HANSER

Leseprobe

zu

„Der C++-Programmierer“

(3. Auflage)

von Ulrich Breymann

ISBN (Buch): 978-3-446-43894-1
ISBN (E-Book): 978-3-446-43953-5

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-43894-1
sowie im Buchhandel

1.3 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. Deshalb sollten Sie die Dienste des Compilers möglichst bald anhand der Beispiele nutzen – 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 die Benutzung der in Abschnitt 1.5 beschriebenen Entwicklungsumgebungen an. Ein viel verwendeter Compiler ist der GNU[1] C++-Compiler [GCC]. Entwicklungsumgebung und Compiler sind kostenlos erhältlich. Ein Installationsprogramm dafür finden Sie auf der Website zum Buch [SW].

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 später 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:

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

[1] Siehe Glossar Seite 960

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 nachfolgendem 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 auf der nächsten Seite zu sehen ist. 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 später (Seite 36).

`#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. Man kann 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 3.3.5.

`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 59 und 143).

Listing 1.1: 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 << endl;
    return 0;
}
```

`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 könnten verwendet werden, um über das Betriebssystem einem nachfolgenden Programm einen Fehler zu 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 `summand1` 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 später).

`cout` — Ausgabe: `cout` (Abkürzung für *character 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"!*.
`endl`	bewirkt die sofortige Ausgabe einer neuen Zeile.
`return 0;`	Unser Programm läuft einwandfrei; es gibt daher 0 zurück. Diese Anweisung kann 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`, `summand1` 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 später im Programm versehentlich falsch geschrieben wird, z. B. `sume = summand1 + summand2;` im Programm auf Seite 33, 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).

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.

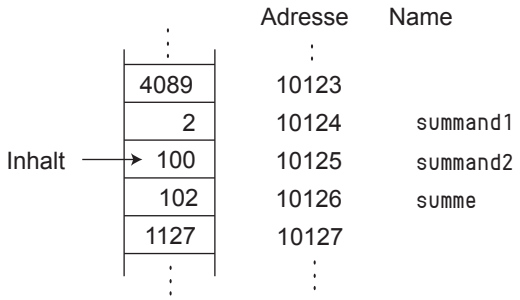


Abbildung 1.1: Speicherbereiche mit Adressen

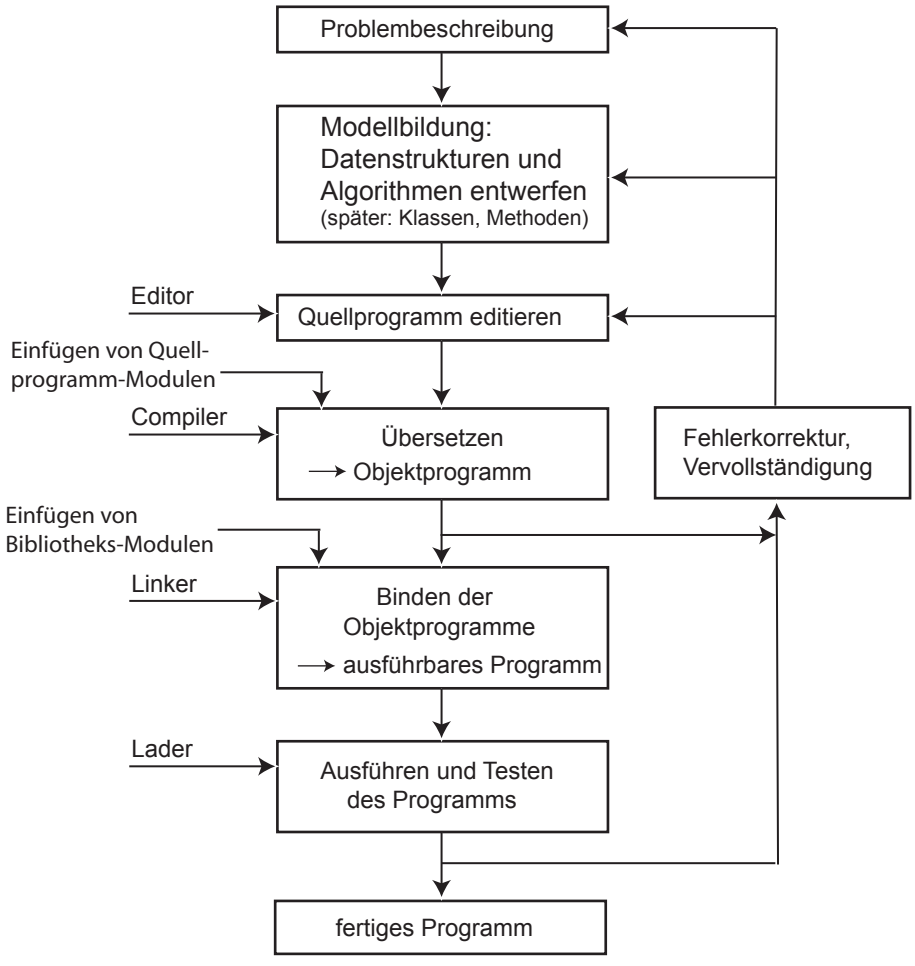


Abbildung 1.2: Erzeugung eines lauffähigen Programms

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 und anschließend das Programm binden oder linken (eine Erklärung folgt bald) und ausführen. Ein Programmtext wird auch »Quelltext« (englisch *source code*) genannt.

Der Compiler erzeugt 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 3.3 erläutert.

Wie bekomme ich ein Programm zum Laufen?

Der erste Schritt ist das Schreiben mit einem Textsystem, Editor genannt. Der Text sollte keine Sonderzeichen zur Formatierung enthalten, weswegen nicht alle Editoren geeignet sind. Integrierte Entwicklungsumgebungen (englisch *Integrated Development Environment, IDE*) haben einen speziell auf Programmierzwecke zugeschnittenen Editor, der darüber hinaus auf Tastendruck oder Mausklick die Übersetzung anstößt. Alternativ besteht die Möglichkeit, Compiler und Linker per Kommandozeile in einer Linux-Shell oder einem Windows-Eingabeaufforderungs-Fenster zu starten. Beispiel für das Programm *summe.cpp* und den Open Source C++-Compiler g++[GCC]:

```
g++ -c summe.cpp        compilieren (summe.o wird erzeugt)
                        (bzw. summe.obj bei anderen Compilern)
g++ -o summe.exe summe.o      linken. Beide Schritte lassen sich zusammenfassen:
g++ -o summe.exe summe.cpp
```

Das Programm wird durch Eintippen von *summe.exe* gestartet. Dabei wird vorausgesetzt, dass der g++-Compiler im Pfad ist – sonst wird er nicht gefunden. Wie man den Pfad um ein Verzeichnis erweitert, lesen Sie im Anhang, Abschnitt A.6. Weitere Einzelheiten zur Bedienung von Compiler und Linker finden Sie in Abschnitt A.3 auf Seite 900 oder in den Hilfedateien Ihres C++-Systems. Der folgenden Abschnitt 1.5 zeigt Installation und Bedienung einer kostenlosen[2] Integrierten Entwicklungsumgebung.

1.4.1 Namenskonventionen

Funktions-, Variablen- und andere Namen unterliegen der folgenden Konvention:

[2] Bitte Lizenzbestimmungen beachten!

- Ein Name ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und Unterstrich (_). Er sollte kurz sein, aber die Bedeutung widerspiegeln. So ist der Name `summe` klarer als ein Name `s`.
- Ein Name beginnt stets mit einem Buchstaben oder einem Unterstrich _. Am Anfang eines Namens sollten Unterstriche vermieden werden, ebenso Namen, die zwei Unterstriche direkt nacheinander enthalten. Solche Namen werden systemintern benutzt.
- Selbsterfundene Namen dürfen nicht mit den vordefinierten Schlüsselwörtern übereinstimmen (zum Beispiel `for`, `int`, `main` ...). Eine Tabelle der Schlüsselwörter ist im Anhang auf Seite 899 zu finden.
- Ein Name kann prinzipiell beliebig lang sein. In den Compilern ist die Länge jedoch begrenzt, zum Beispiel auf 31 oder 255 Zeichen.

Die hier aufgelisteten Konventionen zeigen die Regeln für die *Struktur* eines Namens, auch *Syntax* oder *Grammatik* genannt. Ein Name darf niemals ein Leerzeichen enthalten! Wenn eine Worttrennung aus Gründen der Lesbarkeit gewünscht ist, kann man den Unterstrich oder einen Wechsel in der Groß- und Kleinschreibung benutzen. Beispiele:

`int 1_Zeile;`	falsch! (Ziffer am Anfang)
`int anzahl der Zeilen;`	falsch! (Name enthält Leerzeichen)
`int AnzahlDerZeilen;`	richtig! andere Möglichkeit:
`int anzahl_der_Zeilen;`	richtig!

Zur Abkürzung können Variablen des gleichen Datentyps aufgelistet werden, sofern sie durch Kommas getrennt werden. `int a; int b; int c;` ist gleichwertig mit `int a,b,c;`. Lesbarer ist jedoch die Deklaration auf drei getrennten Zeilen.

1.5 Integrierte Entwicklungsumgebung

Im Folgenden stelle ich Ihnen kurz die Integrierte Entwicklungsumgebung (abgekürzt IDE) Code::Blocks [CB] vor. Sie gibt es für Windows und Linux. Natürlich gibt es noch einige andere, aber Anfängern empfehle ich Code::Blocks, weil diese IDE am einfachsten zu installieren und zu bedienen ist.

Weil hier nur sehr kurz auf die Bedienung eingegangen werden kann, empfehle ich Ihnen die Bedienungsanleitung von Code::Blocks, die Sie auf der Internetseite [CB] finden.

Die Installationsanleitung für die Software zu diesem Buch, die auch Code::Blocks enthält, finden Sie in Abschnitt A.6 (Seite 950) für Windows. Die Installationsanleitung für Linux finden Sie in Abschnitt A.7.3 (Seite 954).

Programm eingeben, übersetzen und starten

Um jetzt ein Programm einzugeben, starten Sie Code::Blocks (Windows: Start →Alle Programme →CodeBlocks). Nach dem Start legen Sie ein neues Projekt an, indem Sie auf »Create a new Project« im Startfenster klicken. Alternativ ist der Weg über den Me-

nübalken möglich (File →New →Project). Es wird Ihnen eine Auswahl verschiedenster Anwendungstypen angeboten. Fürs Erste wählen Sie bitte die einfachste Art der Anwendung, die »Console Application«. Im dann erscheinenden Fenster müssen noch einige Angaben eingetragen werden. Vorschlag:

Project title: *Projekt1*
Folder to create project in: *cpp_projekt*

Die anderen Einstellungen werden belassen. Nun »Next« und »Finish« anklicken. Im linken Teil klicken Sie bitte auf Sources und dort auf *main.cpp*. Bitte ändern Sie das angezeigte Programm so, dass Sie damit die Summe zweier Zahlen berechnen können (Programm von Seite 33). Um einen Fehler zu provozieren und seine Reparatur zu zeigen, wird noch eine nichtexistierende Variable x addiert. Ein Klick auf das Diskettensymbol oben sichert die Datei.

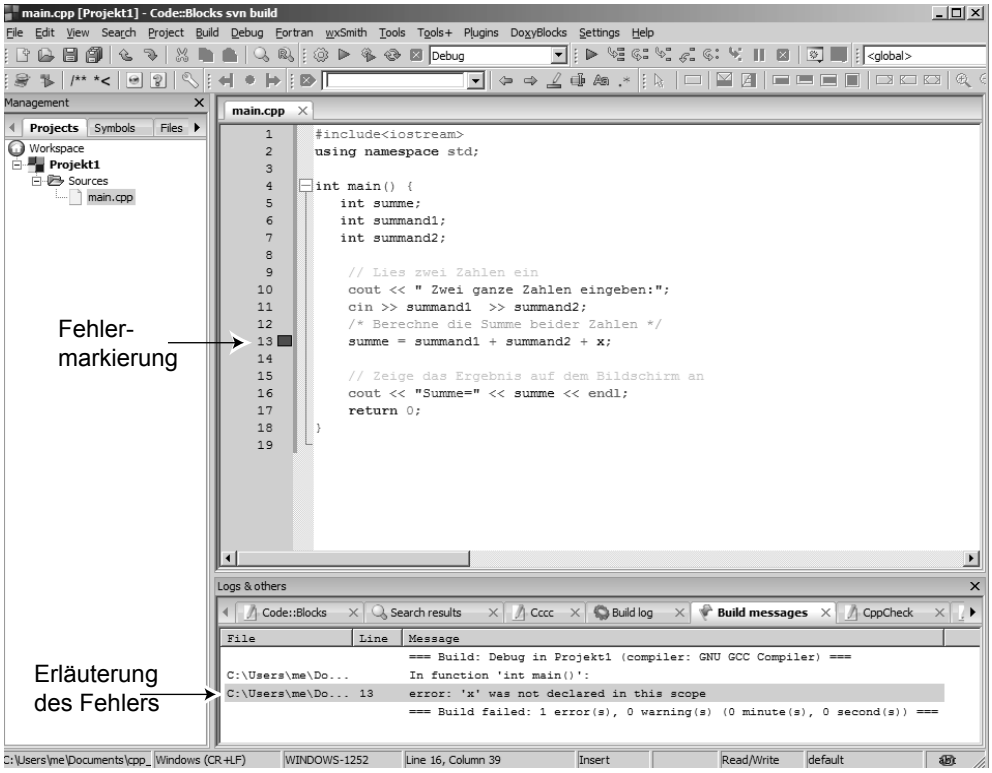


Abbildung 1.3: Die Entwicklungsumgebung Code::Blocks zeigt einen Fehler an.

Ein Klick auf das Build-Icon oder Drücken der Tastenkombination Strg-F9 startet den Übersetzungsprozess. Der mit Absicht erzeugte Fehler wird nun im Programm mit einem roten Balken markiert; Erklärungen dazu finden sich im Fenster unter dem Programmcode (siehe Abbildung 1.3). Es empfiehlt sich, die Hinweise auf Fehler genau zu lesen!

Im Bild und in diesem Buch wird übrigens für die Positionierung der geschweiften Klammern der Kernighan & Ritchie[3]-Stil gewählt (Settings →Editor, links unten »Source formatter« anklicken und dann rechts K&R wählen).

Jetzt korrigieren Sie bitte das Programm, indem die fehlerhafte Addition der Variablen `x` entfernt wird. Ein weiterer Klick auf das Build-Icon wird nun von Erfolg gekrönt: 0 errors, 0 warnings! Ein Klick auf das Run-Icon (oder die Tastenkombination Strg-F10) führt zur Ausführung des Programms. Im erscheinenden Fenster geben Sie einfach zwei Zahlen ein und drücken ENTER. Das Ergebnis wird dann angezeigt. Mit einem weiteren Tastendruck wird das Fenster geschlossen. Man kann eine weitere Pause erzwingen, falls das Fenster sich vorher zu schnell schließt. Dazu werden am Programmende die Zeilen

