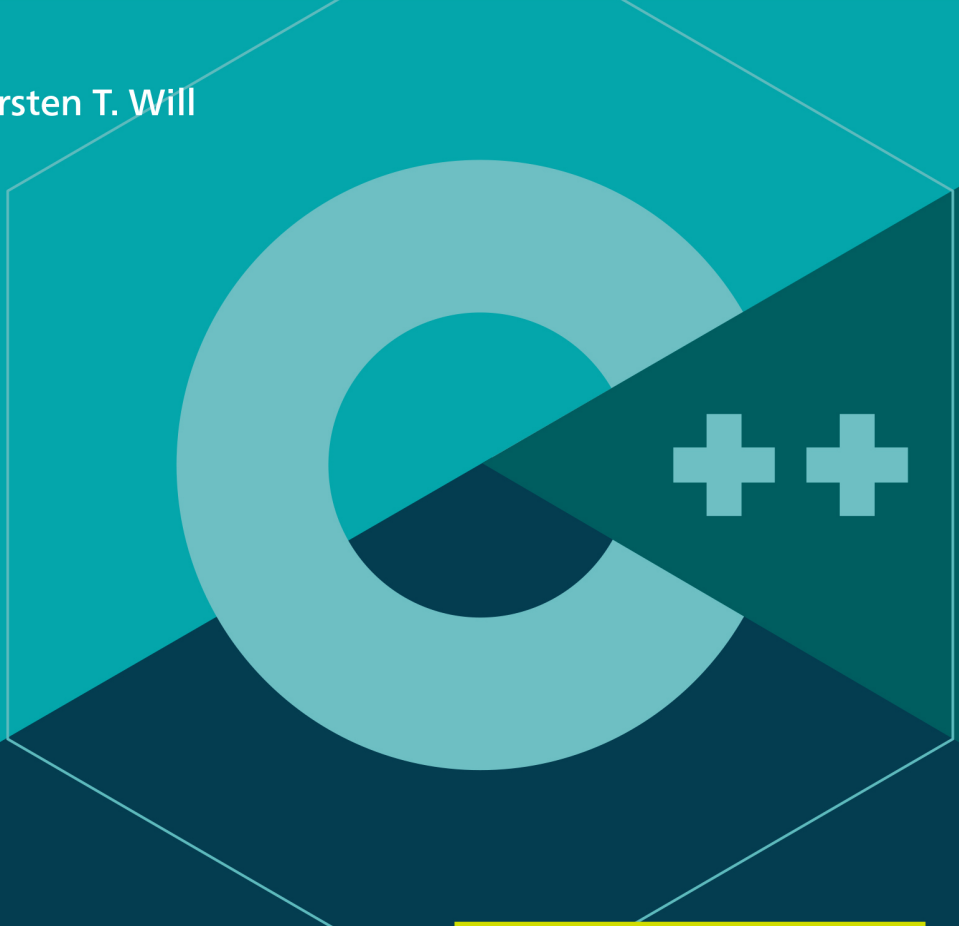
Torsten T. Will



C++

Das umfassende Handbuch

Aktuell zu C++20

- Das Lehr- und Nachschlagewerk zu Modern C++
- C++ Core Guidelines und Techniken für guten Code
- Sprachgrundlagen, OOP, Standardbibliothek, GUI-Programmierung mit Qt u. v. m.

Alle Beispielprojekte zum Download

Rheinwerk Computing

Liebe Leserin, lieber Leser,

mit dem Standard C++11 hat die Programmiersprache einen großen Schritt in die Zukunft gemacht. C++ seit diesem Jahrzehnt eine moderne Programmiersprache, die ihre Stärken hinsichtlich nachhaltigem und effizientem Code voll ausspielt.

Mit den jüngeren Features hat sich auch eine neue Art, in C++ zu programmieren, etabliert. Das Schlagwort Modern C++ ist auch das Motto dieses Buches. Torsten T. Will, ein ausgewiesener C++-Experte, möchte Ihnen zeigen, wie Sie C++ voll ausschöpfen und von vornherein guten Code schreiben. Dazu hat er an den passenden Stellen eigene Kapitel eingefügt, in denen er Ihnen Richtlinien, Techniken und Tipps dafür an die Hand gibt.

Aber auch die Grundlagen kommen nicht zu kurz: Dieses Werk geht ausführlich auf den Sprachkern und die objektorientierte Programmierung in C++ ein. In dem Teil über die Standardbibliothek erfahren Sie, welche Bordmittel C++ mitbringt und wie Sie diese nutzen, um effizient zu programmieren. Abgerundet wird das Buch durch ein eigenes Kapitel zur GUI-Entwicklung mit Qt.

Natürlich bleibt das alles nicht graue Theorie: Zahlreiche kleinere und umfangreichere Beispielprojekte geben Ihnen konkrete Vorschläge für die Umsetzung in eigene Projekte. Den Code können Sie von der Seite zum Buch unter *www.rheinwerk-verlag.de/5093* als .zip-Datei herunterladen. Für die abgedruckten Listings können Sie den Links zum Live-Code bei *http://godbolt.org* folgen.

Sollten Sie nach der Lektüre des Buches noch Fragen haben, Lob oder Kritik äußern wollen, dann wenden Sie sich an mich. Ich freue mich über Feedback.

Ihre Almut Poll
Lektorat Rheinwerk Computing

almut.poll@rheinwerk-verlag.de
www.rheinwerk-verlag.de
Rheinwerk Verlag · Rheinwerkallee 4 · 53227 Bonn

Auf einen Blick

Impressum

Dieses E-Book ist ein Verlagsprodukt, an dem viele mitgewirkt haben, insbesondere:

Lektorat Almut Poll
Korrektorat Sibylle Feldmann, Düsseldorf
Herstellung E-Book Norbert Englert
Covergestaltung Mai Loan Nguyen Duy
Satz E-Book Torsten T. Will

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.

ISBN 978-3-8362-7595-8

2., aktualisierte Auflage 2020

www.rheinwerk-verlag.de

Inhalt

TEIL III Fortgeschrittene Themen

28 Standardbibliothek – Extras 855

31 GUI-Programmierung mit Qt 1003

Vorwort

Wir schreiben das Jahr 2020. Der mit C++11 eingeführte »moderne C++-Stil« hat sich durchgesetzt, und auf breiter Front wird C++17 genutzt. Dementsprechend finden Sie in diesem Buch `string_view`, `if constexpr`, Erweiterungen bei den Standardcontainern und Algorithmen und vieles mehr. Insbesondere die Beispiele sind, sofern sinnvoll, in der neuen Template-Parameter-Deduktion gehalten: `vector{1,2,3}` statt `vector<int>{1,2,3}`. Auch das strukturierte Binden sehen Sie ab und zu, also das Deklarieren mehrerer Variablen anhand eines Initialisierungsausdrucks wie `auto [a,b] = pair{1,2}`.

Die aktuellste Version der Sprache heißt C++20, ist aber – Stand Februar – noch nicht abgesegnet. Im vorliegenden Buch präsentiere ich Ihnen Features, die mit nahezu an Sicherheit grenzender Wahrscheinlichkeit Teil des offiziellen Standards sein werden. Ich bespreche Module ebenso wie sofortige Funktionen und den `<=>`-Operator. Auch einige Änderungen in der Standardbibliothek erwähne ich. Features, die sich auf C++20 beziehen, hebe ich aber als solche hervor, entweder mit einem Balken und »C++20« im Titel oder `unterpunktet` im Code.

Auf einige Dinge von C++20 gehe ich jedoch bewusst nicht ein: Module und Coroutinen bespreche ich nur oberflächlich. Und die lang erwarteten Concepts als Metasprache über Templates bespreche ich gar nicht. Das hat mehrere Gründe: Erstens sind diese Features so umwälzend und zugleich so jung, dass noch niemand eine definitive »Best Practice« dafür bereitstellen kann. Vieles wird sich erst in den nächsten Jahren aus der täglichen Arbeit der Entwickler mit diesen Features ergeben. Zweitens sind sie groß und komplex. Sie würden einfach den Rahmen dieses Buchs sprengen. Ich würde Ihnen vielleicht vieles präsentieren, was sich letztendlich in der Praxis als zu selten eingesetzt herausgestellt, und hätte Mühe, Papier und Ihre Geduld verschwendet. Und drittens ist es in Moment unmöglich, einen Compiler zu finden, der zuverlässig alle C++-20-Features schon unterstützt. Einzelne ja, mal so, mal der, aber alle gemeinsam nicht. Ich verspreche aber, mich der Sache in einer kommenden Auflage noch einmal anzunehmen – dann, wenn auch die Überarbeitung des Qt-Kapitels ansteht.

Bleibt noch, mich für das viele positive Feedback zu bedanken, das ich erhalten habe, und ebenso für die konstruktive Kritik. Ein Punkt war, dass die Listings der 1. Auflage zwar als Download verfügbar waren, sich aber aus mehrerlei Gründen schwer darin zurechtzufinden sei. Und weil wir im Jahr 2020 angekommen sind – Bogen zum einleitenden Satz geschlossen –, sind nun die Listings einzeln und interaktiv für Sie zum Ausprobieren online. Über den meisten Listings finden Sie einen Link zum *Compiler-Explorer* von Matt Godbolt, *https://godbolt.org/z/...*, der Sie direkt zu dem entsprechenden Listing bringt.

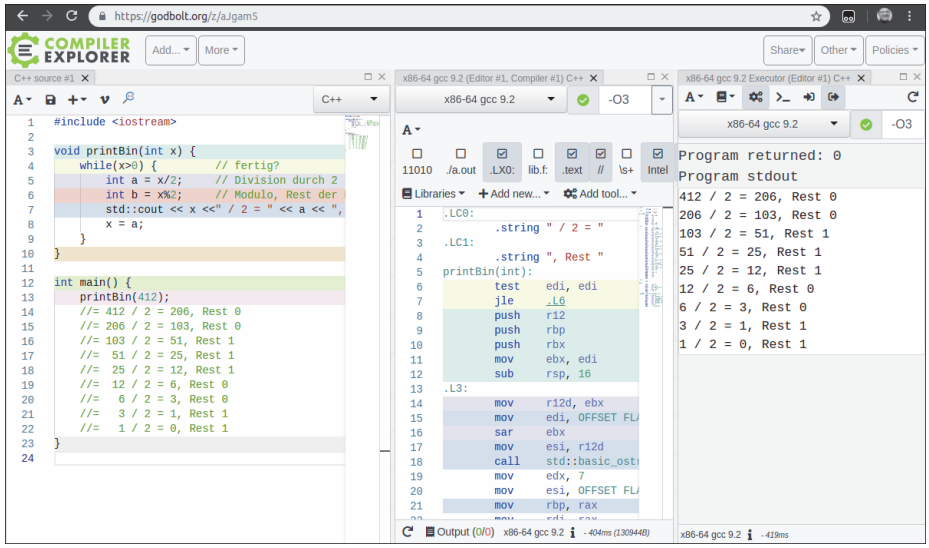


Abbildung 1 Der Compiler-Explorer lädt zum Experimentieren ein.

Damit nicht genug: Das Listing wurde dort schon kompiliert und ausgeführt. Sie sehen die Compilermeldungen und die Programmausgaben. But wait, there is more: Sie können direkt im Browser mit dem Quellcode herumspielen und sogar zwischen verschiedenen Compilern umschalten. Einen herzlichen Dank an Matt Godbolt, der mir freundlicherweise erlaubt hat, seinen exzellenten Service für dieses Buch zu nutzen.

Nun wünsche ich Ihnen viel Erfolg bei den Studien des aktuellen C++.

Mai 2020, Bielefeld

Torsten T. Will

Vorwort zur 1. Auflage

C++ ist eine moderne Programmiersprache. Wenn man sie richtig benutzt. Dieses Handbuch soll Ihnen dabei helfen, in C++ so zu programmieren, dass Sie von dem Programm auch in vielen Jahren noch etwas haben – dass Ihr Programm *nachhaltig* ist.

Aber selbstverständlich spricht nichts dagegen, wenn Sie Ihr Programm »traditionell« schreiben. Das heißt, kurz zusammengefasst, für mich, dass Ihr Programm mehr nach C aussieht, als es aussehen könnte. Daran ist nichts falsch, natürlich nicht. Einige der besten Programme sind in C geschrieben. Dennoch, wenn Sie *heute* ein Projekt beginnen und sich für eine in Maschinencode übersetzte Programmiersprache entscheiden, dann nehmen Sie doch besser C++. Denn in der Sprache tut sich etwas – oder besser, hat sich was getan. Sie haben mit C++14 (und ganz frisch C++17) eine Sprache, die Sie auf aktuelle Art und Weise darin unterstützt, *gute* Programme zu schreiben. Das heißt, Ihre Programme sind schnell, fehlerresistent, wartbar. Sie können produktiv programmieren.

Für dieses Buch habe ich lange überlegt, wie man C++ am besten vermittelt. Bjarne Stroustrup hat auf der C++Con 2017 eine Keynote gehalten, die genau dieses Thema zum Kern hatte. Und er sagte dort Dinge, die, so finde ich, weltbewegend sind. Zumindest, was die C++-Welt angeht. Denn er sagte: »Wir (Lehrenden) haben bis C++98 schlechte darin Arbeit geleistet, Menschen C++ beizubringen.« Und er habe sich Gedanken gemacht, warum das so war. Er schließt sich dabei mit ein und resümiert, dass die meisten C++-Bücher lang, eintönig und langsam sind. Sie brächten »bottom-up 1990-C++« bei und benutzten dabei C++11-Syntax. Und das sei verkehrt. Nun habe ich dieses Buch zu schreiben begonnen, lange bevor Bjarne Stroustrup diese Keynote gehalten hat. Und gerade deshalb fühle ich mich im ausklingenden Jahr 2017 in der Art und Weise, wie dieses Buch am Ende des Arbeitsprozesses nun aussieht, bestätigt. Denn ich sehe das genauso und habe mich von Grund auf bemüht, es anders zu machen.

Zum Beispiel werden Sie in diesem Buch Zeiger erst weit hinten erklärt bekommen. Das ist ziemlich gewagt. Zeiger sind wichtig, in C++ dreht sich vieles um Adressen – aber seit C++11 eben nicht alles. Viel wichtiger ist es, das Konzept hinter Zeigern zu verstehen, manifestiert in Iteratoren. Denn wenn man den Mechanismus versteht, kann man das Detail mit anderen Dingen kombinieren und Neues erschaffen. Ich möchte immer das Warum in den Vordergrund gestellt sehen.

Stroustrup sagt in seiner Keynote, dass das neue C++ unter anderem Ressourcensicherheit in den Vordergrund stelle. Er fragt danach, welches Buch RAII deswegen in den Vordergrund stelle? Es seien wenige. Der Begriff RAII wird Ihnen in diesem Buch mehrmals begegnen. Er kritisiert, dass viele Bücher Typsicherheit, Abstraktion, Klassendesign und generische Programmierung nicht einmal erwähnen. Dieses Buch tut es.

Mir sind aber auch Genauigkeit und Sorgfalt wichtig, und darum gibt es hier ebenfalls einen eher technischen Teil, der sich um Syntax und Semantik der kleinen und großen C++-Konstrukte kümmert. Diese kann man in einem Handbuch nicht überspringen. In einem einzelnen Projekt reicht es vielleicht, eine einzelne Regel dazu zu kennen, welche Defaultoperationen man für eine Klasse definieren sollte. In einer Architektur und für das Verständnis des Warum muss man aber wissen, welche Defaultoperationen es gibt und wie sie miteinander interagieren. Mein Ansatz ist daher, dass ich Ihnen die Dinge in drei Bissgrößen vermittle: Der erste Überblick ist in wenigen Seiten erledigt und gibt Ihnen das erste Gefühl für ein C++-Programm. Es folgt die größere Schleife, in der ich auf nahezu jedes Sprachelement kurz eingehe, damit Sie die Interaktionen verstehen: Sie lernen Ausdrücke, Typen, Anweisungen, Variablen und die Standardbibliothek kennen. Erst in der dritten Runde gehe ich in einzelnen Kapiteln auf alle diese Elemente im Detail ein. Dort finden Sie die Dinge mit Hintergrund und Interaktion mit der Welt erklärt: Bits, Bytes, Big-Endian, Fließkommaformate, Exceptions, Klassen und so weiter und so fort.

Besonders am Herzen liegen mir dabei die Kapitel über `vector`, `map` und Konsorten – also das Thema Container. Die Container der Standardbibliothek werden unterschätzt und durchweg zu wenig eingesetzt. Warum? Zurück zu Bjarne: Weil wir es nicht gut genug vermittelt haben. Ich bemühe mich hier um einen anderen Ansatz. Statt nur aufzuzählen, welche Container es gibt und was für Schnittstellen und Eigenschaften sie haben, möchte ich Ihre Aufmerksamkeit mehr auf die Konzepte lenken, besonders auf die Gemeinsamkeiten und Unterschieden zwischen den Containern. Sie sollen sich nicht von Beginn an alle Methoden von `vector` merken, sondern, was die Container können, was sie nicht können und wann Sie welchen wählen sollten. Und genau zu Letzterem habe ich deshalb noch ein eigenes Kapitel geschrieben. Wenn ein Problem gegeben ist, können Sie mit einer Referenz doch nur *dann* den richtigen Container finden, wenn Sie alle Containerbeschreibungen gelesen und verinnerlicht haben. Ich beschreibe also typische Probleme und stelle Ihnen Kriterien vor, nach denen Sie die passenden Container auswählen können.

Und das stellt mich immer noch nicht zufrieden. Wer C++ kann, kann nicht automatisch programmieren. Wer das neue C++ aber richtig anwendet, hat verstanden, worum es geht. Und weil »worum es geht« nicht nur in C++ wichtig ist, sondern auch in anderen Programmiersprachen, war es mir wichtig, dass Sie auch über den Tellerrand hinausschauen. Die eingestreuten Dan-Kapitel beschäftigen sich mit Dingen, die überall in der Softwareentwicklung vorkommen. Egal, ob Sie in Java, PHP oder SQL programmieren. Sie müssen testen, Ihren Code modularisieren, und in den meisten Fällen schadet es nicht, OOP zu kennen. Auf diese Punkte gehe ich ein und wende sie mit C++ an, nehmen Sie sie davon unabhängig mit auf Ihren Weg in Sachen Programmierung und Architektur.

Ich hoffe, dass ich Ihnen mit diesem Buch bei der Arbeit mit C++ helfe. Und wenn dieses Buch Ihnen hilft, C++ anzuwenden, dann wird durch Ihre Arbeit auch C++ besser.

November 2017, Bielefeld

Torsten T. Will

TEIL I
Grundlagen

Sie benötigen einen Compiler oder eine IDE und müssen wissen, wie beide funktionieren und was der Computer mit Ihrem Programm macht.

Dann erfahren Sie, wie die Sprache C++ aufgebaut ist und wie Sie die verschiedenen Bausteine zu einem lauffähigen Programm zusammensetzen.

Kapitel 1
Das C++-Handbuch

Mit diesem Buch auf Ihrem Schreibtisch (denn für Ihre Hände ist es wohl zu schwer) hoffe ich, Ihnen ein Werk zu liefern, dass für Sie die Brücke zwischen Lehrbuch und Referenz darstellt. Ich möchte Ihnen umfassend den Weg zum Programmieren mit C++ bis zu dem Punkt ebnen, an dem Sie wissen, wo Sie weiter nachschlagen können. Das ist vielleicht ein etwas seltsames Ansinnen, aber ich gehe hier bewusst zwei Kompromisse ein: Erstens hoffe ich, dass Sie schon ein wenig über Programmieren im Allgemeinen wissen und vielleicht schon erste Erfahrungen mit der »Denkweise« des Computers gesammelt haben. C++ selbst müssen Sie noch nicht unbedingt kennen, hier setzt dieses Buch auf. Zweitens gibt es zu jedem Sprachelement viele Details, Einsatzmöglichkeiten und Interaktionen mit anderen Sprachelementen – und von diesen *sehr, sehr* viele –, sodass ich Ihnen zwar jedes Sprachelement beschreibe, aber nur bis zu einer gewissen Tiefe. Ich bette die Beschreibungen aber in einen Kontext ein, der Ihnen dabei hilft, ein Verständnis zu entwickeln. Der reine Text des Sprachstandards von C++ inklusive der dazugehörigen Standardbibliothek umfasst über 1400 Seiten – eng gedruckt und formal aufgeschrieben. Ein Werk wie das vorliegende Buch kann nicht anders, als Ihnen diese auf etwa 1000 Seiten verständlich, aber umfassend aufzubereiten, um dann durch eine umfassende Referenz an anderer Stelle ergänzt zu werden. Ich empfehle Referenzseiten (*http://cppreference.com*), Foren (*http://stackoverflow.com*) und die Suche (*http://google.com*, *http://bing.com*) im Internet.

Ich habe dieses Buch so aufgebaut, dass Sie zunächst das Werkzeug, mit dem Sie arbeiten, besser kennenlernen werden. Sie bekommen also Antworten auf die Fragen, was ein Compiler bzw. eine Entwicklungsumgebung ist und wie Sie beides einrichten. Dann erhalten Sie einen schnellen Überblick aus der Vogelperspektive, damit Ihnen die Lektüre der späteren Kapitel leichter fällt.

Danach wird es ausführlicher. Zunächst lernen Sie den Sprachkern kennen: Wie ist ein C++-Programm aufgebaut, und was für Elemente enthält Ihr Code? Hier geht es also hauptsächlich um Syntax und Semantik, Sie sehen außerdem die eingebauten Datentypen und erfahren, wie der Computer mit ihnen rechnet.

Es folgt eine umfassende Beschreibung der Standardbibliothek mit all ihren Werkzeugen, aber auch den Konzepten, die für ihren effektiven Einsatz wichtig sind.

Zu guter Letzt ermögliche ich Ihnen einen kleinen Blick über den Tellerrand hinaus. Sie lernen weitere wichtige Bibliotheken kennen, bekommen einige Tipps dazu, wie Sie selbst

eine Bibliothek entwerfen, und setzen am Schluss Qt (GUI-Toolkit zur plattformübergreifenden Programmierung) ein, um selbst ein Fensterprogramm zu schreiben.

1.1 Neu und modern

Bei all dem, lernen Sie durchgehend das *neue* C++ kennen. Fragen Sie jetzt: »Warum neu?«, dann antworte ich Ihnen: Weil beginnend mit C++11 für C++ ein *neues* Zeitalter angefangen hat. C++ ist runderneuert worden, und wird es immer noch. Mit C++11, C++14, C++17 und dem neuen C++20 ist es in C++ möglich geworden, so zu programmieren, dass man verständlichere, fehlerfreiere und nachhaltigere Programme schreibt – wenn, ja *wenn*, man die Elemente richtig einsetzt.

Im englischen Sprachraum wird für die neue Art, in C++ zu programmieren, der Begriff *modern* verwendet. Im Deutschen passt er nicht so ganz und hat eine leicht andere und daher nicht ganz passende Konnotation. Ich nehme daher lieber *neu* als Begriff, den Sie in diesem Buch daher an der einen oder anderen Stelle finden werden.

Aber was macht diese *neue* oder *moderne* C++-Programmierung aus?

- Sie können kompakter programmieren, weil der Compiler Ihnen viel Ballast abnehmen kann. *Typinferenz* mit `auto` und das bereichsbasierte `for` sind Beispiele dazu.
- Sie schreiben sichereren Code, wenn Sie die *vereinheitlichte Initialisierung* mit {…} bevorzugen, weniger *rohe Zeiger* verwenden, die *Container* und *Algorithmen* der Standardbibliothek nutzen und nicht zuletzt RAII verinnerlichen (siehe Kapitel 17, »Guter Code, 4. Dan: Sicherheit, Qualität und Nachhaltigkeit«).
- Durch *Verschieben* statt Kopieren werden Sie effizienter, ohne etwas dafür tun zu müssen, außer Dinge wegzulassen (siehe Kapitel 17).
- Neue C++-Konstrukte sind eine Alternative zu weniger sicheren C-Mitteln.

Wenn ich in diesem Buch Beispiele präsentiere, sollen sie instruktiv und sinnvoll sein; aber dennoch einfach, kurz und in der Buchform übersichtlich. Ganz wichtig ist mir Praxisnähe. Und da kann es passieren, dass ich das eine oder andere *Muster* verwende, das für das Beispiel vielleicht nicht nötig gewesen wäre. So wird Ihnen zum Beispiel das *Pimpl-Pattern* in Kapitel 11, »Guter Code, 2. Dan: Modularisierung«, begegnen. Im Unterschied zu Mustern wie RAII, die eher im neuen C++ wichtig sind, wird deren Erklärung meist knapp ausfallen, denn hier tendiert meine Abwägung zu »kurz«. Alle etablierten Patterns und Idiome gehören durchaus auch in ein Buch, im Fokus dieses Buchs sind jedoch die neuen.

1.2 »Dan«-Kapitel

An strategisch passenden Stellen habe ich besondere Kapitel eingebaut, die sich weniger mit C++ an sich beschäftigen, Ihnen bei der Programmierung mit C++ aber dennoch helfen werden. Die Themen sind allgemeingültiger Natur aus der Softwareentwicklung wie Mo-

dularisierung, Testen und Ressourcenmanagement, aber konkret angewandt im Kontext von C++.

Diese »Dan«-Kapitel stehen an Stellen im Buch, die thematisch zu den umliegenden Kapiteln gehören. Sie sind aber bewusst weit gefasst und bauen nicht nur auf den vorhergehenden Kapiteln auf. Sie haben mehr generellen Handbuchcharakter und laden dazu ein, sich auch später noch praxisnahe Tipps abzuholen. Beim sequenziellen Durcharbeiten ist der eine oder andere Vorgriff akzeptiert und durchaus gewollt.

Ein besonderes Augenmerk gilt Kapitel 30, »Guter Code, 7. Dan: Richtlinien«, das, wenn es auch sehr kompakt, die meisten praktischen Tipps enthält.

1.3 Darstellung in diesem Buch

In diesem Buch verwende ich durchgehend C++11, C++14 und C++17 als Standard, wenn ich Ihnen Dinge erkläre. Alle neuen C++-Compiler unterstützen den Großteil der Features. Wenn Sie in einer Umgebung arbeiten, die einen alten Compiler bedingt, müssen Sie auf einige nützliche Dinge aus dem C++-Sprachkern verzichten und haben nur Zugriff auf einen eingeschränkten Teil der Standardbibliothek. Entnehmen Sie der Dokumentation Ihres alten Compilers, was Sie nutzen können, und schauen Sie bei *boost* nach, ob Ihre Standardbibliothek damit zu erweitern ist.

Dinge, die Sie vielleicht erst in der neuesten C++-Version C++-20 finden, werde ich im Text erwähnen und im Quellcode besonders markieren. Bei einigen C++17-Features, die umfangreicher sind und Ihnen eine größere Umstellung abverlangen, weise ich ebenfalls darauf hin.

1.4 Verwendete Formatierungen

Listings enthalten die folgenden Elemente:

```
// https://godbolt.org/z/HMWvvE
#include <iostream>                 // cout
#include <memory>                   // make_shared

int main() {                        // ein Kommentar
  std::cout << "Blopp\n";           // hervorgehoben
  Typ feh-ler(args);                // Zeile mit einem Fehler
  std::jthread thr{[]{              // C++20-Features
      std::cout << "Ausgabe" << std::endl; }}
  for(;;) break; // andere Markierung, zur Unterscheidung oder Hervorhebung
}
```

Listing 1.1 Ein kleines Formatbeispiel

Kasten

Kästen enthalten wichtige Dinge, die Sie sich besonders merken sollten.

Balken

Mit einem Balken sind Einschübe markiert, die meist weitergehende Hinweise enthalten. Zu Beginn der meisten Kapitel finden Sie ein *Kapiteltelegramm* mit eingeführten Begriffen neben dem Balken.

Wenn ich einen Namen für eine Symbolfolge im Text verwende, bemühe ich mich, direkt dahinter auch die Symbolfolge anzugeben. Beim Lesen anderer Bücher hätte ich das manchmal hilfreich gefunden. Ich setze das Symbol dann nicht extra zwischen Zitatklammern »«, da ich das in vielen Fällen überladen und manchmal sogar verwirrend finde. Hier ein Beispiel: Sie schreiben Strings in doppelte Anführungszeichen ", verwenden eine Referenz &, nutzen runde Klammern () und spitze Klammern <> und setzen ein Komma , zwischen Parameter.

Beispielmaterial zum Download

Unter *https://www.rheinwerk-verlag.de/5093* finden Sie die Programmbeispiele und Listings zum Herunterladen.

Wenn Sie Fragen oder Bemerkungen haben oder Ihnen eine Diskussion hilft, erreichen Sie mich auf *http://cpp11.generisch.de* oder per E-Mail an *torsten.t.will@gmail.com*.

1.5 Sorry for my Denglish

Wenn man übers Programmieren redet oder schreibt, entsteht zwangsläufig ein Konflikt darüber, ob man Englisch oder Deutsch oder beides verwenden sollte. Manche deutschen auf Teufel komm raus ein, was dann die unmöglichsten Wortkonstrukte ergibt. Ich erinnere mich noch an die Übersetzung des »Joysticks« in der Anleitung eines Computerspiels ...

Ich persönlich bevorzuge das andere Extrem und bleibe lieber beim englischen Begriff, denn der ist in seiner Bedeutung meistens klarer definiert, wie zum Beispiel bei »smarten« Zeigern – die Übersetzung mit »schlau« oder Ähnlichem trifft es nicht so ganz.

Für viele Dinge gibt es aber inzwischen auch fest verankerte deutsche Vokabeln, zum Beispiel das Muster, der inzwischen geläufigen Begriff für »Pattern«. Auch den »Zeiger« bevorzuge ich gegenüber »Pointer«.

