AKTUELL ZU C++23

Ulrich **BREYMANN**

7. Auflage

C++ programmieren

// C++ LERNEN
// PROFESSIONELL ANWENDEN
// LÖSUNGEN NUTZEN

Im Internet: alle Beispiele und mehr auf www.cppbuch.de

HANSER

Breymann

C++ programmieren

Ulrich Breymann

C++ programmieren

C++ lernen – professionell anwenden – Lösungen nutzen

7., überarbeitete Auflage

HANSER

Prof. Dr. Ulrich Breymann lehrte Informatik an der Fakultät Elektrotechnik und Informatik der Hochschule Bremen.

Kontakt: breymann@hs-bremen.de

Bibliografische Information der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Petra Kienle, Fürstenfeldbruck
Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München
Umschlagrealisation: Max Kostopoulos
Titelmotiv: © stock.adobe.com/michels
Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Gottingen
Printed in Germany

Print-ISBN: 978-3-446-47689-9

E-Book-ISBN: 978-3-446-47846-6

E-Pub-ISBN: 978-3-446-47964-7

Inhalt

Vorwort

Diese Auflage unterscheidet sich von der vorherigen durch eine gründliche Überarbeitung und die Umstellung auf den 2023 von der zuständigen ISO/IEC-Arbeitsgruppe verabschiedeten C++-Standard. Abschnitt A.7 bietet eine Übersicht der in diesem Buch berücksichtigten Änderungen. Das Buch ist konform zum C++23-Standard, ohne den Anspruch auf Vollständigkeit zu erheben – das Standarddokument [ISOC++] umfasst mehr als 2100 Seiten. Sie finden in diesem Buch eine verständliche und mit vielen Beispielen angereicherte Einführung in die Sprache, unabhängig vom Betriebssystem.

Für wen ist dieses Buch geschrieben?

Es ist für alle geschrieben, die einen kompakten und gleichzeitig in die Tiefe gehenden Einstieg in die Programmierung mit C++ suchen. Es ist für Interessierte ohne Programmiererfahrung gedacht und für andere, die diese Programmiersprache kennenlernen möchten. Beiden Gruppen dient das Buch als Lehrbuch und Nachschlagewerk.

Ein umfassendes Handbuch

Die ersten zehn Kapitel führen in die Sprache ein, die folgenden behandeln fortgeschrittene Themen. Die sofortige praktische Umsetzung des Gelernten anhand von leicht nachvollziehbaren Beispielen steht im Vordergrund. Klassen und Objekte, Templates und Exceptions sind Ihnen bald keine Fremdworte mehr. Es gibt 99 Übungsaufgaben – mit Musterlösungen im Anhang und zum Download. Durch das Studium dieser Kapitel werden aus Neulingen bald Fortgeschrittene – und mithilfe der weiteren Kapitel Experten.

C++ in praktischen Anwendungen

Sie finden kurze Einführungen in die Themen Programmierung paralleler Abläufe, Netzwerk-Programmierung einschließlich eines kleinen Webservers, Datenbankanbindung, grafische Benutzungsoberflächen und Zugriff auf die KIs ChatGPT und DALL·E 2. Durch den Einsatz der Boost-Library und des Qt-Frameworks wird größtmögliche Portabilität erreicht.

Softwareentwicklung ist nicht nur Programmierung

Sie lernen die Automatisierung der Programmerzeugung mit Make kennen. Das Programmdesign wird durch konkrete Umsetzungen von Design-Mustern nach C++ unterstützt. Das Kapitel über Unit-Tests zeigt, wie Programme getestet werden können. Das integrierte »C++-Rezeptbuch« mit mehr als 150 praktischen Lösungen, der Teil über die C++-Standardbibliothek, das umfangreiche Register und das detaillierte Inhaltsverzeichnis machen das Buch zu einem praktischen Nachschlagewerk für alle, die sich mit der Softwareentwicklung in C++ beschäftigen.

Moderne Programmiermethodik

Sie möchten Programme schreiben, die hohen Qualitätsansprüchen gerecht werden. Dazu gehört das Know-how, C++ richtig einzusetzen. Dass ein Programm läuft, reicht nicht. Es soll auch gut entworfen sein, möglichst wenige Fehler enthalten, selbst mit Fehlern in Daten umgehen können, verständlich geschrieben und schnell in der Ausführung sein. Deshalb liegt ein Schwerpunkt des Buchs auf guter Codierpraxis entsprechend den »C++ Core Guidelines«. Die Umsetzung wird an vielen Beispielen gezeigt.

Wie benutzen Sie dieses Buch am besten?

Es eignet sich zum Selbststudium oder als Begleitbuch zu einem Kurs oder einer Vorlesung. Man lernt am besten durch eigenes Tun! Dabei hilft es, die Beispiele herunterzuladen, sie zu studieren und zu modifizieren (*http://www.cppbuch.de/*). Auch wird empfohlen, die Übungsaufgaben zu lösen. Um sowohl Anfängern als auch Fortgeschrittenen gerecht zu werden, gibt es einfache, aber auch schwerere Aufgaben. Wenn Ihnen eine Lösung nicht gelingt – einfach bei den Lösungen im Anhang nachsehen bzw. im Verzeichnis *cppbuch/loesungen* der downloadbaren Beispiele. Und dann versuchen, die Lösungen nachzuvollziehen.

Wo finden Sie was?

Bei der Programmentwicklung wird häufig das Problem auftauchen, etwas nachschlagen zu müssen. Es gibt die folgenden Hilfen: Erklärungen zu Begriffen sind im *Glossar* aufgeführt. Es gibt ein umfangreiches *Stichwortverzeichnis* und ein detailliertes *Inhaltsverzeichnis*. Der Anhang enthält unter anderem verschiedene hilfreiche Tabellen und die Lösungen der Übungsaufgaben. Auf der Webseite *http://www.cppbuch.de/* finden Sie die Software zu diesem Buch. Sie enthält alle Programmbeispiele und die Lösungen zu den Aufgaben. Sie finden dort auch weitere Hinweise, Errata und nützliche Internet-Links.

Zu guter Letzt

Allen Menschen, die dieses Buch durch Hinweise und Anregungen verbessern halfen, sei an dieser Stelle herzlich gedankt. Insbesondere Prof.Dr .Ulrich Eisenecker danke ich für seine hilfreichen Kommentare. Frau Irene Weilhart vom Hanser Verlag und dem Lektorat danke ich für die gute Zusammenarbeit.

Bremen, im Juni 2023

Ulrich Breymann

Teil I:

Einführung in C++

1

Es geht los!

Dieses Kapitel behandelt die folgenden Themen:

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

1.1 Historisches

C++ wurde etwa ab 1980 von Bjarne Stroustrup als die Programmiersprache »C with classes« (englisch *C mit Klassen*), die Objektorientierung stark unterstützt, auf der Basis der Programmiersprache C entwickelt. Später wurde die neue Sprache in C++ umbenannt. ++ ist ein Operator der Programmiersprache C, der den Wert einer Größe um 1 erhöht. Insofern spiegelt der Name die Eigenschaft »Nachfolger von C«. 1998 wurde C++ erstmals von der ISO (International Organization for Standardization) und der IEC (International Electrotechnical Commission) standardisiert. Diesem Standard haben sich nationale Standardisierungsgremien wie ANSI (USA) und DIN (Deutschland) angeschlossen. Die Anforderungen an C++ sind gewachsen, auch zeigte sich, dass manches fehlte und anderes überflüssig oder fehlerhaft war. Das C++-Standardkomitee hat kontinuierlich an der Verbesserung von C++ gearbeitet. Seit 2011 werden im Abstand von drei

Jahren neue Versionen des Standards herausgegeben. Die Kurznamen sind entsprechend den Jahreszahlen C++11, C++14, C++17, C++20 und C++23. C++23 wurde von der zuständigen ISO/IEC-Arbeitsgruppe JTC1/SC22/WG21 verabschiedet und bei der ISO zur Veröffentlichung eingereicht.

1.2 Arten der Programmierung

Es gibt viele verschiedene Arten der Programmierung. C++ unterstützt im Wesentlichen die folgenden:

Imperative Programmierung

Dabei werden die im Programmcode festgelegten Schritte der Reihe nach ausgeführt. Es geht also darum, einem Rechner mitzuteilen, *was* er tun soll und *wie* es zu tun ist. Ein Programm ist ein in einer Programmiersprache formulierter Algorithmus oder anders ausgedrückt eine Folge von Anweisungen, die der Reihe nach auszuführen sind, ähnlich einem Kochrezept. Der Algorithmus wird in einer besonderen Sprache, die der Rechner »versteht«, geschrieben. Der Schwerpunkt dieser Betrachtungsweise liegt auf den einzelnen Schritten oder Anweisungen an den Rechner, die zur Lösung einer Aufgabe abzuarbeiten sind. Mit sogenannten Kontrollstrukturen wird der Programmablauf gesteuert. So können Teile wiederholt ausgeführt oder übersprungen werden.

Objektorientierte Programmierung

Bei der imperativen Programmierung wird ein Aspekt eher stiefmütterlich behandelt: Der Rechner muss »wissen«, *womit* er etwas tun soll. Zum Beispiel soll er

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

Häufig, wie in den ersten beiden Fällen, werden Objekte der realen Welt (Konten, Ampelanlage ...) *simuliert*, das heißt im Rechner abgebildet. Die abgebildeten Objekte haben eine *Identität*. Das *Was* und das *Womit* gehören stets zusammen. Beide sind also Eigenschaften eines Objekts und sollen daher nicht getrennt werden. Ein Konto kann schließlich nicht auf Gelb geschaltet werden und eine Überweisung an eine Ampel ist nicht vorstellbar. Ein *objektorientiertes Programm* kann man sich als Abbildung von Objekten der realen Welt in Software vorstellen. Die Abbildungen werden in C++ selbst wieder Objekte genannt. *Klassen* sind Beschreibungen von *Objekten*.

Generische Programmierung

Die generische Programmierung ermöglicht es, Klassen und Algorithmen für verschiedene Datentypen mit nur einem Schema zu modellieren. Das gilt für die imperative Programmierung ebenso wie für die objektorientierte Programmierung. Andere Arten der Programmierung sind für C++ von geringerer Bedeutung.

1.3 Werkzeuge zum Programmieren

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

Editor/Entwicklungsumgebung

Ein Editor ist ein Programm, mit dem man Texte schreiben kann. Dabei darf ein Text keine versteckten Sonderzeichen enthalten, weswegen LibreOffice oder Word nicht geeignet sind. Es gibt viele passende Texteditoren. In diesem Buch beschränke ich mich auf *Geany*[1], weil es ihn für Windows, Linux und macOS gibt und weil er einfach zu bedienen ist. Gleichzeitig ist er eine einfache IDE (Integrated Development Environment, dt. integrierte Entwicklungsumgebung). In Abschnitt 1.5 finden Sie mehr dazu.

Der Compiler

Der Compiler ist ein Programm, das Ihren Programmtext in eine für den Computer verarbeitbare Form übersetzen kann. Von Menschen geschriebener Programmtext kann vom Computer nicht so verstanden werden, wie wir einen Text verstehen. Das vom Compiler erzeugte Ergebnis der Übersetzung kann der Computer aber ausführen. Das Erlernen einer Programmiersprache ohne eigenes praktisches Ausprobieren ist kaum sinnvoll. Nutzen Sie daher die Dienste des Compilers möglichst bald anhand der Beispiele. Wie, zeigt Ihnen der Abschnitt direkt nach der Vorstellung des ersten Programms.

GNU C++ Compiler: g++

Die am meisten verbreiteten Compiler sind der GNU C++-Compiler [GCC] *g++*, der zu Microsofts Visual Studio gehörende *cl* und *clang++* von LLVM[2]. Um C++ zu lernen, ist es letztlich egal, welchen Compiler Sie nehmen. In diesem Buch liegt der Schwerpunkt auf dem *g++*-Compiler, weil er Open-Source und sehr gut bekannt ist.

Windows

Installationshinweise finden Sie auf der Internetseite *http://cppbuch.de/downloads.html*. Im Folgenden wird davon ausgegangen, dass Sie die Installation wie dort beschrieben vorgenommen haben.

Linux

Bei den meisten Linux-Systemen ist der GNU C++-Compiler enthalten. Wenn nicht, finden Sie Hinweise zur Installation unter *http://www.cppbuch.de/swinstallation.html*.

---

1 *https://www.geany.org/*
2 *https://llvm.org*

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.

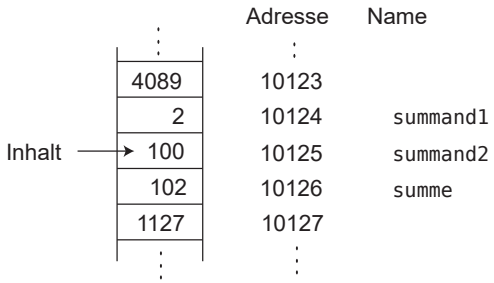


Abbildung 1.1: Speicherbereiche mit Adressen

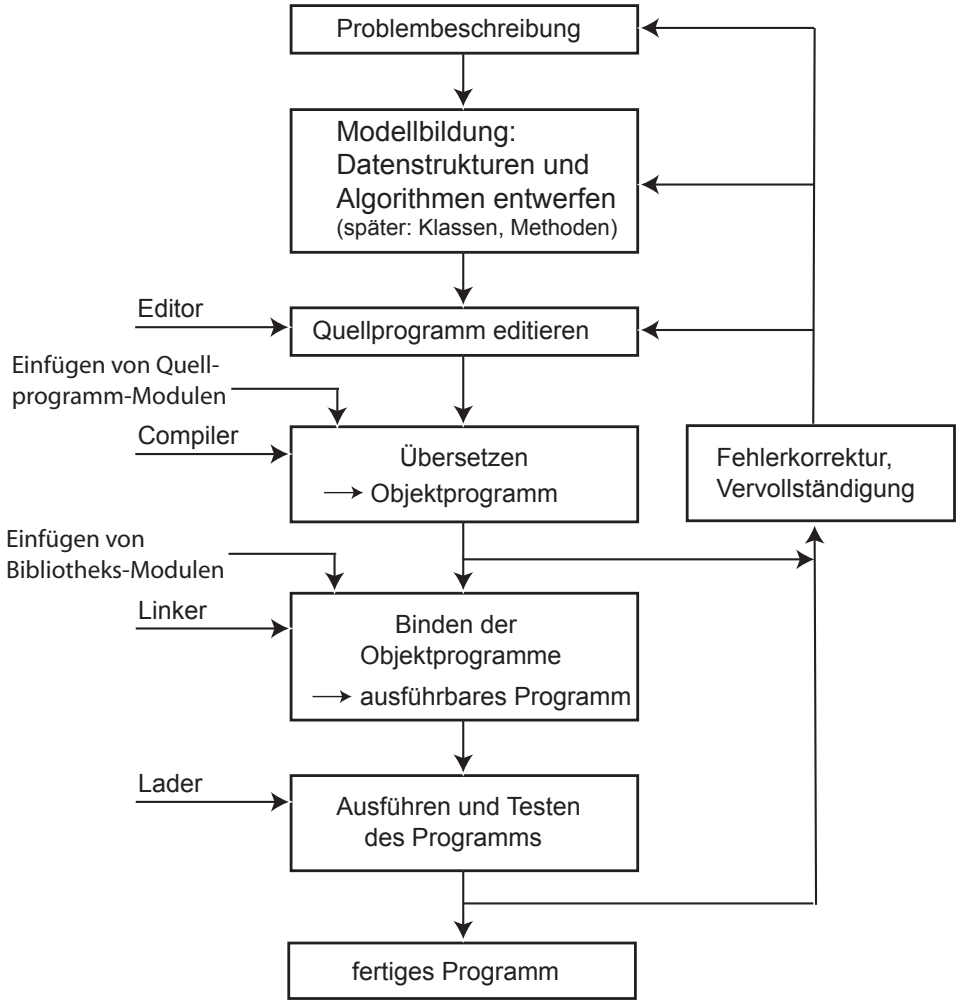


Abbildung 1.2: Erzeugung eines lauffähigen Programms

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.

1.5 Integrierte Entwicklungsumgebung

Integrierte Entwicklungsumgebungen (abgekürzt IDE für Integrated Development Environment) haben einen speziell auf Programmierzwecke zugeschnittenen Editor, der darüber hinaus auf Tastendruck oder Mausklick die Übersetzung anstößt. Es gibt einige davon. Ein paar Beispiele:

- Geany. Diese IDE ist Open Source. Es gibt sie für Windows, Linux und macOS. Sie ist einfach zu konfigurieren und zu benutzen.
- Code::Blocks. Diese IDE ist Open Source. Es gibt sie für Windows, Linux und macOS (für macOS leider nur in einer älteren Version). Sie ist ähnlich einfach zu benutzen.
- Visual Studio C++ von Microsoft. Diese IDE gibt es nur für Windows. Die Community-Edition ist kostenlos.
- Visual Studio Code von Microsoft. Diese IDE gibt es für Windows, Linux und macOS. Sie ist Open Source und kostenlos, aber nicht so einfach wie Geany zu konfigurieren.
- Eclipse und NetBeans sind auch beliebt. Sie haben die Programmiersprache Java als Schwerpunkt, es gibt aber jeweils ein Zusatzmodul (»Plug-in«) für C++.
- Xcode, die von Mac-Entwicklern bevorzugte IDE.

Die Arbeitsweisen der Entwicklungsumgebungen ähneln sich. Weil die IDE Geany sehr einfach und nicht auf ein Betriebssystem beschränkt ist, habe ich sie ausgewählt. Die anderen genannten Entwicklungsumgebungen (außer Code::Blocks) sind deutlich mächtiger und erfordern einen dementsprechend hohen Einarbeitungsaufwand. Der größere Funktionsumfang wird aber für den Einstieg nicht gebraucht. Wenn Sie später professionell programmieren, werden Sie eine mächtigere IDE benutzen, zum Beispiel Visual Studio, wenn Sie nur Programme für Windows schreiben.

In den Verzeichnissen der herunterladbaren Beispiele gibt es viele Dateien, die einzeln für sich ein Programm darstellen, wie das oben genannte Programm *summe.cpp*. Dass es viele einzelne Programmbeispiele gibt, liegt in der Natur eines Lehrbuchs über C++. Die industrielle Wirklichkeit sieht anders aus. Da besteht ein Programm aus vielen, manchmal Hunderten von Dateien. Das ist der Anwendungsbereich von Entwicklungsumgebungen: Sie verwalten viele zu einem Programm gehörende Dateien als sogenanntes Projekt. Der nächste Abschnitt zeigt die Bearbeitung eines einzelnen Programms mit einer Entwicklungsumgebung. Die Arbeit mit Projekten wird für den Anfang zurückgestellt.

Das erste C++-Programm mit Geany

Geany konfigurieren

Es wird angenommen, dass Sie die Entwicklungumgebung von *https://www.geany.org* heruntergeladen und installiert haben. Rufen Sie dann Geany auf. Um bei dem bekannten Beispiel zu bleiben, tippen Sie das bekannte Programm *summe.cpp* ein – oder Sie kopieren es. Speichern Sie es als *summe.cpp*. Nun können Sie Geany konfigurieren. Dazu gehen Sie in der Menüleiste auf »Erstellen« und wählen »Kommandos zum Erstellen

konfigurieren« aus. Tragen Sie nun die Kommandos entsprechend Abbildung 1.3 ein. Die ausgegrauten Teile können unverändert bleiben.

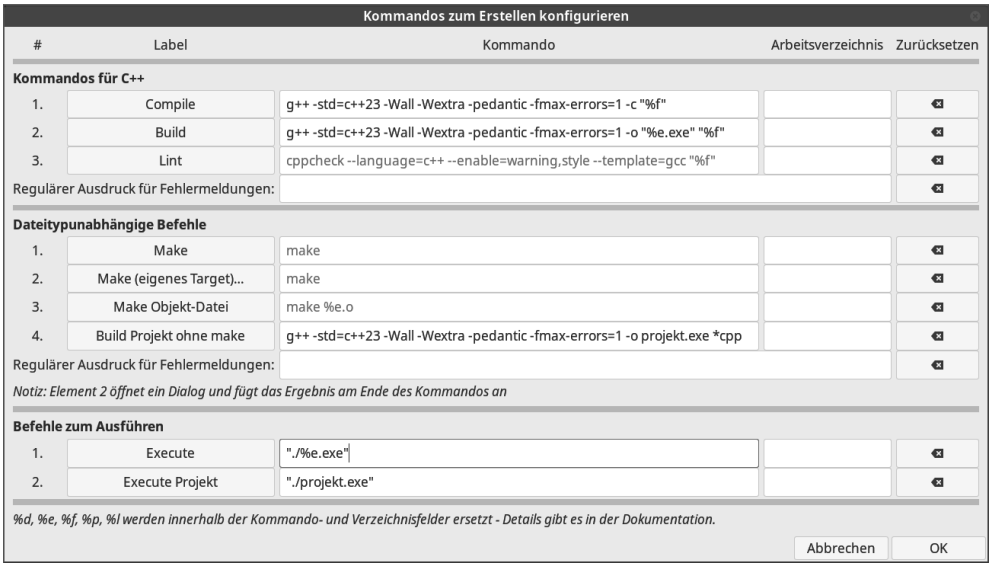


Abbildung 1.3: Kommandos zum Erstellen konfigurieren

Bei den »Kommandos für C++« sind nur die Kommandos für Compile und Build einzutragen. Bei »Dateiunabhängige Befehle« tragen Sie in Zeile 4 nicht nur das Kommando, sondern auch das Label »Build Projekt ohne make« in der linken Spalte ein. Die projektbezogenen Einträge spielen im nächsten Kapitel eine Rolle. Nun fehlen nur noch die Einträge bei »Befehle zum Ausführen«. Bitte achten Sie darauf, sich nicht zu vertippen. Zum Schluss mit OK bestätigen.

Programm compilieren und starten

Mit der Taste F8 wird das Programm nur übersetzt, um zu sehen, ob es vom Compiler akzeptiert wird. Mit F9 wird das ausführbare Programm erzeugt (und auch übersetzt, falls die Übersetzung mit F8 noch nicht stattgefunden hat). Mit der Taste F5 wird das Programm ausgeführt. Es erscheint ein Terminal mit der Ausgabe des Programms. Dort geben Sie nun zwei ganze Zahlen ein. Ein Druck auf die ENTER-Taste zeigt das Ergebnis an. Mit einem weiteren Tastendruck beenden Sie das Terminal.

Um zu zeigen, wie sich Fehler auswirken, ändern Sie bei der Eingabezeile mit `cin` den Namen der Variablen `summand2` in `summandx` ab und drücken wieder F8. Die Abbildung 1.4 zeigt den vom Compiler entdeckten Fehler an.

