ggT(x, y) für zwei natürliche Zahlen x und y nach Euklid. Es gilt:
  - ggT(x, x), also x = y: Das Resultat ist x.
  - ggT(x, y) bleibt unverändert, falls die größere der beiden Zahlen durch die Differenz ersetzt wird, also ggT(x, y) == ggT(x, y-x), falls x < y.

  Das Ersetzen der Differenz geschieht im folgenden Beispiel iterativ, also durch eine Schleife.

Listing 1.14: Beispiel für `while`-Schleife

```
// cppbuch/k1/ggt.cpp    Berechnung des größten gemeinsamen Teilers
#include <iostream>
using namespace std;

int main() {
   unsigned int x;
   unsigned int y;
   cout <<"2␣Zahlen␣>␣0␣eingeben␣:";
   cin >> x >> y;
   cout << "Der␣ggT␣von␣" << x << "␣und␣" << y << "␣ist␣";
   while( x!= y) {
      if(x > y) {
         x -= y;
      }
      else {
         y -= x;
      }
   }
   cout << x << '\n';
}
```

Innerhalb einer Schleife muss es eine Veränderung derart geben, dass die Bedingung irgendwann einmal unwahr wird, sodass die Schleife abbricht (man sagt auch *terminiert*). Unbeabsichtigte »unendliche« Schleifen sind ein häufiger Programmierfehler. Im ggT-Beispiel ist leicht erkennbar, dass die Schleife irgendwann beendet sein *muss*:

1. Bei jedem Durchlauf wird mindestens eine der beiden Zahlen kleiner.
2. Die Zahl 0 kann nicht erreicht werden, da immer eine kleinere von einer größeren Zahl subtrahiert wird. Die `while`-Bedingung schließt die Subtraktion gleich großer Zahlen aus, und nur die könnte 0 ergeben.

Daraus allein ergibt sich, dass die Schleife beendet wird, und zwar in weniger als x Schritten, wenn x die anfangs größere Zahl war. Im Allgemeinen sind es erheblich weniger, wie eine genauere Analyse ergibt.

Tipp

Die Anweisungen zur Veränderung der Bedingung sollen möglichst an das Ende des Schleifenkörpers gestellt werden, um sie leicht finden zu können.

Schleifen mit do while

Abbildung 1.9 zeigt die Syntax einer `do while`-Schleife. *AnweisungOderBlock* ist wie auf Seite 67 definiert. Die Anweisung oder der Block einer `do while`-Schleife wird ausgeführt, und *erst anschließend* wird die Bedingung geprüft. Ist sie wahr, wird die Anweisung ein weiteres Mal ausgeführt usw. Die Anweisung wird also mindestens einmal ausgeführt.

Im Flussdiagramm ist die Anweisung ein Block (siehe rechts in der Abbildung 1.10). `do while`-Schleifen eignen sich unter anderem gut zur sicheren Abfrage von Daten, indem die Abfrage so lange wiederholt wird, bis die abgefragten Daten in einem plausiblen Bereich liegen, wie im Primzahlprogramm unten zu sehen ist.

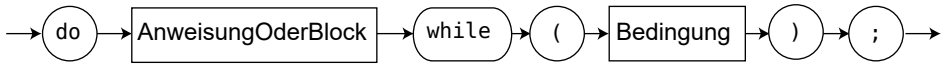


Abbildung 1.9: Syntaxdiagramm einer `do while`-Schleife

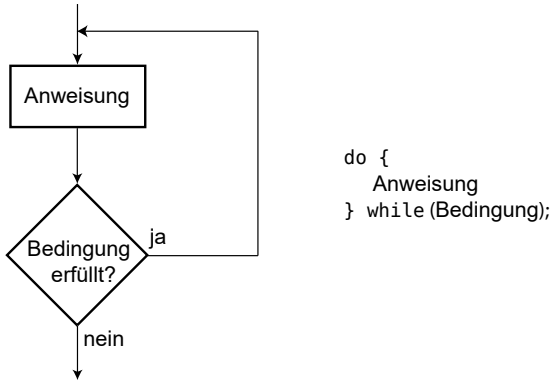


Abbildung 1.10: Flussdiagramm für eine `do while`-Anweisung

Es empfiehlt sich zur besseren Lesbarkeit, `do while`-Schleifen strukturiert zu schreiben. Die schließende geschweifte Klammer soll genau unter dem ersten Zeichen der Zeile stehen, die die öffnende geschweifte Klammer enthält. Dadurch und durch Einrücken des dazwischen stehenden Textes ist sofort der Schleifenkörper erkennbar.

```
do {
    Anweisungen
} while(Bedingung);
```

Das *direkt hinter* die abschließende geschweifte Klammer geschriebene `while` macht unmittelbar deutlich, dass dieses `while` zu einem `do` gehört. Das ist besonders wichtig, wenn der Schleifenkörper in einer Programmliste über die Seitengrenze ragt. Eine `do while`-Schleife kann stets in eine `while`-Schleife umgeformt werden (und umgekehrt).

