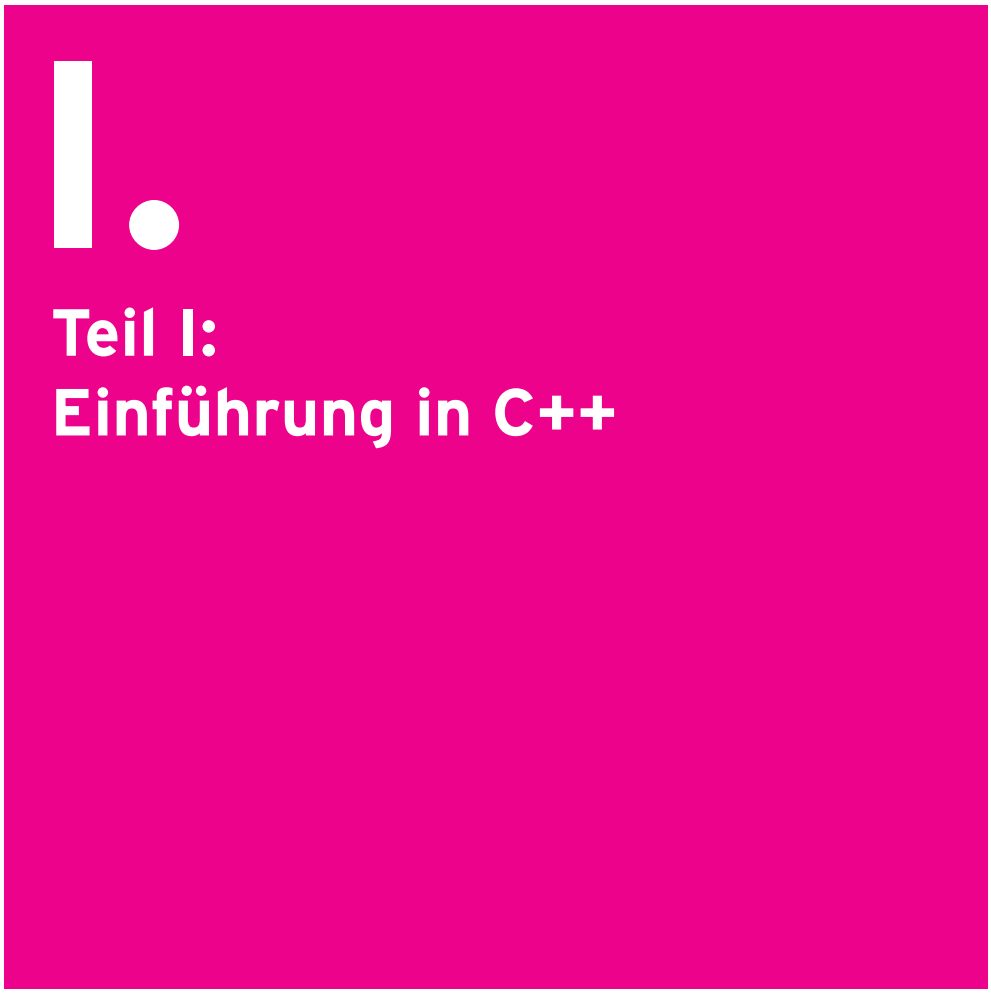


Teil I:
Einführung in C++

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

1.1 Historisches

C++ wurde etwa ab 1980 von Bjarne Stroustrup als Sprache, die Objektorientierung stark unterstützt, entwickelt und 1998 von der ISO (International Standards Organisation) und der IEC (International Electrotechnical Commission) standardisiert. Diesem Standard haben sich nationale Standardisierungsgremien wie ANSI (USA) und DIN (Deutschland) angeschlossen. Mittlerweile ist der C++-Standard, zu dem es 2003 einige Korrekturen gab, in die Jahre gekommen. Die Anforderungen an C++ sind gewachsen, auch zeigte sich, dass manches im Standard fehlte und anderes überflüssig oder fehlerhaft war. Das C++-Standardkomitee hat kontinuierlich an der Entwicklung des nächsten C++-Standards gearbeitet. Der Arbeitsname war C++0x, weil es anfangs die feste Absicht des Standardkomitees war, den neuen C++-Standard spätestens 2009 zu verabschieden. Der Standardisierungsprozess dauerte dann doch zwei Jahre länger als geplant.

2006 wurde beschlossen, den TR1 (Technical Report 1 [TR1]) mit der Ausnahme spezieller mathematischer Funktionen in den Standard aufzunehmen. Der TR1 besteht aus teilweise erheblichen Erweiterungen der C++-Standardbibliothek. Der TR1 wurde von Boost [boost] entwickelt, einer Community, die von Mitgliedern der »C++ Standards Committee Library Working Group« gegründet wurde und an der sich heute Tausende von Entwicklern beteiligen. Bereits vor der Integration in den C++-Standard haben die Boost-Bibliotheken vielfachen Einsatz gefunden. Weitere Erweiterungen sind in Arbeit. Im März 2010 wurde der »Final Committe Draft« veröffentlicht und zur Abstimmung gestellt. Im April 2011 wurde der »Final Draft International Standard (FDIS)« [ISOC++] publiziert, der Ende 2011 in den endgültigen Standard mündet. ISO-Standards sind kostenpflichtig. Wegen des jahrelangen Abstimmungsprozesses wird sich der endgültigen Standard nur minimal vom FDIS unterscheiden. Deshalb verweise ich in diesem Buch auf das per Internet zugängliche Dokument [ISOC++].

1.2 Objektorientierte Programmierung

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

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

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

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

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

Das Motiv hinter der objektorientierten Programmierung ist die rationelle und vor allem ingenieurmäßige Softwareentwicklung. Unter »Softwarekrise« wird das Phänomen verstanden, dass viele Softwareprojekte nicht im geplanten Zeitraum fertig werden, dass das finanzielle Budget überschritten wird oder die Projekte ganz abgebrochen werden. Die objektorientierte Programmierung wird als ein Mittel angesehen, das zur Lösung der »Softwarekrise« beitragen kann.

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

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

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

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

Klassen

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

Attribute:

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

Operationen:

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

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

Tabelle 1.1: Attribute und Werte zweier Konten

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

Julia will Johnny 1000 € überweisen. Dem Objekt k1 wird also der Auftrag mit den benötigten Daten mitgeteilt:

k1.überweisen(54688490, 1000.00).

Johnny will 22 € abheben. Die Aufforderung wird an k2 gesendet:

k2.abheben(22).

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

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

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

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

Die hier verwendete Definition einer Klasse als Beschreibung der Eigenschaften einer Menge von Objekten wird im Folgenden beibehalten. Gelegentlich findet man in der Literatur andere Definitionen, auf die hier nicht weiter eingegangen wird. Weitere Informationen zur Objektorientierung sind in Kapitel 4 und in der Literatur zu finden, zum Beispiel in [Bal].

1.3 Compiler

Compiler sind die Programme, die Ihren Programmtext in eine für den Computer verarbeitbare Form übersetzen. Von Menschen geschriebener und für Menschen lesbarer Programmtext kann nicht vom Computer »verstanden« werden. Das vom Compiler erzeugte Ergebnis der Übersetzung kann der Computer aber ausführen. Das Erlernen einer Programmiersprache ohne eigenes praktisches Ausprobieren ist kaum sinnvoll. Deshalb sollten Sie die Dienste des Compilers möglichst bald anhand der Beispiele nutzen – wie, zeigt Ihnen der Abschnitt direkt nach der Vorstellung des ersten Programms. Falls Sie nicht schon einen C++-Compiler oder ein C++-Entwicklungssystem haben, um die Beispiele korrekt zu übersetzen, bietet sich die Benutzung der in Abschnitt 1.5 beschriebenen Entwicklungsumgebungen an. Ein viel verwendeter Compiler ist der GNU[1] C++-Compiler [GCC]. Entwicklungsumgebung und Compiler sind kostenlos erhältlich. Ein Installationsprogramm dafür finden Sie auf der DVD zum Buch.

1.4 Das erste Programm

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

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

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

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

[1] Siehe Glossar Seite 952

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

Kommentare werden einschließlich der Kennungen vom Compiler vollständig ignoriert. Ein Kommentar, der mit `/*` beginnt, ist erst mit `*/` beendet, auch wenn er sich über mehrere Zeilen erstreckt. Ein mit `//` beginnender Kommentar endet am Ende der Zeile. Auch wenn Kommentare vom Compiler ignoriert werden, sind sie doch sinnvoll für den menschlichen Leser eines Programms, um ihm die Anweisungen zu erläutern, zum Beispiel für den Programmierer, der Ihr Nachfolger wird, weil Sie befördert worden sind oder die Firma verlassen haben.

Kommentare sind auch wichtig für den Autor eines Programms, der nach einem halben Jahr nicht mehr weiß, warum er gerade diese oder jene komplizierte Anweisung geschrieben hat. Sie sehen:

Ein Programm ist »nur« ein Text!

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

Weil die Zeilenstruktur für den Rechner keine Rolle spielt, kann der Programmtext nach Gesichtspunkten der Lesbarkeit gestaltet werden. Im dritten Schritt müssen »nur« noch die Inhalte der Kommentare als C++-Anweisungen formuliert werden. Dabei bleiben die Kommentare zur Dokumentation stehen, wie im Beispielprogramm auf der nächsten Seite zu sehen ist. Es sind einige neue Worte dazugekommen, die hier kurz erklärt werden. Machen Sie sich keine Sorgen, wenn Sie nicht alles auf Anhieb verstehen! Alles wird im Verlauf des Buchs wieder aufgegriffen und vertieft. Wie das Programm zum Laufen gebracht wird, erfahren Sie nur wenige Seiten später (Seite 36).

`#include<iostream>` Einbindung der Ein-/Ausgabefunktionen. Diese Zeile muss in jedem Programm stehen, das Eingaben von der Tastatur erwartet oder Ausgaben auf den Bildschirm bringt. Man kann sich vorstellen, dass der Compiler beim Übersetzen des Programms an dieser Stelle erst alle zur Ein- und Ausgabe notwendigen Informationen liest. Details folgen in Abschnitt 3.3.5.

`using namespace std;` Der Namensraum `std` wird benutzt. Schreiben Sie es einfach in jedes Programm an diese Stelle und haben Sie Geduld: Eine genauere Erklärung folgt später (Seiten 60 und 141).

