C++ programmieren

C++ lernen – professionell anwenden – Lösungen nutzen

1.4 Das erste Programm

Das klassische erste Programm ist ein Mini-Programm, das einfach nur »Hello World!« ausgibt. Das Listing 1.1 zeigt den Programmcode.

Listing 1.1: Hello World-Programm *(cppbuch/k1/hello.cpp)*

```
#include <iostream>

int main()
{
  std::cout << "Hello␣World!\n";
}
```

Die Entwicklung eines einfachen Programms lernen Sie hier an einer ebenfalls einfachen Aufgabe kennen: Es sollen zwei Zahlen addiert werden. Dabei wird Ihnen zunächst das Programm vorgestellt und gleich danach 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 1.2: Programmentwurf

```
int main()                  // Noch tut dieses Programm nichts!
{
    // Lies zwei Zahlen ein
    /* Berechne die Summe beider
       Zahlen   */
    // Zeige das Ergebnis auf dem Bildschirm an
}
```

Sie sehen einen einfachen Entwurf, der gleichzeitig ein C++-Programm ist. Es tut allerdings noch nichts. Es bedeuten:

`int`	ganze Zahl zur Rückgabe
`main`	Name der Funktion, mit der jedes Programm beginnt
`()`	Innerhalb dieser Klammern können der Funktion 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 Block sind lediglich *Kommentare* enthalten und noch keine Anweisungen an den Computer, sodass unser Programm nichts tut.

Kommentare werden einschließlich der Kennungen vom Compiler vollständig ignoriert. Ein Kommentar, der mit `/*` beginnt, wird 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 ignoriert werden, sind sie doch sinnvoll für alle, die ein Programm lesen. Die Anweisungen zu erläutern hilft denjenigen, die Ihre Nachfolge antreten, weil Sie befördert worden sind oder die Firma verlassen haben. Kommentare sind auch wichtig für den Autor eines Programms, der ohne sie nach einem halben Jahr nicht mehr weiß, warum er gerade diese oder jene komplizierte Anweisung geschrieben hat. Sie sehen:

Ein Programm ist 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.

Hinweis

Alle Programmbeispiele sind von der Internet-Seite *http://cppbuch.de/* herunterladbar. In den Listings finden Sie den zugehörigen Dateinamen in der Überschrift oder in der ersten Zeile des Listings.

Listing 1.3: Summe zweier Zahlen berechnen *(cppbuch/k1/summe.cpp)*

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

int main()
{
  int summand1 {0};
  int summand2 {0};
  // Lies zwei Zahlen ein
  cout << "␣Zwei␣ganze␣Zahlen␣eingeben:";
  cin >> summand1 >> summand2;
  /* Berechne die Summe beider Zahlen
  */
  int 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, werden Sie bald erfahren.

`#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.

`using namespace std;` Der Namensraum (englisch *namespace*) `std` wird benutzt. Schreiben Sie es einfach in jedes Programm an diese Stelle und haben Sie Geduld: Erklärungen folgen auf den Seiten 63 und 151.

`int main()` `main()` ist die Funktion, mit der jedes Programm beginnt (es gibt auch andere Funktionen). Der zu `main()` gehörende Programmcode wird durch die geschweiften Klammern `{` und `}` eingeschlossen.

Ein mit `{` und `}` begrenzter Bereich heißt *Block*. Mit `int` ist gemeint, dass die `main()`-Funktion 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 einen Fehler signalisieren.

```
int summand1 {0};
int summand2 {0};
int summe = ...
```

Deklaration (Bekanntmachung) von Objekten: Mitteilung an den Compiler, der ab jetzt die Namen `summand1`, `summand2` und `summe` innerhalb des Blocks `{ }` kennt. Hier wird gleichzeitig Speicherplatz bereitgestellt. Es gibt verschiedene Zahlentypen in C++. Mit `int` sind ganze Zahlen gemeint: `summe`, `summand1`, `summand2` sind ganze Zahlen. Oft ist es sinnvoll, einen Anfangswert festzulegen, etwa 0, wie hier bei `summand1` und `summand2`.

`;` Ein Semikolon beendet jede Deklaration und jede Anweisung (aber keine Verbundanweisung, siehe weiter unten).

`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`.

`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.