Nun korrigieren Sie den Fehler und drücken erneut F9. Jetzt wird das Programm wieder erfolgreich erzeugt und kann ausgeführt werden. Bei manchen anderen Entwicklungsumgebungen schließt sich das Fenster zu schnell. Man kann dann eine Pause erzwingen. Dazu werden am Programmende die folgenden Zeilen hinzugefügt:

```
cout << "Bitte␣drücken␣Sie␣Enter␣zum␣Beenden␣des␣Programms\n";
```

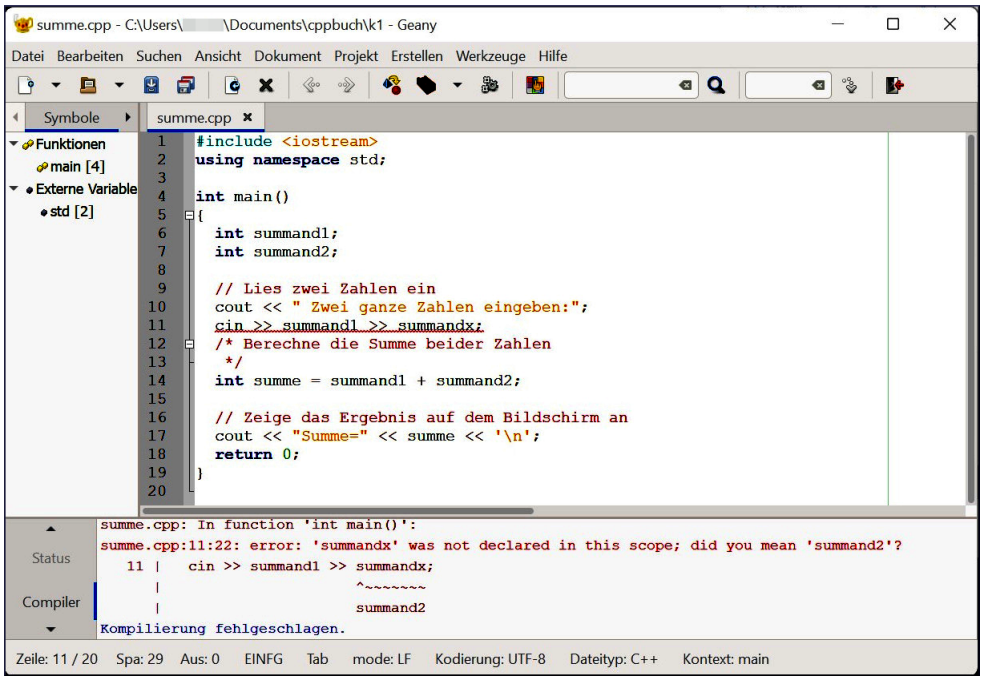


Abbildung 1.4: Geany zeigt einen Fehler an

```
cin.ignore(1000, '\n');       // löscht alle Zeichen bis zum Zeilenende, aber max. 1000
cin.get();
```

Übungen

1.1 Schreiben Sie bei der Ausgabe `caut` statt `cout` und lesen Sie die Fehlermeldungen des Compilers. Vermutlich erscheinen sie Ihnen etwas rätselhaft, aber das wird sich ändern.

1.2 Bauen Sie verschiedene andere Fehler ein und versuchen Sie, die Fehlermeldungen zu verstehen.

1.3 Modifizieren Sie das Programm so, dass auch die eingegebenen Zahlen in jeweils einer Zeile ausgegeben werden.

1.4 Schreiben Sie ein Programm, das eine kompliziertere Rechnung mit drei Variablen ausführt, zum Beispiel $c = (a1 + a2) * a3$. Bei der Eingabe können Sie pro Variable eine Zeile verwenden, etwa `cin << a1;`. Die Deklarationen dürfen nicht vergessen werden. *Hinweis*: Lösungen zu den meisten Aufgaben finden Sie im Anhang.

Hinweis

IDEs sind komplexe Produkte. Weil hier nur sehr kurz auf die Bedienung eingegangen werden kann, lesen Sie bei Fragen bitte die Dokumentation auf den entsprechenden Internetseiten.

1.6 Einfache Datentypen und Operatoren

Sie haben schon den Datentyp `int` für ganze Zahlen 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.

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 Konstante, einer Variable oder einem Literal.[4] Die Operatoren müssen zu den Operanden passen, die zu bestimmten Datentypen gehören. Beispiele:

```
17
1 + 17
a = 1 + 17
»Textliteral«.
```

`a = 1 + 17` ist ein zusammengesetzter Ausdruck. Die Operanden 1 und 17 sind Zahl-Literale, die ganze Zahlen repräsentieren und die durch den +-Operator verknüpft werden. Der resultierende Wert 18 wird dem Objekt a zugewiesen. Der Wert des gesamten Ausdrucks ist der resultierende Wert von a.

1.6.2 Regeln für Namen

Funktions-, Variablen- und andere Namen unterliegen den folgenden Regeln:

- Ein Name ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und Unterstrich (_). So eine Regel für die Struktur eines Namens wird *Syntax* oder *Grammatik* genannt. Darüber hinaus gibt es Empfehlungen, etwa dass der Name kurz sein, aber dennoch die Bedeutung widerspiegeln soll. So ist der Name `summe` klarer als ein Name `s`.
- Ein Name beginnt stets mit einem Buchstaben oder einem Unterstrich _. Am Anfang eines Namens sollen 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 961 zu finden.
- Ein Name kann prinzipiell beliebig lang sein. In den verschiedenen Compilern ist die Länge jedoch oft begrenzt, zum Beispiel auf 2048 Zeichen.

Ein Name darf niemals ein Leerzeichen enthalten! Wenn eine Worttrennung aus Gründen der Lesbarkeit gewünscht ist, helfen der Unterstrich und Wechsel in der Groß- und Kleinschreibung. Beispiele:

[4] Siehe Glossar Seite 981

`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.

1.6.3 Ganze Zahlen

Es gibt verschiedene rechnerinterne Darstellungen von ganzen Zahlen, die sich durch die bereitgestellte Anzahl von Bits pro Zahl unterscheiden. Die verschiedenen Darstellungen werden durch die Datentypen `short`, `int` und `long` repräsentiert, wobei gilt: Bits(`short`) $\leq$ Bits(`int`) $\leq$ Bits(`long`). Die tatsächlich verwendete Anzahl von Bits variiert je nach Rechnersystem. Typische Werte sind:

`short` (oder `short int`)	16 Bits
`int`	32 Bits (oder 16 Bits)
`long` (oder `long int`)	64 Bits (oder 32 Bits)
`long long` (oder `long long int`)	mindestens 64 Bits

Die eingeklammerten Zahlen sind entsprechend dem C++-Standard mindestens erforderlich. Ein Bit wird für das Vorzeichen reserviert. Durch das Schlüsselwort `unsigned` werden Zahlen *ohne* Vorzeichen definiert, zum Beispiel `unsigned int`, `unsigned long` (Langform: `unsigned long int`).

size_t

Der Datentyp `size_t` ist ein vorzeichenloser Ganzzahl-Typ für Größenangaben, die nicht negativ werden können, wie zum Beispiel die Anzahl der Einträge in einer Tabelle. Er ist groß genug, die Größe eines beliebigen Objekts in Bytes abzubilden. Die Definition steht im Header `<cstddef>`, der von vielen Standard-Headern bereits eingeschlossen wird. Auf 32-Bit-Systemen gibt es oft keinen Unterschied zu `unsigned int`, aber auf 64-Bit-Systemen kann `sizeof(unsigned int)` den Wert 4 ergeben und `sizeof(size_t)` den Wert 8. In so einem Fall umfasst `size_t` einen größeren Zahlenbereich. `sizeof` ermittelt die Anzahl der Bytes, die zum Speichern des Datentyps notwendig sind.

ptrdiff_t

Der Datentyp `ptrdiff_t` ist das Gegenstück von `size_t` für Größenangaben, die auch negativ sein können. `ptrdiff_t` ist die Abkürzung für *pointer difference type*, deutsch: Typ für Differenzen von Zeigern – ein weiterer Anwendungsbereich für diesen Typ. `ptrdiff_t` hat dieselbe Bitbreite wie `size_t` und ist auch in `<cstddef>` definiert. Auf 32-Bit-Systemen gibt es oft keinen Unterschied zu `int`, aber auf 64-Bit-Systemen kann `sizeof(int)` den Wert 4 ergeben und `sizeof(ptrdiff_t)` den Wert 8. In so einem Fall umfasst `ptrdiff_t` einen größeren Zahlenbereich.

Hinweis

Für den ersten Einstieg können Sie den bitweisen Aufbau der Zahlen sowie die Bitoperationen überspringen. Kehren Sie bei Bedarf zu diesem Abschnitt zurück. Der bitweise Aufbau der Zahlen und die Bitoperationen sind keine C++-Spezialität, sondern gelten für viele Programmiersprachen. Die Zahlenbereiche sollten Sie allerdings kennen, weil davon die Korrektheit und die Genauigkeit von Programmen abhängen können.

In den folgenden ausgewählten Bitkombinationen für 16-Bit-`int`-Zahlen repräsentiert das links stehende Bit das Vorzeichen:

binär	dezimal
0111 1111 1111 1111	32767
0000 0000 0000 0000	0
1111 1111 1111 1111	-1
1111 1111 1111 1110	-2
1000 0000 0000 0000	-32768

Negative Zahlen werden im Zweierkomplement dargestellt. Es wird gebildet, indem alle Bits invertiert werden und dann auf das Ergebnis 1 addiert wird. Bei `unsigned`-Zahlen braucht es kein Bit für das Vorzeichen und der Zahlenbereich teilt sich anders auf:

`signed`	16 Bits	-2^{15}	...	$2^{15}-1$	=	−32.768 ... 32.767
`unsigned`	16 Bits	0	...	$2^{16}-1$	=	0 ... 65.535
`signed`	32 Bits	-2^{31}	...	$2^{31}-1$	=	−2.147.483.648 ... 2.147.483.647
`unsigned`	32 Bits	0	...	$2^{32}-1$	=	0 ... 4.294.967.295
`signed`	64 Bits	-2^{63}	...	$2^{63}-1$	=	−9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807
`unsigned`	64 Bits	0	...	$2^{64}-1$	=	0 ... 18.446.744.073.709.551.615

Die in Ihrem C++-System zutreffenden Zahlenbereiche finden Sie im Header `<limits>`. Das folgende Programm gibt die Grenzwerte und den benötigten Speicherplatz für die Ganzzahl-Typen aus.

Listing 1.4: Grenzwerte und Speicherplatzbedarf *(cppbuch/k1/intlimits.cpp)*

```
#include <cstddef>                          // size_t, ptrdiff_t
#include <iostream>
#include <limits>                           // hier sind die Bereichsinformationen
using namespace std;

int main()
{
  cout << "Grenzwerte␣für␣Ganzzahl-Typen:" << '\n';          // = neue Zeile
  cout << "int␣Minimum␣=␣␣␣␣␣␣␣" << numeric_limits<int>::min() << '\n';
  cout << "int␣Maximum␣=␣␣␣␣␣␣␣␣" << numeric_limits<int>::max() << '\n';
  cout << "long␣Minimum␣=␣␣␣␣␣␣" << numeric_limits<long>::min() << '\n';
  cout << "long␣Maximum␣=␣␣␣␣␣␣␣" << numeric_limits<long>::max() << '\n';
  cout << "long␣long␣Minimum␣=␣␣" << numeric_limits<long long>::min() << '\n';
  cout << "long␣long␣Maximum␣=␣␣␣" << numeric_limits<long long>::max() << '\n';
  cout << "ptrdiff_t␣Minimum␣=␣␣"  << numeric_limits<ptrdiff_t>::min() << '\n';
  cout << "ptrdiff_t␣Maximum␣=␣␣␣"  << numeric_limits<ptrdiff_t>::max() << '\n';
  cout << "unsigned-Maxima␣(Minimum␣ist␣0):\n";          // ebenfalls neue Zeile
```

```
  cout << "unsigned␣int␣␣=␣␣␣␣␣␣"
       << numeric_limits<unsigned int>::max() << '\n';
  cout << "unsigned␣long␣=␣␣␣␣␣␣"
       << numeric_limits<unsigned long>::max() << '\n';
  cout << "unsigned␣long␣long␣=␣"
       << numeric_limits<unsigned long long>::max() << '\n';
  cout << "size_t␣␣␣␣␣␣␣␣␣␣␣␣=␣"  << numeric_limits<size_t>::max() << '\n';
  cout << "Anzahl␣der␣Bytes␣für:\n";
  cout << "int␣␣␣␣␣␣␣" << sizeof(int) << '\n';
  cout << "long␣␣␣␣␣␣" << sizeof(long) << '\n';
  cout << "long␣long␣" << sizeof(long long) << '\n';
  cout << "size_t␣␣␣␣" << sizeof(size_t) << '\n';
  cout << "ptrdiff_t␣" << sizeof(ptrdiff_t) << '\n';
}
```

Übung

1.5 Erweitern Sie das Programm, sodass auch die Anzahl der Bytes für den Datentyp `short` ausgegeben wird. Lassen Sie auch die Anzahl der Bytes für die `unsigned`-Varianten der Zahltypen ausgeben. Empfehlung, nicht nur für dieses Programm: Probieren Sie *alle* Beispiele selbst aus! Variieren Sie sie und vergleichen Sie die Ergebnisse.

Ganze Zahlen können auf verschiedene Arten dargestellt werden:

1. Die übliche Darstellung ist die als Dezimalzahl. Eine Dezimalzahl, die ungleich 0 ist, beginnt mit einer der Ziffern im Bereich 1 bis 9. Ein Suffix (l, ll, L, LL) kennzeichnet `long`- oder `long long`-Zahlen, etwa 2147483647L oder 9223372036854775806LL. Ein mögliches Vorzeichen gehört nicht zur Syntax der Zahl, obwohl es in einem Programm oder beim Einlesen mit `cin` natürlich berücksichtigt wird.
2. Ein Suffix u oder U kennzeichnet `unsigned`-Zahlen, zum Beispiel 1836u. Kombinationen wie 2036854775806ul sind möglich.
3. Ein Suffix uz (Großschreibung erlaubt) kennzeichnet Zahlen vom Typ `size_t`, zum Beispiel 1836uz. Wenn nur z angehängt wird, also etwa 1836z, ist das Ergebnis die entsprechende ganze Zahl, also mit Vorzeichen. Der Typ dieser vorzeichenbehafteten Zahl ist meistens `ptrdiff_t`, das wird aber nicht ausdrücklich vom Standard vorgeschrieben.
4. Wenn eine Zahl mit 0b oder 0B beginnt, wird sie als *Binärzahl* interpretiert, etwa 0b1011 = $1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 11$ dezimal.
5. Wenn eine Zahl mit einer 0 beginnt, wird sie als *Oktalzahl* interpretiert, zum Beispiel 0377 = $377_8 = 3*8^2 + 7*8^1 + 7*8^0 = 255_{10}$ (dezimal).
6. Wenn eine Zahl mit 0x oder 0X beginnt, wird sie als *Hexadezimalzahl* interpretiert, zum Beispiel 0xAFFE = $10*16^3 + 15*16^2 + 15*16^1 + 14*16^0 = 45054_{10}$ (dezimal).
7. Bei allen Zahlen ist es zur besseren Lesbarkeit erlaubt, zur Gruppierung Hochkommas einzufügen, wie etwa 1'000'000 statt 1000000 oder 0xabc'def statt 0xabcdef. Bei der Auswertung der Zahl durch den Compiler werden die Hochkommas ignoriert. Mehr zeigt das Programm *cppbuch/k1/zahltypen.cpp* (hier nicht abgedruckt).

Beim Rechnen mit ganzen Zahlen ist der *begrenzte Wertebereich* zu beachten! Aufgrund der Tatsache, dass nur eine begrenzte Anzahl von Bits für die rechnerinterne Repräsen-

tation einer Zahl zur Verfügung steht, ergibt sich, dass der von `int` abgedeckte Zahlenbereich nur eine *Untermenge* der ganzen Zahlen darstellt, wie das folgende Programmbeispiel zeigt. Im Programm wird unter *Initialisierung* die Belegung einer Variable mit einem Anfangswert verstanden.

Listing 1.5: Arithmetischer Überlauf (*cppbuch/k1/overflow.cpp*)

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

int main()
{
  int ai{50000};                    // Initialisierung mit 50000
  int bi{1'000'000};           // Initialisierung mit 1000000, mit Trennzeichen
  int ci{ai * bi};
  cout << "int-Zahlen haben auf Ihrem System " << 8 * sizeof(int) << " Bits\n";
  cout << "Rechnung mit int: ";
  cout << ai << " * " << bi << " = " << ci << '\n';
  // Ausgabe -1539607552 statt 50000000000 bei 32 Bit-int
  long al{50000L};           // L (oder l): Kennzeichnung als long
  long bl{1000000L};
  long cl{al * bl};
  cout << "long-Zahlen haben auf Ihrem System " << 8 *sizeof(long) << " Bits\n";
  cout << "Rechnung mit long: ";
  cout << al << " * " << bl << " = " << cl << '\n';
  cout << "long-Overflow produzieren:\n";
  al = numeric_limits<long>::max() / 2 + 1000;
  cout << "2 * " << al << " = " << (2 * al) << " ?\n";
}
```

Aus dem Programmergebnis folgt, dass die Regeln der Mathematik in der Nähe der Grenzen des Zahlenintervalls nur noch eingeschränkt gelten. Das Ergebnis einer Folge arithmetischer Operationen ist nur dann korrekt, wenn kein Zwischenergebnis den durch den Datentyp vorgegebenen maximalen Zahlenbereich überschreitet. 50000000000 liegt außerhalb des durch 32 Bits darstellbaren Zahlenbereichs. Wird das Resultat einer Operation betragsmäßig zu groß, liegt ein *Überlauf* (englisch *overflow*) vor, der vom Computer *nicht* gemeldet wird. Der Ersatz von `int` durch `long` führt im obigen Beispiel nur dann zu einem Programm, das das korrekte Ergebnis ausgibt, falls `long` mehr Bits als `int` hat. Dies ist oft nicht der Fall. Das Problem ist nicht grundsätzlich lösbar; es wird bei Ganzzahl-Datentypen mit mehr Bits pro Zahl nur in Richtung größerer Zahlen verschoben.

Beim Schreiben eines Programms müssen Sie sich also Gedanken über die möglichen vorkommenden Zahlenwerte machen. »Sicherheitshalber« immer den größtmöglichen Datentyp zu wählen, ist nicht sinnvoll, weil Variablen dieses Typs mehr Speicherplatz benötigen und Rechenoperationen mit ihnen länger dauern können.

Initialisierung von ganzen Zahlen

Initialisierung heißt, einer Variablen schon bei der Deklaration einen Anfangswert zuzuweisen. Dabei gibt es verschiedene Möglichkeiten.

```
int var0 {1};                    // mit {}, kennen Sie von oben
int var1 = 1;                    // andere Möglichkeit
int var2 {};                     // var2 ist Null
auto var3 {1};                   // var3 ist int
auto var4 = 1;                   // var4 ist int
auto var5 = 1u;                  // var5 ist unsigned int
```

Bei der Deklaration mit `auto` wird der Typ des Werts auf der rechten Seite genommen. Je nach Art der Initialisierung ist das Verhalten des Compilers unterschiedlich, wenn der Datentyp links und rechts nicht übereinstimmt:

```
int wert1 = 9.42;                // keine Fehlermeldung, wert1 wird 9
int wert2 {9.42};                // Fehler
int wert3 {wert2 * 1.23};        // Warnung
```

Im ersten Fall gibt es einen möglicherweise unbemerkten Genauigkeitsverlust, im zweiten Fall eine Meldung des Compilers, weil 9.42 als Kommazahl keine `int`-Zahl ist und damit ein Genauigkeitsverlust eintritt. Deswegen ist die Initialisierung mit geschweiften Klammern vorzuziehen. Bei nicht direkt angegebenen Werten (Literalen), sondern berechneten Werten, gibt es nur eine Warnung. Die gezeigten Initialisierungsarten gelten sinngemäß auch für andere einfache Datentypen. Die Kombination von = mit geschweiften Klammern ist vom Compiler erlaubt, ergibt hier aber nicht das gewünschte Ergebnis:

```
auto wert4 = {1};                // = : Tun Sie es nicht!
int kopie = wert4;               // Folgefehler
```

Der Grund: `wert4` ist kein `int`, sondern eine sogenannte Initialisierungsliste, auf die erst später eingegangen werden kann.

Empfehlung

Um einen gewünschten Typ klar deutlich zu machen, geben Sie bei der Initialisierung von Objekten den Typ direkt an, zum Beispiel `int`. In fast allen anderen Fällen können Sie `auto` in Verbindung mit dem Initialisierungswert in geschweiften Klammern nehmen. Die Verwendung von `auto` hat den Vorteil, dass sie eine Initialisierung erzwingt, weil der Compiler den Typ nur aus dem Initialisierungswert ermitteln kann.

```
auto wert5;          // Fehler! Der Compiler kann den Typ nicht bestimmen.
auto wert6{9};       // Der Typ ist int.
auto wert7{9u};      // Der Typ ist unsigned int.
auto wert8{9l};      // Der Typ ist long.
```

Initialisierung oder Zuweisung?

Eine *Initialisierung* weist einem Objekt beim Anlegen des Objekts einen Anfangswert zu. Eine *Zuweisung* jedoch bezieht sich immer auf ein bereits existierendes Objekt.