Listing 1.1: Summe zweier Zahlen berechnen

```
// cppbuch/k1/summe.cpp
// Hinweis: Alle Programmbeispiele sind von der Internet-Seite zum Buch herunterladbar
// (http://www.cppbuch.de/).
// Die erste Zeile in den Programmbeispielen gibt den zugehörigen Dateinamen an.
#include<iostream>
using namespace std;

// Programm zur Berechnung der Summe zweier Zahlen
int main() {
    int summe;
    int a;
    int b;

    // Lies die Zahlen a und b ein
    cout << "a und b eingeben:";
    cin >> a >> b;

    /* Berechne die Summe beider Zahlen */
    summe = a + b;

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

`int main()`

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

`int summe;`
`int a;`
`int b;`

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

`;`

Ein Semikolon beendet jede Deklaration und jede Anweisung (aber keine Verbundanweisung, siehe später).

`cout`

Ausgabe: `cout` (Abkürzung für *character out*) ist die Standardausgabe. Der Doppelpfeil deutet an, dass alles, was rechts davon steht, zur Ausgabe `cout` gesendet wird, zum Beispiel `cout << a;`. Wenn mehrere Dinge ausgegeben werden sollen, sind sie durch `<<` zu trennen.

`cin`	Eingabe: Der Doppelpfeil zeigt hier in Richtung des Objekts, das ja von der Tastatur einen neuen Wert aufnehmen soll. Die Information fließt von der Eingabe `cin` zum Objekt `a` beziehungsweise zum Objekt `b`.
`=`	Zuweisung: Der Variablen auf der linken Seite des Gleichheitszeichens wird das Ergebnis des Ausdrucks auf der rechten Seite zugewiesen.
`"Text"`	beliebige Zeichenkette, die die Anführungszeichen selbst nicht enthalten darf, weil sie als Anfangs- beziehungsweise Endemarkierung einer Zeichenfolge dienen. Wenn die Zeichenfolge die Anführungszeichen enthalten soll, sind diese als \" zu schreiben: `cout << "\"C++\" ist der Nachfolger von \"C\"!";` erzeugt die Bildschirmausgabe *"C++" ist der Nachfolger von "C"!*.
`return 0;`	Unser Programm läuft einwandfrei; es gibt daher 0 zurück. Diese Anweisung kann fehlen, dann wird automatisch 0 zurückgegeben.

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

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

- Objekte müssen deklariert werden. `int summe;` ist eine Deklaration, wobei `int` der *Datentyp* des Objekts `summe` ist, der die Eigenschaften beschreibt. Entsprechendes gilt für `a` und `b`. Die Objektnamen sind frei wählbar im Rahmen der unten angegebenen Konventionen. Unter *Deklaration* wird verstanden, dass der Name dem Compiler bekannt gemacht wird. Wenn dieser Name später im Programm versehentlich falsch geschrieben wird, z. B. `sume = a + b;` im Programm auf Seite 33, kennt der Compiler den falschen Namen `sume` nicht und gibt eine Fehlermeldung aus. Damit dienen Deklarationen der Programmsicherheit.
- Objektnamen bezeichnen Bereiche im Speicher des Computers, deren Inhalte verändert werden können. Die Namen sind symbolische Adressen, unter denen der Wert gefunden wird. Über den Namen kann dann auf den aktuellen Wert zugegriffen werden (siehe Abbildung 1.1).

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

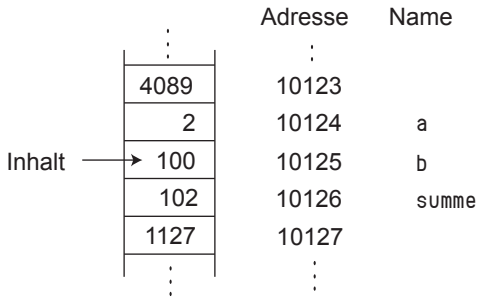


Abbildung 1.1: Speicherbereiche mit Adressen

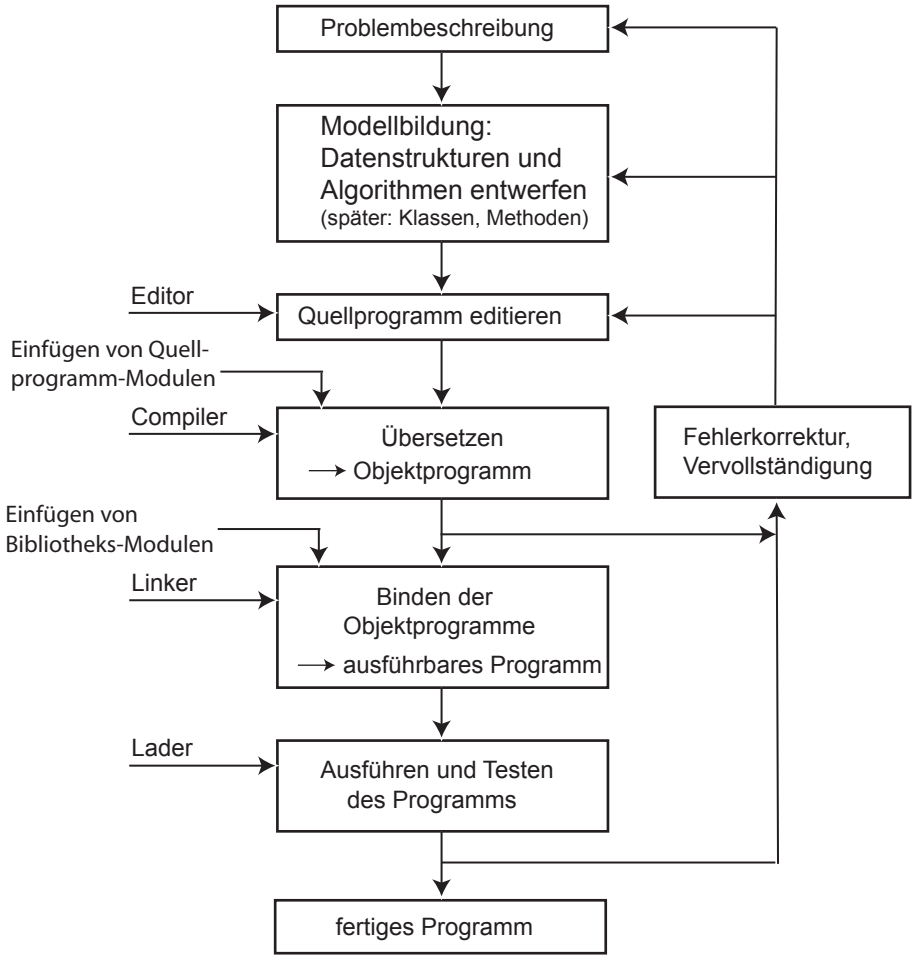


Abbildung 1.2: Erzeugung eines lauffähigen Programms

Der Compiler ist selbst ein Programm, das bereits in maschinenverständlicher Form vorliegt und speziell für diese Übersetzung zuständig ist. Nach Eingabe des Programmtextes mit dem Editor können Sie den Compiler starten und anschließend das Programm binden oder linken (eine Erklärung folgt bald) und ausführen. Ein Programmtext wird auch »Quelltext« (englisch *source code*) genannt.

Der Compiler erzeugt den Objektcode, der noch nicht ausführbar ist. Hinter den einfachen Anweisungen `cin >> ...` und `cout << ...` verbergen sich eine Reihe von Aktivitäten wie die Abfrage der Tastatur und die Ansteuerung des Bildschirms, die nicht speziell programmiert werden müssen, weil sie schon in vorübersetzter Form in Bibliotheksdateien vorliegen. Die Aufrufe dieser Aktivitäten im Programm müssen mit den dafür vorgesehenen Algorithmen in den Bibliotheksdateien zusammengebunden werden, eine Aufgabe, die der *Linker* übernimmt, auch *Binder* genannt. Der Linker bindet Ihren Objektcode mit dem Objektcode der Bibliotheksdateien zusammen und erzeugt daraus ein ausführbares Programm, das nun gestartet werden kann. Der Aufruf des Programms bewirkt, dass der *Lader* eine Funktion des Betriebssystems, das Programm in den Rechnerspeicher lädt und startet. Diese Schritte werden stets ausgeführt, auch wenn sie in den Programmentwicklungsumgebungen verborgen ablaufen. Bibliotheksmodule können auch während der Programmausführung geladen werden (nicht im Bild dargestellt). Weitere Details werden in Abschnitt 3.3 erläutert.

Wie bekomme ich ein Programm zum Laufen?

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

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

Das Programm wird durch Eintippen von *summe.exe* gestartet. Dabei wird vorausgesetzt, dass der g++-Compiler im Pfad ist – sonst wird er nicht gefunden. Wie man den Pfad um ein Verzeichnis erweitert, lesen Sie im Anhang, Abschnitt A.7. Weitere Einzelheiten zur Bedienung von Compiler und Linker finden Sie in Abschnitt A.5 auf Seite 891 oder in den Hilfedateien Ihres C++-Systems. Der folgenden Abschnitt 1.5 zeigt Installation und Bedienung einer kostenlosen[2] Integrierten Entwicklungsumgebung.

1.4.1 Namenskonventionen

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

[2] Bitte Lizenzbestimmungen beachten!

- Ein Name ist eine Folge von Zeichen, bestehend aus Buchstaben, Ziffern und Unterstrich (_).
- Ein Name beginnt stets mit einem Buchstaben oder einem Unterstrich. Am Anfang eines Namens sollten Unterstriche jedoch 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 887 zu finden.
- Ein Name kann prinzipiell beliebig lang sein. In den Compilern ist die Länge jedoch begrenzt, zum Beispiel auf 31 oder 255 Zeichen.

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

`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;`.

1.5 Integrierte Entwicklungsumgebungen

Im Folgenden stelle ich Ihnen kurz zwei ausgewählte Integrierte Entwicklungsumgebungen, abgekürzt IDE, vor: Code::Blocks und Eclipse. Beide gibt es für Windows und Linux.

Tipp

Anfängern empfehle ich die IDE Code::Blocks, sei es unter Linux oder Windows, weil sie am einfachsten zu installieren und zu bedienen ist.

1.5.1 Code::Blocks

Code::Blocks [CB] ist eine Open Source-IDE für C++ für Windows und Linux, die einfach installier- und bedienbar und für unsere Zwecke sehr gut geeignet ist. Weil in diesem Buch nur sehr kurz auf die Bedienung eingegangen werden kann, empfehle ich Ihnen die Bedienungsanleitung von Code::Blocks, die Sie auf der DVD zu diesem Buch oder der Internetseite [CB] finden.

Code::Blocks auf DVD

Code::Blocks ist auf der DVD vorhanden. Die Installationsanleitung für die Software (die auch Code::Blocks enthält) für Windows finden Sie in Abschnitt A.7 (Seite 937), die für Linux in Abschnitt A.8.3 (Seite 944).

Programm eingeben, übersetzen und starten

Um jetzt ein Programm einzugeben, starten Sie Code::Blocks (Windows: Start →Alle Programme →CodeBlocks). Nach dem Start legen Sie ein neues Projekt an, indem Sie auf »Create a new Project« im Startfenster klicken. Alternativ ist der Weg über den Menübalken möglich (File →New →Project). Es wird Ihnen eine Auswahl verschiedenster Anwendungstypen angeboten. Fürs Erste wählen Sie bitte die einfachste Art der Anwendung, die »Console Application«. Im dann erscheinenden Fenster müssen noch einige Angaben eingetragen werden. Vorschlag:

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

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

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

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

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

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

hinzugefügt, wie im Beispiel auf der DVD gezeigt (Datei *cppbuch/k1/summe.cpp*).

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

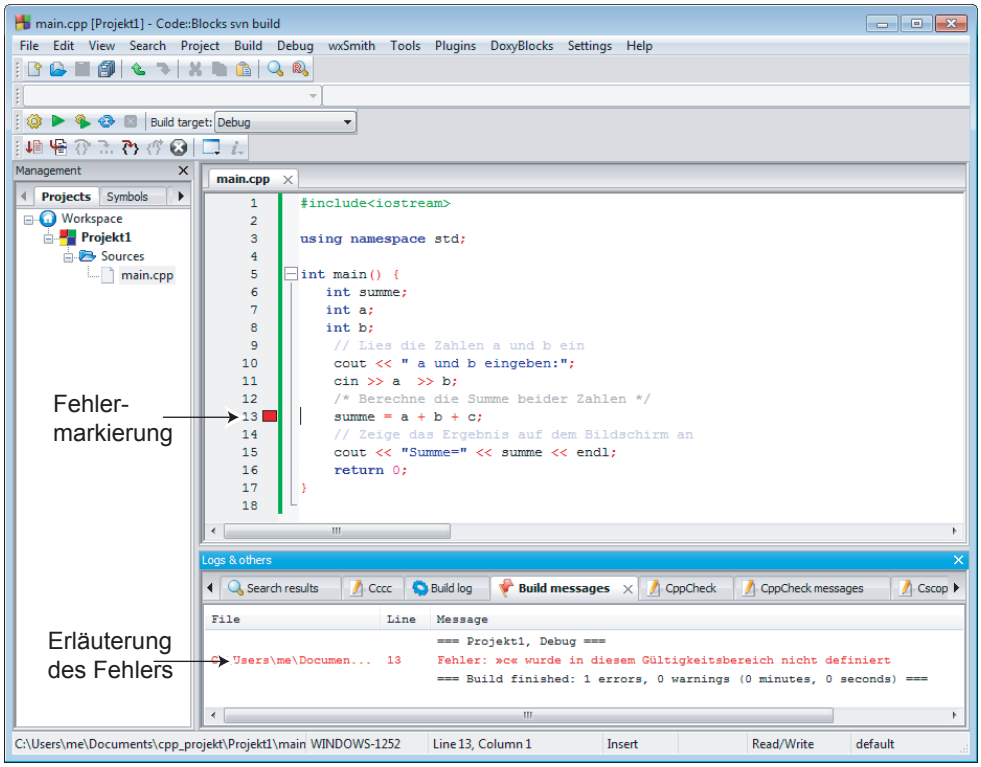


Abbildung 1.3: Die Entwicklungsumgebung Code::Blocks zeigt einen Fehler an.

Tipp

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

1.5.2 Eclipse

Eclipse (*http://www.eclipse.org*) ist eine mächtige Entwicklungsumgebung, die es für viele Betriebssysteme gibt, darunter Linux, Mac OS X und Windows. Sie setzt eine Java-Installation voraus. Für erfahrene Softwareentwickler, die von Java auf C++ umsteigen, ist Eclipse sehr gut geeignet. Eclipse gibt es fertig für C/C++ konfiguriert zum Herunterladen. Ein Hinweis: Bitte loggen Sie sich für alle Installationsvorgänge als »Administrator« (Windows) bzw. »root« (Linux) ein, wenn die Installation für alle Benutzer des Rechners gelten soll. Eclipse kann aber auch lokal installiert werden. Der C++-Compiler ist nicht

Bestandteil von Eclipse. Er muss daher vorher installiert worden sein und sich im Pfad befinden. Dann findet Eclipse ihn automatisch. In Abbildung 1.4 wird der gefundene Compiler rechts angezeigt.

Anlegen eines neuen C++-Projekts

Die Folge File → New → C++ Project ergibt das in Abbildung 1.4 gezeigte Fenster.

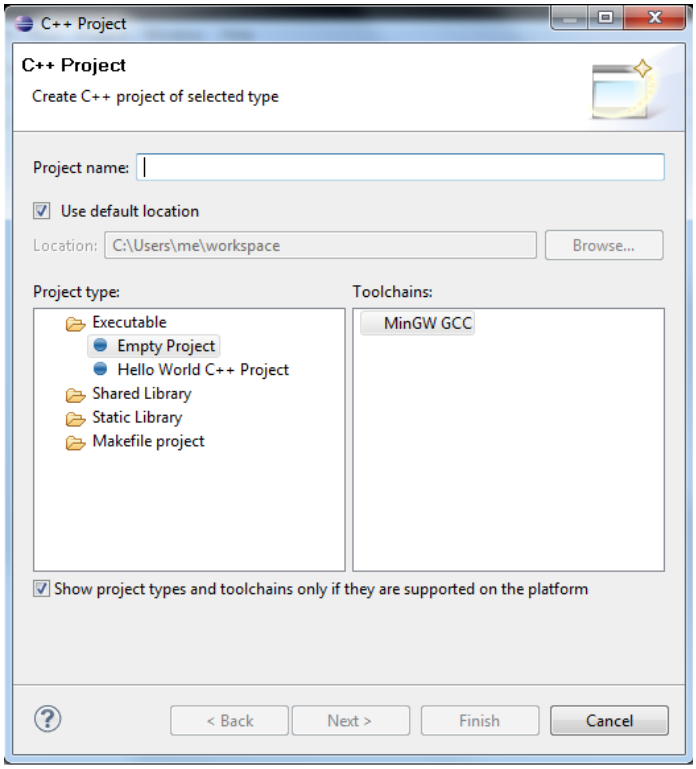


Abbildung 1.4: Neues Projekt anlegen

Dort tragen Sie oben den Projektnamen ein, zum Beispiel »Mein erstes C++-Projekt« und klicken auf Finish. Nun wird als Beispiel das einfache Programm zur Berechnung einer Summe erzeugt. Dazu klicken Sie mit der *rechten* Maustaste auf den Projektnamen links im Fenster und wählen New → Source File. Als Name können Sie zum Beispiel *summe.cpp* eingeben. Mit Finish gelangen Sie in das Editorfenster, in das Sie das Programm nun eingeben können. Wenn Sie die rot markierten Worte in Kommentaren stören: Entweder installieren Sie das deutsche Wörterbuch (im Internet nach BabelLanguagePack-eclipse suchen), oder Sie schalten die Rechtschreibprüfung ab (unter Window → Preferences → General – Editors – Text Editors – Spelling).

Der Compiliervorgang wird durch Anklicken des Hammer-Symbols oben gestartet. Zum Vergleich wurde derselbe Fehler wie oben eingebaut, siehe Abbildung 1.5:

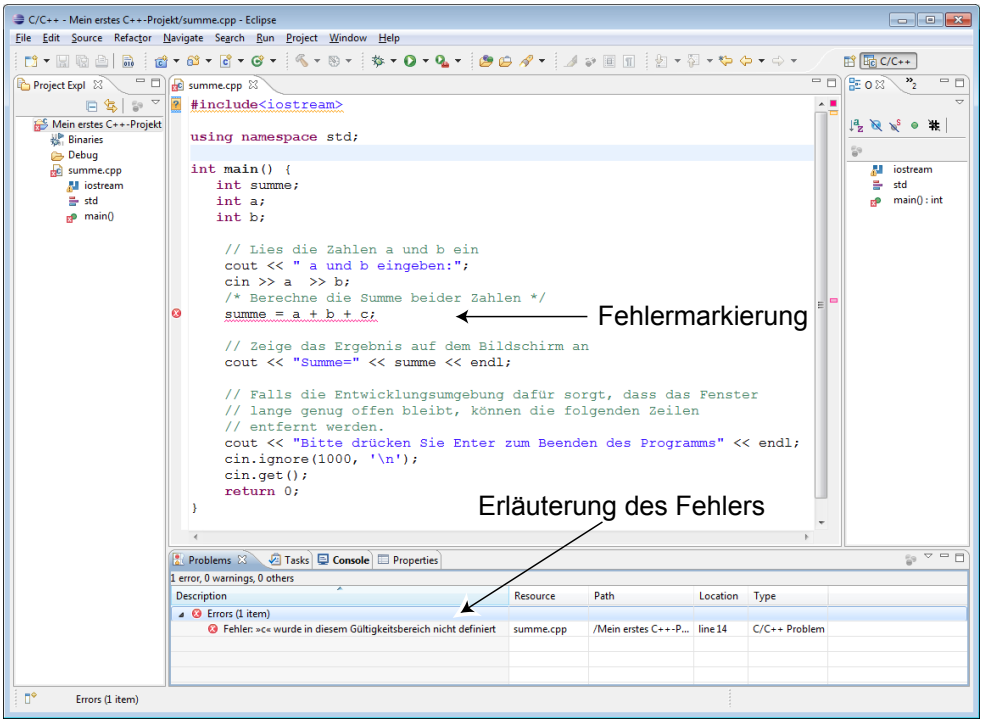


Abbildung 1.5: Die Eclipse-IDE zeigt einen Fehler an.

Nach Korrektur wird das Programm gesichert und noch einmal compiliert. Durch Klicken auf das Run-Symbol oben (helles Dreick im grünen Kreis) bringen Sie das Programm zum Laufen. Im Konsolen-Fenster von Eclipse (unten im Eclipse-Fenster) können Sie nun zwei Zahlen eingeben. Die berechnete Summe wird dort angezeigt.

1.6 Einfache Datentypen und Operatoren

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

1.6.1 Ausdruck

Ein Ausdruck besteht aus einem oder mehreren Operanden, die miteinander durch Operatoren verknüpft sind. Die Auswertung eines Ausdrucks resultiert in einem Wert, der an die Stelle des Ausdrucks tritt. Der einfachste Ausdruck besteht aus einer einzigen Konstante, Variable oder einem Literal. Die Operatoren müssen zu den Operanden passen, die zu bestimmte 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 hier 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 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 soviel wie `long`

Die eingeklammerten Bitzahlen sind entsprechend dem C++-Standard mindestens erforderlich. Ein Bit wird für das Vorzeichen reserviert. 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 üblicherweise im Zweierkomplement dargestellt. Das Zweierkomplement wird gebildet, indem alle Bits invertiert werden und dann auf das Ergebnis 1 addiert wird. Durch das Schlüsselwort `unsigned` werden Zahlen *ohne* Vorzeichen definiert, zum Beispiel `unsigned int`, `unsigned long` (Langform: `unsigned long int`). Durch das damit gewonnene zusätzliche Bit teilt sich der Zahlenbereich anders auf:

`signed`	16 Bits	$-2^{15} \ldots 2^{15}-1$	=	$-32\,768 \ldots 32\,767$
`unsigned`	16 Bits	$0 \ldots 2^{16}-1$	=	$0 \ldots 65\,535$
`signed`	32 Bits	$-2^{31} \ldots 2^{31}-1$	=	$-2\,147\,483\,648 \ldots 2\,147\,483\,647$
`unsigned`	32 Bits	$0 \ldots 2^{32}-1$	=	$0 \ldots 4\,294\,967\,295$
`signed`	64 Bits	$-2^{63} \ldots 2^{63}-1$	=	$-9\,223\,372\,036\,854\,775\,808$
				$\ldots 9\,223\,372\,036\,854\,775\,807$
`unsigned`	64 Bits	$0 \ldots 2^{64}-1$	=	$0 \ldots 18\,446\,744\,073\,709\,551\,615$

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

Listing 1.2: Grenzwerte und Speicherplatzbedarf