`"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` an das Betriebssystem zurück. Diese Anweisung darf in der `main()`-Funktion 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 keine gängige deutsche Entsprechung gibt. Einen Header mit einem Dateinamen gleichzusetzen, ist meistens richtig, nach dem C++-Standard aber nicht zwingend. *Hinweis:* Tatsächlich wird ein Programm zunächst von einem vorgeschalteten *Präprozessor* bearbeitet, der das Ergebnis der Bearbeitung an den eigentlichen Compiler weiterleitet. Wenn im Folgenden also von »Compiler« die Rede ist, ist meistens auch der Präprozessor gemeint.

`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« [3] wird für ein veränderliches Objekt gebraucht. Für Variablen gilt:

- Sie 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 Grammatikregeln. Unter *Deklaration* wird verstanden, dass der Name dem Compiler bekannt gemacht wird. Wenn dieser Name danach im Programm versehentlich falsch geschrieben wird, kennt der Compiler den falschen Namen nicht und gibt eine Fehlermeldung aus. Somit 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 mehr. 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.

[3] Anmerkung für Menschen mit Vorkenntnissen: Aus reiner C++-Sicht ist eine Variable eine Deklaration eines Objekts (oder einer Referenz) und sagt nichts darüber aus, ob es konstant oder veränderlich ist. Die Eigenschaft »konstant« wird durch das Schlüsselwort `const` bewirkt.

