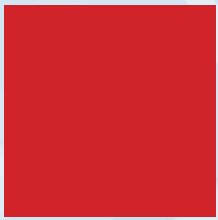
ulrich BREYMANN

C++



eine Einführung

Entwicklungsumgebung, Compiler, alle Beispiele und mehr auf www.cppbuch2.de

Für Windows, Linux & Mac

HANSER

Breymann

C++
eine Einführung

Ulrich Breymann

C++

eine Einführung

HANSER

Prof. Dr. Ulrich Breymann lehrte Informatik an der Fakultät Elektrotechnik und Informatik der Hochschule Bremen
Kontakt: breymann@hs-bremen.de

Bibliografische Information der Deutschen Nationalbibliothek:

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

Lektorat: Brigitte Bauer-Schiewek
Herstellung: Irene Weilhart
Copy editing: Petra Kienle, Fürstenfeldbruck
Layout: Ulrich Breymann mit LaTeX
Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München
Umschlagrealisation: Stephan Rönigk
Druck und Bindung: Kösel, Krugzell
Printed in Germany

Print-ISBN: 978-3-446-44637-3
E-Book-ISBN: 978-3-446-44912-1

Inhalt

Vorwort

Zur Entstehung dieses Buchs

Als Verfasser des recht umfangreichen Werks »Der C++-Programmierer« habe ich überlegt, dass manche Interessierte einen leichten Einstieg suchen, noch ohne sehr in die Tiefe gehen zu wollen. Ein anderer Grund ist, dass es während der Entstehung dieses Buchs keines in einem vergleichbaren Format gab, das sowohl aktuell war als auch fachlich überzeugte.

Für wen ist dieses Buch geschrieben?

Dieses Buch ist für alle geschrieben, die eine kompakte Einführung in die Programmierung mit C++ suchen und noch keine Programmiererfahrung in C++ haben, aber die Bedienung ihres Computers kennen. Erstes Ziel ist es, schnell einen Einstieg in die eigene Programmierung mit C++ zu ermöglichen. Es ist wie beim Autofahren: Dazu müssen Sie nur in Grundzügen wissen, wie der Motor im Einzelnen funktioniert. Das zweite Ziel ist, Sie mit den modernen Konzepten von C++ vertraut zu machen.

Die verwendeten Beispiele sind zum großen Teil Spiele, auch gibt es eine Einführung in die Grafikprogrammierung. Der Vorteil ist, dass Sie nicht nur die objektorientierte Programmierung kennenlernen, sondern auch den Umgang mit Mausklicks und Grafik, obwohl Grafik kein Bestandteil des C++-Standards ist. Das ist für diejenigen, die keine Spiele programmieren wollen, kein Nachteil. Die Programmierung von Spielen ist so

anspruchsvoll wie die Programmierung einer Steuerberechnung – aber interessanter. In diesem Buch wird die Grafik durch eine externe Bibliothek realisiert, die für Spiele besonders geeignet ist. Compiler und alle Beispiele zum Download finden Sie auf *http://www.cppbuch2.de*.

Ist C++ einfach oder schwer?

C++ ist einfach, wenn man sich an die in diesem Buch vorgestellten Empfehlungen zur Programmentwicklung hält. Der Lernaufwand für das Ziel, C++ programmieren zu können, wird durch eine verständliche Darstellung möglichst klein gehalten – er ist aber nicht Null! Mal eben schnell C++ lernen ist aufgrund der Komplexität der Sprache unrealistisch. Der Schwerpunkt des Buchs liegt auf den modernen Konzepten der Programmiersprache C++, nicht im Versuch, sie vollständig zu beschreiben.

C++ kann schwer sein, wenn eine fehleranfällige Konstruktion wie etwa eine eigene Speicherverwaltung mit Zeigern gebaut werden soll. C++ ist auch schwer, wenn Sie ohne entsprechende Vorbildung versuchen, manche Teile der Dokumentation [ISOC++] oder den Programmcode der C++-Standardbibliothek zu verstehen. Die Schwierigkeiten verringern sich natürlich mit zunehmender Kenntnis der Sprache. Dabei helfen neben diesem Buch mein genanntes Werk [Br] und die anderen im Literaturverzeichnis genannten Quellen.

Danksagung

Meinen Testleserinnen und Testlesern Jendrik Bulk, Katrin Grunert, Tim Pissarczyk, Rahel Schmied und Sophia Zell danke ich herzlich für die verschiedenen Anregungen. Frau Weilhart vom Hanser-Verlag danke ich für die zahlreichen Hinweise zur Verbesserung des Layouts.

Bremen, im April 2016

Ulrich Breymann

1 Erste Grundlagen

Fragen, die dieses Kapitel beantwortet:

- Wie arbeitet C++ mit dem Computer zusammen?
- Was macht ein Compiler und wie ist er zu benutzen?
- Wo erhalte ich die Beispiele zu diesem Buch?
- Wo gibt es Hilfe?
- Wie sieht ein einfaches Programm aus?
- Welche Zahltypen gibt es?
- Was sind Zeichen und Strings und wie nutze ich sie?

1.1 Historisches

C++ wurde etwa ab 1980 von Bjarne Stroustrup als die Programmiersprache »C with classes«, die Objektorientierung stark unterstützt, auf der Basis der Programmiersprache C entwickelt.

1998 wurde C++ erstmals von der ISO (International Standards Organisation) und der IEC (International Electrotechnical Commission) standardisiert. Diesem Standard haben sich nationale Standardisierungsgre-

mien wie ANSI (USA) und DIN (Deutschland) angeschlossen. Das C++-Standardkomitee hat kontinuierlich an der Verbesserung von C++ gearbeitet, sodass 2003, 2011 und 2014 neue Versionen des Standards herausgegeben wurden. Die Kurznamen sind dementsprechend C++03, C++11 und C++14. Dieses Buch berücksichtigt den neuesten Standard.

Viele Entwickler[1] im technisch-wissenschaftlichen Bereich bevorzugen C++ wegen der Vielseitigkeit, der Eignung zur hardwarenahen Programmierung und der Geschwindigkeit. Es gibt sogar (mindestens einen) Menschen mit einem C++-Tattoo (Bild) – das nenne ich echte Begeisterung! C++ wird in vielen großen Projekten eingesetzt. Dazu gehören[2] die bekannten Adobe-Produkte Photoshop, Acrobat und Illustrator, das Windows- und andere Betriebssysteme, Microsoft-Office, die Web-Browser Chrome und Firefox, einige Facebook-Komponenten, die Datenbank MySQL sowie viele weitere.