```
// cppbuch/k1/limits.cpp
#include<iostream>
#include<climits>          // hier sind die Bereichsinformationen
using namespace std;
int main() {
   cout << "Grenzwerte für Ganzzahl-Typen:" << endl;  // = neue Zeile (endline)
   cout << "INT_MIN =" << INT_MIN << endl;
   cout << "INT_MAX =" << INT_MAX << endl;
   cout << "LONG_MIN =" << LONG_MIN << endl;
   cout << "LONG_MAX =" << LONG_MAX << endl;
   cout << "LLONG_MIN =" << LLONG_MIN << endl;
   cout << "LLONG_MAX =" << LLONG_MAX << endl;
   cout << "unsigned-Grenzwerte:" << endl;
   cout << "UINT_MAX =" << UINT_MAX << endl;
   cout << "ULONG_MAX =" << ULONG_MAX << endl;
   cout << "ULLONG_MAX =" << ULLONG_MAX << endl;
   cout << "Anzahl der Bytes für:" << endl;
   cout << "int       "  << sizeof(int) << endl;
   cout << "long      "  << sizeof(long) << endl;
   cout << "long long "  << sizeof(long long) << endl;
}
```

Statt `INT_MAX` können Sie `numeric_limits<int>::max()` schreiben usw. Diese Alternative wird im Listing auf Seite 47 am Beispiel der Rechnerdarstellung für reelle Zahlen gezeigt. Beim Rechnen mit ganzen Zahlen ist der *begrenzte Wertebereich* zu beachten! Aufgrund der Tatsache, dass nur eine begrenzte Anzahl von Bits für die rechnerinterne Repräsentation einer Zahl zur Verfügung steht, ergibt sich, dass der von `int` abgedeckte Zahlenbereich nur eine *Untermenge* der ganzen Zahlen darstellt, wie das Programmbeispiel »Arithmetischer Überlauf« zeigt.

Listing 1.3: Arithmetischer Überlauf

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

int main() {
   int ai = 50000;
   int bi = 1000000;
   int ci = ai * bi;
   cout << "int-Zahlen haben auf Ihrem System "
        << 8*sizeof(int) << " Bits" << endl;
   cout << "Rechnung mit int: ";
   cout << ai << " * " << bi << " = " << ci << endl;
   // Ausgabe -1539607552 statt 50000000000 bei 32 Bit-int
   long al = 50000; long bl = 1000000; long cl = al*bl;
   cout << "long-Zahlen haben auf Ihrem System "
```

```
        << 8*sizeof(long) << " Bits" << endl;
    cout << "Rechnung mit long: "
        << a1 << " * " << b1 << " = " << c1 << endl;

    // Falls Ihr Compiler long long unterstützt:
    long long all = 50000, long long bll = 1000000;
    long long cll = all * bll;
    cout << "long long-Zahlen haben auf Ihrem System "
        << 8*sizeof(long long) << " Bits" << endl;
    cout << "Rechnung mit long long: ";
    cout << all << " * " << bll << " = " << cll << endl;
}
```

Daraus 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 durch den 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 muss man 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 weil Rechenoperationen mit ihnen länger dauern. Ganze Zahlen können auf dreierlei Arten dargestellt werden:

1. 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).
2. Wenn eine Zahl mit 0x oder 0X beginnt, wird sie als *Hexadezimalzahl* interpretiert, zum Beispiel 0xAFFE = 45054 dezimal.
3. Dezimal wie üblich. Ein Suffix (l, ll, L, LL) kennzeichnet `long`- oder `long long`-Zahlen, zum Beispiel 2147483647L oder 9223372036854775806LL. Ein Suffix u oder U kennzeichnet `unsigned`-Zahlen.