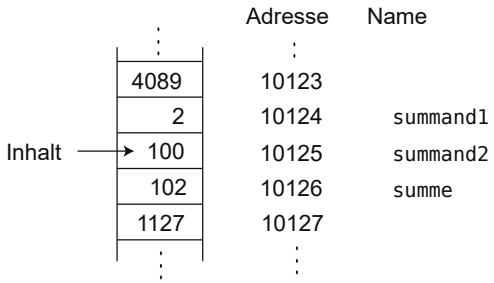


Abbildung 1.1: Speicherbereiche mit Adressen

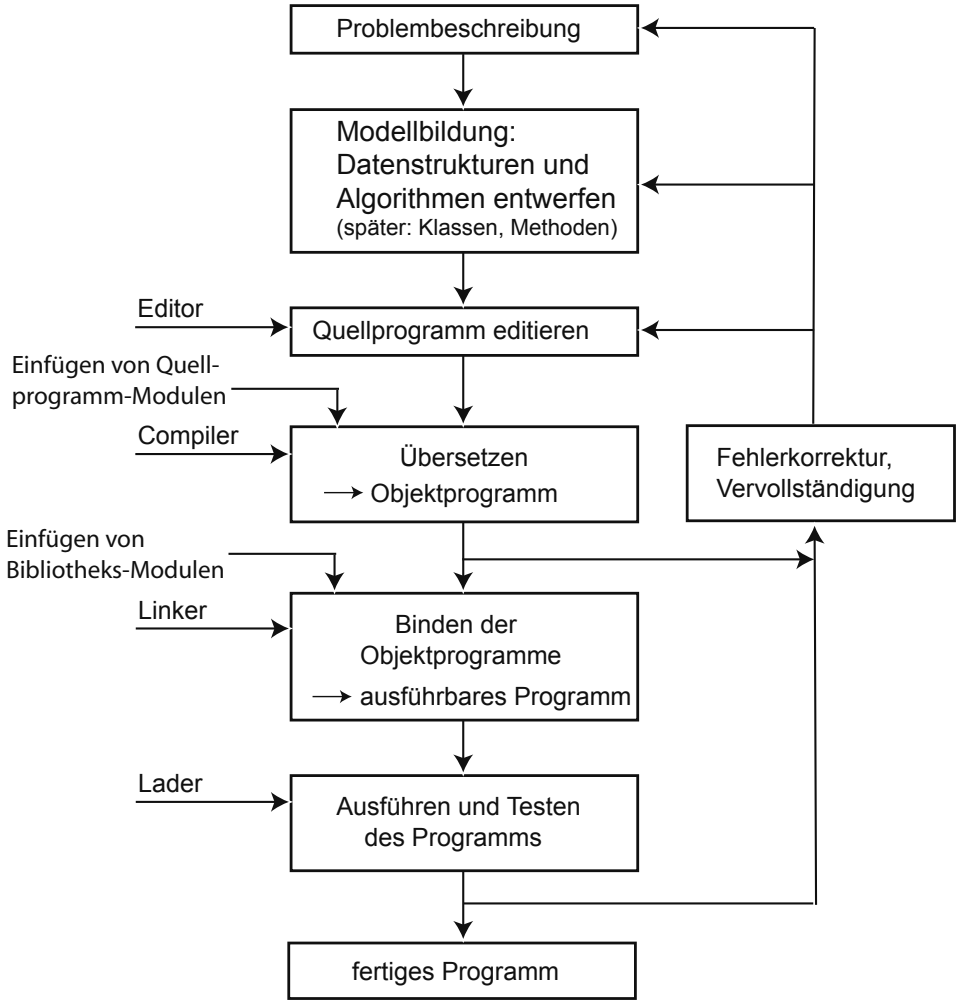


Abbildung 1.2: Erzeugung eines lauffähigen Programms

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.

Ein Programmtext wird auch »Quelltext« oder »Quellcode« (englisch *source code*) genannt. Der Compiler erzeugt aus dem Quellcode 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).

Wie bekomme ich ein Programm zum Laufen?

Nachdem Sie den Programmtext mit einem Editor geschrieben haben, speichern Sie ihn als Datei *summe.cpp* ab. Öffnen Sie nun unter Windows eine Konsole (cmd-Eingabeaufforderung oder PowerShell) bzw. ein Terminal unter Linux oder macOS, und wechseln mit `cd` in das Verzeichnis, wo Sie *summe.cpp* abgespeichert haben. Die Übersetzung, auch Compilation genannt, wird mit

```
g++ -o summe.exe summe.cpp
```

gestartet. Das Programm wird durch Eintippen von *summe.exe* (oder *./summe.exe*, wenn das aktuelle Verzeichnis nicht im Pfad ist) gestartet. Eigentlich verbergen sich hinter dem Aufruf des Compilers zwei Schritte:

`g++ -c summe.cpp`	compilieren (*summe.o* wird erzeugt)
`g++ -o summe.exe summe.o`	linken
`g++ -o summe.exe summe.cpp`	beide Schritte zusammengefasst

Die Objektdateien können je nach System die Endung *.o* oder *.obj* tragen. Wenn `g++ -std=c++23 -o summe.exe summe.cpp` geschrieben wird, soll der Compiler den C++-Standard 2023 verwenden. Das ist bei diesem einfachen Programm nicht notwendig, weil es keine der neueren Eigenschaften nutzt.

Eine integrierte Entwicklungsumgebung lässt die genannten Schritte per Tastendruck ablaufen, wie Sie gleich sehen.

switch/case-lokale Variablen

Wenn in der `case`-Anweisung eine lokale Variable angelegt wird, muss sie in einem eigenen Gültigkeitsbereich (scope) mit geschweiften Klammern `{}` gekapselt werden:

```
case 'D':
  zahl = 500;
  {
    auto x{zahl*10};              // Beginn des lokalen Gültigkeitsbereichs.
    cout << x << '\n';
  }                               // Die Gültigkeit von x endet hier.
  break;
```

Das Weglassen der geschweiften Klammern führt zu einer Fehlermeldung des Compilers. Wie bei der `if`-Anweisung ist es möglich, eine lokale Variable für die `switch()`-Anweisung anzulegen. Die Variable wird in den runden Klammern initialisiert, also etwa `switch(int x=1; auswahl)`.

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 1.8 zeigt die Syntax von `while`-Schleifen.

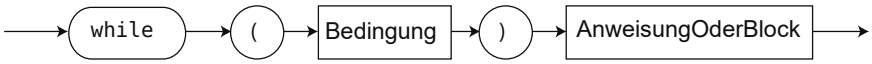
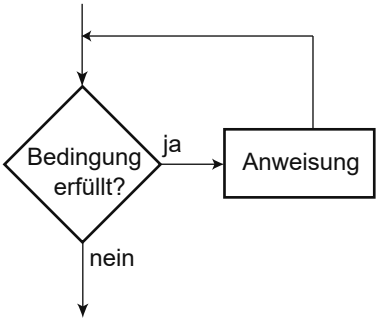


Abbildung 1.8: Syntaxdiagramm einer `while`-Schleife

AnweisungOderBlock ist wie auf Seite 66 definiert. Die Bedeutung einer `while`-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis `true` oder ungleich 0 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 1.9).