Für manches muss man jedoch Begriffe festlegen. Besonderes im Umfeld von C++-eigenen Bezeichnungen bemühe ich mich, einen festen Begriff anderen vorzuziehen, die vielleicht präzisere Übersetzungen oder seltsame deutsch-englische Mixturen wären:

- *Destruktor* bleibt Destruktor
- *Kopierkonstruktor* statt Copy-Konstruktor, Copy-Constructor, Copy-C'tor `T(const T&)`
- *Verschiebekonstruktor* statt Move-Konstruktor, »Movator« oder `T(T&&)`
- *Kopieroperator* oder *Zuweisungsoperator* statt Kopier-Zuweisungsoperator, Copy-Operator, Copy-Assign-Operator oder `T::operator=(const T&)`
- *Verschiebeoperator* statt Verschiebe-Zuweisungsoperator, Verschiebe-Operator, Move-Operator, Move-Assign-Operator oder `T::operator=(T&&)`

Ansonsten bemühe ich mich vor allem um eine eindeutige Sprache. Sollte mir die Gratwanderung der Eindeutigkeit und Ästhetik mal nicht gelingen, bitte ich das zu entschuldigen.

Kapitel 2
Programmieren in C++

C++ ist eine Programmiersprache für viele Zwecke. Generell kann man sie für nahezu alles einsetzen. Durch ihren Fokus auf Performance und Interoperabilität findet man sie häufig in der Systemprogrammierung. Betriebssysteme, Treiber und andere maschinennahe Programme sind besonders häufig in C++ geschrieben.

Vielleicht kommen Sie von Java, C# oder JavaScript. Dann kennen Sie sich mit Programmieren an sich schon aus. Dennoch will ich Ihnen einige Besonderheiten von C++ nahebringen, die Ihnen auf den ersten Blick vielleicht nicht klar sind.

- **C++ wird in Maschinencode übersetzt.**
  JavaScript wird interpretiert, Java in einen Zwischencode übersetzt, der dann interpretiert wird. C++ wird vom Compiler direkt in die Sprache übersetzt, die die Maschine spricht.[1]
- **C++ ist typsicher.**
  C ist in diesem Maße nicht typsicher, ebenso wenig wie Python oder JavaScript.
- **C++ ist generisch.**
  Sie schreiben mit Templates allgemeingültige Vorgehensweisen für mehrere Datentypen. Das ist in C schwerer. In Java haben Sie *Generics*, die aber nicht so mächtig sind. Das prototypbasierte Klassenkonzept von JavaScript und Python erlaubt ähnliche Ergebnisse, geht aber einen anderen Weg.
- **C++ erlaubt Metaprogrammierung.**
  Sie können Programme schreiben, die zur Compilezeit ausgeführt werden.
- **C++ ist ISO-Standard.**
  Das heißt, ein weltweit internationales Komitee entscheidet über die Sprache. Hinter Java steht hauptsächlich Oracle, hinter Python die »Community«.

Der wichtigste Unterschied ist jedoch, dass Sie in C++ das Beste der unterschiedlichsten Paradigmen auswählen *können*, um Ihr Ziel zu erreichen. Sie *müssen* aber keinem der Paradigmen folgen. Wollen Sie Objektorientierung durch Vererbung und dynamischen Polymorphismus wie in Java, gibt das ebenfalls auch in C++, Sie können aber auch ganz ohne programmieren. Koppeln Sie lieber statisch über *Traits*, was man informell *statischen Polymorphismus* nennen könnte, geht dies auch in C++ und wird in der Standardbibliothek zuhauf angewandt, aber Sie können stattdessen auch *virtuelle Methoden* ausreizen.

1 Zumindest ist das der übliche Weg. Der Standard schreibt dies nicht vor, und es gibt C++-Varianten, die das anders machen.

Sie mögen funktionale Ansätze, dann sind Sie bei C++ mit Funktoren und Lambdas besser aufgehoben als bei Java, können aber auch darauf verzichten. Sie können sogar viel der Ausführungszeit in die Compilezeit verschieben und so Benutzern Wartezeit ersparen, indem Sie dem Compiler mit *Metaprogrammierung* mehr Aufgaben übertragen – Sie können aber auch darauf verzichten, wenn Ihnen das nicht wichtig ist.

2.1 Übersetzen

Wenn Sie ein C++-Programm schreiben, dann heißt das, Sie schreiben *Quellcode* als Text, den C++-Werkzeuge in ein ausführbares Programm übersetzen. Wir reden bei diesen Werkzeugen häufig vom *Compiler*, doch in Wirklichkeit sind damit mehrere Tools gemeint. Ich möchte in diesem Abschnitt präzise sein und Ihnen die Aufgaben der unterschiedlichen Werkzeuge nennen. Später werde ich sie wieder unter dem eigentlich nicht ganz richtigen Begriff »Compiler« zusammenfassen.

Diese unklare Benennung liegt unter anderem auch daran, dass heutzutage die Werkzeuge selten noch getrennte Programme sind. Häufig sind es lediglich *Phasen* eines einzigen Programms. Und tatsächlich spiegelt auch Abbildung 2.1 nur einen vereinfachten Ablauf wider. Außen vor habe ich die Optimierungen gelassen sowie die Tatsache, dass ein Programm, wenn Sie es ausführen, noch zusätzliche Bibliotheken verwendet (dynamische Bibliotheken).

Dieser ganze Prozess wird entweder aus der *integrierten Entwicklungsumgebung* (IDE) heraus angestoßen, oder man führt ihn von Hand auf der Kommandozeile aus. Wobei »von Hand« eigentlich übertrieben ist, denn sehr häufig verwendet man auch hier ein Werkzeug, ein *Buildtool* (in etwa: Bauwerkzeug). Sehr verbreitet sind »Makefiles«. Das sind Textdateien, in denen steht, welche Komponenten zum Programm gehören.

Später im Kapitel werden wir ein kleines Programm sowohl mit einer IDE bauen als auch auf der Kommandozeile ausführen und uns dafür eines Makefiles bedienen.

2.2 Übersetzungsphasen

Um aus dem Quellcode, den Sie schreiben, letztendlich ein ausführbares Programm zu machen, führt die Compilersuite mehrere Phasen aus. Heutzutage ist es häufig nur noch ein Programm, das alles übernimmt. Eventuell teilt man noch in Frontend und Backend auf. Früher wurden die Phasen mehr oder weniger von getrennten Programmen ausgeführt. Die wichtigsten Phasen sind:

- **Präprozessor**
  Liest die Include-Dateien und expandiert C-Makros, das heißt, setzt textuell eine vorherige Definition anstelle des Makronamens ein.

- **Lexer und Parser**
  Übersetzt die Zeichenfolgen in *Token*, gruppiert also Zeichen zu semantischen Einheiten und erkennt dabei zum Beispiel, ob es sich um einen String, eine Zahl oder eine Funktion handelt.
- **C++-Semantik**
  Macht aus Templates echte Funktionen und Klassen und aus Klassen deren Instanzen.
- **Zwischenrepräsentation (»Intermediate Representation« oder »IR«)**
  Erzeugt einen maschinennahen, aber dennoch plattformunabhängigen Code und optimiert diesen auf hoher Ebene.
- **Codegeneriererung (»CG«)**
  Erzeugt plattformabhängigen Maschinencode in Assemblersprache und optimiert auf niedriger Ebene.
- **Objektdateien**
  Übersetzt die Assemblersprache in maschinenlesbare Bytefolgen, gibt diese in Objektdateien und statische Bibliotheken aus und fügt Debuginformationen hinzu.
- **Zusammenfügen (»Linken«)**
  Erstellt eine ausführbare Datei oder dynamische Bibliothek aus Objektdateien und Bibliotheksdateien.

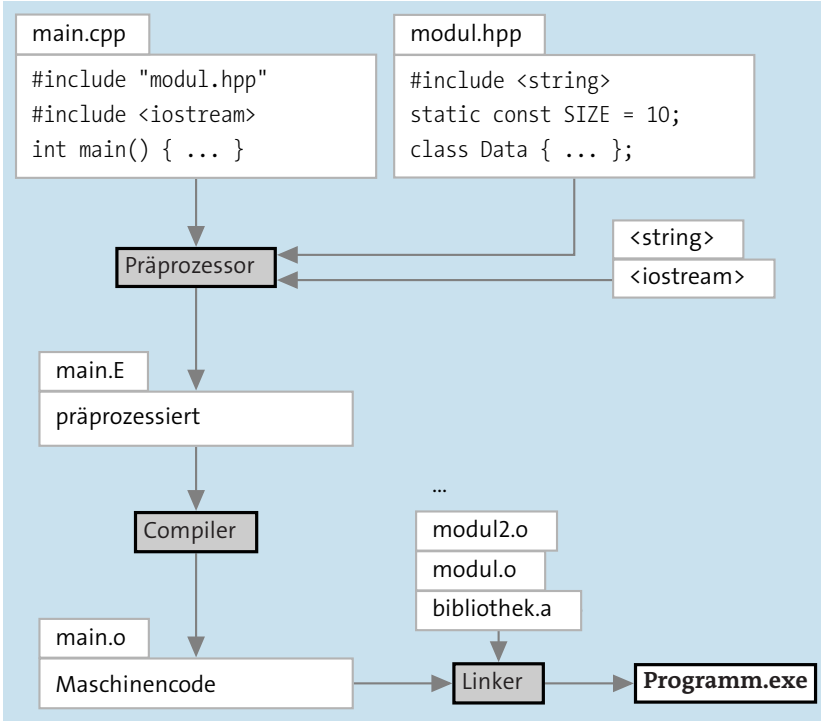


Abbildung 2.1 Die Phasen der Kompilierung vom Quellcode zum Programm

2.3 Aktuelle Compiler

Wie bereits gesagt, umfasst der Begriff *Compiler* aber nicht ein einzelnes Übersetzungsprogramm, sondern die gesamte Werkzeugkette vom Präprozessor bis zum Linker. Hinzu kommt, dass die *Standardbibliothek* integraler Bestandteil von C++ ist. Wenn Sie also einen Compiler auf Ihrem System installieren, erhalten Sie auch immer eine Standardbibliothek dazu.

Mit wohlmeinender Absicht bezieht sich dieses Buch zu großen Teilen auf den aktuellen Standard, der unter dem Namen C++17 bekannt ist. Bei der Generalüberholung seit C++11 hat die Sprache viel Potenzial dazubekommen, um eine rundere, sicherere, konsistentere und nicht zuletzt auch einfacher zu erlernende Sprache zu sein. Wenn Sie die C++17-Möglichkeiten nutzen, werden Sie die Sprache schneller erlernen und besser einsetzen können, als das zum Beispiel noch mit den Vorläuferversionen bis C++98 der Fall gewesen war.

C++ macht mit einem aktuellen Compiler auf jeden Fall mehr Spaß. Zum Glück sind die meisten Compiler mehr oder weniger auf dem neuesten Stand.

2.3.1 Gnu C++

Der C++-Compiler *g++* aus der *Gnu Compiler Collection* (*GCC* genannt) ist der auf den meisten Plattformen verfügbare Compiler. Er ist gleichzeitig die Experimentierwiese, um neue Dinge auszuprobieren, sodass Sie hier beinahe immer zuerst die neuen Features implementiert finden. Auf Linux ist GCC meist die erste Wahl. GCC ist zwar weit verbreitet, hat aber den Ruf, eine sehr komplexe Codebasis zu haben. Die kompilierten Programme fallen gegenüber kostenpflichtigen Compilern, was die Geschwindigkeit angeht, etwas zurück.

2.3.2 Clang++ der LLVM

Was die Codebasis angeht, hat der *LLVM* mit seinem C++-Compiler namens *Clang++* einen besseren Ruf. Die Umsetzung der C++11- bis C++20-Features ist vorbildlich. Manche neuen Features werden hier zuerst implementiert. Clang++ ist der Standardcompiler für die macOS-Entwicklung. Für Linux steht er kostenlos zur Verfügung, muss jedoch zu einer bestehenden Standardbibliothek hinzuinstalliert werden, sodass Sie den g++ am besten vorher installieren.

2.3.3 Microsoft Visual Studio

Die Compilersuite des Windows-Herstellers steht den anderen Compilern nur wenig nach, was die Umsetzung des C++17- und auch des C++20-Standards angeht. Die meisten und wichtigsten Features sind enthalten.

2.3.4 Compiler im Container

Mit einem geeigneten Docker-Container können Sie andere Compiler mit Ihrem Code ausprobieren, ohne sie auf Ihrem System installieren zu müssen und es dadurch möglicherweise durcheinanderzubringen.

Die Gnu Compiler Collection bietet eine ganze Reihe fertiger Container an. So übersetzen Sie beispielsweise ein Stück Quellcode mit dem g++ in Version 9.2, der schon einiges von C++20 unterstützt (*https://hub.docker.com/_/gcc*):

```
docker run --rm -i -t --volume $PWD:/workdir --workdir /workdir gcc:9.2 \
    g++ -std=c++2a -Wall -Wextra -pedantic meineDatei.cpp -o meineDatei.x
```

Weil sich die dynamischen Bibliotheken im Container befinden, müssen Sie die Datei auch darin ausführen:

```
docker run --rm -i -t --volume $PWD:/workdir --workdir /workdir gcc:9.2 \
    ./meineDatei.x
```

Wenn Sie in Linux einmal Clang++ ausprobieren wollen, dann bauen Sie sich einen Docker-Container mit einer fertigen Clang++-Installation. Mein Dockerfile basiert auf Paul Silisteanus Dockerfile[2]:

```
FROM ubuntu:18.04

RUN apt-get update && apt-get install -y \
  xz-utils \
  build-essential \
  curl \
  && rm -rf /var/lib/apt/lists/* \
  && curl -SL \
    http://releases.llvm.org/9.0.0/clang+llvm-9.0.0-x86_64-linux-gnu-ubuntu-18.04.tar.xz \
  | tar -xJC . && \
  mv clang+llvm-9.0.0-x86_64-linux-gnu-ubuntu-18.04 clang_9.0.0 && \
  echo 'export PATH=/clang_9.0.0/bin:$PATH' >> ~/.bashrc && \
  echo 'export LD_LIBRARY_PATH=/clang_9.0.0/lib:$LD_LIBRARY_PATH' >> ~/.bashrc
ENV PATH="/clang_9.0.0/bin:${PATH}"
ENV LD_LIBRARY_PATH="/clang_9.0.0/lib:${LD_LIBRARY_PATH}"
CMD [ "/bin/bash" ]
```

Von mir stammen die beiden ENV-Zeilen. So können Sie den Container bauen (beachten Sie den abschließenden Punkt, der für das aktuelle Verzeichnis steht):

```
docker build -t clang9 .
```

2 *Solarian Programmer, https://solarianprogrammer.com/2017/12/14/clang-in-docker-container-cpp-17-development/*, [2019-11-03]

Und so können Sie dann eine Datei übersetzen:

```
docker run --rm -i -t --volume $PWD:/workdir --workdir /workdir clang9 \
    clang++ -std=c++2a -stdlib=libc++ -Wall -Wextra -pedantic \
    meineDatei.cpp -o meineDatei.x
```

Auch hier müssen Sie das fertige Programm im Container ausführen:

```
docker run --rm -i -t --volume $PWD:/workdir --workdir /workdir clang9 \
    ./meineDatei.x
```

2.4 Entwicklungsumgebungen

Es gibt für Sie hauptsächlich zwei Möglichkeiten, C++-Programme zu entwickeln:

- **Kommandozeile**
  Sie arbeiten auf der Kommandozeile und rufen den Compiler und andere Werkzeuge von Hand auf. Später nutzen Sie dann Hilfsmittel wie Makefiles, um diese Aufgaben zu automatisieren. Ich empfehle, den Weg über die Kommandozeile zumindest auszuprobieren. Zum einen lernt man dabei auch andere nützliche Dinge über Programme und das Programmieren. Zum anderen lassen sich Abläufe auf der Kommandozeile besser automatisieren – und um das Automatisieren geht es uns beim Programmieren ja letzten Endes.
- **Integrierte Entwicklungsumgebung**
  Sie verwenden eine sogenannte IDE (*Integrated Development Environment*; dt. *Integrierte Entwicklungsumgebung*). Gerade im späteren Programmieralltag kann eine auf einen persönlich zugeschnittene IDE die Produktivität immens erhöhen. Auf der anderen Seite kann eine IDE einen Anfänger mit ihrer Feature-Flut auch erschlagen. Es gibt Assistenten, die den Einstieg zu beschleunigen versuchen. Ob das klappt, hängt von Ihrer Persönlichkeit ab. Wenn Sie mit der Kommandozeile absolut nicht vertraut sind, können Sie hiermit einen Versuch wagen.

In manchen Fällen gibt die Wahl des Compilers die IDE vor. Wenn Sie sich für Microsoft entscheiden, dann ist *Visual Studio* Ihre IDE. Die schon sehr umfangreiche Basisversion mit *Visual Studio Community* ist kostenlos. Die Editionen *Professional* und *Enterprise* sind kostenpflichtig. Zur Drucklegung dieses Buchs war die Version *Visual Studio 2017* aktuell. Auf die werde ich mich im Verlauf dieses Buchs beziehen, sie enthält die Version 14.1 des C++-Compilers. Sehen Sie sich auf der Webseite von Visual Studio (*https://www.visualstudio.com/vs/community*) die Optionen an. Inzwischen gibt es *Visual Studio 2019*, und der aktuellste Compiler hat Version 19.25. Die für dieses Buch relevanten Unterschiede halten sich in Grenzen, außer dass der neuere Compiler natürlich die aktuellen Features besser unterstützt.

Auf dem Mac ist das von Apple gelieferte XCode der De-facto-Standard. Damit haben Sie die Wahl zwischen einer exzellenten IDE und einer Sammlung an Werkzeugen für die Kommandozeile. Bei Apple können Sie diese herunterladen (*https://developer.apple.com/xcode*). Aktuell erhalten Sie XCode 11 mit der Compilerversion 11. Der Compiler ohne IDE lässt sich auch unter Linux nutzen, da ist als aktuellste die Version 9 verfügbar (*https://releases.llvm.org/*).

Sowohl unter Unix als auch unter Windows und auf dem Mac steht Ihnen als Alternative die *Gnu Compiler Collection* zur Verfügung. Der C++-Compiler heißt *g++* – aktuell in der Version 9.3 –, und 10.0 steht in den Startlöchern. Sie bedienen ihn in erster Linie von der Kommandozeile aus, er integriert sich aber auch in IDEs wie Eclipse mit CDT, Netbeans, KDevelop, Code::Blocks, den Qt Creator und andere. Manche dieser Tools gibt es sogar für mehrere Plattformen. Wenn Sie mit dem g++ unter Windows entwickeln wollen, dann schauen Sie nach der MinGW-Integration und ob Sie sie getrennt herunterladen müssen oder ob sie schon mitgeliefert wird (*Minimal Gnu for Windows*).

Linux-Subsystem unter Windows 10

Seit Oktober 2017 gibt es unter Windows 10 optional ein *Windows Subsystem for Linux* (WSL). Sie können dann eine Bash-Shell starten und haben darin dieselben Befehle zur Verfügung, wie zum Beispiel unter Ubuntu – inklusive des Installationsbefehls `apt-get`. So können Sie sich unter Windows dann einen gcc und viele andere Dinge nachinstallieren. Das Thema ist für dieses Buch noch zu frisch, als dass ich Ihnen eine umfangreiche Anleitung liefern könnte.

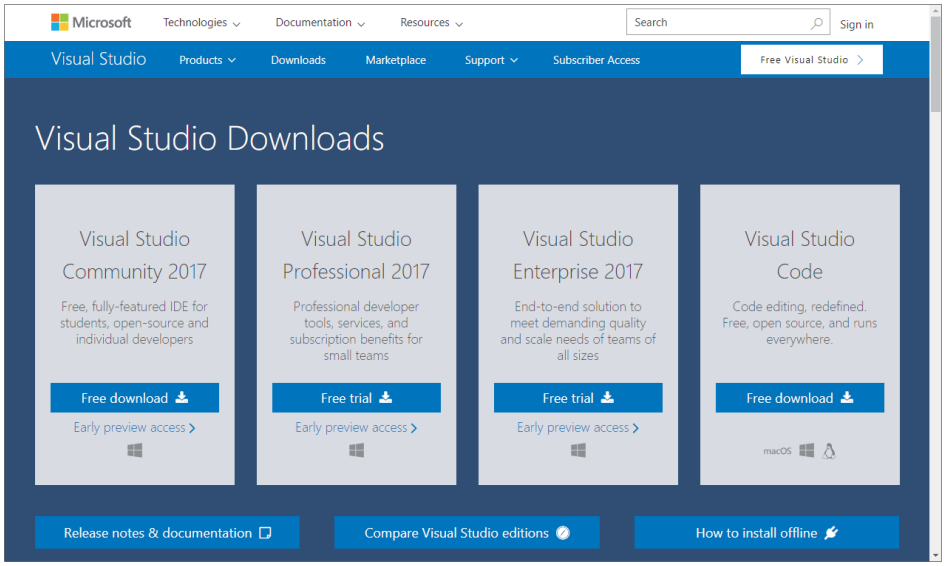


Abbildung 2.2 Microsoft-Produkte für die C++-Entwicklung

Eine kommerzielle, aber interessante Lösung ist JetBrains *CLion*. Die IDE ist sehr durchdacht und lehnt sich an das erfolgreiche *IntelliJ Idea* an, was sie für manche Java-Entwickler interessant macht. Der eigene Compiler ist, was die Standards angeht, etwas hinterher, doch kann man einen installierten GCC- oder Clang-Compiler konfigurieren.

2.5 Die Kommandozeile unter Ubuntu

Exemplarisch für die Entwicklung mit der Kommandozeile gebe ich Ihnen eine Kurzanleitung für die aktuelle Langzeitversion von Ubuntu, einer weitverbreiteten Linux-Distribution. Wenn Sie ein anderes Linux verwenden, unterscheiden sich die Kommandos vielleicht.

Sie installieren den g++ und einige nützliche Werkzeuge so:

```
sudo apt-get install g++ make
```

Den Programmcode geben Sie in einem Editor ein, und da beginnt die wirkliche Qual der Wahl. Wenn Sie eine IDE verwenden, ist der Editor mit dabei. Ohne IDE geben Sie Ihr Programm in einem beliebigen allgemeinen Texteditor ein. Texteditoren gibt es wie Sand am Meer.

Ich schlage hier nur drei vor, die unterschiedlichen Anforderungen gerecht werden: *jedit*, *gedit* und *kate*. Weil jedit in Java geschrieben ist, gibt es ihn wiederum auf allen Plattformen. Die Wahl zwischen gedit und kate sollten Sie davon abhängig machen, ob Sie als Desktop Gnome oder KDE einsetzen. Probieren Sie einfach aus, welcher der Befehle

```
sudo apt-get install gedit
sudo apt-get install kate
```

weniger Pakete automatisch installieren würde, und wählen Sie danach den Editor aus.

Wenn Sie in ein Team mit mehreren Entwicklern kommen, erkundigen Sie sich, ob *Emacs* oder *Vim* eingesetzt wird. Dabei handelt es sich um unter Programmierern sehr verbreitete Texteditoren, die aber eine steile Lernkurve haben. Wenn Sie Kollegen haben, die Ihnen beim Einstieg helfen, wählen Sie ruhig einen dieser beiden Editoren.

```
sudo apt-get install emacs
sudo apt-get install vim
```

2.5.1 Ein Programm erstellen

Wie gesagt, zur IDE kommen wir gleich, wenn wir exemplarisch das Microsoft Developer Studio unter Windows besprechen. Jetzt gehen Sie den Weg einmal zu Fuß.

Öffnen Sie eine Kommandozeile, manchmal auch *Terminal* oder *Konsole* genannt. Dazu finden Sie im Menü sicherlich einen Eintrag. Bei Ubuntu mit Gnome können Sie auch

Strg + Alt + T drücken. Es sollte sich ein neues Fenster mit einem blinkenden Cursor öffnen, das eine Kommandozeile ähnlich wie diese zeigt, das sogenannte Prompt:

```
towi@havaloc:~$
```

Im weiteren Verlauf dieses Buchs werde ich `towi@havaloc:`, das für Benutzer- und Rechnernamen steht, und meist auch die Tilde ~ für das aktuelle Arbeitsverzeichnis nicht mehr extra erwähnen. Mit dem Prompt `$` meine ich, dass Sie dahinter ein Kommando eingeben sollen. Üben Sie einmal, ein neues Verzeichnis zu erstellen und dieses zu betreten.

```
~$ mkdir quellcode
~$ cd quellcode
```

Nun sollte Ihr Prompt das Verzeichnis beinhalten, in das Sie gewechselt sind:

```
~/quellcode$
```

Da es so viele Linux-Geschmacksrichtungen gibt, ist es durchaus möglich, dass Ihre Anzeige anders aussieht, obwohl Sie alles richtig gemacht haben. Sie können mit `pwd` überprüfen, in welchem Verzeichnis Sie gerade stehen.

Öffnen Sie nun den Editor Ihrer Wahl. Sie können das über die Menüs erledigen oder auf der Kommandozeile gleich den Namen der Datei angeben, die Sie bearbeiten wollen. Fügen Sie noch ein Ampersand `&` an, damit Sie trotz geöffnetem Editor weitertippen können (sollten Sie das vergessen, drücken Sie in der Kommandozeile Strg + Z und tippen danach den Befehl `bg`, gefolgt von ↵, ein). Ich selbst bin ein Emacs-Nutzer, Sie setzen hier Ihren Lieblingseditor ein:

```
$ emacs modern101.cpp &
```

Tippen Sie den folgenden Quellcode in das Editorfenster. Sie sollten irgendwo erkennen, dass Sie wirklich `modern101.cpp` bearbeiten.

```
// https://godbolt.org/z/KhybaP
// modern101.cpp : Fibonacci-Konsole
#include <iostream>
int fib(int n) {
    return n<2 ? 1 : fib(n-2) + fib(n-1);
}
int main() {
    std::cout << "Die wievielte Fibonacci-Zahl? ";
    int n = 0;
    std::cin >> n;
    std::cout << "fib(" << n << ")=" << fib(n) << "\n";
}
```

Listing 2.1 Jede Fibonacci-Zahl ist die Summe der beiden Zahlen davor.

Übersetzen Sie diesen Quellcode in das ausführbare Programm `modern101.x`:

```
$ g++ modern101.cpp -o modern101.x
```

Hier ist g++ der Compiler. Mit modern101.cpp geben Sie die Quelldatei an. Haben Sie mehrere Quelldateien, die Sie zu einem Programm zusammensetzen wollen, geben Sie hier mehrere *.cpp-Dateien an. Mit -o modern101.x teilen Sie diesem Compiler den gewünschten Ausgabedateinamen mit. Wenn Sie den vergessen, ist das nicht schlimm, dann landet das fertige Programm bei g++ in a.out.

Probieren Sie es aus:

```
$ ./modern101.x
Die wievielte Fibonacci-Zahl? 33
fib(33)=5702887
```

Ihr erstes C++-Programm – herzlichen Glückwunsch!

Übrigens: Unter Windows werden Sie ausführbare Programme normalerweise mit der Endung *.exe versehen. Unter Linux ist eine Endung für ausführbare C++-Programme eher unüblich. Zur Verdeutlichung erzeuge ich hier Linux-Programme, aber mit der Endung *.x – eine Praxis, die ich auch in der »wirklichen Welt« zuweilen pflege. Ob Sie mir das nachmachen oder nicht, bleibt Ihnen überlassen.

Wenn Sie interessiert, wie Sie das Programm beschleunigen können, blättern Sie zu Abschnitt 2.7, »Das Beispielprogramm beschleunigen«, vor.

2.5.2 Automatisieren mit Makefile

Da es aber mühselig ist, den Compiler auf diese Art immer wieder aufzurufen, erstellen Sie sich am besten ein Makefile, in dem die nötigen Befehle verzeichnet sind. Erstellen Sie eine dazu die folgende Datei Makefile:

```
$ emacs Makefile &
```

Der Inhalt der Datei ist dann simpel:

```
# -*- Makefile -*-
all: modern101.x
modern101.x: modern101.cpp
  →  g++ modern101.cpp -o modern101.x
# aufräumen:
clean:
  →  rm -f *.x *.o
```

Die Kommentarzeilen mit # sind nicht essenziell. Achten Sie *unbedingt* darauf, dass die eingerückten Zeilen nicht mit Leerzeichen, sondern einem *Tabulator* anfangen, das deute ich hier mit → an. Auf alle Details gehe ich hier nicht ein, aber die beiden Zeilen

```
modern101.x: modern101.cpp
  →  g++ modern101.cpp -o modern101.x
```

sagen make: Wenn du modern101.x erstellen sollst, dann benötigst du dazu modern101.cpp; um es zu erstellen, führe den Befehl g++ modern101.cpp -o modern101.x aus.

Wenn Sie nun

```
$ make
```

ausführen, wird bei der `all`:-Regel nachgeschaut, was Sie alles gebaut haben wollen. Dort können Sie andere Regeln auflisten, die `make` dann nacheinander ausführt. Nun sollten Sie wieder Ihr Programm gebaut bekommen. Eventuell merkt `make`, dass sich nichts geändert hat, dann hilft ein `make clean` (Aufräumregel ausführen) oder `make -B` (tu so, als hätte sich alles geändert).

Sie können auch mit `make all`, `make modern101.x` oder `make clean` eine der anderen Regeln ausführen lassen. Das ist alles schon sehr praktisch.

2.6 Die IDE »Microsoft Visual Studio Community« unter Windows

Der Download (*http://www.visualstudio.com/downloads*) der Edition *Visual Studio Community* gliedert sich in zwei Schritte. Zunächst laden Sie nur ein Programm zum Downloaden und Installieren herunter. Nach dem Start können Sie wählen, für welche Komponenten Sie sich interessieren. In Abbildung 2.3 sehen Sie eine Auswahl für simple C++-Projekte. Es steht Ihnen frei, hier mehr zum Experimentieren auszuwählen.