Operatoren für ganze Zahlen

Die Tabelle 1.2 zeigt die für ganze Zahlen möglichen Operatoren, also für die Datentypen `short`, `int` und `long`. Die zusammengesetzten Operatoren wie `+=` heißen Kurzform-Operatoren. Auf Daten eines bestimmten Typs kann man nur bestimmte Operationen durchführen. Eine Zeichenkette kann man zum Beispiel nicht durch eine andere dividieren. Man kann jedoch ganze Zahlen a und b addieren. Daraus folgt: Ein Datum und die zugehörigen Operationen gehören zusammen! Einige der Operatoren der Tabelle 1.2 werden durch Beispiele erläutert. Manche Operatoren setzen jedoch hier noch nicht besprochene Dinge voraus, weshalb gelegentlich auf spätere Abschnitte verwiesen wird. Beispiele für Operatoren:

Tabelle 1.2: Operatoren für Ganzzahlen

Operator	Beispiel	Bedeutung
Arithmetische Operatoren:		
`+`	`+i`	unäres Plus (kann weggelassen werden)
`-`	`-i`	unäres Minus
`++`	`++i`	vorherige Inkrementierung um eins
	`i++`	nachfolgende Inkrementierung um eins
`--`	`--i`	vorherige Dekrementierung um eins
	`i--`	nachfolgende 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
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`

```
int a;
int i = 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 = 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, n = 5;
int ergebnis = m / n; // 1
int rest = m % n;     // 4
```

Bit-Operatoren

Weil ganze Zahlen auch 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 und Zuweisung des Ergebnisses an c; das entspricht der Multiplikation mit 2^2, also 4.

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

Würde man auf c noch 1 addieren, erhielte man das Zweierkomplement, also die Darstellung der negativen Zahl -5.

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.

1.6.3 Reelle Zahlen

Reelle Zahlen, auch Gleitkommazahlen genannt, sind wegen der beschränkten Bit-Anzahl in der Regel nicht beliebig genau darstellbar. Sie werden in C++ wie folgt geschrieben:

Vorzeichen (optional), Vorkommastellen, Dezimalpunkt, Nachkommastellen, `e` oder `E` und Ganzzahl-Exponent (optional), Suffix `f`, `F` oder `l`, `L` (optional, 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 1.8×10^6 oder 1800000. Reelle Zahlen werden durch die drei Datentypen der Tabelle 1.3 (ungenau) dargestellt.

Tabelle 1.3: Datentypen und Bereiche für reelle Zahlen

Typ	Bits	Zahlenbereich		Genauigkeit in Dezimalstellen
`float`	32	$\pm$ 1.1*10^{-38}	... $\pm$ 3.4*10^{38}	ca. 7
`double`	64	$\pm$ 2.2*10^{-308}	... $\pm$1.8*10^{308}	ca. 16
`long double`	96	$\pm$ 3.4*10^{-4932}	... $\pm$1.2*10^{4932}	ca. 24

Anstelle des Kommas tritt ein Dezimalpunkt, wie im angelsächsischen Sprachraum üblich. Die Festlegungen Ihres C++-Systems sind im Header `<limits>` oder in der C-Datei *float.h* zu finden. Anders als im Programm auf Seite 43 können die `numeric_limits<float>`- bzw. `numeric_limits<double>`-Funktionen Auskunft geben, die es auch für `int` usw. gibt.

Listing 1.4: Grenzwerte von float-Zahlen

```
// cppbuch/k1/floatlimits.cpp
#include<iostream>
#include<limits>          // hier sind die Bereichsinformationen
using namespace std;

int main() {
  cout << "Grenzwerte für Float-Zahl-Typen:" << endl;
  cout << "Float-Min: "
       << numeric_limits<float>::min() << endl;
  cout << "Float-Max: "
       << numeric_limits<float>::max() << endl;

  cout << "Double-Min: "
       << numeric_limits<double>::min() << endl;
  cout << "Double-Max: "
```

```
        << numeric_limits<double>::max() << endl;

    cout << "Long-Double-Min: "
         << numeric_limits<long double>::min() << endl;
    cout << "Long-Double-Max: "
         << numeric_limits<long double>::max() << endl;
}
```

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 *nur als Beispiel*. Die Tabelle 1.4 zeigt ein Beispiel der Repräsentation reeller Zahlen im Rechner.

Tabelle 1.4: Anzahl der Bits (Beispiel)

	float	double	long double
Vorzeichen	1	1	1
Mantisse	23	52	80
Exp-Vorzeichen	1	1	1
Exponent	7	10	14
Gesamtanzahl Bits	32	64	96

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 darf nicht schlechter sein als die von `double`-Zahlen. Die Genauigkeit hängt von der Anzahl der Bits ab, die für die Mantisse verwendet werden. Der Zahlenbereich wird wesentlich durch die Anzahl der Bits für den Exponenten bestimmt, der Einfluss der Mantisse ist minimal. Es sei angenommen, dass für den Typ `double` die oben angegebenen Bitzahlen gelten. Da 2^{52} ca. $4.5 * 10^{15}$ ist, ergibt sich eine etwa 15-stellige Genauigkeit. Der Zahlenbereich leitet sich aus der Bitanzahl für den Exponenten ab: 2^{10} ist 1024. Der Exponent kann damit im Bereich von 0 bis $2^{1024} - 1$ liegen. Übertragen auf das Dezimalsystem ergibt sich mithilfe der Schulmathematik ein maximaler Exponent von 1024 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 1.5: Genauigkeit von float-Zahlen

```
// cppbuch/k1/genau.cpp
#include<iostream>
using namespace std;
int main() {
    float a = 1.234567E-7;
    float b = 1.000000;
```

```
    float c = -b;
    float s1 = a + b;
    s1 += c;                // entspricht s1 = s1 + c;
    float s2 = a;
    s2 += b + c;
    cout << "Ungenauigkeit bei float-Arithmetik:" << endl;
    cout << "(a+b)+c= " << s1 << '\n';    // 1.19209e-7
    cout << "a+(b+c)= " << s2 << '\n';    // 1.23457e-7
}
```

Folgen der nicht exakten Darstellung können sein:

- 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 betragsmäßig zu kleine Werte ergibt einen *Überlauf* (englisch *overflow*).
- Eine *Unterschreitung* (englisch *underflow*) tritt auf, wenn der Betrag des Ergebnisses zu klein ist, um mit dem gegebenen Datentyp darstellbar zu sein. 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: Um `s1` zu berechnen, werden erst `a` und `b` addiert und danach `c`, während zur Berechnung von `s2` zu `a` die Summe von `b` und `c` addiert wird. Rein mathematisch betrachtet müsste das gleiche Ergebnis herauskommen, die Computerarithmetik liefert jedoch abweichende Ergebnisse. Daher gehört zu kritischen Rechnungen immer eine Genauigkeitsbetrachtung.

Die Ausgabe von '\n' im Beispielprogramm bedeutet, dass nach den Zahlen `s1` und `s2` jeweils eine neue Zeile auf dem Bildschirm begonnen wird (siehe Abschnitt 1.6.5).

Es gibt verschiedene Methoden, den Fehler bei ungenauen Rechnungen zu minimieren. Zum Beispiel könnte man bei der Addition einer großen Menge verschiedener Zahlen so vorgehen, dass zunächst die Zahlen sortiert werden und erst danach die Addition vorgenommen wird, 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 1.2 zur Verfügung. Tabelle 1.5 auf der nächsten Seite zeigt einige Beispiele zur Umsetzung mathematischer Ausdrücke in die Programmiersprache C++. Einige mathematische Funktionen wie `sqrt()`, `exp()` u.a. sind vordefiniert. Die Deklaration der meisten dieser Funktionen befindet sich im Header `<cmath>`.

Aus historischen Gründen befindet sich die Funktion `abs()` für `int`-Zahlen jedoch im Header `<cstdlib>`. Zum Nachschlagen der verschiedenen Möglichkeiten bieten sich die Tabellen 35.3 und 35.5 an (ab Seite 876). Um dem Compiler die Deklarationen bekannt zu machen, genügt es, im Programm am Anfang der Programmdatei, die Zeile `#include<cmath>` einzufügen. Die Ausgabe von `endl` bewirkt genau wie '\n' den Sprung in eine neue Zeile:

Tabelle 1.5: Umsetzen mathematischer Ausdrücke

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

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

Listing 1.6: Mathematische Funktionen

```
// cppbuch/k1/mathexpr.cpp : Berechnung mathematischer Ausdrücke
#include<iostream>
#include<cmath>
using namespace std;

int main() {
   float x;
   cout << "x eingeben:";
   cin >> x;
   cout << " x      = " << x      << endl;
   cout << " fabs(x) = " << fabs(x) << endl;
   cout << " sqrt(x) = " << sqrt(x) << endl;
   cout << " sin(x)  = " << sin(x) << endl;  // Argument von sin() im Bogenmaß!
   cout << " exp(x)  = " << exp(x) << endl;
   cout << " log(x)  = " << log(x) << endl;  // log() ist der natürliche Logarithmus!
}
```

Übung

1.1 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 Anteil von unter 10% bei 86 Aufgaben insgesamt.)

1.6.4 Konstante

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 ist

```
const float PI = 3.1415926;  // noch besser: vordefinierte Konstante M_PI nehmen
umfang = PI * durchmesser;
```

weil bei anschließendem häufigerem Gebrauch der Zahl `PI` Schreibfehler leicht ausgeschlossen werden können, und Änderungen oder Korrekturen einer Konstante nur an einer Stelle vorgenommen werden müssen. `PI` ist nur einmal als Konstante zu vereinbaren (= zu deklarieren). `float` bedeutet, dass die Konstante eine Gleitkommazahl ist. Eine ganzzahlige Konstante würde zum Beispiel mit `const int GROESSE = 1000;`. vereinbart.

- Eine Konstante besteht aus einem Namen und dem zugeordneten Wert, der *nicht veränderbar* ist. Der Name wird üblicherweise großgeschrieben.
- Konstanten müssen wie Variablen deklariert werden. Das Schlüsselwort `const` leitet die Deklaration ein, Arithmetik ist erlaubt, z.B. `const int MAX_INDEX = GROESSE-1;`

Eine Zahl auf der rechten Seite zur Initialisierung einer Konstanten heißt »Zahlliteral«. Ein Literal für Zahlen muss den dafür erlaubten Regeln entsprechen. Ein Literal steht im Programmtext und ist daher nicht veränderbar. Die Bedeutung besteht nur in seinem Wert. Ein Literal für ganze Zahlen besteht nur aus einem optionalen Vorzeichen, gefolgt von Ziffern und möglicherweise einem Suffix, um den Typ zu spezifizieren, zum Beispiel `l` oder `L` für `long`-Zahlen.

Unveränderliche Größen sollten *stets* als `const` deklariert werden! Begründung: Der Compiler wird damit in die Lage versetzt, fehlerhafte Zuweisungen zu finden:

```
const float PI = 3.1415926;
// weiterer Programmtext
PI = 17.5;  // ergibt eine Fehlermeldung des Compilers!
```

1.6.5 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 Hochkommata 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[4]. Die ASCII-Tabelle definiert die ersten 7 Bits eines Bytes, also 128 Zeichen (siehe Abschnitt A.3). Es gibt drei verschiedene `char`-Datentypen:

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

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'??'`.

Mehr über `wchar_t` und Zeichenliterale lesen Sie in Abschnitt 31.2.

[4] Dies muss nicht für jedes System gelten, ist aber verbreitet.