Die Anweisung oder der Block innerhalb der Schleife heißt *Schleifenkörper*. Schleifen können wie `if`-Anweisungen beliebig geschachtelt werden.

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




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

Abbildung 1.9: 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};
constexpr 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 1.15: Beispiel für `while`-Schleife *(cppbuch/k1/ggt.cpp)*

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

int main()
{
  int x {0};
  int y {0};
  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 1.10 zeigt die Syntax einer `do while`-Schleife.

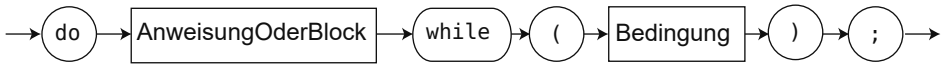
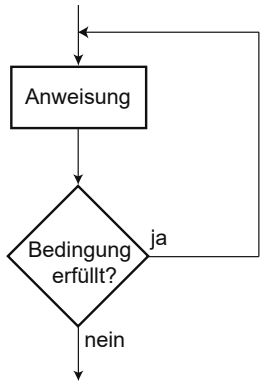


Abbildung 1.10: Syntaxdiagramm einer `do while`-Schleife

AnweisungOderBlock ist wie auf Seite 66 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 1.11).

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

Abbildung 1.11: Flussdiagramm für eine `do while`-Anweisung

```
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 1.16: Berechnen einer Primzahl mit `do while` *(cppbuch/k1/primzahl.cpp)*

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

int main()
{
  cout << "Berechnung␣der␣ersten␣Primzahl,␣die␣>="
          "␣der␣eingegebenen␣Zahl␣ist\n";
  // Hinweis: Mehrere, durch " getrennte Texte ergeben eine lange Zeile in der Ausgabe.
  long zahl {0L};
  // do while-Schleife zur Eingabe und Plausibilitätskontrolle
  do {
    // Abfrage, solange zahl ≤ 3 ist
    cout << "Zahl␣>␣3␣eingeben␣:";
    cin >> zahl;
  } while (zahl <= 3);

  if (zahl % 2 == 0) {   // Falls zahl gerade ist, wird die nächste
                         // ungerade Zahl als Startwert genommen.
    ++zahl;
  }
  bool gefunden {false};
  do {
    // limit = Grenze, bis zu der gerechnet werden muss.
    // sqrt() arbeitet mit double, daher wird der Typ explizit umgewandelt.
    const long limit {1 + static_cast<long>(sqrt(static_cast<double>(zahl)))};
    long rest {0L};
```

```
    long teiler {1L};
    do {                        // Kandidat zahl durch alle ungeraden Teiler dividieren
      teiler += 2;
      rest = zahl % teiler;
    } while (rest > 0 && teiler < limit);

    if (rest > 0 && teiler >= limit) {
      gefunden = true;
    }
    else {                      // sonst nächste ungerade Zahl untersuchen
      zahl += 2;
    }
  } while (!gefunden);
  cout << "Die␣nächste␣Primzahl␣ist␣" << zahl << '\n';
}
```

Tipp

In `do-while`-Schleifen wird die Bedingung erst am Ende abgefragt, sie ist beim Lesen also nicht sofort zu finden. Deswegen verwenden Sie `do-while` nur, wenn Sie erreichen wollen, dass eine Schleife mindestens einmal durchlaufen werden soll.

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 1.12 zeigt die Syntax einer `for`-Schleife.

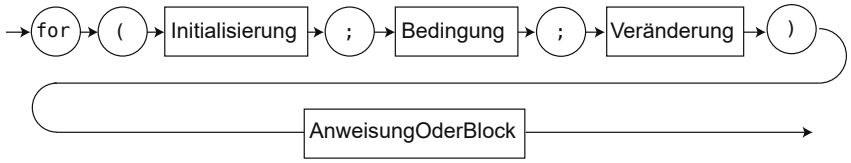


Abbildung 1.12: Syntaxdiagramm einer `for`-Schleife

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`- und `switch`-Anweisungen.