Listing 1.15: Beispiel für `do while`-Schleifen

```
// cppbuch/k1/primzahl.cpp: Berechnen einer Primzahl, die auf eine gegebene Zahl folgt
#include <iostream>
#include <cmath>
using namespace std;

int main() {
  // Mehrere, durch " getrennte Texte ergeben eine lange Zeile in der Ausgabe.
  cout << "Berechnung␣der␣ersten␣Primzahl,␣die␣>="
          "␣der␣eingegebenen␣Zahl␣ist\n";
  long z;
  // do while-Schleife zur Eingabe und Plausibilitätskontrolle
  do {                                          // Abfrage, solange z ≤ 3 ist
      cout << "Zahl␣>␣3␣eingeben␣:";
      cin >> z;
  } while(z <= 3);
  if(z % 2 == 0) {                  // Falls z gerade ist: nächste ungerade Zahl nehmen
     ++z;
  }
  bool gefunden {false};
  do {
      // limit = Grenze, bis zu der gerechnet werden muss.
      // sqrt() arbeitet mit double, daher wird der Typ explizit umgewandelt.
      long limit {1 + static_cast<long>( sqrt(static_cast<double>(z)))};
      long rest;
      long teiler {1};
      do {                           // Kandidat z durch alle ungeraden Teiler dividieren
          teiler += 2;
          rest    = z % teiler;
      } while(rest > 0 && teiler < limit);
      if(rest > 0 && teiler >= limit) {
         gefunden = true;
      }
      else {                              // sonst nächste ungerade Zahl untersuchen:
         z += 2;
      }
  } while(!gefunden);
  cout << "Die␣nächste␣Primzahl␣ist␣" << z << '\n';
}
```

Schleifen mit for

Die letzte Art von Schleifen ist die `for`-Schleife. Sie wird häufig eingesetzt, wenn die Anzahl der Wiederholungen vorher feststeht, aber das muss durchaus nicht so sein. Abbildung 1.11 zeigt die Syntax einer `for`-Schleife.

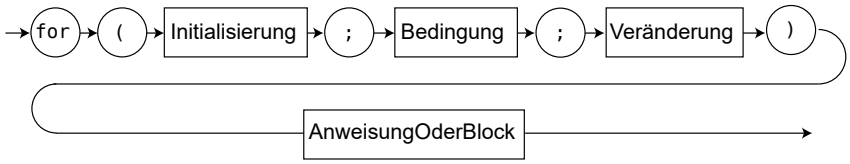


Abbildung 1.11: Syntaxdiagramm einer `for`-Schleife

Der zu wiederholende Teil (Anweisung oder Block) wird auch Schleifenkörper genannt. Beispiel: ASCII-Tabelle im Bereich 65 ... 69 ausgeben

```
for(int i = 65; i <= 69; ++i) {
    cout << i << "␣" << static_cast<char>(i) << '\n';
}
```

Bei der Abarbeitung werden die folgenden Schritte durchlaufen:

1. Durchführung der Initialisierung, zum Beispiel Startwert für eine Laufvariable festlegen. Eine Laufvariable wird wie `i` in der Beispielschleife als Zähler benutzt.
2. Prüfen der Bedingung.
3. Falls die Bedingung wahr ist, zuerst die Anweisung und dann die Veränderung ausführen.

Die Laufvariable `i` kann auch außerhalb der runden Klammern deklariert werden, dies gilt aber als schlechter Stil. Der Unterschied besteht darin, dass außerhalb der Klammern deklarierte Laufvariablen noch über die Schleife hinaus gültig sind.

```
int i;                                          // nicht empfohlen
for(i = 0; i < 100; ++i) {
    // Programmcode, i ist hier bekannt
}
// i ist weiterhin bekannt ...
```

Im Fall der Deklaration innerhalb der runden Klammern bleibt die Gültigkeit auf den Schleifenkörper beschränkt:

```
for(int i = 0; i < 100; ++i) {                  // empfohlen
    // Programmcode, i ist hier bekannt
}
// i ist hier nicht mehr bekannt
```

Die zweite Art erlaubt es, `for`-Schleifen als selbstständige Programmteile hinzuzufügen oder zu entfernen, ohne Deklarationen in anderen Schleifen ändern zu müssen. Derselbe Mechanismus gilt für Deklarationen in den runden Klammern von `if`-, `while`- und `switch`-Anweisungen.

Listing 1.16: Beispiel für `for`-Schleife

```
// cppbuch/k1/fakultaet.cpp
#include <iostream>
using namespace std;

int main() {
   cout << "Fakultät␣berechnen.␣Zahl␣>=␣0?␣:";
   unsigned int n;
```

```
   cin >> n;
   unsigned long fak {1L};
   for(unsigned int i = 2; i <= n; ++i) {
      fak *= i;
   }
   cout << n << "!␣␣␣=␣␣␣" << fak << '\n';
}
```

Verändern Sie niemals die Laufvariable innerhalb des Schleifenkörpers! Das Auffinden von Fehlern würde durch die Änderung erschwert.

```
for(int i = 65; i < 70; ++i) {
    // eine Seite Programmcode
    --i; // irgendwo dazwischen erzeugt eine unendliche Schleife
    // noch mehr Programmcode
}
```

Auch wenn der Schleifenkörper nur aus einer Anweisung besteht, wird empfohlen, ihn in geschweiften Klammern `{ }` einzuschließen.