Alle Basisfunktionen der Standardbibliothek (Kapitel 26) 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 STERN = '*';
char a;
a = 'a';
```

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

```
a = 'x';
a = STERN;
```

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*). Außerhalb der ASCII-Tabelle gibt es ANSI-Escape-Sequenzen zur Bildschirmansteuerung, die mehrere Bytes umfassen können. Eine Hilfsdatei namens *ansi_esc.h* finden Sie im Verzeichnis *cppbuch/include* der Beispiele. Lesen Sie vor Benutzung den Anfang der Datei.

Tabelle 1.6 zeigt einige Beispiele. Das `endl` auf Seite 49 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 Kapitel 10, Stichwort `unitbuf`).

Tabelle 1.6: 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
\x◇, \X◇	◇ = Zeichenfolge aus Hex-Ziffern, Beispiel: \xDB oder \x1f	

Da ein ASCII-Zeichen genau *ein* Byte beansprucht, ist ein Zeichen der Oktaldarstellung \777 nicht erlaubt (\377 = 255_{10})[5]. Die Zuordnung von Zeichen und Zahl geschieht über die ASCII-Tabelle, über die eine Ordnungsrelation definiert ist: ...'0' < '1' < .. < '9'< .. < 'A' < .. < 'Z' < .. < 'a' < .. < 'z'... Ein Zeichen hat eine eindeutige Position innerhalb der ASCII-Tabelle (siehe Seite 887).

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.

Mehr über das Thema Zeichensatz lesen Sie in Abschnitt 31.2.

Die Position eines Zeichens in der erweiterten 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;
int i;
i = static_cast<int>(c); // Typumwandlung char → int
```

bedeutet, dass der Wert der Variablen `i` nun 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 wie eine 1-Byte-`int`-Zahl interpretiert wird, ist auch eine implizite Typumwandlung möglich:

```
i = c;
```

Die einfacheren Schreibweisen werden jedoch nicht empfohlen, weil sie in bestimmten Zusammenhängen unsicherer sind. Eine Begründung wird in Abschnitt 7.9 gegeben. Die Schreibweise `int(c)` erinnert an die später zu besprechenden Funktionen. Die aus der Sprache C übernommene traditionelle Schreibweise `(int) c` bewirkt genau dasselbe und ist im Gegensatz zur Funktionsschreibweise auch für komplexere Datentypen möglich. 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
```

[5] Es sind beliebig viele Oktal- oder Hexadezimalziffern erlaubt, wenn der Zeichentyp (z.B. `wchar_t`) sie darstellen kann.

Es gelten die Identitäten

```
c == static_cast<char>( static_cast<int>(c)); und
i == static_cast<int>( static_cast<char>(i));,
```

falls -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. Mehr zu Standard-Typumwandlungen erfahren Sie auf Seite 57. Wie kann man 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 << endl;  // 5
ziffer = static_cast<int>(c) - static_cast<int>('0'); // explizite Typumwandlung!
```

Weil ein `char` vom Compiler wie eine 1-Byte-`int`-Zahl interpretiert 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 (siehe Tabelle 1.2 auf Seite 45) möglich, aber im Sinne der Bedeutung von *Zeichen* sind nur die Operatoren aus Tabelle 1.7 sinnvoll.

Tabelle 1.7: 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

1.6.6 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. Variablen eines logischen Datentyps 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. Das folgende Programmstück gibt eine 1 aus, falls das eingelesene Zeichen ein Großbuchstabe war, ansonsten eine 0.

```
bool istGrossBuchstabe;
char c;
cin >> c;
istGrossBuchstabe = (c >= 'A') && (c <= 'Z');
```

```
cout << istGrossBuchstabe;        // Wandlung in int
```

Die Ausgabe als Text *true* bzw. *false* kann eingestellt werden:

```
cout.setf(ios_base::boolalpha); // Textformat einschalten
cout << istGrossBuchstabe;     // Wandlung in Text
```

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 dann durch das logische UND (`&&`) verbunden, nicht zu verwechseln mit dem bitweisen UND (`&`) der Tabelle 1.2 auf Seite 45. Das Ergebnis wird der Variablen `istGrossBuchstabe` zugewiesen. Die Klammern sind hier nur zur Verdeutlichung angegeben. Sie können entfallen, weil die relationalen Operatoren eine höhere Priorität als die logischen haben. Tabelle 1.8 zeigt Operatoren für logische Datentypen. Der Datentyp `bool` wird an allen Stellen, die nicht ausdrücklich `bool` verlangen, nach `int` gewandelt (siehe obiges 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 1.8: Operatoren für logische Datentypen

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

```
bool wahrheitswert = true;
wahrheitswert = !wahrheitswert;      // Negation
cout << wahrheitswert << endl;       // 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
```

1.6.7 Referenzen

Eine *Referenz* ist ein Datentyp, der einen *Verweis* auf ein Objekt liefert. Referenzen werden in C++ häufig zur Parameterübergabe benutzt; nähere Erklärungen und umfangreichere Beispiele finden sich erst einige Kapitel später (ab Kapitel 3). Eine Referenz bildet einen *Alias-Namen* für ein Objekt, über den es ansprechbar ist. Ein Objekt hat damit zwei Namen! Der Compiler »weiß« aufgrund der Deklaration, dass es sich um eine Referenz

handelt, und nicht etwa um ein neues Objekt. Um eine Variable als Referenz zu deklarieren, wird das &-Zeichen benutzt, das neben dem bitweisen UND und dem (noch nicht benutzten) Adressoperator nun die dritte Bedeutung hat. Beispiele:

```
int i = 2;
int j = 9;
int& r = i;       // Referenz auf i (r ist ein Alias für i)
r  = 10;          // ändert i
r = j;            // Wirkung: i = j;
```

Wo das &-Zeichen zwischen int und r steht, ist unerheblich. In diesem Buch wird die Schreibweise int& r bevorzugt, um zu verdeutlichen, dass die Referenzeigenschaft zum Typ gehört. Eine Referenz wird genau wie eine Variable benutzt, der Compiler weiß, dass sie ein Alias-Name ist. Referenzen müssen bei der Deklaration initialisiert werden. Es ist nicht möglich, eine Referenz nach der Initialisierung so zu ändern, dass sie ein Alias-Name für eine andere Variable als die erstzugewiesene wird. Die Deklaration int& s = r; könnte vordergründig so interpretiert werden, dass die Referenz s eine Referenz auf r wäre, weil sie ja mit r initialisiert wird. Der Compiler setzt aber, wie oben beschrieben, auf der rechten Seite für r das referenzierte Objekt i ein. s ist daher nur ein weiterer Alias-Name für i. Mit anderen Worten, wenn nach den obigen Deklarationen einer der Namen i, r oder s benutzt wird, könnte man ihn durch einen der anderen beiden ersetzen, ohne dass ein Programm in seiner Bedeutung geändert wird. Zusammengefasst:

- Auf Objekte wird nur über symbolische Namen (Bezeichner) oder Zeiger zugegriffen. Zeiger (in Kapitel 5 beschrieben) seien hier ausgeklammert.
- Die Bezeichner (Namen) von Referenzen sind nichts anderes als Alias-Namen. Für ein Objekt kann es keinen oder beliebig viele Alias-Namen geben, die wie andere Bezeichner auch verwendet werden.
- Alle Bezeichner für dasselbe Objekt sind in der Verwendung semantisch gleichwertig. Die obige Deklaration int& s = r; hat daher dieselbe Wirkung wie int& s = i;.

1.6.8 Regeln zum Bilden von Ausdrücken

Es gelten im Allgemeinen Vorrangregeln der Algebra beim Auswerten eines Ausdrucks inklusive der Klammerregeln. Die Tabelle 1.9 zeigt die Rangfolge einiger ausgewählter Operatoren. Einige der Operatoren werden erst in folgenden Kapiteln erklärt. Eine ausführliche Auflistung der Operatorenrangfolge finden Sie im Anhang A.4, Seite 890.

Auf gleicher Prioritätsstufe wird ein Ausdruck von links nach rechts abgearbeitet (linksassoziativ), mit Ausnahme der Ränge 2, 14 und 15 der Tabelle, die von rechts abgearbeitet werden (rechtsassoziativ). Zuerst werden jedoch die Klammern ausgewertet. Beispiel:

```
a = b + d + c;    ist gleich mit:   a = ((b + d) + c);    → linksassoziativ
a = b = d = c;    ist gleich mit:   a = (b = (d = c));    → rechtsassoziativ
```

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

```
int total = 0;
int sum = (total = 3) + (++total); // Fehler!
int i = 2;
i = 3 * i++;                       // Fehler !
```

Tabelle 1.9: Präzedenz ausgewählter Operatoren

Rang	Operatoren
1	`.` `[ ]` `f()` (Funktionsaufruf) `++` `--` (postfix)
2	`sizeof` `++` `--` (präfix) `~` `!` `+` `-` (unär) `(Typ) Ausdruck` (C-Stil-Typumwandlung)
4	`*` `/` `%`
5	`+` `-` (binär)
6	`<<` `>>`
7	`<` `>` `<=` `>=`
8	`==` `!=`
9	`&` (bitweises UND)
10	`^` (bitweises exklusiv-ODER)
11	`|` (bitweises ODER)
12	`&&` (logisches UND)
13	`||` (logisches ODER)
14	`?:` (Bedingungsoperator)
15	alle Zuweisungsoperatoren `=`, `+=`, `<<=` usw.

Die Variable `sum` kann hier den Wert 4 oder den Wert 7 annehmen, abhängig von der Reihenfolge, in der die Unterausdrücke in den Klammern berechnet werden. Der Wert von `i` ist undefiniert, weil die Reihenfolge der Auswertung nicht feststeht. Es gibt zwei Möglichkeiten:

1. `3*i` wird berechnet und ergibt 6. Dieser Wert wird `i` zugewiesen. Erst anschließend wird `i` inkrementiert, sodass zum Schluss `i` gleich 7 gilt.
2. `3*i` wird berechnet und ergibt 6. Gleich nach der Berechnung wird `i` von 2 auf 3 inkrementiert. Erst dann erfolgt die Zuweisung des berechneten Ergebnisses an `i`, sodass zum Schluss `i` gleich 6 gilt.

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 gibt. 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 soviele Mantissen-Bits wie eine `double`-Zahl hat (siehe Aufstellung Seite 47).
- 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`).

- Vorzeichenverlust und gleichzeitige Wertänderung, wenn zum Beispiel eine negative `int`-Zahl in eine `unsigned int`-Zahl umgewandelt wird.

Typumwandlung integraler Typen nach int

Jeder R-Wert (Rechts-Wert, (englisch *rvalue*), vgl. Seite 62) 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 55) und `enum` (siehe Seite 80).

float-Typumwandlungen

Jeder R-Wert des Typs `float` kann in einen `double`-Wert 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 geschweifter 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.

Auf verschiedene Arten der Sichtbarkeitsbereiche (Funktionen, Dateien, Klassen) wird später eingegangen. Das folgende Programm zeigt Beispiele für verschiedene Gültigkeits-

bereiche (englisch *scope*). Der im Programm verwendete Operator :: bewirkt den Zugriff auf Variablen, die *global* sichtbar sind, also nicht Zugriff auf den nächsten äußeren Block.

Listing 1.7: 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, b = 2;

int main() {   // Ein neuer Block beginnt.
    cout << "globales a= " << a << endl;  // 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 << endl;
    // Das globale a lässt sich nach der Deklaration des lokalen a nur noch mithilfe des
    // Bereichsoperators :: (englisch scope operator) ansprechen. Ausgabe von ::a :
    cout << "globales ::a= " << ::a << endl;
    {  // 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 << endl;
       cout << "lokales c = " << c << endl;
       // Wie oben beschrieben, ist das globale b nur über den
       // Scope-Operator ansprechbar. Ausgabe von ::b:
       cout << "globales ::b = " << ::b << endl;
    } // Der innere Block wird geschlossen. Damit ist das globale b
      // auch ohne Scope-Operator wieder sichtbar:
    cout << "globales b wieder sichtbar: b = " << b << endl;
    // cout << "c = " << c << endl; // 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 letzten 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 3.6 (ab Seite 141) erklärt. Hier soll nur vorab darauf hingewiesen werden, dass man auch ohne pauschale Nutzung der Standardbibliothek auskommt, wenn die betreffenden Elemente mit einem sogenannten qualifizierten Namen angesprochen werden, der den Namensraum angibt. Bezogen auf den Standard-Namensraum `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;
```

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 zweite oder dritte Möglichkeit wird in allen anderen Fällen empfohlen. Sie werden in den Beispielen daher alle drei Varianten antreffen.

Übung

1.2 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. Hinweis: Studieren Sie die möglichen Operatoren für ganze Zahlen und die Datei *limits.h*.

1.8 Kontrollstrukturen

Kontrollstrukturen dienen dazu, den Programmfluss zu steuern. Im einfachsten Fall werden 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[6] sein:

- 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 (siehe Seite 42), 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) wie `x = 3;`. Eine Zuweisung hat einen Wert, weswegen verkettete Zuweisungen möglich sind:

```
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. Selbst eine Ausgabe auf den Bildschirm ist ein Ausdruck. Der Wert ist das Objekt `cout` selbst, das die Ausgabe bewerkstelligt.

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

[6] Eine vollständige Auflistung ist nicht beabsichtigt.

bedeutetet dasselbe wie

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

Das Ergebnis des Ausdrucks in den runden Klammern ist `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:

- Einem linken Teil, auch *L-Wert* (englisch *lvalue*) genannt, was eine Abkürzung für *Links-Wert* ist.
- Dem Zuweisungsoperator `=`.
- Einem rechten Teil, auch *R-Wert* (englisch *rvalue*) genannt, was eine Abkürzung für *Rechts-Wert* ist.

Dabei wird der L-Wert[7] als (symbolische) *Adresse*, der R-Wert als *Wert* interpretiert. Die Bedeutung einer Zuweisung `a = b;` ist also: Der Wert der Variablen `b` wird an die Adresse der Variablen `a` kopiert, sodass danach Variable `a` denselben Wert hat.