1.2 Die Programmiersprache C++ und die Maschine

Ein Algorithmus ist eine Abfolge von festgelegten Schritten zur Lösung einer Aufgabe, ähnlich einem Rezept zum Backen eines Kuchens. So haben Sie in der Schule gelernt, wie man Zahlen addiert, sodass Sie im Prinzip die Rechnung in einem Restaurant auch ohne Taschenrechner kontrollieren könnten. Ein von Ihnen geschriebenes Programm, auch Quellprogramm genannt, ist nichts anderes als ein in einer Programmiersprache geschriebener Algorithmus. Der Prozessor[3] (CPU = central processing unit) Ihres Computers (= die Maschine) versteht Ihr Programm jedoch nicht ohne Weiteres. Der Prozessor versteht nur binäre Daten, die also als Folge von Bits[4] vorliegen. Ein ausführbares Programm besteht

1 Geschlechtsbezogene Formen meinen stets Frauen, Männer und alle anderen.
2 *http://www.stroustrup.com/applications.html*
3 Siehe Erläuterung im Glossar Seite 387.
4 Abkürzung des englischen Worts »binary digits«, also Nullen und Einsen

damit aus einer Folge von Bits, die zu Rechnerworten gruppiert werden. Deswegen wird der Programmtext (Ihr programmierter Code) vom Compiler in die Sprache des Prozessors (die Maschinensprache) übersetzt. Das Ergebnis ist ein *ausführbares* Programm, das Sie direkt starten, also von dem Prozessor ausführen lassen können. Betrachten Sie die C++-Anweisung

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

Die Größen a, b und c seien Variablen (d.h. veränderbar), die ganze Zahlen darstellen. Die Anweisung besagt, dass die Werte von a und b addiert werden. Das Ergebnis wird dann c zugewiesen. Wenn also a = 3 ist und b = 5, hat c nach der Anweisung den Wert 8.

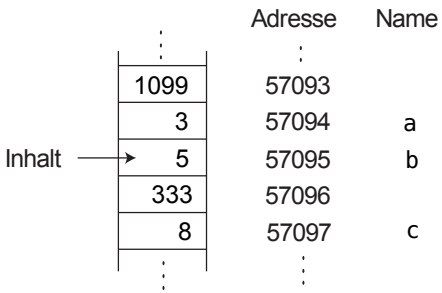


Bild 1.1 Speicherbereiche mit Adressen

Die CPU holt sich den Wert von a aus dem Speicher (siehe Abbildung 1.1) und legt ihn in einem Register ab. Anschließend wird der Befehl »addiere dazu den Wert von der Stelle b« ausgeführt. Dazu muss natürlich auch der Wert von b geholt werden. Dann wird die Addition ausgeführt und das Ergebnis an die Stelle c geschrieben. Das Gute ist, dass man sich um die Adressen der Speicherzellen gar nicht kümmern muss – das erledigt der Compiler. Im Programmtext greifen wir auf die entsprechenden Stellen einfach über die Namen a, b und c zu.

Ein Compiler ist ebenso ein ausführbares Programm, das auf Ihrem Computer bereits vorhanden ist oder das Sie sich aus dem Internet

herunterladen können (dazu später mehr). Der Übersetzungsvorgang, Compilation genannt, durchläuft im Groben(!) die folgenden Schritte:

- *Vorverarbeitung.* Unter anderem ist das die Beseitigung von Kommentaren. Kommentare sind Beschreibungen für den menschlichen Leser, die das Verständnis des Programmcodes erhöhen, für die Maschine aber unwesentlich sind. Der C++-Compiler ruft dazu ein spezielles Programm auf, den *Präprozessor*.
- *Analyse der Grammatik* (auch Syntax genannt). In dieser Phase prüft der Compiler, ob Sie sich an die Regeln der Programmiersprache gehalten haben. Wenn nicht, gibt es eine Fehlermeldung.
- *Zuordnung von Bedeutungen zu den Symbolen.* So weiß der Compiler, wenn er das Symbol + zwischen zwei Zahlen findet, dass eine Addition ausgeführt werden soll.
- *Erzeugung von Zwischencodedateien für die einzelnen Programmdateien.* Ein Programm besteht meistens aus mehreren Dateien.
- *Erzeugung des ausführbaren Programms durch Zusammenbinden der Zwischencodedateien.* Das Zusammenbinden heißt auch Linken. Das vom Compiler zu diesem Zweck aufgerufene Programm ist der Linker.

1.3 Werkzeuge zum Programmieren

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

1.3.1 Der Editor

Wenn Sie ein Programm schreiben wollen, brauchen Sie einen passenden Editor. Word und ähnliche Programme sind nicht das Richtige, weil in den Dateien versteckte Formatierungen enthalten sind. Für Windows-Systeme eignet sich Notepad++ sehr gut, für Linux etwa Kwrite, um nur eine Auswahl zu nennen. OS X (Mac) stellt TextEdit zur Verfügung, besser geeignet ist jedoch der Editor von Xcode (siehe unten). Wenn Sie im

Finder eine der *cpp*-Dateien der heruntergeladenen Beispiele anklicken, wird sie automatisch im Xcode-Editor geöffnet. Sie können sie modifizieren und mit ⇧+⌘+S unter einem anderen Namen abspeichern, um damit zu experimentieren.

Solche Editoren werden auch ASCII[5]-Editoren genannt, obwohl sie auch Umlaute wie ä, ö, ü usw. verarbeiten können. Der ASCII-Standard definiert 128 Zeichen, wie etwa Ziffern, Kleinbuchstaben, Großbuchstaben und einige Sonderzeichen. Umlaute sind nicht enthalten (siehe Tabellen ab Seite 376). In einem Programmtext sollten Sie auf Umlaute verzichten, weil manche Systeme sie nicht vertragen.

1.3.2 Der Compiler

Auf Linux-Systemen ist der Open-Source-C++-Compiler g++[6] meistens vorhanden, oder Sie können ihn leicht mit Hilfe der Systemverwaltung installieren. Weitere Hinweise finden Sie ab Seite 371.

Von dem genannten g++-Compiler gibt es eine Portierung für das Windows-Betriebssystem, siehe *http://www.cppbuch2.de*. Weitere Hinweise finden Sie ab Seite 368.

Auf einem Mac mit OS X sollten Sie die Entwicklungsumgebung Xcode mit dem Clang-Compiler[7] installieren. Weitere Hinweise finden Sie ab Seite 373.

Für dieses Buch verwende ich nur die genannten Compiler. In einem Windows-Eingabeaufforderungsfenster[8] wird er durch Eingabe des Befehls `g++` mit einigen Parametern aufgerufen. In einer Linux- oder OS X-Konsole rufen Sie den Compiler ebenfalls mit `g++` auf, auch wenn sich im Fall von OS X der Clang-Compiler hinter dem Aufruf verbirgt.