Äquivalenz von for und while

Eine `for`-Schleife entspricht direkt einer `while`-Schleife, sie ist im Grunde nur eine Umformulierung, solange nicht `continue` vorkommt (das im folgenden Abschnitt beschrieben wird):

```
for(Initialisierung; Bedingung; Veraenderung)
  Anweisung
```

ist äquivalent zu:

```
{
   Initialisierung;
   while(Bedingung) {
      Anweisung
      Veraenderung;
   }
}
```

Die äußeren Klammern sorgen dafür, dass in der Initialisierung deklarierte Variablen wie bei der `for`-Schleife nach dem Ende nicht mehr gültig sind. `Anweisung` kann wie immer auch eine Verbundanweisung (Block) sein, in der mehrere Anweisungen stehen können, durch geschweifte Klammern begrenzt. Die umformulierte Entsprechung des obigen Beispiels (ASCII-Tabelle von 65 ... 69 ausgeben) lautet:

```
{
  int i {65};                                                  // Initialisierung
  while(i < 70) {                                              // Bedingung
      cout << i << "␣" << static_cast<char>(i) << '\n';        // Anweisung
      ++i;                                                     // Veränderung
  }
}
```

Objekt 1 wird erzeugt.
neuer Block
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. Ähnlichkeiten mit der Aufgabe 1.18 von Seite 108 sind beabsichtigt. 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.

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 1.18 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 halber 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 [Oe13]. In diesem einfachen Fall konzentrieren wir uns gleich auf eine Lösung mit C++. Ein mögliches `main()`-Programm könnte wie folgt aussehen:

Listing 3.35: `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 3.36: 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 auto& getName() const {
    return name;
  }

  const auto& 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 3.37: 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 1.18 von Seite 108 gelöst oder deren Lösung nachgesehen haben.

```
if(pa == nullptr) {                          // Fehlerhafte Annahme
...
}
```

Der Fehler liegt in dem undefinierten Wert von `pa` nach der Löschoperation. Falls ein Zeiger nach dem Löschen noch verwendet werden kann, setzen Sie ihn direkt nach dem `delete` mit `pa = nullptr;` auf Null. Dann kann er geprüft werden und es gibt eine definierte Fehlermeldung. Besser noch ist jedoch die Vermeidung solcher Konstruktionen zugunsten der Kapselung von `new` und `delete` oder der Verwendung von `unique_ptr` bzw. `shared_ptr`, siehe folgenden Abschnitt.

21.2.18 Speicherbeschaffung und -freigabe kapseln

Die Operatoren `new` und `delete` sind stets paarweise zu verwenden. Um Speicherfehler zu vermeiden, empfiehlt sich das »Verpacken« dieser Operationen in Konstruktor und Destruktor wie bei der Vektorklasse des Kapitels 8 oder die Verwendung der »Smart Pointer« (`unique_ptr`, `shared_ptr`), siehe unten. Ein weiterer Vorteil ist die korrekte Speicherfreigabe bei Exceptions (siehe unten).

21.2.19 Programmierrichtlinien einhalten

Das Einhalten von Programmierrichtlinien unterstützt das Schreiben gut lesbarer Programme. Es gibt einige dieser Richtlinien, die sich in großen Teilen ähneln. Oft hat eine softwareentwickelnde Firma eine eigene Richtlinie.

21.3 Exception-sichere Beschaffung von Ressourcen

Wenn eine Ressource beschafft werden soll, kann ein Problem auftreten. Das kann eine Datei sein, die nicht gefunden wird oder ein Fehlschlag beim Beschaffen von Speicher. Weil die Probleme strukturell ähnlich sind, beschränke ich mich hier auf Probleme bei der dynamischen Beschaffung von Speicher. Das kann in einer Methode oder auch schon im Konstruktor auftreten. Ziel ist es, beim Auftreten von Exceptions kein Speicherleck zu erzeugen und die betroffenen Objekte in ihrem Zustand zu belassen.

21.3.1 Sichere Verwendung von unique_ptr und shared_ptr

Bei der Konstruktion eines `unique_ptr` bzw. `shared_ptr` (Beschreibung in Kapitel 32) soll die Erzeugung des Zeigers mit `new` stets innerhalb der Parameterliste geschehen.

```
Ressource *pr = new Ressource(id);
// weiterer Code
shared_ptr<Ressource> sptr(pr);                    // 1. falsch!

shared_ptr<Ressource> p(new Ressource(id));        // 2. nicht perfekt, aber richtig!
```

Begründung: Im Fall 1 kann es die folgenden Fehler geben:

- Es wäre möglich, `delete pr` aufzurufen. Bei der Zerstörung von `sptr` wird der Destruktor für `*pr` auch aufgerufen, dann also insgesamt *zweimal*.
- Es könnte sein, dass im Bereich »weiterer Code« eine Exception auftritt. Der resultierende Sprung des Programmablaufs aus dem aktuellen Kontext führt dazu, dass `delete` nicht mehr möglich ist. Das erzeugte Objekt bleibt unerreichbar im Speicher.