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
}
```

[7] Mehr dazu im Glossar auf Seite 953.

1.8.2 Sequenz (Reihung)

Im einfachsten Fall werden die Anweisungen der Reihe nach durchlaufen:

```
a  = b + 1;
a += a;
cout << "\n Ergebnis =" << 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 geschweifter Klammern, die beliebig viele Anweisungen umschließen können, also auch keine oder nur eine. Abbildung 1.6 zeigt das Syntaxdiagramm einer `if`-Anweisung.

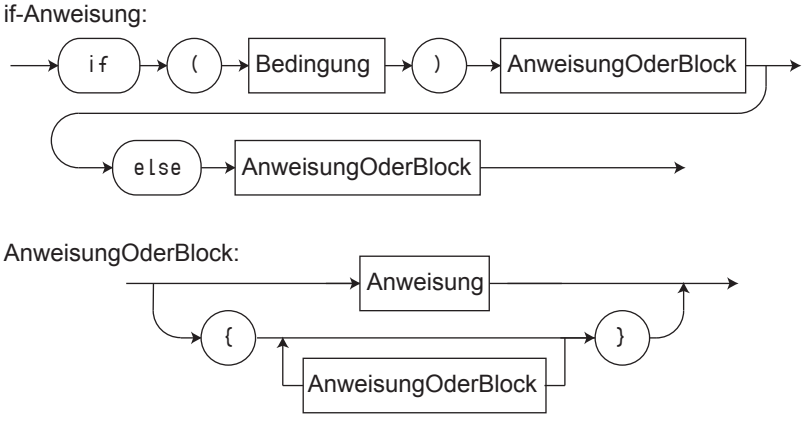


Abbildung 1.6: Syntaxdiagramm einer `if`-Anweisung

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 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 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 abgebrochen werden kann, das den Wahrheitswert *wahr* liefert. Grund ist, dass sich das Ergebnis nach weiteren Berechnungen nicht ändern kann. Umgekehrt braucht man bei UND-Verknüpfungen nicht weiterzurechnen, 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, gewissermaßen 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 nachfolgenden 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, j = 2;  // stimmt's oder nicht?
if(i++ || j++) i++; // i == 2 und j == 3
```

```
int i = 1, j = 2;
if(i++ || j++) i++; // i == 3 und j == 2
```

```
int i = 0, j = 2;
if(i++ && j++) i++; // i ==  1 und j ==  2
```

```
int i = 1, 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 1.8: Umwandlung römischer Ziffern mit if / else

```
// cppbuch/k1/roemzif1.cpp
#include<iostream>
using namespace std;

int main( ) {
   int a = 0;
   char c;
   cout << "Zeichen ?";
   cin >> c;
   if(c == 'I')      a = 1;
   else if(c == 'V') a = 5;
   else if(c == 'X') a = 10;
   else if(c == 'L') a = 50;
   else if(c == 'C') a = 100;
   else if(c == 'D') a = 500;
   else if(c == 'M') a = 1000;
   if(a == 0) {
      cout << "keine römische Ziffer!\n";
   }
   else {
      cout << a << endl;
   }
}
```

Achtung, Falle: Fehler in Verbindung mit if

Ein häufiger Fehler ist die versehentlich falsche Schreibweise des Gleichheitsoperators, sodass sich unfreiwillig der Zuweisungsoperator ergibt.

```
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 Wertes von a. Freundliche Compiler geben an solchen Stellen eine Warnung aus, damit man sich noch einmal überlegen kann, ob man wirklich eine Zuweisung gemeint hat.

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 sollten 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!

Abschließend die Empfehlung, int und unsigned nicht zu mischen. 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 = 0;

if(u < i) {
   cout << u << '<' << i << endl;
}
```

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! Der Grund besteht darin, dass der Compiler im Fall verschiedener Datentypen eine Typumwandlung des zweiten Bedingungsoperanden vornimmt, um den Vergleich durchführen zu können. 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 1.7 zeigt die Syntax.

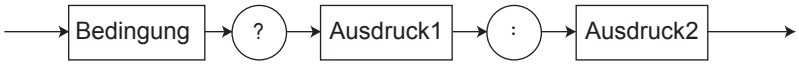


Abbildung 1.7: Syntaxdiagramm des Bedingungsoperators

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

1.3 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.

1.4 Schreiben Sie ein Programm, das drei ganze Zahlen als Eingabe verlangt und dann die größte der Zahlen 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 übersicht-

licher ist daher die Auswahl unter vielen Anweisungen mit `switch`. Abbildung 1.8 zeigt die Syntax, ein Beispiel folgt unten.

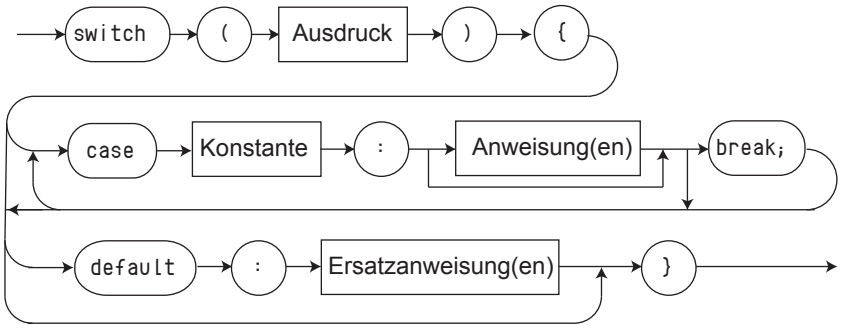


Abbildung 1.8: Syntaxdiagramm einer `switch`-Anweisung

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 passenden Konstante gehörigen Anweisungen ausgeführt. Nach Ausführung wird nicht automatisch aus der `switch`-Anweisung herausgesprungen. Erst `break` führt zum Verlassen der `switch`-Anweisung und sollte stets verwendet werden, um unnötige Tests auf die Folgewerte zu vermeiden. 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 fängt man an dieser Stelle nicht vorgesehene Werte des `switch`-Ausdrucks ab 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 65 gelöst werden, wie das folgende Programm zeigt.

Listing 1.9: Umwandlung römischer Ziffern mit switch

```
// cppbuch/k1/roemzif2.cpp
#include<iostream>
using namespace std;

int main() {
   int a = -1;
   char c;
   cout << "Zeichen ?";
   cin >> c;
   switch(c) {
      case 'I'    : a = 1;   break;
      case 'V'    : a = 5;   break;
```

```
      case 'X'    : a = 10;  break;
      case 'L'    : a = 50;  break;
      case 'C'    : a = 100; break;
      case 'D'    : a = 500; break;
      case 'M'    : a = 1000; break;
      default     : a = 0;
   }
   if (a > 0) {
      cout << "a = " << a << endl;
   }
   else {
      cout <<"keine römische Ziffer!" << endl;
   }
}
```

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

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

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

1.8.5 Wiederholungen

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

Schleifen mit while

Abbildung 1.9 zeigt die Syntax von `while`-Schleifen. *AnweisungOderBlock* ist wie auf Seite 63 definiert. Die Bedeutung einer `while`-Schleife ist: Solange die Bedingung wahr ist, die Auswertung also ein Ergebnis ungleich 0 oder `true` liefert, wird die Anweisung bzw. der Block ausgeführt. Die Bedingung wird auf jeden Fall zuerst geprüft. Wenn die

Bedingung von vornherein unwahr ist, wird die Anweisung gar nicht erst ausgeführt (siehe Abbildung 1.10). Die Anweisung oder den Block innerhalb der Schleife nennt man *Schleifenkörper*. Schleifen können wie `if`-Anweisungen beliebig geschachtelt werden.

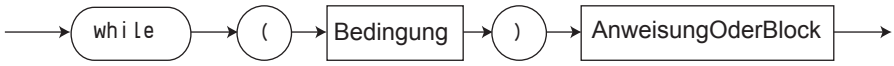
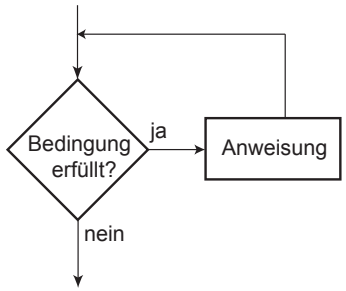


Abbildung 1.9: Syntaxdiagramm einer `while`-Schleife

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




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

Abbildung 1.10: Flussdiagramm für eine `while`-Anweisung

Beispiele

- Unendliche Schleife:

```
while(true) Anweisung
```

- Anweisung wird nie ausgeführt (unerreichbarer Programmcode):

```
while(false) Anweisung
```

- Summation der Zahlen 1 bis 99:

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

- Berechnung des größten gemeinsamen Teilers GGT(x,y) für zwei natürliche Zahlen x und y nach Euklid. Es gilt:

- GGT(x,x), also x = y: Das Resultat ist x.
- GGT(x,y) bleibt unverändert, falls die größere der beiden Zahlen durch die Differenz ersetzt wird, also GGT(x, y) == GGT(x, y–x), falls x<y.

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

Listing 1.10: Beispiel für while-Schleife

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

int main() {
   unsigned int x, y;
   cout << "2 Zahlen > 0 eingeben :";
   cin >> x >> y;
   cout << "Der GGT von " << x << " und " << y << " ist ";
   while( x!= y) {
      if(x > y) {
         x -= y;
      }
      else {
         y -= x;
      }
   }
   cout << x << endl;
}
```

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

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

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

Tipp

Falls der Schleifenkörper aus vielen Anweisungen besteht, sollten die Anweisungen zur Veränderung der Bedingung an den Schluss gestellt werden, um sie leicht finden zu können.

Schleifen mit do while

Abbildung 1.11 zeigt die Syntax einer `do while`-Schleife. *AnweisungOderBlock* ist wie auf Seite 63 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.

Abbildung 1.11: Syntaxdiagramm einer `do while`-Schleife

Im Flussdiagramm ist die Anweisung ein Block (siehe rechts in der Abbildung 1.12). `do while`-Schleifen eignen sich unter anderem gut zur sicheren Abfrage von Daten, indem die Abfrage so lange wiederholt wird, bis die abgefragten Daten in einem plausiblen Bereich liegen, wie im Primzahlprogramm unten zu sehen ist.

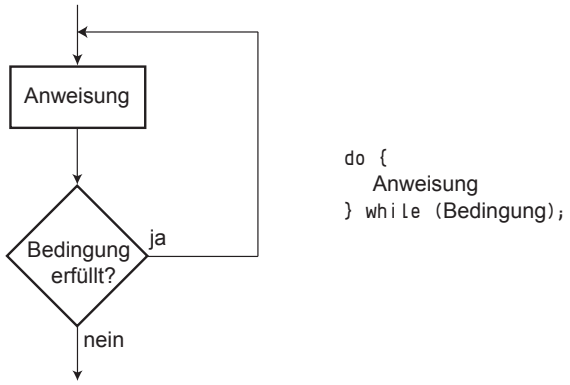


Abbildung 1.12: Flussdiagramm für eine `do while`-Anweisung

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

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

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

Listing 1.11: Beispiel für do while-Schleife

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

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

Schleifen mit for

Die letzte Art von Schleifen ist die `for`-Schleife. Sie wird häufig eingesetzt, wenn die Anzahl der Wiederholungen vorher feststeht, aber das muss durchaus nicht so sein. Abbildung 1.13 zeigt die Syntax einer `for`-Schleife.

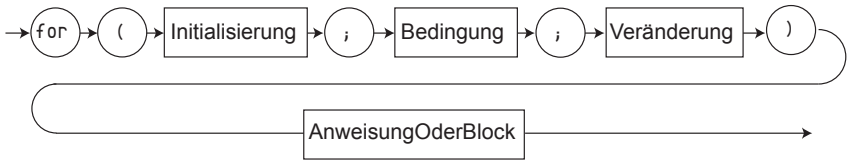


Abbildung 1.13: Syntaxdiagramm einer `for`-Schleife

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

```
for(int i = 65; i <= 69; ++i) {
    cout << i << " " << static_cast<char>(i) << endl;
}
```

Bei der Abarbeitung werden die folgenden Schritte durchlaufen:

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

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

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

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

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

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

Listing 1.12: Beispiel für for-Schleife

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

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

Um Fehler zu vermeiden und zur besseren Verständlichkeit sollte man niemals Laufvariablen in der Anweisung verändern. Das Auffinden von Fehlern würde durch die Änderung erschwert.

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

```
    // noch mehr Programmcode
}
```

Auch wenn der Schleifenkörper nur aus einer Anweisung besteht, wird empfohlen, sie in 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) << endl;  // 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 << endl;
}
```

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 << endl;
}
```

ausführe, werden wie erwartet die drei Zahlen 0.5, 1 und 1.5 angezeigt. Der Grund liegt darin, dass 0.5 intern 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 << endl;
}
```

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:

```
// Formatierung siehe Kapitel 10
cout.setf(ios::showpoint|ios::fixed, ios::floatfield);
cout.precision(15); // angezeigte Genauigkeit
                                  // Ausgabe auf meinem System:
cout << 10.4 << endl;             // 10.400000000000000
cout << (double)10.4f << endl; // 10.399999618530273
```

Wenn man bei `double`-Laufvariablen nur die Operatoren `<` oder `>` zum Vergleich verwendet, 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 in den Bestandteilen `Initialisierung; Bedingung; Veraenderung` 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. Sowohl `i` als auch `sum` bekommen im Initialisierungsteil ihre Anfangswerte zugewiesen:

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

Theoretisch könnte auch die Deklaration von `sum` in den Initialisierungsteil verlegt werden, nur kommt man dann außerhalb der Schleife nicht mehr an den Wert der Variablen. Der Kommaoperator hat die niedrigste Priorität von allen Operatoren (siehe Tabelle A.4 auf Seite 890).