```
cin.ignore(1000, '\n');                    // genaue Erklärung folgt in Kap. 10
cin.get();
```

hinzugefügt, wie in der Datei *cppbuch/k1/summe.cpp* gezeigt. Die Datei ist in den herunterladbaren Beispielen enthalten [SW].

Tipp

Man kann das Programm direkt aus einem bestehenden Fenster, das dann weiter bestehen bleibt, starten, hier gezeigt für das Betriebssystem Windows: Dazu rufen Sie in Windows Start →Alle Programme →Zubehör →Eingabeaufforderung auf und gehen durch Anwendung des Befehls `cd` in das Verzeichnis, in dem das Programm abgelegt worden ist, zum Beispiel *cpp_projekt/Projekt1/bin/Debug*. In diesem Verzeichnis befindet sich die vom Compiler erzeugte ausführbare Datei *Projekt1.exe*, wie der Befehl `dir` zeigt. Wenn Sie nun `Projekt1.exe` eintippen, wird das Programm ausgeführt.

1.6 Einfache Datentypen und Operatoren

Sie haben schon flüchtig die Datentypen `int` für ganze Zahlen und `float` für Gleitkommazahlen kennengelernt. Es gibt darüber hinaus noch eine Menge anderer Datentypen. Hier wird näher auf die *Grunddatentypen* eingegangen. Sie sind definiert durch ihren Wertebereich sowie die mit diesen Werten möglichen Operationen. Nicht-veränderliche Daten sind für alle Grunddatentypen möglich; sie werden in Abschnitt 1.6.4 erläutert.

1.6.1 Ausdruck

Ein Ausdruck besteht aus einem oder mehreren Operanden, die miteinander durch Operatoren verknüpft sind. Die Auswertung eines Ausdrucks resultiert in einem Wert, der an die Stelle des Ausdrucks tritt. Der einfachste Ausdruck besteht aus einer einzigen Kon-

[3] Ritchie hat unter der Mitwirkung von Kernighan die Programmiersprache C entwickelt.