Im Fall 2 kann dies nicht geschehen: Wenn eine Exception geworfen wird, werden automatisch die Destruktoren aller auf dem Laufzeit-Stack befindlichen Objekte des verlassenen Gültigkeitsbereichs aufgerufen, also auch der Destruktor des `shared_ptr`-Objekts, der wiederum für das Löschen des übergebenen Objekts sorgt – eine Realisierung des Prinzips »Resource Acquisition Is Initialization« (RAII, siehe Glossar). Entsprechendes gilt für `unique_ptr`. Noch besser, weil einfacher, ist die gänzliche Vermeidung von `new`, wie der folgende Abschnitt zeigt.

21.3.2 So vermeiden Sie new und delete!

Wie gezeigt, muss man sich um `delete` nicht mehr kümmern, wenn `unique_ptr` oder `shared_ptr` eingesetzt werden. Die Hilfsfunktionen `make_unique` (siehe Abschnitt 32.1.1) und `make_shared` (siehe Abschnitt 32.2.1) vereinfachen die Schreibweise weiter, sodass auch `new` entfällt. Dabei werden nur noch die Argumente für den Konstruktor übergeben. Im folgenden Beispiel benötigt der Konstruktor nur ein `int`-Argument:

Listing 21.13: `new` und `delete` vermeiden

```
// vector mit shared_ptr
std::vector<std::shared_ptr<Ressource>> vec1(10);
vec1[0] = std::shared_ptr<Ressource>(new Ressource(1));                // mit new
// einfacher ist:
vec1[0] = std::make_shared<Ressource>(1);                              // ohne new

// vector mit unique_ptr
std::vector<std::unique_ptr<Ressource>> vec2(10);
vec2[0] = std::unique_ptr<Ressource>(new Ressource(2));                // mit new
// einfacher ist:
vec2[0] = std::make_unique<Ressource>(2);                              // ohne new
```

Die Zuweisung im zweiten Beispiel ist nur möglich, weil auf der rechten Seite ein R-Wert (temporäres Objekt) steht. Wäre es nicht temporär, gäbe es eine Fehlermeldung des Compilers. Beispiel:

```
auto uniqueptr999 = std::make_unique<Ressource>(999);
vec2[0] = uniqueptr999;                                                // Fehler!
```

Damit wird verhindert, dass es zwei `unique_ptr`-Objekte geben kann, die auf dasselbe Heap-Objekt verweisen. Eine Kopie ist nicht erlaubt.

So vermeiden Sie new[] und delete[]!

Nach `new[]` nur `delete` statt `delete[]` zu schreiben, wäre ein Fehler. Er ist leicht zu vermeiden, wenn auf `new[]` zugunsten von `vector` verzichtet wird. In den meisten Fällen wird das ohne Probleme möglich sein. Ein Beispiel dafür ist die String-Klasse von Seite

257. Manche empfehlen die Verwendung von `unique_ptr<T>` (siehe unten). Innerhalb einer Klasse, deren Objekte kopierbar sein sollen und für die Speicherplatz beschafft werden soll, würde man nur den Destruktor sparen, nicht aber den Kopierkonstruktor und Zuweisungsoperator. Die Verwendung von `vector` spart auch diese ein.

21.3.3 shared_ptr für Arrays korrekt verwenden

Der Destruktor eines `shared_ptr`-Objekts wendet `delete` auf den intern gespeicherten Zeiger an, wenn kein anderer `shared_ptr` auf die Ressource verweist. Dies kann zu einem Speicherleck führen, wenn der Zeiger mit `new []` erzeugt wurde, wie auf Seite 220 beschrieben. Zwar kann es sein, dass im Fall der falschen Anweisung das Speicherleck nicht bemerkt wird oder dass der Compiler aus dem Kontext den Fehler erkennt und korrigiert. Nach [ISOC++] ist das Verhalten jedoch undefiniert, das heißt, alle Möglichkeiten vom Weiterlaufen des Programms bis zum Absturz des Programms sind »legal«. Damit ist auch das Verhalten des folgenden Programms undefiniert:

```
void funktion() {
   shared_ptr<int> p(new int[10]);          // falsch
   // ... etliche Zeilen weggelassen
}                                           // Memory-Leak möglich
```

Die Lösung des Problems ist die Übergabe eines `std::default_delete<X[]>`-Objekts an den Konstruktor. Der `shared_ptr`-Destruktor ruft den `operator()()` des übergebenen Objekts auf, wenn kein anderer `shared_ptr` auf die Ressource verweist. Der überladene
C++17 Funktionsoperator enthält die `delete[]`-Anweisung. Einfacher ist es, beim Typ gleich den Arraytyp `X[]` statt nur `X` zu vermerken. Das Listing 21.14 zeigt beide Fälle. Das Programm dokumentiert die Löschung der Objekte.