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 1.13: Menü mit break und continue

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

int main() {
   char c;
   while(true) {                    // unendliche Schleife
      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, kann man eine Hilfsvariable eingeführen, die den Abbruch signalisiert (siehe Übungsaufgabe 1.9).

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

1.5 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. 32 Bit-Positionen, als »Maske«. Beispielergebnisse:

5 → `00000000000000000000000000000101`

-5 → `11111111111111111111111111111011`

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

1.6 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;
}
```

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

(c)
```
int n = 100;
int sum = n*(n+1)/2;
```

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

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

1.8 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.

1.9 Formulieren Sie das Programm in Abschnitt 1.8.6 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.

1.9 Benutzerdefinierte und zusammengesetzte Datentypen

Abgesehen von den Grunddatentypen gibt es Datentypen, die aus den Grunddatentypen zusammengesetzt sind und die Sie selbst definieren können (benutzerdefinierte Datentypen).

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 unserem 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 Erweiterung der vordefinierten Typen durch den Programmierer darstellen. Abbildung 1.14 zeigt das Syntaxdiagramm einer `enum`-Deklaration.

Abbildung 1.14: Syntaxdiagramm einer `enum`-Deklaration

Typname oder Variablenliste können weggelassen werden (in diesem Fall aber nicht beide). Sinnvoll ist meistens nur das Weglassen der Variablenliste. Der neue Datentyp `Farbtyp` wird deklariert:

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

Der neue Datentyp `Wochentag` wird deklariert:

```
enum 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, einWerktag, // Definitionen
                heute = dienstag; // Definition + Initialisierung
```

Falls ein Aufzählungstyp nur ein einziges Mal in einer Variablendefinition benötigt wird, kann der Typname weggelassen werden. Man erhält dann eine *anonyme Typdefinition:*

```
enum {fahrrad, mofa, lkw, pkw} einFahrzeug;
```

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. Eine Voreinstellung mit anderen Zahlen ist möglich, wird aber nur gelegentlich erforderlich sein, vielleicht bei einer gewünschten binären Codierung, zum Beispiel

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

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

// Deklaration mit Bitshift-Operator
enum Bitmaske { wert1 = 1 << 0, wert2 = 1 << 1, wert3 = 1 << 2,
                wert4 = 1 << 3, wert5 = 1 << 4, wert6 = 1 << 5,
               // ... usw.
};
```

Die in `Farbtyp` definierten Farben `rot`, `gruen`, `blau`, `gelb` dürfen in `Palette` nicht mehr verwendet werden, sofern sich `Palette` im gleichen Gültigkeitsbereich befindet. Eine Umwandlung in `int` ist erlaubt, aber nicht die Umwandlung einer `int`-Zahl in einen `enum`-Typ. Als Operation auf `enum`-Typen ist nur die Zuweisung erlaubt, bei allen anderen Operatoren wird vorher in `int` gewandelt. Welche Anweisungen möglich oder falsch sind, zeigen die nächsten Zeilen, wobei die Variablennamen sich auf die obigen Definitionen beziehen:

```
int i = dienstag;       // richtig (implizite Umwandlung nach int)
heute = montag;         // richtig
heute = i;              // Fehler, Datentyp inkompatibel
montag = heute;         // Fehler (montag ist Konstante)
i = rot + blau;         // möglich (implizite Umwandlung nach int)
mischung = weiss + lila; // Fehler, Rückwandlung des int-Ergebnisses
                        // in Typ Palette ist nicht möglich
++mischung;             // Fehler, gleicher Grund
if(mischung > grau)     // implizite Typumwandlung
   mischung = lila;     // richtig
```

Aufgrund der Umwandlung in `int` bei arithmetischen Operationen sind Operationen wie `mischung++` fehlerhaft, da sich ein undefinierter Wert ergeben kann.

1.9.2 Arrays: Der C++-Standardtyp vector

Im täglichen Leben benutzen wir häufig Tabellen, auch Arrays 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++, sind aber nicht besonders komfortabel und werden daher erst in Abschnitt 5.2 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 Softwareentwickler sind beide Sichten wichtig. Hier wird der Standardtyp `vector` zunächst nur benutzt, und erst viel weiter unten wird erklärt, was im Innern vorgeht. Die Klasse ist eine Beschreibung für Objekte, wie aus Abschnitt 1.2 der Einführung bekannt ist. Die Anweisung

```
vector<int> v(10);
```

stellt einen Vektor `v` bereit, der 10 Elemente des Typs `int` aufnehmen kann. Die Feldelemente sind stets von 0 bis (Anzahl der Elemente –1) durchnummeriert, hier also von 0 bis 9. Die Klasse für Vektoren stellt einige Dienstleistungen zur Verfügung, die mit der in der Einführung beschriebenen Notation abgerufen werden können. Eine dieser Dienstleistungen ist die Ermittlung der Zahl der Elemente, das heißt die Größe (englisch *size*) des Vektors:

```
cout << v.size() << endl;          // 10
```

Daten müssen in diesem Fall nicht zwischen den runden Klammern übergeben werden, das Vektor-Objekt enthält alle nötigen Informationen. Der Zugriff auf ein spezielles Element wird mit dem Indexoperator `[]` realisiert. Zum Beispiel zeigt

```
cout << v[0] << endl;              // 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

Den Compiler so einzustellen, dass der Index bei jedem Zugriff über die eckigen Klammern geprüft wird, geht leider nicht. Es gibt in C++ jedoch andere Möglichkeiten dafür (Abschnitt 9.2), die natürlich ein Programm verlangsamen (meist nur geringfügig). Man kann alternativ auf die eckigen Klammern verzichten und einen Vektor nach einem Wert *an* (englisch *at*) einer Position fragen. Diese Art des Zugriff wird geprüft:

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

Im Beispiel unten kann man sich dadurch helfen, 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 man ganz sicher sein, in der `for`-Schleife `i < v.size()` abzufragen anstatt `i <= v.size()`... Wer sich leicht vertippt, sollte also doch lieber `v.at(i)` statt `v[i]` schreiben.

Ein Beispiel

Das Programm unten verdeutlicht die Arbeitsweise mit Vektoren. Es werden einige typische Operationen demonstriert, die sich weitgehend selbst erklären. Das verwendete Sortierverfahren ist eine Variante des Bubble-Sorts und für große Tabellen zu langsam. Große Tabellen sollten mit schnellen Verfahren sortiert werden (siehe Seite 135 und die nach dem Programm angegebene Alternative).

Listing 1.14: Standardklasse vector

```
// cppbuch/k1/vektor.cpp
#include<iostream>
#include<vector>   // Standard-Vektor bekannt machen
#include<cstddef>  // size_t
using namespace std;

int main() {        // Programm mit typischen Vektor-Operationen
   vector<float> kosten(12);  // Tabelle mit 12 float-Werten
   // Füllen der Tabelle mit beliebigen Daten, dabei Typumwandlung int → float
   for(size_t i = 0; i < kosten.size(); ++i) {
      kosten[i] = static_cast<float>(150-i*i)/10.0;
   }
   // Als Typ der Laufvariablen i in der Schleife wird der auf Seite 47 beschriebene Typ
   // size_t statt int gewählt, weil die Anzahl der Elemente ja ≥ 0 sein muss. Tabelle ausgeben:
```

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

    // zweite Tabelle sortierteKosten deklarieren und mit der ersten initialisieren
    vector<float> sortierteKosten = kosten;
    // zweite Tabelle aufsteigend sortieren (Bubble-Sort-Variante)
    // (schnellere Möglichkeit siehe Text unten)
    for(size_t i = 1; i < sortierteKosten.size(); ++i) {
       for(size_t j = 0; j < i; ++j) {
          if(sortierteKosten[i] < sortierteKosten[j]) {
              // Elemente an i und j vertauschen
              float temp = sortierteKosten[i];
              sortierteKosten[i] = sortierteKosten[j];
              sortierteKosten[j] = temp;
          }
       }
    }
    // Tabelle ausgeben
    for(size_t i = 0; i < sortierteKosten.size(); ++i) {
        cout << i << ": " << sortierteKosten[i] << endl;
    }
}
```

Die Standardbibliothek bietet eine schnellere und kürzere Alternative: Dazu muss erst `#include<algorithm>` am Programmanfang eingefügt werden. Dann wird der Bubble-Sort durch den Aufruf

```
sort(sortierteKosten.begin(), sortierteKosten.end());
```

ersetzt. Die Grundlagen dafür finden sich in den Abschnitten 11.2 und 24.4.2. An der Stelle

```
vector<float> sortierteKosten = kosten; // Initialisierung
```

wäre auch Folgendes möglich gewesen:

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

In C++ ist eine Initialisierung *keine* Zuweisung. Initialisierung und Zuweisung werden in C++ unterschieden. Beides ist trotz desselben Operators (=) leicht auseinander zu halten:

Merke:
Eine Initialisierung kann nur bei der gleichzeitigen Definition (= Erzeugung) eines Objekts auftreten, eine Zuweisung setzt immer ein schon vorhandenes Objekt voraus.

Wenn man die Wahl hat so wie hier, sollte 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. Im Programm oben wurde der Vektor in einer Schleife mit Werten versehen. Es ist aber auch eine direkte Initialisierung möglich, etwa

```
vector<double> einVektor = {1.1, 2.2, 3.3, 4.4, 5.5}; // direkte Initialisierung
```

Lineare Suche in Tabellen

Hier seien vier programmiertechnische Möglichkeiten gezeigt, in einer *unsortierten* Tabelle `a` mit N Elementen auf den Positionen 0...N – 1 ein bestimmtes Element `key` zu suchen. 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 `a` 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 alle vier Fälle
const int N = ...
vector<int> a(N+1);  // letztes Element nur für Fall 4
int key = ...        // gesuchtes Element
int i;               // Laufvariable
                     // Ergebnis: i = 0..N – 1 : gefunden, i = N : nicht gefunden!
```

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

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

2. `do while`-Schleife
   Wie oben, nur dass die Zählvariable *vorher* inkrementiert und daher anders vorbesetzt wird. Die vorhergehende Lösung sollte 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 && a[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(a[i] == key) {
      break;
   }
}
```

4. (N+1). Element als »Wächter« (sentinel)
   In das zusätzliche Element `a[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;
a[N] = key;              // garantiert Abbruch der Schleife
while(a[++i] != key);
```

Vektoren sind dynamisch!

Oft weiß man nicht, 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 Computerspeicher sein sollte, wird der gesamte Vektor an eine neue, ausreichend große Stelle im Speicher verlagert. Dies geschieht ohne Zutun des Programmierers. Mit `push_back` wird er dazu aufgefordert, wobei ihm der anzuhängende Wert in runden Klammern übergeben wird (siehe Beispielprogramm). Eine tabellarische Übersicht der Möglichkeiten von Objekten der Klasse `vector` ist in Abschnitt 28.2.1 zu finden. Ein Großteil der Möglichkeiten ist aber erst nach Kenntnis der folgenden Kapitel bis einschließlich Kapitel 9 verständlich.

Listing 1.15: Vektor dynamisch vergrößern