```
int a = 1;                       // Initialisierung
int b;                           // nicht initialisiert (d.h. undefinierter Wert)!
b = 3;                           // Zuweisung (b existiert bereits)
```

Eine Variable sollte initialisiert werden, auch wenn sie berechnet oder eingelesen wird. Das erleichtert eine mit der Variablen verbundene Fehlersuche.

Operatoren für ganze Zahlen

Tabelle 1.1 zeigt die Operatoren für ganze Zahlen. Die zusammengesetzten Operatoren wie += heißen Kurzform-Operatoren.

Tabelle 1.1: Operatoren für Ganzzahlen

Operator	Beispiel	Bedeutung
Arithmetische Operatoren:		
+	+i	unäres Plus (kann weggelassen werden)
-	-i	unäres Minus
++	++i	vorherige Inkrementierung um eins
	i++	folgende Inkrementierung um eins
--	--i	vorherige Dekrementierung um eins
	i--	folgende Dekrementierung um eins
+	i + 2	binäres Plus
-	i - 5	binäres Minus
*	5 * i	Multiplikation
/	i / 6	Division
%	i % 4	Modulo (Rest mit Vorzeichen von i)
=	i = 3 + j	Zuweisung
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3
%=	i %= 3	i = i % 3
+=	i += 3	i = i + 3
-=	i -= 3	i = i - 3
relationale Operatoren:		
<	i < j	kleiner als
>	i > j	größer als
<=	i <= j	kleiner gleich
>=	i >= j	größer gleich
==	i == j	gleich
!=	i != j	ungleich
<=>	(i <=> j) > 0	3-Wege-Vergleich, Erklärung in Abschnitt 8.7
Bit-Operatoren:		
<<	i << 2	Linksschieben (Multiplikation mit 2er-Potenzen)
>>	i >> 1	Rechtsschieben (Division durch 2er-Potenzen)
&	i & 7	bitweises UND
^	i ^ 7	bitweises XOR (Exklusives Oder)
|	i | 7	bitweises ODER
~	~i	bitweise Negation
<<=	i <<= 3	i = i << 3
>>=	i >>= 3	i = i >> 3
&=	i &= 3	i = i & 3
|=	i |= 3	i = i | 3

Auf Daten eines bestimmten Typs können nur bestimmte Operationen durchgeführt werden. Eine Zeichenkette kann man zum Beispiel nicht durch eine andere dividieren, sehr wohl jedoch eine ganze Zahl durch eine andere $\neq 0$. Daraus folgt: Ein Datum und die zugehörigen Operationen gehören zusammen!

Einige der Operatoren aus der Tabelle 1.1 werden durch Beispiele erläutert. Manche Operatoren setzen jedoch hier noch nicht besprochene Dinge voraus, weshalb gelegentlich auf spätere Abschnitte verwiesen wird. Beispiel für die Anwendung des ++-Operators:

```
int a;                                  // undefinierter Anfangswert
int i {5};                              // Anfangswert 5
a = ++i;                                // vorangestelltes ++
```

`i` wird *erst* um eins *inkrementiert*, *dann benutzt*. Unter Inkrementierung wird die Addition von 1 verstanden, unter Dekrementierung die Subtraktion von 1. ++ ist der Inkrementierungsoperator, der je nach Stellung eine Variable vor oder nach Benutzung des Werts hochzählt. Nach dieser Anweisung haben sowohl `a` als auch `i` den Wert 6.

```
int j {2};
int b {j++};                            // nachgestelltes ++
```

`j` wird *erst benutzt*, *dann inkrementiert*. Nach dieser Anweisung hat `b` den Wert 2 und `j` den Wert 3.

```
j = j + 4;
j += 5;
```

Hier wird 4 zu `j` addiert. Ergebnis: `j` ist gleich 7. Anschließend wird 5 mit dem Kurzformoperator hinzugefügt. `j` hat danach den Wert 12.

Ganzzahlige Division

Wenn bei der Division nur ganze Zahlen beteiligt sind, ist das Ergebnis auch ganzzahlig! Der Rest wird verworfen und bei Bedarf mit dem Modulo-Operator `%` ermittelt:

```
int m {9};
int n {5};
int ergebnis { m / n};                  // 1
int rest {m % n};                       // 4
```

Übungen

1.6 Schreiben Sie ein Programm, das den Unterschied zwischen dem vor- und dem nachgestellten ++ demonstriert.

1.7 Es gelte `int a = 5;` und `int b = 7;`. Was ergibt die Division `a/b`?

Bit-Operatoren

Weil ganze Zahlen als Bitvektoren aufgefasst werden können, sind zusätzlich Bit-Operationen möglich. Es folgen Beispiele für zwei `int`-Zahlen `c` und `k`. Die Zahl `k` soll den Wert 5 repräsentieren. Die binäre Darstellung der Zahl 5 ist für 16-Bit-`int`-Zahlen `0000 0000 0000 0101`. Die Anweisung `c = k << 2;` bewirkt eine Bitverschiebung um 2 Stellen nach links (= Multiplikation mit 2^2, also 4) und Zuweisung des Ergebnisses an `c`.

```
0000 0000 0000 0101    k ist gleich 5
0000 0000 0001 0100    c = k << 2, d.h. 2 Stellen verschoben. c hat den Wert 20.
```

Bei der Verschiebung nach links werden von rechts Nullen nachgezogen. Wenn nach rechts verschoben wird, werden links Nullen eingefügt, falls der Operand vom Typ `unsigned` ist. Bei vorzeichenbehafteten Typen wird links entweder das Vorzeichenbit kopiert oder es werden Nullbits eingefügt – je nach C++-System. Die Anweisung `c = c & k;` bewirkt die bitweise UND-Verknüpfung:

```
0000 0000 0000 0101   k ist gleich 5
0000 0000 0001 0100   c ist gleich 20
0000 0000 0000 0100   Das Ergebnis von c & k ist 4.
```

Die Anweisung `c = ~k;` bewirkt die bitweise Negation:

```
0000 0000 0000 0101   k ist gleich 5
1111 1111 1111 1010   c = ~k (also –6)
```

Die Addition von 1 auf `c` ergibt das Zweierkomplement, also die Darstellung der negativen Zahl -5.

Weitere Typen für ganze Zahlen[5]

Im Header `<cstdint>` gibt es weitere Typen für ganze Zahlen. Sie sind weniger gebräuchlich und werden in diesem Buch daher nicht weiter benutzt. Sie sollen nur wissen, dass es sie gibt, und im Zweifel die Bedeutung nachschlagen können.

- `int_fast8_t` `int_fast16_t` `int_fast32_t` `int_fast64_t`
  `uint_fast8_t` `uint_fast16_t` `uint_fast32_t` `uint_fast64_t`
  Jeder dieser Typen ist normalerweise der schnellste für `int`-Operationen. Die Zahl gibt an, wie viele Bits mindestens für den Typ zur Verfügung stehen. So ist `uint_fast32_t` ein `unsigned int`-Typ mit mindestens 32 Bits.
- `int_least8_t` `int_least16_t` `int_least32_t` `int_least64_t`
  `uint_least8_t` `uint_least16_t` `uint_least32_t` `uint_least64_t`
  Jeder dieser Typen hat mindestens die angegebene Bitbreite.
- `intmax_t` `uintmax_t`
  Diese Typen sind `int` bzw. `unsigned int`-Typen mit der maximal zur Verfügung gestellten Bitbreite. Sie kann größer sein als die von `size_t`.

Die oben genannten Typen *müssen* laut [ISOC++] vorhanden sein. Es kann jedoch optional weitere Typen geben:

- `int8_t` `int16_t` `int32_t` `int64_t`
  `uint8_t` `uint16_t` `uint32_t` `uint64_t`
  Jeder dieser Typen hat *exakt* die angegebene Bitbreite.
- `intptr_t` `uintptr_t`
  Jeder gültige Zeiger auf `void` kann in einen dieser Typen umgewandelt werden. Die Rückumwandlung ergibt wieder den ursprünglichen Zeiger.

Mit Hilfe des ggf. zu modifizierenden Programms *cppbuch/k1/inttypen.cpp* können Sie später (sobald Sie wissen, was `static_assert` bedeutet) prüfen, welche Typen auf Ihrem System vorhanden sind.

[5] Dieser Abschnitt (bis »Reelle Zahlen«) kann wegen des Bezugs auf folgende Kapitel beim ersten Lesen übersprungen werden.

1.6.4 Reelle Zahlen

Reelle Zahlen, auch Gleitkommazahlen oder Fließkommazahlen genannt, sind wegen der beschränkten Bit-Anzahl in der Regel nicht beliebig genau darstellbar. An die Stelle des Kommas tritt ein Dezimalpunkt, wie im angelsächsischen Sprachraum üblich. Reelle Zahlen werden in C++ wie folgt geschrieben:

- Übliche Schreibweise
  Vorzeichen (optional), Vorkommastellen, Dezimalpunkt, Nachkommastellen, `e` oder `E` und Ganzzahl-Exponent (optional), Suffix `f`, `F` oder `l`, `L` (für `long double`) Zahlen ohne Suffix sind `double`. Es können wegfallen:
  - entweder Vorkommastellen oder Nachkommastellen (aber nicht beide) oder
  - entweder Dezimalpunkt oder e (oder E) mit Exponent (aber nicht beide).

  Einige Beispiele für Gleitkommazahlen sind:
  -123.789e6f 1.8E6 88.009 1e-03 1.8L
  Der Exponent meint Zehnerpotenzen. 1.8e6 ist dasselbe wie $1.8 \cdot 10^6$ oder 1800000. Reelle Zahlen werden durch die drei Datentypen der Tabelle 1.2 (ungenau) dargestellt.

Tabelle 1.2: Datentypen und Bereiche für reelle Zahlen

Typ	Bits	Zahlenbereich			Genauigkeit in Dezimalstellen
`float`	32	$\pm$ 1.2*10^{-38}	...	$\pm$ 3.4*10^{38}	ca. 6
`double`	64	$\pm$ 2.2*10^{-308}	...	$\pm$1.8*10^{308}	ca. 15
`long double`	80	$\pm$ 3.4*10^{-4932}	...	$\pm$1.2*10^{4932}	ca. 18

- Hexadezimale Schreibweise[6]
  Diese Schreibweise wird nur in den seltenen Fällen gebraucht, in denen das Bitmuster der Zahl exakt vorgegeben werden soll. Die Schreibweise wurde aus dem Standard für die Programmiersprache C [ISOC] übernommen. So ist die hexadezimale Zahl 0xC.68p+2f dasselbe wie die `float`-Zahl 49.625. Die Interpretation:
  »f« bedeutet `float`. 0xC.68 ist dasselbe wie 0xC*16^0 + 0x6*16^{-1} + 0x8*16^{-2}, d.h. 12 + 6/16 + 8/256 = 12.40625. Die Zahl nach dem »p« ist die 2er-Potenz, mit der der Teil davor multipliziert werden soll, also $2^2 = 4$. Das Zwischenergebnis 12.40625 mit 4 malgenommen ergibt 49.625. Test: `std::cout << 0xC.68p+2f << '\n';`

Für Gleitkommazahlen sind die meisten Operationen der Tabelle 1.1 (Seite 45) ebenfalls möglich. Ausgenommen sind der Modulo-Operator und die Bit-Operatoren. Die Festlegungen der Minimal- bzw. Maximal-Werte Ihres C++-Systems sind im Header `<limits>` zu finden. Die `numeric_limits<float>`- bzw. `numeric_limits<double>`-Funktionen können Auskunft geben, wie das folgende Programm zeigt:

Listing 1.6: Grenzwerte von `float`- und `double`-Zahlen (Auszug aus *cppbuch/k1/floatlimits.cpp*)

```
#include<iostream>
#include<limits>            // hier sind die Bereichsinformationen
using namespace std;

int main()
{
  cout << "Grenzwerte für Float-Zahl-Typen:\n";
```

[6] Beim ersten Lesen überspringen.

```
  cout << "Float-Min:␣"  << numeric_limits<float>::min() << '\n';
  cout << "Float-Max:␣"  << numeric_limits<float>::max() << '\n';
  cout << "Double-Min:␣"  << numeric_limits<double>::min() << '\n';
  cout << "Double-Max:␣"  << numeric_limits<double>::max() << '\n';
  cout << "Long-Double-Min:␣" << numeric_limits<long double>::min() << '\n';
  cout << "Long-Double-Max:␣" << numeric_limits<long double>::max() << '\n';
  cout << "float-␣und␣double-Zahlen␣entsprechen";
  if (!numeric_limits<float>::is_iec559) {
    cout << "␣nicht";
  }
  cout << "␣dem␣IEC␣559␣(=IEEE␣754)-Standard.\n";
}
```

Intern werden Mantisse und Exponent jeweils durch Binärzahlen einer bestimmten Bitbreite verkörpert. Die hier angegebenen für Mantisse und Exponent aufsummierten Bitbreiten und damit Zahlenbereiche und Rechengenauigkeiten sind *implementationsabhängig* und dienen als Beispiel. Meistens entsprechen die Zahlen dem Standard IEC559 (=IEEE 754). Ob das auf Ihrem System so ist, können Sie mithilfe des obigen Programms herausfinden. Tabelle 1.3 zeigt ein Beispiel der Repräsentation reeller Zahlen im Rechner.

Tabelle 1.3: Anzahl der Bits (Beispiel)

	float	double	long double
Vorzeichen	1	1	1
Mantisse	23	52	64
Exponent	8	11	15
Gesamtanzahl Bits	32	64	80

Der Exponent ist eine `unsigned`-Zahl mit einem Offset, um negative Exponenten darstellen zu können. Zum Beispiel entsprechen die 11 Bits bei `double` einem Bereich von 0 bis 2047, der auf Exponenten von -1022 bis 1023 abgebildet wird (-1023 und 1024 sind für spezielle Zahlen reserviert). In C++ ist festgelegt, dass die Genauigkeit von `double`-Zahlen nicht schlechter sein darf als die von `float`-Zahlen, und die von `long double`-Zahlen[7] darf nicht schlechter sein als die von `double`-Zahlen. Die Genauigkeit hängt von der Anzahl der Bits ab, die für die Mantisse verwendet werden. Es sei angenommen, dass für den Typ `double` die oben angegebenen Bitzahlen gelten. Da 2^{52} ca. $4.5 \cdot 10^{15}$ ist, ergibt sich eine etwa 15- bis 16-stellige Genauigkeit. Sie können sich die Genauigkeit für die verschiedenen Datentypen direkt ausgeben lassen:

Listing 1.7: Genauigkeiten von `float`- und `double`-Zahlen *(cppbuch/k1/genauigkeiten.cpp)*

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

int main()
{
```

[7] `sizeof(long double)` kann größer als 10 (= 80 Bits) sein, etwa 12 oder 16, um ein Vielfaches der Länge eines Rechnerworts (32 oder 64 Bits) zu erreichen. Interne Operationen sind einfacher zu gestalten, wenn Datentypen auf Wortgrenzen enden. Wenn nur mit 80 Bits gerechnet wird, werden also Bits »verschenkt«.

```
  cout << "Die␣Genauigkeit␣von␣float␣beträgt␣etwa␣"
       << numeric_limits<float>::digits10 << "␣Dezimalstellen.\n";
  cout << "Die␣Genauigkeit␣von␣double␣beträgt␣etwa␣"
       << numeric_limits<double>::digits10 << "␣Dezimalstellen.\n";
  cout << "Die␣Genauigkeit␣von␣long␣double␣beträgt␣etwa␣"
       << numeric_limits<long double>::digits10 << "␣Dezimalstellen.\n";
}
```

Auf meinem System lautet die Ausgabe des Programms:

```
Die Genauigkeit von float beträgt etwa 6 Dezimalstellen.
Die Genauigkeit von double beträgt etwa 15 Dezimalstellen.
Die Genauigkeit von long double beträgt etwa 18 Dezimalstellen.
```

Der Zahlenbereich wird wesentlich durch die Anzahl der Bits für den Exponenten bestimmt, der Einfluss der Mantisse ist minimal. Entsprechend Tabelle 1.3 können `double`-Zahlen 10 Bits für den Exponenten haben. 2^{10} ist 1024. Der Exponent kann damit im Bereich von 0 bis $2^{10} - 1$ (d.h. 0 bis 1023) liegen. Übertragen auf das Dezimalsystem ergibt sich mithilfe der Schulmathematik ein maximaler Exponent von $1023 \cdot \log 2 / \log 10$, also etwa 308. Der darstellbare Bereich geht also bis ca. 10^{308}. Insgesamt ergibt sich, dass eine beliebige Genauigkeit nicht für alle Zahlen möglich ist. Falls 32 Bits für die Darstellung einer reellen Zahl verwendet werden, existieren nur 2^{32} = 4 294 967 296 verschiedene Möglichkeiten, eine Zahl zu bilden.

Mit dem mathematischen Begriff eines reellen Zahlenkontinuums hat das nur näherungsweise zu tun, und alle Illusionen von der computertypischen Genauigkeit und Korrektheit sind dahin, wie das folgende Programm demonstriert.

Listing 1.8: Fehler bei Rechnung mit `float`-Zahlen *(cppbuch/k1/ungenau.cpp)*

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

int main()
{
  float a{1.234567E-7f};
  float b{1.000000f};
  float c{-b};
  cout << "Ungenauigkeit␣bei␣float-Arithmetik:\n";
  cout << "(a+b)+c␣=␣" << (a + b) + c << '\n';          // 1.19209e-7
  cout << "a+(b+c)␣=␣" << a + (b + c) << '\n';          // 1.23457e-7
}
```

Die Ausgabe von '\n' im Beispielprogramm bedeutet, dass jeweils eine neue Zeile auf dem Bildschirm begonnen wird. Die nicht exakte interne Darstellung kann verschiedene Folgen haben:

- Bei der Subtraktion zweier fast gleich großer Werte heben sich die signifikanten Ziffern auf. Die Differenz wird damit ungenau. Dieser Effekt ist unter dem Namen *numerische Auslöschung* bekannt.
- Die Division durch einen betragsmäßig zu kleinen Wert ergibt einen *Überlauf* (englisch *overflow*).

- Eine *Unterschreitung* (englisch *underflow*) tritt auf, wenn der Betrag des Ergebnisses zu klein ist, um mit dem gegebenen Datentyp dargestellt zu werden. Das Resultat wird dann gleich 0 gesetzt.
- Ergebnisse können von der Reihenfolge der Berechnungen abhängen. Im Programmbeispiel wird `a+b+c` auf zwei verschiedene Arten berechnet: Zuerst werden `a` und `b` addiert und danach `c`, während im zweiten Fall zu `a` die Summe von `b` und `c` addiert wird. Die Auswertungsreihenfolge wird durch die Klammerung bestimmt. Mathematisch müsste `((a+b)+c)` das Gleiche sein wie `(a+(b+c))`. Die Computerarithmetik liefert jedoch abweichende Ergebnisse. Daher gehört zu kritischen Rechnungen immer eine Genauigkeitsbetrachtung.

Es gibt verschiedene Methoden, den Fehler bei ungenauen Rechnungen zu minimieren. Ein Beispiel: Bei der Addition einer großen Menge verschiedener Zahlen werden zunächst die Zahlen sortiert. Erst danach folgen die Additionen, beginnend mit der kleinsten Zahl.

Für die »reellen« Zahlentypen `float`, `double` und `long double` stehen mit Ausnahme des Modulo-Operators `%` alle arithmetischen und relationalen Operatoren der Tabelle 1.1 zur Verfügung. Bei zu großer Ungenauigkeit von `float`-Rechnungen könnte man also `double`-Zahlen nehmen, die etwas mehr Speicher kosten. Mit diesen wird das Genauigkeitsproblem nicht grundsätzlich gelöst, die Genauigkeit ist jedoch größer.

Einfache mathematische Ausdrücke und Funktionen

Tabelle 1.4 zeigt einige Beispiele zur Umsetzung mathematischer Ausdrücke in die Programmiersprache C++.

Tabelle 1.4: Umsetzen mathematischer Ausdrücke

Mathematische Schreibweise	C++-Notation	Hinweise
$a - \sqrt{1 + x^2}$	`a - sqrt(1+x*x)`	
$\frac{a-b}{1+\frac{x}{y}}$	`(a-b)/(1+x/y)`	

$\lvert x \rvert$	`abs(x)`
	`labs(x)`
	`fabs(x)`
		`int`
		`long int`
		`float`, `double`, `long double`

$\frac{\frac{x}{y}}{z}$	`x/y/z`	klarer: `(x/y)/z`
$e^{-\delta t}\ \sin(\omega t + \varphi)$	`exp(-delta*t)*sin(omega*t+phi)`	

Etliche mathematische Funktionen sind vordefiniert. Dazu gehören einfache Funktionen wie `sqrt()`, `exp()` und `log()`, aber auch Funktionen, für die sich wohl nur Mathematiker interessieren, wie etwa Laguerre-Polynome, Elliptische Integrale und Legendre-Funktionen. Die Deklaration der Funktionen befindet sich im Header `<cmath>`. `abs()` ist auch im Header `<cstdlib>` deklariert. Zum Nachschlagen einer Auswahl der einfacheren Funktionen bieten sich die Tabellen 33.3 und 33.5 an (ab Seite 953). Um dem Compiler die Deklarationen bekannt zu machen, genügt es, im Programm am Anfang der Programmdatei die Zeile `#include <cmath>` einzufügen, wie unten im Programm gezeigt.

Listing 1.9: Auswahl mathematischer Funktionen *(cppbuch/k1/mathexpr.cpp)*

```
#include <cmath>
#include <iostream>
using namespace std;
int main()                              // Berechnung mathematischer Ausdrücke
{
  double x {0.0};                       // Initialisierung hier nicht unbedingt notwendig
  cout << "x␣eingeben:";
  cin >> x;
  cout << "␣x␣␣␣␣␣␣=␣" << x << '\n';
  cout << "␣fabs(x)␣=␣" << fabs(x) << '\n';
  cout << "␣sqrt(x)␣=␣" << sqrt(x) << '\n';
  cout << "␣sin(x)␣␣=␣" << sin(x) << '\n';    // Argument von sin() im Bogenmaß!
  cout << "␣exp(x)␣␣=␣" << exp(x) << '\n';
  cout << "␣log(x)␣␣=␣" << log(x) << '\n';    // log() ist der natürliche Logarithmus
}
```

Übung

1.8 Probieren Sie das vorstehende Programm aus! Wie verhält sich Ihr Rechner bei Eingabe von 0 oder einer negativen Zahl? (Dies ist eine von etwa sechs Übungsaufgaben, die mathematisch mehr als die Kenntnis der vier Grundrechenarten voraussetzen – ein kleiner Anteil bei 99 Aufgaben insgesamt.)

float-Zahlen definierter Bitbreite[8]

Wie am Ende des vorherigen Abschnitts gesehen, gibt es `int`-Zahlen mit definierter Bitbreite. Seit C++23 gibt es auch entsprechende `float`-Zahlen. Die folgenden Zeilen zeigen die Datentypen und auch die passenden Suffixe.