Listing 21.14: `shared_ptr` mit Arrays verwenden

```
// cppbuch/k32/sharedptr_arrays.cpp
#include <iostream>
#include <memory>

struct X {
  X(int i = 0)
    : wert(i) {
  }
  ~X() {
    std::cout << "X(" << wert << ")␣gelöscht\n";;
  }
  int wert;
};

int main() {                                // Zwei Varianten
  std::shared_ptr<X> p1(new X[5], std::default_delete<X[]>());
  std::shared_ptr<X[]> p2(new X[5]);        // <X[]> statt <X>!
  for(int i=0; i < 5; ++i) {
    p1.get()[i].wert = i + 1;               // Zuweisen eines Werts
    p2[i].wert = 10*i + 11;                 // Kurzform geht nur bei shared_ptr<X[]>
  }
}
```

Statt `std::default_delete<X[]>` können Sie eine selbstgeschriebene Klasse nehmen. Sie muss nur die folgende Funktion enthalten:

```
void operator()(T* ptr) {                    // T = Template-Parameter
  delete[] ptr;
}
```

Tipp

Sie können die beschriebenen möglichen Probleme vermeiden, wenn Sie auf `shared_ptr` für Arrays verzichten und stattdessen einen `shared_ptr` mit einem `vector` verwenden, etwa so: `auto ptr = std::make_shared<std::vector<X>>();`, siehe auch Abschnitt 21.3.2 oben. *Oder noch einfacher*: Genügt vielleicht nur ein `vector<X>` statt eines `shared_ptr` für den Vektor?

21.3.4 unique_ptr für Arrays korrekt verwenden

Das Problem des vergessenen Deleters tritt bei `unique_ptr` nicht auf, weil der Typ des für die Löschung zuständigen Objekts zur Schnittstelle gehört.

```
template <class T, class D = default_delete<T>> class unique_ptr;
```

Wenn ein Arraytyp, gekennzeichnet durch `[]`, eingesetzt wird, kann der zweite Typ entfallen. Er wird dann durch den vorgegebenen (default) Typ für den Deleter ersetzt, der `delete []` aufruft. Die Funktion `f()` zeigt, wie es geht.

Listing 21.15: `unique_ptr` und Array

```
// cppbuch/k32/arrayunique.cpp
#include<memory>

void f() {
  std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);
  // entspricht std::unique_ptr<int[]> arr(new int[10]);
  // Benutzung des Arrays weggelassen
}     // kein Memory-Leak, Array wird korrekt gelöscht

int main() {
  f();
}
```

Um den voreingestellten (englisch *default*) Template-Parameter sichtbar zu machen, könnte die erste Zeile in `f()` so geschrieben werden:

```
std::unique_ptr<int[], std::default_delete<int[]>>
                            arr = std::make_unique<int[]>(10);
```

Tipp

Im Verzeichnis *cppbuch/k11/move/unique_ptr* finden Sie ein Beispiel für einen String-Typ, der einen `unique_ptr` auf ein `char`-Array verwendet. Überlegen Sie sich bei einem ähnlichen Problem aber, ob nicht doch ein `vector` die einfachere Lösung ist.

21.3.5 Exception-sichere Funktion

Listing 21.16: Exception-unsichere Funktion

```
void func() {                                   // fehlerhaft, siehe Text
  Datum heute;                                  // Stack-Objekt
  Datum *pD =  new Datum;                       // Heap-Objekt beschaffen
  heute.aktuell();                              // irgendeine Berechnung
  pD->aktuell();                                // irgendeine Berechnung
  delete pD;                                    // Heap-Objekt freigeben
}
```

Wenn die Funktion `aktuell()` eine Ausnahme auswirft, wird der Destruktor von Objekt `heute` gerufen und das Objekt vom Stack geräumt. Das Objekt, auf das `pD` zeigt, wird jedoch niemals freigegeben, weil `delete` nicht mehr erreicht wird und `pD` außerhalb des Blocks unbekannt ist:

Listing 21.17: Anwendung der Funktion von Listing 21.16

```
int main() {
  try {
     func();
  }
  catch(...) {
     //... pD ist hier unbekannt
  }
}
```

Aus diesem Grund sollen ausschließlich Stack-Objekte (automatische Objekte) verwendet werden, wenn Exceptions auftreten. Dies ist immer möglich, wenn Beschaffung und Freigabe eines dynamischen Objekts innerhalb eines Stack-Objekts versteckt werden. Das Hilfsmittel dazu kennen wir bereits, nämlich die »intelligenten« Zeiger aus Abschnitt 8.5:

Listing 21.18: Exception-sichere Funktion

```
void func() {
     // shared_ptr der Standardbibliothek, siehe Abschnitt 8.5.1. Header: <memory>
     auto pDshared = std::make_shared<Datum>();
     pDshared->aktuell();                       // irgendeine Berechnung
}
```

Nun ist `pDshared` ein automatisches Objekt. Wenn jetzt eine Exception auftritt, gibt es kein Speicherleck, weil der Destruktor von `pDshared` den beschafften Speicher freigibt.