```
    default     : ziffer = 0;
  }
  if (ziffer > 0) {
    cout << "Ziffer = " << ziffer;
  }
  else {
    cout <<"keine römische Ziffer!";
  }
  cout << endl;
}
```

Wenn eine `case`-Konstante mit der `switch`-Variablen übereinstimmt, werden *alle nachfolgenden Anweisungen bis zum ersten* `break` ausgeführt. Mit einem fehlenden `break` und einer fehlenden Anweisung nach einer `case`-Konstante lässt sich eine ODER-Verknüpfung realisieren. Bei der Umwandlung römischer Ziffern lässt sich damit die Auswertung von Kleinbuchstaben bewerkstelligen:

```
switch(zeichen) {
   case 'i'     :
   case 'I'     : ziffer = 1;  break;
   case 'v' : case 'V' : ziffer = 5; break;    // andere Schreibweise
   // Rest weggelassen
```

Ein fehlendes `break` sollte kommentiert werden, sofern sich die Absicht nicht klar aus dem Programm ergibt. Alle interessierenden `case`-Konstanten müssen einzeln aufgelistet werden. Es ist in C++ *nicht* möglich, *Bereiche* anzugeben, etwa der Art `case 7..11 : Anweisung break;` anstelle der länglichen Liste `case 7: case 8: case 9: case 10: case 11: Anweisung break;`. In solchen Fällen können Vergleiche aus der `switch`-Anweisung herausgenommen werden, um sie als Abfrage `if(ausdruck >= startwert && ausdruck <= endwert)...` zu realisieren.

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 62 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 den Block innerhalb der Schleife nennt man *Schleifenkörper*. Schleifen können wie `if`-Anweisungen beliebig geschachtelt werden.

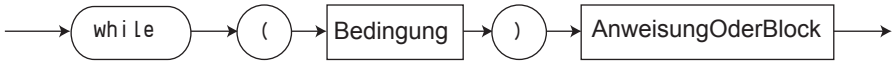
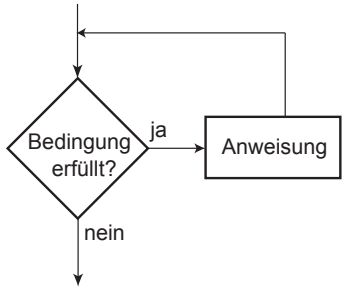


Abbildung 1.7: Syntaxdiagramm einer `while`-Schleife

```
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 1.10: Beispiel für `while`-Schleife

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

int main() {
   unsigned int x, 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 << endl;
}
```

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

Falls der Schleifenkörper aus vielen Anweisungen besteht, sollten die Anweisungen zur Veränderung der Bedingung an den Schluss 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 62 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

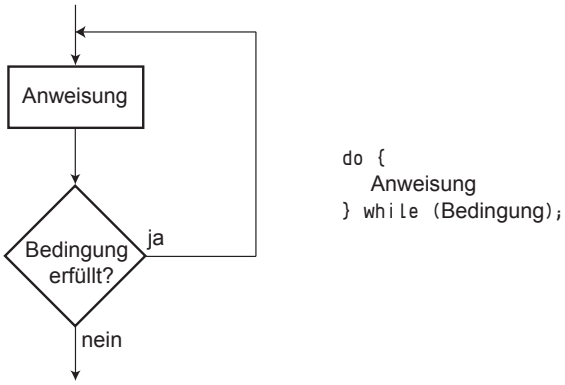


Abbildung 1.10: Flussdiagramm für eine `do while`-Anweisung

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, auch bei längeren Programmen.

```
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 1.11: Beispiel für `do while`-Schleife

```
// 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 << endl;
}
```

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.