Wenn Sie eine Entwicklungsumgebung benutzen möchten, empfehle ich für Windows und Linux Netbeans (siehe Abschnitte A.1.2 und A.2.2) mit

---

5 American Standard Code for Information Interchange

6 *https://gcc.gnu.org/*

7 *http://clang.llvm.org/get_started.html*

8 Statt des Wortungetüms Eingabeaufforderungsfenster oder des Begriffs Terminal werde ich von nun an das Wort *Konsole* verwenden.

C++-Plug-in bzw. Xcode für OS X (Abschnitt A.3.1). Die entsprechenden Kommandos, um ein Programm zu compilieren und auszuführen, werden in den ersten Kapiteln angegeben – sowie immer, wenn sich etwas grundlegend ändert.

1.4 Die Beispiele

Alle Beispiele des Buchs können Sie aus dem Internet von der Adresse *http://www.cppbuch2.de/* herunterladen, wenn Sie keine Lust zum Abtippen haben. Meiner Erfahrung nach ist Abtippen fehlerträchtig – andererseits lernt man durch eigenes Schreiben. Es ist allemal sinnvoll, die heruntergeladenen Beispiele auszuprobieren, im Editor zu modifizieren (etwa einen Fehler einzubauen) und die Wirkung der Modifikation zu sehen. Außerdem können Sie Teile für eigene Zwecke herauskopieren, wenn Sie wollen.

1.5 Wo gibt es Hilfe?

Bei der Programmentwicklung wird gelegentlich das Problem auftauchen, etwas nachschlagen zu müssen. Es kann auch sein, dass man etwas nicht ganz verstanden hat und nach weiteren Erläuterungen sucht. Es gibt die folgenden Hilfen:

- Es gibt ein recht umfangreiches *Stichwortverzeichnis* ab Seite 393 und ein sehr detailliertes *Inhaltsverzeichnis*.
- Referenzen zur C++-Standardbibliothek finden Sie unter *http://stdcxx.apache.org/doc/stdlibref/* und *http://www.cplusplus.com/reference/*.
- Wenn Sie nicht mehr weiterwissen, kann das Forum *http://www.c-plusplus.de/forum/viewforum-var-f-is-15.html*

recht hilfreich sein. Um eine möglichst gute Hilfestellung zu bekommen, lohnt es sich, die ersten, mit »Wichtig« gekennzeichneten Einträge zu beherzigen.

- Erklärungen zu Begriffen sind im *Glossar* ab Seite 379 aufgeführt.

1.6 Das erste Programm!

Wenn Sie sich die genannten Werkzeuge besorgt haben, kann es losgehen. Am besten legen Sie sich zunächst ein Verzeichnis an, in dem sich Ihre Programme befinden sollen. Der Lehrbuch-Klassiker ist das »Hello world!«-Programm, das nur eben diesen Text anzeigt. Schreiben Sie den folgenden Programmtext (ohne Zeilennummern) und speichern Sie ihn in einem Verzeichnis als Datei *hello.cpp* ab:

Listing 1.1 Das erste Programm *(beispiele/kap1/hello.cpp)*

```
1 #include <iostream>
2
3 int main() {
4   std::cout << "Hello world!\n";
5   return 0;
6 }
```

Die Endung *cpp* des Dateinamens sagt, dass es sich um ein C++-Quellprogramm handelt. Öffnen Sie eine Konsole in dem Verzeichnis und geben Sie ein:

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

`g++` ruft den Compiler auf, der wiederum den Präprozessor und den Linker aufruft. Um die Letzteren müssen Sie sich also nicht kümmern. `-o hello.exe` bedeutet, dass die zu erzeugende ausführbare Datei *hello.exe* heißen soll. `hello.cpp` schließlich ist der Name Ihres soeben geschriebenen Programms. Wenn Sie nun mit `dir` (Windows) oder `ls` (Linux) nachsehen, finden Sie die Datei *hello.exe* im Verzeichnis. Die Abkürzung

```
g++ hello.cpp
```

funktioniert auch, nur dass der Compiler für die ausführbare Datei einen Standardnamen wählt, etwa *a.exe* oder *a.out* je nach System. Tippen Sie nun `hello.exe` ein beziehungsweise `./hello.exe` unter Linux oder OS X, wenn das aktuelle Verzeichnis nicht im Pfad liegt. Das Programm wird ausgeführt und auf dem Bildschirm erscheint die Meldung »Hello world!«. Was bedeuten nun die einzelnen Zeilen des Programms?

1 `#include <iostream>`
Mit einem Programm kann man nicht nur rechnen, sondern auch Ausgaben auf die Konsole veranlassen und Eingaben entgegennehmen. Die Absicht wird dem Compiler in Zeile 1 mitgeteilt. Sie können die Zeile so interpretieren: Der Compiler soll in Ihr Programm alles einschließen (englisch *include*), was er für die Ein- und Ausgabe braucht. `iostream` bedeutet *input output stream*. Mit »stream« (dt. Strom) ist gemeint, dass es einen Strom von Zeichen in das Programm hinein bzw. hinaus gibt. Das Zeichen # bedeutet eine Anweisung an den Präprozessor. Solche Anweisungen heißen *Makro*. Letzlich liest der vom Compiler aufgerufene Präprozessor die Systemdatei *iostream* an dieser Stelle ein, sodass ihr Inhalt Bestandteil Ihres Programms ist. Dateien wie *iostream* werden *Header*-Dateien genannt, weil sie am Anfang eines Programms eingebunden werden (head = engl. für Kopf).

2 Leerzeile. Eine Leerzeile spielt für den Compiler keine Rolle. Leerzeilen werden zur Strukturierung eines Programms benutzt, um die Lesbarkeit zu erhöhen. Dasselbe gilt für Einrückungen. So wird ein *Block*, das ist der zwischen den Klammern `{` (Zeile 3) und `}` (Zeile 6) stehende Text, eingerückt, indem zwei Leerzeichen am Anfang der Zeile eingefügt werden. Diese Leerzeichen werden vom Compiler ignoriert.

3 `int main() {`
Jedes C++-Programm fängt mit `main()`, dem Hauptprogramm, an. `main()` ist eine Funktion, die einen ganzahligen Wert zurückgibt. Die Information, dass der Wert eine ganze Zahl ist, wird dem Compiler mit dem Schlüsselwort `int` (Abk. für Integer) mitgeteilt. `int` wird auch *Typ* oder *Datentyp* der ganzen Zahlen genannt. Ein Typ definiert letztlich, welche Operationen mit einem Wert des Typs

möglich sind. Im Fall `int` sind das Operationen wie addieren, subtrahieren usw.

Funktionen kennen Sie vermutlich aus der Mathematik. So würde der Funktionsaufruf *sin 45°* bedeuten, dass für weitere Berechnungen das Ergebnis der Funktion an die Stelle des Aufrufs tritt. So ist es auch in C++, nur dass die Funktionsparameter in runden Klammern angegeben werden. Wenn keine Parameter zu übergeben sind, bleiben die Klammern leer, so wie hier bei `main()`. Die ganze Zahl, die von `main()` zurückgegeben wird, kann vom Betriebssystem ausgewertet werden.