21.3.6 Exception-sicherer Konstruktor

Das Ziel, den Zustand eines Objekts bei Auftreten einer Exception unverändert zu lassen, ist in diesem Fall nicht erreichbar – das Objekt wird ja erst durch den Konstruktor erzeugt. Es geht also darum, dass

1. Ressourcen, die innerhalb des Konstruktors beschafft werden, freigegeben werden, und dass
2. Exceptions beim Aufrufer aufgefangen werden können.

In diesem Zusammenhang ist es wichtig zu wissen, wie sich C++ verhält, wenn in einem Konstruktor eine Exception auftritt.

Verhalten bei einer Exception im Konstruktor

- Für alle vollständig erzeugten (Sub-)Objekte wird der Destruktor aufgerufen. »Vollständig erzeugt« heißt, dass der Konstruktor bis zum Ende durchlaufen wurde.
- Für *nicht* vollständig erzeugte (Sub-)Objekte wird *kein* Destruktor aufgerufen.

Damit ist klar, dass es nicht mehrere rohe Zeiger, die auf Heap-Objekte verweisen, als Attribute einer Klasse geben sollte. Bei einer Exception bei der Erzeugung des letzten würde der Destruktor des ersten nicht aufgerufen werden. Abhilfe: Heap-Objekte nur mit `unique_ptr` bzw. `shared_ptr` realisieren – oder auf Heap-Objekte verzichten, z.B. weil ein Vektor genommen werden könnte.

21.3.7 Exception-sichere Zuweisung

Wenn bei einer Kopie Speicher beschafft werden muss, ist es zuerst zu erledigen! Der Grund: Falls es dabei eine Exception geben sollte, würden alle folgenden, den Zustand des Objekts verändernden Anweisungen gar nicht erst ausgeführt. Die Problematik findet sich typischerweise beim Kopierkonstruktor und dem Zuweisungsoperator. Dazu gehören auch der Kurzformoperator += und die Bildung temporärer Objekte. Sehen wir uns dazu eine andere (und umständliche) Lösung für den Zuweisungsoperator von Seite 358 an:

Listing 21.19: Exception-sicherer Zuweisungsoperator

```
template<typename T>                                          // Exception-sicher
Vektor<T>& Vektor<T>::operator=(const Vektor<T>& v) {   // Zuweisung
  T* temp = new T[v.anzahl];                          // zuerst neuen Platz beschaffen
  for(std::size_t i = 0; i < v.anzahl; ++i) {              // kopieren
    temp[i] = v.start[i];
  }
  delete [] start;                                  // alten Platz freigeben
  anzahl = v.anzahl;                                // Verwaltungsinformation aktualisieren
  start = temp;
  return *this;
}
```

Man könnte vordergründig daran denken, erst den alten Platz freizugeben, weil er ohnehin nicht mehr gebraucht wird, und dabei in der Summe sogar Speicher sparen, wenn nämlich bei `new` der alte Speicherplatz wiederverwendet werden sollte. Auch bräuchte man die Variable `temp` nicht:

Listing 21.20: *Nicht* Exception-sicherer Zuweisungsoperator

```
template<typename T>                                          // NICHT Exception-sicher!
Vektor<T>& Vektor<T>::operator=(const Vektor<T>& v) {     // Zuweisung
  delete [] start;                                  // weg damit, es wird schon gutgehen!
  start = new T[v.anzahl];                                 // neuen Platz beschaffen
  for(std::size_t i = 0; i < v.anzahl; ++i) {              // kopieren
    start[i] = v.start[i];
  }
```

```
  anzahl = v.anzahl;                              // Verwaltungsinformation aktualisieren
  return *this;
}
```

Wenn bei der Speicherplatzbeschaffung ein Problem auftreten sollte, wäre der Inhalt des Objekts durch das direkt vorangegangene `delete` zerstört! Von dem Problem, dass `&v == this` sein könnte, will ich gar nicht erst reden.

Der »swap-Trick« liefert eine noch bessere Möglichkeit, die Zuweisung Exception-sicher zu gestalten. Sie sahen ihn bereits auf Seite 358. Dieses Muster lässt sich auf jede Klasse übertragen. Sie muss nur eine passende `swap()`-Methode besitzen:

Listing 21.21: Schema für einen einfachen Exception-sicheren Zuweisungsoperator

```
Klasse& Klasse::operator=(Klasse kopie)  {        // temporäre Kopie per Wert
  swap(kopie);                                    // wirft keine Exception
  return *this;
}
```

21.4 Empfehlungen zur Thread-Programmierung

21.4.1 Warten auf die Freigabe von Ressourcen

Verwenden Sie die Konstruktionen

```
while(ressourcenNochNichtBereit) {
  cond.wait(lock);
}
```

mit einer Bedingungsvariablen `cond` und einem Lock-Objekt `lock`. Eine Begründung finden Sie auf Seite 517 und davor auch ein Beispiel. Die Alternative

```
while(ressourcenNochNichtBereit) {
  sleep(zeitdauer);
}
```

soll nur genommen werden, wenn sich die Variante mit `wait()` als schwierig erweist; solche Fälle gibt es. Keinesfalls sollten Sie

```
while(ressourcenNochNichtBereit) {
}
```

schreiben – es würde sinnlos CPU-Zeit verbraten. Warum wird oben nicht

```
if(ressourcenNochNichtBereit) {                   // nicht empfehlenswert
  cond.wait(lock);
}
```