Listing 1.17: Beispiel für `for`-Schleife *(cppbuch/k1/fakultaet.cpp)*

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

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

  long fak {1L};
  for (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
  }
}
```

float- oder double-Laufvariablen vermeiden!

Wegen der Rechenungenauigkeit kann es bei nicht-integralen Typen wie `double` oder `float` zu nicht vorhersagbarem Verhalten kommen. Das Beispiel:

```
for (double d = 0.4; d <= 1.2; d += 0.4) {
  cout << d << '\n';
}
```

lässt auf den ersten Blick die Ausgabe 0.4, 0.8, 1.2 erwarten, tatsächlich werden auf meinem System nur die zwei Zahlen 0.4 und 0.8 ausgegeben. Wenn ich jedoch

```
for (double d = 0.5; d <= 1.5; d += 0.5) {
  cout << d << '\n';
}
```

ausführe, werden wie erwartet die drei Zahlen 0.5, 1 und 1.5 angezeigt. Der Grund liegt darin, dass 0.5 im Binärsystem exakt darstellbar ist, 0.4 jedoch nicht. Schon ein Unterschied im letzten Bit lässt den Vergleich auf Gleichheit scheitern. Ganz ungünstig kann sich die Prüfung auf Ungleichheit mit `!=` auswirken:

Die Ausgabe des Programms ist:

```
Objekt 0 wird erzeugt.
main wird begonnen
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 von Objekten mit statischer Lebensdauer (`static` oder globale Objekte) wird nicht nur beim Verlassen eines Programms mit `return`, sondern auch beim Verlassen mit `exit()` aufgerufen. Im Gegensatz zum normalen Verlassen eines Blocks wird der Speicherplatz bei `exit()` jedoch nicht freigegeben.

3.7 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 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 1.25 von Seite 112 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.

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 (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-objektorientierten Lösung zur Vorläuferaufgabe 1.25 werden alle Aktivitäten in `main()` abgehandelt. Das ist unvorteilhaft, 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 Auswahl 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 Auswahl 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 behandelt, zum Beispiel [Oe]. In diesem einfachen Fall konzentrieren wir uns gleich auf eine Lösung mit C++. Eine `main()`-Funktion könnte wie folgt aussehen:

Listing 3.36: `main`-Programm zur Personalverwaltung *(cppbuch/k3/personalverwaltung/main.cpp)*

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

int main()
{
  Personalverwaltung personalverwaltung("daten.txt");
  cout << "Gelesene Namen und Personalnummern:\n";
  personalverwaltung.ausgeben();
  personalverwaltung.dialog();
  cout << "Programmende\n";
}
```

Die Klasse `Person` ist einfach zu entwerfen:

Listing 3.37: 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_}
  { }

  [[nodiscard]] auto getName() const { return name; }

  [[nodiscard]] 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 3.38: Klasse `Personalverwaltung` *(cppbuch/k3/personalverwaltung/Personalverwaltung.h)*

```
#ifndef PERSONALVERWALTUNG_H
#define PERSONALVERWALTUNG_H
#include "Person.h"
#include <vector>

class Personalverwaltung {
public:
  explicit 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 1.25 von Seite 112 gelöst oder deren Lösung nachgesehen haben.

Übungen

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

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

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 [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, die Umsetzung zu gestalten. Im Einzelfall kann eine Variation sinnvoll sein.

22.1 Vererbung

Über Vererbung als »ist ein«-Beziehung wurde in diesem Buch schon einiges gesagt, was 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.

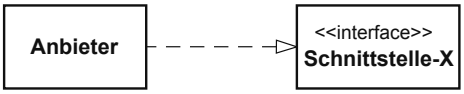


Abbildung 22.2: Interface-Anbieter

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&) = delete;
  SchnittstelleX& operator=(const SchnittstelleX&) = delete;
};

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.

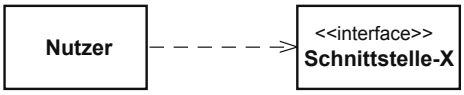


Abbildung 22.3: Interface-Nutzer

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 `nullptr` 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 zunächst 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);`. `anbieter` kennt durch die Parameterübergabe das Objekt `daten`, wird aber vermutlich die Verbindung nach Ablauf der Funktion lösen.

Einfache gerichtete Assoziation

Die Abbildung 22.4 zeigt das UML-Diagramm einer einfachen gerichteten Assoziation.