Am Ende der Zeile steht eine öffnende geschweifte Klammer. Sie kennzeichnet den Beginn eines Blocks, wie bei Zeile 2 beschrieben. Programmcode, der zu einer Funktion gehört, steht stets innerhalb eines Blocks.

4 `std::cout << "Hello world!\n";`

Diese Anweisung bewirkt die Ausgabe auf dem Bildschirm. Sie wird, wie jede einzelne[9] Anweisung, mit einem Semikolon abgeschlossen. `cout` ist ein Objekt, das für die Ausgabe verantwortlich ist. In C++ gibt es sogenannte Namensräume, um Bereiche voneinander abzugrenzen (mehr dazu weiter unten). Der zum C++-Standard gehörende Namensraum hat die Bezeichnung `std`. Weil `cout` zu diesem Namensraum gehört, heißt es `std::cout`. Der Name ist eine Abkürzung für *c*haracter *out*put (dt. Zeichenausgabe). Das Objekt ist in `iostream` definiert (siehe Zeile 1). Mit dem Operator `<<` wird der Text »zur Ausgabe geschoben« (zum Begriff Operator siehe unten). Wie Sie später sehen werden, ist diese Ausdrucksweise mehr bildlich als genau, das reicht aber für den Moment. Der auszugebende Text wird in ASCII-Anführungszeichen eingeschlossen. Die im Deutschen gebräuchliche Unterscheidung zwischen Anführungszeichen oben und Anführungszeichen unten gibt es in einem C++-Programm nicht. Am Ende des Textes sehen Sie `\n`. Diese Zeichenkombination wird wie ein einziges Zeichen interpretiert. Es steht für eine neue Zeile (Zeilenumbruch), sodass nachfolgen-

[9] Es gibt auch Verbundanweisungen, doch dazu später.

de Ausgaben in die nächste Zeile geschrieben werden. Zeichen wie dieses oder auch das Tabulatorzeichen sind nicht druckbare Steuerzeichen, trotzdem möchte man sie in einem Programm ausdrücken. Zur Kennzeichnung wird ein Backslash `\` davorgestellt. Das Tabulatorzeichen wird in einem Programm als `\t` geschrieben.

5 `return 0;`

`main()` gibt die ganze Zahl 0 zurück. Das Semikolon kennzeichnet auch hier das Ende der Anweisung. 0 steht dafür, dass alles in Ordnung ist. Im Falle eines Fehlers könnte eine andere Zahl zurückgegeben und vom Aufrufer ausgewertet werden. Jede Funktion, die einen Wert zurückgibt, muss eine `return`-Anweisung enthalten. `main()` ist die einzige Ausnahme, das heißt, die `return`-Anweisung darf hier fehlen. In diesem Fall wird 0 zurückgegeben.

6 `}`

Hier wird der in Zeile 3 begonnene Block geschlossen: Ende der Funktion.

Der Begriff *Operator* wird in C++ häufig verwendet. Ein Operator bedeutet, dass etwas getan wird, und zwar mit einem oder mehreren *Operanden*. So bezeichnet in der Addition a+b das Zeichen + den Operator. a und b sind die Operanden. Ein Operator wird durch ein Symbol wie + oder << oder durch einen Namen dargestellt, zum Beispiel `plus(a, b)`.

1.6.1 Fehlermeldungen

Nun modifizieren Sie das Programm, indem Sie die ersten beiden Zeilen streichen. Wenn Sie es nun erneut übersetzen, meldet sich der C++-Compiler mit der Fehlermeldung

```
hello.cpp: In Funktion »int main()«:
hello.cpp:2:3: Fehler: »cout« ist kein Element von »std«
   std::cout << "Hello world!\n";
   ^
```

Weil `#include <iostream>` nun fehlt, kennt der Compiler `cout` nicht. Die Fehlermeldung ist allerdings irreführend, und daran müssen Sie sich leider gewöhnen. Denn natürlich ist `cout` Element von `std`, es wird nur nicht bekannt gemacht. `hello.cpp:2:3:` besagt, dass der Compiler den Fehler in der Datei *hello.cpp* bei dem dritten Zeichen der Zeile zwei entdeckt hat. Auch das ist oft irreführend, wenn nämlich angenommen wird, dass das auch die Stelle des Fehlers ist. Tatsächlich bedeutet es, dass der Fehler an dieser Stelle *oder davor* liegt. Sie wissen ja, wie Sie diesen Fehler leicht reparieren können. Experimentieren Sie mit weiteren Veränderungen des Programms und studieren Sie die dann entstehenden Fehlermeldungen.

Weitere Optionen des Compilers

Der Compiler bietet die Möglichkeit, Warnungen auszugeben, wenn Anweisungen verwendet werden, die zwar sprachkonform sind, vermutlich aber einen Fehler enthalten. Die Option zur Anzeige der Warnungen ist `-Wall`. Wie die Option übergeben wird, sehen Sie unten im Kasten. In manchen Fällen bekommen Sie seitenlange Fehlermeldungen, entweder weil es mehrere Fehler gibt oder weil der Compiler nach einem Fehler das restliche Programm nicht mehr versteht. Wenn Sie sich nur auf den ersten angezeigten Fehler konzentrieren, reicht das in aller Regel! Die Fehlerausgabe wird mit `-fmax-errors=1` (Clang: `-ferror-limit=1`) auf diesen Fehler beschränkt.

Tipp: Verwenden Sie die genannten Optionen des Compilers.
Ein Beispiel für Windows und Linux:
`g++ -Wall -fmax-errors=1 -o prog.exe prog.cpp`
(Mac mit OS X: `-ferror-limit=1` statt `-fmax-errors=1`)

`-fmax-errors=1` und `-ferror-limit=1` werden in folgenden Kommandos zur Übersetzung nicht mehr erwähnt.

Vereinfachung der Schreibweise

Wenn klar ist, dass der Namespace `std` benutzt wird, wird abgekürzt statt `std::cout` einfach nur `cout` geschrieben, wenn vorher `using namespace std;` angegeben wird, wie Sie im nächsten Programm sehen. In C++-Dateien mit der Endung *.cpp* kann `using namespace std;` in der Regel problemlos verwendet werden. Wenn nur ein Teil des Namespace `std` benutzt wird, genügt es, nur diesen Teil anzugeben, also etwa `using std::cout;`. Die Verwendung von `using` erspart nachfolgende Schreibarbeit.