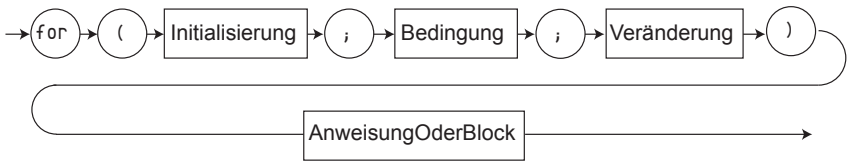


Abbildung 1.11: Syntaxdiagramm einer `for`-Schleife

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) << endl;
}
```

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, unten im Beispielprogramm verwendete 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 1.12: Beispiel für `for`-Schleife

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

int main() {
   cout << "Fakultät berechnen. Zahl >= 0? :";
   int n;
   cin >> n;
   unsigned long fak {1L};
   for(int i = 2; i <= n; ++i) {
      fak *= i;
   }
   cout << n << "! =   " << fak << endl;
}
```

Um Fehler zu vermeiden und zur besseren Verständlichkeit sollte man niemals Laufvariablen in der Anweisung verändern. 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.

4.7 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 kommen kann. Es wird jedoch anhand eines Beispiels ein erster Eindruck vermittelt, wie man von einer Problemstellung zum objektorientierten Programm kommen kann.

Es geht hier um ein Programm, das zu einer gegebenen Personalnummer den Namen heraussucht[4]. 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.

[4] Ähnlichkeiten mit der Aufgabe 2.5 von Seite 100 sind weder zufällig noch unbeabsichtigt.

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

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

Diese drei Punkte und die Kenntnis der Datei führen zu entsprechenden Schlussfolgerungen. Dabei kann man im ersten Schritt die Substantive (Hauptworte) als Kandidaten für Klassen 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` sollte 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 sollte nicht als `int` vorliegen, sondern als `string`, damit nicht führende Nullen (siehe Datei oben) beim Einlesen verschluckt werden oder zu einer Interpretation als Oktalzahl führen. Außerdem könnte es Nummernsysteme mit Buchstaben und Zahlen geben.
   Die Klasse `Personalverwaltung` soll die Daten speichern. Dafür bietet sich ein `vector<Person>` als Attribut an.

2. Das Tätigkeitswort *ausgeben* legt nahe, eine gleichnamige Methode `ausgeben()` vorzusehen. In der Methode werden Name und Personalnummer einer Person ausgegeben. Es muss also entsprechende Methoden in der Klasse `Person` geben, etwa `getName()` und `getPersonalnummer()`. Diese Methoden würden innerhalb der Funktion `ausgeben()` aufgerufen werden.
3. *Dialog führen* legt nahe, eine Methode `dialogfuehren()` oder kurz `dialog()` vorzusehen.

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

```
// cppbuch/k4/personalverwaltung/main.cpp
#include "personalverwaltung.h"
#include<iostream>
using namespace std;

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

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

Die Klasse `Person` ist einfach zu entwerfen:

```
// cppbuch/k4/personalverwaltung/person.h
#ifndef PERSON_H
#define PERSON_H
#include<string>

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

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

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

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

```
// cppbuch/k4/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 2.5 von Seite 100 gelöst oder deren Lösung auf Seite 911 nachgesehen haben.

Übungen

4.5 Implementieren Sie die oben deklarierten Methoden der Klasse `Personalverwaltung` in einer Datei *personalverwaltung.cpp*.

4.6 Wie können Sie mit C++ erreichen, dass ein Attribut *direkt*, also ohne Einsatz einer Methode, zwar gelesen, aber nicht verändert werden kann? Beispiel:

```
int main() {
    MeineKlasse objekt;

    // Fehler! nicht erlaubte Aktion:
    objekt.readonlyAttribut = 999;

    // erlaubter direkter lesender Zugriff:
    cout << "objekt.readonlyAttribut="
         << objekt.readonlyAttribut << endl;    // ok
}
```

Wie sieht die Realisierung in der Klasse `MeineKlasse` aus?

4.7 Schreiben Sie auf Basis der Lösung der Taschenrechner-Aufgabe von Seite 134 eine Klasse `Taschenrechner`, die einen eingegebenen String verarbeitet. Die Anwendung könnte wie folgt aussehen:

```
#include"taschenrechner.h"
#include<iostream>
using namespace std;
```

Der Fehler liegt in dem undefinierten Wert von `pa` nach der Löschoperation. Falls ein Zeiger nach dem Löschen noch verwendet werden kann, sollte man ihn direkt nach dem `delete` mit `pa = nullptr;` auf Null setzen. 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 `shared_ptr`, siehe folgenden Abschnitt.

20.2.18 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 Beispielklasse `MeinString` (Seite 233) oder die Verwendung der »Smart Pointer« (`shared_ptr`), siehe unten. Ein weiterer Vorteil ist die korrekte Speicherfreigabe bei Exceptions (siehe unten).

20.2.19 Programmierrichtlinien einhalten

Das Einhalten von Programmierrichtlinien unterstützt das Schreiben gut lesbarer Programme. Es gibt einige dieser Richtlinien, die sich in großen Teilen ähneln. Deshalb sei hier nur auf die GNU C++ Language Conventions [CConv] hingewiesen. Ein einfaches Beispiel für solche Regeln sind Vorschriften für die Schreibweise, etwa:

- Die Namen von eigenen Klassen sollen stets mit einem Großbuchstaben beginnen (im Gegensatz zu denen der C++-Standardbibliothek).
- Die Namen von Variablen und Funktionen beginnen mit einem Kleinbuchstaben.
- Konstantennamen sind vollständig groß zu schreiben, z.B. `FAKTOR`.
- Worttrennungen sind durch Wechsel in der Groß-/Kleinschreibung oder durch einen Unterstrich zu kennzeichnen, z.B. `anzahlDerObjekte` oder `anzahl_der_objekte`.

20.3 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.

20.3.1 Sichere Verwendung von shared_ptr

Bei der Konstruktion eines `shared_ptr` (Beschreibung Seite 867) 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. 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).

20.3.2 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 (siehe Seite 867). Dies kann zu einem Speicherleck führen, wenn der Zeiger mit `new []` erzeugt wurde, wie auf Seite 203 beschrieben. Hier ein Beispiel:

```
int* p = new int[10];
// p verwenden
// delete p; falsch!
delete [] p;                                 // richtig
```

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 besteht in der Übergabe eines `deleter`-Objekts an den `shared_ptr`. Wenn es so ein Funktionsobjekt gibt, wird dessen `operator()()` aufgerufen.