```
std::float16_t  a16 = 3.141f16;
std::float32_t  a32 = 3.141f32;
std::float64_t  a64 = 3.141f64;
std::float128_t a128 = 3.141f128;
std::bfloat16_t ab16 = 3.141bf16;
```

Die Einbindung des Headers `<stdfloat>` ist Voraussetzung. Die Typen `float16_t` und `bfloat16_t` haben dieselbe Bitbreite. `bfloat16_t` hat aber mehr Bits für den Exponenten und dafür weniger Mantissenbits [bfloat].

1.6.5 Konstanten

In einem Programm kommen häufig Zahlen oder andere Datenstrukturen vor, die im Programmlauf nicht verändert werden dürfen. Sie heißen *Konstanten*. Zum Beispiel könnte man »`umfang = 3.1415926 * durchmesser;`« schreiben. Besser und etwas genauer ist

```
const double pi {3.1415926535897932};
umfang = pi * durchmesser;
```

weil bei anschließendem häufigerem Gebrauch der Zahl `pi` Schreibfehler leicht ausgeschlossen werden können und Änderungen oder Korrekturen einer Konstante nur an

[8] Dieser Abschnitt (bis »Konstanten«) kann beim ersten Lesen übersprungen werden.

einer Stelle vorgenommen werden müssen (siehe Stichwort »magic number« im Glossar). `pi` ist nur einmal als Konstante zu vereinbaren (= zu deklarieren). `double` bedeutet, dass die Konstante eine Gleitkommazahl ist. Um Schreibfehler zu vermeiden, ist es noch besser, die vordefinierte Konstante `pi` zu nehmen:

```
#include <numbers>                          // Hier ist pi definiert.
// ...
umfang = numbers::pi * durchmesser;
```

- Eine Konstante besteht aus einem Namen und dem zugeordneten Wert, der *nicht veränderbar* ist.
- Eines der Schlüsselwörter `const` oder `constexpr` leitet die Deklaration einer Konstanten ein. Arithmetik ist erlaubt, z.B. `const int max {groesse-1};`
- Das Schlüsselwort `const` wird verwendet, wenn der Wert nicht unbedingt schon zur Compilationszeit bestimmt werden kann. Beispiel:

```
int a;
cin >> a;                   // a eingeben
const int b {a};            // b hat den Wert von a, kann aber nicht verändert werden.
b = 99;                     // Fehlermeldung!
```

Eine Zahl auf der rechten Seite zur Initialisierung einer Konstante heißt »Zahlliteral«, weil sie genau so in den Programmcode geschrieben wird. Die Auswertung geschieht nicht erst, wenn das Programm läuft, sondern schon durch den Compiler. Ein Literal für eine ganze Zahl besteht nur aus Ziffern und möglicherweise einem Suffix, um den Typ zu spezifizieren, zum Beispiel `l` oder `L` für `long`-Zahlen. Ein vorangestelltes '-'-Zeichen macht daraus eine negative Zahl.

- Das Schlüsselwort `constexpr` wird verwendet, wenn der Wert zur Compilationszeit bestimmt werden kann. Das ist bei Literalen der Fall.

constexpr und const häufig verwenden!

Größen, deren Änderung nicht beabsichtigt ist, sollten *stets* als `const` oder, wenn möglich, als `constexpr` deklariert werden! Der Compiler wird damit in die Lage versetzt, fehlerhafte Zuweisungen zu finden.

Konstanten unterscheiden sich syntaktisch von Variablen nur durch die Qualifizierung mit `const`. »Syntaktisch« heißt »die Syntax betreffend«, also die Grammatik der Programmiersprache. Einige Beispiele für Konstanten:

```
const double  px {45.99};                   // constexpr ist hier auch möglich
constexpr unsigned int anzahl {1000};
// weiterer Programmtext ...
px = 17.5;                                  // Fehlermeldung des Compilers!
anzahl = 10;                                // Fehlermeldung des Compilers!
int a {3};                                  // nicht konstant
a = 5;                                      // a ändert sich
const int b {a};                            // ok, b ist konstant
constexpr int c {a};                        // Fehler! nicht aus Literal ableitbar
constexpr int d {3};                        // ok, Literal
const int e {4};                            // ok, e ist konstant
constexpr int f {e};                        // ok, indirekt aus Literal ableitbar
```

1.6.6 Zeichen

Zeichen sind in diesem Zusammenhang Buchstaben wie A, b, c, D, Ziffernzeichen wie 1, 2, 3 und Sonderzeichen wie ; , . ! , und andere. Dabei werden Zeichenkonstanten (= Zeichenliterale) immer in Hochkommas eingeschlossen, also zum Beispiel 'a', '1', '?' usw. Für Zeichen wird der Datentyp `char` bereitgestellt (Beispieldeklaration siehe einige Zeilen weiter unten). Ein Zeichen ist in diesem Sinne stets auch *nur* ein Zeichen, insbesondere sind Ziffernzeichen etwas anderes als die Ziffern selbst, das heißt '1' ist nicht 1! Ein Zeichen wird intern als 1-Byte-Ganzzahl interpretiert (0 ... 255 `unsigned char` beziehungsweise -128 ... +127 `signed char`). Hier und im Folgenden wird angenommen, dass ein Byte aus 8 Bits besteht. Dies muss nicht für jedes System gelten, ist aber verbreitet. Ein Zeichen gehört zu einem *Zeichensatz*. Die gebräuchlichsten Zeichensätze sind ASCII und UTF-8, wobei ASCII der einfachere ist. Die ASCII-Tabelle definiert die ersten 7 Bits eines Bytes, also 128 Zeichen. Eine Auflistung finden Sie in Abschnitt A.1. Es gibt drei verschiedene `char`-Datentypen:

```
signed char
unsigned char
char    // bedeutet systemabhängig unsigned oder signed
```

Mit `numeric_limits<char>::is_signed` können Sie herausfinden, ob Ihr System Zeichen des Typs `char` als `unsigned` oder `signed` interpretiert:

Listing 1.10: `signed`-Eigenschaft von `char` *(cppbuch/k1/isSigned.cpp)*

```
#include <iostream>
#include <limits>           // hier sind die Bereichsinformationen
using namespace std;

int main()
{
  if (numeric_limits<char>::is_signed) {
    cout << "char wird auf diesem System als signed interpretiert.\n";
  }
  else {
    cout << "char wird auf diesem System als unsigned interpretiert.\n";
  }
  cout << "Grenzwerte für char, in int umgewandelt:\n";
  cout << "Minimum = " << static_cast<int>(numeric_limits<char>::min()) << '\n';
  cout << "Maximum = " << static_cast<int>(numeric_limits<char>::max()) << '\n';
}
```

`static_cast<int>` wandelt den `char`-Wert in eine `int`-Zahl um. Zusätzlich gibt es »lange Zeichen« (englisch *wide characters*), die den Typ `wchar_t` haben. »Wide characters« sind für Zeichensätze gedacht, bei denen ein Byte nicht zur Darstellung eines Zeichens ausreicht, zum Beispiel japanische Zeichen. Ein Zeichenliteral vom Typ `wchar_t` beginnt mit einem `L`, zum Beispiel `L"abc"`. Es gibt auch sogenannte Unicode-Zeichentypen (siehe Glossar, Seite 985), auf die an dieser Stelle nicht weiter eingegangen wird.

Alle Basisfunktionen der C++-Standardbibliothek gelten ebenso für `wchar_t` wie für `char`. Falls nicht ausdrücklich anders erwähnt, wird für `char` im Folgenden stets `signed char` angenommen. Die Art der Voreinstellung variiert von Compiler zu Compiler. Beispiele für Deklarationen und Zuweisungen:

```
const char sternchen {'*'};
char a;
a = 'a';
```

a und 'a' haben hier eine unterschiedliche Bedeutung. a ist eine Variable, die nur dank der Zuweisung den Wert 'a' hat. Ein anderer Wert wäre ebenso möglich:

```
a = 'x';
a = sternchen;
```

Es gibt besondere Zeichenkonstanten der ASCII-Tabelle, die nicht direkt im Druck oder in der Anzeige sichtbar sind. Um sie darstellen zu können, werden sie als Folge zweier Zeichen geschrieben, nehmen aber dennoch ebenfalls nur ein Byte in Anspruch. Diese Zeichen heißen auch *Escape-Sequenzen*, weil \ als Escape-Zeichen dient, um der normalen Interpretation als einzelnes Zeichen zu entkommen (englisch *to escape*). Tabelle 1.5 zeigt einige Beispiele. Auch die Ausgabe von endl bewirkt eine neue Zeile. endl ist allerdings nicht als Zeichenkonstante zu verstehen, weil es zusätzlich für das sofortige Erscheinen der Zeile auf dem Bildschirm sorgt, was bei '\n' durch die gepufferte Ausgabe nicht immer so sein muss (siehe Abschnitt 1.10.1 auf Seite 107).

Tabelle 1.5: Besondere Zeichenkonstanten (Escape-Sequenzen)

Zeichen	Bedeutung	ASCII-Name
\a	Signalton	BEL
\b	Backspace	BS
\f	Seitenvorschub	FF
\n	neue Zeile	LF
\r	Zeilenrücklauf	CR
\t	Tabulator	HT
\v	Zeilensprung	VT
\\	Backslash	
\'	'	
\"	"	
	◇ = Platzhalter:	
\◇	◇ = Folge von Oktalziffern, Beispiel: \377	
\0	Spezialfall davon (Nullbyte)	NUL
\o{◇}	Alternative seit C++23, Beispiel: \o{377}	
\x◇, \X◇	◇ = Folge von Hexadezimalziffern, Beispiel: \xDB oder \xdb	
\x{◇}	Alternative seit C++23, Beispiel: \x{DB} oder \x{db}	

Da ein char-Zeichen genau *ein* Byte beansprucht, ist ein Zeichen der Oktaldarstellung \777 oder \o{777} nicht erlaubt ($\backslash 377 = 255_{10}$). Es sind beliebig viele Oktal- oder Hexadezimalziffern erlaubt, wenn der Zeichentyp sie darstellen kann. Ein Zeichen hat eine eindeutige Position innerhalb der ASCII-Tabelle. Der für eine Programmdatei verwendete Zeichensatz kann ein anderer als ASCII sein, zum Beispiel UTF-8. Der Einfachheit halber bleibe ich im Folgenden bei ASCII, dem am häufigsten verwendeten Zeichensatz. Die Positionen zweier aufeinander folgender Ziffernzeichen liegen genau nacheinander (ohne andere Zeichen dazwischen), was die Umwandlung einer Folge von Ziffernzeichen in eine Zahl erleichtert.

Genau genommen definiert die ASCII-Tabelle nur alle Zeichen mit 7 Bits, insgesamt 128. Der Datentyp `char` stellt 8 Bits, also 1 Byte zur Verfügung, sodass 256 Zeichen darstellbar sind. Die 128 zusätzlichen Zeichen sind nicht genormt, sondern unterscheiden sich für verschiedene Rechner- und Betriebssystemtypen. Meistens werden in diesen 128 Zusatzzeichen Blockgrafiksymbole und nationale Sonderzeichen wie ä, ö, ß untergebracht.

Die Position eines Zeichens in der Tabelle kann über die Umwandlung in eine `int`-Zahl bestimmt werden, ebenso wie aus einer Position über die Umwandlung in `char` das zugehörige Zeichen ermittelt werden kann. Die Umwandlung geschieht einfach über den `static_cast`-Operator mit Angabe des gewünschten Datentyps in spitzen und Angabe der Variable in runden Klammern. Die Typumwandlung heißt in der englischsprachigen Literatur *type cast* oder einfach *cast*. Der `static_cast`-Operator verlangt bestimmte, in [ISOC++] festgelegte Verträglichkeiten zwischen den zu wandelnden Typen.

```
char c {'x'};
int i {static_cast<int>(c)};                    // Typumwandlung char → int
```

bedeutet, dass der Wert der Variablen `i` eine `int`-Repräsentation der `char`-Variablen `c` ist. Andere, einfachere Schreibweisen sind ebenfalls möglich:

```
i = int(c);                                     // oder
i = (int) c;
```

Weil ein `char` vom Compiler als eine 1-Byte-`int`-Zahl interpretiert wird, ist auch eine implizite Typumwandlung möglich:

```
i = c;
```

Normalerweise kann man die einfachere Schreibweise nehmen. Wenn jedoch die Umwandlung im Programm hervorgehoben werden soll, empfiehlt sich `static_cast<...>`. Außerdem können implizite Typumwandlungen im Programmtext leichter übersehen werden als `static_cast<...>`. Hier wird von `int` nach `char` und zurück gewandelt:

```
i = 66;
c = static_cast<char>(i);                       // Typumwandlung int -> char
cout << c;                                      // 'B'
c = '1';                                        // Das Ziffernzeichen '1' hat die Position
i = static_cast<int>(c);                        // 49 innerhalb der ASCII-Tabelle:
cout << i;                                      // 49
```

Die Operation `c = static_cast<char>(i);` ist nur gültig, wenn $-128 \leq i \leq 127$ ist (beziehungsweise $0 \leq i \leq 255$ bei `unsigned char`). Falls `i` außerhalb dieses Bereichs liegt, gibt es einen Datenverlust, weil die überzähligen Bits bei der Umwandlung nicht berücksichtigt werden können. Wie können Sie aus einem Ziffernzeichen die repräsentierte Ziffer erhalten? Da die Folge der Ziffernzeichen in der ASCII-Tabelle mit '0' beginnt, genügt es, '0' abzuziehen:

```
char c {'5'};
int ziffer {c - '0'};                                       // implizite Typumwandlung!
cout << ziffer << '\n';                                     // 5
ziffer =  static_cast<int>(c) - static_cast<int>('0');      // explizite Typumwandlung!
```

Weil ein `char` vom Compiler als 1-Byte-`int`-Zahl aufgefasst wird, ist das Rechnen ohne explizite Typumwandlung möglich. Zum Vergleich sind beide Möglichkeiten angegeben.

Operatoren für Zeichen

Da der Datentyp `char` intern als 1-Byte-Ganzzahl dargestellt wird, sind eigentlich alle Ganzzahl-Operatoren der Tabelle 1.1 von Seite 45 möglich, aber im Sinne der Bedeutung von *Zeichen* sind nur diejenigen aus Tabelle 1.6 sinnvoll.

Tabelle 1.6: Operatoren für `char`

Operator	Beispiel	Bedeutung
`=`	`d = 'A'`	Zuweisung
`<`	`d < f`	kleiner als
`>`	`d > f`	größer als
`<=`	`d <= f`	kleiner gleich
`>=`	`d >= f`	größer gleich
`==`	`d == f`	gleich
`!=`	`d != f`	ungleich
`<=>`	`(d <=> f) > 0`	3-Wege-Vergleich, Erklärung in Abschnitt 8.7

1.6.7 Logischer Datentyp bool

Ein logischer Datentyp wird zur Erinnerung an den englischen Mathematiker George Boole (1815–1864) mit `bool` bezeichnet. Boole hat die später nach ihm benannte Boolesche Algebra entwickelt. Logische Variablen können nur die Wahrheitswerte *wahr* (englisch *true*) beziehungsweise *falsch* (englisch *false*) annehmen. Falls notwendig, wird der Datentyp `bool` zu `int` gewandelt, wobei `false` der Wert 0 ist und `true` der Wert 1. Die Ausgabe als Text *true* bzw. *false* statt 1 oder 0 kann eingestellt werden:

```
bool istGrossBuchstabe {false};
char c {'*'};
cin >> c;
istGrossBuchstabe = (c >= 'A') && (c <= 'Z');
cout << istGrossBuchstabe;                  // Wandlung in int
cout.setf(ios_base::boolalpha);             // Textformat einschalten
cout << istGrossBuchstabe;                  // true oder false
cout.unsetf(ios_base::boolalpha);           // Textformat ausschalten
cout << istGrossBuchstabe << '\n';          // 1 oder 0
```

Zunächst werden die Klammern ausgewertet, die jeweils für sich Wahrheitswerte von `false` oder `true` ergeben. Die Relationen `>=` und `<=` beziehen sich dabei auf die ASCII-Tabelle. Es wird also geprüft, ob das Zeichen `c` gleich dem Zeichen 'A' ist oder in der Tabelle nach ihm folgt, und ob es gleich dem Zeichen 'Z' ist oder in der Tabelle vor dem 'Z' liegt. Die Wahrheitswerte werden durch das logische UND (`&&`) verbunden, nicht zu verwechseln mit dem bitweisen UND (`&`) der Tabelle 1.1 auf Seite 45. Das Ergebnis wird der Variablen `istGrossBuchstabe` zugewiesen. Die Klammern sind nur zur Verdeutlichung angegeben. Sie können entfallen, weil die relationalen Operatoren eine höhere Priorität als die logischen haben. Tabelle 1.7 zeigt die Operatoren für den Datentyp `bool`.

Der Datentyp `bool` wird an allen Stellen, die nicht ausdrücklich `bool` verlangen, nach `int` gewandelt (siehe Beispiel). Dabei wird `true` zu 1 und `false` zu 0. Die umgekehrte Wandlung von `int` nach `bool` ergibt `false` für 0 und `true` für alle anderen `int`-Werte. Hier ist die Wirkungsweise der *logischen Negation* zu sehen:

Tabelle 1.7: Operatoren für logische Datentypen

Operator	Beispiel	Bedeutung
`!`	`!i`	logische Negation
`&&`	`a && b`	logisches UND
`||`	`a || b`	logisches ODER
`==`	`a == b`	gleich
`!=`	`a != b`	ungleich
`=`	`a = true`	Zuweisung

```
bool wahrheitswert {true};
wahrheitswert = !wahrheitswert;                    // Negation
cout << wahrheitswert << '\n';                     // 0, d.h. false
// Beispiel mit int-Zahlen: Aus 0 wird 1 und aus einer Zahl ungleich 0
// wird durch die Negation eine 0:
int i {17};
int j {!i};                                        // 0 (implizite Typumwandlung)
i = !j;                                            // 1 (implizite Typumwandlung)
// Typumwandlung von int nach bool
wahrheitswert = 99;                                // true
wahrheitswert =  0;                                // false
```

Wegen der Umwandelbarkeit in `int`-Werte erlaubt der Compiler die arithmetischen Operationen, die mit `int`-Werten möglich sind – mit Ausnahme der Inkrementierungs- und Dekrementierunsoperatoren `++` und `--`. Das ergibt jedoch keinen Sinn. Nehmen Sie daher ausschließlich die Operatoren der Tabelle 1.7 für `bool`-Wahrheitswerte.

1.6.8 Regeln zum Bilden von Ausdrücken

Es gelten im Allgemeinen Vorrangregeln der Algebra beim Auswerten eines Ausdrucks inklusive der Klammerregeln. Tabelle 1.8 zeigt die Rangfolge einiger ausgewählter Operatoren. Sie ist nicht im C++-Standard definiert, ergibt sich aber aus der grammatischen Struktur von C++. So bindet das `*`-Zeichen stärker als das +-Zeichen – ganz wie man es gewohnt ist.

Einige der Operatoren werden erst in den folgenden Kapiteln erklärt. Eine ausführliche Auflistung der Operatorenrangfolge finden Sie im Anhang A.4, Seite 963. Manche vermuten, der `^`-Operator könne für Potenzen verwendet werden. Das ist falsch! `2^3` ist nicht $2^3 = 8$, sondern 1. Das Ergebnis der exklusiv-oder-Operation hat ein Bit nur genau an den Stellen, an denen die Operanden unterschiedliche Bits haben. Auch verträgt sich die Priorität nicht mit der Potenzierung: `2^3*4` ist nicht `(2^3)*4`, sondern `2^(3*4)`.

Auf gleicher Prioritätsstufe wird ein Ausdruck von links nach rechts abgearbeitet (linksassoziativ), mit Ausnahme der Ränge 2, 15 und 16 der Tabelle, die von rechts abgearbeitet werden (rechtsassoziativ). Zuerst werden jedoch die Klammern ausgewertet. Beispiele:

- +-Operator: Auswertung von links (linksassoziativ)
  `a = b + d + c;` ist gleichbedeutend mit `a = ((b + d) + c);`
- =-Operator: Auswertung von rechts (rechtsassoziativ)
  `a = b = d = c;` ist gleichbedeutend mit `a = (b = (d = c));`

Tabelle 1.8: Präzedenz *einiger ausgewählter* Operatoren

Rang	Operatoren	Beschreibung
2	`++ -- f() a[]`	Postfix-Increment/-Decrement, Funktionsaufruf, Index-Operator
3	`++ --`	Präfix-Increment/-Decrement
	`! ~ + -`	Logische und bitweise Negation, unäres `+ -`
	`*a &a sizeof`	Dereferenzierung, Adressoperator
5	`* / %`	Multiplikation, Division, Rest
6	`+ -`	Addition, Subtraktion
7	`<< >>`	Bitshift
8	`<=>`	Drei-Wege-Vergleich
9	`< <= > >=`	Relationale Operatoren
10	`== !=`	Gleich bzw. ungleich
11	`&`	Bitweises UND
12	`^`	Bitweises XOR (exklusiv-oder)
13	`|`	Bitweises ODER
14	`&&`	Logisches UND
15	`||`	Logisches ODER
16	`? :`	Bedingungsoperator
	`= += -=` usw.	Alle Zuweisungsoperatoren
17	`,`	Komma

Die Reihenfolge der Auswertung von Unterausdrücken untereinander, also auch Klammerausdrücken, ist jedoch undefiniert. Daher sollen Ausdrücke vermieden werden, die einen Wert sowohl verändern als auch benutzen. Ein Beispiel:

```
int i {2};
i = i++ + i;                                    // Fehler!
```

Der Wert von `i` ist undefiniert, weil die Reihenfolge der Auswertung der beiden Operanden nicht feststeht. Es gibt zwei Möglichkeiten für die Auswertungsreihenfolge:

1. `i++` wird erst für die Addition reserviert und dann hochgezählt. Das hochgezählte `i` ist der zweite Summand. Es wird also 2 + 3 = 5 ausgerechnet und der linken Seite zugewiesen.
2. `i++` wird berechnet. Damit ist `i` gleich 3. Auf diesen Wert wird `i` (also dasselbe) addiert. Das Ergebnis ist 6 und wird der linken Seite zugewiesen.

Der Ausdruck `i = ++i + i;` kann mit dem Kommaoperator in einen definierten Ausdruck verwandelt werden: `i = ++i, i + i;`. Durch den Kommaoperator wird die Auswertungsreihenfolge von links nach rechts festgelegt. Das Ergebnis des Ausdrucks ist das Ergebnis des Teilausdrucks nach dem letzten Komma. Besser ist natürlich das Zerlegen in zwei Anweisungen.

1.6.9 Standard-Typumwandlungen

Standard-Typumwandlungen sind implizite Typumwandlungen für eingebaute Typen. Implizit heißt, dass eine Typumwandlung nicht ausdrücklich hingeschrieben wird und der Compiler dennoch keine Fehlermeldung bei der Typumwandlung ausgibt. Es kann auch eine Folge von hintereinandergeschalteten Standard-Typumwandlungen geben. In diesem Abschnitt werden die wichtigsten Möglichkeiten beschrieben. *Im Einzelfall kann ein Informationsverlust auftreten.* Mögliche Ursachen:

- Genauigkeitsverlust, weil zum Beispiel eine `float`-Zahl nicht so viele Mantissen-Bits wie eine `double`-Zahl hat
- Überschreitung des Grenzbereichs, weil der mögliche Exponent einer `float`-Zahl kleiner als der einer `double`-Zahl ist
- Verlust des Nachkommateils bei der Umwandlung z.B. `double` nach `int`
- Umwandlung z.B. einer zu großen `double`- oder `float`-Zahl in einen integralen Typ
- Die Bitzahl reicht nicht, etwa wenn eine den `int`-Bereich überschreitende `long`-Zahl in eine `int`-Zahl umgewandelt wird. Dasselbe gilt für die Umwandlung einer Zahl, die größer als 127 ist, in den Typ `signed char` (bzw. > 255 in `unsigned char`). Die überzähligen Bits werden einfach abgeschnitten. Die folgenden Zahlen zeigen die 8-Byte-`long`-Zahl 4294967364 in binärer Darstellung und die entstehende 4-Byte-`int`-Zahl in binärer Darstellung nach Abschneiden der höchstwertigen 32 Bits:

  ```
  0000000000000000000000000000000100000000000000000000000001000100
                                  00000000000000000000000001000100
  ```

  So wird aus der `long`-Zahl 4294967364 die `int`-Zahl 68.
- Vorzeichenverlust und gleichzeitige Wertänderung, wenn zum Beispiel eine negative `int`-Zahl in eine `unsigned int`-Zahl umgewandelt wird
- Nichtbeachtung der Rechenregeln. Ein Beispiel:

  ```
  double x {3 / 4};          // x ist 0.0
  double y {3 / 4.};         // y ist 0.75
  ```

  `3/4` ist eine Rechenoperation mit *ganzen* Zahlen. Das Ergebnis ist die ganze Zahl 0. Erst bei der Zuweisung wird die 0 in eine `double`-Zahl umgewandelt. 4. ist jedoch vom Typ `double` (beachten Sie den Punkt nach der 4!). Um die Rechenoperation ausführen zu können, wird die 3 vom Compiler ebenfalls in eine `double`-Zahl umgewandelt, sodass `3./4.` (= `3.0/4.0`) berechnet wird. Das Ergebnis ist 0.75, also auch `double`.

Typumwandlung integraler Typen nach int

Ein Wert des Typs `char`, `signed char`, `unsigned char`, `short int` oder `unsigned short int` kann ohne Informationsverlust in einen `int`-Wert umgewandelt werden (englisch *integral promotion*).

Typumwandlung integraler Typen untereinander

Integrale Typen sind untereinander konvertierbar. Im C++-Standard werden alle Fälle, die nicht zur obigen »integral promotion« gehören, *integral conversion* genannt. Sonderfälle sind die Typen `bool` (siehe Seite 57) und `enum` (siehe Seite 86). Bei Operationen mit mehreren Operanden kann es implizite, also vom Compiler automatisch vorgenommene, Typumwandlungen geben. Das wird am Beispiel einer einfachen Subtraktion gezeigt:

Wenn ein Operand `signed` ist und der andere `unsigned` und beide haben *dieselbe* Bitbreite, dann wandelt der Compiler den `signed`-Operand in einen `unsigned`-Operand um, um beide vergleichen zu können. Bei der Subtraktion `(1u - 2)` wird die `2` in `2u` (d.h. `unsigned int`) umgewandelt, dann wird subtrahiert. Da nun beide Operanden `unsigned` sind, ist auch das Ergebnis `unsigned`, kann also nicht kleiner als 0 sein. Das ist in diesem Fall, wo eigentlich -1 erwartet würde, falsch! Wenn die Bitbreite unterschiedlich ist, wird vor

der Rechnung in den Typ mit der größeren Bitbreite umgewandelt. Wenn `sizeof(long)` größer als `sizeof(int)` ist, ergibt `(1u - 2l)` (`l` = `long`!) dann korrekt -1.

Tipps

1. Vermeiden Sie `unsigned`.
2. Mischen Sie nie `unsigned`- mit `signed`-Zahlen.

float-Typumwandlungen

Ein Wert des Typs `float` kann in einen `double`- oder einen Integer-Wert umgewandelt werden und umgekehrt. Für `bool` gilt ähnlich wie im `int`-Fall: `true` wird 1.0, `false` wird 0.0, 0.0 wird `false` und ein `float`- bzw. `double`-Ausdruck ungleich 0 wird `true`.

1.7 Gültigkeitsbereich und Sichtbarkeit

In C++ gelten Gültigkeits- und Sichtbarkeitsregeln für Namen. Es gibt folgende Regeln:

- Namen sind nur *nach der Deklaration* und nur *innerhalb des Blocks* gültig, in dem sie deklariert wurden. Sie sind *lokal* bezüglich des Blocks. Zur Erinnerung: Ein Block ist ein Programmbereich, der durch ein Paar geschweifte Klammern `{ }` eingeschlossen wird. Blöcke können verschachtelt sein, also selbst wieder Blöcke enthalten.
- Namen von Variablen sind auch gültig für innerhalb des Blocks neu angelegte innere Blöcke.
- Die Sichtbarkeit (englisch *visibility*) zum Beispiel von Variablen wird eingeschränkt durch die Deklaration von Variablen gleichen Namens. Für den Sichtbarkeitsbereich der inneren Variablen ist die äußere unsichtbar.

Der Datenbereich für lokale Daten wird bei Betreten des Gültigkeitsbereichs auf einem besonderen Speicherbereich mit dem Namen *Stack* angelegt und am Ende des Gültigkeitsbereichs, also am Blockende, wieder freigegeben. Der Stack (deutsch: »Stapel«, auch Kellerspeicher) ist ein Bereich mit der Eigenschaft, dass die zuletzt darauf abgelegten Elemente zuerst wieder freigegeben werden (*last in, first out*). Damit lässt sich das beschriebene Anlegen von Variablen bei Blockbeginn und ihre Freigabe bei Blockende gut verwalten, ohne dass *wir* uns darum kümmern müssen.

Das folgende Programm zeigt Beispiele für verschiedene Gültigkeitsbereiche (englisch *scope*). Der im Programm verwendete Operator `::` bewirkt den Zugriff auf Variablen, die *global* sichtbar sind, also keinen Zugriff auf den nächsten äußeren Block.

Listing 1.11: Beispielprogramm: Variablen und Blöcke *(cppbuch/k1/bloecke.cpp)*

```
#include <iostream>
using namespace std;
// a und b werden außerhalb eines jeden Blocks deklariert. Sie sind damit innerhalb
// eines jeden anderen Blocks gültig und heißen daher globale Variablen.
int a {1};
int b {2};

int main()
{          // Ein neuer Block beginnt.
  cout << "globales␣a=␣" << a << '\n';            // Ausgabe von a

  // Innerhalb des Blocks wird eine Variable a deklariert. Ab jetzt ist das globale a noch
  // gültig, aber nicht mehr unter dem Namen a sichtbar, wie die Folgezeile zeigt.
  int a {10};
  // Der Wert des lokalen a wird ausgegeben:
  cout << "lokales␣a=␣" << a << '\n';

  // Das globale a lässt sich nach der Deklaration des lokalen a nur noch mithilfe des
  // Bereichsoperators :: (engl. scope operator) ansprechen. Ausgabe von ::a.
  cout << "globales␣::a=␣" << ::a << '\n';
  {          // Ein neuer Block innerhalb des bestehenden beginnt.
    int b {20};
    // Variable b wird innerhalb dieses Blocks deklariert.
    // Damit wird das globale b zwar nicht ungültig, aber unsichtbar.
    int c {30};          // c wird innerhalb dieses Blocks deklariert.
    // Die Werte von b und c werden ausgegeben.
    cout << "lokales␣b␣=␣" << b << '\n';
    cout << "lokales␣c␣=␣" << c << '\n';

    // Wie oben beschrieben, ist das globale b nur über den
    // Scope-Operator ansprechbar. Ausgabe von ::b.
    cout << "globales␣::b␣=␣" << ::b << '\n';
  }          // Der innere Block wird geschlossen. Damit ist das globale b
             // auch ohne Scope-Operator wieder sichtbar:
  cout << "globales␣b␣wieder␣sichtbar:␣b␣=␣" << b << '\n';

  // cout << "c = " << c << '\n'; // Fehler, siehe Text
}          // Ende des äußeren Blocks
```

Das Programm wird im Kommentar zeilenweise erklärt. Es zeigt, dass Gültigkeit und Sichtbarkeit nicht das Gleiche sind, und erzeugt folgende Ausgabe:

```
globales a= 1
lokales a= 10
globales ::a= 1
lokales b = 20
lokales c = 30
globales ::b = 2
globales b wieder sichtbar: b = 2
```

Die Kommentarzeichen `//` in der vorletzten Programmzeile sind erforderlich, weil der Compiler diese Zeile sonst als fehlerhaft bemängeln würde. Grund: Durch Schließen des

inneren Blocks ist der Gültigkeitsbereich aller in diesem Block deklarierten Variablen beendet, also ist `c` außerhalb des Blocks unbekannt.

Vermeiden Sie lokale Objekte mit Namen, die Objekte in einem äußeren Gültigkeitsbereich verdecken! Die Verständlichkeit eines Programms wird durch verschiedene Objekte mit demselben Namen erschwert, wie das Beispiel hoffentlich zeigt.

1.7.1 Namespace std

Eine weitere Möglichkeit zur Schaffung von Sichtbarkeitsbereichen sind *Namensräume* (englisch *namespaces*). Bisher wurde der zur C++-Standardbibliothek gehörende Namensraum `std` benutzt, wie Sie an den Zeilen `using namespace std;` in den Beispielen gesehen haben. Namensräume spielen bei der Benutzung verschiedener Bibliotheken eine Rolle. Einzelheiten werden weiter unten in Abschnitt 2.5 (ab Seite 151) erklärt. Hier wird vorab darauf hingewiesen, dass eine pauschale Nutzung der Standardbibliothek nicht notwendig ist, wenn die betreffenden Elemente mit einem sogenannten qualifizierten Namen angesprochen werden, der den Namensraum angibt. Bezogen auf den Standardnamensraum `std` gibt es im Wesentlichen drei Möglichkeiten:

```
// 1. Pauschale Nutzung
using namespace std;                          // macht alles aus std ab jetzt bekannt
// ... ggf. weiterer Programmtext
cout << "Ende" << endl;                       // zur Abwechslung mal endl statt '\n'
```

oder

```
// 2. Nutzung von cout und endl aus std mit qualifizierten Namen:
// using namespace std; sei nicht deklariert.
std::cout << "Ende" << std::endl;             // richtig
cout << "Ende" << endl;                       // Fehlermeldung des Compilers!
```

oder

```
// 3. Deklaration ausgewählter Teile; using namespace std; sei nicht deklariert.
using std::cout;
using std::endl;
cout << "Ende" << endl;
```

Die erste Möglichkeit wird in den `main()`-Programmen dieses Buchs bevorzugt, weil sie Schreibarbeit spart. Die dritte Möglichkeit kann ebenfalls in **.cpp*-Dateien verwendet werden. Die zweite Möglichkeit wird in allen anderen Fällen empfohlen. Sie werden in den Beispielen alle drei Varianten antreffen.

Übung

1.9 Schreiben Sie ein Programm, das die größtmögliche `unsigned int`-Zahl (`int`, `long`, `unsigned long`) ausgibt, *ohne* dass die Kenntnis der systemintern verwendeten Bitanzahl für jeden Datentyp benutzt wird. *Hinweise*: Studieren Sie die möglichen Operatoren für ganze Zahlen. Der ~-Operator invertiert alle Bits, macht also aus einer `0` die maximal größte `unsigned`-Zahl. Sie ist die größte Zahl, weil alle Bits gesetzt sind. Bei `signed`-Zahlen wird ein Bit für das Vorzeichen gebraucht (Zweierkomplement-Darstellung).

1.8 Kontrollstrukturen

Kontrollstrukturen dienen dazu, den Programmfluss zu steuern. Im einfachsten Fall werden die Anweisungen eines Programms eine nach der anderen in derselben Reihenfolge ausgeführt, wie sie hingeschrieben worden sind. Dies ist nicht immer erwünscht. Manchmal ist es notwendig, dass der Programmfluss sich in Abhängigkeit von den Daten ändern soll, oder es müssen Teile des Programms wiederholt durchlaufen werden. Erst mit Kontrollstrukturen lassen sich überhaupt Programme von einiger Komplexität schreiben.

1.8.1 Anweisungen

In den folgenden Abschnitten wird des Öfteren der Begriff »Anweisung« gebraucht, der deswegen an dieser Stelle erläutert werden soll. Eine Anweisung kann unter anderem sein (eine vollständige Auflistung ist nicht beabsichtigt):

- eine Deklarationsanweisung,
- eine Ausdrucksanweisung,
- eine Schleifenanweisung,
- eine Auswahlanweisung,
- eine Verbundanweisung, auch Block genannt.

Deklarationsanweisung

Eine Deklarationsanweisung führt einen Namen in das Programm ein. Sie kann in verschiedenen Formen vorkommen, unter denen die einfachen Deklarationen wie zum Beispiel `int x;` die häufigsten sind. Nach dieser Deklaration ist das Objekt `x` in einem Programm bekannt und kann benutzt werden. Eine einfache Deklaration wird stets mit einem Semikolon `;` abgeschlossen.

Ausdrucksanweisung

Eine Ausdrucksanweisung ist ein Ausdruck, gefolgt von einem Semikolon. Ein Ausdruck repräsentiert nach der Auswertung einen Wert. Zum Beispiel kann der Ausdruck `x == 1` den Wert `true` oder `false` annehmen. In C++ ist mit einer Ausdrucksanweisung in der Regel eine Aktivität verbunden, zum Beispiel eine Zuweisung (siehe unten) oder eine Ausgabe auf den Bildschirm. Der Wert einer Anweisung mit `cout` ist das Objekt `cout` selbst, das die Ausgabe bewerkstelligt.

```
cout << a << b;
```

bedeutet dasselbe wie

```
(cout << a) << b;
```

Das Ergebnis des Ausdrucks in den runden Klammern ist wieder `cout`, weswegen der zweite Teil der Ausgabe als `cout << b;` gelesen werden kann.

Ein alleinstehendes Semikolon ist eine Leeranweisung.

Zuweisung

Eine Zuweisung ist ein Spezialfall einer Ausdrucksanweisung. Ein Zuweisungsausdruck, zum Beispiel `a = b`, besteht aus drei Teilen:

- einer linken Seite,
- dem Zuweisungsoperator =,
- einer rechten Seite.

Dabei wird die linke Seite als (symbolische) *Adresse*, die rechte Seite als *Wert* interpretiert. Die symbolische Adresse wird durch einen *Namen* repräsentiert. Hier ist es der Name `a`. Die Bedeutung einer Zuweisung `a = b;` ist also: Der Wert der Variablen oder Konstanten `b` wird an die Adresse der Variablen `a` kopiert, sodass danach Variable `a` denselben Wert wie `b` hat. Weil eine Zuweisung einen Wert hat, sind verkettete Zuweisungen möglich :

```
a = b = c;
```

meint, dass `b` (= der Wert der Zuweisung `b = c`) der Variablen `a` zugewiesen wird. Der Wert der letzten Zuweisung `a = b` wird nicht mehr verwendet.

Schleifenanweisung

Diese Anweisungen sind mit den Schlüsselwörtern `for`, `while` und `do .. while` verbunden. Einzelheiten folgen in den nächsten Abschnitten.

Auswahlanweisung

Diese Anweisungen sind mit den Schlüsselwörtern `if` und `switch` verbunden. Einzelheiten folgen in den nächsten Abschnitten.

Verbundanweisung, Block

Eine Verbundanweisung, auch Block genannt, ist ein Paar geschweifte Klammern, die eine Folge von Anweisungen enthalten. Die Folge kann leer sein. Die enthaltenen Anweisungen können selbst wieder Verbundanweisungen sein.

```
{ }             // leerer Block

{               // Block mit einer Anweisung
  Anweisung
}

{               // Block mit zwei Anweisungen
  Anweisung1
  Anweisung2
}
```

1.8.2 Sequenz (Reihung)

Im einfachsten Fall werden die Anweisungen der Reihe nach durchlaufen:

```
a  = b + 1;
a += a;
```

```
cout << "\nErgebnis␣=" << a;
```

Nebenbei sehen wir, dass das Zeichen '\n' in eine Zeichenkette eingebaut werden kann, sodass vor der Ausgabe von *Ergebnis* eine neue Zeile auf dem Bildschirm begonnen wird.

1.8.3 Auswahl (Selektion, Verzweigung)

Häufig hängt die Ausführung von Anweisungen von einer Bedingung ab. C++ stellt für solche Zwecke die `if`-Anweisung bereit.

```
if (Bedingung)
   Anweisung1
```

bedeutet, dass *Anweisung1* nur dann ausgeführt wird, wenn die *Bedingung* wahr ist, das heißt zu einem Ausdruck mit dem Wert `true` (oder ungleich 0) ausgewertet wird. Die Bedingung kann ein arithmetisches Ergebnis haben. Alternativ kann eine zweite Anweisung angegeben werden, sodass *Anweisung1* ausgeführt wird, falls die *Bedingung* wahr ist, und andernfalls *Anweisung2* (`else`-Zweig):

```
if (Bedingung)
   Anweisung1
else
   Anweisung2
```

Zu einem `if`- oder `else`-Zweig gehört stets nur genau *eine* Anweisung! Diese kann natürlich eine Verbundanweisung (Block) sein, also ein Paar geschweifte Klammern, die beliebig viele Anweisungen umschließen können, also auch keine oder nur eine. Abbildung 1.5 zeigt das vereinfachte Syntaxdiagramm einer `if`-Anweisung.

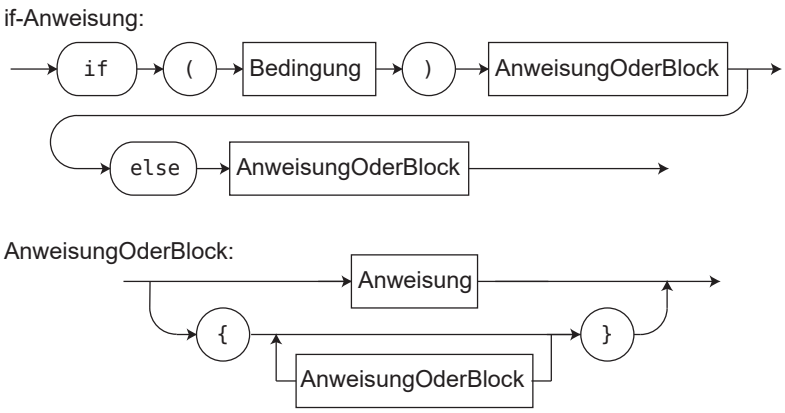


Abbildung 1.5: Vereinfachtes Syntaxdiagramm einer `if`-Anweisung

Ein Semikolon nach einem Block bedeutet eine zweite (leere) Anweisung. Diese ist immer unnütz und im Fall eines folgenden `else` sogar falsch. Beispiele für `if`-Anweisungen:

```
if (x < 100)
   cout << "x␣<␣␣100";                    // nur eine Anweisung
else
   cout << "x␣>=␣100";
```

```
if (a > b) {
   x = a - b;                                   // zwei Anweisungen im Block
   y = 2*a;
}
```

```
if ((c >= 'A') && (c <= 'Z')) {                  // if-Anweisung mit else
   istGrossBuchstabe = true;
}
else {
   istGrossBuchstabe = false;
}
```

Die letzte `if`-Anweisung ist gleichwertig mit: `istGrossBuchstabe = c >= 'A' && c <= 'Z';`. Der Bedingungsausdruck muss vom Typ `bool` sein oder in `bool` umgewandelt werden können. Relationale Ausdrücke wie `(a < b)` werden zu `true` ausgewertet, falls `a < b` ist, ansonsten zu `false`. `if (a == 0)` kann daher auch (aber weniger schön) als `if (!a)` geschrieben werden: `a` ist eine Zahl vom Typ `int`, die in einen Wahrheitswert umgewandelt wird. Dabei wird ein Wert, der gleich 0 ist, in `false` und ein Wert ungleich 0 in `true` umgewandelt.

Auf dieses Ergebnis wird der Negationsoperator `!` angewendet, womit sich das gewünschte Verhalten ergibt. Bedingungsausdrücke werden von links nach rechts ausgewertet. Dabei werden unnötige Berechnungen übersprungen. Damit ist gemeint, dass in einer Bedingung, die aus mehreren ODER-Verknüpfungungen besteht, die Berechnung nach dem ersten Ergebnis, das den Wahrheitswert *wahr* liefert, abgebrochen werden kann. Grund ist, dass sich das Ergebnis nach weiteren Berechnungen nicht ändern kann. Umgekehrt ist ein Weiterrechnen bei UND-Verknüpfungen nicht sinnvoll, sobald auch nur ein Teilergebnis *falsch* liefert. Diese Art der Auswertung wird *Kurzschlussauswertung* genannt (englisch *short circuit evaluation*).

Mit Teilbedingungen verbundene *Seiteneffekte* werden daher nur bei Auswertung der jeweiligen Teilbedingung ausgeführt. Seiteneffekte sind Ergebnisse, die zusätzlich zum eigentlichen Zweck nebenbei entstehen. In den folgenden Beispielen ist die Hauptsache die Auswertung der Bedingung. Als Seiteneffekt werden *nach* Auswertung der Bedingung, aber noch *vor* Ausführung des folgenden Programmcodes, `i` beziehungsweise `j` durch den Operator `++` modifiziert, sofern es nötig ist, die Teilbedingung zu berechnen. Vollziehen Sie die Beispiele nach! Das Ergebnis ist im Kommentar aufgeführt. Das Beispiel ist nur zur Übung gedacht. Im Allgemeinen sind Seiteneffekte zu vermeiden!