1.7 Eingaben und Kommentare

Im zweiten Programm lernen Sie die Eingabe von Zahlen, das Durchführen einer Berechnung und den Einsatz von Kommentaren kennen. Wie vorher wird erst das Beispiel gezeigt. Die einzelnen Zeilen werden anschließend erläutert:

Listing 1.2 Eingabe und Kommentare *(beispiele/kap1/summe.cpp)*

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

int main() {
  /* Berechnung einer Summe. Dazu werden zwei ganze
     Zahlen eingelesen, die dann addiert werden.
  */
  int summand1;    // Der erste Summand.
  int summand2;    // Der zweite Summand.
  cout << "Bitte zwei ganze Zahlen eingeben, "
          "getrennt durch Leerzeichen: ";
  cin >> summand1 >> summand2;
  int ergebnis = summand1 + summand2;
  cout << summand1 << " + " << summand2 << " ergibt "
       << ergebnis << '\n';
}
```

Übersetzen Sie die Datei mit `g++ -o summe.exe summe.cpp`. Sie können das Programm durch Eingabe von `summe.exe` ausführen.

Falls Sie den empfohlenen Compiler installiert und eine Konsole in einem Verzeichnis der heruntergeladenen Dateien geöffnet haben, können Sie einfach *make* eingeben und alle Programme in diesem Verzeichnis werden übersetzt. *make* bewerkstelligt dies mit Hilfe einer Steuerdatei namens *makefile*. Mehr dazu finden Sie in [Br]. ■

5-7 Zwischen den Zeichenkombinationen `/*` und `*/` können Sie einen beschreibenden Text einfügen, Kommentar genannt, der über mehrere Zeilen gehen darf. Solche Kommentare werden vom Präprozessor herausgefiltert, sodass der Compiler sie gar nicht erst zu sehen bekommt.

8-9 Hier werden die Variablen `summand1` und `summand2` *deklariert*, Deklarieren heißt, einen Namen in das Programm einführen, also dem Compiler bekannt machen. Beide Variablen sind vom Typ `int`, repräsentieren also eine ganze Zahl. Auch Deklarationen schließen mit einem Semikolon ab. Ab dieser Stelle ist der Name dem Compiler bekannt. Der mit `//` eingeleitete Teil danach ist ebenfalls ein Kommentar. Der Unterschied zu dem oben beschriebenen ist, dass er nur bis zum Ende der Zeile reicht.

10-11 Eine einfache Ausgabeanweisung. Lange Zeichenketten können mit Hilfe von Anführungszeichen aufgespalten werden. Die Wirkung ist, als ob der ganze auszugebende Text in einer Zeile stände. Die Aufspaltung ist ein Hilfsmittel zur besseren Lesbarkeit des Programmtextes.

12 Dies ist die Eingabeanweisung. Beachten Sie, dass der Operator jetzt in Richtung der einzulesenden Variablen zeigt (`>>`)! `cin` (für *character input*) ist das für die Eingabe zuständige Objekt. Wie Sie sehen, kann die Eingabe hintereinander geschaltet werden. Die Zeile ist äquivalent zu den zwei Zeilen

```
cin >> summand1;
cin >> summand2;
```

Bei der Eingabe müssen die Werte durch ein oder mehrere Zwischenraumzeichen (englisch *whitespace*) getrennt sein. Dass kann ein Leerzeichen, ein Tabulatorzeichen oder auch eine neue Zeile sein. Probieren Sie es aus!

13 Hier wird die Variable `ergebnis` angelegt und mit dem Ergebnis der Berechnung *initialisiert*, also mit einem Anfangswert versehen. Siehe dazu die Textbox unten.

14-15 Es können mehrere Objekte nacheinander ausgegeben werden, wie man sieht. Neu ist, dass `\n` in *einzelne* Hochkommas eingeschlossen wird. Der Unterschied: Mit `""` werden Zeichenketten begrenzt, die aus einem oder mehr als einem Zeichen bestehen. `''` begrenzen hingegen nur einzelne Zeichen. Wie oben schon gesagt, wird `\n` als ein einziges (Steuer-)Zeichen interpretiert.

16 Hier wird der in Zeile 3 begonnene Block geschlossen: Ende der Funktion, diesmal ohne `return` (in `main()` darf `return` fehlen).

Wenn ein Objekt mit = schon bei der Anlage einen Wert erhält, spricht man von *Initialisierung*. Existiert das Objekt schon vorher, bewirkt = eine *Zuweisung*.

```
int a = 3;          // Initialisierung mit 3
int b;              // keine Initialisierung: undefinierter Wert
                    // (muss nicht 0 sein!)
b = 7;              // Zuweisung von 7 (b existiert schon vorher)
```

Wenn nun keine Zahlen, sondern zum Beispiel Buchstaben eingegeben werden, verhält sich das Programm nicht korrekt. Wie damit umzugehen ist, erfahren Sie schon im nächsten Kapitel.

1.7.1 Kommentare und Namen

Kommentare als Erläuterungen zu einem ansonsten nur schwer zu verstehenden Programmcode sind besonders sinnvoll.

Namen wie `summand1` usw. sind frei wählbar, sie dürfen aber nur aus kleinen oder großen Buchstaben, Ziffern und dem Unterstrich _ bestehen und sie dürfen keine Leerzeichen enthalten. Namen müssen mit einem (großen oder kleinen) Buchstaben beginnen. Der Unterstrich am Anfang ist auch möglich, wird aber nicht empfohlen, weil manche Systemvariablen mit ihm anfangen. Vermeiden Sie überflüssige Kommentare, etwa

```
a = a + 1;                  // a wird um 1 vergrößert
```

Der Kommentar enthält keine zusätzliche Information und ist daher überflüssig. Wenn Sie Kommentare durch »sprechende« Namen einsparen können, tun Sie es! Beispiel:

```
int i;                      // Verschiebung der Grafik (in Pixeln)
int verschiebung;
```

Der Vorteil der Benennung in der zweiten Zeile im Vergleich zur ersten ist, dass die Bedeutung auch etliche Zeilen weiter im Programm noch klar erkennbar ist, ohne sie durch Zurückblättern des Bildschirms suchen zu müssen, wenn Sie zum Beispiel den Programmtext nach längerer Zeit wieder lesen wollen. Auch anderen, die den Code lesen, helfen gut gewählte Variablennamen, ihn zu verstehen.

1.8 Zahltypen und -bereiche

1.8.1 Ganze Zahlen

Eine Zahl beansprucht Platz im Speicher. Die tatsächlich verwendete Anzahl von Bytes variiert je nach Rechnersystem. Typische Werte sind:

`short` (oder `short int`)	2 Bytes
`int`	4 Bytes
`long` (oder `long int`)	4 oder 8 Bytes
`long long` (oder `long long int`)	mindestens 8 Bytes

`00000000000000000000000000000101` ist zum Beispiel die interne Darstellung der Zahl 5 mit 32 Bits (= 4 Bytes). Ganz rechts sind die Bits mit

der geringsten Bedeutung. `101` (binär) entspricht $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, also 5 (dezimal). Ein Bit wird für das Vorzeichen reserviert. Sie können die in Ihrem System verwendete Anzahl von Bytes mit `sizeof` feststellen. Bei wohl den meisten Computern führt die folgende Anweisung zur Ausgabe »4«:

```
cout << sizeof(int) << '\n';
```

Der Zahlenbereich hängt von der Anzahl der Bytes ab. Bei einem `int`-Wert mit 4 Bytes = 32 Bits geht er von -2^{31} bis $2^{31}-1$, das ist der Bereich von −2.147.483.648 bis 2.147.483.647. Es gibt zu jedem der genannten Zahltypen auch eine Variante ohne Vorzeichen, die mit dem Wort `unsigned` gekennzeichnet wird. Der kleinste Wert so einer Zahl ist 0. Jeder andere Wert ist positiv. Es sind genau so viele Zahlen darstellbar. Der Zahlenbereich ist nur in Richtung der positiven Zahlen verschoben. So ist der größtmögliche Wert einer `unsigned int`-Zahl, die 4 Bytes belegt, 4.294.967.295.

size_t – ein besonderer Zahltyp

`size_t` ist ein Typ für nichtnegative (also 0 oder positive) ganze Zahlen wie `unsigned int`. Der Unterschied ist, dass `size_t` groß genug ist, um die Größe des größtmöglichen Objekts des Systems aufzunehmen. Auf 32-Bit-Systemen gibt es oft keinen Unterschied zu `unsigned int`, aber auf 64-Bit-Systemen kann `sizeof(unsigned int)` den Wert 4 ergeben und `sizeof(size_t)` den Wert 8. Deswegen ist es im Allgemeinen besser, `size_t` statt `unsigned int` zu verwenden, wenn es um Größeninformationen oder Adressen im Speicher geht. Generell schadet es nicht, wenn Sie `size_t` immer dann verwenden, wenn es um nichtnegative ganze Zahlen geht. Ein Beispiel:

```
std::size_t anzahl = 0;    // zu std:: siehe unten
```

`size_t` ist im Namespace `std`. Wenn Sie also nicht vorher im Programm `using namespace std;` stehen haben, schreiben Sie `std::size_t`. Genau so, wie Sie `#include <iostream>` in einem Programm mit Ausgaben auf dem Bildschirm schreiben, damit der Compiler `cout` kennt (siehe das

erste Programm auf Seite 21), fügen Sie oben im Programm `#include <cstddef>` ein, wenn Sie `std::size_t` bzw. `size_t` verwenden. Es kann sein, dass das Programm auch ohne diese Include-Anweisung compiliert wird. Der Grund ist, dass eine andere Include-Anweisung (etwa die für `<iostream>`) schon für das Einschließen von `<cstddef>` gesorgt hat. Guter Programmierstil ist es jedoch, sich nicht darauf zu verlassen.

Rechnen mit ganzen Zahlen

Für das Rechnen mit ganzen Zahlen gibt es die üblichen Operatoren für die Grundrechenarten, wie etwa + für die Addition. Es gibt aber auch Operatoren, die die Rechenoperation und die Zuweisung in sich vereinen (Kurzformoperatoren). Das folgende Listing zeigt Beispiele:

Listing 1.3 Rechenoperationen für ganze Zahlen

```
int a = 4;
int b = 9;
int c = a + b;      // Addition. c ist 13.
c += 7;             // Kurzform, dasselbe wie c = c + 7;
c = a - b;          // Subtraktion
c -= a;             // Kurzform, dasselbe wie c = c - a;
c = a * 3;          // Multiplikation
c *= 10;            // Kurzform, dasselbe wie c = c * 10;
c = 9 / 5;          // Division. Ergebnis: 1
                    // (ganze Zahl, Rest entfällt!)
c /= 2;             // Kurzform, dasselbe wie c = c / 2;
c = 9 % 5;          // Restbildung (Modulo-Operator). Ergebnis: 4
c %= 2;             // Kurzform, dasselbe wie c = c % 2;
```

Darüber hinaus gibt es noch den Inkrement-Operator ++ und den Dekrement-Operator --. Der Erste erhöht einen Wert um 1, der Zweite erniedrigt einen Wert um 1. Diese Operatoren haben die Eigenschaft, dass sie *vor* oder *nach* einer Variablen stehen können. Dabei ist das Verhalten unterschiedlich, wenn die Variable in derselben Anweisung benutzt wird, wie das Listing zeigt. Für den Dekrement-Operator -- gilt Entsprechendes. Meine Empfehlung: Programmieren Sie selbst ein Beispiel, in dem ++ und -- mal vor und mal nach einer Variablen stehen, mal mit

Benutzung durch Zuweisung an eine andere Variable und mal alleinstehend. Dann werden Sie den Unterschied gut nachvollziehen können. Das nachfolgende Listing zeigt ein paar Beispiele:

Listing 1.4 Inkrementieren (hochzählen) mit ++