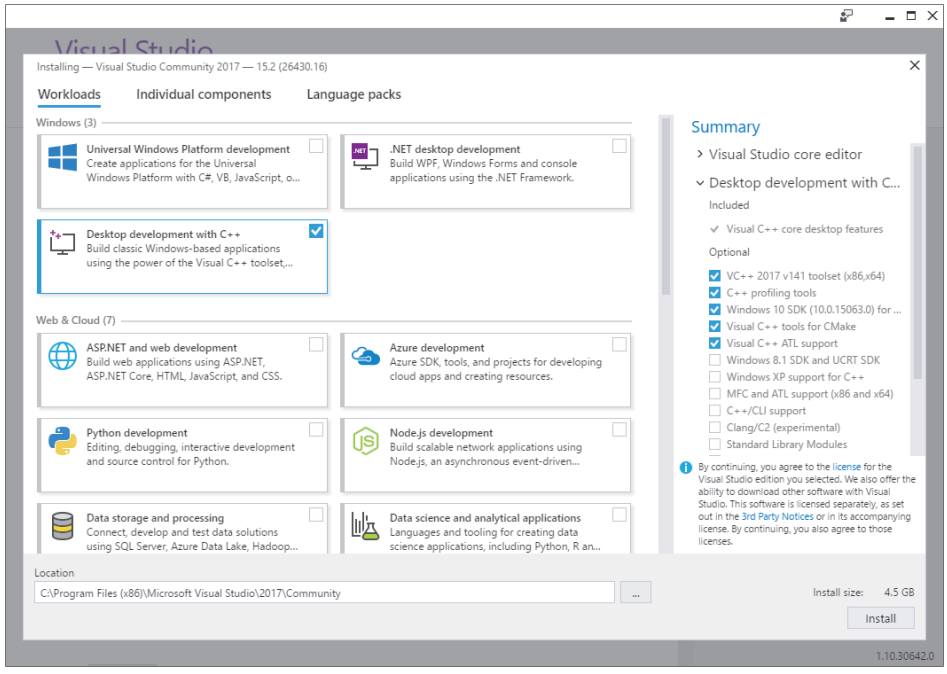


Abbildung 2.3 Wählen Sie die gewünschten C++-Komponenten zur Installation aus.

Wenn Sie die Auswahl bestätigen, geht es mit Download und Installation los. Beim ersten Start fragt Sie *Visual Studio* noch nach Ihrer »Lieblingsumgebung«. Hier wählen Sie sinnvollerweise Visual C++.

Wenn diese Hürde genommen ist, begrüßt die IDE Sie mit einem Startbildschirm. In IDEs dreht sich meist alles um *Projekte*.

- Beginnen Sie mit FILE • NEW • PROJEKT...
- Wählen Sie in dem Assistenten INSTALLED • TEMPLATES • VISUAL C++ • WIN32 • WIN32-CONSOLE-APPLICATION.
- Tragen Sie als NAME `Modern101` ein.
- Überprüfen Sie, ob Ihnen der Speicherort LOCATION gefällt. Der Haken CREATE DIRECTORY FOR SOLUTION sollte gesetzt bleiben.
- Klicken Sie auf OK, um zum eigentlichen Assistenten zu gelangen.
- Sie beginnen mit den APPLICATION SETTINGS (siehe Abbildung 2.4).

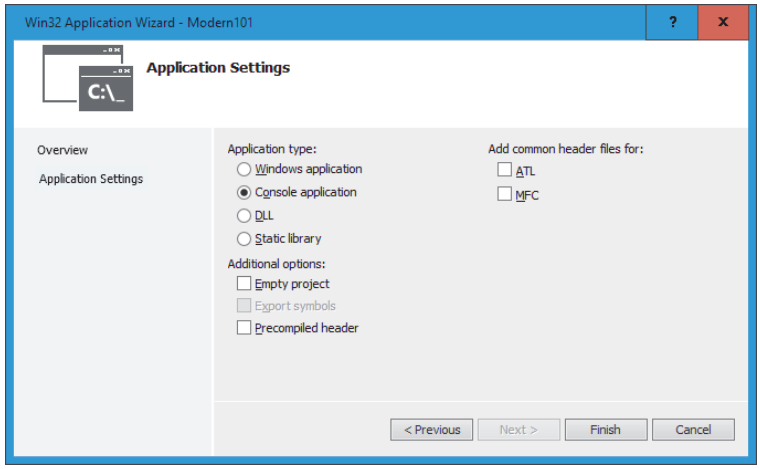


Abbildung 2.4 So erstellen Sie das Grundgerüst einer Konsolenanwendung.

- Stellen Sie sicher, dass als Anwendungstyp CONSOLE APPLICATION ausgewählt ist. Damit bekommen Sie das Grundgerüst für ein Programm mit einem Textfenster. Die anderen Möglichkeiten ergäben ein grafisches Windows-Programm oder Bibliotheken, die später Teil eines anderen Programms werden.
- In den ADDITIONAL OPTIONS lassen Sie EMPTY PROJECT und PRECOMPILED HEADER frei. Letztere beschleunigen den Kompiliervorgang erheblich, basieren aber möglicherweise auf nicht portablen Erweiterungen im Microsoft-Compiler.
- Die Checkboxen für weitere HEADER FILES benötigen Sie zunächst nicht. Sowohl die MFC als auch die ATL sind Microsoft-spezifisch.
- Nach dem Klick auf FINISH erstellt die IDE Ihnen das Grundgerüst des Programms.

Das Grundgerüst beinhaltet Projekt-, Quell-, Header- und Dokumentationsdateien im Projektverzeichnis.

Sie können mit dem vom Assistenten erstellten Quellcode beginnen. Eventuell enthält dieser das plattformspezifische `_tmain` statt `main` und die Dateien `stdafx.h` und `stdafx.cpp`. Wenn Sie `_tmain` sehen, dann ersetzen Sie dies bitte durch `main`. Die Dateien `stdafx.h` und `stdafx.cpp` können Sie für dieses Miniprojekt löschen. Ersetzen Sie dann im Editor den Inhalt der Datei `Modern101.cpp` durch den Text aus Abbildung 2.5.

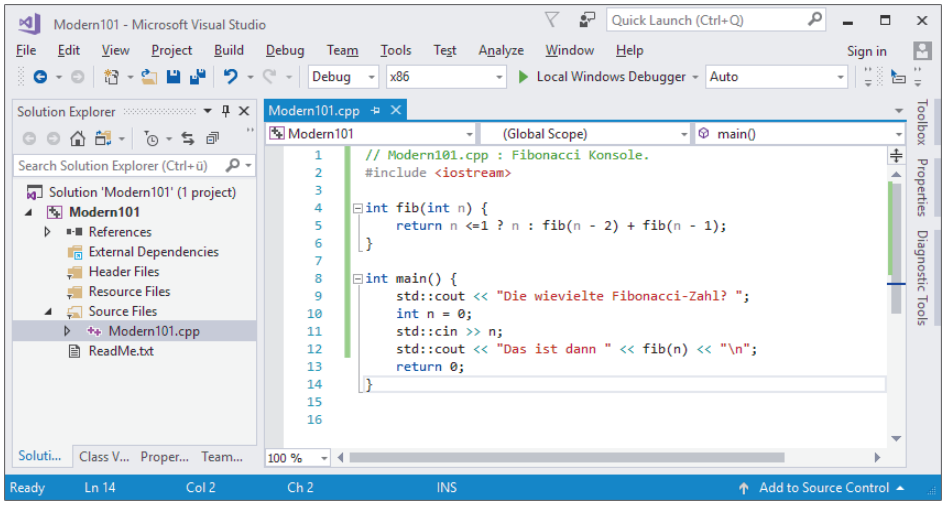


Abbildung 2.5 Das Grundgerüst einer Win32-Konsolenapplikation

Wählen Sie den Menüpunkt DEBUG • START WITHOUT DEBUGGING und bestätigen Sie den Bau des Programms mit YES. Wenn alles geklappt hat (und Sie sich beim Abtippen nicht vertan haben), erscheint nach kurzer Übersetzungszeit ein schwarzes Konsolenfenster und erwartet von Ihnen die Eingabe einer Zahl. Herzlichen Glückwunsch, Sie haben Ihr erstes C++-Programm geschrieben. Geben Sie `43` ein, und die Antwort `433494437` sollte nach einiger Zeit auf dem Bildschirm erscheinen (siehe Abbildung 2.6).

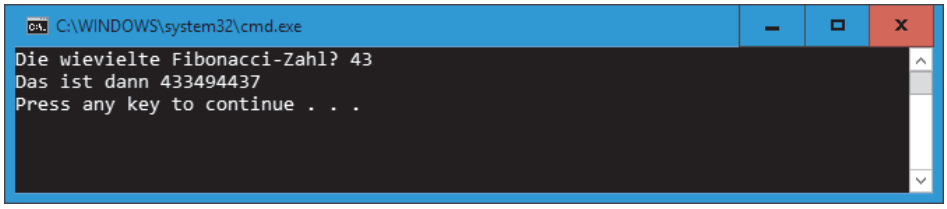


Abbildung 2.6 Ein selbst erstelltes Konsolenprogramm unter Windows

Auf meinem Computer dauert die Berechnung etwa eine Minute. Sie können auch mit einer kleineren Zahl wie `20` starten. Wenn Sie eine größere Zahl als `45` wählen, sprengen Sie die Fähigkeiten des Programms und bekommen unsinnige Ausgaben.

Ein Hinweis: Die Zahlenbereiche beziehen sich auf eine Win32-Applikation auf einem 64-Bit-Windows. Auf anderen Plattformen mögen die Grenzen andere sein.

2.7 Das Beispielprogramm beschleunigen

Wenn Ihnen das Programm zu lange läuft, dann *tabulieren* Sie die Zwischenergebnisse. Das bedeutet, Sie speichern sie in einer tabellenartigen Datenstruktur ab. Der nullte Eintrag bekommt das Ergebnis von `fib(0)`, der erste das Ergebnis von `fib(1)` etc. Wenn Sie mit `fib(n)` den *n*-ten Eintrag in der Tabelle berechnen wollen, rufen Sie nicht `fib(n-1)` und `fib(n-2)` auf, sondern schauen stattdessen in die Tabelle. Dann erhalten Sie die Ergebnisse schneller, als Sie gucken können. Erstellen Sie ein neues Projekt oder modifizieren Sie das Programm:

```
// https://godbolt.org/z/Ev74bbPjb
// modern102.cpp : Fibonacci-Konsole
#include <iostream>
#include <map>
int fib(int n) {
    static std::map<int, int> table{};
    table[n] = n<=1 ? n : table[n-2] + table[n-1];
    return table[n];
}
int main() {
    std::cout << "Wie viele Fibonacci-Zahlen? ";
    int n = 0;
    std::cin >> n;
    for (int i = 1; i <= n; ++i)
        std::cout << "fib(" << i << ")=" << fib(i) << "\n";
}
```

Listing 2.2 Eine zügig erstellte Tabelle von Fibonacci-Zahlen

Wenn Sie hier zum Beispiel `50` eingeben, sehen Sie, dass die Ergebnisse ab `46` auch mal negativ sind – ein Zeichen für einen Überlauf.[3] Das heißt, die Zahlen werden für dieses Programm zu groß, und das Programm liefert Unsinn. Einen Überlauf in einem Programm zu haben, ist normalerweise keine gute Idee. Daher werden Sie in Abschnitt 4.12, »Eingebaute Datentypen«, lernen, worauf Sie achten müssen und wie Sie sie vermeiden.

3 Wieder: Auf anderen Plattformen als 32-/64-Bit-Windows-10 haben Sie vielleicht andere Grenzen.

Kapitel 3
C++ für Umsteiger

Dieses Kapitel richtet sich vor allem an Umsteiger von Java oder C#, aber auch Umsteiger höheren Sprachen, vor allem OO-Sprachen, profitieren ebenso wie Auffrischer, deren C++-Kenntnisse länger zurückliegen. Ich gebe einen allgemeinen Überblick über Eigenheiten von C++, die Umsteiger überraschen oder ihnen Schwierigkeiten machen können.

Alle anderen Leser können dieses Kapitel überspringen, Überblick und Einstieg kommen im Anschluss.

Die folgenden Elemente sind Beispiele, die viele Entwickler wiedererkennen sollten:

- **Anweisungen**
  Programme werden Anweisung für Anweisung eine nach der anderen ausgeführt – zumindest pro Thread, zumindest im Modell. Es gilt die Faustregel, dass Semikolons Anweisungen voneinander trennen. Anweisungen können zu Blöcken zusammengefasst werden.
- **Ausdrücke**
  Ein Ausdruck besteht rekursiv wieder aus Ausdrücken bis runter zu unteilbaren Einheiten wie Literalen oder Variablen. Arithmetische Ausdrücke enthalten zum Beispiel mathematische Berechnungen. In C++ kann jedem Ausdruck ein exakter Typ zugeordnet werden.
- **Datentypen**
  C++ bietet eine Reihe von einfachen Datentypen wie `int` und `double`. Hinzu kommen Zeiger und Referenzen, die in C++ eigene Typen sind. Ein `int` und ein Zeiger darauf `int*` sind unterschiedliche Typen. Sie können mehrere Typen zusammenaggregieren und so neue Typen erhalten.
- **Funktionen alias Methoden**
  Damit Programme nicht zu langen Spaghetti werden, kann man wiederverwendete Bereiche in Funktionen auslagern. Funktionen, die in einem Datentyp stehen, nennt man Methoden.
- **Klassen**
  Datentypen, die Sie zusätzlich mit Verhalten bündeln, also Methoden hinzufügen, nennt man Klassen.
- **Funktionsaufrufe**
  Ein Funktionsaufruf nimmt Parameter und liefert ein Ergebnis zurück. Was innerhalb der Funktion passiert, ist teilweise außen unsichtbar.

- **Parameter**
  Funktionen bekommen Parameter. In C++ entscheidet die Funktion darüber, ob der Parameter als Wert (*by Value*) oder als Referenz – oder Zeiger – (*by Reference*) übergeben werden soll, nicht der Aufrufer.
- **Rückgaben**
  Gleiches gilt für die Rückgabe aus Funktionen. Das Ergebnis kann einer Variablen zugewiesen oder innerhalb eines Ausdrucks weiterverwendet werden. Die Funktion entscheidet, ob die Rückgabe als Wert eine unabhängige Kopie ist oder eine Referenz.

Manche Dinge mögen erfahrenen Programmierern auf den ersten Blick bekannt vorkommen, weisen im Detail jedoch wichtige konzeptionelle Unterschiede auf. Wenn zum Beispiel Java-Entwickler hier falsche Verständnisvoraussetzungen mitbringen, könnten sie später verwirrt sein und böse Überraschungen erleben. Daher will ich ein paar Dinge kurz erwähnen, die Stolperfallen sein könnten:

- **Stack und Heap statt Garbage Collection**
  Es ist sicher keine Überraschung, wenn ich Ihnen sage, dass es in C++ kein automatisches Aufräumen der Objekte gibt. Betrachten Sie dies nicht als Nachteil, leben Sie den Vorteil. Trennen Sie zwischen Dingen, die auf dem *Stack* automatisch vom Compiler verwaltet werden, wenn der Block verlassen wird, und denen, die Sie auf dem *Heap* mit `new` anfordern und für die Sie die Verantwortung fürs Wegräumen übernehmen. Schluren Sie nicht herum, nutzen Sie besser *RAII* (siehe Kapitel 17, »Guter Code, 4. Dan: Sicherheit, Qualität und Nachhaltigkeit«).
- **Virtuelle versus echte Maschine**
  Und es ist ebenso bekannt, dass Java-Code, einmal zu `class`-Dateien übersetzt (dank der *virtuellen Maschine*, der *JVM*), auf allen Plattformen läuft (»write once, run anywhere«). C++-Code müssen Sie für jede Plattform getrennt übersetzen, denn die Ausgabe des Compilers ist direkt ausführbarer Maschinencode (»write once, compile anywhere«). Das ist zwar dem Standard nach nicht Bedingung, doch normalerweise der Fall.
- **C++-`char` versus Java-`char`**
  Der C++-`char` ist normalerweise 8 Bit breit und kommt auf unterschiedlichen Systemen mal als `signed` und mal als `unsigned` vor. Ohne die entsprechende Auszeichnung können Sie sich deshalb nur auf einen Wertebereich von 0 bis 127 verlassen. Nur wenn Sie `signed char` schreiben, entspricht er dem Java-`byte`. Dem Java-`char` entspricht eher der C++-`short` und garantiert aber der `int16_t`. Letzterer muss nicht vorhanden sein, ist es aber de facto.
- **C++-`optional` versus Java-`Optional`**
  In Java nutzen Sie `Optional` oft als Teil der Java-Stream-API. In C++ entsprechen die Container der Java-Stream-API am ehesten. `optional` ist in C++ aber kein Container, und daher werden Sie `optional` auch nicht wie in Java verwenden.

- **Funktionsobjekte versus Lambdas**
  Auf den ersten Blick sind Java-Lambdas den C++-Lambdas ähnlich. Bei genauerem Hinsehen sind die anonymen Funktionsobjekte in C++ abgerundeter. In Java muss zur Aufrufzeit in der JVM noch etwas Aufwand betrieben werden, um dynamisch ein Funktionsobjekt zu erzeugen. In C++ hat der Compiler die Funktion komplett ausgelagert und ihr nur einen unsichtbaren Namen gegeben. Das Binden an lokale Variablen ist ähnlich, jedoch können Sie in C++ zwischen dem Binden als Wert oder als Referenz wählen.
- **Werte/Referenzen in C++ versus Java u. a.**
  Alles, was `Object` ist, ist in Java eine Referenz. In C++ ist alles ein Wert und wird für Parameter und Rückgaben kopiert. Ausschließlich bei spezieller Vorsorge mit `&` und `*` kann eine Funktion stattdessen explizit Referenzen und Zeiger anfordern.
- **Werte statt Zeiger werfen**
  In C++ schreiben Sie nicht `throw new X(...)`, sondern nur `throw X(...)`. Sollten Sie das nicht tun, handeln Sie sich mittelfristig Ärger ein. Sie müssten zum Beispiel die `Exception`-Objekte selbst wieder wegräumen. Und das geht gar nicht immer, wie zum Beispiel im `catch(...)` (*catch-all*). Und was ist mit einem Weiterwerfen? Sie sollten sich lieber auf die C++-Mechanismen verlassen, bei denen garantiert ist, dass eine `Exception`-Instanz so lange existiert, wie sie benötigt wird. Die Ausnahme von dieser Regel ist, wenn ein Framework von Ihnen explizites Wegräumen verlangt, weil es `Exception`-Instanzen als Zeiger wirft. Dann halten Sie sich natürlich an das, was das Framework vorschreibt. Lesen Sie in dessen Dokumentation nach, ob Sie im `catch` die `Exception` wegräumen sollen.
- **`const` versus `final`**
  Weil in Java alle Objekte Referenzen sind, bezieht sich das Java-`final` auch nur auf diese Referenz. Sie wissen sicherlich, dass Sie dessen Inhalt trotz `final` wild verändern können, wenn die Schnittstelle das zulässt (was zum Glück bei Dingen wie `Integer` nicht der Fall ist). Weil in C++ zunächst alles ein Wert ist, schützt ein `const` auch den Inhalt. Im Zusammenhang mit Referenzen und Zeigern haben Sie sogar noch mehr Kontrolle, wie Sie in Listing 3.6 und dessen Erklärung sehen werden.
- **Templates versus Generics**
  Beide verwenden spitze Klammern, und doch sind sie völlig unterschiedlich. In Java entsteht pro Generic lediglich eine einzige Funktion oder Klasse, Ihnen werden nur die Typumwandlungen abgenommen, zum Beispiel für die Rückgabe aus Methoden. Das Einzige, was Sie über den Typparameter wissen, ist, dass er entweder ein Object ist oder dass er ein bestimmtes Interface implementiert. Ein Generic gilt daher immer für eine bestimmte Gruppe von Objekten. In C++ ist ein Template stattdessen eine Schablone, die vom Compiler erst einmal nur parsebarer C++-Code sein muss, mehr nicht. Erst bei der Verwendung legen Sie den Typ der Parameter fest, und C++ setzt diese dann ein – und generiert in diesem Moment (zur Compilezeit) die wirkliche Funktion. Das können dann pro Typ unterschiedliche Funktionen werden.

- **Interface-Konzepte in C++ und Java**
  In C++ gibt es *multiple Vererbung* ohne Einschränkung. Und da dem Wort »multiple« nach Terry Pratchett nie etwas Gutes folgt,[1] wollten sich auch die Java-Erdenker diesen spezifischen Teufel nicht ins Haus holen. Aber ganz ohne geht es auch nicht, denn dann wäre Objektorientierung unmöglich. In Java gibt es deshalb die implementationslosen[2] `interface`-Deklarationen. Ein sinnvolles Konzept, das dem Missbrauch der multiplen Vererbung Einhalt gebietet und es schwerer macht, überkomplizierte Schnittstellenhierarchien zu designen. In C++ ist es andersherum: Die Einschränkung, sich nicht zu verdesignen, muss von den Entwicklern kommen, nicht von der Sprache. Als sinnvolle Einschränkung kann man damit anfangen, von Java die Idee zu übernehmen: Also zum Beispiel, *wenn* multipel geerbt wird, dann darauf zu achten, dass maximal eine der Elternklassen Implementierungen beisteuert und der Rest nur sogenannte *Signaturklassen* sind – also Klassen, die nur *pur virtuelle Methoden* enthalten. So nennt man in C++ *abstrakte Methoden*, also *virtuelle* Methoden, die statt einer Implementierung null zugewiesen haben.
- **Standardbibliothek und J2EE**
  Wenn Sie Java komplett installieren, bekommen Sie eine riesige API mit dazu, mit Bibliotheken, die so ziemlich alle Lebensbereiche abdecken. In C++ ist das nur eingeschränkt der Fall. Die Standardbibliothek gibt Ihnen zwar ein solides Fundament, bei vielen wichtigen Dingen sind Sie aber auf Drittanbieter angewiesen. Zum einen ist dies im Laufe der letzten Sprachversionen C++11, C++14 und C++17 jedoch immer seltener geworden, zum anderen werden erstaunlicherweise Java-Applikationen heutzutage in der Praxis dennoch immer größer und größer. Aber es stimmt: Netzwerkkommunikation und viele andere Dinge fehlen in der C++-Standardbibliothek sehr. Noch.

Methodendeklaration, Methodendefinition, vor Ort, getrennt und abstrakt

Für Ihr genaues Verständnis: In C++ definieren Sie eine Methode entweder direkt an ihrer Deklaration in der Klasse oder an anderer Stelle irgendwo in den Quellen. Im ersten Fall folgt die Definition dem Funktionskopf:

```
struct Demo {
    int x;
    virtual void func() { x = 12; }   // Definition vor Ort
}
```

Für Methoden mit mehr als ein paar Zeilen trennt man Deklaration und Definition. Der Funktionskörper steht dann außerhalb der Klassendefinition, gerne auch in einer anderen Datei:

1 *Guards! Guards!*, Terry Pratchett, »The noun doesn't matter after an adjective like multiple, nothing good ever follows multiple«

2 Default-Implementierungen von Java 8 einmal ignorierend

```
struct Demo {
    int x;
    virtual void func();   // Definition (wahrscheinlich) woanders
}
...
void Demo::func() { x = 12; }
```

Sie müssen aber nicht unbedingt eine Methodendefinition haben. Nur, wenn Sie die Methode auch verwenden, ist das wirklich nötig. Üblicherweise meldet der Linker einen Fehler beim Zusammenfügen des Gesamtprogramms. Es ist also durchaus in Ordnung, eine Methodendeklaration ohne Methodendefinition zu haben.

Schreiben Sie aber `= 0` bei der Methodendeklaration, dann ist dies auch gleichzeitig dessen Definition – nämlich als pur virtuelle, also abstrakte, Methode.

```
struct Demo {
    virtual void func() = 0;   // abstrakte Methode
}
```

Hier sind einige Beispiele, bei denen sich Java und C++ unterscheiden.

In Java benötigen Sie für das Erzeugen einer neuen Instanz ein `new`. Die Instanz wird auf dem Heap erzeugt:

Java

```
Data data = new Data(5);
Data mehr = new Data(6);
data = mehr;
mehr.value = 7;
System.out.println(data.value);  // auf jeden Fall 7
```

Listing 3.1 Ein kleines Java-Beispiel mit den allgegenwärtigen Objektreferenzen

`data` und `mehr` sind *Referenzen*. Tatsächlich sogar eher wie Zeiger in C++, denn sie können `data` auch den Nullzeiger `null` zuweisen, was bei Referenzen in C++ nicht ginge. Die Zuweisung `data = mehr` bewirkt, dass nun `data` und `mehr` die gleiche Instanz referenzieren. In C++ schreiben Sie stattdessen eher:

C++

```
Data data{5};
Data mehr{6};
data = mehr;
mehr.value = 7;
cout << data.value << '\n';  // immer noch 6
```

Listing 3.2 In C++ hat man statt Referenzen erst einmal Werte.

Hier sind `data` und `mehr` getrennte Werte. Unter der Voraussetzung, dass `mehr{6}` die Initialisierung `mehr.value=6` zur Folge hatte, ist auch nach der Zuweisung von `mehr.value=7` in `data.value` immer noch die `6` enthalten.

Und wenn Sie in C++ Heap und Zeiger mit `new` verwenden, um das Java-Verhalten nachzubilden, dann vergessen Sie nicht, dass Sie keine *Garbage Collection* haben, sondern den Heap irgendwie wieder freigeben müssen. Nutzen Sie dafür entweder eine Hilfsklasse wie `shared_ptr` (empfohlen) oder ein explizites `delete`:

C++
```
// https://godbolt.org/z/wADDsv
auto data = make_shared<Data>(5);
auto mehr = make_shared<Data>(6);
data = mehr;
mehr->value = 7;
cout << data->value << '\n'; // jetzt auch 7
```

Listing 3.3 Modernes C++ mit Heapspeicher nutzt Hilfsklassen wie »shared_ptr«.

Das ist schon sehr »modern«, Sie können es aber auch »purer« haben. Hier sehen Sie das Ganze ohne die von `make_shared` erzeugten Wrapper-Objekte.

C++
```
// https://godbolt.org/z/JM5wDG
Data* dataOwner = new Data(5);
Data* data = dataOwner;
Data* mehr = new Data(6);
data = mehr;
mehr->value = 7;
cout << data->value << '\n'; // jetzt auch 7
/* selbst wegräumen */
delete mehr;
delete dataOwner;
```

Listing 3.4 Ohne moderne C++-Mittel muss man besonders auf den Zeigerbesitz achten.

Aber, ach je, hier sehen Sie schon, welchen Ärger man sich einhandeln kann, wenn man selbst aufräumen muss. `delete data` wäre am Ende falsch gewesen, denn `data` zeigt auf `mehr`, und das ist schon weggeräumt. Die Zuweisung `data = mehr` ist der Bösewicht, der `data` verdeckt hat. Freigabe nicht mehr möglich. Nur deshalb gibt es `dataOwner`, um den ursprünglichen Zeiger später freigeben zu können.

Deshalb ist es in C++ nützlich, sich bei Zeigern das Konzept des *Besitzes von Zeigern* zu eigen zu machen: Rohe Zeiger gehören immer jemandem, der für das Wegräumen zuständig ist. Entweder ist das derjenige, der das Objekt mit `new` erzeugt hat, oder der Besitz wird transferiert. Das kann man »im Kopf« machen, sich von Werkzeugen wie der *Guideline Support Library* (*GSL*, siehe Kapitel 30, »Guter Code, 7. Dan: Richtlinien«) helfen lassen, selbst Wrapper-Klassen schreiben oder einfach immer die mitgelieferten Hilfsmittel wie `shared_ptr` und `unique_ptr` verwenden.

Hier wähle ich ein `final`, das in Java bedeutet, der Referenz kann nichts Neues zugewiesen werden:

Java

```
final Data data = new Data(5);
data.value = 7;          // das ist okay
data = new Data(6);      // dies verhindert final
```

Listing 3.5 In Java blockiert »final« nur die Referenz.

In C++ ist `const` viel differenzierter einzusetzen:

C++

```
// https://godbolt.org/z/7UpKnc
Data const * data = new Data(5);
data->value = 7;         // dieses const schützt Data
data = new Data(6);      // Zeiger neu zuweisen ist okay
Data * const mehr = new Data(8);
mehr->value = 9;         // jetzt okay
mehr = new Data(10);     // Referenz ist geschützt
```

Listing 3.6 »const« kann den Wert oder die Referenz schützen.

So lässt sich in C++ sowohl die Referenz oder der Zeiger als auch der Inhalt, also der Wert, schützen. Man kann auch beides kombinieren zu `Data const * const`. Was geschützt ist, erkennen Sie daran, *hinter* wem das `const` steht:

- `Data * const mehr` – Der Zeiger bzw. die Referenz ist geschützt.
- `Data const * data` – Der Wert `Data` ist geschützt.
- `const Data * data` – Hierbei handelt es sich um eine alternative Schreibweise für `Data const * data`.
- `Data const * const mehr` – Sowohl Wert also auch Zeiger sind geschützt.

Außerdem gibt es noch Varianten von `const`: `const`-Methoden dürfen ihr eigenes Objekt nicht verändern, `constexpr` erzwingt, dass der Ausdruck schon zur Übersetzungszeit vom Compiler berechnet werden kann.

Kapitel 4
Die Grundbausteine von C++

Kapiteltelegramm

- `main`
  Der Einstiegspunkt in jedes Programm
- `#include`
  Einbinden anderer Programmteile und Bibliotheken
- **Variable**
  Name für einen Speicherbereich, der einen Wert aufnehmen kann
- **Initialisierung**
  Dies ist der Wert, den eine Variable bei ihrer Entstehung haben soll. Bei eingebauten Typen ist die Initialisierung bei der Definition besonders wichtig, da eine Variable sonst einen undefinierten Zustand erhält.
- **Zuweisung**
  Die Veränderung des Inhalts einer Variablen mittels = (engl. *Assignment*)
- `return`
  Das Verlassen einer Funktion; in `main` das Ende des Programms
- **Kommentar**
  Anmerkung zum Programmcode, die der Compiler nicht auswertet
- **Anweisung**
  Ein Programm ist die sequenzielle Abarbeitung unterschiedlicher Anweisungen (engl. *Statements*).
- **Ausdruck**
  Eine Folge von Operationen auf Operanden für Zuweisungen etc. (engl. *Expression*)
- **Block**
  Eine Gruppe von Anweisungen zwischen geschweiften Klammern
- **Typ oder Datentyp**
  Für den Compiler hat jeder Ausdruck einen Typ.
- **Standardbibliothek**
  Teil des C++-Standards und damit mit dem Compiler geliefert. Alles, was nicht pure Syntax oder Semantik oder ein eingebauter Datentyp ist, liefert die Standardbibliothek.
- **Eigene Funktion**
  Mit einer eigenen Funktion lagern Sie Code an eine andere Stelle aus. Dies ist die Basis für Übersichtlichkeit und Wiederverwendbarkeit.

- **Funktionsparameter**
  Gibt dem an eine Funktion übergebenen »Ding« lokal einen eigenen Namen innerhalb einer Funktion.
- **Seiteneffekt-Operator**
  Operator, der den Wert einer Variablen verändert, aber keine Zuweisung ist
- **`int` und `bool`**
  Zwei elementare (eingebaute) und einfache Typen
- **Vereinheitlichte Initialisierung**
  Eine eindeutige Schreibweise für den ersten Wert einer Variablen oder eines Default-Parameters (engl. *Unified Initialization*)
- **Token**
  Für den Compiler die kleinsten Bausteine des Programmtexts
- **Bezeichner**
  Namen von Programmelementen (engl. *Identifier*), also Variablen, Typen, Funktionen etc.
- **`for`-Anweisung**
  Dies ist eine Möglichkeit, um die Wiederholung von Anweisungen zu implementieren.
- **`if`-Anweisung**
  Diese verwenden Sie, um Verzweigungen umzusetzen.
- **Zuweisung**
  Ein sehr spezieller und nützlicher *Ausdruck*, um den Wert einer Variablen zu ändern
- **Operator**
  Meist ein Symbol, das zwischen zwei Operanden steht (oder vor einem); funktioniert wie eine Funktion mit den Operanden als Argumente
- **Operand**
  Argument für einen Operator
- **Arithmetischer Operator**
  Dient zum klassischen Rechnen mit `+`, `-`, `*`, `/`, `%` sowie dem bitweisen Rechnen mit `|`, `&`, `~`, `<<` und `>>`
- **Relationaler Operator**
  Größer-als, kleiner-als, gleich, Kombinationen davon und ungleich: `>`, `<`, `==`, `<=`, `>=`, `!=`.
- **Logischer Operator**
  Verknüpft boolesche Werte: `&&`, `||` und `!`
- **Zuweisungsoperator oder zusammengesetzte Zuweisung**
  Der Zuweisungsoperator ist das Gleichheitszeichen `=`. Eine zusammengesetzte Zuweisung kombiniert `=` mit einem arithmetischen Operator.
- **Binärsystem**
  Das Stellenwert-Zahlensystem des Computers mit Nullen und Einsen
- **Eingebauter Typ**
  Ein Typ, der Ihnen ohne `#include` zur Verfügung steht
- **Ganzzahltyp**
  Einer der eingebauten Typen `short`, `int`, `long`, `long long`; jeweils `signed` oder `unsigned`

- **Fließkommatyp**
  Einer der eingebauten Typen `double`, `float` und `long double`
- **Zeichentyp**
  Meistens `char`, aber mit internationalen Zeichen auch `wchar_t`, `char16_t` und `char32_t`; alles eingebaute Typen
- **Zeichenkette**
  Als Literal `const char[]`, zusammen mit C oft `const char*`, in C++ `string`; jeweils auch eventuell mit einem der anderen Zeichentypen
- **Wahrheitswerttyp**
  Der eingebaute Typ `bool` mit seinen Literalen `true` und `false`
- **Typinferenz mit `auto`**
  Bei der Definition einer Variablen den Typ vom Compiler aus dem Typ des Ausdrucks der Initialisierung ermitteln lassen
- **Überlauf**
  Der Versuch, den Wert eines Typs über seinen Wertebereich hinaus zu verändern; ein Überlauf kann sowohl im positiven als auch im negativen Bereich passieren.
- **Literal**
  Ein im Quelltext direkt genannter Wert
- **`using` und `typedef`**
  Definieren einer Typabkürzung

In diesem Kapitel machen wir einen ganz schnellen »Rundflug« über ein einfaches C++-Programm. Dadurch lernen Sie die wichtigsten Elemente kennen und haben ein besseres Verständnis für die Dinge, die ich in den nächsten Kapiteln erkläre.

Hier ist also ein einfaches C++-Programm.

```
// https://godbolt.org/z/axyRK3
#include <iostream>                          // Module/Bibliotheken einbinden
int main()                                   // main() ist der Beginn des Programms
{
    int wert = 100;                          // Variable mit Anfangswert
    std::cout << "Teiler von " << wert << " sind:\n";     // Ausgabe von Text
    for(int teiler=1; teiler <= wert; teiler = teiler+1) // Schleife von 1 bis 100
    {                                        // hier beginnt der Wiederholungsteil
        if(wert % teiler == 0)               // Test für eine bedingte Ausführung
            std::cout << teiler << ", ";     // nur bei positivem Test
    }                                        // Ende der Schleife
    std::cout << "\n";                       // einmalige Ausgabe
    return 0;                                // bedeutet in main() Programmende
}                                            // Ende von main()
```

Listing 4.1 Ein sehr einfaches C++-Programm

Wenn Sie dieses Programm übersetzen und laufen lassen, erhalten Sie die Ausgabe

```
Teiler von 100 sind:
1, 2, 5, 10, 20, 25, 50, 100,
```

auf dem Bildschirm. An diesem einfachen Programm können Sie schon viele grundlegende und wichtige Dinge von C++ sehen.

4.1 Kommentare

Wie Sie sehen, habe ich hier Programmtext und erklärende Worte gemischt. Die Zeilen beginnen immer mit Programmtext, dann folgt manchmal ein Doppel-Schrägstrich //, und dann kommen die erklärenden Worte – der *Kommentar*. In C++ können Sie hinter // beliebigen Text schreiben, der Compiler ignoriert diesen (oder, um genau zu sein, interpretiert ihn ähnlich wie ein Leerzeichen) bis auf wenige Ausnahmen. So können Sie anderen Programmierern, sich oder der Nachwelt Ihre Intentionen mitteilen, die zu der aktuellen Programmzeile führten.

4.2 Die »include«-Direktive

Die allererste Zeile des Beispiels lautet:

```
#include <iostream>
```

Mit `#include` machen Sie dem Compiler bekannt, dass Sie Elemente eines Moduls in dieser Datei verwenden wollen. Der Name zwischen den Klammern ist der Name einer Headerdatei, in der sich die Deklarationen jenes Moduls befinden.

4.3 Die Standardbibliothek

Das spezielle Modul `iostream` binde ich ein, weil sich darin`std::cout` befindet. `cout` benötigt das Programm, um die Bildschirmausgaben zu erzeugen. Das Modul `iostream` ist Teil der *Standardbibliothek* und wird mit dem Compiler mitgeliefert.

Alle Namen der Standardbibliothek beginnen mit `std`, gefolgt vom *Bereichsauflösungsoperator* `::` (engl. *Scope Resolution Operator*). Das ist ein hässlicher, wenn auch präziser Begriff, den niemand verwendet – der Name Doppel-Doppelpunkt (engl. *Double Colon*) tut es auch.

Ich gehe später auf die Möglichkeiten, den Bezeichner `std` zu verwenden, noch genauer ein. Mit der Standardbibliothek an sich beschäftigt sich ein Großteil des Buchs.

4.4 Die Funktion »main()«

Betrachten wir nun die Zeile, in der `main` steht:

```
int main()
```


Dies definiert eine *Funktion* mit einem besonderen Namen. Die `main`-Funktion ist immer der Einstiegspunkt in ein C++-Programm – es geht nicht ohne, und es kann nie zwei geben. Wenn Ihr System das Programm ausführt, dann wird `main()` aufgerufen.

Ansonsten bietet eine Funktion in C++ die Möglichkeit, dass Sie von einer anderen Stelle des Programms zu dieser Funktion springen und später wieder an den Aufrufort zurückkehren. Funktionen können Argumente entgegennehmen – das sind die *Funktionsparameter* – und ein Ergebnis zurückgeben (siehe Kapitel 7, »Funktionen«).

Konkret lesen Sie die Definition dieser `main`-Funktion so:

- `main` soll eine Zahl zurückgeben – einen Wert vom Typ `int`, um genau zu sein.
- Der Name der Funktion ist `main`.
- Das leere runde Klammerpaar `()` bedeutet, dass die Funktion keine Parameter erhält.
- Dann folgt, was die Funktion eigentlich macht. Dieser *Funktionskörper* steht immer zwischen zwei geschweiften Klammern {…}.

Andere Funktionen als `main` könnten beliebige andere Typen zurückgeben.

Welchen Wert `main` zurückgibt, bestimmt das erste `return`, dem der Computer beim Programmablauf begegnet. Hier wird das in jedem Fall `return 0` sein. Der Rückgabewert ist also immer `0`.

Je nach Betriebssystem kann der Rückgabewert ausgewertet werden. Soll Ihr Programm Argumente auf der Kommandozeile bekommen können, wären die runden Klammern für die Parameter nicht leer. Sie sehen später, was Sie dafür anstellen müssen.

4.5 Typen

In C++ hat fast alles einen Typ, zum Beispiel Variablen und Zwischenergebnisse. Der Typ legt fest, welche Eigenschaften das Konstrukt hat und welche Werte es aufnehmen kann.

In Listing 4.1 wird nur `int` als Typ konkret genannt. Alles andere erschließt sich der Compiler selbst. Im Programm werden mit `int wert` und `int teiler` zwei *Variablen* mit diesem Typ eingeführt. Immer, wenn Sie sich im Verlauf Ihres Programms auf diese Variablen beziehen, müssen Sie deren Typ `int` berücksichtigen. Auf die genauen Eigenschaften von `int` werde ich noch eingehen. Für den Moment ist es ausreichend, zu wissen, dass `int` für eine »Ganzzahl« steht.

4.6 Variablen

Eine *Variable* ist der Name für einen Speicherbereich, der einen Wert aufnehmen kann. Ja, in C++ muss eine Variable immer einen Typ haben. Wenn Sie eine Variable das erste Mal verwenden, müssen Sie dem Compiler ihren Typ mitteilen. Der Typ begleitet die Variable, solange sie lebt, und kann nicht mehr geändert werden.

Im Programm verwende ich zwei *Variablen*. `wert` repräsentiert die Zahl, deren Teiler ich ausgebe, und `teiler` ist die Zahl, mit der ich prüfe, ob sich eine Division ohne Rest durchführen lässt. Zunächst definiere ich `wert`. Ab dem Zeitpunkt kann ich `wert` verwenden, was für die Ausgabe auch gleich geschieht:

```
int wert = 100;                                    // Variable mit Anfangswert
std::cout << "Teiler von " << wert << " sind:\n";  // Ausgabe von Text
```

Weil `wert` vom Typ `int` ist, können Sie diese Variable nur für Ganzzahlen verwenden – das heißt sie in Berechnungen verwenden oder verändern.

Die andere Variable ist `teiler` in der `for`-Schleife:

```
for(int teiler = 1; teiler <= wert; teiler = teiler+1) // Schleife von 1 bis 100
```

Für sie gilt Ähnliches wie für `wert` – neben dem Namen gibt es zwei weitere Unterschiede:

- `teiler` wird tatsächlich im Programmablauf verändert: Bei `teiler = …` wird ihr eine neue Zahl zugewiesen.
- Sie ist nur innerhalb von `for` bekannt.

Der *Gültigkeitsbereich* einer Variablen (engl. *Scope*) beschränkt sich auf ihren Block, danach ist sie buchstäblich »weg«. Und zwar so »weg«, dass Sie außerhalb des `for` eine *neue* andere Variable `teiler` definieren könnten. Die kann dann auch einen ganz anderen Typ haben. In Listing 8.4 (Seite 199) finden Sie ein Beispiel.

4.7 Initialisierung

Das Gleichheitszeichen `=` erfüllt im Beispielprogramm zwei Zwecke, die oft miteinander vermischt werden. Weil die Unterscheidung aber so wichtig ist, möchte ich Sie schon früh dafür sensibilisieren.

Sie sehen im Beispiel die folgenden Gleichheitszeichen:

```
int wert = 100;
int teiler = 1;
teiler = teiler+1;
```

Die ersten beiden Zeilen sind jeweils die *Initialisierung* einer im gleichen Atemzug definierten Variablen. Dieser Zeitpunkt, zu dem Sie auch ihren Typ festlegen, ist ihre *Deklaration*. Nur bei der Deklaration können Sie etwas initialisieren.

In der letzten Zeile ist die Variable schon deklariert. Somit weisen Sie einer bestehenden Variablen einen *neuen* Wert zu – daher sprechen wir von einer *Zuweisung*. Bei einer Zuweisung können Sie den Typ der Variablen nicht ändern. Sind die Typen unterschiedlich, kann der Compiler in gewissen Grenzen eine Konvertierung vornehmen.

Zuweisung versus Initialisierung

Das Gleichheitszeichen bei der Deklaration ist immer eine *Initialisierung*.

Nur wenn die Variable woanders deklariert wurde, ist es eine *Zuweisung*.

4.8 Ausgabe auf der Konsole

Mit `#include <iostream>` haben Sie den Teil der Standardbibliothek importiert, der für die Ein- und Ausgabe zuständig ist. Die Ausgabe auf die Konsole geschieht mittels des Operators `<<`.

```
std::cout << teiler << ", ";
```

Links sehen Sie mit `std::cout` die aus `<iostream>` stammende Variable, die für die Ausgabe auf der Konsole steht. Rechts von jedem `<<` stehen die Dinge, die Sie ausgeben wollen. Wie Sie sehen, können Sie `<<` ähnlich verketten wie ein normales Plus `+` und somit mehrere Dinge nacheinander ausgeben.

Manches Auszugebende lässt sich schwer in Quellcode schreiben. Dafür sieht C++ dann besondere Mechanismen vor. Eine Möglichkeit ist das *Escapen* bestimmter Zeichen mit einem *Backslash* \ (umgekehrter Schrägstrich). Wollen Sie einen Zeilenvorschub ausgeben, schreiben Sie zum Beispiel `"\n"` in den Quellcode.

4.9 Anweisungen

Die geschweiften Klammern {…} von `main()` halten eine Gruppe von *Anweisungen* zusammen – sie definieren einen *Anweisungsblock*. Sie bilden die Begrenzung dessen, was für `main()` ausgeführt wird:

```
int main()
{
    ...
}
```

Dazwischen stehen Anweisungen, die nacheinander ausgeführt werden (engl. *Statements*). Anweisungen sind wichtige Grundelemente in C++, und es gibt unterschiedliche Arten. Zu erkennen, was eine Anweisung ist und welcher Art sie ist, wird Sie mit C++ schnell

voranbringen. In den nächsten Abschnitten werden Sie alle kennenlernen. An dieser Stelle zeige ich Ihnen, welche Anweisungen Sie in Listing 4.1 finden:

- Die *Deklaration* `int wert = 100;` macht die Variable `wert` bekannt und initialisiert sie mit einem Anfangswert – zusammengenommen manchmal *Initialisierungsanweisung* genannt.
- Bei `cout << "Teiler von " << wert << " sind:\n";` handelt es sich um einen *Ausdruck*, der etwas auf der Konsole ausgibt.
- Dann folgt eine *for-Schleife*. Sie wird verwendet, um andere Anweisungen wiederholt auszuführen. Ich gehe später genauer auf die `for`-Anweisung ein, hier achten Sie bitte auf die Besonderheit, dass der Teil, der wiederholt werden soll, wieder in *geschweiften Klammern* {…} hinter dem `for` steht.
- Denn die beiden Klammern, die zum `for` gehören, sind mit ihrem Inhalt eine *zusammengesetzte Anweisung* oder auch ein *Anweisungsblock*. Darin ist wieder eine Serie von Anweisungen enthalten, die von den umschließenden Klammern zusammengehalten wird. Diese Gruppierung von Anweisungen hat in mehrerlei Hinsicht eine besondere Bedeutung. Einerseits können sie so gemeinsam durch die `for`-Schleife wiederholt werden, und andererseits bildet diese Gruppierung einen *Gültigkeitsbereich* für darin enthaltene Variablen.
- Bei `if(wert % teiler == 0)…` handelt es sich um eine `if`-Anweisung, eine *Verzweigung*. Es wird eine Bedingung getestet, und die dann folgende *Anweisung* `std::cout << teiler << ", ";` wird nur ausgeführt, wenn diese Bedingung wahr ist. Wie bei der `for`-Schleife des Beispiels hätten wir hier auch einen Anweisungsblock in {…} folgen lassen können.[1] Zur Demonstration folgt dem `if` nur eine einzelne Anweisung. So konnte ich mir die umgebenden {…} für einen Anweisungsblock sparen.
- Die *Return-Anweisung* `return 0;` schließt diese Aufzählung ab.

Anweisungen setzen sich aus *Ausdrücken* zusammen. Ein Ausdruck kann zur Laufzeit als Ergebnis einen Wert haben. Zur Übersetzungszeit kennt der Compiler für jeden Ausdruck dessen Typ. Wir gehen später sehr detailliert auf Ausdrücke ein. Hier nur einige Beispiele aus dem Programm:

- `wert % teiler` berechnet den Modulo, also den Rest einer Division.
- `… == 0` prüft die Gleichheit zweier Werte.
- `teiler+1` ist das Ergebnis einer Addition.
- `teiler = …` – auch eine Zuweisung ist ein Ausdruck.

1 Das wäre guter Stil gewesen.

4.10 Ohne Eile erklärt

Nachdem Sie im vorigen Abschnitt ein Beispielprogramm von vorne bis hinten betrachtet haben, gehe ich jetzt etwas mehr ins Detail. Sie sehen die ersten formalen Definitionen und lernen, die wichtigen Sprachelemente zu erkennen.

Bauen wir Listing 4.1 ein wenig aus und werfen wir einen genaueren Blick auf die Elemente, die wir schon kennen:

```
// https://godbolt.org/z/9hqfaWjYG
#include <iostream>                          // für std::cin, std::cout, std::endl
#include <string>                            // std::stoi

void berechne(int n) {                       // eine eigene Funktion
    using namespace std;                     // für std::cout und std::endl
    /* Teiler ausgeben */
    cout << "Teiler von " << n << " sind:\n";
    if(n == 0) { cout << "0\n"; return; }    // 0 ist Teiler von 0
    for(int teiler=1; teiler <= n; ++teiler) {   // statt teiler=teiler+1
        if(n % teiler == 0)
            cout << teiler << ", ";
    }
    cout << endl;
}
int main(int argc, const char* argv[]) {     // Argumente für main
    /* Zahl ermitteln */
    int wert = 0;
    if(argc<=1) {
        std::cout << "Geben Sie eine Zahl ein: ";
        std::cin >> wert;                    // in Variable wert lesen
        if(!std::cin) {                      // prüfen, ob lesen klappte
            return 1;                        // Fehler bei Benutzereingabe
        }
    } else {
        wert = std::stoi(argv[1]);
    }
    berechne(wert);                          // Funktionsaufruf
    return 0;
}
```

Listing 4.2 Dieses Programm fragt seine Benutzer nach einer Zahl.

Zu Beginn sehen Sie einiges Neues: Ich habe nun eine eigene Funktion `berechne` eingeführt. Sie bekommt einen *Parameter* vom Typ `int`, den ich innerhalb der Funktion unter dem Namen `n` anspreche.

Auch `main()` hat auf einmal zwei Parameter, nämlich `int argc` und `const char* argv[]`. Damit können Benutzer das Programm schon auf der Kommandozeile mit der Zahl aufrufen, für die die Berechnung durchgeführt werden soll. Dabei enthält `argc` die Anzahl der Argumente, die Sie mit `argv[…]` abfragen können. Da `argv[0]` immer den Namen des aufgerufenen Programms enthält, steht der erste Parameter in `argv[1]` etc.

Wenn das Programm ohne Argumente aufgerufen wird, muss bei bei `std::cin >> wert` etwas eingegeben werden.

`std::stoi(argv[1]);` wandelt das erste Mal einen Datentyp um, nämlich hier einen textuellen Wert (`const char*`) in eine Zahl (`int`). Dann rufe ich mit `berechne(wert)` die eigene Funktion `berechne` mit der Variablen `wert` auf.

4.10.1 Leerräume, Bezeichner und Token

Neben den für den Compiler (so gut wie) bedeutungslosen Kommentaren gibt es noch das Leerzeichen, den Tabulator und den Zeilenwechsel, die der Compiler beim Lesen des Quellcodes stark vereinfacht: Alle diese *Leerräume* (engl. *Whitespaces*) werden »kollabiert« und jeweils nur noch als ein »Trenner« betrachtet. Es ist somit egal, ob Sie `return 0;` oder `return  0  ;` schreiben, Zeilenwechsel oder gar Leerzeilen einbauen.

Eine kleine Anmerkung zum Zeilenwechsel: Unter diesen Oberbegriff fallen die verschiedenen Varianten, die auf den unterschiedlichen Betriebssystemen existieren. Wenn Sie im Editor einen Zeilenwechsel sehen, landen in der Textdatei unter Windows und Linux unterschiedliche Bytesequenzen, und bei einem alten Mac war es noch eine andere. Während Linux den Wert `13` (CR, »Carriage Return«, Wagenrücklauf) wegspeichert, sind es unter Windows die beiden Werte `13` und `10` (zusätzlicher LF, »Line Feed«, Zeilenvorschub). Die meisten Editoren kommen heute mit beidem klar. Aber dieser Unterschied ist der Grund dafür, dass Sie beim programmgesteuerten Öffnen von Dateien immer angeben müssen, ob Sie eine Binärdatei oder eine Textdatei öffnen wollen.

Letztendlich ist nur wichtig, dass der Compiler die kleinsten Programmeinheiten sauber voneinander getrennt bekommt – die *Token*. Handelt es sich um Namen (von Variablen oder Funktionen etc.), dann ist die Grenze klar, nämlich dort, wo der Name aufhört: Das erste Zeichen, das nicht für einen Namen taugt, ist dann die Grenze. In Namen (oder genauer: *Bezeichnern*) kommen Buchstaben und Ziffern sowie der Unterstrich _ vor. Nur eine Ziffer darf nicht am Anfang stehen. Somit sind `hallo`, `Tag`, `GoodDay`, `w3lcOme` und `moin_moin` alles mögliche Bezeichner. Dagegen sind `Hanni-Nanni`, `3Fragezeichen`, `Fuenf Freunde` und `Tim&Struppi` nicht als einzelne Namen verwendbar. Sollten Sie Umlaute wie in `Bärenbrücke` verwenden wollen, prüfen Sie, ob Ihr Compiler dies kann – hier hat der Standard Spielraum gelassen. Sollte Ihr Programm von unterschiedlichen Compilern übersetzt werden müssen, verzichten Sie besser auf Umlaute im Programmtext.

Neben dem Leerraum kann ein Wort auch von Sonderzeichen wie Klammern oder Satzzeichen begrenzt sein. Die meisten dieser Zeichen sind jeweils ein Token, und Sie kön-

nen beliebig Leerräume zwischen diesen Token einstreuen, ohne die Bedeutung des Programms zu verändern. Es gibt allerdings ein paar Kombinationszeichen, die nur dann ihre Bedeutung erhalten, wenn sie zusammengeschrieben werden – und so als ein Token gelten. Achten Sie hauptsächlich auf `>>` und `<<` sowie auf `++` und `--`, mit denen ursprünglich Rechenoperationen durchgeführt wurden, die in C++ aber teilweise weitere Bedeutungen bekommen haben. Geht es um Wahrheitswerte, werden Ihnen `&&`, `||` sowie `==` und `!=` begegnen. Daneben sind `->` und `::` noch wichtig, mit denen Sie in verschachtelten Datenstrukturen navigieren. Alle anderen (Sonder-)Zeichen sind jeweils ein eigene Token. Das sind also zum Beispiel `+ - * / = % , . ( ) [ ] { } < > : ;`.

Der Hauptgrund dafür, dass Sie wissen sollten, wo die Token in Ihrem Quelltext sind, ist, dass Sie Leerräume weglassen und einfügen können, ohne den Sinn des Programms zu verändern. Denn zwischen Token können Sie beliebige Leerzeichen, Tabulatoren und Zeilenwechsel einfügen.

Ausnahmen sind die *Literale*: Mit diesen schreiben Sie einen festen Wert direkt in den Quellcode. Das kann eine Ganzzahl wie `100`, eine Fließkommazahl wie `99.95` oder ein Text wie `"Donald E. Knuth"` sein – und zu all diesen gibt es jeweils noch Varianten in der Schreibweise. Ich gehe in der Besprechung der jeweiligen Datentypen in Abschnitt 4.12, »Eingebaute Datentypen«, detailliert auf deren Literale ein. Literale können teilweise so aussehen, als bestünden sie aus mehreren Token, zählen aber als ein einziges.

```
// https://godbolt.org/z/ND9Xgk
#          include <iostream>        // # muss am Zeilenanfang stehen
int             main(
    ){
    std::cout
<<"Dies ist "
        "Text mit <Klammern>\n"      // String-Literal unterbrochen durch neue Zeile
        ;

    /*Typ:*/ int
    /*Variable:*/ ein_Wert
    /*Init:*/ = 100;                 // innere Kommentare

std::cout<<ein_Wert<<"\n";}          // keine Leerzeichen
```

Listing 4.3 Ein sehr außergewöhnlich formatiertes Stück Quellcode

Knifflig wird es vor allem, wenn etwas aus mehreren Bezeichnern zusammengesetzt wird. So gibt es zum Beispiel in `std::sin()` den *Bereichsauflösungsoperator* `::`. Mit Templates gibt es Paare von spitzen Klammern `<…>`, zum Beispiel in `numeric_limits<int>::max()` oder `map<int,string>`. Alle Elemente sind einzelne gültige Bezeichner, aber erst zusammen ergeben sie eine Einheit.

4.10.2 Kommentare

Sie haben den Kommentar mit // schon kennengelernt. Sollte der Raum bis zum Ende der Zeile einmal nicht reichen, können Sie einen Kommentar auch über mehrere Zeilen gehen lassen, indem Sie ihn mit /* beginnen und mit */ beenden. Zum Beispiel so:

```
int main() {
    /* Mein erstes Programm. Es wurde
       geschrieben von Max Muster.*/
    return /* Die Null des Erfolgs */ 0;
}
```

Listing 4.4 Kommentare mit »/*« und »*/« können über mehrere Zeilen gehen oder eine Programmzeile auch unterbrechen.

Und weil ein solcher Kommentar durch ein */ begrenzt ist, kann es mit dem Programmcode danach in derselben Zeile weitergehen, wie Sie in der Zeile mit `return 0;` sehen.

Innerhalb des Kommentars dürften auch // vorkommen. Wenn Sie hauptsächlich // für Kommentare in Ihrem Programm verwenden, können Sie auf diese Weise einfach ganze Codeblöcke deaktivieren, indem Sie den Bereich mit /* und */ umschließen – und durch das Entfernen wieder aktivieren. Der Compiler würde auch Kombinationen wie `/* /*` oder `// */` schlucken, doch vermeiden Sie besser alle derartigen Einbettungen: Das verwirrt den Leser und Sie selbst.

In den Beispielen dieses Buchs verwende ich /*…*/-Kommentare bevorzugt, wenn ich den Programmcode selbst durch Text ergänzen will – ohne Bezug auf dieses Buch. So leite ich zum Beispiel in Listing 4.2 mit

```
/* Teiler ausgeben */
...
```

und

```
/* Zahl ermitteln */
...
```

Programmabschnitte ein.

4.10.3 Funktionen und Argumente

Lassen Sie sich nicht davon verwirren, dass die Variable beim Aufruf `wert` heißt, aber in der Funktion `berechne` dann `n` genannt wird. Wie in der Mathematik werden Funktionen so definiert, dass sie ihre *Parameter* mit Namen versehen, unter denen Sie sie dann verwenden – womit sie jedoch aufgerufen werden, ist eine völlig andere Sache. Sie können in der Mathematik ja auch Funktionen haben wie $f(x) = x^2 + 2$, sin(x) oder

$$\text{Zinseszins(Euro, Jahre, Zins)} = \text{Euro} \times \left(1 + \frac{\text{Zins}}{100}\right)^{\text{Jahre}}$$

die alle ihre eigenen Namen vergeben, um sie in der Formel zu verwenden. Beim Aufruf müssen Sie diese Namen dann nicht verwenden:

- $g(z) = z^3 - 2 \times f(z)$ – für eine schöne Kurve – `z` wird zu `x` in `f(x)`.
- $\sin(\pi)$ – hier ist `x` die Zahl π.
- Zinseszins(1000, 12, 3) – Euro bekommt in der Funktion den Wert 1000, Jahre erhält 12 und Zinsen 3.

Zurück zu der C++-Funktion aus Listing 4.2. Zur Erinnerung, sie sieht wie folgt aus:

```
// https://godbolt.org/z/ta9nsE
void berechne(int n) {                          // eine eigene Funktion
    using namespace std;                        // für std::cout und std::endl
    /* Teiler ausgeben */
    cout << "Teiler von " << n << " sind:\n";
    for(int teiler=1; teiler <= n; ++teiler) {  // statt teiler=teiler+1
        if(n % teiler == 0)
            cout << teiler << ", ";
    }
    cout << endl;
}
```

Listing 4.5 Eine eigene C++-Funktion

In der Funktion selbst habe ich im Vergleich zu Listing 4.1 ab `/* Teiler ausgeben */` alles beinahe identisch gelassen. Sie finden eigentlich nur im Inkrement-Teil der `for`-Schleife mit `++teiler` eine alternative Schreibweise zu `teiler = teiler + 1`. Bisher haben Sie die *Zuweisung* verwendet. Hier wird mit `++teiler` ein *Seiteneffekt-Operator* verwendet, um den Wert von `teiler` zu verändern.

4.10.4 Seiteneffekt-Operatoren

In Listing 4.5 habe ich `++teiler` verwendet, um die Variable um eins zu erhöhen.

Es handelt sich hierbei um den *Präfixoperator* `++`, der im Normalfall eine Variable um eins erhöht (»inkrementiert«). Das ist ein *Ausdruck*, dessen Wert der um eins erhöhte Wert der Variablen ist. Das Ergebnis des Ausdrucks ist im Update-Teil der `for`-Schleife unwichtig, doch Sie können den Wert innerhalb eines komplexeren Ausdrucks verwenden, wie bei:

```
// https://godbolt.org/z/aeGy2i
#include <iostream> // cout
int main() {
    int basis = 2;
    int index = 10;
    int zahl = 3 * (basis + ++index) - 1;  // zuerst wird index erhöht
    std::cout << zahl << '\n';             // Ausgabe: 38
}
```

Listing 4.6 Präfixoperatoren werden vor der Berechnung ausgeführt.

Hier wird `index` zunächst um eins erhöht und der neue Wert `11` in der weiteren Berechnung verwendet.

Wenn man den Wert einer Variablen innerhalb eines Ausdrucks verändert, ist dies ein *Seiteneffekt*. Verändern Sie niemals zweimal dieselbe Variable innerhalb einer Anweisung mit Seiteneffekten!

```
int wert = 0;
std::cout << ++wert << ++wert << ++wert;
```

Der Compiler wird es zulassen, aber das Programm kann auf unterschiedlichen Systemen unterschiedliche Ergebnisse haben. Auf einem Mac habe ich hier mal »123« bekommen, auf einem Linux-System »321« – und beides ist richtig.

Wie Sie in Kapitel 9, »Ausdrücke im Detail«, sehen werden, können Sie sich auf die Reihenfolge der Auswertung von Ausdrücken nicht immer verlassen. Daher dürfen Sie solche Konstrukte nicht verwenden. In C++ wird diese kleine Unannehmlichkeit in Kauf genommen, da es dem Compiler Flexibilität erlaubt, die zu besserer Performance führt.

In Tabelle 4.1 habe ich die Seiteneffekt-Operatoren aufgeführt.

Ich empfehle, dass Sie, wenn Sie die Wahl haben, die ersten beiden Varianten bevorzugen (»Präfixoperatoren«). Denn im Allgemeinen sind die letzten beiden Varianten (»Postfixoperatoren«) aufwendiger, weil der Computer sich den alten Wert in einer temporären Variablen merken muss.

Operator	Beschreibung
`++var`	Wie gesehen, erhöht `var` um eins und liefert den neuen Wert zurück.
`--var`	Verringert `var` um eins und liefert den neuen Wert zurück.
`var++`	Erhöht `var` um eins und liefert den alten Wert zurück.
`var--`	Verringert `var` um eins und liefert den alten Wert zurück.

Tabelle 4.1 Präfix- und Postfix-Seiteneffekt-Operatoren

Daneben gibt es noch die Familie der *Seiteneffekt-Zuweisungen*. Alle arithmetischen Operationen stehen in diesen Varianten zur Verfügung – alle sind Ausdrücke und haben somit einen Wert, den Sie als Teil eines größeren Ausdrucks verwenden können. Üblicherweise werden sie jedoch nicht als Ausdruck, sondern wie eine einfache Zuweisung als Anweisung verwendet:

```
// https://godbolt.org/z/pXfctH
#include <iostream>
int main() {
    int var = 10;
    var += 2;
```

```
    var *= 3;
    var /= 4;
    var -= 5;
    std::cout << var << "\n"; // ergibt 4
}
```


Neben -, +, * und / gibt es noch einige andere, auf die ich in Abschnitt 4.11, »Operatoren«, näher eingehe.

4.10.5 Die »main«-Funktion

Lassen Sie mich kurz das kleinere Beispielprogramm noch weiter vereinfachen, sodass fast nur noch `main` und `return` übrig bleiben:

```
// https://godbolt.org/z/oAbaPQ
int main() {
    if(2 < 1) return 1;     // ein return
    return 0;               // anderes return
}                           // Ende von main
```

Listing 4.7 Ein Programm, das nur aus »main« und »return« besteht

Nun sehen Sie, dass `main` auch nur eine Funktion ist – aber eine *spezielle* Funktion: Jedes ausführbare C++-Programm muss genau eine `main`-Funktion haben, denn hier beginnt die Ausführung des Hauptteils dessen, was Ihr Programm tun soll.

Ohne `main()` wissen Compiler und Computer nicht, wo sie in das Programm einsteigen sollen. Nur wenn Sie ein Modul oder eine Bibliothek schreiben, werden Sie kein `main()` haben. Fügen Sie anschießend alle Bausteine zu einem fertigen Programm zusammen, enthält einer der Bausteine – und zwar *genau einer* – ein `main()`.

Vereinfacht gesagt, beginnt die Ausführung Ihres Programms mit der ersten Zeile von `main()`, direkt nach der öffnenden geschweiften Klammer `{`. Es gibt jedoch allerlei Aufgaben, die schon vorher für Sie erledigt wurden. Die Initialisierung globaler Variablen ist eine davon.

Läuft Ihr Programm fehlerfrei durch, dann läuft es, bis es auf eine der `return`-Anweisungen in `main()` stößt. Die Zahl hinter dem `return` ist der *Rückgabewert* der Funktion. Weil es sich bei `main()` um eine spezielle Funktion handelt, die letztlich vom umgebenden Betriebssystem aufgerufen wird, hat dieser Wert eine besondere Bedeutung. Auf Unix-Systemen bedeutet eine `0`, dass das Programm erfolgreich durchgelaufen ist, und andere Programme können dies überprüfen. Und auch unter Windows können bestimmte Programme durch den Rückgabewert von `main()` anderen Programmen den Erfolg signalisieren. Während es Konvention ist, dass `0` für Erfolg steht, stehen alle Werte ungleich `0` für Misserfolg. Darüber hinaus kann man keine allgemeingültigen Regeln nennen, außer dass der am häufigsten verwendete Wert für Misserfolg `1` ist. Auch bei dem möglichen Wertebereich gibt es zwi-

schen den Systemen große Unterschiede, sodass man mit Werten zwischen `0` und `127` auf der sicheren Seite ist.

Speziell an `main()` ist, dass Sie die `0` von `return 0` weglassen können. Sie können sogar die ganze Zeile weglassen, dann endet das Programm, wenn die schließende geschweifte Klammer von `main()` erreicht ist. Beides ist in `main()` möglich – aber nur hier und nirgendwo anders.

Da wir schon bei den Besonderheiten von `main` sind: Es gibt noch eine weitere. Wie Sie in Listing 4.2 und Listing 4.7 sehen, kann man `main` in einer kurzen und einer langen Form deklarieren:

- `int main()` oder
- `int main(int argc, const char* argv[])`

Erstere verwenden Sie, wenn Sie kein Interesse an den Aufrufparametern des ausführbaren Programms haben. Mit der zweiten Variante können Sie herausfinden, ob die Benutzer Ihr Programm mit Argumenten aufgerufen hat, und können diese auswerten.

Nehmen wir an, Ihr Programm heißt `teiler.exe`, dann können (Windows-)Benutzer das Programm zum Beispiel so aufrufen:

```
$ teiler.exe 1001
```

Hier wird `argc` den Wert `2` haben, und mit `argv[1]` können Sie auf die `1001` zugreifen.

Es gibt nur diese beiden Möglichkeiten, um `main` zu definieren. Die `()`-Form ist eine Abkürzung, für den Fall, dass Sie sich für die Aufrufparameter des Programms nicht interessieren.

4.10.6 Anweisungen

Lassen Sie uns näher auf das eingehen, was zwischen den {…} unserer beiden Funktionen steht: die *Anweisungen*. Sie wissen schon, dass diese *nacheinander* ausgeführt werden. Das unterscheidet sie von *Ausdrücken*, denn für die kann der Compiler zaubern, wie er möchte. Moderne Compiler und Computer, auf denen Ihr Programm läuft, machen im Hintergrund die verrücktesten Dinge mit Ihrem Programm, vor allem, um es schneller ablaufen zu lassen. Die einzige Einschränkung für diese Transformationen ist, dass diese die *Bedeutung* des Programms nicht verändern dürfen. Und diese Bedeutung ist, dass die Anweisungen Ihres Programms nacheinander ausgeführt werden. Das ist genau der Unterschied zu *Ausdrücken*, die nicht auf diese Art in einer bestimmten Reihenfolge ausgeführt werden. Denken Sie sich bei denen besser, dass der Ausdruck einen Wert annimmt oder der Computer sie *auswertet*, aber nicht wörtlich *ausführt*.

Als Faustregel gilt, dass eine Anweisung an ihrem Semikolon `;` zu Ende ist. Die häufigste Ausnahme von dieser Regel ist der *Anweisungsblock*: die in geschweifte Klammern eingeschlossene Folge von Anweisungen.

Hier sehen Sie zur Erinnerung einen Ausschnitt aus unserem Beispielprogramm:

```
for(int teiler=1; teiler <= n; ++teiler)   // for-Schleife
{                                          // Beginn des Schleifenblocks
    if(n % teiler == 0)
        std::cout << teiler << ", ";
}                                          // Ende des Schleifenblocks
```

Listing 4.8 Hier setzen Sie nur die äußeren geschweifte Klammern.

Um es noch einmal deutlich zu machen: Bei der `for`-Schleife handelt es sich um *eine einzelne* Anweisung, die von `for` bis `}` reicht. Das Gleiche gilt für die `if`-Verzweigung: Im folgenden Listing gehören der Bedingungsteil `if(argc<=1)` und der Verzweigungsteil bis zur letzten `}` integral zusammen und bilden gemeinsam *eine einzelne* Anweisung:

```
if(argc<=1) {                              // Beginn der if-Anweisung
    std::cout << "Geben Sie eine Zahl ein: ";
    std::cin >> zahl;
    if(!std::cin) {
        return 1;
    }
} else {
    wert = std::stoi(argv[1]);
}                                          // Ende der if-Anweisung
```

Listing 4.9 So sieht es aus, wenn Sie alle geschweiften Klammern setzen.

Warum ist das wichtig? Weil somit klar ist, wo man die Schleife verwenden kann und wie weit sie reicht. Formal hat eine `for`-Schleife immer das folgende Format:

`for(` *Schleifenvariablendefinition* `)` *Anweisung*

Und eine der formalen Möglichkeiten für eine `if`-Verzweigung ist:

`if(` *Bedingung* `)` *Anweisung*

Da beides jeweils selbst eine Anweisung darstellt, ist klar, dass ich im Beispiel mit dem `for` einige Klammern auch hätte weglassen können. Denn Anweisungen können wiederum Anweisungen enthalten. Zum Beispiel enthält der Anweisungsblock des `for` eine einzige Anweisung, nämlich das `if`:

```
for(int teiler=1; teiler <= wert; ++teiler)
    if(wert % teiler == 0)
        std::cout << teiler << ", ";
```

Listing 4.10 Das »if« ist eine Anweisung und benötigt eigentlich keine geschweiften Klammern.

Das hat die gleiche Bedeutung wie:

```
for(int teiler=1; teiler <= wert; ++teiler) {
    if(wert % teiler == 0)
        std::cout << teiler << ", ";
}
```

Doch ist es guter Stil, die geschweiften Klammern bei `if` und `for` nicht wegzulassen. Richtig guter Stil und ebenfalls gleichbedeutend wäre es gewesen, auch für das `if` die {…} des Anweisungsblocks zu verwenden:

```
for(int teiler=1; teiler <= wert; ++teiler) {
    if(wert % teiler == 0) {
        std::cout << teiler << ", ";
    }
}
```

Listing 4.11 Setzen Sie besser auch einzelne Anweisungen in geschweifte Klammern.

Und noch etwas deutlicher: Es gibt unterschiedliche Schreibweisen dafür, *wo* die Klammern gesetzt werden. Manche bevorzugen sie als letztes Zeichen in der Zeile mit dem `if` oder `for`, andere setzen sie in eine eigene Zeile. Wie Sie schon bei der Besprechung der Leerzeichen gelesen haben, spielt es keine Rolle, ob Sie zusätzliche Zeilenwechsel einfügen. Für welche der Möglichkeiten Sie sich entscheiden, ist gleichgültig, es ist aber sicherlich vorteilhaft, einen Stil konsequent zu verfolgen.

Ans Ende einer Anweisung gehört zwar ein Semikolon, es sollte aber nicht nach den geschweiften Klammern eines Anweisungsblocks gesetzt werden. Da müssen Sie etwas aufpassen, denn unnötige Semikolons erzeugen eine *leere Anweisung*. Das ist in den meisten Fällen harmlos: Zum Beispiel sind zwischen den `;;;;` drei leere Anweisungen ohne jegliche Auswirkung. Aber ein Semikolon nach einem `if` führt dazu, dass die leere Anweisung zum `if` gehört und es beendet. Im folgenden Beispiel gehört so `cout << "kleiner";` nicht mehr zum `if` und wird nun immer ausgeführt.

```
if(wert<10);    // eine kritische leere Anweisung
    std::cout << "kleiner";
```

Geben Sie also besser keine zusätzlichen Semikolons ein. Achten Sie auch darauf, dass Sie nach den {…} eines Anweisungsblocks kein ; setzen. Auch wenn es keine direkten Auswirkungen hat, vermeiden Sie besser das folgende überflüssige Semikolon:

```
if(wert<10) {
    std::cout << "kleiner";
};                          // auch harmlose leere Anweisungen vermeiden
std::cout << "weiter";
```

Etwas ärgerlich ist, dass ich Ihnen in Kapitel 12, »Von der Struktur zur Klasse«, über `struct` und `class` erzählen werde, dass Sie deren Definition mit einem Semikolon abschließen *müssen*. Als Vorgriff: Sie werden also `struct Typ {…};` schreiben, und das Weglassen des Semikolons wäre hier ein Fehler. Der Grund ist, dass es sich um eine *Definition* handelt, die immer mit einem Semikolon abgeschlossen wird – es ist keine *Anweisung*.

4.10.7 Ausdrücke

Zu den Anweisungen gesellen sich die *Ausdrücke* (engl. *Expressions*). Ausdrücke kommen an ziemlich vielen Stellen in C++-Programmen vor – nicht zuletzt sind sie Teile von Anweisungen, aber auch an vielen anderen Stellen erlaubt, die keine Anweisungen sind.

Ein Ausdruck ist »eine Folge von *Operatoren* und *Operanden*, die eine Berechnung ausführen«. Das ist so trocken, dass es nur aus dem Text des C++-Sprachstandards kommen kann. Doch was bedeutet das?

Es bedeutet, dass etwas wie `3+4` ein Ausdruck ist, `3` und `4` sind die Operanden, `+` der Operator. Es kann auch komplizierter werden, wie zum Beispiel in `(3+4)*PI/sin(x)`, wo es mehrere Operatoren und Operanden gibt. Jeder Operand muss selbst wieder ein gültiger Ausdruck sein, und ein Ausdruck ist nur dann *gültig*, wenn er auch komplett ist – `1+2+3+` ist *kein* Ausdruck, denn dem letzten `+` fehlt sein Operand. Genauso wenig ist `3+4)` ein Ausdruck, denn es fehlt eine Klammer.

An diesem Beispiel sehen Sie, dass Ausdrücke aus kleineren Ausdrücken aufgebaut sind. Brechen Sie `(3+4)*PI/sin(x)` auseinander, sind darin folgende Ausdrücke enthalten:

- `(3+4) * PI / sin(x)`
- `(3+4) * PI` und `sin(x)`
- `(3+4)`, `PI` zum einen, `x` und sogar `sin` zum anderen
- `3+4`
- `3` und `4`

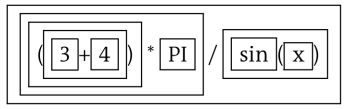


Abbildung 4.1 Alle Teilausdrücke in einem Ausdruck

In Beispielprogramm 4.2 und in Listing 4.1 finden Sie allerlei Ausdrücke, von denen ich hier nur einige auflisten kann:

- `100` ist ein *Zahlenliteral* – eine Zahl, die direkt in den Quellcode geschrieben ist.
- `"Teiler von "` ist ein *Textliteral* – ein Text direkt im Quellcode.
- `std::cout << "Teiler von "` ist ein Ausdruck, der auf dem Bildschirm ausgibt.[2]
- `wert % teiler` berechnet den Rest der Division `wert / teiler`.
- `wert % teiler == 0` prüft, ob dieser Rest `0` war und somit `teiler` ein Teiler von `wert` ist.
- `0` als Teil von `return 0;` ist ein Ausdruck, der die Rückgabe von `main()` »berechnet«.

2 Oder in eine Datei schreibt.

An ein paar Stellen haben wir etwas anderes als einen Ausdruck, und darauf will ich hier kurz eingehen. `int wert = 100` ist *kein* Ausdruck, sondern eine *Deklaration* (`int wert`) mit einer *Initialisierung* (`= 100`). Die Initialisierung rechts von `=` muss jedoch ein Ausdruck sein. Auch `return 0` ist kein Ausdruck, sondern eine `return`-Anweisung. Sie besteht aus dem besonderen Wort (*Schlüsselwort*) `return` und in diesem Fall dem Ausdruck `0`.

Eine der Haupteigenschaften von Ausdrücken ist, dass jeder einen genau festgelegten *Typ* hat, und das gilt auch für jeden Teilausdruck. Da Typen in C++ sehr wichtig sind, ist auch das Verständnis der Typen von Ausdrücken ein wichtiger Schritt zum Erfolg mit C++.

Mehr zu Anweisungen und Ausdrücken

Weil Anweisungen und Ausdrücke so elementare Sprachelemente sind, widmen wir ihnen ein späteres Kapitel (Kapitel 8, »Anweisungen im Detail«). Hier haben Sie eine erste Einführung und einen Überblick erhalten.

4.10.8 Zuweisungen

Einer der wichtigsten Ausdrücke ist in C++ die *Zuweisung*. Ich habe die Zuweisung nicht bei der Aufzählung der Anweisungen erwähnt. Das war kein Versehen.

Es gibt nur wenige Zuweisungen in Listing 4.2, abgesehen von den Initialisierungen, und zwar:

```
wert = std::stoi(argv[1]);
```

Listing 4.12 Zuweisung des Ergebnisses eines Funktionsaufrufs

Und bevor ich `++teiler` in die `for`-Schleife geschrieben habe, stand dort:

```
teiler = teiler + 1
```

Listing 4.13 Zuweisung des Ergebnisses einer Berechnung

In Listing 4.12 wird `wert` das Ergebnis eines Funktionsaufrufs zugewiesen. Listing 4.13 enthält eine Berechnung, deren Ergebnis die Variable `teiler` zugewiesen bekommt. Entscheidend ist, dass die Variable der Zuweisung zuvor schon existierte und nun einen neuen Wert erhält. Dies geschieht mittels des Gleichheitszeichens: Auf der linken Seite befindet sich das Ziel der Zuweisung und auf der rechten ein Ausdruck für den neuen Wert der Variablen.

Aber: Eine Zuweisung fällt in die Kategorie der *Ausdrücke*, und Sie können sie somit überall dort einsetzen, wo Ausdrücke erlaubt sind. Dass eine Zuweisung wie eine Anweisung verwendet wird, ohne weiter in einem größeren Ausdruck eingebettet zu sein, ist eigentlich ein Spezialfall – im Fall von Zuweisungen aber durchaus häufig, wie Sie in den Listings 4.12 und 4.13 sehen.

In dem folgenden Beispiel wird die Verwendung einer Zuweisung als Ausdruck deutlicher:

```
// https://godbolt.org/z/o5hGta
#include <iostream>
int main() {
    int a = 3;
    int b = 7 + (a = 12) + 6;    // enthält eine Zuweisung
    std::cout << a << std::endl;
}
```

Listing 4.14 Eine Zuweisung ist ein Ausdruck mit dem Typ der zugewiesenen Variablen.

Hier ist `a = 12` die Zuweisung. Dieser Ausdruck hat bei der Ausführung den Wert `12` und für den Compiler den Typ von `a`, also `int`. Die Variable `b` wird also mit der Berechnung `7 + 12 + 6` initialisiert.

Sie sollten sich hieran kein Beispiel nehmen und nun nicht überall Zuweisungen als Ausdrücke verwenden. Die Zuweisung wird dann zu einem Ausdruck mit *Seiteneffekt*, wenn neben dem Haupteffekt – `b` zu verändern – noch eine weitere Variable ihren Wert ändert. Wir zeigen Ihnen später die wenigen Stellen, an denen dies in C++ üblich und daher beim Lesen und Verstehen keine Überraschung ist.

> **Wert versus Referenz**
>
> In manchen Programmiersprachen ist jede Variable nur eine *Referenz* auf ein Objekt. Dies gilt zum Beispiel in Smalltalk für alle Variablen und in Java für alles, was von `Object` abgeleitet ist. Dort verändert eine Zuweisung nicht das Objekt selbst, sondern weist nur eine weitere Referenz zu. Wenn Sie in Java zum Beispiel `String s = "a"; String t = "b";` haben, dann ist `s = t;` eine Zuweisung. Nun »zeigen« `s` und `t` auf das gleiche `String`-Objekt. Das ist in C++ anders: Eine Zuweisung geschieht immer an den *Wert* einer Variablen. Wenn Sie in C++ `string s = "a"; string t = "b";` haben, dann *kopiert* die Zuweisung `s = t;` die *Werte* der Variablen. Ohne weitere Vorkehrungen arbeiten Sie in C++ also immer mit Werten, nicht mit Referenzen. Ja, Sie können in C++ auch mit Referenzen arbeiten, müssen das aber dann kenntlich machen. Vorgriff: Sie verwenden dann `&` für Referenzen und `*` für Zeiger und Adressen.

4.10.9 Typen

In C++ hat fast alles einen *Typ*. Variablen und Konstanten haben einen Typ, ebenso Parameter und auch Funktionen, Daten, Klassen und Ausdrücke. Typen sind überall.

Der Typ legt fest, wie viel Speicher der Computer für eine Variable oder das Zwischenergebnis eines Ausdrucks bereitstellen muss, welche Art Werte dieser aufnehmen kann, wie diese Werte im Speicher zu interpretieren sind und welche Operationen darauf erlaubt sind. Letzteres ist die Stärke von C++. Denn die Hauptarbeit des Compilers besteht darin, die Typinformation zu verarbeiten. Zunächst ermittelt er den Typ und findet die

Dinge heraus, die darauf erlaubt sind. Das können Funktionsaufrufe, Operationen oder Umwandlungen sein. Was ist mit diesem Programmcode gemeint? Wenn der Compiler das herausfindet, wendet er seine Auswahl an und legt damit den Typ für den Ausdruck fest zu.

Deshalb gehören Typ und Wert immer zusammen – sie ergänzen sich. Konzeptionell existiert der Typ nur zur Übersetzungszeit, als Werkzeug für den Compiler. Wenn Sie das übersetzte Programm starten (*Laufzeit*), sind die Typen verschwunden, und es existieren nur noch die Werte.[3] Ihr Programm manipuliert also die Werte und der Compiler die Typen. Durch seine Berechnungen stellt der Compiler sicher, dass die gewünschten Operationen auf den Werten in das Programm geraten.

Man kann für jeden (Teil-)Ausdruck dessen genauen Typ ermitteln. Dieser hängt von dem Operator und seinen Operanden ab. Manchmal ist das Ermitteln des Typs leicht, manchmal nicht, aber es geht immer. Wenn es schwierig ist, kann man sich beim Computer Hilfe holen. Das richtige Programm dafür ist der C++-Compiler! Denn dieser »rechnet« mit den Typen der Ausdrücke, und zwar nach den Regeln, die der C++-Standard und Ihr Programm vorgeben.

Wenn Sie schon andere Programmiersprachen kennen, dann sind Ihnen vielleicht weniger stringente Typkonzepte als das von C++ bekannt. Manche Sprachen (wie Python, PHP oder JavaScript) ignorieren sie fast komplett und fahren in ihrer jeweiligen Domäne gut damit. Andere Sprachen (wie Java oder C) haben Typen, sehen deren Nutzen aber etwas anders. In C++ können Sie Typen viel eher dazu verwenden, sich vom Compiler dabei helfen zu lassen, ein korrektes Programm zu schreiben.

Steht im Programm zum Beispiel irgendwo `100+99`, dann ist das für den Compiler ein *Operator* `+` mit zwei *Operanden* `100` und `99` jeweils vom Typ `int`. Nach den Regeln, die der Compiler eingebaut hat, ergibt ein `+` mit zwei `int`-Operanden wieder einen Wert vom Typ `int`, und somit hat der Gesamtausdruck auch den Typ `int`. An dieser Stelle rechnet der Compiler also nicht mit den konkreten Zahlen `100` und `99` und weiß auch noch nichts von der Bedeutung von `+`, sondern er wendet nur Wissen »über« Operanden und Operator an – deren Typen.

Wenn Sie in der Schule im Physikunterricht den richtigen Lehrer gehabt haben, könnte Ihnen das vielleicht bekannt vorkommen. Um beim Umgang mit langen Formeln zu überprüfen, ob Ihr Ergebnis wahrscheinlich richtig war, konnten Sie statt auf den konkreten Zahlen auf den Maßeinheiten und Größen rechnen (»Dimensionsanalyse«). Kam dabei die falsche Maßeinheit heraus, hatten Sie einen Fehler gemacht (in der Rechnung oder bei der Überprüfung). [4]

3 Das ist nicht ganz korrekt, denn der Compiler behält für bestimmte Sprachfeatures Laufzeit-Typinformation (RTTI) im Programm. Das kann man meistens abschalten. Es hilft jedoch, wenn Sie sich das hier so vorstellen.

4 *Conversion of Units of Measurement*, Gordon S. Novak Jr., *http://www.cs.utexas.edu/users/novak/units95.html*, IEEE Trans. on Software Engineering, vol. 21, no. 8 (August 1995), pp. 651-661, [2014-01-31]

Zum Beispiel ist:

$$\text{Kraft in } [\text{kg} \times \text{m/s}^2] = \frac{\text{Masse in } [\text{kg}] \times \text{Länge in } [\text{m}]}{\text{Zeit in } [\text{s}^2]}$$

Wollen Sie nun bei gegebener Masse 8 kg, Länge 12 m und 4 s wissen, wie viel Kraft wirkt, dann rechnen Sie $8 \times 12/4 = 24$ und gleichzeitig kg × m/s… und merken dadurch: »Hoppla, mit den Sekunden stimmt was nicht!« Richtig wäre gewesen $8 \times 12/4^2 = 6$ mit der Probe kg × m/s^2 – das passt.

Ähnlich macht es der Compiler. Die Regeln der Rechnung auf Typen geben Sie durch die Klassen, Funktionen und Templates Ihres Programms vor. Diese werden ergänzt durch die Standardbibliothek und andere Bibliotheken, die Sie verwenden, sowie durch die fest eingebauten Regeln. Und genau wie die Dimensionsanalyse Ihnen in der Physikklausur geholfen hat, kann Ihnen der Compiler mit seiner Überprüfung der Typ-Rechenregeln helfen, ein korrektes Programm zu schreiben:

```
// https://godbolt.org/z/jSKrwZ
#include <vector>
class Image {
    std::vector<char> data_;
public:
    void load(const char* filename); // lädt Bilddaten
};

class Screen {
public:
    void show(Image& image);          // image sollte const sein
};

void paint(Screen &screen, const Image& image) {
     screen.show(image);
}

int main() {
    Image image {};
    image.load("peter.png");
    Screen screen {};
    paint(screen, image);
}
```

Listing 4.15 Der Compiler hilft, korrekte Programme zu schreiben, indem er die Typen überprüft.

Das Pendant zu einem vergessenen »hoch 2« ist in C++ vielleicht ein fehlendes `const`: Der Bildschirm `Screen` hat eine Methode `show`, mit der er ein Bild malt. Man sollte meinen, das Bild `image` würde beim Malen nicht verändert. Somit ist es wahrscheinlich, dass nicht `Image&`, sondern dort `const Image&` als Typ des Parameters von `show` hätte stehen müssen.

Ein vergessenes `const` allein ist noch nicht kritisch, wohlgemerkt. Was würde aber passieren, wenn jemand »aus Versehen« in `show image.load("paul.png");` aufrufen würde – das würde das Bild verändern. Mit dem `const` am Parameter kann das nicht passieren: Der Compiler meldet einen Fehler.

Die Ressource *Typsicherheit* sollten Sie nutzen, wo immer Sie können. Sie stellt einen der mächtigsten Vorteile von C++ gegenüber dem guten alten C dar. Dort waren (und sind) die auf Typen möglichen Transformationen viel eingeschränkter – und die Überprüfungen sehr viel lapidarer.

> **Erste einfache Typen**
>
> Ohne es vielleicht besonders zu bemerken, hatten Sie es in den kleinen Beispielen, die wir besprochen haben, bereits mit allerlei Typen zu tun. Die wichtigsten behandeln wir jetzt.

Der Typ »int«

`int` ist in diesem Kapitel ein Repräsentant all jener Typen, mit denen Sie in C++ *Ganzzahlen* darstellen, also negative ganze Zahlen, die Null und positive. Ein Computer hat aber Beschränkungen, was seinen Zahlenbereich angeht. Mit `int` stehen Ihnen je nach System unterschiedlich viele *Bits* zur Verfügung. Sollten Sie auf einem aktuellen System arbeiten, sind es wahrscheinlich 32 Bit, es könnten aber auch 16, 64 oder eine ganz andere Zahl sein. Mit 32 Bit kann `int` ganze Zahlen von etwa –2 Milliarden bis +2 Milliarden speichern. Mit Listing 23.8 (Seite 578) können Sie es genau herausfinden. Wollen Sie andere Zahlenbereiche abdecken, sollten Sie sich die Verwandten `char` und `short` sowie `long` und `long long` in Abschnitt 4.12, »Eingebaute Datentypen«, einmal genauer anschauen.

Der Typ »bool«

Während eine `int`-Variable sehr viele Zustände annehmen kann, ist eine `bool`-Variable auf einen von zwei Zuständen festgelegt, nämlich `true` oder `false`, für »wahr« oder »falsch.«

Vergleiche sind Ausdrücke, deren Ergebnis vom Typ `bool` ist, und Sie können ihr Ergebnis, anstatt es direkt zu verwenden, in einer Variablen dieses Typs speichern:

```
// https://godbolt.org/z/oGwWsz
#include <iostream>                      // cout
int main(int argc, const char* argv[]) {
    bool mitParametern = argc > 1;       // Vergleichsergebnis zwischengespeichert
    if(mitParametern) {                  // ... und verwendet
        std::cout << "Sie haben das Programm mit Parametern aufgerufen.\n";
    }
    return 0;
}
```

Listing 4.16 Variablen vom Typ »bool« können das Ergebnis eines Vergleichs zwischenspeichern.

Die Typen »const char*« und »std::string«

Diese beiden sehr unterschiedlichen Typen haben den gleichen Zweck: Ihre Variablen stellen Zeichenketten dar, also Text, Nachrichten, Meldungen, Namen etc. Je nach Aufgabe bietet sich mal die »Pointervariante« `const char*` an, die ihre Herkunft aus C nicht leugnen kann, und mal die »Klasse« `std::string` aus der *C++-Standardbibliothek*.

Zeiger, Adressen und Speicherbereiche

Beachten Sie den Stern »*« in `char*` bzw. `const char*`. Der sagt Ihnen »Ich speichere nur die Adresse auf etwas.« Das bedeutet, dort, wo die Variable hinzeigt, befindet sich ein `char`. Es gibt auch `int*` als Zeiger auf einen `int` und so weiter.

Wenn es um Zeichen*ketten* gehen soll, fehlt aber noch eine weitere Information, denn der Zeiger enthält nur die Adresse des *ersten* Zeichens, weitere mögen folgen. In C ist es üblich, dass ein Text bis zum nächsten `char` mit dem Wert `0` reicht, auch `'\0'` geschrieben – dies ist dann ein C-String. Die C-Funktion `printf(const char* fmt, …)` funktioniert zum Beispiel auf diese Weise.

Andere Funktionen nehmen lieber Zeiger und eine Länge, zum Beispiel `memcpy(void *dest, const void * src, size_t n)`. Hier werden von `src` beginnend die nächsten `n` Byte nach `dest` kopiert.

Die dritte Variante, mit Zeigern einen Bereich zu abzustecken, finden Sie in C++ häufig bei Containern und Algorithmen. Sie können zum Beispiel mit `std::find(const char* beg, const char* end, char c)` das Zeichen `c` zwischen `beg` und `end` finden lassen.

Wenn es nur um die Ausgabe einer fixen Zeichenkette geht, arbeiten Sie meist mit `const char*`, denn in dieser Form tauschen viele Programmteile Zeichenketten untereinander. Sie haben den Parameter zu `main` schon gesehen, den ich in Listing 4.16 zum Beispiel `argv` genannt habe. Dort bringt das System möglicherweise mehrere Programmargumente in Form von `const char*`-Zeichenketten ins Programm hinein. Und auch wenn Sie direkt eine Zeichenkette mit "…" in den Programmtext hineinschreiben, legt der Compiler dieses »Zeichenketten-Literal« als `const char*` im Speicher des Computers ab. (Genauer gesagt, ist es ein `const char[]`, aber das ist sehr ähnlich und soll an dieser Stelle reichen.) Solange Sie diesen Text nur ausgeben wollen, wie in

```
std::cout << "Sie haben das Programm mit Parametern aufgerufen.\n";
```

brauchen Sie sich darum jedoch nicht weiter zu kümmern.

Für die meisten darüber hinausgehenden Fälle empfehle ich Ihnen, dass Sie stattdessen `std::string` verwenden. Wenn Sie Zeichenketten speichern, kopieren oder manipulieren wollen, ist dies die beste Wahl. Vor allem in Bezug auf die Speicherverwaltung – das dynamische Anlegen immer neuer Zeichenketten – fällt die Nutzung der C++-Klasse `std::string` deutlich leichter.

Zwar verwendet man der Einfachheit halber auch bei der Initialisierung von `string` einfach ein `const char*`-Literal, aber seit C++14 können Sie auch ein `s` an ein "…"-Literal anhängen und machen es somit zu einem echten `string`-Literal. Dazu muss in dem Block jedoch irgendwo `using namespace std::literals` oder eine der Alternativen stehen:

```
// https://godbolt.org/z/Tf_bQ9
#include <string>
#include <iostream>
int main() {
    std::cout << "C-Zeichenkettenliteral\n";
    // using std::literals::string_literals::operator""s;
    // using namespace std::string_literals;
    // using namespace std::literals::string_literals;
    using namespace std::literals;
    std::cout << "Echter string\n"s;
}
```

Listing 4.17 So können Sie einen C++-»string« als Literal in den Quellcode schreiben.

Das macht selten einen großen Unterschied, kann aber manchmal wichtig sein.

Weil Sie `std::string` sehr häufig verwenden werden (zumindest in diesem Buch), erlauben Sie mir, dass ich dies einfach mit `string` abkürze.

Parametrisierte Typen

Manche Typen enthalten spitze Klammern `<…>` in ihrem Namen. Sie haben

```
std::vector<char>
```

im Beispiel gesehen. Auch wenn das anders aussieht als `int`, `bool` oder `string`, so handelt es sich doch bei dem *ganzen Konstrukt* um einen einzigen Typ. Der Teil in den Klammern ist dabei ein Parameter, um den Gesamttyp zu bilden. Hier ist `vector` der umgebende oder Haupttyp, und `char` ist der Parameter. Zusammen sind die beiden ein »`vector` von `char`«.

Sie könnten auch einen `vector<int>` bilden und erhalten so einen anderen Typ. Das ist zum Beispiel wichtig, wenn der Compiler entscheiden soll, welche Funktionen zur Auswahl stehen oder was das Ergebnis einer Operation ist. Wie ich schon erwähnt habe, rechnet der Compiler in der Übersetzungsphase mit Typen, erst zur Laufzeit rechnet das Programm auf den Werten.

Eines ist hier besonders wichtig: Bei den parametrisierten Typen muss das, was innerhalb der spitzen Klammern steht, immer schon zur Übersetzungszeit feststehen. Logisch, wenn der Compiler doch zur Übersetzungszeit mit Typen rechnet und das gesamte Konstrukt, bestehend aus Haupttyp, Klammern und Parametern, den Typ ausmacht. Wenn Sie sich das merken, werden Sie bei Experimenten nicht verwundert sein. So gibt es zum Beispiel auch den parametrisierten Typ `std::array<…>`, der als zweiten Parameter eine Zahl bekommt. Die Zahl muss eine Konstante sein oder etwas, das der Compiler schon

ausrechnen kann, wie `3+4`. Sie können `std::array<int,5>` schreiben, aber nicht `int n=5; std::array<int,n>`. `n` ist eine Variable und kann als solche nicht Teil des Typs sein.

Merke

Ein Typ muss zur Compilezeit feststehen.

Template-Parametertyp-Deduktion

Nicht immer müssen Sie die Argumente eines parametrisierten Typs angeben. Wenn Sie einen Konstruktor aufrufen, kann der Compiler auf die Typen selbst schließen:

```
// https://godbolt.org/z/vc6Pw8
std::vector vec { 1, 2, 3 };          // statt vector<int>
std::tuple tpl { 5, 'x' };            // statt tuple<int,char>
std::shared_ptr<int> ptr { new int(5) };
std::shared_ptr ptr2 { ptr };         // statt shared_ptr<int>
```

Listing 4.18 Seit C++17 bestimmt Compiler die Typparameter von Templates anhand der Konstruktorargumente.

So können Sie Ihren Code in manchen Fällen knapper und übersichtlicher halten. Das geht aber nicht immer, und Sie werden die Typparameter noch oft genug selbst nennen.

Deduzierte Templateparameter in diesem Buch

Dieses Feature gibt es erst seit C++17. Lesen Sie mehr dazu in Abschnitt 23.4.3, »C++17 und Template-Parametertyp-Deduktion«. Sollten Sie einen älteren Compiler benutzen, schreiben Sie die Parameter der Typen aus.

4.10.10 Variablen – Deklaration, Definition und Initialisierung

Es ist ein Unterschied, ob Sie schreiben:

- `int wert;` – eine *Deklaration*
- `wert = 0;` – eine *Zuweisung*
- `int wert = 0;` – eine *Definition* mit *Initialisierung*

Außerdem gibt es Variablen, die Sie nach deren Einführung nicht mehr verändern können. Sie sind unveränderbar – und heißen dann eigentlich *Konstanten*. Bis auf diesen Fakt sind für den Compiler Konstanten *auch* Variablen, also »mengenlehremäßig« eine Teilmenge davon. Diesen können Sie niemals mehr etwas neu zuweisen, daher haben Sie nur die Wahl, sie bei der Deklaration mit einem Wert zu initialisieren. Auf `const` gehe ich später noch genauer ein, aber Sie können den Hauptzweck dieses Zusatzes sicher erraten:

```
int main() {
    const int fest = 33; // Initialisierung als Konstante
    fest = 80;           // eine Zuweisung ist unmöglich
}
```

Listing 4.19 Manchen Variablen können Sie nichts zuweisen, Sie können sie nur initialisieren.

Es gibt noch eine weitere Möglichkeit, wie Sie Zuweisung und Initialisierung voneinander unterscheiden können bzw. selbst unterscheidbar machen können. Mit C++11 wurde die *vereinheitlichte Initialisierung* eingeführt (engl. *Unified Initialization*). An allen Stellen, an denen Sie Variablen initialisieren, können Sie statt des = nun geschweifte Klammern verwenden.

```
int index = 1;          // alter Stil, sieht wie eine Zuweisung aus
int zaehler { 1 };      // C++11-Stil, eindeutig eine Initialisierung
int counter = { 1 };    // beim C++11-Stil ist das »=« optional und wird ignoriert
```

Listing 4.20 Statt des »=« können Sie »{...}« zur Initialisierung verwenden.

Eine Verwechslungsgefahr gab es zuvor nämlich nicht nur bei der Zuweisung, sondern auch bei einer Hand voll anderer C++-Konstrukte.

Verwechslungsgefahr()

Es ist ein Vorgriff, aber ich nenne Ihnen ein Beispiel. Sie können einer Funktion `f` ein temporäres `Ding` übergeben, das Sie mit dem Wert `12` initialisieren, also dessen Konstruktor aufrufen:

```
f(Ding(12));
```

Wenn das nicht temporär sein soll, funktioniert zum Beispiel:

```
Ding d = Ding(12);
```

Wollen Sie den Konstruktor ohne Argumente nutzen, dann geht das auf eine dieser Arten:

```
f(Ding());
Ding d = Ding();
```

Auch das Folgende ist in Ordnung:

```
Ding d(12);
```

Sie haben nun ein `d`, das mit `12` initialisiert wurde. Jetzt könnten Sie schlussfolgern, dass Sie so ein `d` ohne Konstruktorargumente initialisieren:

```
Ding d();
```

Und täuschen sich! Sie haben nun eine Funktion `d` deklariert, die keine Argumente nimmt und ein `Ding` zurückliefert. Beachten Sie bitte: Sie haben sie *deklariert* und nicht *definiert*. Das ist vergleichbar mit `int mydata();` als Funktionsdeklaration in einer Headerdatei. Sobald Sie `d` in irgendeiner Weise verwenden wollen, zum Beispiel mit `d.print()`, beschwert sich der

Compiler mit den obskursten Fehlermeldungen, etwa dass »d noch nicht definiert wurde« oder »ein Funktionsaufruf erwartet wurde«.

Wenn Sie es ausprobieren wollen, brauchen Sie sich kein `Ding` zu machen – probieren Sie es mit `int` oder `short` aus:


```
// https://godbolt.org/z/jzmzPz
int main() {
    int d1(12);       // definiert einen int und initialisiert ihn mit 12
    int b1 = d1+1;
    int d2();         // deklariert eine Funktion d2 ohne Parameter, Rückgabe int
    int b2 = d2+1;
    int d3{};         // definiert einen int und initialisiert ihn mit 0
    int b3 = d3+1;
}
```

Meine Fehlermeldung heute:

```
arithmetic on a pointer to the function type 'int ()'
```

Die Rettung ist, durchgehend `{…}` zu nutzen, denn `Ding d{};` tut, was Sie erwarten.

Altes Gleichheitszeichen oder vereinheitlichte Initialisierung?

Die neue Initialisierung mit `{…}` beherrschen alle aktuellen Compiler. Sie haben also die Wahl. Der »alte Stil« wird sich nicht ausmerzen lassen (und es ist diskussionswürdig, ob das sinnvoll wäre), aber Sie sollten den neuen Stil überall dort verwenden, wo Sie nicht nur einen nativen Datentyp wie `int` initialisieren. Das rüstet Sie für die kniffligen Fälle.

4.10.11 Initialisieren mit »auto«

Ein Wort noch zur Initialisierung. Die besteht immer aus:

- dem Typ der Variablen, die Sie initialisieren,
- dem Namen der Variablen sowie
- einem Ausdruck für den ersten Wert der Variablen.

Oft ist der Typ des Ausdrucks derselbe wie der Typ der Variablen. Manchmal lassen Sie den Compiler noch eine *implizite Umwandlung* durchführen. Häufig können Sie sich aber die Arbeit sparen, den Typ des Ausdrucks noch einmal zu nennen. Sie können stattdessen `auto` verwenden. Der Compiler nimmt dann den Typ des Ausdrucks, um den Variablentyp zu bestimmen – man nennt das *Typinferenz*.

```
auto zahl = 10;
auto wert = 3.1415 * sin(alpha);
for(auto it = container.begin(); ... ) ...
```

`zahl` wird hier ein `int`, `wert` ein `double`, und `it` ist von dem Typ, den `begin()` zurückliefert, wahrscheinlich ein `iterator` der Variablen `container`.

`auto` legt den Typ der Variablen fest, als hätten Sie den Typ explizit hingeschrieben. Es ist mitnichten so, dass `auto` ein eigener veränderlicher Typ ist. Einmal festgelegt, behält die Variable ihren Typ.

Der Compiler ermittelt den Typ `int` für `val`. Eine spätere Zuweisung eines Textliterals ist ein Fehler:

```
auto val = 12;
val = "Name";
```

Wie `auto` genau funktioniert, erfahren Sie in Abschnitt 12.12, »Typinferenz mit auto«.

4.10.12 Details zur »include«-Direktive und »include« direkt

Die Includes beginnen immer mit einen Doppelkreuz #. Ein solches Zeichen am Anfang einer Zeile steht für eine Anweisung an den *Präprozessor*. Wenn Sie sich an Abbildung 2.1 erinnern, kennen Sie diese Phase der Übersetzung schon.

Die Datei, die hier mit `#include` angegeben wird, wird vom Präprozessor fast wie per Copy-and-Paste an diese Stelle in Ihr Programm eingefügt. Der Effekt ist, dass alle Deklarationen aus jener Datei nun Teil Ihres Programms sind.

Es gibt noch andere Präprozessordirektiven, doch `#include` ist für Sie die wichtigste. Die restlichen finden Sie in siehe Kapitel 21, »Makros«.

Sie sollten Includes immer zuoberst in Ihre Quelldateien schreiben. In Listing 4.2 habe ich zwei davon verwendet:

```
#include <iostream>          // cin, cout
#include <string>            // stoi
```

Beide, `<iostream>` und `<string>`, sind Teil der *C++-Standardbibliothek*. Sie enthalten also Dinge, die nicht Teil der *Kernsprache* sind. Etwas wie `int` ist immer und auf allen Systemen vorhanden, ohne dass Sie einen Include benötigen. Die Standardbibliothek wird meistens vorhanden sein – und davon gehen wir in diesem Buch aus –, aber nicht immer.

Da Sie mit einem `#include` viele Bezeichner in den aktuellen Quellcode einführen, ist es hilfreich, in einem Kommentar hinzuzufügen, welche davon Sie hier verwenden. Dadurch können die Leser bei einem unbekannten Bezeichner mittels einer einfachen Suche in der Datei herausfinden, wo dieser herkommt. Manche Modulnamen sind selbsterklärend: `#include <vector>` importiert `vector`, das muss man nicht erklären.

Die Includes der Standardbibliothek erkennen Sie daran, dass der Name keine Endung wie `.h` oder Ähnliches hat. Es ist Konvention, dass nur *Headerdateien* der Standardbibliothek ohne Endung verwendet werden.[5] Wenn Sie selbst welche erzeugen, dann hängen Sie ein `.h`, `.hpp`, `.H`, `.hh` oder `.hxx` an. Das `.h` sieht man in der Praxis am häufigsten, auch wenn

5 Sie werden jedoch bekannte Bibliotheken finden, die ebenfalls keine Endung verwenden, wie zum Beispiel die Qt-Bibliothek.

manche Programmierer dies weniger gerne sehen, da sich der Header dann von außen nicht von C-Headern unterscheidet. `.hpp` wird oft als Alternative gewählt. Egal, welche Endung Sie nehmen, beim `#include` nennen Sie diese:


```
#include <iostream>              // Modul der Standardbibliothek
#include <asteroids.h>           // Modul eines Drittanbieters
#include "meinModul.hpp"         // Modul des aktuellen Projekts
#include "algo/meinModul.h"      // in einem Unterverzeichnis
```

In spitzen Klammern schreiben Sie den Namen der Headerdatei, wenn die Datei auf Ihrem System *installiert* ist – der Compiler soll danach suchen. Verwenden Sie die Anführungszeichen "…", sofern die Headerdatei Teil Ihres aktuellen Projekts ist und nicht irgendwo installiert. Manche Compiler suchen dann zuerst im aktuellen Projekt nach der Datei.

Don'ts für Includes

Achten Sie bei der Nennung des Dateinamens auf Groß- und Kleinschreibung, denn diese wird auf manchen Systemen beachtet, liebe Windows-Benutzer! Sollte sich die Headerdatei in einem Unterverzeichnis befinden, dann verwenden Sie immer / zur Trennung:

```
#include "MEINMODUL.H"   // nicht gut, wenn die Datei »meinModul.h« heißt
#include "algo\meinModul.h"          // auch unter Windows »/« verwenden
#include "c:/projekt/meinModul.h"    // keine absoluten Pfadangaben
#include "../algo/meinModul.h"       // keine relativen Pfade mit »..«
```

Statt den absoluten Pfad oder `..` zu nutzen, sollten Sie die Headerdateien woanders in Ihrem Projekt ablegen oder/und müssen die Suchpfade für Headerdateien in den Optionen des Compilers anpassen. Das erleichtert Ihnen das Verschieben Ihres Projekts und dessen Umorganisation, falls nötig.

Noch eine Anmerkung zu Klammern versus Anführungszeichen: Ursprünglich bedeuteten <…> gegenüber "…", dass der Header möglicherweise Teil des Compilers ist und gar nicht als Datei vorliegt. Es hat sich aber durchgesetzt, dass die Klammern auch dann verwendet werden, wenn es sich einfach um eine Headerdatei handelt, von der erwartet wird, dass sie von Ihnen auf dem System installiert wurde. Verwenden Sie daher "…" nur für Dateien, die Teil Ihres Projekts sind und Vorrang gegenüber anderen Bibliotheken haben sollen. Ob das mit dem Vorrang klappt, hängt aber vom jeweils verwendeten Compiler ab.

C++20: Module

Mit C++20 werden *Module* eingeführt. Statt textuell per `#include` Deklarationen in eine Übersetzungseinheit »hineinzukopieren«, teilen Sie Ihren Code in Interface und Implementierung auf. Dazu verwenden Sie die neuen Schlüsselwörter `module` und `import`. Die genaue Nutzung finden Sie in der C++-Referenz und in Tutorials wie *https://youtu.be/tjSuKOz5HK4*

> von Dos Reis. Allerdings gibt es noch keine »Best Practice«, und auch die Standardbibliothek ist noch nicht auf Module umgestellt, was aber für C++23 angestrebt wird.
>
> Module und Includes schließen sich gegenseitig nicht aus. Wenn Sie bestimmte Schemata einhalten, können Sie beides parallel nutzen – und müssen es wahrscheinlich auch noch lange, denn auch wenn Module die bessere Alternative sind, wird es aus historischen Gründen Includes noch sehr lange geben.

4.10.13 Eingabe und Ausgabe

Jetzt kennen Sie alle wichtigen Elemente des Programms, bis auf – so würden manche sagen – die wichtigsten. Denn was ist ein Programm ohne *Eingabe* und *Ausgabe*?

Sie haben schon gesehen, dass `<<` als Operator für die Ausgabe verwendet wird. Und außerdem wurde `#include <iostream>` wegen `std::cout` mit eingebunden. Hier folgt ein Ausschnitt aus Listing 4.2, bei dem es hauptsächlich um die Ausgabe geht:

```
std::cout << "Teiler von " << n << " sind:\n";
for(int teiler=1; teiler <= n; ++teiler) {
     if(n % teiler == 0)
        std::cout << teiler << ", ";
}
std::cout << std::endl;
```

Sie sehen, dass Sie mehrere `<<` einfach aneinanderreihen können. Genauer gesagt, ist `std::cout << "Teiler von "` ein Ausdruck, der als Nebeneffekt die Ausgabe erledigt und danach wieder `std::cout` zurückliefert. So wird danach effektiv `std::cout << n` ausgeführt, wieder mit der Ausgabe und dem Ergebnis `std::cout`. Zu guter Letzt kommt der dritte Operator der Anweisung zum Zuge, und `std::cout << " sind:\n"` wird ausgeführt. An diesem ist besonders, dass er einen Zeilenwechsel enthält (engl. *Newline*). In C++-Zeichenketten wird ein solcher durch `\n` dargestellt. Genau dieser Zeilenwechsel wurde innerhalb der Schleife bei der Ausgabe weggelassen – daher erscheinen die Zahlen in einer Zeile.

Mit `std::cout << std::endl` gibt das Programm den Zeilenwechsel am Ende der Liste aus. Doch hoppla, wo ist das `\n`? Diesmal steht im Listing ein besonderes Element von `std::cout`, der *Manipulator* `std::endl`. Dieser sorgt für den Zeilenwechsel, und zusätzlich garantiert er, dass das System die Ausgaben nicht puffert – es erzwingt, dass alles wirklich auf dem Bildschirm erscheint, was Sie bisher ausgegeben haben. Denn (Bildschirm-)Ausgabe ist zeitintensiv, und das System bemüht sich, so viele Schritte wie möglich zusammenzufassen und auf einmal zu erledigen. Mit `std::endl` stellen Sie sicher, dass das System hier alles Angefallene wirklich ausgibt.

4.10.14 Der Namensraum »std«

Nun bleibt noch eine Sache, die ich Ihnen erklären muss, bevor wir tiefer in die einzelnen Themen eintauchen. In Listing 4.2 sehen Sie ein `using namespace std`. Mit diesem Hilfsmittel sparen Sie sich das `std::` vor Bezeichnern, das Sie sonst gebraucht hätten:


```
#include <iostream>                          // für std::cin, std::cout, std::endl
#include <string>                            // für std::stoi
void berechne(int n) {
    using namespace std;                     // für std::cout und std::endl
    /* Teiler ausgeben */
    cout << "Teiler von " << n << " sind:\n"; // cout statt std::cout
    for(int teiler=1; teiler <= n; ++teiler) {
        if(n % teiler == 0)
            cout << teiler << ", ";          // cout statt std::cout
    }
    cout << endl;
}
```

Listing 4.21 Sie können einen Namensraum einbinden, um Programmtext kürzer zu machen.

Normalerweise prüft der Compiler zu jedem Bezeichner, ob dessen Name lokal oder auf den äußeren Ebenen existiert. Zum Beispiel ist `teiler` eine lokale Variable und `n` ein Parameter innerhalb der Funktion. Würden Sie ohne das `using` dann `cout` verwenden, würde der Compiler prüfen, ob es eine lokale Variable oder einen Parameter mit diesem Namen gibt, dann, ob eine globale Variable `cout` existiert, und Ihnen anschließend mit einem Fehler melden, dass dem nicht so ist – dass der Bezeichner also unbekannt ist.

Mit dem `using` wird zu jedem Bezeichner, der nicht gefunden werden konnte, probiert, ob es mit einem vorangestellten `std::` funktionierte. So können Sie sich einiges an Tipparbeit einsparen, wo Sie sonst viele `std::` tippen würden.

Das `using namespace` wirkt sich innerhalb des Bereichs aus, in dem es auch definiert ist – im Beispiel also nur innerhalb der Funktion `berechne`. Sie können ein solches `using namespace` auch global schreiben, dann ist der Wirkungsbereich größer:

```
#include <iostream>
#include <string>
using namespace std; // wirkt sich global aus; klappt, ist aber kritisch
void berechne(int n) {
    /* Teiler ausgeben */
    cout << "Teiler von " << n << " sind:\n";
    // ...
}
// ... auch in weiteren Funktionen ...
```

Listing 4.22 »using namespace« können Sie auch global verwenden, sollten es aber selten tun.

Klingt das nun so verlockend, dass Sie `using namespace …` ständig einsetzen wollten? Tun Sie das nicht! Durch ein `using namespace …` wissen Sie nämlich ab jetzt nicht mehr, wo die Bezeichner Ihres Programms herkommen. Wenn Sie zum Beispiel die Standardbibliothek noch nicht kennen und auf ein frei stehendes `cout` stoßen, woher sollen Sie nun wissen, dass es eigentlich `std::cout` ist?

Vermeiden Sie globales »using namespace«

Verwenden Sie `using namespace …` nicht auf globaler Ebene in einer Datei. Zu jedem Bezeichner fragt man sich beim Lesen, woher dieser kommt, und man hat auch mit einer Suche in der aktuellen Datei keine Chance, das herauszufinden.

Innerhalb eines Blocks, zum Beispiel lokal in einer Funktion, wie ich es in Listing 4.21 gezeigt habe, ist ein `using namespace …` jedoch nicht verpönt. Der Bereich, in dem man sich fragt, aus welchem Namensraum ein Bezeichner kommt, ist eingeschränkt genug. Und die Wahrscheinlichkeit, dass mehrere `using namespace` gleichzeitig aktiv sind, ist geringer.

Doch es gibt eine Lösung: Holen Sie sich nicht alle Bezeichner eines Namensraums, sondern nur die Bezeichner, die Sie benötigen:

```
#include <iostream>                          // cin, cout, endl
using std::endl;                             // gilt global in dieser Datei
void berechne(int n) {
    using std::cout;                         // gilt lokal in dieser Funktion
    /* Teiler ausgeben */
    cout << "Teiler von " << n << " sind:\n";
    for(int teiler=1; teiler <= n; ++teiler) {
        if(n % teiler == 0)
            cout << teiler << ", ";
    }
    cout << endl;
}
```

Listing 4.23 Holen Sie sich mit »using« einzelne Bezeichner.

Das Einbeziehen von Bezeichnern in dieser Art und Weise bereitet wenig Probleme. Taucht irgendwo in der Datei mal ein allein stehendes `cout` auf, findet man den Ursprungsnamensraum mit einer einfachen Suche innerhalb der Datei.

In diesem Buch

Da die Listings in diesem Buch selten lang und unübersichtlich werden, mache ich hier eine Ausnahme. Es spart Raum und Wiederholung, zu Beginn nach den Includes `using …` zu schreiben. Daher werde ich vor allem `std::cout`, `std::endl` und `std::string` abkürzen, andere Bezeichner jedoch mit `std::` ausschreiben, damit Sie wissen, woher sie kommen.

4.11 Operatoren

Sehr häufig werden ein- und zweistellige Operatoren in Ausdrücken verwendet. `3+4` ist ein einfaches Beispiel, es könnte aber auch ein Ausdruck wie `!isBad && (x >= x0) && (x <= x1)` auftauchen. Mit solchen Aneinanderreihungen von Operatoren und Operanden können Sie in C++ eine Menge bewegen. Da Sie nun über Variablen, Typen und Ausdrücke eine Menge wissen, sollen Sie die möglichen Operatoren kennenlernen.

Exemplarisch erkläre ich Operatoren hauptsächlich anhand der Typen `int` und `bool`, damit Sie das Repertoire erst einmal kennenlernen. Aber viele Operatoren sind auch auf andere Typen anwendbar. Das sind durchaus eingebaute Typen, wie zum Beispiel `float`, aber auch solche der Standardbibliothek, wie `std::string` und `std::stream`.

In C++ können Sie eigene Typen definieren, die Operatoren ebenfalls unterstützen. Dass diese dann auch etwas machen, was für die eingebauten Typen gilt, liegt in Ihrer Hand. Wir erwarten zum Beispiel, dass `+` eine Addition ausführt – wie bei einem `int`. Die Klasse `string` verwendet `+` aber zur Konkatenation, was noch »additionsartig« ist. Aber Sie können, wenn Sie wollen, eine Klasse `Image` schreiben, die mit sich `+` gefolgt von einem `string` in eine Datei speichert. (Bitte tun Sie das nicht!) Das behandeln wir an geeigneter Stelle. Hier werden Sie zunächst erfahren, welche Operatoren es überhaupt gibt.

Operatoren für eingebaute Typen können nicht überschrieben werden

Wenn ich also in diesem Kapitel Operatoren beschreibe, dann meine ich die für die eingebauten Typen. In C++ können Sie diese nicht verändern. Ein `+` für `int` gibt es schon, und Sie als Programmierer können dessen Bedeutung nicht verändern. An einigen Stellen gehe ich auf die Typen der Standardbibliothek ein, doch für vieles muss die Referenz herhalten.

4.11.1 Operatoren und Operanden

Ein *Operator* ist etwas, das sich so ähnlich verhält wie eine Funktion, ohne aber eine zu sein. Anders ist, dass Sie bei einer Funktion erwarten würden, dass bei ihrer Anwendung ihr Name vorne steht und die Argumente in Klammern dahinter, also `func(arg1, arg2)`. Mit ein bisschen kreativer Freiheit könnten Sie `func` in die Mitte schreiben und die Klammern weglassen, und Sie erhielten `arg1 func arg2`. Und wenn der Funktionsname `+` statt `func` wäre, hätten Sie einen Operator `+` mit seinen beiden Operanden `arg1` und `arg2`.

Die meisten Operatoren schreiben Sie mit Symbolen, wie `+` oder `<<`. Viele haben zwei Argumente und heißen deshalb zweistellig oder binär (engl. *Binary Operators*). In C++ sieht das dann immer so aus:

Operand Operatorsymbol Operand

Sind sie einstellig, also unär (engl. *Unary Operators*), steht der Operator vor dem Operanden wie in `-4` oder seltener auch danach wie in `idx++`:

Operatorsymbol Operand

Operand Operatorsymbol

Es gibt noch weitere Formen von Operatoren, die im Folgenden erklärt werden.

4.11.2 Überblick über Operatoren

Man kann die Operatorsymbole in einige Gruppen einteilen:

- **arithmetische Operatoren**
  Dies sind die vier Grundrechenarten `+`, `-`, `*`, `/` sowie `%` für den Divisionsrest (Modulo). Das Vorzeichen können Sie mit den unären Operatoren `+` und `-` beeinflussen.
- **bitweise Arithmetik**
  Zahlen können Sie mit `|`, `&`, `^`, `~`, `<<` und `>>` bitweise miteinander verknüpfen.
- **zusammengesetzte Zuweisung**
  Neben dem `=` gibt es auch die Zuweisungen, die gleichzeitig eine andere Operation ausführen (engl. *Compound Assignments*), also `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=` und `|=`.
- **Inkrement und Dekrement**
  Die beiden einstelligen Operatoren `++` und `--` gibt es jeweils in einer vorangestellten und einer nachgestellten Variante (Präfix und Postfix). Bevorzugen Sie möglichst die Präfixvariante, denn die kommt ohne temporäre Variable aus.
- **relationale Operatoren**
  Relationale Operatoren führen einen Vergleich aus und liefern einen Wahrheitswert `bool` zurück: `==`, `<`, `>`, `<=`, `>=` und `!=`. Wenn Sie mit der Standardbibliothek arbeiten, sind `==` und `<` die wichtigsten, denn viele Algorithmen benutzen nur diese beiden, um nötigenfalls die anderen herzuleiten. Das ist wichtig, wenn Sie eigene Datentypen für die Standardbibliothek fit machen wollen.
- **logische Operatoren**
  `&&`, `||` und `!` verknüpfen Wahrheitswerte zu komplexeren Ausdrücken.
- **Pointeroperator und Dereferenzierungsoperator**
  Mit den unären Operatoren `&`, `*` sowie den binären Operatoren `->` und `.` adressieren und dereferenzieren Sie. Das heißt, Sie holen eine Adresse, machen aus einer Adresse ein Datum oder greifen in eine Struktur hinein. Sie werden später den Einsatz im Detail sehen.
- **besondere Operatoren**
  Mit `?` und `:` zusammen können Sie einen Entweder-oder-Ausdruck schreiben. Das Komma `,` kann als Sequenzoperator in Ausdrücken verwendet werden. Mit C++20 kommt der Spaceship-Operator `<=>` hinzu.

- **funktionsähnliche Operatoren**
  Streng genommen gehören auch einige Sonderlinge zu den Operatoren, die echte Namen haben und wie Funktionen verwendet werden. Das sind die Typumwandlungen wie `(int)wert` sowie `sizeof(a)` und einige andere. Dass es sich hierbei um Operatoren handelt, hat zum Beispiel zur Konsequenz, dass Sie von `sizeof` nicht die Adresse erfragen können. Einer Funktion könnten Sie nur Werte und Variablen als Parameter mitgeben, bei `sizeof` kann es aber auch ein Typ sein, wie `sizeof(int)`. Auch sind die Klammern streng genommen optional; Sie könnten `sizeof a` schreiben, das wäre aber sehr unüblich.

4.11.3 Arithmetische Operatoren

Sie können in C++ ganz normal mit Zahlen rechnen. Neben den Grundrechenarten `+`, `-`, `*` und `/` gibt es noch `%` für den Divisionsrest. Wenn Sie keine Klammern verwenden, gilt Punkt- vor Strichrechnung.

```
// https://godbolt.org/z/aDIyAM
#include <iostream>

int main() {
    std::cout << "3+4*5+6=" << 3+4*5+6 << "\n";           // Punkt vor Strich; = 29
    std::cout << "(3+4)*(5+6)=" << (3+4)*(5+6) << "\n";   // Klammern; = 77
    std::cout << "22/7=" << 22/7 << " Rest " << 22%7 << "\n"; // 22/7 = 3 Rest 1

    for(int n=0; n < 10; ++n) {
        std::cout << -2*n*n + 13*n - 4 << " ";           // mit unärem Minus
    }
    std::cout << "\n";
    // Ausgabe: –4 7 14 17 16 11 2 –11 –28 –49
}
```

Listing 4.24 Arithmetische Operatoren in der Anwendung

Was hier für `int` gezeigt wurde, geht mit allen Ganzzahltypen. Und außer bei `%` funktioniert es auch mit allen Fließkommatypen (`float` etc.). In der Standardbibliothek finden Sie `std::complex<>`, mit dem Sie diese Operatoren ebenfalls anwenden können.

Den Plusoperator `+` verwenden viele Typen zum Zusammenfügen. Sie können zum Beispiel aus

```
std::string vor="Hans";
std::string nach="Huber";
```

mit `vor+" "+nach` den neuen String `"Hans Huber"` machen.

4.11.4 Bitweise Arithmetik

Die bitweise Arithmetik sieht wahrscheinlich zu Anfang etwas seltsam aus.

```
int a = 41;       // dezimale 41
int b = a & 15;  // ergibt 9
```

Die Erklärung ist, dass Zahlen im Computer ja als Folge von 0 und 1 dargestellt werden – eben als »Bits«. Die dezimale 41 ist in Bitdarstellung 101001 – »binär«.

Binärsystem

Weil im Dezimalsystem 10 die Basis ist, schreiben wir »vierhundertzwölf« als 412 als Abkürzung für $4\times10^2 + 1\times10^1 + 2\times10^0 = {}_{10}412$. Für den Computer mit der Basis 2 ist das $1\times2^8 + 1\times2^7 + 0\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0$ oder abgekürzt ${}_{2}110011100$. Die unten stehende 10 bzw. 2 illustriert, in welchem Zahlensystem die Zahl dargestellt ist.

Jede Umwandlung vom Dezimalsystem in ein anderes erfolgt einfach mittels wiederholter Division durch die Basis und Notieren des Rests, bis der Wert null erreicht ist. Um zum Beispiel ${}_{10}412$ ins Binärsystem umzuwandeln, dividieren Sie wiederholt durch 2:

```
412 / 2 = 206, Rest 0
206 / 2 = 103, Rest 0
103 / 2 =  51, Rest 1
 51 / 2 =  25, Rest 1
 25 / 2 =  12, Rest 1
 12 / 2 =   6, Rest 0
  6 / 2 =   3, Rest 0
  3 / 2 =   1, Rest 1
  1 / 2 =   0, Rest 1
```

Die Reste lesen Sie von unten nach oben und schreiben sie als Binärfolge erst einmal auf. Die unterste 1 repräsentiert also die höchste Stelle: ${}_{2}1\ 1001\ 1100$.

Das geht natürlich auch einfach in C++. Listing 4.25 ist teilweise ein Ausblick auf Dinge, die Sie in C++ noch kennenlernen werden, aber ich denke, es ist dennoch lehrreich.

```
// https://godbolt.org/z/nQbPA6
#include <iostream>
void printBin(int x) {
    while(x>0) {          // fertig?
        int a = x/2;      // Division durch 2
        int b = x%2;      // Modulo, Rest der Division
        std::cout << x <<" / 2 = " << a << ", Rest " << b<<'\n'; // Ausgabe
        x = a;
    }
}
```

```
int main() {
    printBin(412);
}
```

Listing 4.25 Programmierbeispiel für die Umwandlung einer Ganzzahl in eine Bitfolge

Das können jedoch nur die wenigsten Menschen im Kopf. Ich selbst rechne für kleinere Zahlen im Kopf andersherum: Im Kopf ziehe ich von oben beginnend so lange die 2er-Potenzen 32, 16, 8, 4, 2 und 1 ab, bis ich bei null angekommen bin. In $_{10}25$ passt 16, dann 8, dann 1, das ergibt binär $_{2}1\ 1001$.

Der Computer rechnet bei einem `int` oft mit 32 Bit. Daher füllt er vorne mit `0` auf, für $_{10}412$ also $_{2}0000\ 0000\ \ 0000\ 0000\ \ 0000\ 0001\ \ 1001\ 1100$.

Die Arithmetik ist nun die bitweise Kombination der im Zweiersystem geschriebenen Zahlen. Wenn Sie sich die `1` als »wahr« denken und die `0` als »falsch«, können Sie bitweise die Operationen für *Und*, *Oder* und *Exklusiv-Oder* (*Xor*) selbst durchführen. Der Einfachheit halber beschränke ich mich für das Beispiel in Tabelle 4.2 auf 4 Bit für vorzeichenlose Ganzzahlen.

Operation	Binär	Dezimal
`a`	1001	9
`b`	0011	3
Und `a & b`	0001	1
Oder `a | b`	1011	11
Xor `a ^ c`	1010	10
Nicht `~b`	1100	12

Tabelle 4.2 Bitweise Arithmetik mit einem (hypothetischen) vier Bit breiten »unsigned int«

Negative Ganzzahlen und das Zweierkomplement

Bei vorzeichenbehafteten Ganzzahlen, also zum Beispiel (`signed`) `int`, repräsentiert das höchstwertige Bit das Vorzeichen; `0` steht für eine positive Zahl, `1` für eine negative. Ist das Vorzeichen positiv, steckt wie gehabt in den restlichen Bits der Wert der Zahl.

Ist es jedoch negativ, ändert sich die Interpretation der Wertbits. Diese speichert der Computer im *Zweierkomplement*. Um die Bitdarstellung einer negativen binären Zahl in eine Dezimalzahl umzuwandeln, gehen Sie wie folgt vor:

- Invertieren Sie alle Wertbits,
- wandeln Sie diese wie sonst in eine Dezimalzahl um,
- addieren Sie eins und
- setzen Sie das negative Vorzeichen.

Die Wertbits von $_2$1100 sind $_2$100, invertiert also $_2$011. Dies entspricht $_{10}$3, was nach der Addition von eins den Wert $_{10}$4 ergibt. Insgesamt repräsentiert $_2$1100 im Zweierkomplement also $_{10}$-4.

Das Zweierkomplement nutzt den Wertebereich der Datentypen optimal aus und hat sich für Berechnungen mit dem Computer als am praktischsten erwiesen. Übrigens: Ein Nebeneffekt dieser Darstellung ist, dass die kleinste darstellbare negative Zahl stets einen um eins größeren Betrag hat als die größte darstellbare positive Zahl. Der Bereich der 16 möglichen Werte für 4 Bit geht von `-8` bis `7`. Bei 8 Bit geht der Bereich von `-128` bis `127` etc.

Die Operatoren << und >> sollten Sie nun auch nicht mehr erschrecken: Schieben Sie die Bitdarstellung des ersten Operanden einfach um die Anzahl Bits des zweiten Operanden nach links oder rechts. Im Computer ist es das Gleiche, als würden Sie ebenso oft mit zwei multiplizieren oder dividieren:

- `345 << 3` ist wie `345*2*2*2` und ergibt `2760`.
- `345 >> 3` ist wie `345/2/2/2` und ergibt `43`.

Sobald Sie solche Zahlenmagie benötigen, müssen Sie sich mit der Zahlendarstellung im Computer noch einmal beschäftigen, siehe Abschnitt »Physische Repräsentation von Ganzzahlen« in Abschnitt 4.12.3. Bis dahin merken Sie sich, dass C++ diese Operationen hat.

So beschränkt sich der Einsatz in der wirklichen Welt auf die Sicht der Zahlen als Serie von Bits, denn sonst ergibt `*2` und `/2` viel mehr Sinn. Damit Sie den Einsatz mal sehen, dreht der folgende Code die Bits eines `unsigned short` um:

```
// https://godbolt.org/z/An5Thu
#include <iostream>
#include <bitset>

constexpr unsigned n_bits = sizeof(unsigned short)*8;  // 8 Bit pro char
auto bits_umdrehen(unsigned val) -> unsigned short {
  unsigned short ret = 0;
  for (unsigned i = 0; i < n_bits; ++i ) {
    ret = (ret << 1) | (val & 1);      // eins zur Seite, unterstes evtl. setzen
    val >>= 1;                         // eins in die andere Richtung
  }
  return ret;
}
```

```
void zeig(unsigned short val) {
    std::bitset<n_bits> bits{val};
    std::cout << val << "=" << bits << " -> ";
    auto lav = bits_umdrehen(val);
    std::bitset<n_bits> stib{lav};
    std::cout << lav << "=" << stib << "\n";
}

int main() {
    zeig(36u);   // Ausgabe: 36=0000000000100100 -> 9216=0010010000000000
    zeig(199u);  // Ausgabe: 199=0000000011000111 -> 58112=1110001100000000
    zeig(255u);  // Ausgabe: 255=0000000011111111 -> 65280=1111111100000000
    zeig(256u);  // Ausgabe: 256=0000000100000000 -> 128=0000000010000000
}
```

Operatoren für Streams

Die Ein- und Ausgabedatenströme der Standardbibliothek (engl. *Streams*) verwenden die eigentlich für die bitweise Arithmetik vorgesehenen Operatoren << und >> zum Schreiben und Lesen. Das haben Sie für `std::cout` und `std::cin` schon in vielen Listings gesehen.

In C++ werden << und >> heutzutage weit häufiger für Streams verwendet als für echte Bitarithmetik.

4.11.5 Zusammengesetzte Zuweisung

Wenn Sie einen arithmetischen Operator anwenden, entsteht ein neuer Wert. Benötigt der Ausdruck Zwischenergebnisse, sind diese meistens im Computerspeicher kurz vorhanden. Es werden dann innerhalb eines Ausdrucks ständig neue *Tempwerte* erzeugt und wieder verworfen. Um das zu vermeiden, gibt es alle arithmetischen Operatoren in Varianten, die stattdessen eine Variable direkt verändern.

Sie können statt `int a = 3; a = (a * 4 + 7 - 3)/4;` auch Folgendes schreiben:

```
int a = 3;
a *= 4;
a += 7;
a -= 3;
a /= 4;
```

In beiden Fällen enthält `a` dann den Wert `4`. Sie sollten aber solche langen Rechnungen vermeiden, denn der Code wird doch schnell unübersichtlich. Die Tempwerte einzusparen, ist nur in den seltensten Fällen wirklich ein großer Gewinn.

Auch die Operatoren der Bitarithmetik können Sie auf diese Weise anwenden.

```
// https://godbolt.org/z/HTceZM
#include <iostream>
#include <bitset>                       // hilft bei der Ausgabe von Zahlen als Bitfolge
int main() {
    int a = 0;
    for(int idx=0; idx<8; idx++) {
        a <<= 2;                        // um zwei Bit nach links schieben: "...100"
        a |= 1;                         // unterstes Bit setzen: "...1"
    }
    std::cout << std::bitset<16>(a) << "\n"; // 01010101010101
    std::cout << a << "\n";             // 21845
}
```

Somit sind die verfügbaren *zusammengesetzten Zuweisungen* (engl. *Compound Assignments*):

- die standardarithmetischen `+=`, `-=`, `*=`, `/=` und `%=`
- und die binärarithmetischen `|=`, `&=`, `^=`, `<<=` und `>>=`

4.11.6 Post- und Präinkrement sowie Post- und Prädekrement

Zu den unären Operatoren `++` und `--` habe ich eigentlich das Wichtigste schon gesagt. Ich fasse es noch einmal zusammen:

- Bei `++zahl` wird `zahl` zunächst um eins erhöht, bei `--zahl` um eins verringert. Das Ergebnis dieser Berechnung können Sie im umgebenden Ausdruck weiterverwenden. Weil die Operation hier zuerst ausgeführt wird, ist dies die »Prä«-Variante.
- Wenn Sie den Operator nachstellen, wenden Sie die »Post«-Variante an. Der Wert des Ausdrucks (zum Beispiel `zahl++`) ist dann der Wert der Variablen vor der Veränderung. Sie wird erst am Ende der gesamten Anweisung wirklich ausgeführt. Bis dahin muss der Computer sich den neuen Wert irgendwo merken. Das verbraucht möglicherweise Speicher und Zeit. Deswegen ist es generell besser, sich die »Prä«-Varianten anzugewöhnen.
- Noch wichtiger ist, dass Sie niemals zwei dieser Operatoren auf der gleichen Variablen innerhalb einer Anweisung anwenden. Der Compiler lässt das zu, das Ergebnis ist aber nicht definiert.

4.11.7 Relationale Operatoren

Sie können Werte auch miteinander vergleichen. Sehr häufig benötigen Sie `==` für Gleichheit und `!=` für deren Gegenteil. Zahlen können Sie natürlich auch auf kleiner und größer mit `<` und `>` vergleichen sowie in Kombinationen mit gleich: `<=` und `>=`.

Das Ergebnis eines solchen Vergleichs ist ein »wahr« oder »falsch«, also `true` oder `false`, und somit vom Typ `bool`. In Schleifen- und `if`-Bedingungen verwendet man diese dann

besonders gerne. Sie können das Ergebnis aber auch in einer bool-Variablen zwischenspeichern oder aus einer Funktion zurückgeben:

```
// als Bedingungen:
if(x < 10) ...
for(int idx=0; idx < 12; ++idx) ...
while(it != end) ...
// zwischenspeichern:
bool isLarge = value >= 100;
// zurückgeben:
bool isPositive(int a) {
    return a > 0;
}
```

Mit C++20 kommt der *Drei-Wege-Vergleich* <=> hinzu. Wegen der optischen Ähnlichkeit mit einem Raumschiff wird er gern *Spaceship-Operator* genannt. Die Rückgabe eines solchen Vergleichs ist ein neues (simples) Objekt, das mittels <, > oder == mit 0 verglichen werden kann, zum Beispiel:

```
if(a <=> b > 0) ...
```

Das entspricht in etwa dem, was in C strcmp für Strings oder in Java compareTo erledigt.

Dies ist dazu da, eigene Datentypen leichter mit einer Ordnung zu versehen. Denn früher mussten Sie <, >, ==, <=, >= und != für einen neuen Datentyp implementieren, wenn dieser komplett und geordnet sein sollte. Ab C++20 reicht es, operator<=> zu definieren, und der Compiler leitet die normalen Vergleichsoperatoren ab: Richtig implementiert, ist (a<=>b)>0 dasselbe wie a>b etc.

4.11.8 Logische Operatoren

Wenn Sie wissen wollen, ob x zwischen 100 und 200 liegt, müssen Sie beide relationalen Ausdrücke x>100 und x<200 prüfen, und es müssen beide zutreffen. Das könnten Sie mit zwei aufeinanderfolgenden ifs machen. Doch um Ausdrücke vom Typ bool miteinander zu kombinieren, gibt es die *logischen Operatoren*: Sie haben jeweils zwei Operanden und liefern wieder bool zurück. Wenn die Operanden u und v jeweils bool-Ausdrücke sind, dann ist

- u && v »wahr«, wenn u und v beide true sind,
- u || v »wahr«, wenn u oder v true sind, und
- ! u »wahr«, wenn u false ist.

Wenn Ihnen das komplett neu ist, finden Sie in Tabelle 4.9 eine ausführliche Darstellung. So können Sie dann den Ausdruck kombinieren:

```
if( x > 100 && x < 200 ) ...
```

Kurzschlussauswertung

Die obige `if`-Anweisung ist in der Tat äquivalent zu den verschachtelten `if`-Anweisungen:

```
if(x > 100)
    if(x < 200)
        ...
```

Beachten Sie, dass der Vergleich `x < 200` nur dann ausgewertet wird, wenn `x > 100` auch tatsächlich `true` war. Falls `x` zum Beispiel `2` ist, wird `x < 200` nicht erreicht.

Das ist wichtig, wenn der zweite Vergleich nur dann Sinn ergibt (oder ausgeführt werden darf), wenn der erste positiv war. Zum Beispiel:

```
if( y != 0 && x/y > 5 ) ...
```

»Und« ist immer »und«, »oder« ist immer »oder«

Wenn Sie später eigene Typen definieren, werden Sie lernen, dass Sie die Operatoren darauf ebenfalls selbst definieren können. Dazu gehören auch die logischen Operatoren.

Definieren Sie jedoch *niemals* einen logischen Operator so um, dass er etwas anderes macht, als die intuitive Berechnung auszuführen. Denn im Zusammenspiel mit der Kurzschlussauswertung wären böse Überraschungen vorprogrammiert: Teile Ihres Ausdrucks werden überraschenderweise ausgeführt.

Wenn Sie für einen eigenen Typ && oder || überladen, gibt es dort keine Kurzschlussauswertung. Auf anderen Typen als `bool` wird ein Ausdruck immer ganz ausgewertet.

Sie wissen sicherlich, dass man niemals durch null teilen darf. Wenn also in `x/y` die Variable `y` den Wert `0` hat, dann tun Sie etwas Verbotenes. Wenn Sie dies jedoch vorher prüfen, dann kann nichts mehr schiefgehen. In C++ wird

- in `u && v` der Ausdruck `v` nur dann ausgewertet, wenn `u` »wahr« ist, und
- in `u || v` der Ausdruck `v` nur dann ausgewertet, wenn `u` »falsch« ist.

Die Kurzschlussauswertung wird im Englischen *Short-Circuit Evaluation* genannt.

Alternative Token

Noch eine kleine Anmerkung zum `!`-Operator: Ich persönlich finde ihn im Quelltext etwas schwer zu sehen. Trotz seiner Unauffälligkeit kehrt er schließlich die Bedeutung des ganzen Ausdrucks komplett um. Es ist Geschmackssache, aber ich greife ab und zu auf einen Syntax-Trick zurück. Sie können das `!` durch das Wort `not` ersetzen. So wird zum Beispiel aus `while(!file.eof())…` in Listing 10.1 (Seite 240) `while(not file.eof())…`

Man nennt dies *alternatives Token*, und es gibt ein paar davon – am nützlichsten finde ich das `not`. Ich mag Ihnen gar nicht alle nennen – nicht dass Sie anfangen, sie dann überall einzusetzen. Wenn Sie neugierig sind, schauen Sie in der Sprachreferenz unter »alternative Token« oder den verwandten »Di- und Trigraphen« nach (Trigraphen sind seit C++17 jedoch nicht mehr erlaubt).

Wie gesagt, `not` ist nicht jedermanns Geschmack, und Sie sollten sich dazu in Ihrem Team absprechen. Vom breiten Einsatz dieser Zeichenkombinationen kann ich eher abraten, weil sie wirklich sehr selten eingesetzt werden. Ihre Leser könnten verwirrt werden.

4.11.9 Pointer- und Dereferenzierungsoperatoren

Sie haben schon erfahren, dass wir Methoden mit einem Punkt `.` aufrufen, zum Beispiel bei `string`:

```
void checkName(std::string& name) {
    if( name.length() ) ...
}
```

Sie werden später sehen, wenn wir Zeiger und C-Arrays besprechen, dass Sie statt der Referenz `&` auch einen *Zeiger* (engl. *Pointer*) auf diese Variable verwenden können:

```
void checkName(std::string* pname) {   // Pointer auf einen string
    if( (*pname).length() ) ...
}
```

Wenn der Typ Ihrer Variablen `string*` und nicht `string&` oder `string` ist, dann ist sie nur »indirekt« mit dem Wert verbunden. Um zum Wert zu kommen, verwenden Sie den einstelligen `*`-Operator, also hier `*pname`. Der Methodenaufruf lautet dann `(*pname).length()`. Da das jedoch etwas umständlich ist, gibt es den zweistelligen `->`-Operator als kürzere Form:

```
void checkName(std::string* pname) {
    if( pname->length() ) ...
}
```

Im Besonderen ist das alles erwähnenswert, weil beide Dereferenzierungsoperatoren (das unäre `*` und das binäre `->`) auf eigenen Typen selbst definiert werden können. Sie zu kennen, ist also nicht nur für Zeiger und C-Arrays wichtig.

Tatsächlich sind Zeiger lediglich eine sehr spezielle Form von Indirektion. Die Standardbibliothek ist durchzogen von *Iteratoren* – der Verallgemeinerung des Zeigerkonzepts. Sie werden bei den Containern und Algorithmen auf Iteratoren stoßen und dort `*` und `->` wie selbstverständlich anwenden, ohne dass Zeiger im Spiel sind. (siehe Kapitel 24, »Container«, und Kapitel 20, »Zeiger«).

4.11.10 Besondere Operatoren

Da wir hier Operatoren besprechen, müssen auch zwei Sonderlinge erwähnt werden.

Es gibt einen einzigen ternären (dreistelligen) Operator `? :`, der eine `if-else`-Abfrage als Ausdruck ermöglicht. Er hat die Form

Bedingungs-Ausdruck ? Dann-Ausdruck : Sonst-Ausdruck

Je nachdem, ob die Bedingung zu »wahr« oder »falsch« ausgewertet wird, ist entweder der »Wenn-Ausdruck« oder der »Sonst-Ausdruck« das Gesamtergebnis. Beachten Sie, dass deswegen die beiden Teile vom gleichen Typ sein müssen, damit der Compiler den Typ des Gesamtausdrucks festlegen kann.

```
// https://godbolt.org/z/TNy45i
int main() {
    for(int w1 = 1; w1 <= 6; ++w1) { // 1..6
        for(int w2 = 0; w2 < 10; ++w2) { // 1..6
            int max = w1 > w2 ? w1 : w2;  // ternärer Operator
        }
    }
}
```

Hier wird `max` der größere der beiden Werte `w1` und `w2` zugewiesen.

Das Komma kann als *Sequenzoperator* in Ausdrücken verwendet werden. Wenn Sie mehrere Ausdrücke in runde Klammern (…) schreiben und mit Kommas trennen, dann wertet der Compiler die Ausdrücke von links nach rechts aus, behält aber nur das Ergebnis des letzten Ausdrucks als Gesamtergebnis. Das Konstrukt ergibt also nur Sinn, wenn die führenden Ausdrücke Nebeneffekte haben.

Für zwei Ausdrücke sieht das so aus:

(Ausdruck-1 , Ausdruck-2)

Der Compiler berechnet zunächst *Ausdruck-1* und danach *Ausdruck-2*. Das Ergebnis des ersten Ausdrucks versickert, sodass der Gesamtausdruck den Wert und Typ des zweiten Ausdrucks erhält:

```
// https://godbolt.org/z/csJ6Wt
int main() {
    int a = 0;
    int b = 0;
    for(int w1 = 1; w1 <= 6; ++w1) { // 1..6
        for(int w2 = 0; w2 < 10; ++w2) { // 1..6
            int max = w1 > w2 ? (a+=b , w1) :( b+=1 , w2); // Sequenzoperator
        }
    }
}
```

Listing 4.26 Mit Kommas in Klammern können Sie mehrere Ausdrücke verketten.

Wenn `w1 > w2` ist, dann wird der Ausdruck `(a+=b, w1)` ausgewertet. Zwar wird `w1` als Ergebnis für `max` zurückgegeben, aber zuvor wird noch `a+=b` ausgeführt. Im Fall von `w <= w2` wird `w2` aus `(b+=1, w2)` der Variablen `max` zugewiesen, nachdem noch `b+=1` ausgewertet wurde.

Ich habe absichtlich ein besonders »skurriles« Beispiel für den Sequenzoperator gewählt, denn erstens fällt es schwer, ein sinnvolles Beispiel zu finden, und zweitens sollten Sie dieses Spezialkonstrukt meiden wie Darth Vader den Vulkan. Der Sequenzoperator lebt von Seiteneffekten, aber innerhalb eines Ausducks darf jede Variable nicht von mehr als einem Seiteneffekt betroffen sein – der Einsatz ist also gefährlich.

Es gibt Ausnahmen, in denen der Einsatz sinnvoll ist, nämlich dort, wo nur ein einzelner Ausdruck erlaubt ist und es sonst komplizierter würde.

Das kann zum Beispiel in dem Inkrementierungsteil von `for`-Schleifen der Fall sein:

```
// https://godbolt.org/z/3Opzfv
#include <iostream>
int main() {
    int arr[] = { 8,3,7,3,11,999,5,6,7 };
    int len = 9;
    for(int i=0, *p=arr; i<len && *p!=999; ++i, ++p) { // erst ++i, dann ++p
        std::cout << i << ":" << *p << " ";
    }
    std::cout << "\n";
    // Ausgabe: 0:8 1:3 2:7 3:3 4:11
}
```

Listing 4.27 In »for«-Schleifen kann das Sequenzkomma nützlich sein.

Beachten Sie, dass in einer Deklaration `int a, b, c;` das Komma nicht der Sequenzoperator ist. Das gilt auch für den Deklarationsteil der `for`-Schleife, `int i=0, *p=arr`, bei dem das Komma nur die Deklarationen voneinander trennt.

Auch das Komma, das Funktionsargumente in `func(x,y,z)` oder Listenelemente in `{1,2,3}` trennt, ist nicht der Sequenzoperator.

4.11.11 Funktionsähnliche Operatoren

Auch wenn sie nicht so aussehen, so gehören auch die folgenden Fälle zu den Operatoren:

- **art…_cast<zieltyp>(wert)**
  All dies sind *Typumwandlungen*. `wert` wird in der C++-Schreibweise in `zieltyp` umgewandelt. Es gibt die unterschiedlichen Arten `static_cast`, `const_cast`, `dynamic_cast` und `reinterpret_cast`. Sie unterscheiden sich leicht darin, was sie tun und welche Konvertierungen sie vornehmen können, doch der Typ des Ausdrucks ist immer `zieltyp`. Wenn eine Typumwandlung nicht möglich ist, meldet der Compiler einen Fehler. Ansonsten wird `wert` »passend gemacht«, was Gefahren birgt. So kann zum Beispiel bei der Umwandlung von `long` zu `short` Information verloren gehen, ohne dass Sie es mer-

ken. Typen zu erzwingen, ist nicht wünschenswert, dennoch kommt es manchmal vor. In einer optimalen Welt kommen Sie ohne explizite Typumwandlungen aus.

- **`(typ)wert`**
  Dies ist die aus C übernommene Schreibweise der Typumwandlung. In `(int)wert` versucht der Compiler, `wert` in einen `int` umzuwandeln, egal, welchen Typ `wert` hat. Je nach Kontext entspricht diese Notation einer der C++-Schreibweisen, am häufigsten dem `static_cast<>`. Alleine schon weil Sie die Art der Umwandlung hervorheben, sollten Sie immer die C++Schreibweise wählen.
- **`sizeof`**
  Mit **`sizeof`**`(typ)` und **`sizeof`**`(wert)` können Sie die Größe eines Typs oder einer Variablen in `char`-Einheiten herausfinden. Ein `sizeof(char)` liefert immer `1`.
- **`new` und `delete`**
  `new` `Klasse{}` und **`delete`** `var` verwenden Sie, um dynamischen Speicher zu verwalten, wie Sie in Kapitel 20, »Zeiger«, sehen werden.
- **`throw`**
  Mit **`throw`** `ExceptionClass{};` lösen Sie eine Ausnahme aus, was in Kapitel 10, »Fehlerbehandlung«, erklärt wird.

4.11.12 Operatorreihenfolge

Wenn Sie in einem Ausdruck mehrere Operatoren verwenden – möglicherweise unterschiedliche –, dann werden diese in einer bestimmten Reihenfolge ausgewertet.

So, wie Sie von einem ordentlichen Taschenrechner verlangen können, dass er bei `3+4*5+6` die Regel *Punkt- vor Strichrechnung* beachtet und das korrekte Ergebnis von 29 produziert, so beherrscht C++ dies auch. Darüber hinaus haben alle anderen Operatoren ebenfalls eine *Präzedenz* – je höher ein Operator in dieser Rangfolge ist, desto früher wird er im Vergleich zu anderen Operatoren ausgewertet.

Merken Sie sich diese einfache Reihenfolge, die mit der stärksten Bindung beginnt:

- multiplikative: `*`, `/` und `%`
- additive: `+` und `-`
- Streamoperatoren: `<<` und `>>`
- Vergleiche `<`, `<=`, `>` und `>=`
- Gleichheit `==` und `!=`
- logische, in dieser Reihenfolge: `&&`, `||`
- Zuweisungen mit `=`, aber auch alle zusammengesetzten wie `+=`

So können Sie so manchen komplexen Ausdruck schreiben, ohne mit Klammern die Bedeutung korrigieren zu müssen:

```
bool janein = 3*4 > 2*6 && 10/2 < 13%8;
```

spart Ihnen die Klammern:

```
bool janein = (((3*4) > (2*6)) && ((10/2) < (13%8)));
```


Beachten Sie, dass der Streamausgabeoperator `<<` loser bindet als normale Arithmetik mit `+` und `*`. Haben Sie aber Vergleiche in der Ausgabe, benötigen Sie Klammern um den Vergleich:

```
std::cout << 2*7 << x+1 << n/3-m << "\n"; // keine Klammern zwischen << nötig
std::cout << (x0 >= x1) << (a<b || b<c);  // Klammern: << würde enger binden
```

Aber was passiert, wenn mehrere Operatoren der gleichen Präzedenz nebeneinanderstehen? In Ausdrücken wie `10-5-2` wird das linke `-` zuerst ausgewertet, denn `-` ist *linksassoziativ* – als wäre der Ausdruck `((10-5)-3)` geklammert. Das Ergebnis ist also `2`. Alle zweistelligen arithmetischen, booleschen und vergleichenden Operatoren sind linksassoziativ, sodass Sie intuitiv damit rechnen können. Manchmal wird dies auch *links-nach-rechts-assoziativ* oder *LR-assoziativ* genannt.

Die Gruppe der Zuweisungsoperatoren wiederum ist durchgehend *rechtsassoziativ*, weswegen der Compiler für `x += y += z += 1` erwartungsgemäß `(x += (y += (z += 1)))` ausführt und zuerst `z` um `1` inkrementiert, um sich dann nacheinander mit den anderen Variablen zu beschäftigen. Würde der Ausdruck wie `(((x += y) += z) += 1)` ausgewertet, dann würde `x` mehrmals einen neuen Wert erhalten, denn `x += y` gibt ja auch `x` zurück, und darauf würde dann `+= z` ausgeführt; `y` würde nicht verändert. Und noch einmal würde `x` zurückgeliefert, und `+= 1` inkrementierte dieses um `1`, anstatt `z` zu erhöhen.

Meistens funktionieren Präzedenz und Assoziativität intuitiv und wie erwartet. Im Zweifelsfall klammern Sie besser, denn spätere Leser stellen sich wahrscheinlich die gleichen Fragen wie Sie.

4.12 Eingebaute Datentypen

Bisher haben Sie Typen nur im Überflug kennengelernt – und drumherum dabei genau so viel fundamentales C++, wie zum Verständnis bisher nötig war.

Auf diesen Grundlagen baut dieses Kapitel auf. Sie wissen nun genug über die Sprache, um die umfassenden Erklärungen aller eingebauten Typen verstehen.

In diesem Abschnitt lernen Sie, welche Datentypen Ihnen in C++ zur Verfügung stehen, wenn Sie kein `#include` in Ihrem Programm nutzen bzw. nicht die *Standardbibliothek* verwenden: Welches sind die *eingebauten Datentypen*, und was kann man mit ihnen machen?

Achten Sie im folgenden Beispiel auf die Typen der Variablen und Funktionsparameter:

```
// https://godbolt.org/z/LLQVov
#include <iostream>    // cin, cout für Eingabe und Ausgabe

void eingabe(unsigned &gebTag_,
             unsigned &gebMonat_,
             unsigned &gebJahr_,
             unsigned long long &steuernummer_,
             double &koerperlaenge_)
{
    /* Eingaben noch ohne gute Fehlerbehandlung... */
    std::cout << "Geb.-Tag: "; std::cin >> gebTag_;
    std::cout << "Geb.-Monat: "; std::cin >> gebMonat_;
    std::cout << "Geb.-Jahr: "; std::cin >> gebJahr_;
    std::cout << "Steuernummer: "; std::cin >> steuernummer_;
    std::cout << "Koerperlaenge: "; std::cin >> koerperlaenge_;
}
int main() {
    /* Daten */
    unsigned gebTag_ = 0;
    unsigned gebMonat_ = 0;
    unsigned gebJahr_ = 0;
    unsigned long long steuernummer_ = 0;
    double koerperlaenge_ = 0.0;
    /* Eingabe */
    eingabe(gebTag_, gebMonat_, gebJahr_, steuernummer_, koerperlaenge_);
    /* Berechnungen */
    // ...
}
```

Listing 4.28 Hier werden einige neue Datentypen verwendet.

Ich habe hier die folgenden Datentypen eingesetzt:

- `unsigned` und `unsigned long long` als Vertreter der *Ganzzahldatentypen*
- `double` zum Speichern einer (Fließ-)Kommazahl
- die Variablen `std::cin` und `std::cout`, deren Typ nicht explizit erwähnt wird

Die erste wichtige Unterscheidung zwischen diesen Datentypen ist, ob sie eingebaut sind oder nicht.

4.12.1 Übersicht

Wenn Sie kein `#include` verwenden, steht Ihnen eine sehr begrenzte Menge an Datentypen zur Verfügung. Sie können zum Beispiel mit `class` Typen hinzudefinieren oder mit `typedef` Aliase erzeugen, aber Ihre Auswahl ist dann sehr übersichtlich:

- **Ganzzahlen – `int`, `short`, `long`, `long long`, jeweils `signed` oder `unsigned`**
  Ganzzahlen speichern Zahlen ohne Komma, also zum Beispiel `3`, `-12` und `987654321`. Die unterschiedlich großen Varianten speichern entweder vorzeichenbehaftet oder vorzeichenlos jeweils einen Zahlenbereich. Überlegen Sie bei jeder Anwendung, welche Variante Sie benötigen. Im Zweifel ist `int` eine gute Wahl. Nach der Wahl des Typs müssen Sie vor allem darauf achten, dass Sie den Zahlenbereich des Typs einhalten, um einen gefährlichen *Überlauf* zu vermeiden.
- **Fließkommazahlen – `double`, `float` und `long double`**
  Für manche Dinge sind Ganzzahlen ungeeignet. Verwenden Sie `double`, um Dezimalzahlen wie `3.14`, `-0.00001`, `6.281e+26` zu speichern. Doch auch deren Raum ist begrenzt, selbst wenn er groß ist. Ihr größtes »Problem« ist aber die *Genauigkeit* – selbst einen Wert von $^1/_3$ können Sie nicht exakt speichern. Verwenden Sie Fließkommazahlen immer mit Vorsicht.
- **Wahrheitswerte – `bool`**
  Mit `bool` gibt es einen Datentyp zum Speichern von Wahrheitswerten. Ein `bool` ist entweder `true` oder `false`. Sie haben diesen Typ in `if`-Anweisungen und Ähnlichem schon oft verwendet. Auch wenn für `bool` eigentlich ein einziges Bit zum Speichern ausreichen würde, ist dieser Datentyp mitnichten platzsparender als `char`.
- **Zeichentypen – `char`, `char16_t`, `char32_t` und `char8_t`**
  In C++ gibt es an wenigen Stellen eine wirkliche Unterscheidung zwischen Zahlen und Zeichen, sodass Sie auch den Zeichentyp `char` als Zahl verwenden können. Dieser bildet die kleinste Einheit, und wenn Sie mit `sizeof(x)` nach der wirklichen Größe eines Typs oder einer Variablen fragen, dann ist es die Anzahl an `char`-Einheiten, die Ihnen zurückgegeben wird. Zum Zeichentyp wird `char`, weil Sie in `char`-Sequenzen Zeichenketten speichern – entweder als Array in `char[]` oder in `string` aus der Standardbibliothek. Da `char` nicht für internationale Zeichen ausgelegt ist (Unicode), gibt es `char16_t`, `char32_t` und das etwas veraltete `wchar_t` mit mehr Platz. Beispiele für Zeichen sind `'a'`, `'Z'` sowie internationale (»Unicode«-)Zeichen. In C++20 kommt `char8_t` hinzu.
- **Referenzen – `&`**
  Indirektionen mit `&` auf eine andere Variable. So können Sie dieselbe Variable dann unter einem anderen Namen ansprechen, zum Beispiel `int x = 7; int& ref = x;`. Änderungen an der Referenz, zum Beispiel `ref=12`, verändern tatsächlich das Original, hier `x`. Eine Referenz verweist, solange sie existiert, immer auf dieselbe Variable. Eine Referenz kann nicht auf »nichts« verweisen.
- **Zeiger und C-Arrays – `*` und `[]`**
  Indirektionen mit `*` auf eine Variable als Speicheradresse, im Fall eines C-Arrays als `[]` mit Längeninformation. Eine Zeichenkette wie `"Ihr Name"` ist ein Beispiel für `char[9]`[6] und wird an Funktionen als `char*` übergeben. Wir besprechen diese erst in Kapitel 20,

6 Solche Zeichenketten werden intern mit einem Ende-Zeichen abgeschlossen, daher ist die Arraylänge 9 und nicht 8, wie die die sichtbaren Zeichen.

»Zeiger«. Im Unterschied zu Referenzen mit `&` können Zeiger verändert werden, sodass sie auf eine andere Variable verweisen. Verweisen Sie auf `nullptr`, steht das für den Verweis auf »nichts«.

Im Quellcode werden Daten durch Variablen (oder Ähnliches) oder Literale repräsentiert. Wenn das kompilierte Programm aber läuft, sieht das ganz anders aus. Dann benötigen die Daten eine physische Repräsentation: im Prozessor, wo mit ihnen gerechnet wird, und im Speicher, wo sie geladen, gespeichert und verändert werden. Speicher kann hier vieles bedeuten: vom schnellen Register über den Zwischenspeicher Cache, den Hauptspeicher und Grafikkartenspeicher bis zur Festplatte und im weiteren Sinne auch Netz oder Cloud.

Jeder Datentyp hat eine physische Repräsentation, die der Prozessor versteht. In diesem Kapitel gehen wir auf diese Repräsentation kurz ein, denn C++ ist auch eine Sprache, die nah an dieser Repräsentation dran ist und sie direkt manipulieren kann.

4.12.2 Eingebaute Datentypen initialisieren

Einer der wichtigsten Unterschiede dieser Datentypen im Vergleich zum Großteil der anderen – nicht eingebauten – Datentypen ist, dass Sie die Variablen dieser Typen *initialisieren müssen*! Schreiben Sie zum Beispiel `int x = 7;` oder `int x{7};`. Wenn Sie die Initialisierung mit `7` weglassen, wird `x` gar nicht initialisiert. Sein anfänglicher Wert ist dann zufällig, was Folgen haben kann.

Wollen Sie es nicht dem Zufall, sondern dem Compiler überlassen, einen »guten« Wert für die Initialisierung auszusuchen, dann schreiben Sie ein Paar leere geschweifte Klammern `{}` hinter die Variable. Die Faustregel ist, dass dann mit null oder etwas Äquivalentem initialisiert wird.

```
int x{};           // initialisiert x mit 0 – eingebauter Typ
double y{};        // y wird 0.0 – eingebauter Typ
std::string s{};   // leerer String – Klasse
struct tm mytm{}; // alle Felder 0 – tm ist Aggregat aus <time.h>
```

Eine Wert-Initialisierung überlässt nichts dem Zufall

Die Regel, immer mindestens mit `{}` zu initialisieren, wenn Sie keine andere sinnvollere Initialisierung haben, können Sie sich angewöhnen, denn auch die nicht eingebauten Typen profitieren davon. Sie machen also nichts falsch, wenn Sie dies in Ihr Repertoire aufnehmen. Man nennt es *Wert-Initialisierung*, was so viel heißt wie »Initialisierung mit einem *sinnvollen* typspezifischen Wert«. Bei Typen, die nichts weiter mit `{}` vorsehen, erhalten Sie so zumindest etwas anderes als kosmisches Rauschen im Computerspeicher.

Unter den nicht eingebauten Typen gibt es einige wenige Ausnahmen, die die Initialisierung mit `{}` nicht vertragen. Auf die wird der Compiler Sie hinweisen. Es handelt sich dann meist um eine Klasse, bei der Sie Argumente zwischen den Klammern angeben müssen.

4.12.3 Ganzzahlen

Wir haben im Beispiel an mehreren Stellen den Typ `unsigned` verwendet. Dabei handelt es sich nur um eine Abkürzung von `unsigned int`. Gegenüber dem normalen `signed int`, abgekürzt `int`, kann dieser etwas größere positive Zahlen speichern, aber keine negativen.

Normalerweise sollten Sie für »ganze Zahlen« den Typ `int` verwenden. Der Zahlenbereich ist jedoch eingeschränkt – benötigen Sie mehr, stehen Ihnen `long` und `long long` zur Verfügung. Müssen Sie Platz sparen, gibt es `short`. Alle diese Varianten stehen Ihnen auch in der `unsigned`-Variante zur Verfügung, zum Beispiel `unsigned long`. Stattdessen `signed` vor den Zahlentyp zu schreiben, ist nicht nötig, da dies der Default ist. Ein `int` (ob `signed` oder `unsigned`) ist das, womit Ihr System am natürlichsten umgehen und meist am schnellsten rechnen kann.

Mit den Ganzzahltypen können Sie arithmetisch rechnen – also multiplizieren und dergleichen. Sie stoßen aber schon mit den Grundrechenarten an die Grenzen des Datentyps, denn logischerweise kann ein `unsigned int` keinen negativen Wert annehmen. Und sowohl die `signed`- als auch die `unsigned`-Varianten haben für große Zahlen eine Grenze.

Um zu verstehen, was beim Rechnen mit `int`-Typen und Verwandten passiert, müssen Sie nur wissen, dass der Computer im *Binärsystem* arbeitet. Das ist nichts anderes als das *Dezimalsystem*, das Sie gewohnt sind, nur dass an jeder Stelle nicht `0` bis `9` stehen kann, sondern lediglich `0` oder `1` – einer von zwei möglichen Werten (daher »bi«-när). Im vorigen Abschnitt 4.11, »Operatoren«, bin ich im Kasten »Binärsystem« darauf eingegangen.

Die einzelnen Positionen, die 0 oder 1 annehmen können, nennt man *Bits*. Wie viele Bits in eine Variable eines Ganzzahltyps passen, hängt von Ihrem System ab. Der Standard sagt, dass `char` der kleinste und `long long` der größte Typ ist, mit `short`, `int` und `long` in dieser Reihenfolge dazwischen. Auf heute üblichen Maschinen hat `char` 8 Bit und `long long` 64 Bit. Schauen Sie in Tabelle 4.3 für Beispiele zweier heute üblicher Systeme. Die Windows-Spalte gilt sowohl für 32- als auch für 64-Bit-Windows, ebenso für 32-Bit-Linux.

Bit	Linux, 64 Bit	Windows	Minimum für
8	char	char	char
16	short	short	short, int
32	int	int, long	long
64	long, long long	long long	long long

Tabelle 4.3 Bitbreiten zweier Architekturen und das Minimum als Beispiele

Wenn ein `unsigned int` 32 Bit hat, dann ist die größte Zahl, die dieser Datentyp darstellen kann, $1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^{1} + 1 \times 2^{0} = 2^{32}–1$, was etwas mehr als 4 Milliarden entspricht. Das ist in Tabelle 4.4 dargestellt. Sie müssen durch Ihr Programm selbst dafür sorgen, dass ein Ausdruck vom Typ `int` niemals einen größeren Wert annimmt.

> **C++20: Mindestbreiten für Ganzzahltypen**
>
> Seit C++20 gelten für die in der Spalte »Minimum für« angegebenen Datentypen die unter »Bit« gelisteten Zahlen als Mindestbreite der Implementierung. Sie müssen sich aber nicht umgewöhnen, denn de facto hielten alle Implementierungen diese Breiten schon vor der Standardisierung ein. Nur werden Sie wahrscheinlich noch lange Leute finden, die der Meinung sind, dass C++ außer für `char` keine Mindestbreiten vorgibt. Hier können Sie nun mit neuem Wissen glänzen.

Eine Anmerkung zu den Namen der Ganzzahltypen. Offiziell heißen die Typen `int`, `unsigned int`, `long int` und `long long int`. Ich selbst kürze `unsigned int` gewohnheitsmäßig mit `unsigned` ab, aber Sie werden es häufig ausgeschrieben sehen. Bei `long` und `long long` wird die ausgeschriebene Version mit `int` dahinter nur selten verwendet.

Bit	unsigned	signed
8	0 … 255	–128 … 127
16	0 … 65 535	–32 768 … 32 767
32	0 … 4 294 967 295	–2 147 483 648 … 2 147 483 647
64	0 … 18 446 744 073 709 551 615	–9 223 372 036 854 775 808 … 9 223 372 036 854 775 807

Tabelle 4.4 Zahlenbereiche

Ist Ihnen aufgefallen, dass die Anzahl der Bits, die Ganzzahlen breit sein können, durch acht teilbar ist? Ich sage Ihnen wahrscheinlich auch nichts Neues damit, dass in den meisten Computersystemen 8 Bit ein *Byte* ausmachen (es gibt Ausnahmen). Ein Byte ist in C++ die kleinste *adressierbare* Dateneinheit. Ein Byte im Speicher hat eine Adresse x, das Byte daneben Adresse x+1.

Auf einem typischen System hat ein `unsigned int` 32 Bit, also 4 Byte, wie Sie in Tabelle 4.3 sehen. Die Einheit, die auf einer gegebenen Architektur besonders vorteilhaft ist, nennt man ein *Datenwort* (oder einfach engl. *word*). Lassen Sie uns für dieses Buch davon ausgehen, dass 4 Byte ein Datenwort ausmachen. Auf anderen Architekturen ist das anders.

Daraus leiten sich dann die Begriffe *Halbwort*, *Doppelwort* und *Vierfachwort* ab, die 16 Bit, 64 Bit (8 Byte) und 128 Bit (16 Byte) umfassen. Die englischen Bezeichnungen *half word*, *double word* und *quad word* (oder kurz *qword* oder *quad*) sind wahrscheinlich häufiger zu finden. Wenn man von einer kleineren Wortgröße ausgeht, gibt es auch den Begriff Achtfach- oder Doppelvierfachwort. Statt *half word* oder *double word* wird oft *short word* oder *long word* verwendet, und da sieht man dann die Nähe von C und C++ zur Maschinenre-

präsentation. Es ist jedoch besondere Vorsicht geboten, denn hier divergieren Linux- und Windows-Jargon – speziell *long word* kann auch mal das Gleiche bedeuten wie *word*.

Zu guter Letzt spricht man noch vom *Nibble*, mit dem man ein halbes Byte meint: Das obere Nibble bezeichnet die vier höherwertigen, das untere Nibble die vier niederwertigen Bits. Ein Nibble können Sie mit einem hexadezimalen Zeichen darstellen, siehe Tabelle 4.6.

Wie schon gesagt, sind die Begrifflichkeiten auf unterschiedlichen Architekturen eventuell verschoben. Die Begriffe zu kennen, ist dennoch wichtig, weil sie oft in Dokumentationen stehen – Schnittstellenbeschreibungen zu spezialisierter Hardware wie Grafikkarten beziehen sich darauf (und haben oft ihre eigene Interpretation der Breiten). Hardwarenahe Werkzeuge beziehen sich ebenfalls auf diese Begriffe. Im Assemblercode (AT&T- oder GAS-Syntax) auf einem x86-64-Linux bedeutet `imull` zum Beispiel »multiply long« und wird eingesetzt, wenn ein C++-`int` oder -`short` multipliziert wird. Ein `imulq` für »multiply quad« multipliziert `long` oder `long long`. Der gleiche Code auf einem ebenfalls 64-bittigen MIPS verwendet `mul` für `short` und `int` und `dmult` für `long` und `long long`. Damit will ich sagen, dass Sie immer aufpassen müssen, was das von Ihnen eingesetzte Werkzeug auf Ihrer Architektur gerade meint. In Listing 4.29 habe ich den gleichen Code auf zwei verschiedenen 64-Bit-Architekturen in Assembler übersetzen lassen. Beachten Sie die Suffixe der Befehle, die sich auf die Operandenbreite beziehen.

```
C++ ____________________________     x86-64-Linux ______     MIPS64 __________
long long sq(long long n) {          sq(long long):          sq(long long):
  return n * n;                        imulq %rdi, %di         dmult $4,$4
}                                      movq  %rdi, %rax        j     $31
                                       ret                     mflo  $2
long sq(long n) {                    sq(long):               sq(long):
  return n * n;                        imulq %rdi, %di         dmult $4,$4
}                                      movq  %rdi, %rax        j     $31
                                       ret                     mflo  $2
int sq(int n) {                      sq(int):                sq(int):
  return n * n;                        imull %edi, %di         j   $31
}                                      movl  %edi, %eax        mul $2,$4,$4
                                       ret
short sq(short n) {                  sq(short):              sq(short):
  return n * n;                        imull %edi, %di         andi $4,$4,0xffff
}                                      movl  %edi, %eax        mul  $2,$4,$4
                                       ret                     j    $31
                                                               seh  $2,$2
char sq(char n) {                    sq(char):               sq(char):
  return n * n;                        movl  %edi, %eax        andi $4,$4,0xff
}                                      imull %edi, %eax        mul  $2,$4,$4
                                       ret                     j    $31
                                                               seb  $2,$2
```

Listing 4.29 Selbst auf zwei 64-Bit-Architekturen können unterschiedliche Werkzeuge für Wortgrößen unterschiedliche Begriffe wählen.

Ob nun Wörter oder Bytes, im Speicher liegen sie nebeneinander. Angenommen, ein `unsigned int` hat 32 Bit, dann ist die abstrakte Ganzzahl 123 456 789 in 4 Byte mit den Werten 7, 91, 205 und 21 gespeichert. Sie können daraus wieder den ursprünglichen Wert errechnen, indem Sie die einzelnen Werte als Positionen in einem 256er-Zahlensystem auffassen, also $7 \times 256^3 + 91 \times 256^2 + 205 \times 256^1 + 21 \times 256^0$.

Die Werte von Bytes werden selten als Dezimalzahlen geschrieben. Es ist sehr verbreitet, sie als Hexadezimalzahlen zu schreiben, also `0x07`, `0x5b`, `0xcd` und `0x15`. So hat jedes Byte genau zwei Buchstaben (plus `0x`-Präfix). Fügt man die zusammen, ergibt sich eine 8 Zeichen lange Hexadezimalzahl: `0x075bcd15` oder ohne führende Null auch `0x75bcd15`.

Mit dieser Notation können Sie nun wunderschön eine dicht gepackte Folge von `int`-Werten im Speicher darstellen, siehe Abbildung 4.2

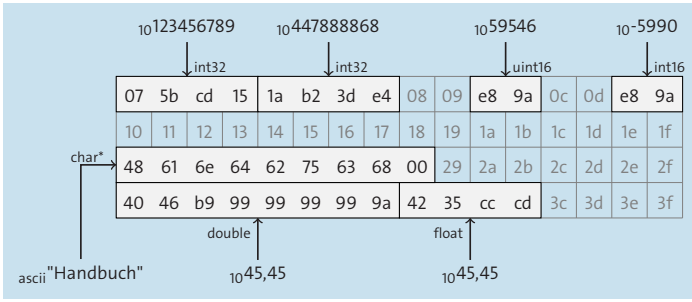


Abbildung 4.2 Daten im Speicher (Big Endian)

Die beiden dezimalen Zahlen 123 456 789 und 447 888 868 zu Beginn sind direkt nebeneinander abgelegt. Der Inhalt der Speicherzellen ist in hexadezimalen Zahlen dargestellt.

Speicherausrichtung

Es ist nicht ganz zufällig, dass die Datenelemente in der Abbildung immer so abgelegt sind, dass ihr erstes Byte auf einer durch vier teilbaren Adresse liegt. Man nennt das *Speicherausrichtung* (engl. *Alignment*). Vier Byte entsprechen einem 32-Bit-Alignment. Die meisten modernen Maschinen funktionieren schneller, wenn das der Fall ist. Um die optimale Geschwindigkeit zu erhalten, werden Daten nötigenfalls nicht dicht nebeneinandergepackt, sondern mit »leeren« Bytes dazwischen verschoben. Im Beispiel ist dies bei 59 546 und –5 990 der Fall: Die Adressen `0c` und `0d` dazwischen enthalten keine sinnvollen Daten.

Manche Maschinen kommen sogar gar nicht damit klar, wenn die Daten bestimmter Datentypen nicht mit passendem Alignment abgelegt werden. Dann muss man beim Lesen die Daten erst zeitraubend kopieren. Neben 32-Bit-Alignment gibt es auch 16-Bit-, 64-Bit- und sogar 128-Bit-Alignment. Kein Alignment bedeutet, die Startposition egal ist, entspricht 8-Bit-Alignment und kommt ebenfalls vor.

Um die Speicherausrichtung müssen Sie sich normalerweise nicht kümmern, das erledigt der Compiler für Sie.

Zur Veranschaulichung enthält die Abbildung noch weitere Daten. Die hexadezimale Bytefolge `0xe89a` wird für einen `uint16_t` (zum Beispiel `unsigned short`) als 59546 interpretiert. Wenn die Bytefolge mit einem `int16_t` (zum Beispiel `short`) gelesen wird, entspricht das –5990. Der C-String `"Handbuch"` hat ein abschließendes `\0`-Byte. Welche Buchstaben welches hexadezimale Byte bedeuten, ist im *ASCII-Standard* festgelegt. Zum Schluss habe ich die dezimale Fließkommazahl 45,45 noch als `double` und `float` hexadezimal abgelegt.

Bytereihenfolge

Für den Rest dieses Abschnitts möchte ich das `0x`-Präfix weglassen. Wenn nicht anders erwähnt, sind Dezimalzahlen wie 123 in normaler Schrift gesetzt, Hexadezimalzahlen wie `a87f` in Listingschrift.

Die Zahl 447888596 ergibt in Bytes aufgeteilt also `1a`, `b2`, `3c` und `d4`. An Adresse x speichere man `1a` und an Adresse x+1 `b2`, bei x+2 `3c` und zuletzt bei x+3 `d4` – oder zusammengeschrieben an Adressen x..x+3 `1a b2 3c d4`. In Abbildung 4.3 habe ich das dargestellt. Nummeriert man die acht Hexziffern von ihrer niedrigsten Wertigkeit so, dass die Ziffer mit dem höchsten Stellenwert die `8` bekommt und die mit dem niedrigsten die `1`, ist die Nummerierung also `87 65 43 21`.

Diese Speicherung kommt uns natürlich vor, weil sie wie unser Zahlensystem die höchstwertige Stelle der gesamten Zahl zuerst nennt und die niederwertigste zuletzt – genauso wie wir bei einer 123 der ersten Ziffer 1 die höchste Wertigkeit zuordnen und der letzten 3 die niedrigste. Diese Art der Speicherung nennt man *Big Endian* (frei übersetzt »Großender«). Sie speichert das *Most Significant Byte* zuerst, abgekürzt *MSB*.

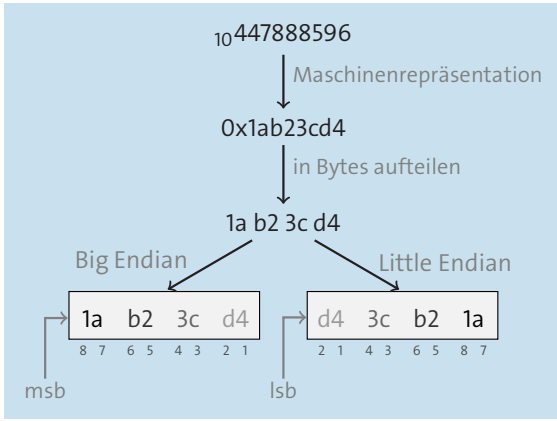


Abbildung 4.3 Zahlen können mit dem höherwertigen oder dem niederwertigen Byte zuerst gespeichert werden.

Die andere Möglichkeit, die Zahl 447888596 zu speichern, ist *Little Endian* – das niederwertigste Element zuerst. Das ergibt dann im Speicher an den Adressen x bis x+3 die Werte `d4 3c b2 1a`. Es kommt also das *Least Significant Byte* zuerst, kurz *LSB*. Meinem persönlichen