```
int i {0};                                       // stimmt's oder nicht?
int j {2};
if (i++ || j++) i++;                             // i == 2 und j == 3
```

```
int i {1};                                       // stimmt's oder nicht?
int j {2};
if (i++ || j++) i++;                             // i == 3 und j == 2
```

```
int i {0};                                       // stimmt's oder nicht?
int j {2};
if (i++ && j++) i++;                             // i == 1 und j == 2
```

```
int i {1};                                  // stimmt's oder nicht?
int j {2};
if (i++ && j++) i++;                        // i == 3 und j == 3
```

`if`-Anweisungen können beliebig tief geschachtelt werden. Das Beispielprogramm hat die Aufgabe, ein Zeichen einzulesen und zu prüfen, ob es einer römischen Ziffer entspricht. Falls ja, soll die zugehörige arabische Zahl angezeigt werden, falls nein, eine passende Meldung.

Listing 1.12: Umwandlung römischer Ziffern mit `if`/`else` *(cppbuch/k1/roemzif1.cpp)*

```
// Umwandlung römischer Ziffern
#include <iostream>
using namespace std;

int main()
{
  int zahl {0};
  char zeichen {'*'};          // Initialisierung nicht zwingend notwendig
  cout << "Zeichen ?";
  cin >> zeichen;

  if (zeichen == 'I') {
    zahl = 1;
  }
  else if (zeichen == 'V') {
    zahl = 5;
  }
  else if (zeichen == 'X') {
    zahl = 10;
  }
  else if (zeichen == 'L') {
    zahl = 50;
  }
  else if (zeichen == 'C') {
    zahl = 100;
  }
  else if (zeichen == 'D') {
    zahl = 500;
  }
  else if (zeichen == 'M') {
    zahl = 1000;
  }
  if (zahl == 0) {
    cout << "keine römische Ziffer!\n";
  }
  else {
    cout << zahl << '\n';
  }
}
```

if und Initialisierung

Manchmal merkt man sich das Ergebnis eines Funktionsaufrufs in einer Variablen, um die Funktion nicht mehrfach aufrufen zu müssen. Wenn diese Variable nur innerhalb der `if`-Anweisung gelten soll, kann man die Anweisung mit einem Block umschließen. Ein Beispiel:

```
float eingabe {0.0f};
cin >> eingabe;
{ // Blockbeginn
  float x {sin(eingabe)};
  if (x >= 0) {
    cout << x << "␣ist␣null␣oder␣positiv!\n";
  } else {
    cout << x << "␣ist␣negativ!\n";
  }
} // x ist ab hier nicht mehr gültig
```

Alternativ kann die Initialisierung in die runden Klammern der `if`-Bedingung verlegt werden. Der umschließende Block kann entfallen. Das Ergebnis ist kürzer, aber gleichwertig:

```
float eingabe {0.0f};
cin >> eingabe;
if (float x {sin(eingabe)}; x >= 0) {        // if mit Initialisierung
  cout << x << "␣ist␣null␣oder␣positiv!\n";
} else {
  cout << x << "␣ist␣negativ!\n";
} // x ist hier ungültig
```

Achtung, Falle: Fehler in Verbindung mit if

Ein häufiger Fehler ist die versehentlich falsche Schreibweise des Gleichheitsoperators, sodass sich unfreiwillig der Zuweisungsoperator ergibt. `a` und `b` seien `int`-Zahlen.

```
if (a = b)  {                                // Vorsicht! Vermutlich anders gemeint!
   cout << "a␣ist␣gleich␣b";
}
```

bewirkt, dass `a ist gleich b` immer dann ausgegeben wird, wenn `b` ungleich `0` ist. Die richtige Schreibweise ist:

```
if (a == b) {
   cout << "a␣ist␣gleich␣b";
}
```

Die Verwendung von `=` statt `==` hat die Wirkung einer *Zuweisung*. Zunächst erhält `a` den Wert von `b`. Das *Ergebnis* dieses Ausdrucks, nämlich `a`, wird dann als logische Bedingung interpretiert. Die falsche Schreibweise führt also nicht nur zu einem falschen Ergebnis für die Bedingung, sondern auch zur nicht beabsichtigten Änderung des Werts von `a`. Freundliche Compiler geben an solchen Stellen eine Warnung aus – eine Chance, sich noch einmal zu überlegen, ob wirklich eine Zuweisung gemeint ist.

Eine weitere Gefahr sind Mehrdeutigkeiten durch falschen Schreibstil, das heißt falsche, wenn auch richtig gemeinte Einrückungen. Ohne Klammerung gehört ein `else` immer zum letzten `if`, dem kein `else` zugeordnet ist:

```
if (x == 1)
  if (y == 1)
    cout << "x␣==␣y␣==␣1␣!";
else
    cout << "x␣!=␣1";                          // falsch
```

Trotz der augenfälligen Übereinstimmung des Zeilenanfangs der untersten Zeile mit dem Zeilenanfang von `if (x == 1)` gehört das `else` syntaktisch zur `if (y == 1)`-Zeile! Richtig ist:

```
if (x == 1) {
  if (y == 1) {
    cout << "x␣==␣1␣und␣y␣==␣1␣!";
  }
}
else {
  cout << "x␣!=␣1";                           // nun korrekt
}
```

Auch einzelne Anweisungen nach `if` bzw. `else` sollen daher immer geklammert werden. Ein weiterer gelegentlicher Fehler aus der Praxis ist ein überflüssiges Semikolon (also eine Leeranweisung) nach der Bedingung, das beim Lesen leicht übersehen wird.

```
if (a == b);                                  // Fehler!
  cout << "a␣ist␣gleich␣b";
```

Auf diese Art wird die `if`-Abfrage zwar durchgeführt, aber ohne Folgen, da sie bereits beim ersten Semikolon endet. Die anschließende Ausgabe wird also in jedem Fall durchgeführt, da die Ausgabe nun eine eigenständige Anweisung ist und somit nicht mehr zur `if`-Anweisung gehört. Solche Fälle akzeptiert der Compiler widerspruchslos!

Probleme kann es geben, wenn `int` und `unsigned` in einer Bedingung gemischt werden. Eine `unsigned`-Zahl kann nicht negativ sein. Wenn so ein Wert unüberlegt mit einem `int`-Wert verglichen wird, kann es schwer zu entdeckende Fehler geben:

```
int i {-1};
unsigned int u {0u};                  // u für unsigned (darf entfallen)
if (u < i) {
  cout << u << '<' << i << '\n';
}
```

Da diese Art der Verwendung prinzipiell zulässig ist, wird ein Compiler sie akzeptieren und allenfalls eine Warnung ausgeben. Ganz klar ist $0 > -1$, dennoch wird $0 < -1$ ausgegeben! Den Grund kennen Sie von Seite 60. Bei der Umwandlung von -1 in eine `unsigned`-Zahl wird das Bitmuster beibehalten. Damit ist das Ergebnis die größtmögliche `unsigned`-Zahl, die natürlich größer als 0 ist.

Bedingungsoperator ?:

Dieser Operator ist der einzige in C++, der drei Operanden benötigt. Abbildung 1.6 zeigt die Syntax.

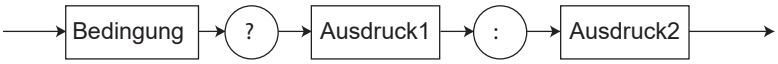


Abbildung 1.6: Syntaxdiagramm des Bedingungsoperators

Falls die *Bedingung* zutrifft, ist der Wert des gesamten Ausdrucks der Wert von *Ausdruck1*, ansonsten der Wert von *Ausdruck2*. Ein Ausdruck mit dem Bedingungsoperator kann lesbarer durch eine `if`-Anweisung ersetzt werden, wird aber wegen seiner Kürze geschätzt. Die Berechnung des Maximums zweier Zahlen lautet:

```
max = a > b ? a : b;
```

Das `if`-Äquivalent dazu ist:

```
if (a > b) {
  max = a;
}
else {
  max = b;
}
```

Übungen

1.10 Schreiben Sie ein Programm, das drei ganze Zahlen als Eingabe verlangt. Die erste ist als Anfang eines Zahlenbereichs, die zweite als Ende des Bereichs gemeint. Das Programm soll prüfen, ob die dritte Zahl innerhalb des Bereichs einschließlich der Grenzen liegt, und eine entsprechende Meldung ausgeben. Geben Sie eine Fehlermeldung aus, wenn die Zahl für den Anfang größer als die Zahl für das Ende ist.

1.11 Schreiben Sie ein Programm, das drei ganze Zahlen als Eingabe verlangt und dann den Wert der größten Zahl ausgibt.

1.8.4 Fallunterscheidungen mit switch

Eine `if`-Anweisung erlaubt nur zwei Möglichkeiten. Erst durch die Verschachtelung konnte die Auswahl unter mehreren Möglichkeiten getroffen werden. Die Gefahr besteht jedoch, dass die gesamte Anweisung bei größerer Schachtelungstiefe unübersichtlich wird und Änderungen nur umständlich nachzutragen sind. Einfacher und übersichtlicher ist daher die Auswahl unter vielen Anweisungen mit `switch`. Abbildung 1.7 zeigt die Syntax, ein Beispiel folgt unten.

Der *Ausdruck* wird ausgewertet und muss ein Ergebnis vom Typ `int` haben oder nach `int` konvertierbar sein wie zum Beispiel `char`. Dieses Ergebnis wird mit den `case`-Konstanten *const1, const2* ... verglichen, die zum Einsprung an die richtige Stelle dienen. Bei Übereinstimmung werden die zur Konstante gehörigen Anweisungen ausgeführt. `break` führt zum Verlassen der `switch`-Anweisung und sollte stets verwendet werden. Die `case`-Konstanten *const1, const2* ... müssen eindeutig und auf `int` abbildbar sein. Zeichen (Typ `char`) sind erlaubt, `float`-Zahlen nicht.

Die nach `default` stehenden Anweisungen (meistens nur eine) werden immer dann ausgeführt, wenn der `switch`-Ausdruck einen Wert liefert, der mit keiner der `case`-Konstanten übereinstimmt. `default` ist optional, doch ist es sinnvoll, `default` mit anzugeben. Insbesondere werden an dieser Stelle nicht vorgesehene Werte des `switch`-Ausdrucks abge-

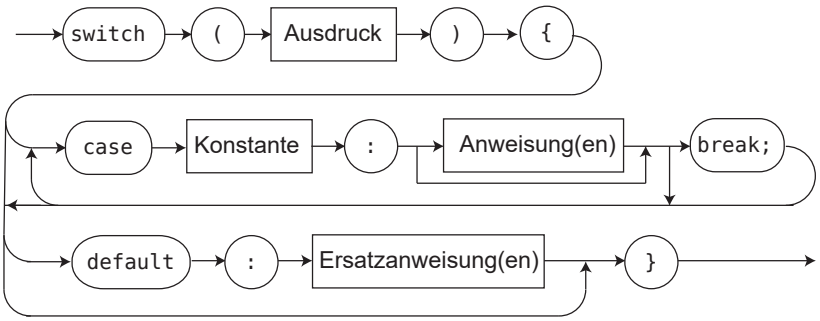


Abbildung 1.7: Vereinfachtes Syntaxdiagramm einer `switch`-Anweisung

fangen oder Daten, die nicht berücksichtigt werden sollen, wie zum Beispiel fehlerhafte Tastatureingaben. Die Aufgabe, römische Ziffern zu erkennen, kann übersichtlicher mit der Fallunterscheidung durch `switch` als mit verschachtelten `if`-Anweisungen wie auf Seite 68 gelöst werden, wie das folgende Programm zeigt.

Listing 1.13: Umwandlung römischer Ziffernzeichen mit `switch` *(cppbuch/k1/roemzif2.cpp)*

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

int main()
{
  int zahl {-1};
  char zeichen {'*'};
  cout << "Zeichen␣?";
  cin >> zeichen;
  switch (zeichen) {
  case 'I':
    zahl = 1;
    break;
  case 'V':
    zahl = 5;
    break;
  case 'X':
    zahl = 10;
    break;
  case 'L':
    zahl = 50;
    break;
  case 'C':
    zahl = 100;
    break;
  case 'D':
    zahl = 500;
    break;
  case 'M':
    zahl = 1000;
    break;
  default:
    zahl = 0;
```

```
  }
  if (zahl > 0) {
    cout << "Zahl␣=␣" << zahl;
  }
  else {
    cout << "keine␣römische␣Ziffer!";
  }
  cout << '\n';
}
```

[[fallthrough]]

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

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

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

Ein fehlendes `break` soll kommentiert werden. Das geschieht am besten durch Hinzufügen des Attributs `[[fallthrough]]`. Es kennzeichnet, dass `break` mit Absicht weggelassen wird. Gleichzeitig wird eine mögliche Warnung des Compilers wegen der fehlenden `break`-Anweisung unterdrückt. Zwei doppelte eckige Klammernpaare kennzeichnen ein Attribut, wie im Programmauszug gezeigt. Weitere Attribute finden Sie im Anhang A.5.

Listing 1.14: Attribut `fallthrough` (Auszug aus *cppbuch/k1/roemzif3.cpp*)

```
switch (zeichen) {
  case 'i':
     [[fallthrough]];
  case 'I':
     zahl = 1;
     break;
  case 'v':
     [[fallthrough]];
  case 'V':
     zahl = 5;
     break;
```

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.

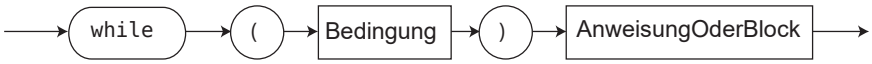
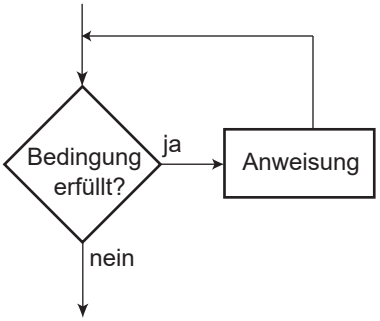


Abbildung 1.8: Syntaxdiagramm einer while-Schleife

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.

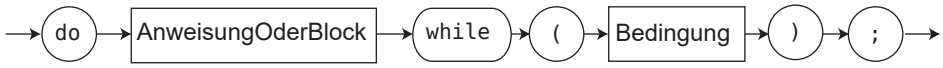
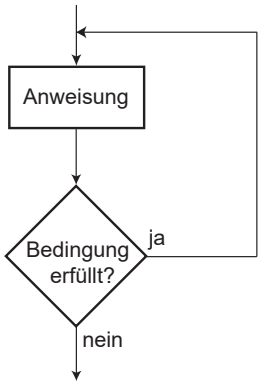


Abbildung 1.10: Syntaxdiagramm einer `do while`-Schleife

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.

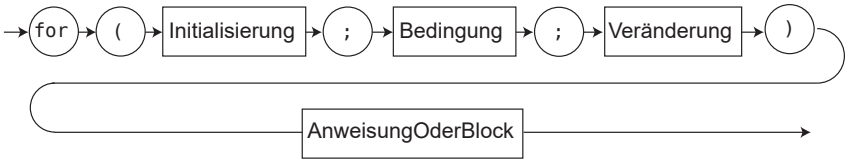


Abbildung 1.12: Syntaxdiagramm einer `for`-Schleife

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:

```
for (float f = 0.4; f != 10.4; ++f) {          // ∞-Schleife
  cout << f << '\n';
}
```

Abgesehen von möglichen arithmetischen Problemen wird in der letzten Schleife die `float`-Variable `f` mit der `double`-Konstante 10.4 verglichen. Um den Vergleich durchführen zu können, bringt der Compiler beide Größen auf dieselbe Bitbreite, das heißt, er interpretiert den Vergleich als `(double)f != 10.4`. Die im Vergleich zu `double` wenigen Bits der `float`-Zahl reichen nicht für die erforderliche Genauigkeit aus, wie die folgenden Zeilen zeigen:

```
cout.setf(ios::showpoint|ios::fixed, ios::floatfield); // Formatierung siehe Kapitel 9
cout.precision(15);                                    // anzuzeigende Genauigkeit
                                                       // Ausgabe auf meinem System:
cout << 10.4 << '\n';                                  // 10.400000000000000
cout << (double)10.4f << '\n';                         // 10.399999618530273
```

Wenn Sie bei `double`-Laufvariablen nur die Operatoren `<` oder `>` zum Vergleich verwenden, ist das Problem zum Teil entschärft. Aber die Anzahl der Schleifendurchläufe bleibt möglicherweise undefiniert: Nur eine sehr geringe Abweichung im Wert der Variablen oder der begrenzenden Konstanten entscheidet, ob der Schleifenkörper einmal mehr ausgeführt wird oder nicht. Also: Verwenden Sie nur integrale Datentypen wie `int`, `size_t` oder `char` als Laufvariable in Schleifen!

Kommaoperator

Der Kommaoperator wird gelegentlich im Bedingungs- oder Veränderungsteil einer `for`-Schleife benutzt, um die Schleife kompakter zu schreiben, meistens mit dem Ergebnis einer schlechteren Lesbarkeit. Er gibt eine Reihenfolge von links nach rechts vor. Das folgende Programmstück summiert die Zahlen 1 bis 100:

```
int sum {0};
for (int i = 1; i <= 100; ++i) {
  sum += i;
}
```

Mit Hilfe des Kommaoperators wird der Schleifenkörper in den Veränderungsteil verlegt.

```
int sum {0};
for (int i = 1; i <= 100; sum += i, ++i);
```

Im Initialisierungsteil wirkt ein Komma *nicht* wie der Kommaoperator. Vielmehr wird eine neue Variable `sum` angelegt. Sie ist lokal innerhalb der Schleife und überdeckt die äußere Variable `sum`. Letztere wird durch die Schleife nicht verändert und bleibt undefiniert. Die innere Variable `sum` ist nach Ablauf der Schleife nicht zugreifbar.

```
int sum;                                               // nicht initialisiert
for (int i = 1, sum = 0; i <= 100; sum += i, ++i);     // falsch
```

Der Kommaoperator hat die niedrigste Priorität von allen Operatoren (siehe Tabelle A.4 auf Seite 964). Verwenden Sie ihn sparsam.

1.8.6 Kontrolle mit break und continue

`break` und `continue` sind Sprunganweisungen. Bisher wurde `break` in der `switch`-Anweisung verwendet, um sie zu verlassen. `break` wirkt genauso in einer Schleife, das heißt, dass die Schleife beim Erreichen von `break` beendet wird. `continue` hingegen überspringt den Rest des Schleifenkörpers. In einer `while`- oder `do while`-Schleife würde als Nächstes die Bedingung geprüft werden, um davon abhängig die Schleife am Beginn des Schleifenrumpfs fortzusetzen. Das folgende kleine Menü-Programm zeigt `break` und `continue`.

Listing 1.18: Menü mit `break` und `continue` *(cppbuch/k1/menu.cpp)*

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

int main()
{
  while (true) {                          // unendliche Schleife
    char c {'x'};
    cout << "Wählen␣Sie:␣a,␣b,␣x␣=␣Ende␣:␣";
    cin >> c;
    if (c == 'a') {
      cout << "Programm␣a\n";
      continue;                           // zurück zur Auswahl
    }
    if (c == 'b') {
      cout << "Programm␣b\n";
      continue;                           // zurück zur Auswahl
    }
    if (c == 'x') {
      break;                              // Schleife verlassen
    }
    cout << "Falsche␣Eingabe!␣Bitte␣wiederholen!\n";
  }
  cout << "\n␣Programmende␣mit␣break\n";
}
```

In einer `for`-Schleife würde als Nächstes die Veränderung ausgeführt und dann erst die Bedingung erneut geprüft. Insofern stimmt die oben erwähnte Äquivalenz der `for`-Schleife mit einer `while`-Schleife exakt nur für `for`-Schleifen ohne `continue`. Ohne `break` und `continue` wären gegebenenfalls viele `if`-Abfragen notwendig, die die Lesbarkeit eines Programms verschlechtern.

In einem größeren Programm können viele verteilte `break`- oder `continue`-Anweisungen die Verständlichkeit beeinträchtigen. Deshalb gibt es die Meinung, dass jeder Block nur einen einzigen Einstiegs- und einen einzigen Ausstiegspunkt haben soll (englisch *single entry/single exit*). Um dies zu erreichen, wird eine Hilfsvariable eingeführt, die den Abbruch signalisiert (siehe Übungsaufgabe 1.16).

Eine Alternative besteht darin, `break`-Anweisungen mit einem deutlichen Kommentar wie zum Beispiel `// EXIT!` am rechten Rand zu kennzeichnen. Wenn eine Schleife mit `break` nur wenige Zeilen umfasst, trägt eine Hilfsvariable nicht zur Übersichtlichkeit bei.

Übungen

1.12 Schreiben Sie eine Schleife, die eine gegebene Zahl binär ausgibt, indem Sie mit geeigneten Bit-Operationen prüfen, welche Bits der Zahl gesetzt sind. Tipp: Verwenden Sie die Zahl 1, verschoben um 0 bis z.B. 31 Bit-Positionen, als »Maske«. Mögliche Ergebnisse können sein:

5 → `00000000000000000000000000000101`

-5 → `11111111111111111111111111111011`

Es ist zu sehen, dass `-5` intern als Zweierkomplement von 5 dargestellt wird.

1.13 Welche Fallstricke sind in den folgenden Schleifen verborgen? Dabei soll auch darauf geachtet werden, unter welchen Umständen die Schleifen terminieren (sich beenden).

a)
```
while (i > 0)
    k = 2 * k;
```

b)
```
while (i != 0)
    i = i - 2;
```

c)
```
while (n != i) {
    ++i;
    n = 2 * i;
}
```

1.14 Fünf Personen haben versucht, die Summe der Zahlen von 1 bis 100 zu berechnen. Beurteilen Sie die folgenden Lösungsvorschläge:

(a)
```
int n = 1, sum = 0;
while (n <= 100) {
  ++n;
  sum += n;
}
```

(b)
```
int n {1};
while (n < 100) {
  int sum = 0;
  n = n + 1;
  sum = sum + n;
}
```

(c)
```
int n = 1, sum = 1;
while (n < 100)
  n += 1;
  sum += n;
```

(d)
```
int n = 1, sum = 0;
while (n <= 0100) {
  sum += n;
  ++n;
}
```

(e)
```
int n {100};
int sum {n*(n+1)/2};
```

1.15 Berechnen Sie die Summe aller natürlichen Zahlen von `n1` bis `n2` mit a) einer `for`-Schleife, b) einer `while`-Schleife, c) einer `do while`-Schleife, d) ohne Schleife. Es sei `n2` $\geq$ `n1` vorausgesetzt. Tipp: Erst die vorstehende Aufgabe lösen.

1.16 Formulieren Sie das Programm in Listing 1.18 (Seite 82) um, sodass ein funktionsäquivalentes Programm ohne `continue` in der Schleife entsteht. Anstelle der `if`-Anweisungen soll eine `switch`-Anweisung eingesetzt werden, um das Programm zu verkürzen.

1.8.7 goto

Die goto-Anweisung bewirkt einen Sprung. Das Sprungziel wird *Label* genannt. goto soll nur dann eingestzt werden, wenn die Absicht mit strukturierten Anweisungen wie for, break usw. nur umständlich erreicht werden kann. Der Grund: Ersatz dieser Anweisungen durch goto führt bei Unerfahrenen leicht zu verworrenen und schwer verständlichen Programmabläufen, Spaghetticode genannt. Beispiel für den Ersatz einer for-Schleife:

```
for (int i{0}; i < 5; ++i) {
  cout << i << '␣';
}
// Mit goto realisiert:
int i{0};
anfang:                                    // Label
if (i < 5) {
  cout << i << '␣';
  ++i;
  goto anfang;
}
```

Nur in wenigen Fällen ist goto sinnvoll. Zum Beispiel kann mit break nur aus einer Schleifenebene herausgesprungen werden. Der Sprung aus einer verschachtelten Schleife ist mit goto leichter realisierbar.