```
int a = 10;
int b;
// Vorangestellt: Erst inkrementieren, dann benutzen (zuweisen):
b = ++a;                    // a ist 11, b ist 11.
// Nachgestellt: Erst benutzen, dann inkrementieren:
int c = b++;                // c ist 11, b ist 12.
c++;          // nur Inkrementierung, keine Benutzung: c ist 12.
++c;          // nur Inkrementierung, keine Benutzung: c ist nun 13.
```

Zahlendarstellung

Ganze Zahlen können auf mehrere Arten dargestellt werden. Die folgende Übersicht hilft, Zahlen in einem Programm richtig zu schreiben.

- Die übliche Darstellung ist die als Dezimalzahl. Eine Dezimalzahl ist entweder 0 oder sie beginnt mit einer der Ziffern im Bereich 1 bis 9. Ein Suffix (l, ll, L, LL) kennzeichnet `long`- oder `long long`-Zahlen, zum Beispiel 2147483647L oder 9223372036854775806LL. Ein mögliches Vorzeichen gehört nicht zu der Zahl, obwohl es in einem Programm oder beim Einlesen mit `cin` berücksichtigt wird.
- Ein Suffix u oder U kennzeichnet `unsigned`-Zahlen, zum Beispiel 1836u. Kombinationen wie in 2036854775806ul sind möglich.
- Wenn eine Zahl mit 0b oder 0B beginnt, wird sie als *Binärzahl* interpretiert, etwa $0b1011 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$ dezimal.
- Wenn eine Zahl mit einer 0 beginnt, wird sie als *Oktalzahl* interpretiert, zum Beispiel $0377 = 3 \cdot 8^2 + 7 \cdot 8^1 + 7 \cdot 8^0 = 255$ dezimal.
- Wenn eine Zahl mit 0x oder 0X beginnt, wird sie als *Hexadezimalzahl* interpretiert, zum Beispiel 0xAFFE (= 45054 dezimal). Um Hexadezimalzahlen darstellen zu können, werden A, B, C, D, E und F als Ziffern hinzugenommen, wobei die Kleinschreibung ebenfalls möglich ist.

- Bei allen Zahlen ist es seit C++14 zur besseren Lesbarkeit erlaubt, zur Gruppierung Hochkommas einzufügen. So kann man 1'000'000 statt 1000000 schreiben oder 0xabc'def statt 0xabcdef. Bei der Auswertung der Zahl durch den Compiler werden die Hochkommas ignoriert.

1.8.2 Kommazahlen

Natürlich gibt es in C++ auch Kommazahlen. Allerdings gilt in einem Programm die anglo-amerikanische Schreibweise! Das heißt, statt des Kommas wird ein *Punkt* geschrieben. Die Zahl 1,23 muss in einem Programm als 1.23 geschrieben werden. Es gibt drei Typen für Kommazahlen: `float`, `double` und `long double`. Der erste benötigt am wenigsten Platz im Speicher, ist aber nicht so genau wie die anderen. Wie oben gezeigt, können Sie den Speicherbedarf in Bytes mit `sizeof` feststellen. Das obige Beispiel zur Summenberechnung wird im Folgenden leicht abgewandelt, damit Kommazahlen berechnet werden können.

Listing 1.5 Kommazahlen *(beispiele/kap1/summedouble.cpp)*

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

int main() {
  /* Berechnung einer Summe mit Kommazahlen.
     Dazu werden zwei Zahlen eingelesen, die dann addiert werden. */
  double summand1;    // Der erste Summand.
  double summand2;    // Der zweite Summand.
  cout << "Bitte zwei Zahlen eingeben: ";
  cin >> summand1 >> summand2;
  double ergebnis = summand1 + summand2;
  cout << summand1 << " + " << summand2 << " ergibt "
       << ergebnis << '\n';
  cout << "Die Genauigkeit von double beträgt etwa "
       << numeric_limits<double>::digits10
       << " Dezimalstellen\n";
}
```

Mit `g++ -Wall -o summedouble.exe summedouble.cpp` wird das Programm übersetzt, mit `summedouble.exe` wird es ausgeführt. Die letzte Anweisung gibt die Genauigkeit von `double`-Kommazahlen aus. Die Zeile am Anfang der Datei (`#include <limits>`) ist dazu notwendig. Es gibt verschiedene Schreibweisen für Kommazahlen, gezeigt an Beispielen. Beachten Sie dabei die Suffixe `F` und `L` (kann auch `f` bzw. `l` sein).

- `double`-Zahl `d1`, `double`-Wert 3.14159265:
  `double d1 = 3.14159265;`
  Kommazahlen ohne Suffix sind `double`.
  `4.` ist dasselbe wie `4.0`, ein Punkt genügt zur Kennzeichnung als Kommazahl.
- `f1` wird mit einem `double`-Wert initialisiert:
  `float f1 = 3.14159265;`
  Dies führt zu Genauigkeitsverlust, weil `f1` (`float`) weniger Stellen hat als eine `double`-Zahl.
- Initialisierung mit `float`-Wert (Suffix `f`):
  `float f2 = 3.14159f;`
- Größte Genauigkeit (Suffix `L`):
  `long double d2 = 3.14159265358979323846264338328L;`
- Exponenten-Schreibweise:
  `double d3 = 1.23e-3;`
  `e`X steht für 10^X. `1.23e-3` bedeutet also $1.23 \cdot 10^{-3} = 0.00123$

Mit Kommazahlen können Sie mit Ausnahme der Restbildung genauso rechnen wie mit ganzen Zahlen.

1.8.3 Bit-Operatoren

C++ bietet die Möglichkeit, einzelne Bits einer Zahl abzufragen oder zu ändern. Dafür gibt es die Bit-Operatoren. So sorgt etwa der &-Operator dafür, dass ein Bit im Ergebnis genau dann gesetzt wird, wenn beide entsprechenden Bits der zwei Operanden gesetzt sind. Ein Beispiel:

```
00000000000000000000000000000101 &
00000000000000000000000000000110 ergibt
00000000000000000000000000000100
```

Außer dem &-Operator gibt es noch einige andere, wie die Tabelle 1.1 zeigt. Bit-Operatoren werden zum Beispiel benötigt, wenn der Farbwert eines Pixels in seine Komponenten rot, gelb und grün zerlegt werden soll. Für weitere Einzelheiten zu Bit-Operationen verweise ich auf [Br].

Tabelle 1.1 Bit-Operatoren

Operator	Beispiel	Bedeutung
`<<`	`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 (aus 0 wird 1 und umgekehrt)
`<<=`	`i <<= 3`	`i = i << 3`
`>>=`	`i >>= 3`	`i = i >> 3`
`&=`	`i &= 3`	`i = i & 3`
`|=`	`i |= 3`	`i = i | 3`

1.8.4 Vorsicht Falle: Umwandlung des Zahltyps

Bitte beachten Sie, dass das Ergebnis der Rechenoperation vom Zahlentyp abhängt:

```
double x = 3 / 4;          // x ist 0.0 (siehe Text)
double y = 3 / 4.;         // y ist 0.75
```

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

Die Vermischung von `unsigned int` oder `size_t` und `int` kann zu Fehlern führen, weil der Compiler bei der automatischen Typumwandlung einfach nur die Bitmuster übernimmt. So wird -1 typischerweise intern als eine Folge von Einser-Bits repräsentiert. Bei einem Zahltyp ohne Vorzeichen ist das aber gerade die größtmögliche Zahl. Das folgende Programm zeigt den Effekt:

Listing 1.6 Typumwandlung *(beispiele/kap1/typumwandlung.cpp)*

```
#include <cstddef>
#include <iostream>
using std::cout;   // um unten nur cout schreiben zu können
                   // siehe Ende von Abschnitt 1.6
int main() {
  int intZahl = -1;
  cout << "-1 als int-Zahl = " << intZahl << '\n';
  unsigned int uintZahl = -1;          // Typumwandlung!
  cout << "-1 als unsigned int-Zahl = " << uintZahl
       << '\n';
  std::size_t szZahl = -1;             // Typumwandlung!
  cout << "-1 als size_t-Zahl = " << szZahl << '\n';
}
```

Auf meinem 32-Bit-System gibt das Programm aus:

```
-1 als int-Zahl = -1
-1 als unsigned int-Zahl = 4294967295
-1 als size_t-Zahl = 4294967295
```

Auf meinem anderen, dem 64-Bit-System, gibt das Programm aus:

```
-1 als int-Zahl = -1
-1 als unsigned int-Zahl = 4294967295
-1 als size_t-Zahl = 18446744073709551615
```

Mischung von `int` und `unsigned int` (oder `size_t`) vermeiden!