```
// cppbuch/k1/dynvekt.cpp
#include<iostream>
#include<vector>   // Standard-Vektor
#include<cstddef>  // size_t
using namespace std;

int main() {
   vector<int> meineDaten;   // anfängliche Größe ist 0
   int wert;
   do {
      cout << "Wert eingeben (0 = Ende der Eingabe):";
      cin >> wert;
      if(wert != 0) {
         meineDaten.push_back(wert); // Wert anhängen
      }
```

```
   } while(wert != 0);
   cout << "Es wurden die folgenden Werte eingegeben:\n";
   for(size_t i = 0; i < meineDaten.size(); ++i) {
      cout << i << ". Wert : "
           << meineDaten[i] << endl;
   }
}
```

1.9.3 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 tabellarische Übersicht der Möglichkeiten von Strings ist in Kapitel 32 zu finden. Wie bei der Standard-Vektorklasse ist ein Großteil der Möglichkeiten erst nach Kenntnis der folgenden Kapitel bis einschließlich Kapitel 9 verständlich. Wie die Klasse `string` im Innern aufgebaut ist, wird an anderer Stelle erläutert (Abschnitt 6.1).

Listing 1.16: Standardklasse string

```
// cppbuch/k1/zbstring.cpp
#include<iostream>
#include<string>   // Standard-String einschließen
#include<cstddef>  // size_t
using namespace std;

int main() {       // Programm mit typischen String-Operationen
   // String-Objekt einString anlegen und mit "hallo" initialisieren.
   // einString kann ein beliebiger Name sein.
   string einString("hallo");
   cout << einString << endl; // String ausgeben

   // String zeichenweise ausgeben, ungeprüfter Zugriff wie bei vector:
   for(size_t i = 0; i < einString.size(); ++i) {
      cout << einString[i];
   }
   cout << endl;
   // String zeichenweise mit Indexprüfung ausgeben. Die Prüfung geschieht wie beim Vektor.
   // Ein Versuch, einString.at(i) mit einem i ≥ einString.size() abzufragen,
   // führt zum Programmabbruch mit Fehlermeldung. Die Anzahl der Zeichen kann bei Strings
   // auch mit length() ermittelt werden.
   for(size_t i = 0; i < einString.length(); ++i) {
      cout << einString.at(i);
   }
   cout << endl;
   string eineStringKopie(einString); // Kopie des Strings erzeugen
   cout << eineStringKopie << endl; // hallo
```

```
    string diesIstNeu("neu!");
    eineStringKopie = diesIstNeu;     // Kopie durch Zuweisung
    cout << eineStringKopie << endl; // neu!

    eineStringKopie = "Buchstaben";   // Zeichenkette zuweisen
    cout << eineStringKopie << endl; // Buchstaben

    // Zuweisung nur eines Zeichens vom Typ char
    einString = 'X';
    cout << einString << endl;        // X

    // Strings mit dem +=-Operator verketten
    einString += eineStringKopie;
    cout << einString << endl;        // XBuchstaben

    // Strings mit dem +-Operator verketten
    einString = eineStringKopie + " ABC";
    cout << einString << endl;        // Buchstaben ABC
    einString = "123" + eineStringKopie;
    cout << einString << endl;        // 123Buchstaben
    einString = "123" + "ABC";  // geht nicht! Erklärung folgt in Kap. 9
} // Ende von main()
```

Übungen

1.10 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.

1.11 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 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. (Etwas schwierig und nur für Knobelfreunde!)

1.12 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`). Zur Speicherung der Intervalle soll ein `vector<int>` verwendet werden.

1.13 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, wieviele 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 Höchstwertes 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`.

1.9.4 Strukturen

Ein Vektor hat nur Elemente desselben Datentyps. Häufig möchte man logisch zusammengehörige Daten zusammenfassen, die *nicht* vom selben Datentyp sind, zum Beispiel Vornamen und Nachnamen je vom Typ `string`, Status vom Typ `enum {Sachbearbeiter, Gruppenleiter, Abteilungsleiter}`, Monatsgehalt vom Typ `double` sowie weitere Daten, die zu einem Personaldatensatz zusammengefasst werden sollen. Ein anderes Beispiel sind die zusammengehörigen Daten eines Punktes auf dem Bildschirm, also seine Koordinaten `x` und `y`, seine Farbe, und ob er sichtbar ist. Für diese Zwecke wird von C++ die *Struktur* bereitgestellt, die einen Datensatz zusammenfasst.

Vorweg sei bemerkt, dass eine Struktur in C++ nichts anderes als eine Klasse mit öffentlichen Elementen ist. In Einschränkung dazu enthält eine Struktur im Sinne dieses Abschnitts nur Daten (und keine Funktionen) und entspricht einem Record der Sprache Pascal. Mehr über Klassen finden Sie in Kapitel 4.

Strukturen sind benutzerdefinierte Datentypen. Sie werden mit einer Syntax definiert, die bis auf den inneren Teil der Syntax von Aufzählungstypen ähnelt (siehe Abbildung 1.15).

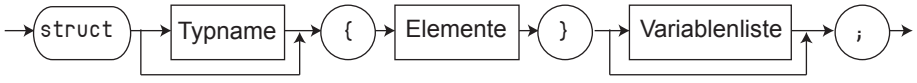


Abbildung 1.15: Syntaxdiagramm einer `struct`-Definition

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 *Felder* oder

Komponenten). Im Beispiel werden zwei Variablen `p` und `q` vom Datentyp `Punkt` definiert und der Zugriff auf die internen Größen gezeigt.

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

struct Punkt {              // Punkt ist ein Typ.
    int x;
    int y;
    bool istSichtbar;
    Farbtyp dieFarbe;
} p;                        // p ist ein Punkt-Objekt

// q ist ebenfalls ein Punkt-Objekt:
Punkt q;                    // noch undefinierter Inhalt

// Zugriff, hier: mit Werten versehen
p.x = 270;   p.y = 20;      // Koordinaten von p
p.istSichtbar = false;
p.dieFarbe  = gelb;
```

Die internen Elemente sind nicht allein zugreifbar, weil sie nur in Verbindung mit einem Objekt existieren. Die Anweisung `dieFarbe = rot;` ist unsinnig, weil nicht klar ist, welcher Punkt rot werden soll. `p.dieFarbe = rot;` hingegen sagt eindeutig, dass der Punkt `p` gefärbt werden soll. Der Zugriff geschieht über einen Punktoperator ».« zwischen Variablenname und Elementnamen, wenn die Variable vom Typ `struct` ist.

Ein `struct` kann in einem Schritt initialisiert werden, wie das folgende Programm zeigt. Bei der Ausgabe wird der `bool`-Wert `false` in 0 umgewandelt, und der `enum`-Wert `gelb` in 1 (`rot` ergäbe 0).

Listing 1.17: Beispiel mit struct

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

enum Farbtyp { rot, gelb, gruen };

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

int main() {
   Punkt p1 = { 100, 200, false, gelb }; // direkte Initialisierung

   cout << "p1.x = " << p1.x << " p1.y = " << p1.y
        << "  p1.istSichtbar= " << p1.istSichtbar
        << "  p1.dieFarbe= " << p1.dieFarbe << endl;
}
```

Übung

1.14 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.

1.9.5 Typermittlung mit auto

Das Schlüsselwort `auto` sagt dem Compiler, er soll selbst den Typ bei der Initialisierung ermitteln. Das kann die Schreibarbeit verringern und vermeidet Tippfehler bei komplexen Datentypen. Beispiele:

```
auto a = 2;      // a ist vom Typ int
auto b = 2.9;    // b ist vom Typ double
```

Das folgende Programm zeigt verschiedene Anwendungen von `auto` bei selbstgeschriebenen und Standard-Datentypen:

Listing 1.18: Beispiele mit auto

```
// cppbuch/k1/auto.cpp
#include<iostream>
#include<vector>   // Standard-Vektor einschließen
#include<string>   // Standard-String einschließen
#include<cstddef>  // size_t
using namespace std;

struct Punkt {
   int x;
   int y;
};

int main() {
   Punkt p1 = { 100, 200 };
   auto p2 = p1;         // p2 ist vom Typ Punkt
   cout << "p2.x= " << p2.x << " p2.y= " << p2.y << endl;

   vector<double> v1 = {1.1, 2.2, 3.3, 4.4, 5.5};
   auto v2 = v1;         // v2 ist vom Typ vector<double>
   for(size_t i = 0; i < v2.size(); ++i) {
      cout << i << ": " << v2[i] << endl;
   }

   string s1("Ende!");
   auto s2 = s1;         // s2 ist vom Typ string
   cout << s2 << endl;
}
```

In den obigen Fällen sind die Typnamen recht kurz. Dies ist jedoch in Teilen der C++-Standardbibliothek anders, so dass dort das Schlüsselwort `auto` erheblich zur Erleichte-

rung der Schreibarbeit beiträgt. Im Programm liegen die Anweisungen mit `auto` direkt nach der Deklaration, auf die Bezug genommen wird. Bei größeren Programmen ist das nicht unbedingt der Fall, sodass sich als weiterer Vorteil ergibt, dass man nicht irgendwo weiter oben im Programm oder in der Dokumentation der Standardbibliothek die genaue Deklaration nachschlagen muss.

1.9.6 Unions und Bitfelder[8]

Unions

Unions sind Strukturen, in denen verschiedene Elemente *denselben* Speicherplatz bezeichnen. Daher sind Unions nur in Sonderfällen einzusetzen. Im folgenden Beispiel werden eine `int`-Zahl und ein `char`-Array überlagert. Eine Variable vom Typ `char` belegt genau ein Byte. Das Array hat hier also genau so viele Elemente, wie eine `int`-Zahl Bytes hat. Der gesamte Speicherplatz einer Variablen vom Typ `union` ist identisch mit dem Speicherplatzbedarf des jeweils größten internen Elements, in diesem Beispiel also `sizeof(int)` Bytes. Das `unsigned char`-Array `c` belegt denselben Platz wie `zahl`. Die Bezeichnungen `was.zahl` und `was.c[]` beziehen sich auf denselben Bereich im Speicher, nur dass die Interpretation verschieden ist. Das Programmbeispiel gibt die interne Repräsentation einer `int`-Zahl Byte für Byte aus.

Listing 1.19: Union

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

union HiLo {
    int zahl;
    unsigned char c[sizeof(int)];
};

int main() {
    HiLo was;
    do {
       cout << " Zahl eingeben (0 = Ende):";
       cin >> was.zahl;
       cout << "Byte-weise Darstellung von "
            << was.zahl << endl;
       for(int i = sizeof(int)-1; i >= 0; --i) {
           cout << "Byte Nr. " << i << " = "
                << static_cast<int>(was.c[i]) << endl;
       }
    } while(was.zahl);
}
```

Bitfelder

Gerade in der Hardware-nahen oder Systemprogrammierung ist es oft wünschenswert, Bitfelder verschiedener Länge anzulegen, in denen die einzelnen Bitpositionen verschie-

[8] Dieser Abschnitt kann beim ersten Lesen übersprungen werden.

dene Bedeutungen haben. Die Deklaration eines Bitfelds in C++ hat die Syntax *Datentyp Name* `:`*Konstante*`;`, wobei *Datentyp* `char`, `short`, `int`, `long` (mit den vorzeichenlosen `unsigned`-Varianten) oder ein Aufzählungstyp sein kann. Beispiel: `int Bitfeld : 13;`.

Es dürfen keine Annahmen darüber getroffen werden, wo die 13 Bits innerhalb eines 16- oder 32-Bit-Worts angeordnet sind – dies kann je nach Implementation verschieden sein. Es gibt keine Zeiger oder Referenzen auf ein Bitfeld. Bitfelder sollten nicht zum Sparen von Speicher benutzt werden, weil der Aufwand des Zugriffs auf einzelne Bits beträchtlich sein kann und nicht einmal sicher ist, ob wirklich Speicher gespart wird: Es *könnte* sein, dass eine Implementation jedes Bitfeld auch der Länge 1 stets an einer 32-Bit-Wortgrenze beginnen lässt. Üblich ist allerdings, dass aufeinanderfolgende Bitfelder aneinandergereiht werden. Beispielprogramm zum Zugriff auf Bitfelder (es sind die für `int`-Typen erlaubten Bit-Operatoren möglich):

Listing 1.20: Bitfeld

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

struct Bitfeldstruktur {
    unsigned int a : 4; // a und b sind Bitfelder
    unsigned int b : 3;
};

int main() {
    Bitfeldstruktur x;
    x.a = 06;
    x.b = x.a | 03;
    // Umwandlung in unsigned und Ausgabe
    cout << x.b << endl;
}
```