22

Von der UML nach C++

Dieses Kapitel behandelt die folgenden Themen:

- Vererbung
- Interfaces
- Assoziationen
- Multiplizität
- Aggregation
- Komposition

Die Unified Modeling Language (UML) ist eine weit verbreitete grafische Beschreibungssprache für Klassen, Objekte, Zustände, Abläufe und noch mehr. Sie wird vornehmlich in der Phase der Analyse und des Softwareentwurfs eingesetzt. Auf die UML-Grundlagen wird hier nicht eingegangen; dafür gibt es gute Bücher wie [Oe13]. Hier geht es darum, die wichtigsten UML-Elemente aus Klassendiagrammen in C++-Konstruktionen, die der Bedeutung des Diagramms möglichst gut entsprechen, umzusetzen. Die vorgestellten C++-Konstruktionen sind Muster, die als Vorlage dienen können. Diese Muster sind nicht einzigartig, sondern nur Empfehlungen, die Umsetzung zu gestalten. Im Einzelfall kann eine Variation sinnvoll sein.

22.1 Vererbung

Über Vererbung als »ist ein«-Beziehung ist in diesem Buch schon einiges gesagt worden, das hier nicht wiederholt werden muss. Sie finden alles dazu in Kapitel 6. Die Abbildung 22.1 zeigt das zugehörige UML-Diagramm.

Abbildung 22.1: Vererbung (»ist ein«-Beziehung)

In vielen Darstellungen wird die Oberklasse oberhalb der abgeleiteten Unterklasse dargestellt; in der UML ist aber nur der Pfeil mit dem Dreieck entscheidend, nicht die relative Lage. In C++ wird Vererbung syntaktisch durch »`: public`« ausgedrückt:

Listing 22.1: Syntaktische Repräsentation der Vererbung

```
class Unterklasse : public Oberklasse {
  // ... Rest weggelassen
};
```

22.2 Interface anbieten und nutzen

Interface anbieten

Abbildung 22.2 zeigt das zugehörige UML-Diagramm. Die Klasse Anbieter implementiert das Interface Schnittstelle-X. Bei der Vererbung stellt die abgeleitete Klasse die Schnittstelle der Oberklasse zur Verfügung. Insofern gibt es eine Ähnlichkeit, auch gekennzeichnet durch die gestrichelte Linie im Vergleich zum vorherigen Diagramm.

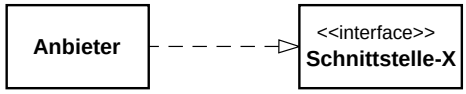


Abbildung 22.2: Interface-Anbieter

Die Ähnlichkeit wird in der Umsetzung nach C++ abgebildet: `Anbieter` wird von dem Interface `SchnittstelleX`[1] abgeleitet. Um klarzustellen, dass es um ein Interface geht, soll `SchnittstelleX` abstrakt sein. Das Datenobjekt `d` wird nicht als `const`-Referenz übergeben, weil `service()` damit auch die Ergebnisse an den Aufrufer übermittelt. Ein einfaches Programmbeispiel finden Sie im Verzeichnis *cppbuch/k22/interface*.

[1] Die UML erlaubt Bindestriche in Namen, C++ nicht.

Listing 22.2: Schnittstellenklasse

```
class SchnittstelleX {
public:
  virtual void service(Daten& d) = 0;       // abstrakte Klasse
  virtual ~SchnittstelleX() = default;      // virtueller Destruktor
  SchnittstelleX() = default;
  SchnittstelleX(const SchnittstelleX&) = default;
  SchnittstelleX& operator=(const SchnittstelleX&) = default;
};

class Anbieter : public SchnittstelleX {
public:
   void service(Daten& d) {
     // ... Implementation der Schnittstelle
   }
};
```

Interface nutzen

Bei der Nutzung des Interfaces bedient sich der Nutzer einer entsprechenden Methode des Anbieters. Die Abbildung 22.3 zeigt das zugehörige UML-Diagramm.

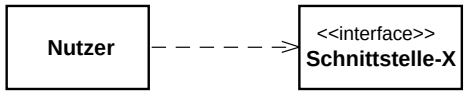


Abbildung 22.3: Interface-Nutzer

Ein `Nutzer` muss ein `Anbieter`-Objekt kennen, damit der Service genutzt werden kann. Aus diesem Grund wird in der folgenden Klasse bereits dem Konstruktor von `Nutzer` ein `Anbieter`-Objekt übergeben, und zwar per Referenz, nicht per Zeiger. Der Grund: Zeiger können NULL sein, aber undefinierte Referenzen gibt es nicht.

Listing 22.3: Nutzer der Schnittstelle

```
class Nutzer {
public:
  Nutzer(SchnittstelleX& a)
  : anbieter(a) {
     daten = ...
  }

  void nutzen() {
    anbieter.service(daten);
  }

private:
  Daten daten;
  SchnittstelleX& anbieter;
};
```

Warum wird die Referenz oben nicht als `const` übergeben? Das kann je nach Anwendungsfall sinnvoll sein oder auch nicht. Es hängt davon ab, ob sich der Zustand des `Anbieter`-Objekts durch den Aufruf der Funktion `service(daten)` ändert. Wenn ja, zum Beispiel durch interne Protokollierung der Aufrufe, entfällt `const`.