Auch wenn die Zahltypen unterschiedlichen Speicherplatz einnehmen, kann die Vermischung problematisch sein. Der Grund: Bei der Typum-

wandlung werden einfach die überzähligen Bits abgeschnitten. Die folgenden Zahlen zeigen die 8-Byte-`long`-Zahl 4294967364 in binärer Darstellung und die entstehende 4-Byte-`int`-Zahl in binärer Darstellung nach Abschneiden der höchstwertigen 32 Bits:

```
0000000000000000000000000000000100000000000000000000000001000100
                                00000000000000000000000001000100
```

So wird aus der `long`-Zahl 4294967364 die `int`-Zahl 68. Unser Ziel ist natürlich, Programme zu schreiben, die auf 32-Bit-Systemen *und* 64-Bit-Systemen laufen!

Listing 1.7 Auszug aus *beispiele/kap1/typumwandlung2.cpp*

```
long longzahl = 4294967364L;                  // Suffix L für long
int z =  longzahl;
cout << "4294967364L als int: " << z << '\n';         // 68
```

Auf einem 32-Bit-System, auf dem `sizeof(long)` nur 4 (Bytes) ergibt, liefert schon die erste Zeile eine Warnung des Compilers, weil `long` nur 4 Bytes hat und 4294967364 nicht mit 4 Bytes darstellbar ist.

Mischung von Zahltypen verschiedener Bitbreite vermeiden!

Das gilt auch für Kommazahlen. So ist zwar 0.5 ($= 2^{-1} = 1/2$) im Binärsystem, das vom Computer verwendet wird, exakt darstellbar. Für viele andere Zahlen gilt das aber nicht. »Binär exakt darstellbar« bedeutet für eine Nachkommazahl, dass sie als endliche Summe von Zweierpotenz-Brüchen dargestellt werden kann. Ein Beispiel:

$0.8125 = 1 \cdot 1/2 + 1 \cdot 1/2^2 + 0 \cdot 1/2^3 + 1 \cdot 1/2^4$, das heißt
$0.8125 = 1 \cdot 1/2 + 1 \cdot 1/4 + 0 \cdot 1/8 + 1 \cdot 1/16$ oder
$0.8125 = 1 \cdot 0.5 + 1 \cdot 0.25 + 0 \cdot 0.125 + 1 \cdot 0.0625$

Die Nullen und Einsen entsprechen der binären Darstellung im Rechner. Vor den folgenden Subtraktionen wird die `float`-Zahl (Suffix `f`) in eine `double`-Zahl umgewandelt. Die Differenz zweier auf den ersten Anschein gleicher Zahlen ist nicht unbedingt 0, wie die folgenden Zeilen zeigen:

Listing 1.8 Auszug aus *beispiele/kap1/typumwandlung2.cpp*

```
// 0.5 ist binär exakt darstellbar
cout << "0.5f - 0.5 =" << 0.5f - 0.5 << '\n';     // exakt 0
// 10.4 ist binär nicht exakt darstellbar
cout << "10.4f - 10.4 =" << 10.4f - 10.4 << '\n';   // ≠ 0!
```

1.9 Zeichen und Zeichenketten

Oben haben Sie schon das (Steuer-)Zeichen `\n` kennengelernt. Es steht für die Ausgabe einer neuen Zeile. Natürlich können Sie auch alle anderen Zeichen in einem Programm verwenden, wobei sich dieses Buch auf ASCII-Zeichen (andere bekannte Zeichensätze sind ISO-8859-1 und UTF-8) beschränkt. Eine Zeichenkette, auch String genannt, besteht aus einer Folge von Zeichen.

In einem Programm wird ein einzelnes Zeichen (das also nicht Teil einer Zeichenkette ist) mit einfachen Anführungszeichen eingeschlossen. Eine Zeichenkette hingegen wird in einem Programm mit doppelten Anführungszeichen begrenzt.

Der folgende Programmausschnitt gibt *Hello!A* aus:

```
cout << "Hello!";                    // Zeichenkette
cout << 'A';                         // Zeichen A
```

Die Zeichenkette und das Zeichen stehen fest im Programm, sie heißen *Literale*. Sie werden zur *Compilierzeit* ausgewertet, also während der Compiler den Programmtext analysiert, und direkt in das erzeugte lauffähige Programm eingetragen. Demgegenüber ist die *Laufzeit* die Zeitdauer ab Start des Programms bis zu seinem Ende. Weil fest im Programm stehend, können Literale während der Laufzeit des Programms nicht geändert werden. Das gilt natürlich auch für andere Literale wie etwa direkt in das Programm geschriebene Zahlen. In der Anweisung

```
x = 42 * y;
```

ist die Zahl 42 ein Literal. Eine Änderung der Werte ist dann möglich, wenn sie in Variablen (dt. Veränderliche) gespeichert werden. Für Zeichen (englisch *character*) ist der Datentyp `char`, für Strings (Zeichenketten) ist er `string`.

Ein Datentyp definiert den Wertebereich und die Operationen, die mit Objekten dieses Datentyps möglich sind. ■

Der Wertebereich von Variablen des Typs `char` sind alle Zeichen, die mit einem Byte darstellbar sind. Die Buchstaben 'A' bis 'Z' sind eine Untermenge davon. Eine der möglichen Operationen ist die Zuweisung mit =, siehe Programm unten. Die Operation + bedeutet beim Datentyp `int` eine Addition, während sie beim Datentyp `string` das Aneinanderhängen zweier Strings meint. Der Datentyp `string` gehört zum Namespace `std` und ist im Header `<string>` definiert. Man schreibt daher `std::string` oder fügt vorher im Programm `using namespace std;` ein. Ein Beispiel für die Angabe des Namenraums beim Datentyp `string`:

Listing 1.9 Zeichen und String *(beispiele/kap1/zeichen.cpp)*

```
#include <iostream>
#include <string>                // neuer Header

int main() {
  char zeich = 'x';              // Zeichen zeich mit dem Wert x
  std::string text {"Hello"};    // String text, Wert: Hello
  // Ausgabe, gefolgt von \n für eine neue Zeile
  std::cout << text << "   " << zeich << '\n';    // Hello x
  // e ist das Zeichen Nr. 1 (Zählung ab 0!):
  std::cout << text.at(1) << '\n';
  zeich = 'a';                   // Wert von zeich ändern
  text.at(1) = zeich;      // Wert von text an der Stelle 1 ändern
  text.at(1) = 'a';              // hat hier dieselbe Wirkung
  std::cout << text << '\n';     // jetzt Hallo statt Hello
  text = "Guten Morgen!";        // ganz neuer Inhalt
```