```
// Beenden einer verschachtelten Schleife mit goto:
int sum{0};
for (int i{0}; i < 100; ++i) {
  for (int j{0}; j < 100; ++j) {
    sum += i + j;
    if (sum > 10000) {
      goto ende;
    }
  }
}
ende:
cout << sum << '\n';
```

Um goto zu vermeiden, kann man eine Abbruchbedingung formulieren, die auf jeder Schleifenebene abgefragt wird:

```
// Beenden einer verschachtelten Schleife mit Abbruchbedingung:
int sum1{0};
bool ende{false};
for (int i{0}; i < 100 && !ende; ++i) {
  for (int j{0}; j < 100 && !ende; ++j) {
    sum1 += i + j;
    ende = sum1 > 10000;
  }
}
cout << sum1 << '\n';
```

1.9 Selbst definierte und zusammengesetzte Datentypen

Außer den Grunddatentypen gibt es Datentypen, die aus den Grunddatentypen zusammengesetzt sind und die Sie selbst definieren können (benutzerdefinierte Datentypen). Darüber hinaus gibt es zusammengesetzte Datentypen in der C++-Bibliothek, von denen hier die sehr gebräuchlichen Datentypen `vector` und `string` beschrieben werden.

1.9.1 Aufzählungstypen

Häufig gibt es nicht-numerische Wertebereiche. So kann zum Beispiel ein Wochentag nur die Werte Sonntag, Montag, Dienstag, Mittwoch, Donnerstag, Freitag und Samstag annehmen. Oder ein Farbwert in einem C++-Programm soll nur den vier Farben Rot, Grün, Blau, und Gelb entsprechen. Eine mögliche Hilfskonstruktion wäre die Abbildung auf den Datentyp `int`:

```
int eineFarbe;                          // rot = 0
                                        // grün = 1
                                        // blau = 2
                                        // gelb = 3

int einWochentag;                       // Sonntag = 0
                                        // Montag = 1 usw.
```

Dieses Verfahren hätte einige Nachteile:

- Die Bedeutung muss im Programm als Kommentar festgehalten werden.
- Zugeordnete Zahlen sind nicht eindeutig: 0 kann rot, aber auch Sonntag bedeuten, und 1 kann für grün oder auch Montag stehen.
- Schlechter Dokumentationswert, zum Beispiel `if (eineFarbe == 2) ...`
  Hieraus ist nicht zu ersehen, welche Farbe gemeint ist. Oder:
  `if (eineFarbe == 5) ...` Der Wert 5 ist undefiniert!

Die Lösung für solche Fälle sind die *Aufzählungs- oder Enumerationstypen*, die eine selbst definierte Erweiterung der vordefinierten Typen darstellen. Abbildung 1.13 zeigt das etwas vereinfachte[9] Syntaxdiagramm einer `enum`-Deklaration.

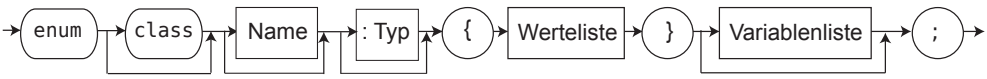


Abbildung 1.13: Vereinfachtes Syntaxdiagramm einer `enum`-Deklaration

Der Name wird als Typname bei folgenden Deklarationen verwendet. Wenn der Typ samt vorangestelltem Doppelpunkt weggelassen wird, gilt automatisch `int`. Ebenso kann die Variablenliste weggelassen werden, etwa um Variablen des Aufzählungstyps erst später anzulegen. `class` wegzulassen ist möglich, wird aber nicht empfohlen (siehe unten). Der neue Datentyp `Farbtyp` wird deklariert:

[9] Der (integrale) Typ der Werte in der Werteliste kann festgelegt werden, auch ist `struct` statt `class` möglich.

```
enum class Farbtyp { rot, gruen, blau, gelb};
```

Der neue Datentyp `Wochentag` wird deklariert:

```
enum class Wochentag {sonntag, montag, dienstag, mittwoch,
                      donnerstag, freitag, samstag};
```

Wenn der Datentyp erst einmal bekannt ist, können weitere Variablen definiert werden:

```
Farbtyp gewaehlteFarbe;                    // Definition
Wochentag derFeiertag;                     // Definition
Wochentag heute {Wochentag::dienstag};     // Definition + Initialisierung
```

Falls ein Aufzählungstyp nur ein einziges Mal in einer Variablendefinition benötigt wird, kann der Name weggelassen werden. Das Ergebnis ist eine *anonyme Typdefinition:*

```
enum {fahrrad, mofa, lkw, pkw} einFahrzeug;     // ohne class und Name
```

Den mithilfe von Aufzählungstypen definierten Variablen können ausschließlich Werte aus der zugehörigen Liste zugewiesen werden, Mischungen sind nicht erlaubt. Aufzählungstypen sind eigene Datentypen, werden aber intern auf die natürlichen Zahlen abgebildet, beginnend bei 0, wenn nichts anderes angegeben wird. Eine Voreinstellung mit beliebigen negativen oder positiven Zahlen ist möglich, wird aber nur gelegentlich erforderlich sein, vielleicht bei einer gewünschten Codierung als Bitmaske, zum Beispiel:

```
// Abweichung von der Standardeinstellung 0, 1, 2, 3 ...:
// Deklaration:
enum class Farbtyp {rot = 0, gruen = 1, blau = 2, gelb = 4};

// Deklaration mit Variablendefinition:
enum class Palette { weiss = 0, grau = 1, braun = 2, amber = 4, lila = 8
                   } mischung;
```

Die in `Farbtyp` definierten Farben `rot`, `gruen`, `blau`, `gelb` dürfen in `Palette` nicht mehr verwendet werden, wenn `class` fehlt und sich `Palette` im gleichen Gültigkeitsbereich befindet. Eine direkte Umwandlung in `int` ist nicht erlaubt, wohl aber eine mit expliziter Typumwandlung, vorzugsweise mit dem `static_cast`-Operator. Die nächsten Zeilen zeigen weitere richtige oder falsche Anweisungen, wobei die Variablennamen sich auf die obigen Definitionen beziehen:

```
int j = Wochentag::dienstag;                    // Fehler! Typ nicht kompatibel
int i = static_cast<int>(Wochentag::dienstag);  // jetzt ok
Wochentag morgen = Wochentag::montag;           // richtig
morgen = i;                                     // Fehler, Datentyp inkompatibel
morgen = static_cast<Wochentag>(i);             // erlaubt, aber...
// Falls die Variable i einen Wert hat, der nicht einem der Werte des
// Aufzählungstyps entspricht, ist der Wert der Variablen morgen undefiniert!
Wochentag::montag = morgen;                     // Fehler (montag ist Konstante)
++mischung;                                     // Fehler, undefinierter Operator
if (mischung > Palette::grau) {                 // erlaubter Vergleich
   mischung = Palette::lila;                    // richtig
}
```

class weglassen?

Das Schlüsselwort `class` wegzulassen ist weiterhin gültiges C++, nur ist die Typsicherheit eingeschränkt, weil implizite Umwandlungen nach `int` möglich sind. Ein Beispiel:

```
enum Farbtyp1 { rot, gruen, blau, gelb };       // ohne class
int i = rot + blau;                             // nun möglich
```

Außer der impliziten Umwandlung von `rot` und `blau` nach `int` fällt auf, dass die Typqualifizierung `Farbtyp1::` fehlen kann.

Tipp

Wegen der erhöhten Typsicherheit wird empfohlen, `enum class` zu verwenden und nur in begründeten Ausnahmefällen darauf zu verzichten.

Schreiberleichterung mit using enum

Wenn Sie mehrere `enum class`-Werte eines Aufzählungstyps verwenden, muss der Aufzählungstyp `Farbtyp` mit angegeben werden, zum Beispiel

```
switch(farbe) {
  case Farbtyp::rot   : cout << "rot!\n";  break;
  case Farbtyp::gruen : cout << "grün!\n"; break;
  case Farbtyp::blau  : cout << "blau!\n"; break;
  case Farbtyp::gelb  : cout << "gelb!\n";
}
```

Das ist umständlich zu schreiben und nicht gut lesbar. Einfacher und verständlicher ist es, mit `using enum` den geforderten Typ vorher zu benennen, wie Listing 1.19 zeigt.

Listing 1.19: Beispiel mit `using enum` *(cppbuch/k1/usingenum.cpp)*

```
#include <iostream>
using namespace std;
enum class Farbtyp { rot, gruen, blau, gelb };

int main()
{
  Farbtyp farbe = Farbtyp::rot;              // oder gruen usw.
  switch (farbe) {
    using enum Farbtyp;                      // !
     case rot   : cout << "rot!\n"; break;
     case gruen : cout << "grün!\n"; break;
     case blau  : cout << "blau!\n"; break;
     case gelb  : cout << "gelb!\n";
  }
}
```

Beispiel std::byte

Es gibt im Namensraum `std` einen Typ für Bytes: `byte`. Er ist im Header `<cstddef>` (Seite 954) mithilfe von `enum class` definiert:

```
enum class byte : unsigned char {};
```

Der Typ der Aufzählungswerte ist als `unsigned char` festgelegt. `byte` kann für Zugriff auf »rohen« Speicher genommen werden. Das Wort »roh« bedeutet in diesem Zusammenhang, dass der Speicher eben nicht als Inhalt eines Datentyps interpretiert wird. Ein Byte ist kein Zeichen, nur eine Sammlung von Bits. Die bei ganzen Zahlen möglichen Bitoperationen des letzten Teils der Tabelle 1.1 von Seite 45 gelten auch für `byte`. Sie sind ebenfalls im Header `<cstddef>` definiert.

1.9.2 Strukturen

Häufig möchte man logisch zusammengehörige Daten zusammenfassen, die nicht vom selben Datentyp sind, zum Beispiel die zusammengehörigen Daten eines Punkts auf dem Bildschirm, also seine Koordinaten `x` und `y`, seine Farbe und ob er sichtbar ist. Für diesen Zweck stellt C++ die *Struktur* bereit. Sie fasst einen Datensatz zusammen. Eine Struktur im Sinne dieses Abschnitts enthält ausschließlich Daten (und keine Funktionen, was auch möglich wäre). Strukturen sind benutzerdefinierte Datentypen. Sie werden mit einer Syntax definiert, die bis auf den inneren Teil der Syntax von Aufzählungstypen ähnelt (siehe Abbildung 1.14).

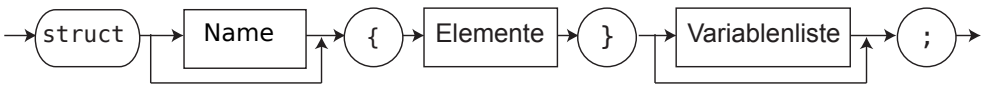


Abbildung 1.14: Syntaxdiagramm einer `struct`-Definition

In einem Grafikprogramm gehören zu jedem Punkt auf dem Bildschirm verschiedene Daten wie die Koordinaten in x- und y-Richtung, die Farbe und die Information, ob er gerade sichtbar ist. Alle diese logisch zusammengehörenden Daten werden in der Struktur `Punkt` zusammengefasst. Die strukturinternen Daten heißen *Elemente* (auch *Attribute*, *Felder* oder *Komponenten*). Im Beispiel werden zwei Variablen `p` und `q` vom Datentyp `Punkt` definiert und der Zugriff auf die internen Größen gezeigt.

```
enum class Farbtyp {rot, gelb, gruen};

struct Punkt {                  // Punkt ist ein Typ
  int x;
  int y;
  bool istSichtbar;
  Farbtyp dieFarbe;
} p;                            // p ist ein Punkt-Objekt, noch undefinierter Inhalt

// q ist ebenfalls ein Punkt-Objekt:
Punkt q;                        // noch undefinierter Inhalt
// Zugriff, hier: mit Werten versehen
p.x = 270;                      // x-Koordinate von p
p.y = 20;                       // y-Koordinate von p
p.istSichtbar = false;
p.dieFarbe    = Farbtyp::gelb;
cout << "p.x␣=␣" << p.x << "␣␣p.y␣=␣" << p.y
       << "␣␣p.istSichtbar=␣" << p.istSichtbar
       << "␣␣p.dieFarbe=␣" << static_cast<int>(p.dieFarbe) << '\n';
```

Die internen Elemente sind nicht allein zugreifbar, weil sie nur in Verbindung mit einem Objekt existieren. Die Anweisung `dieFarbe = Farbtyp::rot;` ist unsinnig, weil nicht klar ist, welcher Punkt rot werden soll. `p.dieFarbe = Farbtyp::rot;` hingegen sagt eindeutig, dass der Punkt `p` gefärbt werden soll. Der Zugriff geschieht über einen Punktoperator ».« zwischen Variablen- und Elementnamen. Bei der Ausgabe wird der `bool`-Wert `false` in 0 umgewandelt und der `enum`-Wert `gelb` in 1 (`rot` ergäbe 0).

Bestimmte einfache Datenstrukturen heißen *Aggregate*. Dazu gehören Arrays, die Sie weiter unten kennenlernen werden, und einfache `struct`-Typen. Die Struktur `Punkt` ist auch ein Aggregat. Ein Aggregat wird mit einer Liste in geschweiften Klammern initialisiert, innerhalb derer die Elemente durch ein Komma getrennt aufgeführt werden. Nicht initialisierte Attribute werden mit 0 oder einem 0 entsprechendem Wert initialisiert, zum Beispiel `false` bei einem `bool`-Attribut. Das ist für alle `struct`-Elemente der Fall, wenn mit `{}` initialisiert wird, also einer leeren Liste. Das Listing 1.20 zeigt verschiedene Möglichkeiten.

Listing 1.20: Aggregat-Initialisierungen *(cppbuch/k1/aggregatinit.cpp)*

```
#include <iostream>
using namespace std;
enum class Farbtyp { rot, gelb, gruen };

struct Punkt {
  int x;
  int y;
  bool istSichtbar;
  Farbtyp dieFarbe;
};

int main()
{
  constexpr Punkt p1 {100, 200, false, Farbtyp::gelb};
  constexpr Punkt p2 {};                                          // alles 0
  constexpr Punkt p3 { .x = 10, .istSichtbar = true, .dieFarbe = Farbtyp::gruen};
  for(int nr = 1; auto p : {p1, p2, p3}) {                        // Liste von 3 Punkten
    cout << nr++ << ":␣" << p.x << "␣␣" << p.y  << "␣␣" << p.istSichtbar
         << "␣␣" << static_cast<int>(p.dieFarbe) << '\n';
  }
}
```

Das Listing 1.20 zeigt zwei Besonderheiten: Erstens wird `y` für `p3` nicht eigens initialisiert und erhält deswegen den Wert 0. Und zweitens wird in der Schleife eine Liste der drei Punkte abgearbeitet.

1.9.3 Der C++-Standardtyp vector

Im täglichen Leben benutzen wir häufig Tabellen, oft Arrays (engl. für Reihe oder Feld) genannt. Einspaltige Tabellen, und um die geht es hier zunächst, werden in C++ durch eine *Vektor* genannte Konstruktion abgebildet. Aus der Sprache C sind primitive Arrays bekannt. Diese sind zwar auch Bestandteil von C++, aber nicht besonders komfortabel und werden daher erst in Abschnitt 4.2 behandelt.

Ein *Vektor ist eine Tabelle von Elementen desselben Datentyps*, also eine Tabelle zum Beispiel nur mit ganzen Zahlen oder nur mit `double`-Zahlen. Mit Ausnahme von Referenzen kann der Datentyp beliebig sein, insbesondere kann er selbst wieder zusammengesetzt sein. Auf ein Element der Tabelle wird über die *Positionsangabe* zugegriffen, also über die Nummer der Reihe, in der sich das Element befindet.

In C++ ist ein Vektor eine vordefinierte Klasse, um deren internen Aufbau wir uns erst später kümmern. Zunächst geht es nur um die Benutzung als Baustein in eigenen Programmen. Es kommen dabei zwei Sichtweisen zum Ausdruck:

1. Bausteine werden benutzt, um Programme oder Programmteile zu entwickeln, die selbst wieder Bausteine sein können. Dazu ist die Kenntnis notwendig, was ein Baustein leistet und wie er verwendet werden muss, aber nicht wie er intern funktioniert.
2. Bausteine sollen von Grund auf entwickelt oder weiterentwickelt werden. Dann ist eine gründliche Kenntnis der inneren Funktion unerlässlich.

Für alle, die Software entwickeln, sind beide Sichten wichtig. Hier wird der Standardtyp `vector` zunächst nur benutzt, und erst später wird erklärt, was im Innern vorgeht. Die Anweisung

```
vector<int> v(10);                          // runde Klammern
```

stellt einen Vektor `v` bereit, der aus zehn Elementen des Typs `int` besteht. Alle Werte sind 0. Die Feldelemente sind stets von 0 bis (Anzahl der Elemente –1) durchnummeriert, hier also von 0 bis 9. Mit geschweiften Klammern können die Elemente des Vektors direkt angegeben werden:

```
vector<int> v1 {};                          // leerer Vektor
vector<int> v2 {7, 0, 9};                   // Vektor mit den Elementen 7, 0, 9
```

Wenn der Typ des Vektors aus den Elementen abgeleitet werden kann, ist die explizite Typangabe mit `<int>` nicht notwendig:

```
vector v3 {7, 0, 9};                        // Vektor mit den Elementen 7, 0, 9
```

Die Klasse für Vektoren stellt einige Dienstleistungen zur Verfügung, die mit der Notation *Objektname.Anweisung (*gegebenenfalls *Daten)* abgerufen werden können. Eine dieser Dienstleistungen ist die Ermittlung der Zahl der Elemente, das heißt die Größe (englisch *size*) des Vektors:

```
cout << v.size()  << '\n';                  // 10
cout << v1.size() << '\n';                  // 0
cout << v2.size() << '\n';                  // 3
```

Daten müssen in diesem Fall nicht zwischen den runden Klammern übergeben werden, das Vektor-Objekt enthält alle nötigen Informationen. Alternativ kann `std::size()` verwendet werden:

```
cout << size(v2) << '\n';
```

Der Zugriff auf ein spezielles Element wird mit dem Indexoperator `[]` realisiert. Zum Beispiel zeigt

```
cout << v[0] << '\n';                       // Zählung beginnt bei 0
```

das erste Element des Vektors an. Der zwischen den eckigen Klammern stehende Wert heißt *Index*. Zugriffe auf Vektor-Elemente kommen typischerweise in Schleifen vor, weil meistens eine Operation für die gesamte Tabelle ausgeführt werden soll. Dabei ist sorgfältig darauf zu achten, dass die Indexwerte die Vektorgrenzen nicht unter- oder überschreiten.

Vorsicht Falle!

Es gibt keine Überprüfung der Bereichsüber- oder -unterschreitung! Zugriffe auf nicht existierende Elemente wie `c = v[100]` erzeugen *keine* Fehlermeldung, weder durch den Compiler noch zur Laufzeit, sofern nicht der zulässige Speicherbereich für das Programm überschritten wird! Der Grund: Schnelligkeit ist ein Designprinzip von C++ und die Überprüfung eines jeden Zugriffs kostet eben Zeit.

Umgehen der Falle

Man kann auf die eckigen Klammern verzichten und einen Vektor nach einem Wert *an* (englisch *at*) einer Position fragen. Diese Art des Zugriffs wird geprüft:

```
cout << v.at(0)    << '\n';                  // alles bestens, dasselbe wie v[0]
// 1000 ist zuviel!
cout << v.at(1000) << '\n';                  // Programmabbruch mit Fehlermeldung
```

Im Beispiel unten wird dafür gesorgt, dass der Index niemals einen falschen Wert haben kann – dies ist sowieso ein besseres Verfahren, als das Programm zu korrigieren, nachdem das Kind in den Brunnen gefallen ist. Der laufende Index wird einfach mit `v.size()` verglichen. `v.size()` gibt die Anzahl der Elemente als nicht-vorzeichenbehafteten Wert zurück. Natürlich muss in der `for`-Schleife `i < v.size()` abgefragt werden anstatt `i <= v.size()`! Wer sich leicht vertippt, schreibt also besser `v.at(i)` statt `v[i]`.

Ein Beispiel

Das Programm unten verdeutlicht die Arbeitsweise mit Vektoren. Es werden einige typische Operationen demonstriert, die sich weitgehend selbst erklären.

Listing 1.21: Standardklasse `vector` *(cppbuch/k1/vektor.cpp)*

```
#include <algorithm>                         // enthält Sortierfunktion sort()
#include <cstddef>                           // size_t
#include <iostream>
#include <vector>                            // Standardvektor bekannt machen
using namespace std;

int main()                                   // Programm mit typischen Vektoroperationen
{
  vector<double> kosten(12);                 // Tabelle mit 12 double-Werten
  // Füllen der Tabelle mit beliebigen Daten, dabei Typumwandlung int → double.
  // Die Funktion size() gibt eine Zahl vom Typ size_t zurück.
  // Deshalb hat die Laufvariable i denselben Typ.
  for (size_t i = 0; i < kosten.size(); ++i) {
    kosten[i] = static_cast<double>(150 - i * i) / 10.0;
  }
```

```
  for (size_t i = 0; i < kosten.size(); ++i) {  // Tabelle ausgeben
    cout << i << ":␣" << kosten[i] << '\n';
  }
  // Berechnung und Anzeige von Summe und Mittelwert
  double sum = 0.0;
  for (size_t i = 0; i < kosten.size(); ++i) {
    sum += kosten[i];
  }
  cout << "Summe␣=␣" << sum << '\n';
  cout << "Mittelwert␣=␣" << sum / kosten.size() << '\n';
                      // implizite Typumwandlung des Nenners nach double
  // Maximum anzeigen
  double maxi{kosten[0]};
  for (size_t i = 1; i < kosten.size(); ++i) {
    if (maxi < kosten[i]) {
      maxi = kosten[i];
    }
  }
  cout << "Maximum␣=␣" << maxi << '\n';

  // zweite Tabelle sortierteKosten deklarieren und mit der ersten initialisieren
  vector<double> sortierteKosten{kosten};
  // zweite Tabelle aufsteigend sortieren
  ranges::sort(sortierteKosten);
  // und mit der laufenden Nummer ausgeben
  for (size_t i = 0; i < sortierteKosten.size(); ++i) {
    cout << i << ":␣" << sortierteKosten[i] << '\n';
  }
  // Kurzform, wenn die lfd. Nummer nicht gebraucht wird:
  for (auto diekosten : sortierteKosten) {
    cout << diekosten << '\n';
  }
}
```

Die Bibliotheksfunktion `std::ranges::sort()` ist sehr schnell. Die im Listing 1.21 verwendete Variante ist im Namespace `std::ranges`, weswegen `ranges::` vor `sort()` geschrieben wird. Eine weitere `sort()`-Variante ist im Namespace `std`, deren Aufruf ist aber umständlicher, weil Anfang und Ende des zu sortierenden Bereichs angegeben werden müssen: `sort(sortierteKosten.begin(), sortierteKosten.end())`. An der Stelle

```
vector<float> sortierteKosten {kosten};      // Objekt anlegen und initialisieren
```

wäre auch Folgendes möglich gewesen:

```
vector<float> sortierteKosten;               // Objekt anlegen
sortierteKosten = kosten;                    // Zuweisung
```

Wenn man so wie hier die Wahl hat, soll stets der ersten Variante der Vorzug gegeben werden, weil das Initialisieren während der Objekterzeugung schneller vonstatten geht, als erst das Objekt zu erzeugen und dann im zweiten Schritt die Zuweisung der Werte vorzunehmen.

Lineare Suche in Tabellen

Hier seien vier programmiertechnische Möglichkeiten gezeigt, in einer Tabelle `tabelle` mit n Elementen auf den Positionen $0...n-1$ ein bestimmtes Element `key` zu suchen. Die Tabelle kann sortiert oder unsortiert sein. Die C++-Standardbibliothek bietet die `find()`-Funktion, aber hier sollen programmiertechnisch verschiedene Schleifenvarianten verglichen werden. Die Variable `i` gibt anschließend die Position an, an der das gesuchte Element `key` erstmalig auftritt. Falls `key` nicht in `tabelle` enthalten ist, muss `i` einen Wert außerhalb $0..(n-1)$ annehmen. Die folgenden Algorithmen enden bei erfolgloser Suche mit `i` = n. Die letzte Variante erlaubt eine kürzere Formulierung der Schleife, setzt aber voraus, dass das Feld um einen Eintrag erweitert wird, der als »Wächter« (englisch *sentinel*) für den Abbruch der Schleife dient.