22.3 Assoziation

Eine Assoziation sagt zunächt einmal nur aus, dass zwei Klassen in einer Beziehung (mit Ausnahme der Vererbung) stehen. Die Art der Beziehung und zu wie vielen Objekten sie aufgebaut wird, kann variieren. In der Regel gelten Assoziationen während der Lebensdauer der beteiligten Objekte. Nur kurzzeitige Verbindungen werden meistens nicht notiert. Ein Beispiel für eine kurzzeitige Verbindung ist der Aufruf `anbieter.service(daten);` oben: `anbieter` kennt durch die Parameterübergabe das Objekt `daten`, wird aber vermutlich die Verbindung nach Ablauf der Funktion lösen.

Einfache gerichtete Assoziation

Das UML-Diagramm einer einfachen gerichteten Assoziation sehen Sie in der Abbildung 22.4.

Abbildung 22.4: Gerichtete Assoziation

Mit »gerichtet« ist gemeint, dass die Umkehrung nicht gilt, wie zum Beispiel die Beziehung »ist Vater von«. Falls zwar Klasse1 die Klasse2 kennt, aber nicht umgekehrt, wird dies durch ein kleines Kreuz bei Klasse1 vermerkt. Es kann natürlich sein, dass eine Beziehung zwischen zwei Objekten *derselben* Klasse besteht. Im UML-Diagramm führt dann der von einer Klasse ausgehende Pfeil auf dieselbe Klasse zurück. In C++ wird eine einfache gerichtete Assoziation durch ein Attribut `zeigerAufKlasse2` realisiert:

Listing 22.4: Gerichtete Assoziation: `Klasse1` kennt `Klasse2`

```
class Klasse1 {
public:
  Klasse1() {
  : zeigerAufKlasse2(nullptr) {
  }

  void setKlasse2(Klasse2* ptr2) {
    zeigerAufKlasse2 = ptr2;
  }
private:
  Klasse2* zeigerAufKlasse2;
};
```

Ein Zeiger ist hier besser als eine Referenz geeignet, weil es sein kann, dass das Kennenlernen erst nach dem Konstruktoraufruf geschieht.

Gerichtete Assoziation mit Multiplizität

Die Multiplizität, auch Kardinalität genannt, gibt an, zu wie vielen Objekten eine Verbindung aufgebaut werden kann. In Abbildung 22.5 bedeutet die 1, dass jedes Objekt der Klasse2 zu genau einem Objekt der Klasse1 gehört. * bei Klasse2 besagt, dass einem Objekt der Klasse1 beliebig viele (auch 0) Objekte der Klasse2 zugeordnet sind.

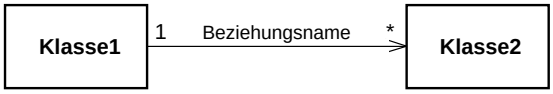


Abbildung 22.5: Gerichtete Assoziation mit Multiplizitäten

Im folgenden C++-Beispiel entspricht `Fan` der Klasse1 und `Popstar` der Klasse2. Ein Fan kennt N Popstars. Die Beziehung ist also »kennt«. Der Popstar hingegen kennt seine Fans im Allgemeinen nicht. Um die Multiplizität auszudrücken, bietet sich ein `vector` an, der Verweise auf `Popstar`-Objekte speichert. Wenn die Verweise eindeutig sein sollen, ist ein `set` die bessere Wahl.

Listing 22.5: Gerichtete Assoziation mit Muliplizität: Ein `Fan` kennt `Popstar`s, aber nicht umgekehrt.

```
class Fan {
public:
  void werdeFanVon(Popstar* star) {
    meineStars.insert(star);                // zu insert() siehe Seite 841
  }

  void denKannsteVergessen(Popstar* star) {
    meineStars.erase(star);                 // Rückgabewert ignoriert
  }
  // Rest weggelassen

private:
  std::set<Popstar*> meineStars;
};
```

Die Objekte als Kopie abzulegen, also `Popstar` als Typ für den Set statt `Popstar*` zu nehmen, hat Nachteile. Erstens ist es wenig sinnvoll, die Kopie zu erzeugen, wenn es doch das Original gibt, und zweitens kostet es Speicherplatz und Laufzeit. Es gibt nur einen Vorteil: Es könnte ja sein, dass es das originale `Popstar`-Objekt nicht mehr gibt, zum Beispiel durch ein `delete` irgendwo. Ein noch existierender Zeiger wäre danach auf eine undefinierte Speicherstelle gerichtet. Eine noch existierende Kopie könnte als Wiedergänger auftreten.

Einfache ungerichtete Assoziation

Eine ungerichtete Assoziation wirkt in beiden Richtungen und heißt deswegen auch bidirektionale Assoziation. Die Abbildung 22.6 zeigt das UML-Diagramm.