```
// Auszug aus cppbuch/k33/arrayshared.cpp
template<typename T>
struct ArrayDeleter {
   void operator()(T* ptr) {
      delete [] ptr;
   }
};
```

```
void funktion() {
   shared_ptr<int> p(new int[10], ArrayDeleter<int>()); // richtig
   // ... etliche Zeilen weggelassen
}                                                  // kein Memory-Leak, Array wird korrekt gelöscht
```

20.3.3 unique_ptr für Arrays korrekt verwenden

Bei `unique_ptr`-Objekten tritt dieselbe Problematik auf, wie oben in Abschnitt 20.3.1 beschrieben. Die Schnittstelle ist etwas anders:

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

Der Typ des für die Löschung zuständigen Objekts gehört zur Schnittstelle. 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 20.1: `unique_ptr` und Array

```
// cppbuch/k33/arrayunique.cpp
#include<memory>

void f() {
   std::unique_ptr<int[]> arr(new int[10]); // int[] statt int
   // 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(new int[10]);
```

20.3.4 Exception-sichere 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 wird 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:

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

Aus diesem Grund sollten ausschließlich Stack-Objekte (automatische Objekte) verwendet werden, wenn Exceptions auftreten. Dies ist immer möglich, wenn man Beschaffung und Freigabe eines dynamischen Objekts innerhalb eines Stack-Objekts versteckt. Das Hilfsmittel dazu kennen wir bereits, nämlich die »intelligenten« Zeiger aus Abschnitt 9.5:

```
void func() {
    // shared_ptr der Standardbibliothek, siehe Abschnitt 9.5.1. Header: <memory>
    std::shared_ptr<Datum> pDshared(new 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.

20.3.5 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.

Das folgende Beispiel demonstriert dieses Verhalten. Ein Objekt der Klasse `Ganzes` enthält zwei Subobjekte der Typen `Teil1` und `Teil2`, die wie folgt definiert sind. Beachten Sie, dass der Konstruktor von `Teil2` zur Demonstration eine Exception wirft! Die Klasse `Ganzes` folgt im Anschluss.

Listing 20.2: Klassen `Teil1` und `Teil2`

```
// cppbuch/k20/teil.h
#ifndef TEIL_H
#define TEIL_H
#include<iostream>
```

```
class Teil1 {
public:
   Teil1(int x)
      : attr(x) {
   }
   ~Teil1() {
      std::cout << "Teil1::Destruktor gerufen!" << std::endl;
   }
private:
   int attr;
};

class Teil2 {           // Konstruktor wirft zur Demonstration eine Exception
public:
   Teil2() {
      throw std::exception(); // auskommentieren:
      // dann wird der Destruktor gerufen, ansonsten NICHT!
   }
   ~Teil2() {
      std::cout << "Teil2::Destruktor gerufen!" << std::endl;
   }
};
#endif
```

Bei der Konstruktion des Objektes `ganzes` (siehe Beispiel unten) wird das Subobjekt `teil1` initialisiert. Die Konstruktion des Subobjekts vom Typ `Teil2`, die mit `new` versucht wird, schlägt jedoch fehl, weil der `Teil2`-Konstruktor eine Exception wirft.

Listing 20.3: Klasse `Ganzes`

```
// cppbuch/k20/ganzes.h
#ifndef GANZES_H
#define GANZES_H
#include<iostream>
#include"teil.h"

class Ganzes {
public:
   Ganzes() : teil1(99) {
      ptr = new Teil2;
   }
   ~Ganzes() {
      std::cout << "Ganzes::Destruktor gerufen!" << std::endl;
      delete ptr;
   }
private:
   Teil1 teil1;
   Teil2* ptr;
   Ganzes(const Ganzes&) = delete;                // für Beispiel nicht erforderlich
   Ganzes& operator=(const Ganzes&) = delete;     // für Beispiel nicht erforderlich
};
#endif
```

Listing 20.4: Exception im Konstruktor

```
// cppbuch/k20/excImKonstruktor1.cpp
#include"ganzes.h"

int main() {
   try {
      Ganzes ganzes;
   }
   catch(const std::exception& e) {
      std::cout << "Exception gefangen: " << e.what() << std::endl;
   }
}
```

Weil nur das Subobjekt `teil1` vollständig konstruiert wird, kommt auch nur dessen Destruktor zum Tragen. Die anderen Destruktoren werden nicht aufgerufen. Wird nun die Anweisung `throw exception();` auskommentiert oder gelöscht, werden alle Destruktoren aufgerufen. Die am Anfang dieses Abschnitts genannten Punkte werden in diesem Beispiel bei Auftreten der Exception erreicht: Die »Ressource« `teil1` wird freigegeben, und die Exception wird in `main()` aufgefangen.

Ein Gegenbeispiel: Wenn die Klasse `Ganzes` *beide* Sub-Objekte mit `new` erzeugen würde, aber so, dass erst die Erzeugung des zweiten Sub-Objekts eine Exception wirft, wird der Destruktor für das erste Sub-Objekt *nicht* aufgerufen. Der Speicherplatz wird nicht wieder freigegeben.

Listing 20.5: Fehlerhafter Konstruktor

```
// Auszug aus cppbuch/k20/ganzesMitFehler.h
class GanzesMitFehler {
public:
   GanzesMitFehler() : ptr1(new Teil1(99)), ptr2(new Teil2) {
   }
   ~GanzesMitFehler() {
      std::cout << "GanzesMitFehler::Destruktor gerufen!" << std::endl;
      delete ptr1;
      delete ptr2;
   }
private:
   Teil1* ptr1;
   Teil2* ptr2;
   // Kopierkonstruktor und Zuweisungsoperator weggelassen
};
```

Mit `shared_ptr` wie in Abschnitt 20.3.1 geht man sicher. Wenn der `Teil2`-Konstruktor eine Exception wirft, wird der Destruktor von `Teil1` aufgerufen und gibt den Speicherplatz frei:

Listing 20.6: Konstruktor mit `shared_ptr`

```
// Auszug aus cppbuch/k20/ganzesKorrigiert.h
class GanzesKorrigiert {
public:
   GanzesKorrigiert() : ptr1(new Teil1(99)), ptr2(new Teil2) {
   }
```

```
  ~GanzesKorrigiert() {
     std::cout << "GanzesKorrigiert::Destruktor gerufen!" << std::endl;
      // delete nicht notwendig wegen shared_ptr
  }
private:
  std::shared_ptr<Teil1> ptr1;
  std::shared_ptr<Teil2> ptr2;
};
```

20.3.6 Exception-sichere Zuweisung

Wenn bei einer Kopie Speicher beschafft werden muss, sollte man das zuerst tun! Der Grund: Falls es dabei eine Exception geben sollte, würden alle nachfolgenden, 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 mögliche Lösung der Aufgabe 9.6 von Seite 337 an:

```
// Exception-sicher
MeinString& MeinString::operator=(const MeinString& m) { // Zuweisung
    char *p = new char[m.len+1];        // zuerst neuen Platz beschaffen
    strcpy(p, m.start);                 // kopieren
    delete [] start;                    // alten Platz freigeben
    len   = m.len;                      // Verwaltungsinformation aktualisieren
    start = p;
    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 `p` nicht:

```
// NICHT Exception-sicher!
MeinString& MeinString::operator=(const MeinString& m) { // Zuweisung
    delete [] start;                    // weg damit, es wird schon gutgehen!
    start = new char[m.len+1];          // neuen Platz beschaffen
    strcpy(start, m.start);
    len   = m.len;                      // 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 `&m == this` sein könnte, will ich gar nicht erst reden. Eine andere Möglichkeit, die Zuweisung exception-sicher zu gestalten, ist ein »swap-Trick«. Dazu wird eine temporäre Kopie erzeugt, und anschließend werden die Verwaltungsdaten vertauscht. Danach hat `*this` die richtigen Daten, und das temporäre Objekt wird korrekt vom Destruktor zerstört:

```
// Exception-sicher
MeinString& MeinString::operator=(MeinString temp) { // Zuweisung
        // temporäre Kopie durch Übergabe per Wert
```

21

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 [Oe]. 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, wie man die Umsetzung gestalten kann. Im Einzelfall kann eine Variation sinnvoll sein.

21.1 Vererbung

Über Vererbung ist in diesem Buch schon einiges gesagt worden, das hier nicht wiederholt werden muss. Die Abbildung 21.1 zeigt das zugehörige UML-Diagramm.

Abbildung 21.1: Vererbung

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:

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

21.2 Interface anbieten und nutzen

Interface anbieten

Die Abbildung 21.2 zeigt das zugehörige UML-Diagramm. Die Klasse Anbieter implementiert das Interface Schnittstelle-X. Bei der Vererbung stellt die abgeleitete Klasse die Schnittstellen der Oberklasse zur Verfügung. Insofern gibt es eine Ähnlichkeit, auch gekennzeichnet durch die gestrichelte Linie im Vergleich zum vorherigen Diagramm.

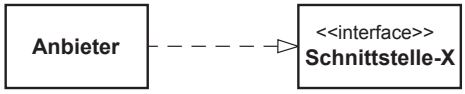


Abbildung 21.2: Interface-Anbieter

Die Ähnlichkeit wird in der Umsetzung nach C++ abgebildet: `Anbieter` wird von `SchnittstelleX`[1] abgeleitet. Um klarzustellen, dass um ein Interface geht, sollte `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/k21/interface*.

```
class SchnittstelleX {
public:
```

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

```
   virtual void service(Daten& d) = 0;  // abstrakte Klasse
};

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 21.3 zeigt das zugehörige UML-Diagramm.

Abbildung 21.3: 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.

```
class Nutzer {
public:
   Nutzer(SchnittstelleX& a)
   : anbieter(a) {
      daten = ...
   }
   void nutzen() {
      anbieter.service(daten);
   }
private:
   Daten daten;
   SchnittstelleX& anbieter;
};
```

Nun kann man sich fragen, warum die Referenz oben nicht als `const` übergeben wird. 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`.

21.3 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 Abbildung 21.4.

Abbildung 21.4: 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:

```
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 21.5 bedeutet die 1, dass jedes Objekt der Klasse2 zu genau einem Objekt der Klasse1 gehört. Das Sternchen * bei Klasse2 besagt,

dass einem Objekt der Klasse1 beliebig viele (auch 0) Objekte der Klasse2 zugeordnet sind.

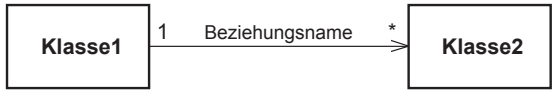


Abbildung 21.5: Gerichtete Assoziation mit Multiplizitäten

Im folgenden C++-Beispiel entspricht `Fan` der Klasse1 und `Popstar` der Klasse2. Eine 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.

```
class Fan {
public:
   void werdeFanVon(Popstar* star) {
      meineStars.insert(star);          // zu set::insert() siehe Seite 803

   }
   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 21.6 zeigt das UML-Diagramm.

Abbildung 21.6: Ungerichtete Assoziation

Wenn zwei sich kennenlernen, kann das mit einer ungerichteten Assoziation modelliert werden. Zur Abwechslung sei die Umsetzung in C++ nicht mit zwei, sondern nur mit

einer Klasse (namens `Person`) gezeigt. Das heißt, die Klasse hat eine Beziehung zu sich selbst, siehe Abbildung 21.7. Solche Assoziationen werden auch rekursiv genannt und dienen zur Darstellung der Beziehung verschiedener Objekte derselben Klasse.

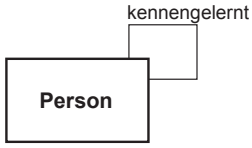


Abbildung 21.7: Rekursive Assoziation

Die Umsetzung in C++ wird am Beispiel von Personen gezeigt, die sich gegenseitig kennenlernen. Ein Aufruf `A.lerntkennen(B);` impliziert, dass `B` auch `A` kennenlernt:

Listing 21.1: Assoziation: Personen lernen sich kennen

```
// cppbuch/k21/bidirektionaleAssoziation/main.cpp
#include "Person.h"

int main() {
   Person mabuse("Dr. Mabuse");
   Person klicko("Witwe Klicko");
   Person holle("Frau Holle");
   mabuse.lerntkennen(klicko);
   holle.lerntkennen(klicko);
   // ...
}
```

Die entscheidende Methode der Klasse `Person` ist `lerntkennen(Person& p)` (siehe unten). Beim Eintrag in die Menge der Bekannten wird festgestellt, ob der Eintrag vorher schon vorhanden war. Wenn nicht, wird er auch auf der Gegenseite vorgenommen.

Listing 21.2: Klasse `Person`

```
// cppbuch/k21/bidirektionaleAssoziation/Person.h
#ifndef PERSON_H
#define PERSON_H
#include<iostream>
#include<set>
#include<string>

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

   virtual ~Person() = default;

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

```
   void lerntkennen(Person& p) {
     bool nichtvorhanden = bekannte.insert(p.getName()).second;
      if (nichtvorhanden) {              // falls unbekannt, auch bei p eintragen
        p.lerntkennen(*this);
      }
   }

   void bekannteZeigen() const {
      std::cout << "Die Bekannten von " << getName() << " sind:" << std::endl;
      for (auto bekannt : bekannte) {
         std::cout << bekannt << std::endl;
      }
   }
private:
   std::string name;
   std::set<std::string> bekannte;
};
#endif
```

21.3.1 Aggregation

Die »Teil-Ganzes«-Beziehung (englisch *part of*) wird auch *Aggregation* genannt. Sie besagt, dass ein Objekt aus mehreren Teilen besteht (die wiederum aus Teilen bestehen können). Die Abbildung 21.8 zeigt das UML-Diagramm. Die Struktur entspricht der gerichteten Assoziation, sodass deren Umsetzung in C++ hier Anwendung finden kann. Ein Teil kann für sich allein bestehen, also auch vom Ganzen gelöst werden. Letzteres geschieht in C++ durch Nullsetzen des entsprechenden Zeigers.

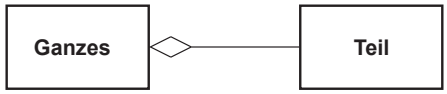


Abbildung 21.8: Aggregation

21.3.2 Komposition

Die Komposition ist eine spezielle Art der Aggregation, bei der die Existenz der Teile vom Ganzen abhängt. Damit ist gemeint, dass die Teile zusammen mit dem Ganzen erzeugt und auch wieder vernichtet werden. Ein Teil ist somit stets genau einem Ganzen zugeordnet; die Multiplizität kann also nur 1 sein. Die Abbildung 21.9 zeigt das UML-Diagramm.

Abbildung 21.9: Komposition

Es empfiehlt sich, bei der Umsetzung in C++ Werte statt Zeiger zu nehmen. Dann ist gewährleistet, dass die Lebensdauer der Teile an das Ganze gebunden ist:

```
class Ganzes {
public:
   Ganzes(int datenFuerTeil1, int datenFuerTeil2)
   : ersterTeil(datenFuerTeil1),
     zweiterTeil(datenFuerTeil2) {
     // ...
   }
   // ...
private:
   Teil ersterTeil;
   Teil zweiterTeil;
};
```