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. Das Sternchen * bei Klasse2 besagt, dass einem Objekt der Klasse1 beliebig viele Objekte der Klasse2 zugeordnet sind, also möglicherweise auch keins.

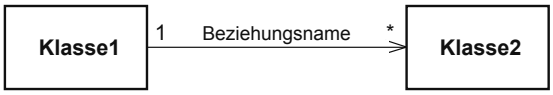


Abbildung 22.5: Gerichtete Assoziation mit Multiplizitäten

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 Multiplizität: Ein `Fan` kennt `Popstars`, aber nicht umgekehrt.

```
class Fan {
public:
  void werdeFanVon(Popstar* star)
  {
    meineStars.insert(star);                // einfügen
  }

  void denKannsteVergessen(Popstar* star)
  {
    meineStars.erase(star);                 // entfernen. 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.

Abbildung 22.6: 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 22.7. Solche Assoziationen werden auch rekursiv genannt und dienen zur Darstellung der Beziehung verschiedener Objekte derselben Klasse.

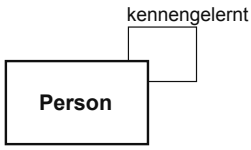


Abbildung 22.7: Rekursive Assoziation

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. Natürlich kann es vorkommen, dass es zwei Personen mit demselben Namen gibt, hier Frau Holle.

Listing 22.6: Assoziation: Personen lernen sich kennen *(cppbuch/k22/bidirektAssoziation/main.cpp)*

```
#include "Person.h"

int main()
{
  Person mabuse("Dr. Mabuse");
  Person klicko("Witwe Klicko");
  Person holle1("Frau Holle");
  Person holle2("Frau Holle");                    // eine Namensvetterin!
  mabuse.lerntkennen(klicko);
  holle1.lerntkennen(klicko);
  holle1.lerntkennen(holle2);
  mabuse.bekannteZeigen();
  klicko.bekannteZeigen();
  holle1.bekannteZeigen();
}
```

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. Wenn in der Menge der Bekannten nur die Namen als String gespeichert würden, könnten sich zwei Personen mit demselben Namen nicht kennenlernen. Deswegen werden die Adressen der `Person`-Objekte gespeichert (siehe Attribut `set<Person*>` in der Klasse unten).

Listing 22.7: Klasse `Person` *(cppbuch/k22/bidirektAssoziation/Person.h)*

```
#ifndef PERSON_H
#define PERSON_H
#include <iostream>
#include <set>
#include <string>
#include <utility>

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

  auto getName() const
  {
    return name;
  }

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

  void bekannteZeigen() const
  {
    std::cout << "Die Bekannten von " << getName() << " sind:\n";
    for (const auto& bekannt : bekannte) {
      std::cout << bekannt->getName() << '\n';
    }
  }

private:
  std::string name;
  std::set<Person*> bekannte;
};
#endif
```

Hinweis

Die Methode `insert()` eines Sets gibt ein `pair`-Objekt zurück. Das ist eine Struktur mit den Elementen `first` und `second`. `second` ist ein Wahrheitswert, der angibt, ob das Einfügen stattgefunden hat. Wenn nein, war das Element schon vorhanden. `first` ist ein Iterator auf das Element, ob gerade eingefügt oder schon vorhanden gewesen.

22.3.1 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 22.8 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.

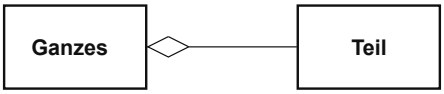


Abbildung 22.8: Aggregation

22.3.2 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. Formal ist auch 0 erlaubt. Für ein isoliertes Objekt ist jedoch der Begriff »Teil« nicht sinnvoll. Die Abbildung 22.9 zeigt das UML-Diagramm.

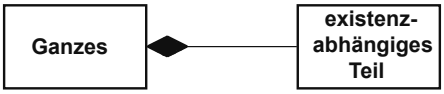


Abbildung 22.9: Komposition

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:

Listing 22.8: Umsetzung der Komposition

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

private:
  Teil ersterTeil;
  Teil zweiterTeil;
};
```

Register

Symbole

A

B

C

D

E

F

J

K

M

N

O

P

Q

R

S

T

U

X

Y

Z