```
// Definitionen für die nächsten Fälle
const int n = ...                                   // ggf. constexpr
vector<int> tabelle(n);
int key = ...                                       // gesuchtes Element
int i {n};          // Laufvariable. Ergebnis: i = 0..n – 1 : gefunden, i = n : nicht gefunden!
```

1. `while`-Schleife
   In der Bedingung wird abgefragt, ob die Zählvariable noch im gültigen Bereich und das aktuelle Element ungleich `key` ist. So lange wird die Zählvariable inkrementiert.

   ```
   i = 0;
   while (i < n && tabelle[i] != key) {
     ++i;
   }
   ```

2. `do while`-Schleife
   Wie oben, nur dass die Zählvariable *vorher* inkrementiert und daher anders vorbesetzt wird. Die vorhergehende Lösung soll im Vergleich bevorzugt werden, weil es generell besser ist, eine Bedingung zu prüfen und dann zu handeln, als umgekehrt.

   ```
   i = -1;
   do {
     ++i;
   } while (i < n && tabelle[i] != key);
   ```

3. `for`-Schleife
   Die Schleife wird mit `break` verlassen, wenn das Element gefunden wird. Es wäre auch möglich gewesen, die Bedingung `i < n` zu erweitern.

   ```
   for (i = 0; i < n; ++i) {
     if (tabelle[i] == key) {
       break;
     }
   }
   ```

4. (n+1). Element als »Wächter« (sentinel)
   Achtung! Die Definition muss hier das zusätzliche Element berücksichtigen, sodass die Tabelle (n+1) statt n Elemente enthält:

   ```
   vector<int> tabelle(n+1); // letztes Element nur für diesen Fall
   ```

In das zusätzliche Element `tabelle[n]` wird `key` eingetragen. Die Schleife muss spätestens hier abbrechen, auch wenn `key` vorher nicht gefunden wurde. Die Zählvariable wird als Seiteneffekt beim Zugriff auf ein Vektorelement hochgezählt.

```
i = -1;
tabelle[n] = key;                              // garantiert Abbruch der Schleife
while (tabelle[++i] != key);
```

Vektoren sind dynamisch!

Oft ist unklar, wie groß ein Vektor sein soll, zum Beispiel beim Einlesen von Daten per Dialog oder aus einer Datei unbekannter Größe. Ein Vektor der C++-Standardbibliothek hat den Vorteil, dass er bei Bedarf Elemente hinten anhängt und dabei seine Größe ändert. Falls »hinten« kein Platz mehr im Vektor sein sollte, wird der gesamte Vektor automatisch an eine neue, ausreichend große Stelle im Speicher verlagert. Mit `push_back` wird der Vektor dazu aufgefordert, ein Element anzufügen, wobei ihm der anzuhängende Wert in runden Klammern übergeben wird. Eine tabellarische Übersicht der Möglichkeiten von Objekten der Klasse `vector` ist in Abschnitt 27.2.1 zu finden. Ein Großteil der Möglichkeiten ist aber erst nach Kenntnis der folgenden Kapitel bis einschließlich Kapitel 8 verständlich. Das Listing 1.22 zeigt, wie ein Vektor entsprechend zur Anzahl der eingegebenen Daten wächst. Die Schleife bricht ab, wenn ein Wert eingegeben wird, der in den abzuspeichernden Daten nicht vorkommt.

Listing 1.22: Vektor dynamisch vergrößern *(cppbuch/k1/dynvekt.cpp)*

```
#include <cstddef>
#include <iostream>
#include <vector>                              // Standardvektor einschließen
using namespace std;

int main()
{
  vector<int> meineDaten;                      // anfängliche Größe ist 0
  cout << "Bitte␣Werte␣>=␣0␣eingeben\n"; // negative Werte sollen nicht vorkommen
  int wert{0};
  while (true) {                               // Abbruch siehe unten
    cout << "Wert␣(<␣0␣=␣Ende␣der␣Eingabe):";
    cin >> wert;
    if (wert < 0) {
      break;                                   // ABBRUCH der Schleife
    }
    meineDaten.push_back(wert); // Wert anhängen
  }
  cout << "Es␣wurden␣die␣folgenden␣Werte␣eingegeben:\n";
  for (size_t i = 0; i < meineDaten.size(); ++i) {
    cout << i << ".␣Wert␣:␣" << meineDaten[i] << '\n';
  }
}
```

1.9.4 Der C++-Standardtyp array

Zu den Vektoren des vorigen Abschnitts gibt es eine Alternative: die Standardklasse `array` (Header `<array>`). Ein `array` ist nicht dynamisch, d.h. im Unterschied zur Klasse `vector` wird der Speicherplatz schon bei der Definition festgelegt. Das Listing 1.23 zeigt die Anwendung.

Listing 1.23: Alternative `array` statt `vector` *(cppbuch/k1/array.cpp)*

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

int main()
{
  array<double, 5> arr1;                // 5 undefinierte double-Zahlen
  for (size_t i = 0; i < arr1.size(); ++i) {
    cout << arr1.at(i) << '␣';
  }
  cout << '\n';
  array<double, 4> arr2{1.2, 2.3};      // 4 double-Zahlen 1.2, 2.3, 0.0, 0.0
  for (size_t i = 0; i < arr2.size(); ++i) {
    cout << arr2.at(i) << '␣';
  }
  cout << '\n';
  array arr3{0, 1, 2, 3};               // 4 int-Zahlen 0, 1, 2, 3
  for (size_t i = 0; i < arr3.size(); ++i) {
    cout << arr3[i] << '␣';             // [i] statt at(i) ist auch möglich
  }
  cout << '\n';
}
```

Bei der Deklaration werden der Datentyp und die gewünschte Anzahl in spitzen Klammern angegeben (siehe `arr1` in Listing 1.23). Das Array kann mit Werten vorbesetzt werden. Werden weniger Werte angegeben, als Platz vorhanden ist, werden die restlichen mit 0 initialisiert (siehe `arr2`). Wenn das Array genau so viele Werte enthalten soll, wie in der Initialisierungsliste angegeben, kann der Compiler den Typ und die Anzahl ableiten. Die Angaben in spitzen Klammern können dann entfallen (siehe `arr3`).

1.9.5 Zeichenketten: der C++-Standardtyp string

Eine Zeichenkette, auch String genannt, ist aus Zeichen des Typs `char` zusammengesetzt. Im Grunde kann eine Zeichenkette wie eine horizontale Tabelle mit nur einer Reihe aufgefasst werden. Dennoch wird nicht ein `vector<char>`, sondern eine andere Standardklasse mit dem Namen `string` als Baustein verwendet, ohne dass wir uns um ihre Innereien kümmern – jedenfalls jetzt noch nicht. Die Klasse hat gegenüber dem uns bekannten Vektor einige zusätzliche Eigenschaften, von denen eine Auswahl im folgenden Programm beispielhaft gezeigt werden soll. Eine Übersicht der Möglichkeiten von Strings ist in Kapitel 31 zu finden.

Listing 1.24: Standardklasse `string` *(cppbuch/k1/zbstring.cpp)*

```
#include <cstddef>
#include <iostream>
#include <string>                          // Standard-String einschließen
using namespace std;
int main()        // Programm mit typischen String-Operationen
{
  // String-Objekt einString anlegen und mit "hallo" initialisieren.
  string einString {"hallo"};              // einString kann ein beliebiger Name sein.
  cout << einString << '\n';               // String ausgeben

  // Beim Vektor wäre stattdessen für die Ausgabe eine Schleife notwendig.
  // Die Anzahl der Zeichen kann bei Strings mit length() ermittelt werden.
  // String zeichenweise ausgeben, ungeprüfter Zugriff wie bei vector:
  for (size_t i = 0; i < einString.length(); ++i) {
    cout << einString[i];
  }
  cout << '\n';
  // String zeichenweise mit Indexprüfung ausgeben.
  for (size_t i = 0; i < einString.length(); ++i) {
    cout << einString.at(i);
  }
  cout << '\n';
  // Die Prüfung geschieht wie beim Vektor. Ein Versuch, einString.at(i)
  // mit i ≥ einString.size() abzufragen, führt zum Programmabbruch mit Fehlermeldung.
  string eineStringKopie(einString);     // Kopie des Strings einString erzeugen
  cout << eineStringKopie << '\n';       // hallo
  const string diesIstNeu{"neu!"};       // andere Initialisierung einer Kopie
  eineStringKopie = diesIstNeu;          // Kopie durch Zuweisung
  cout << eineStringKopie << '\n';       // neu!
  eineStringKopie = "Buchstaben";        // Zuweisung einer Zeichenkette
  cout << eineStringKopie << '\n';       // Buchstaben
  einString = 'X';                       // Zuweisung nur eines Zeichens vom Typ char
  cout << einString << '\n';             // X
  einString += eineStringKopie;          // Strings mit dem +=-Operator verketten
  cout << einString << '\n';             // XBuchstaben
  einString = eineStringKopie + "␣ABC"; // Strings mit dem +-Operator verketten
  cout << einString << '\n';             // Buchstaben ABC
  einString = "123" + eineStringKopie;
  cout << einString << '\n';             // 123Buchstaben
  // einString = "123"+ "ABC"; geht nicht! Erklärung folgt in Kapitel 8
   einString = string("123") + "ABC";     // ok!

  // Vergleich von Strings
  const string a{"Albert"};
  const string b{a};
  if (a == b) {
    cout << a << "␣==␣" << b << '\n';
  }
  else {
    cout << a << "␣!=␣" << b << '\n';
  }
```

```
  const string z{"Alberta"};
  if (a < z) {
    cout << a << "␣<␣" << z << '\n';
  }
  if (z > a) {
    cout << z << "␣>␣" << a << '\n';
  }
  if (z != a) {
    cout << z << "␣!=␣" << a << '\n';
  }
  const string str{'a', 'b', 'c'};       // String mit Initialisierungsliste 'a', 'b', 'c'
  cout << "mit␣Initialisierungsliste␣erzeugter␣String:" << str << '\n';
} // Ende von main()
```

Achtung Falle!

Auf Seite 44 wird empfohlen, bei der Definition einer Variablen `auto` verbunden mit einem Initialisierungswert einzusetzen. Das gilt nicht ohne Weiteres für Strings, wie die folgenden Zeilen zeigen.

```
using namespace std;
string einString {"str1"};                  // ein String
auto wasistdas{"str2"};                     // kein String!
auto nocheinString{string("str3")};         // ein String
auto einString2 {"str4"s};                  // ein String
```

`str1` ist ein String, weil der Typ explizit angegeben wird. `str2` ist kein String, weil in Anführungszeichen eingeschlossene Zeichenketten als sogenannte C-Strings aufgefasst werden. C-Strings haben den Typ `const char*` und werden in Kapitel 4 behandelt. `auto` bewirkt daher, dass der Compiler den vorliegenden Datentyp `const char*` für `str2` annimmt. `str3` ist ein String, weil der C-String `"str3"` mit `string("str3")` vor der Interpretation durch den Compiler in einen String umgewandelt wird. In der letzten Zeile bewirkt das Suffix `s` nach `"str3"` die Umwandlung des C-Strings in einen String. Das setzt die Einbindung des Namespace `std` voraus.

Übungen

1.17 Schreiben Sie eine Struktur (`struct`) namens `Person`, die Vorname, Nachname und Alter einer Person enthält. Vorname und Nachname seien vom Typ `string`, Alter vom Typ `int`. Verwenden Sie diese Struktur in einem Programm so, dass den Elementen der Struktur Werte mit `cin` (Eingabe über die Tastatur) zugewiesen werden. Anschließend sollen die Elemente auf dem Bildschirm ausgegeben werden.

1.18 Gegeben sei eine Zeichenkette des Typs `string`, die eine natürliche Zahl darstellen soll und daher nur aus Ziffern besteht. Beispiel: "17462309".

a) Wandeln Sie den String in eine Zahl `z` vom Typ `long` um.

b) Berechnen Sie die Quersumme von `z`.

Geben Sie die Zahl und die Quersumme auf dem Bildschirm aus.

1.19 Etwas schwierig und nur für Knobelfreunde: Schreiben Sie ein Programm, das eine einzugebende natürliche Zahl in römischer Darstellung ausgibt. Die römischen Ziffern

seien in einem konstanten String `zeichenvorrat = "IVXLCDM"` gegeben. Die meistens verwendete syntaktische Regel lautet: Keine Ziffer außer 'M' darf mehr als dreimal hintereinanderstehen. Das heißt, ein vierfaches Vorkommen wird durch Subtraktion vom nächsthöheren passenden Wert ersetzt. Subtraktion geschieht durch Voranstellen des kleineren Werts. So wird 4 nicht zu IIII, sondern zu IV, und 9 wird nicht zu VIIII, sondern zu IX.

1.20 Schreiben Sie ein Programm, das beliebig viele Zahlen im Bereich von -99 bis +100 (einschließlich) von der Standardeingabe liest. Der Zahlenbereich sei in 10 gleich große Intervalle eingeteilt. Sobald eine Zahl außerhalb des Bereichs eingegeben wird, sei die Eingabe beendet. Das Programm soll dann für jedes Intervall ausgeben, wie viele Zahlen eingegeben worden sind. Benutzen Sie für -99, +100 usw. Konstanten (`const` oder `constexpr`). Zur Speicherung der Intervalle soll ein `vector<int>` verwendet werden.

1.21 Das folgende Problem ist klassisch und es haben sich schon viele Menschen damit beschäftigt: wenn Zahlen Achterbahn fahren. Gegeben sei eine natürliche Zahl > 0.

1. Wenn die Zahl gerade ist, teile sie durch 2. Wenn nicht, multipliziere sie mit 3 und addiere 1.
2. Wenn die sich ergebende Zahl größer als 1 ist, wende Schritt 1 auf diese Zahl an. Wenn nicht, ist das Verfahren beendet.

Es zeigt sich, dass die Zahlen erheblich anwachsen können und auch wieder kleiner werden – daher der Name Achterbahn. Schreiben Sie ein Programm, das eine Startzahl als Eingabe erwartet und den obigen Algorithmus durchführt. Lassen Sie sich die erreichte Zahl und das erreichte Maximum anzeigen. Am Ende des Programms soll ausgegeben werden, wie viele Iterationen (Durchläufe der Schleife) bis zum Ende des Programms benötigt werden. Mit den Anweisungen

```
string dummy {""};
getline(cin, dummy);                          // weiter mit Tastendruck
```

können Sie die Ausgabe nach Erreichen eines neuen Höchstwerts anhalten. Versuchen Sie die Startzahlen 4096, 142587, 1501353. Bei der ersten Zahl (4096) ist klar, dass der Algorithmus schnell endet, weil 4096 eine Zweierpotenz ist. Die Frage ist letztlich: Gibt es eine Startzahl, mit der der Algorithmus *nicht* irgendwann endet? Dieses Problem tritt auch unter einer Reihe anderer Namen auf: Syracuse-Problem, Ulams-Problem oder Collatz-Problem. *Hinweis:* Bei großen Zahlen wie der letzten angegebenen wird der `int`-Zahlenbereich überschritten; nehmen Sie stattdessen `long long`.

1.9.6 Container und Schleifen

Ein Container ist eine Datenstruktur, die gleichartige Elemente enthält. So enthält ein Objekt des Typs `vector<double>` Elemente des Typs `double`. Auch ein `string` kann als Container aufgefasst werden: Er enthält Elemente des Typs `char`. Die klassische Zählschleife, um die einzelnen Elemente eines Vektors auszugeben, lautet beispielsweise:

```
for (size_t i = 0; i < einVektor.size(); ++i) {
  cout << einVektor[i] << '\n';
}
```

Schöner ist es, `int` statt `size_t` zu verwenden, obwohl `einVektor.size()` eine Zahl vom Typ `size_t` liefert. Schließlich entsprechen die vorzeichenbehafteten `int`-Zahlen mehr den natürlichen Zahlen, bei `size_t` gibt es nur Werte ≥ 0. Die Programmzeilen

```
for (int i = 0; i < einVektor.size(); ++i) {  // ?
  cout << einVektor[i] << '\n';
}
```

ergeben allerdings eine Warnung des Compilers, weil nun zwei ganze Zahlen verglichen werden, von denen nur `i` ein Vorzeichen hat. Wie Sie von Seite 70 wissen, kann die unbedachte Kombination solcher Zahlen zu Fehlern führen – der Grund für die Warnung. Sie lautet etwa

```
Warnung: Vergleich von Ganzzahlausdrücken, von denen nur einer ein Vorzeichen
hat:  »int« und »std::vector<double>::size_type« {»long unsigned int«}
```

Die Lösung ist die Funktion `ssize()`, die die Größe eines Vektors (und auch anderer Container) als *vorzeichenbehaftete* Zahl zurückgibt. Konkret:

```
for (int i = 0; i < ssize(einVektor); ++i) {
  cout << einVektor[i] << '\n';
}
```

Hinweis 1

Um ganzen Zahlen den Vorzug zu geben, wie von [Str18] empfohlen, wird im Folgenden in der Regel `ssize(X)` statt `X.size()` geschrieben. `X` ist dabei das Objekt, dessen Größe abgefragt werden soll, zum Beispiel ein Vektor.

Hinweis 2

Oft genügt `int` als Typ für die Laufvariable. Bei sehr großen Vektoren muss das jedoch nicht so sein, wenn nämlich der Wert von `size()` den maximal möglichen Wert für `int` überschreitet. Sie sollten dann, oder wenn Sie bezüglich der Größe des Vektors unsicher sind, eine der folgenden Varianten nehmen.

```
// Wenn die Laufvariable i gebraucht wird und eine große Ganzzahl sein soll:
for (auto i = 0z; i < ssize(einVektor); ++i) {
  cout << i << ':' << einVektor[i] << '\n';
}
// unsigned liefert einen noch größeren Bereich. i ist vom Typ size_t.
for (auto i = 0uz; i < einVektor.size(); ++i) {
  cout << i << ':' << einVektor[i] << '\n';
}
```

Auf die Zählvariable kann verzichtet werden, wenn auf alle Elemente des Containers zugegriffen werden soll, weil dann die folgende Kurzform (englisch *range based for*) verwendet werden kann:

```
for (double wert : einVektor) {                  // Kopie jedes Elements
  cout << wert << '\n';
}
```

Man kann die Schleife so lesen: Führe den Schleifenkörper für alle Objekte `wert` in `einVektor` aus. Die lokale Variable `wert` ist dabei eine Kopie des jeweiligen Elements des Containers. Die Kopie kann innerhalb der Schleife verändert werden, ohne dass sich die Änderung auf den Container auswirkt, etwa:

```
for (double wert : einVektor) {                 // veränderliche Kopie
  wert = 2.0 * wert;
  cout << wert << '\n';
}
```

Wenn Änderungen der Kopie verhindert werden sollen, hilft `const`:

```
for (const double wert : einVektor) {           // unveränderliche Kopie
  cout << wert << '\n';
}
```

Wenn die Elemente direkt im Container verändert werden sollen, muss eine Referenz angegeben werden:

```
for (double& wert : einVektor) {                // Referenz zum Ändern
  wert *= 2.0;                                  // alle Werte verdoppeln
}
```

Wenn auf die Elemente nur lesend zugegriffen werden soll, ist eine Kopie bei großen Objekten zeitraubend (bei `double` nicht kritisch). In solchen Fällen kann eine Referenz auf `const` verwendet werden:

```
for (const double& wert : einVektor) {          // Referenz zum Lesen
  cout << wert <<  '\t';
}
```

Man kann die Kurzform, die ohne eine Zählvariable auskommt, trotzdem mit einer Zählvariablen ergänzen:

```
for (int i = 0; double wert : einVektor) {
  cout << "Nr. " << ++i << ": " << wert << '\n';
}
```

Für `string`-Objekte und andere Container können dieselben Konstruktionen verwendet werden. Hier werden zum Beispiel alle Leerzeichen des Strings in Unterstriche umgewandelt:

```
for (char& zeichen : einString) {               // Referenz zum Ändern
   if (zeichen == ' ') {
      zeichen = '_';
   }
}
```

1.9.7 Typermittlung mit auto

Das Schlüsselwort `auto` sagt dem Compiler, dass er selbst den Typ bei der Initialisierung ermitteln soll. `auto` ist von Seite 44 bekannt. Neu sind hier die Verbindung mit `const` und Referenzen, die nicht als Eigenschaft automatisch übernommen werden. Beispiele: