AKTUELL ZU C++ 17

dirk LOUIS

2. Auflage

C++

Das komplette Starterkit für den einfachen Einstieg in die Programmierung

HANSER

Inklusive Einführung in Visual Studio (Windows) und den GNU-Compiler (Linux)

Louis

C++

Dirk Louis

C++

Das komplette Starterkit für den einfachen Einstieg in die Programmierung

2. Auflage

HANSER

Der Autor:
Dirk Louis, Saarbrücken, autoren@carpelibrum.de

Bibliografische Information der Deutschen Nationalbibliothek:

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

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Petra Kienle, Fürstenfeldbruck
Umschlagdesign: Marc Müller-Bremer, München, www.rebranding.de
Umschlagrealisation: Stephan Rönigk
Layout: Kösel Media GmbH, Krugzell
Druck und Bindung: Hubert & Co. GmbH & Co. KG BuchPartner, Göttingen
Printed in Germany

Print-ISBN: 978-3-446-44597-0
E-Book-ISBN: 978-3-446-45388-3

Inhalt

Vorwort

Sie besitzen einen Computer und wissen nicht, wie man programmiert?

Das ist ja furchtbar! Jetzt erzählen Sie mir nur nicht, dass Sie Ihren Computer nur zum Spielen und Internetsurfen benutzen. Dann wäre ich wirklich enttäuscht.

Ach so, jemand hat Ihnen erzählt, dass Programmieren sehr kompliziert sei und viel mit Mathematik zu tun hätte.

Tja, dann wollte sich dieser jemand wohl ein wenig wichtigmachen oder hat selbst nichts vom Programmieren verstanden, denn Programmieren ist nicht schwieriger als Kochen oder das Erlernen einer Fremdsprache. Und mehr mathematisches Verständnis als es von einem Schüler der sechsten oder siebten Klasse verlangt wird, ist definitiv auch nicht nötig.

Reizen würde Sie das Programmieren schon, aber Sie wissen ja gar nicht so recht, was Sie programmieren könnten.

Keine Angst, sowie Sie mit dem Programmieren anfangen, werden Ihnen zahlreiche Ideen kommen. Und weitere Anregungen finden sich in guten Lehrbüchern zuhauf beziehungsweise ergeben sich beim Austausch mit anderen Programmierern.

Immer noch Zweifel? Sie würden sich das Programmieren am liebsten im Selbststudium beibringen? Sie suchen ein Buch, das Sie nicht überfordert, mit dem Sie aber dennoch richtig professionell programmieren lernen?

Aha, Sie sind der unsicher-anspruchsvolle Typ! Dann dürfte das vorliegende Buch genau richtig für Sie sein. Es ist nicht wie andere Bücher primär thematisch gegliedert, sondern in Stufen – genauer gesagt vier Stufen, die Sie Schritt für Schritt auf ein immer höheres Niveau heben.

Aufbau des Buchs

Die **erste Stufe** ist der Grundkurs. Hier schreiben Sie Ihre ersten Programme und lernen die Grundzüge der Programmierung kennen. Danach besitzen Sie fundiertes Basiswissen und können eigenständig Ihre ersten Programmideen verwirklichen.

Bestimmte Aufgaben kehren bei der Programmierung immer wieder: beispielsweise das Abfragen des Datums, die Berechnung trigonometrischer Funktionen, die Verwaltung größerer Datenmengen oder das Schreiben und Lesen von Dateien. Für diese Aufgaben gibt es in der C++-Standardbibliothek vordefinierte Elemente. Wie Sie diese nutzen, erfahren Sie im Aufbaukurs – der **zweiten Stufe**.

Die **dritte Stufe** stellt Ihnen die objektorientierte Programmierung vor und lehrt Sie, wie Sie den Code immer größerer Programme sinnvoll organisieren.

Die **vierte Stufe** schließlich stellt Ihnen noch einige letzte C++-Techniken vor, die Sie vermutlich eher selten einsetzen werden, die ein professioneller C++-Programmierer aber kennen sollte.

Abgerundet wird das Buch durch einen umfangreichen **Anhang**, der unter anderem eine C++-Syntaxübersicht und eine Kurzreferenz der Standardbibliothek beinhaltet.

Nicht verzagen!

Natürlich gibt es auch Zeiten des Verdrusses und des Frusts. Oh ja, die gibt es! Aber seien wir ehrlich: Wäre der Weg nicht so steinig, wäre die Freude am Ziel auch nicht so groß. Was sind das denn für trostlose Gesellen, die in ihrer gesamten Zeit als Programmierer noch keine Nacht durchwacht haben, weil sie den Fehler, der das Programm immer zum Abstürzen bringt, nicht finden konnten? Und was soll man von einem Programmierer halten, der noch nie aus Versehen ein Semikolon hinter eine if-Bedingung gesetzt hat? (Und dem die Schamesröte ins Gesicht schoss, als er einen vorbeikommenden Kollegen um Hilfe bat und ihn dieser nach einem flüchtigen Blick auf den Quellcode auf den Fehler aufmerksam machte.) Sind das überhaupt echte Programmierer?

Wer programmieren lernen will, der muss auch erkennen, dass bei der Programmierung nicht immer alles glatt geht. Das ist nicht ehrenrührig, man darf sich nur nicht unterkriegen lassen. Sollten Sie also irgendwo auf Schwierigkeiten stoßen – sei es, dass Sie etwas nicht ganz verstanden haben oder ein Programm nicht zum Laufen bekommen –, versuchen Sie sich nicht zu sehr in das Problem zu verbohren. Legen Sie eine kleine Pause ein oder lesen Sie erst einmal ein wenig weiter – oftmals klärt sich das Problem danach von selbst.

Dirk Louis

Saarbrücken, im Frühjahr 2018

I Teil I: Grundkurs

In diesem Teil erlernen Sie die Grundlagen von C++ und der Programmierung im Allgemeinen.

Das bedeutet, dass wir eine Menge Stoff zu bewältigen haben. Doch wir werden es gelassen angehen. Zuerst eigenen wir uns ein wenig Theorie an, anschließend betrachten wir die technische Seite der Programmerstellung und ab Kapitel 3 tauchen wir dann so richtig in die C++-Programmierung ein.

1 Keine Angst vor C++!

C++ steht in dem Ruf, eine besonders mächtige und leistungsfähige, aber leider auch eine sehr schwer zu erlernende Programmiersprache zu sein. Letzteres ist wohl darauf zurückzuführen, dass die vielfältigen Möglichkeiten und die Freiheiten, die C++ dem Programmierer bietet, einer ebenso großen Zahl an Konzepten, Techniken und unterschiedlichen Syntaxformen gegenüberstehen. Und gerade diese Syntaxformen – das lässt sich nicht leugnen – können auf Anfänger schon recht abschreckend wirken. Einige Kostproben gefällig? Zeilen der Form:

```
virtual const char* f() const noexcept;
```

sind in C++ keineswegs unüblich und auch Berechnungen der Form:

```
i = m++*+n;
```

sind möglich. Besondere Freude bereiten aber Deklarationen wie z. B.:

```
int *(*f(int))(int, int);
```

Falls Ihnen jetzt Zweifel kommen, ob Sie mit C++ wirklich die richtige Wahl getroffen haben, so lassen Sie sich versichern:

- C++ ist viel einfacher, als es manchmal den Anschein hat.
- Schon bald werden Sie Ihre eigenen C++-Programme schreiben.
- Mit jedem Programm, das Sie schreiben, wird Ihnen C++ vertrauter und selbstverständlicher erscheinen.

Am Ende dieses Buchs werden Sie nicht nur in der Lage sein, attraktive und professionelle Programmvorhaben anzugehen, Sie werden auch verstehen, was die erste der obigen Beispielzeilen bedeutet, und Sie werden den Kopf darüber schütteln, warum der Verfasser der eigentlich doch ganz einfachen zweiten Beispielzeile den Code nicht lesefreundlicher formatiert hat.

Nur die Bedeutung der dritten Beispielzeile werden Sie nach der Lektüre dieses Buchs immer noch nicht verstehen. Aber trösten Sie sich: Die Konstruktion, die in dieser Zeile deklariert wird[1], ist so abgehoben, dass Sie unter zehn professionellen C++-Programmie-

[1] eine Funktion, die einen Funktionszeiger als Rückgabetyp besitzt

rern vermutlich höchstens einen finden werden, der diese Konstruktion erkennt, geschweige denn sie selbst schon einmal eingesetzt hätte.

Warum aber ist C++ so mächtig? Warum gibt es so viele Konzepte in der Sprache und warum ist die Syntax so kryptisch? Die Antwort auf diese Fragen liegt in der Geschichte von C++.

1.1 Von C zu C++

„The times they are a changin'" – die Zeiten ändern sich – heißt es in einem berühmten Song von Bob Dylan. Sicherlich hatte Dylan dabei nicht die Entwicklungen in der IT-Branche und der Softwareerstellung im Auge, doch allgemeine Wahrheiten lassen sich eben auf viele verschiedene Bereiche anwenden – und manchmal eben auch auf den Bereich der Softwareentwicklung.

Dort hat im Laufe der letzten Jahrzehnte tatsächlich ein grundlegender Wandel stattgefunden.

1.1.1 Rückblick

Wann die Entwicklung des Computers, der Rechenmaschine, begonnen hat, ist gar nicht so leicht zu sagen. Es hängt sehr davon ab, wie weit man zurückgehen möchte. In einer Aprilausgabe der renommierten Fachzeitschrift „Spektrum der Wissenschaften" wurde vor einigen Jahren beispielsweise von einer Aufsehen erregenden Entdeckung berichtet: Amerikanische Archäologen hatten auf einer Insel bei Neuguinea einen frühzeitlichen Computer entdeckt. Aus Seilen und Rollen hatten die Ureinwohner aus der Elektronik bekannte Schaltbausteine wie AND-Gatter, OR-Gatter und Inverter erstellt und zu einem echten Rechenwerk zusammengesetzt. Die Archäologen nahmen an, dass die damalige Priesterkaste diesen „Computer" als eine Art Orakel betrieb und ihr Wissen um die Konstruktion dieses Orakels zum Machterhalt nutzte. Schematische Abbildungen zur Funktionsweise des Rechenwerks und eine Einführung in die digitale Schaltungslogik rundeten den Artikel ab. Natürlich handelte es sich um einen Aprilscherz, aber immerhin: Unter dem Eindruck von so viel Logik und Wissenschaft blieb der gesunde Menschenverstand einiger Leser auf der Strecke. In den nachfolgenden Monaten ergingen daraufhin einige böse Briefe an die Redaktion von aufgebrachten Lesern, die die sensationelle Nachricht sofort weiterverbreitet und sich dabei bei ihren Professoren und Kollegen blamiert hatten.

Nicht ganz so weit zurückliegend, dafür aber verbrieft, ist die Erfindung des Lochkartensystems durch den Deutsch-Amerikaner Hermann Hollerith. Um das Jahr 1890 entwickelte er ein Verfahren, bei dem Daten durch Lochung bestimmter Felder auf vorgefertigten Karten (eben den Lochkarten) kodiert und festgehalten wurden. Mit Hilfe spezieller Maschinen, den sogenannten Hollerith- oder Lochkartenmaschinen, konnte man diese Daten automatisch auswerten, beispielsweise zur Erstellung von Serienbriefen, zur statistischen Datenerfassung oder allgemein zur Auswertung großer Datenmengen.

Der erste offiziell anerkannte, noch mechanische Computer war der 1936 gebaute Z1 des Berliners Konrad Zuse. Kurz darauf folgten Röhren-, später Transistoren- und schließlich Chip-Rechner. In der Zwischenzeit hatte sich auch bei der Softwareentwicklung Einiges getan: Anstatt Lochkarten zu stanzen, gab man Maschinenbefehlprogramme über ein Terminal ein. Irgendwann wurden die Maschinenbefehle durch die Sprache Assembler ersetzt und schließlich kamen die ersten höheren Programmiersprachen, die interpretiert oder kompiliert wurden.

Interpreter und Compiler

Maschinenbefehle sind „Wörter", die aus einer Folge von Nullen und Einsen bestehen, also beispielsweise 0011000010101011. Das eigentliche Rechenwerk eines Computers, der *Prozessor*, versteht nur diese binären Befehle (wobei noch zu beachten ist, dass jeder Prozessortyp seinen eigenen spezifischen Sprachschatz hat). Da das Programmieren mit diesen Befehlen für Menschen viel zu mühsam und schwierig ist, kam man auf die Idee, die Programmquelltexte in einer anderen Sprache aufzusetzen und dann mit Hilfe eines passenden Übersetzerprogramms in Maschinenbefehle umschreiben zu lassen. Die üblichen menschlichen Sprachen sind aber viel zu komplex und uneindeutig, um sie maschinell übersetzen zu können. Aus diesem Grunde wurden eigene Programmiersprachen wie C oder Basic entwickelt, mit einfacher Grammatik und geringem Wortschatz, die für Menschen leichter zu erlernen und für Übersetzerprogramme leichter in Maschinenbefehle umzusetzen sind.

Grundsätzlich gibt es zwei Kategorien von Übersetzerprogrammen: die Interpreter und die Compiler.

Ein Interpreter lädt den Quelltext des Programms, übersetzt ihn stückweise und lässt die übersetzten Anweisungen direkt vom Prozessor ausführen. Endet das Programm, endet auch die Ausführung des Interpreters.

Ein Compiler lädt den Quelltext des Programms, übersetzt ihn komplett (wobei er auch kleinere Optimierungen vornehmen kann) und speichert das kompilierte Programm in einer neuen Datei (unter Windows eine *.exe*-Datei) auf der Festplatte. Wenn der Anwender die *.exe*-Datei danach aufruft, wird das Programm direkt vom Betriebssystem geladen und ausgeführt.

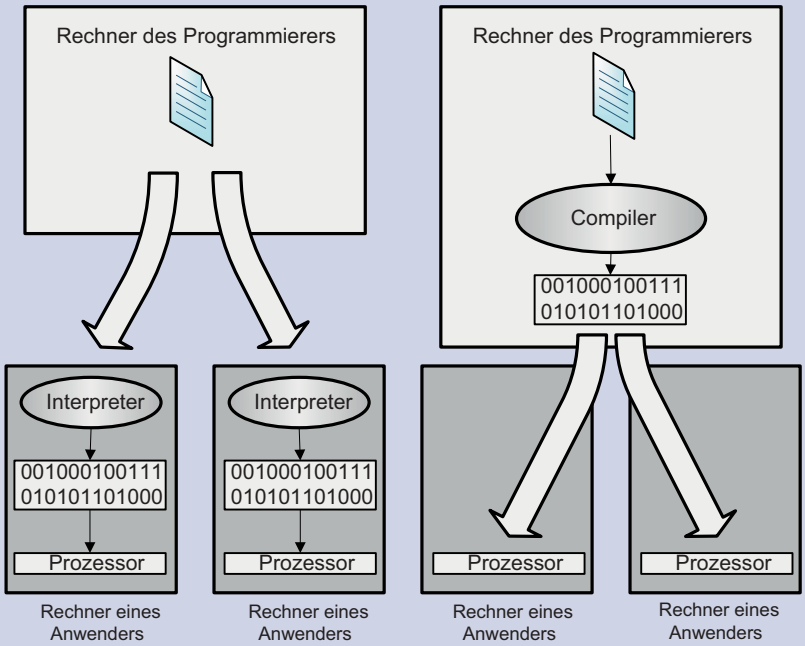

Bild 1.1 Schematische Darstellung der Arbeit eines Interpreters und eines Compilers. (Beachten Sie, dass bei der Kompilierung das fertige Programm nur auf Rechnern ausgeführt werden kann, deren Architektur und Betriebssystemfamilie zu dem Rechner des Programmierers kompatibel sind.)

Ein interpretiertes Programm kann auf jedem Rechner ausgeführt werden, auf dem ein passender Interpreter verfügbar ist. Die Ausführung ist allerdings langsamer und der Quelltext des Programms ist für jeden einsehbar.[2]

Ein kompiliertes Programm kann nur auf Rechnern ausgeführt werden, deren Plattform (Prozessor/Betriebssystem-Kombination) die Maschinenbefehle versteht, die der Compiler erzeugt hat. Dafür kann der Code für die Plattform optimiert werden, ist aufgrund der Binärkodierung vor geistigem Diebstahl weitgehend geschützt und wird schneller ausgeführt, da kein Interpreter zwischengeschaltet werden muss.

1.1.2 Die strukturierte Programmierung

Anfangs erlaubten die Programmiersprachen nur Programme, in denen die einzelnen Anweisungen, die der Computer bei Ausführung des Programms abarbeiten sollte, von oben nach unten im Programmquelltext angegeben werden mussten.

[2] Moderne „Interpreter“, wie sie beispielsweise für die Ausführung von Java-, C#- oder .NET-C++-Programmen verwendet werden, interpretieren daher vorkompilierte Programme und nennen sich JIT-Compiler (= „Just in time“-Compiler), weil sie die Programmausführung kaum verzögern.

1. Tue dies
2. Tue das
3. Mache jetzt jenes
4. …

Programmieranfänger dürfte dieses Prinzip begeistern, denn es ist leicht zu verstehen. Doch in der Praxis stellte sich damals bald heraus, dass diese Art der Programmierung ihre Grenzen hat: Einmal implementierte Teillösungen lassen sich schlecht wiederverwerten, größere Programme sind mangels Strukturierung sehr unübersichtlich. Abhilfe brachte hier die Entwicklung strukturierter Programmiersprachen wie Algol, Pascal oder C. Diese unterstützten

- die Steuerung des Programmablaufs durch Verzweigungen oder Schleifen und
- die Modularisierung des Codes durch Funktionen, d. h., für häufig benötigte Teilprobleme wie zum Beispiel die Ausgabe eines Textes konnte man eine eigene Funktion definieren, die genau diese Aufgabe übernahm. Im Rest des Programms brauchte man dann zur Ausgabe des Textes nur noch die entsprechende Funktion aufzurufen (anstatt immer wieder den Block mit den Anweisungen zur Textausgabe zu kopieren).

C, das Anfang der 1970er-Jahre speziell für die Implementierung des UNIX-Betriebssystems entwickelt wurde, nahm unter diesen Programmiersprachen schnell eine Sonderstellung ein, denn

- C war strukturiert,
- C war für eine höhere Programmiersprache sehr maschinennah und damit auch sehr leistungsfähig,
- C-Programme waren klein und schnell.

Die beiden letzten Punkte waren für den Erfolg von C entscheidend. Bedenken Sie, dass wir uns in den 1970ern und 1980ern befinden. Die ersten PCs kamen auf den Markt, 1976 der erste Apple-Computer, 1981 der erste IBM-PC. 1986 stellte IBM den AT vor, in der Standardausführung mit 265 KByte Arbeitsspeicher, 20 MByte Festplatte und einer Taktung von 6 MHz. Die Dimensionen dürften deutlich machen, worauf es zu diesen Zeiten bei der Programmierung ankam: Die Programme mussten klein sein (damit sie auf die Festplatte und in den Arbeitsspeicher passten) und sie mussten schnell sein, damit man trotz der (aus heutiger Sicht) langsamen Prozessoren vernünftig mit ihnen arbeiten konnte. Mit C konnte man Programme schreiben, die diese Anforderungen bestens erfüllten.

C-Programme waren schnell, weil sie kompiliert wurden und weil die Maschinennähe der Sprache es dem Compiler zudem leicht machte, den Quelltext in optimalen Maschinencode zu übersetzen.

Und C-Programme waren klein, weil die Sprache C auf Symbole und Operatoren setzte statt auf gut lesbare Befehle. Das Grundgerüst einer if-else-Verzweigung (die anhand einer Bedingung entscheidet, welcher von zwei Anweisungsblöcken auszuführen ist) umfasste in Pascal beispielsweise 29 Zeichen:

```
if Bedingung then begin
     Anweisungen
     end
else begin
```

```
     Anweisungen
end;
```

während C mit gerade einmal zwölf Zeichen auskam:

```
if Bedingung {
     Anweisungen
} else {
     Anweisungen
}
```

Diese sehr kryptische, aber eben auch prägnante und Speicherplatz sparende Syntax hat C++ von C übernommen.

1.1.3 Chips sind billig, Programmierer teuer

Mit der rasanten Entwicklung im Hardwarebereich und dem Boom in der Softwarebranche verschoben sich in den 1990er-Jahren zunehmend die Ausgangsbedingungen. Die Rechner wurden immer schneller, die Speichermedien immer dichter – für Standardsoftware lohnte es sich nicht mehr, tage-, ja wochenlang mit der Optimierung eines Programms zu verbringen, nur damit dieses noch einen Tick schneller lief oder einige KByte weniger Speicher belegte. Dafür wurden die Programme immer umfangreicher und damit auch immer unübersichtlicher und schwerer zu warten. In der Softwarebranche begann man umzudenken. Designziele wie Schnelligkeit und Schlankheit der ausgelieferten Software traten hinter den Forderungen nach benutzerfreundlicher, leistungsfähiger Software und kostengünstiger Entwicklung und Wartung zurück. Allein mit strukturierter Programmierung war dies nicht mehr zu leisten, zu hinderlich war vor allem die unnatürliche Trennung von Daten und Funktionen. Diese Trennung wurde im objektorientierten Programmiermodell aufgehoben. Das objektorientierte Programmiermodell führte eine neue Form der Repräsentation von Daten und, darauf aufbauend, neue Techniken zur Strukturierung und Wiederverwertung von Code ein, ohne die moderne Softwareentwicklung kaum noch denkbar ist. Obwohl es schon sehr früh Programmiersprachen gab, die rein objektorientiert waren, begann der Siegeszug der objektorientierten Programmierung erst Mitte der 1980er-Jahre als Bjarne Stroustrup die Sprache C um objektorientierte Konzepte erweiterte und damit C++ aus der Taufe hob. C++ ist eine Übermenge von C, d. h., man kann mit einem C++-Compiler nicht nur in C++ programmierte, objektorientierte Programme kompilieren, sondern auch ältere C-Programme. Dies war und ist sehr wichtig, denn schließlich gibt es große Mengen an bestehendem C-Code, der weiterhin gepflegt werden will.

Paradigmen moderner Programmiersprachen

Für den Prozessor besteht ein Programm aus einer Abfolge von Maschinenbefehlen. Heißt dies aber, dass auch die Quelltexte der verschiedenen Programmiersprachen aus einer Abfolge von Maschinenbefehlen bestehen müssen? Nein! Die höheren Programmiersprachen organisieren daher ihre Quelltexte in Konstrukten, die sich stärker am menschlichen Denken orientieren, und benutzen den

Compiler oder Interpreter dazu, diese Konstrukte in Maschinenbefehle für den Prozessor umzuwandeln.

Die verschiedenen Modelle zur Strukturierung des Quelltextes bezeichnet man auch als *Paradigmen*. Das klassische Paradigma der imperativen Programmierung organisiert den Quelltext z. B. in Variablen und Funktionen. Die Variablen dienen zum Speichern von Daten und mit den Funktionen können die Daten manipuliert und bearbeitet werden. Diesem Paradigma gehören Sprachen wie C oder Pascal an. Die fertigen Programme sind sehr schlank und meist recht schnell in der Ausführung. Bei größeren Projekten ist es aber meist schwierig, die Übersicht zu behalten, welche Daten von welchen Funktionen in welcher Reihenfolge bearbeitet werden dürfen oder müssen.

An diesem Punkt setzt das Paradigma der objektorientierten Programmierung an. Es fasst zusammengehörende Daten und ihre verarbeitenden Funktionen zur höheren Organisationsform eines Objekts zusammen und beschreibt gleichartige Objekte in einer Klassendefinition.[3] (Ein OOP-Programm zur Berechnung von Rechtecken würde z. B. die Variablen `laenge` und `breite` sowie die Funktionen `BerechneUmfang()`, `BerechneInhalt()` zu einer Klasse `Rechteck` zusammenfassen und dann für jedes zu berechnende Rechteck ein Objekt der Klasse `Rechteck` erzeugen.)

Obwohl es beim Schreiben einer Klassendefinition mehr zu beachten gilt als beim Aufsetzen von ein paar Funktionen, lohnt sich der Mehraufwand, denn die anschließende Arbeit mit den Objekten ist viel einfacher und sicherer als die Arbeit mit den Funktionen. Aus diesem Grunde ist das objektorientierte Paradigma für größere Projekte weitaus besser geeignet als die imperative Programmierung. Dass objektorientierte Programme etwas umfangreicher und minimal langsamer als imperative Programme sind, spielt bei der Leistungsfähigkeit der heutigen Rechner meist keine Rolle mehr.

1.1.4 Fassen wir zusammen

Sie haben nun bereits einiges über Herkunft und Konzeption der Sprache C++ erfahren. Wenn Sie bisher noch nie programmiert haben, werden Sie vielleicht erschrocken sein, weil Sie im Zuge der Ausführungen mit einer Vielzahl von Begriffen konfrontiert wurden, die Sie nur schwer einordnen oder verstehen konnten (Maschinencode, Compiler, Funktionen, Klassen, Objekte, Programmierparadigmen etc.). Keine Sorge – bald werden Ihnen diese Begriffe so geläufig sein wie die ersten Zeilen aus Schillers Ballade „Die Bürgschaft“[4]. Im Moment sollten Sie nur Folgendes verstanden haben:

[3] Einige objektorientierte Sprachen erlauben auch die direkte Definition von Objekten oder benutzen andere Begriffe, um die verschiedenen OOP-Konzepte zu bezeichnen.

[4] Sollten Sie die „Bürgschaft“ noch nicht gelesen haben, holen Sie dies unbedingt nach. Programmieren ist ein Spaß, humanistische Bildung eine Muss.

Merksatz

C++ setzt auf C auf und ist letzten Endes so etwas wie eine Überarbeitung und Erweiterung von C um objektorientierte Konzepte.[5]

Dies führte dazu, dass

- C++ eine Hybridsprache ist, in der man sowohl rein strukturiert (wie in C) als auch objektorientiert programmieren kann (und vermischen lassen sich beide Programmierstile natürlich auch).
- C++ die spartanische Syntax von C übernommen hat.
- C++ für etliche Programmieraufgaben (etwa die Arbeit mit Texten – oder „Strings“, wie es im Programmierjargon heißt) doppelte Lösungen anbietet: einmal die von C geerbte Lösung und dann noch die eigene, objektorientierte Lösung.
- C++ ein wahres Ungetüm von Programmiersprache ist: enorm leistungsfähig und vielseitig, aber gerade deshalb auch sehr unübersichtlich. Eine Sprache, die für jeden Programmierer zugleich Segen wie Fluch sein kann.

1.2 Von der Idee zum fertigen Programm

Die Entwicklung von Programmen läuft unabhängig von der verwendeten Sprache üblicherweise nach dem folgenden Muster ab:

1. Man hat ein Problem, eine Idee, eine Aufgabe, zu deren Lösung man einen Computer einsetzen möchte.

   So könnten Sie zum Beispiel daran interessiert sein, aus einem Betrag den Mehrwertsteueranteil herauszurechnen.

2. Als Nächstes wird die Aufgabe als Algorithmus, also als eine Folge von Befehlen in natürlicher Sprache formuliert. Größere Probleme werden dabei in Teilaufgaben und Teilaspekte aufgeteilt. (Ob der Algorithmus tatsächlich auf dem Papier oder nur im Kopf des Programmierers entwickelt wird, hängt von der Komplexität der Aufgabe und der Genialität des Programmierers ab.)

   Unser Algorithmus ist recht einfach und besteht aus vier Teilaufgaben:

   - *den Gesamtbetrag vom Benutzer abfragen,*
   - *den Gesamtbetrag in das Programm einlesen,*
   - *den Anteil der Mehrwertsteuer berechnen,*
   - *das Ergebnis ausgeben.*

[5] Tatsächlich ist jeder gute C++-Compiler auch in der Lage, (nahezu) beliebigen C-Code zu übersetzen. (Es gibt nur ganz wenige Punkte, in denen C++ die von C übernommene Syntax und Semantik verändert hat.) Umgekehrt gab es in der Übergangszeit – und gibt es wohl auch heute noch – nicht wenige Programmierer, die in C++ mehr oder weniger guten, alten C-Code schrieben.

3. Der Algorithmus wird vom Programmierer in die Anweisungen einer Programmiersprache umgesetzt. Dies ergibt den sogenannten Quelltext oder Quellcode.

   Der Quelltext unseres kleinen Beispielprogramms könnte etwa wie folgt aussehen:[6]

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

int main()
{
   double preis;
   double mwst;

   cout << endl;

   // 1. Eingabe von Benutzer anfordern
   cout << " Geben Sie den Preis ein: ";

   // 2. Eingabe einlesen
   cin >> preis;

   // 3. Mehrwertsteueranteil berechnen
   mwst = 0.19 * preis / 1.19;

   // 4. Berechneten Mehrwertsteueranteil ausgeben
   cout << " In dem Preis sind " << mwst
        << " Euro Mehrwertsteuer enthalten." << endl;

   cout << endl;
   return 0;
}
```

4. Dieser Quelltext muss dann durch ein spezielles Programm, den *Compiler*, in Maschinenanweisungen (Binärcode) übersetzt werden, die das eigentliche Herz des Computers – der Prozessor – versteht und ausführen kann. Das Ergebnis ist eine ausführbare Datei (eben ein Programm), die unter Microsoft Windows standardmäßig die Dateierweiterung *.exe* trägt.
5. Das ausführbare Programm kann auf jedem Rechner, dessen Prozessor den erzeugten Maschinencode versteht, gestartet werden.

Was genau ist ein Programm?

Die noch in Deutsch formulierten Befehle? Die in C++ formulierten Befehle? Oder die binär kodierten Maschinenanweisungen? Im weitesten Sinne können Sie in allen drei Fällen von Ihrem Programm reden. Wenn Sie es dagegen genau nehmen wollen, bezeichnen Sie die noch in Ihrer Muttersprache aufgesetzte Befehlsfolge als *Algorithmus*, die in C++ formulierte Version des Algorithmus als Quelltext Ihres Programms und erst den vom Compiler erzeugten Maschinencode als Ihr ausführbares Programm.

[6] Vermutlich wird Ihnen dieser Code vollkommen unverständlich erscheinen. Machen Sie sich darüber keine Gedanken. Zum einen ist der Code schon recht komplex, zum anderen geht es hier ja nicht darum, dass Sie das Beispiel nachstellen, sondern nur darum, dass Sie einmal einen C++-Quelltext gesehen haben.

1.3 Näher hingeschaut: der C++-Compiler

Ich muss noch einmal auf den C++-Compiler zurückkommen, der unsere C++-Quelltexte übersetzt. Nicht nur weil er ein gestrenger Lehrer ist, auf den man Novizen vorbereiten muss, sondern auch, weil mit seiner Arbeit ein wichtiges C++-Konzept verbunden ist: das der Deklaration.

Lesen Sie sich die folgenden Ausführungen entspannt durch. Sie müssen nicht auf Anhieb alles verstehen, aber Sie sollten sich unbedingt mit den angesprochenen Konzepten schon einmal grob vertraut machen. Wenn Sie dann in den weiteren Kapiteln mit den besagten Konzepten nochmals konfrontiert werden, bringen Sie bereits grundlegende Vorkenntnisse mit und wissen, dass Sie hier bei Bedarf noch einmal nachlesen können.

1.3.1 Der Compiler ist ein strenger Lehrer

Bevor der Compiler eine Quelltextdatei übersetzen kann, muss er diese analysieren. Dabei prüft er auch gleich, ob die Syntax korrekt ist und die einzelnen Bausteine richtig verwendet wurden. Bei der kleinsten Ungenauigkeit verweigert er die Arbeit und gibt eine Fehlermeldung aus – was ihn bei Anfängern nicht gerade beliebt macht.

Trotzdem ist der Compiler nicht Ihr Feind. Betrachten Sie ihn vielmehr als strengen, aber gerechten Kritiker, der nur das Beste für Ihr Programm will.

Wenn Sie beim Kompilieren Fehlermeldungen angezeigt bekommen, kümmern Sie sich zunächst nur um die erste Fehlermeldung. Nachdem Sie den zugehörigen Fehler korrigiert haben, kompilieren Sie den Quelltext neu. Nicht selten verschwinden dann auch nachfolgende Fehlermeldungen, bei denen es sich um Folgefehler handelte.

Oben wurde bereits erwähnt, dass der Compiler nicht nur prüft, ob die Syntax korrekt ist, sondern auch, ob die einzelnen Bausteine ihrer Bestimmung gemäß verwendet werden. Dies wirft die Frage auf, woher der Compiler denn weiß, wie die Bausteine korrekt verwendet werden?

Nun, in einem C++-Quelltext gibt es grundsätzlich drei Gruppen von Bausteinen:

- Schlüsselwörter wie `if`, `class` oder `for`,
- symbolische Zeichen wie `+`, `::` oder `//`,
- Elemente, die der Programmierer definiert (oder die aus einer Bibliothek stammen).

Die beiden ersten Gruppen sind fest in der Sprache verankert, sodass es logisch ist, dass ein C++-Compiler weiß, wie die Elemente dieser Gruppen zu verwenden sind. Der Vorrat an diesen Elementen ist allerdings so klein, dass man damit kein sinnvolles Programm schreiben kann. Der C++-Programmierer ist also darauf angewiesen, dass er neue Elemente einführt: beispielsweise Variablen, in denen er Daten abspeichern kann, oder Funktionen, mit denen er Daten bearbeiten kann, oder neue Datentypen, die beschreiben, welche Art von

Daten das Programm verarbeiten soll. Die folgenden Beispiele sollen Ihnen einen ersten Eindruck davon geben, wie diese Definitionen aussehen:

- Variablen – zum Zwischenspeichern von Daten:

```
int eineVar;
```

- Datentypen – zur Repräsentation neuer Arten von Daten, hier z. B. zweidimensionale Koordinatenangaben:

```
struct Koordinate
{
   int x;
   int y;
};
```

- Funktionen (oder Memberfunktionen) – zur Lösung bestimmter Aufgaben oder Teilprobleme, z. B. die Addition zweier Zahlen:

```
int Addieren(int wert1, int wert2)
{
    int summe;

    summe = wert1 + wert2;

    return summe;
}
```

1.3.2 Definition und Deklaration

In C++ muss jedes neu eingeführte Element definiert und deklariert werden. Die Definition ist eine vollständige Beschreibung des Elements, auf ihrer Basis wird das Element später erzeugt. Eine Deklaration ist ebenfalls eine Beschreibung, die allerdings nur die Informationen enthält, die der Compiler benötigt, um die korrekte Verwendung des Elements sicherstellen zu können.

Der folgende C++-Code definiert z. B. eine Funktion, die zwei ganze Zahlen addiert und die Summe als Ergebnis zurückliefert.

```
int Addieren(int wert1, int wert2)
{
    int summe;

    summe = wert1 + wert2;

    return summe;
}
```

Die Funktion hat den Namen `Addieren` und übernimmt als Parameter zwei Werte vom Typ `int` (dem C++-Datentyp für ganze Zahlen). Als Ergebnis liefert die Funktion einen `int`-Wert (den Inhalt der Variablen `summe`) zurück. Der Code zwischen den geschweiften Klammern {

und } gibt an, welche Operationen ausgeführt werden sollen, wenn die Funktion aufgerufen wird. Ein solcher Aufruf sähe dann beispielsweise so aus:

```
int ergebnis;
ergebnis = Addieren(1, 7);   // liefert 8 zurück
```

Die reine Deklaration der Funktion `Addieren` ist etwas kürzer als die Definition:

```
int Addieren(int, int);
```

Sie gibt an, dass die Funktion `Addieren` heißt, zwei `int`-Werte als Parameter übernimmt und einen `int`-Wert zurückliefert. Diese Angaben genügen dem Compiler, um die gewünschte Verwendung sicherzustellen. So kann er z. B. verifizieren, dass der obige Aufruf korrekt ist, denn der Name der Funktion wurde richtig geschrieben, es wurden zwei ganze Zahlen als Parameter übergeben und das Ergebnis wurde in einer `int`-Variablen gespeichert. Was genau die Funktion macht (Inhalt zwischen geschweiften Klammern) oder wie die Parameter der Funktion heißen (hier `wert1` und `wert2`), ist zur Überprüfung der korrekten Verwendung unerheblich.

Zur Verwendung von benutzerdefinierten Elementen gibt es in C++ daher zwei wichtige Regeln:

Merksatz

Jedes Element, das in einer Quelltextdatei verwendet wird, muss dem Compiler zuvor per Deklaration bekannt gemacht werden (wobei eine Definition die Deklaration einschließt).

Merksatz

In einem C++-Programm darf kein Element mehr als einmal definiert werden. (Stellen Sie sich nur einmal vor, es gäbe in einem Programm zwei Funktionen namens `Addieren()`. Woher sollte der Compiler dann wissen, welche dieser Funktionen er ausführen lassen soll, wenn er auf den Namen `Addieren` trifft.)

Was bedeutet dies für Ihre Programmierarbeit?

Bei kleineren Programmen, die nur aus einer Quelltextdatei bestehen, müssen Sie lediglich darauf achten, dass Sie die Elemente zuerst definieren und dann verwenden.

Wenn Ihre Programme aus mehreren Quelltextdateien bestehen und Sie ein Element in mehreren Quelltextdateien verwenden möchten, müssen Sie das Element in einer Quelltextdatei definieren und in allen anderen, in denen es verwendet wird, deklarieren – dabei hilft Ihnen das Konzept der Headerdateien.

1.3.3 Das Konzept der Headerdateien

Bestimmte Elemente benötigt man bei der Programmierung immer wieder: beispielsweise Elemente zum Ausgeben von Daten auf den Bildschirm oder zum Berechnen von Sinus und Kosinus oder zum Öffnen von Dateien auf der Festplatte. Damit man diese Elemente nicht jedes Mal neu definieren muss, werden sie einmalig als Bibliothek kompiliert und dann bei Bedarf immer wieder verwendet.

Als C++-Programmierer steht Ihnen von vorneherein eine Bibliothek mit Grundfunktionen zur Verfügung, über die jeder C++-Compiler verfügt: die C++-Standardbibliothek. Manche Compiler stellen noch weitere Bibliotheken zur Verfügung, andere Bibliotheken kann man zukaufen und natürlich können Sie auch eigene Bibliotheken erstellen.

Wann immer Sie aber ein Element aus einer Bibliothek verwenden möchten, müssen Sie zuvor die Deklaration des Elements in Ihre Quelltextdatei einfügen, damit der Compiler das Element akzeptiert. Da dies schnell zu einer sehr mühseligen und lästigen Aufgabe werden kann, stellen fast alle C++-Bibliotheken sogenannte Headerdateien zur Verfügung. Dies sind C++-Quelltextdateien mit der Extension *.h* (oder *.hpp*), in denen einfach nur die Deklarationen zu den Elementen der Bibliothek stehen. Größere Bibliotheken bieten sogar mehrere Headerdateien an, wobei jede Headerdatei einem bestimmten Aufgabengebiet (z. B. Ein- und Ausgabe oder mathematische Funktionen oder Dateioperationen) gewidmet ist.

Wenn Sie ein Element einer solchen Bibliothek verwenden möchten, müssen Sie nur den Inhalt der Headerdatei in den Anfang Ihrer Quelltextdatei einkopieren. Erfreulicherweise gibt es hierfür sogar einen Befehl, den der Compiler ausführen kann – die sogenannte Präprozessor-Direktive `#include`:

```
#include <iostream>
#include "Statistik.h"
```

Steht der Dateiname wie in der ersten Zeile in spitzen Klammern, sucht der Compiler die angegebene Datei in seinem Include-Pfad (Einstellung des Compilers). Steht der Dateiname in Hochkommata (zweite Zeile) sucht der Compiler relativ zum aktuellen Verzeichnis.

Tabelle 1.1 Wichtige Headerdateien der C++-Standardbibliothek

Headerdatei	Funktionalität
`<iostream>`	Ein- und Ausgabe
`<string>`	Die Klasse `string` zur Programmierung mit Zeichenfolgen („Strings“ genannt)
`<cstdlib>`	Auswahl verschiedener nützlicher Funktionen
`<cmath>`	Mathematische Funktionen
`<ctime>`	Funktionen zur Verarbeitung von Zeit- und Datumsangaben

Warum heißt die Headerdatei *cmath* nicht einfach *math*? Früher war es sehr wichtig, dass jedes C-Programm auch ein gültiges C++-Programm war. Aus diesem Grund übernahm C++ die Headerdateien von C, die alle auf die Extension .h auslauteten – wie z. B. *math.h*. Gleichzeitig stellte man aber auch „modernere" Versionen dieser Headerdateien zur Verfügung, die ohne Extension geschrieben wurden (wie für die neuen C++-Header üblich) und mit „c" begannen (um anzuzeigen, dass es sich um C-Funktionen handelt), daher: *cmath*.

1.3.4 Namensräume

Ebenso wie das Konzept der Headerdateien hat auch das Konzept der Namensräume weniger mit der eigentlichen Programmierung als vielmehr mit der Organisation von Code zu tun, genauer gesagt mit der Vermeidung und Lösung von Namenskonflikten. Das Grundproblem ist schnell skizziert:

Ein Programmierer hat im Quelltext seines Programms ein Element, sagen wir eine Klasse, namens `X` definiert. Dann bindet er per `#include`-Direktive eine weitere Headerdatei ein, um neue Bibliothekselemente verfügbar zu machen, muss aber beim Kompilieren feststellen, dass die Bibliothek ebenfalls ein Element namens `X` definiert. Er könnte nun seine eigene Klasse umbenennen oder auf die Bibliothek verzichten oder eine Kopie der Headerdatei anlegen und aus dieser die `X`-Deklaration entfernen (sofern das Bibliothekselement `X` nicht gerade dasjenige Element ist, welches er benutzen möchte). Wirklich befriedigend ist jedoch keine dieser Lösungen. Deswegen wurde zur Lösung solcher Namenskonflikte, die im Übrigen auch auftreten können, wenn mehrere Programmierer gemeinsam an einem Programm arbeiten und zum Abschluss ihre Quelltextdateien zusammenführen, das Konzept der Namensräume eingeführt.

Namensräume werden mit dem Schlüsselwort `namespace` definiert:

```
namespace demospace
{
   class X
   {
      // ...
   };
}
```

Alle Elemente, die in einem solchen Namensraum definiert sind, können fortan nicht mehr allein über ihren Namen, sondern nur über den um den Namensraum erweiterten Namen, den sogenannten vollqualifizierten Namen, angesprochen werden:

```
demospace::X
```

Auf diese Weise kann der Compiler problemlos zwischen gleichnamigen Elementen aus verschiedenen Namensräumen unterscheiden. (Wobei alle Elemente, die auf Dateiebene, aber außerhalb jeglichen Namensraums definiert sind, dem sogenannten globalen Namensraum angehören.)

Das Konzept der Namensräume gleicht dem Konzept der Verzeichnisse. Innerhalb eines Verzeichnisses können Sie keine zwei Dateien mit dem gleichen Namen ablegen. Wenn Sie die Dateinamen beibehalten wollen, müssen Sie die Dateien auf unterschiedliche Verzeichnisse verteilen und beim Zugriff jeweils den Verzeichnisnamen angeben.

Der folgende Code demonstriert, wie Sie auf die Elemente der C++-Standardbibliothek zugreifen können, die im Namensraum `std` definiert sind. Den Namen der Elemente wird dabei mit Hilfe des `::`-Operators der Namensraumbezeichner vorangestellt:

```
#include <iostream>
#include <string>

int main()
{

   std::string gruss;

   gruss = "Hallo Welt!";
   std::cout << gruss << std::endl;

   return 0;
}
```

So nützlich die vollqualifizierten Namen zur Vermeidung von Namenskonflikten sind, so lästig sind sie, wenn es gar keine Namenskonflikte gibt. C++ erlaubt es daher in solchen Fällen, die gesamten Namen aus einem Namensraum zu importieren:

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

Die obige `using`-Deklaration importiert die Namen aus dem Namensraum `std`, sodass die zugehörigen Elemente im weiteren Quelltext über ihren einfachen Namen angesprochen werden können.

Achtung!

Wenn Sie in einem C++-Programm ausschließlich die alten C-Header einbinden (z. B. `#include <stdio.h>`), streichen Sie die `using`-Deklaration. Gleiches gilt, wenn Sie mit älteren C++-Compilern arbeiten, deren Standardbibliothek und Headerdateien ebenfalls noch keinen Namensraum `std` kennen.

1.3.5 Der Compiler bei der Arbeit

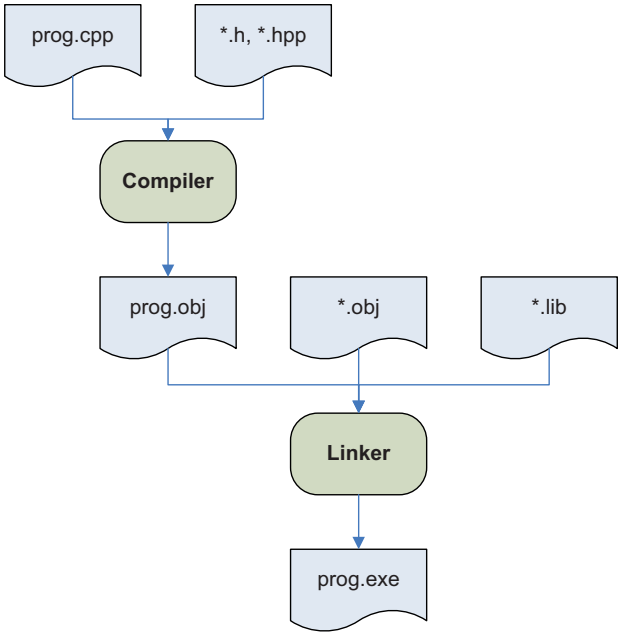

Bild 1.2 Erstellung eines C++-Programms

Wenn Sie den Compiler zur Übersetzung einer cpp-Quelltextdatei aufrufen, sucht der Compiler zuerst alle zugehörigen Headerdateien zusammen, denn die cpp-Quelltextdatei kann nur mit den Headerdateien zusammen übersetzt werden. Die Kombination aus cpp-Quelltextdatei und allen ihren Headerdateien bezeichnet man daher auch als *Übersetzungseinheit*.

Alles zusammen wird dann übersetzt. Das Ergebnis ist eine *Objektdatei*. Die Objektdatei enthält bereits binär kodierten Maschinencode, ist aber noch nicht ausführbar.

Wenn Sie den Compiler nicht nur angewiesen haben, die cpp-Quelltextdatei zu übersetzen, sondern aus der Quelltextdatei auch gleich ein ausführbares Programm zu erstellen, wird in einem zweiten Schritt die Objektdatei mit dem Code der benutzten Bibliotheken sowie etwaigen weiteren Objektdateien (falls der Quelltext des Programms auf mehrere Quelltextdateien verteilt wurde) zur ausführbaren exe-Datei, dem eigentlichen Programm, zusammengebunden. Für das Zusammenbinden der Objektdateien sorgt allerdings nicht der Compiler, sondern ein weiteres Hilfsprogramm: der sogenannte *Linker*. Meist wird dieser Linker allerdings automatisch nach der eigentlichen Übersetzung vom Compiler aufgerufen, sodass der Programmierer mit dem Linker relativ wenig zu tun hat (außer der Linker beschwert sich, dass es in dem Quelltext des Programms uneindeutige, weil doppelt oder mehrfach definierte Elemente gibt).

1.3.6 ISO und die Compiler-Wahl

Es gibt zahlreiche C++-Compiler und es liegt mir fern, Ihnen vorzuschreiben, mit welchem Compiler Sie arbeiten sollen. Wenn Sie mit einem Windows-Betriebssystem arbeiten, werden Sie vielleicht den C++-Compiler von Microsoft benutzen wollen, den Sie für das Selbststudium kostenlos aus dem Internet herunterladen können (siehe Anhang B). Linux-Anwender werden wohl hingegen den GNU-Compiler präferieren, der auf vielen Linux-Systemen vorinstalliert ist. Für die Ausführungen in diesem Buch und das Erlernen von C++ ist es dabei glücklicherweise ganz egal, welchen Compiler Sie verwenden. Und dies verdanken wir der International Organization for Standardization, kurz ISO.

Im Jahre 1998, dreizehn Jahre nach ihrer Einführung, wurde die Sprache C++ offiziell standardisiert (zunächst ANSI später auch ISO) und mit einer umfangreichen Standardbibliothek ausgestattet. Ziel dieser Standardisierung war und ist es, die Entwicklung von unterschiedlichen C++-Dialekten durch die Compiler-Bauer[7] zu verhindern. Seitdem gibt es keine guten und schlechten C++-Compiler mehr, sondern nur noch ISO-kompatible und nicht kompatible.

Erfreulich ist, dass heute das Gros der modernen C++-Compiler ausnahmslos ISO-kompatibel ist. Einige Compiler bieten zwar syntaktische Erweiterungen an, so z. B. der C++-Compiler von Microsoft, erlauben aber auch die Erstellung reinen ISO-C++-Codes.

Für den Programmierer hat die Beschränkung auf reines ISO-C++ einen gewichtigen Vorteil: Er ist nicht an einen speziellen Compiler gebunden, sondern kann seine Quelltexte jederzeit, auf jeder Plattform[8] mit jedem beliebigen ISO-kompatiblen Compiler kompilieren und erstellen. Oder übertragen auf dieses Buch: Den ISO-C++-Code aus diesem Buch können Sie mit jedem beliebigen ISO-kompatiblen Compiler nachvollziehen.

Hinweise zu Installation und Bedienung der Compiler finden Sie in Kapitel 2 und im Anhang.

1.3.7 Der neue C++17-Standard

Nachdem der C++-Standard viele Jahre nahezu unverändert geblieben ist, wurde erstmals wieder im Jahr 2011 ein neuer Standard ratifiziert. Seitdem wird der Standard kontinuierlich überarbeitet und weiterentwickelt – bis zum aktuellen Standard C++17 und natürlich stets unter Wahrung größtmöglicher Abwärtskompatibilität, denn schließlich hat niemand ein Interesse daran, massenweise bestehenden C++-Code durch einen neuen Standard ungültig zu machen.

---

[7] Insbesondere Microsoft tendierte früher dazu, die verschiedenen Programmiersprachen, für die das Softwarehaus Compiler anbietet, um neue und abweichende Syntaxformen zu erweitern, die dann ausschließlich von den passenden Microsoft-Compilern verstanden und unterstützt werden. Angeblich dient dies alles dazu, dem Programmierer die Arbeit zu erleichtern, faktisch bindet es den Programmierer aber vor allem an einen einzelnen Compiler.

[8] Kombination aus Prozessor und Betriebssystem.

So gibt es nur ganz wenige Syntaxformen und Elemente, die als „deprecated“ eingestuft wurden (d. h., man soll sie nicht mehr verwenden), aber zahlreiche kleinere und größere Ergänzungen und Erweiterungen.

Soweit diese Änderungen Themen betreffen, die in diesem Buch behandelt werden, wurden sie natürlich berücksichtigt. Da es sich in den meisten Fällen um alternative Syntaxformen handelt, werden sie auch als eben solche beschrieben – als Alternativen. So können Sie sich einerseits schon einmal mit den neuen Techniken vertraut machen und lernen andererseits noch die traditionellen Syntaxformen.

Für eine vollständige Übersicht der im Buch beschriebenen oder erwähnten Neuerungen siehe im Index die Einträge *C++11*, *C++14* und *C++17*)

C++1z

Wenn Sie im Internet stöbern, werden Sie möglicherweise auch auf Quellen stoßen, die vom C++1z-Standard sprechen. C++1z war der provisorische Arbeitstitel des neuen C++17-Standards.

1.4 Übungen

1. Falls Sie es noch nicht getan haben, installieren Sie jetzt auf Ihrem Rechner die Community Edition von Visual Studio oder einen anderen Compiler, siehe Anhang B.

2 Grundkurs: Das erste Programm

Nun ist es endlich so weit! Wir werden unser erstes C++-Programm erstellen.

2.1 Hallo Welt! – das Programmgerüst

Es gibt eine Reihe von typischen Programmelementen, die man in so gut wie jedem C++-Programm wiederfindet. Diese Elemente werden wir uns jetzt einmal näher anschauen.

Bevor ich Ihnen die Programmelemente im Einzelnen vorstelle, sollten wir jedoch einen Blick auf den vollständigen Quelltext des Programms werfen. Wenn Sie bereits über etwas Programmiererfahrung verfügen, werden Sie das Programm womöglich sogar wiedererkennen: Es ist eine Adaption des klassischen „Hello World"-Programms aus der C-Bibel von Kernighan und Ritchie.

Listing 2.1 Das erste Programm (aus HalloWelt.cpp)

```
/*****************************************************
 * Hallo Welt-Programm
 *
 * gibt einen Gruss auf den Bildschirm aus
 */

#include <iostream>
using namespace std;

int main()
{
   cout << "Hallo Welt!" << endl;

   return 0;
}
```

Packen Sie jetzt bitte Ihr Sezierbesteck aus und schärfen Sie Ihren Verstand. Wir beginnen mit der Analyse.

Achtung!

C++ unterscheidet streng zwischen Groß- und Kleinschreibung. Beachten Sie dies, wenn Sie in Abschnitt 2.2 den Quelltext in Ihren Editor eingeben.

2.1.1 Typischer Programmaufbau

Die landläufige Vorstellung von einem Programm ist gemeinhin eine Folge von Anweisungen, die vom Rechner nacheinander ausgeführt werden. Blickt man aber in die Quelltextdatei eines beliebigen C++-Programms, offenbart sich ein ganz anderes Bild.

Tatsächlich bestehen C++-Programme aus:

- Kommentaren
- Präprozessordirektiven
- Namensräumen
- Deklarationen und Definitionen
- einer Eintrittsfunktion namens `main()`

Natürlich gibt es auch Anweisungen, doch existieren diese nur als untergeordnete Elemente in den Definitionen der Funktionen!

Listing 2.2 Die typischen Elemente eines C++-Programms

```
/*******************************************************
 * Hallo Welt-Programm           // mehrzeiliger Kommentar
 *
 * gibt einen Gruss auf den Bildschirm aus
 */

#include <iostream>              // Präprozessor-Direktive
using namespace std;             // Namensraum-Einbindung

int main()                       // Definition der Eintrittsfunktion
{
   cout << "Hallo Welt!" << endl;

   return 0;
}
```

Das Verhältnis aus Anweisungen (in Listing 2.1 fett hervorgehoben) und Elementen, die vornehmlich der Organisation des Quelltextes dienen (Präprozessordirektiven, Definitionen etc.), ist nicht immer so drastisch wie in diesem HalloWelt-Programm. Doch eines können und sollten Sie aus diesem Beispiel bereits ablesen: Programmierung hat auch viel mit Codeorganisation zu tun!

Merksatz

Bei der C++-Programmierung – wie im Übrigen bei der Programmierung mit jeder modernen Programmiersprache – genügt es nicht, sich zu überlegen, welche Anweisungen in welcher Reihenfolge zur effizienten Erledigung einer Aufgabe benötigt werden (Algorithmus). Sie müssen sich auch Gedanken darüber machen, wie Sie Ihren Quelltext organisieren.

Fürs Erste werden wir die Codeorganisation so einfach wie möglich halten. Konkret bedeutet dies, dass wir während unserer ersten Gehversuche mit C++ einfach unsere gesamten Anweisungen in die `main()`-Funktion schreiben werden.

Wo aber kommt diese `main()`-Funktion her? Und welche Bedeutung haben die anderen Elemente des Grundgerüsts aus Listing 2.1?

2.1.2 Die Eintrittsfunktion main()

Wenn Sie ein C++-Programm aufrufen, wird der Code des Programms in den Arbeitsspeicher geladen und vom Prozessor ausgeführt. Doch mit welchem Code beginnt die Ausführung des Programms?

Per Konvention beginnen C++-Programme immer mit einer Funktion namens `main()`. Wenn Sie einen Quelltext zu einer *.exe*-Datei kompilieren lassen, generiert der Compiler automatisch Startcode, der dafür sorgt, dass die Programmausführung mit der ersten Anweisung in `main()` beginnt.

Ihre Aufgabe ist es daher, in Ihrem Programmquelltext eine passende `main()`-Eintrittsfunktion zu definieren:

```
int main()
{
   // hier können Sie eigenen Code einfügen

   return 0;
}
```

Was diese Definition im Einzelnen zu bedeuten hat, werden Sie erst in Kapitel 6 erfahren, wenn wir uns intensiver mit der Definition von Funktionen beschäftigen. Bis dahin ist nur eines wichtig: Sie dürfen den Definitionscode nicht verändern, da die Funktion sonst nicht mehr vom Compiler als Eintrittsfunktion erkannt wird.

Wenn Sie also das `int` vergessen oder den `return`-Befehl falsch schreiben oder versuchen, die Funktion von `main()` in `start()` umzutaufen, so werden Sie dafür bei der Programmerstellung entsprechende Fehlermeldungen ernten. Und achten Sie auch auf die Groß- und Kleinschreibung. Für C++ sind `main` und `Main` nicht zwei Schreibweisen eines Namens, sondern ganz klar zwei verschiedene Namen!

Merksatz

C++ unterscheidet strikt zwischen Groß- und Kleinschreibung!

Ich sollte allerdings noch erwähnen, dass es eine zweite Variante für die Definition der Eintrittsfunktion `main()` gibt:

```
int main(int argc, char *argv[])
{
   // hier können Sie eigenen Code einfügen

   return 0;
}
```

Ja, manche Compiler erlauben sogar noch weitere Varianten. Grundsätzlich sollten Sie sich aber auf die beiden obigen Varianten beschränken, da nur so sichergestellt ist, dass sich Ihr Programm mit jedem ANSI-kompatiblen Compiler übersetzen lässt.

Wenn Sie den Compiler anweisen, aus einem Quelltext, der keine korrekt definierte `main()`-Eintrittsfunktion enthält, ein *.exe*-Programm zu erzeugen, werden Sie am Ende des Erstellungsprozesses vom Linker eine Fehlermeldung erhalten, dass die im Startcode referenzierte `main()`-Funktion nicht gefunden werden konnte.

2.1.3 Die Anweisungen

Innerhalb der geschweiften Klammern unserer `main()`-Funktion können wir nun endlich die Anweisungen aufsetzen, die bei Start des Programms ausgeführt werden sollen. Im Falle unseres ersten Beispielprogramms bescheiden wir uns mit einer einzigen Zeile, die den Text „Hallo Welt!“ ausgeben soll.

```
int main()
{
   cout << "Hallo Welt!" << endl;

   return 0;
}
```

Was bewirkt die obige Anweisung? Zunächst muss man wissen, dass `cout` ein vordefiniertes Objekt ist, welches die Konsole repräsentiert.

Die Konsole ist ein spezielles Programm des Betriebssystems (siehe Kasten), über das der Anwender Befehle ans Betriebssystem schicken kann. Für uns als Programmierer ist sie interessant, weil wir sie zum Datenaustausch zwischen unseren Programmen und den Anwendern nutzen können. Wir ersparen uns also den Aufbau einer eigenen Benutzeroberfläche und können uns ganz auf den reinen C++-Code konzentrieren.

Die Konsole

Die meisten PC-Benutzer, vor allem Windows- oder KDE-Anwender, sind daran gewöhnt, dass die Programme als Fenster auf dem Bildschirm erscheinen. Dies erfordert aber, dass das Programm mit dem Fenstermanager des Betriebssystems kommuniziert und spezielle Optionen und Funktionen des Betriebssystems nutzt. Programme, die ohne fensterbasierte, grafische Benutzeroberfläche (GUI = graphical user interface) auskommen, können hierauf jedoch verzichten und stattdessen die Konsole zum Datenaustausch mit dem Benutzer verwenden.

Die Konsole ist eine spezielle Umgebung, die dem Programm vorgaukelt, es lebe in der guten alten Zeit, als es noch keine Window-Systeme gab und immer nur ein Programm zurzeit ausgeführt werden konnte. Dieses Programm konnte dann uneingeschränkt über alle Ressourcen des Rechners verfügen – beispielsweise die Tastatur, das wichtigste Eingabegerät, oder auch den Bildschirm, das wichtigste Ausgabegerät. Der Bildschirm war in der Regel in den Textmodus geschaltet, wurde also nicht aus Pixelreihen, sondern aus Textzeilen aufgebaut.

Unter Windows heißt die Konsole MS-DOS-Eingabeaufforderung oder auch nur Eingabeaufforderung und kann je nach Betriebssystem über **Start/Programme** oder **Start/Programme/Zubehör** aufgerufen werden.

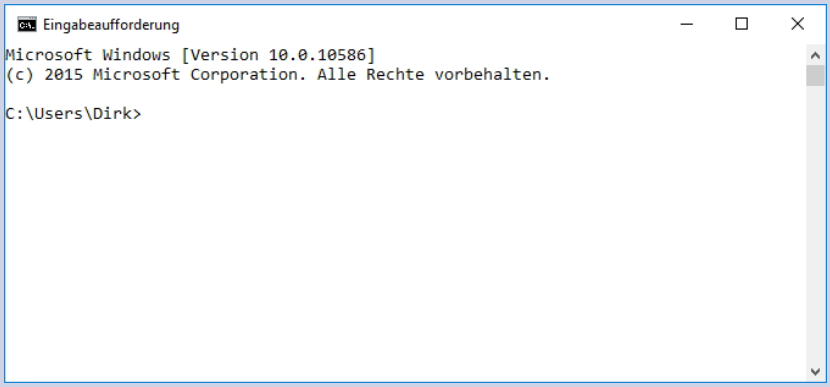

Bild 2.1 Die Konsole von Windows 8 (mit invertiertem Hintergrund)

Damit wir innerhalb eines Programms auf die Konsole zugreifen können, muss es im Programmcode aber ein Element geben, welches die Konsole repräsentiert. Dieses Element ist wie gesagt `cout`.

Allerdings handelt es sich bei `cout` nicht um ein Element, das fest in die Sprache integriert es. Vielmehr verbirgt sich hinter `cout` ein Objekt, das im Code der C++-Standardbibliothek definiert ist. Gleiches gilt im Übrigen auch für den Operator `<<`, der Ausgaben an `cout` schickt, sowie `endl`, das in der Ausgabe einen Zeilenumbruch (**end**-of-**l**ine = Zeilenende) erzeugt.

Unsere Anweisung schickt also zuerst mit Hilfe des `<<`-Operators den Text `Hallo Welt!` und dann noch einen Zeilenumbruch (`endl`) zur Konsole (`cout`):

```
cout << "Hallo Welt!" << endl;
```

Zwei Punkte an diesem Code bedürfen noch einer besonderen Erwähnung:

Merksatz

Texte, die ein Programm verarbeitet, werden auch als *Strings* bezeichnet und stehen in Anführungszeichen `""`, damit der Compiler sie vom Programmcode unterscheiden kann.

Merksatz

Anweisungen dürfen in C++ nur innerhalb von Funktionen stehen und müssen mit einem Semikolon abgeschlossen werden.

2.1.4 Headerdateien

Erinnern Sie sich, was in Kapitel 1 über die Verwendung von Elementen gesagt wurde, die nicht in die Sprache integriert sind: Sie müssen im Programmcode einmal definiert und in jeder Quelltextdatei, in der sie verwendet werden, deklariert werden.

Wie sieht es also mit der Definition aus? Die Definition von `cout`, `<<` und `endl` findet sich im Code der C++-Standardbibliothek. Dieser Code wird bei der Programmerstellung vom Linker automatisch mit Ihrem Code zur ausführbaren *.exe*-Datei verbunden. Wir müssen uns um diesen Teil nicht weiter kümmern. (Außer der Compiler wäre nicht korrekt konfiguriert und findet die Bibliotheksdateien nicht. Diese stehen übrigens meist in einem Verzeichnis *lib* und der Pfad zu diesem Verzeichnis kann über die Compiler-Optionen eingestellt werden.)

Bleibt noch die Deklaration. Da wir `cout`, `<<` und `endl` in unserer Quelltextdatei *HalloWelt.cpp* verwenden, müssen wir die Elemente dem Compiler auch in dieser Datei bekannt machen. Wir könnten dazu so vorgehen, dass wir in der Fachliteratur, der Bibliotheksdokumentation oder – soweit vorhanden – gar direkt im Quelltext der Bibliothek nachschlagen, wie die betreffenden Bibliothekselemente definiert sind, und uns daraus die Deklarationen ableiten, die wir dann über `main()` in den Quelltext einfügen.

Sie werden mir allerdings sicher zustimmen, dass diese Verfahrensweise recht mühselig, kompliziert und fehleranfällig wäre. Die C++-Standardbibliothek stellt daher für jeden Themenbereich, den die Bibliothek abdeckt, eine passende Headerdatei zur Verfügung, in der die benötigten Deklarationen schon gesammelt sind. Die Headerdatei für alle Bibliothekselemente, die mit der Ein- und Ausgabe zu tun haben, heißt *iostream* und kann mit der Präprozessor-Direktive `#include` einkopiert werden

```
#include <iostream>

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

Auch diese Technik ist Ihnen – in der Theorie – bereits in Kapitel 1.3.3 vorgestellt worden. Die wichtigsten Fakten möchte ich aber trotzdem hier noch einmal zusammenfassen.

Die `#include`-Direktive sucht nach der angegebenen Datei. Da der Dateiname in eckigen Klammern steht, wird die Datei im Include-Pfad des Compilers gesucht. (Dieser ist nach der Installation üblicherweise automatisch so eingestellt, dass er auf das Verzeichnis mit den Headerdateien der C++-Standardbibliothek verweist. Sie müssen sich in der Regel also nicht weiter um diese Einstellung kümmern.)

Der Inhalt der Datei wird dann an der Stelle der Direktive in die Quelltextdatei einkopiert.

Faktisch fügt die obige `#include`-Direktive also die Deklarationen aller IO-Elemente der C++-Standardbibliothek ein, sodass wir diese Elemente (darunter eben auch `cout`, `<<` und `endl`) verwenden können. Die Einbindung des Namensraums `std` dient dann nur noch der Bequemlichkeit, damit wir die Bibliothekselemente allein mit ihrem Namen ansprechen können (also `cout` statt `std::cout`, siehe Kapitel 1.3.4).

Das engl. Akronym IO steht für Input/Output, zu deutsch also Ein- und Ausgabe.

2.1.5 Kommentare

Wir haben nun fast alle Bestandteile des Quelltextes analysiert. Übrig geblieben sind allein die ersten einleitenden Zeilen:

```
/*****************************************************
 * Hallo Welt-Programm
 *
 *  gibt einen Gruss auf den Bildschirm aus
 */

#include <iostream>
...
```

Bei diesen Zeilen handelt es sich um einen Kommentar. Kommentare dienen dazu, erklärenden Text direkt in den Quellcode einzufügen – quasi als Erklärung oder Gedankenstütze für den Programmierer.

C++ kennt zwei Formen des Kommentars:

- Will man eine einzelne Zeile oder den Rest einer Zeile als Kommentar kennzeichnen, verwendet man die Zeichenfolge //. Alles, was hinter der Zeichenfolge // bis zum Ende der Quelltextzeile steht, wird vom Compiler als Kommentar angesehen und ignoriert.

```
int main()       // Kommentar
```

- Mehrzeilige Kommentare beginnt man dagegen mit /* und schließt sie mit */ ab. Oder Sie müssen jede Zeile mit // beginnen.

```
/* Kommentar
   über mehrere
   Zeilen */
```

Kommentare werden vom Compiler ignoriert, d. h., er löscht sie, bevor er den Quelltext in Maschinencode umwandelt. Sparsam veranlagte Leser brauchen sich also keine Sorgen darüber zu machen, dass eine ausführliche Kommentierung die Größe der ausführbaren Programmdatei aufplustern könnte.

Sinnvolles Kommentieren

So einfache Programme, wie wir sie am Anfang dieses Buchs erstellen, bedürfen im Grunde keiner Kommentierung. Kommentare sind nicht dazu gedacht, einem Programmieranfänger C++ zu erklären. Kommentare sollen gestandenen C++-Programmierern helfen, sich in einen Quelltext einzudenken und diesen zu erklären. Kommentare sollten daher eher kurz und informativ sein. Kommentieren Sie beispielsweise die Verwendung wichtiger Variablen (siehe nachfolgendes Kapitel) sowie die Aufgabe größerer Anweisungsabschnitte. Einfache Anweisungen oder leicht zu verstehende Konstruktionen sollten nicht kommentiert werden.

2.2 Programmerstellung

Um aus dem Programmquelltext *HalloWelt.cpp* ein ausführbares Programm zu erzeugen, müssen wir den Quelltext mit Hilfe des C++-Compilers in Maschinencode übersetzen.

Wie Sie dabei vorgehen, hängt davon ab, welche Entwicklungsumgebung Sie verwenden. Zwei Entwicklungsumgebungen möchte ich Ihnen im Folgenden vorstellen: die Visual-Studio-Community-Edition für Windows -Desktop und den GNU-Compiler für Linux.

2.2.1 Programmerstellung mit Visual Studio

Wenn Sie mit Visual Studio arbeiten, steht Ihnen eine komplette, leistungsfähige Entwicklungsumgebung zur Verfügung. Viele Leser werden die Arbeit mit der grafischen Benutzeroberfläche von Visual Studio als angenehmer empfinden als die Arbeit mit einem Compiler, der von der Konsole aus bedient wird.

Allerdings fällt bei der Arbeit mit einer Entwicklungsumgebung etwas mehr Verwaltungsarbeit an. Zum Beispiel verwaltet Visual Studio alle Dateien und Daten, die zu einem Pro-

gramm gehören, in Form eines Projekts. Der erste Schritt bei der Programmentwicklung mit Visual Studio besteht daher darin, ein passendes Projekt anzulegen.

Projekt anlegen

1. Rufen Sie Visual Studio auf. Sie können das Programm z. B. über die Programmgruppe auswählen (unter Windows 7 zu finden im Programme-Ordner des Start-Menüs, bei Windows 10 als Kachel auf der Startseite oder in der App-Ansicht) oder suchen Sie mit der Systemsuche (Suchfeld im Start-Menü für Windows 7, Suchfeld in der Taskleiste für Windows 10) nach **Visual Studio**.

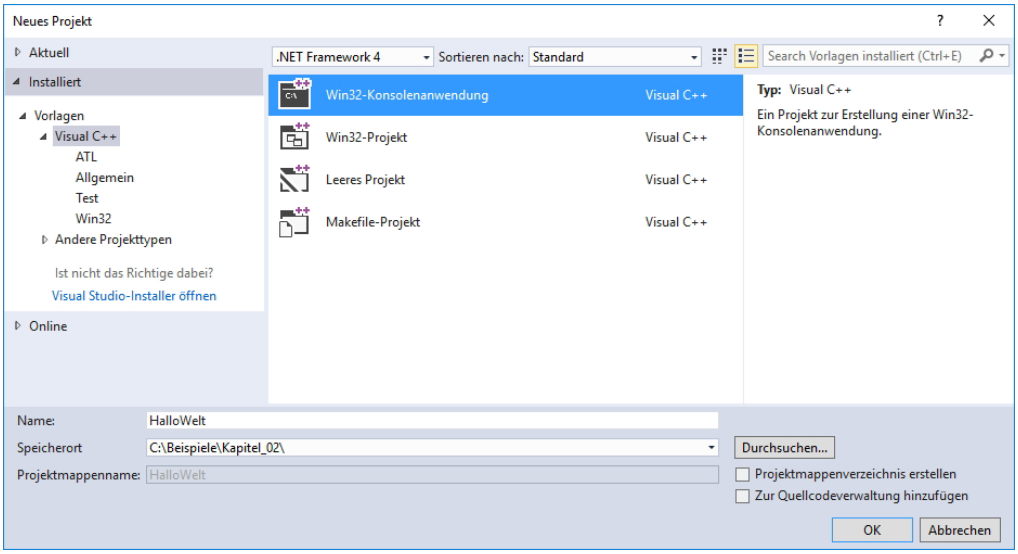

Bild 2.2 Anlegen eines neuen C++-Projekts mit Visual Studio

2. Legen Sie ein neues Projekt an. Rufen Sie dazu den Befehl **Datei/Neu/Projekt** auf.
   - Achten Sie darauf, dass links im Dialogfeld **Neues Projekt** die Vorlagenkategorie **Visual C++** ausgewählt ist. Falls nicht, klicken Sie einfach im linken Teilfenster auf den gleichnamigen Eintrag. Wählen Sie dann im mittleren Fenster die Vorlage **Win32-Konsolenanwendung** aus.
   - Geben Sie einen **Namen** für das Projekt ein, beispielsweise **HalloWelt**, und wählen Sie im Feld **SpeicherOrt** ein übergeordnetes Verzeichnis für das Projekt aus. (Visual Studio wird unter diesem Verzeichnis ein Unterverzeichnis für das Projekt anlegen, das den gleichen Namen wie das Projekt trägt.)
   - Achten Sie darauf, dass die Option **Projektmappenverzeichnis erstellen** deaktiviert ist.
   - Drücken Sie zuletzt auf **OK**.
3. Klicken Sie auf der ersten Seite des aufspringenden Assistenten auf **Weiter**.
4. Deaktivieren Sie auf der zweiten Seite die Option **Vorkompilierter Header** und aktivieren Sie dafür die Option **Leeres Projekt**. Klicken Sie auf **Fertig stellen**.

Wenn Sie die Option **Leeres Projekt** nicht aktivieren, legt Visual Studio für Sie eine *.cpp*-Quelltextdatei mit einem einfachen Programmgerüst an. Wir verzichten allerdings auf dieses Programmgerüst, da es a) nur wenig Arbeitserleichterung bringt und b) kein standardisiertes C++ verwendet.

Vorkompilierte Header dienen dazu, die Programmerstellung zu beschleunigen. Sie werden bei der ersten Kompilierung erstellt und können nachfolgende Kompilierungen beschleunigen. Wenn Sie an größeren Projekten arbeiten, ist dies eine recht nützliche Option. Für unsere kleinen Beispielprogramme können wir allerdings auf den „Header", der viel Speicherplatz belegt, verzichten.

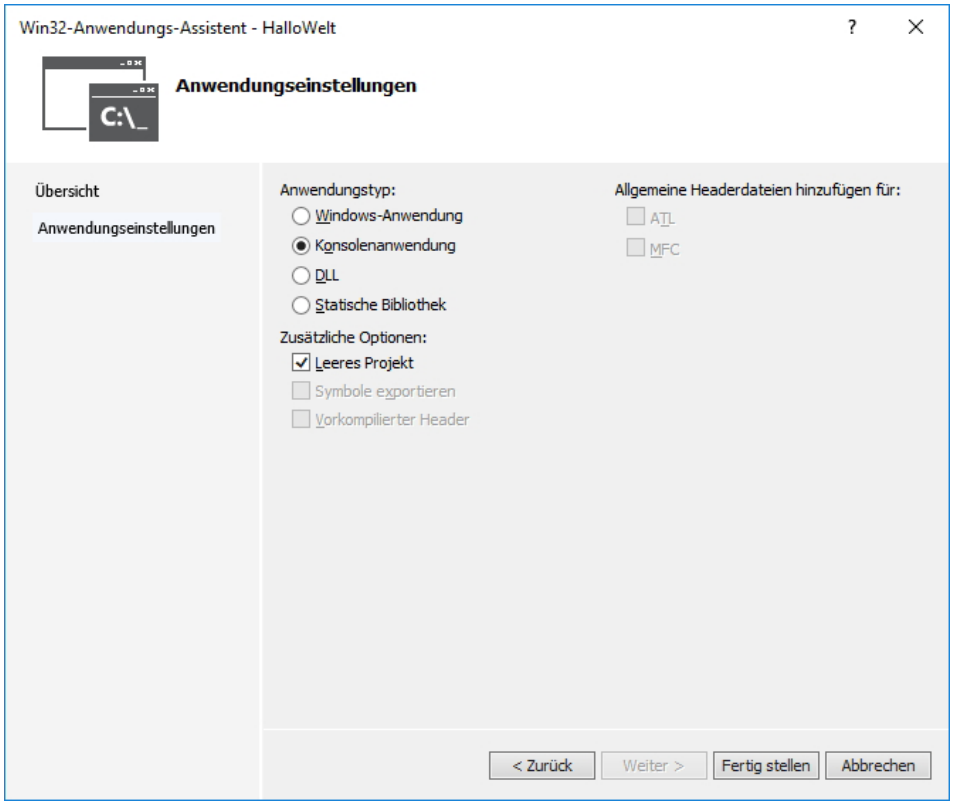

Bild 2.3 Beginnen Sie mit einem leeren Projekt.

Projektmappen

Visual Studio bettet das neue Projekt automatisch in eine Projektmappe ein. Wenn Sie möchten, können Sie über den Befehl **Datei/Neu/Projekt** weitere Projekte in die aktuelle Projektmappe aufnehmen. Sie müssen dann nur im Dialogfenster **neues Projekt** im Listenfeld **Projektmappe** die Option **Hinzufügen** auswählen. Für den Einstieg ist es aber sinnvoller, für jedes neue Programm ein neues Projekt in einer eigenen Projektmappe anzulegen.

Im Projektmappen-Explorer (Aufruf über den gleichnamigen Befehl im Menü **Ansicht**), der standardmäßig links im Visual-Studio-Fenster angezeigt wird, werden alle Projekte der aktuellen Projektmappe zusammen mit den zu den Projekten gehörenden Dateien aufgeführt.

Quelltextdatei hinzufügen

Da wir unsere Arbeit mit einem leeren Projekt begonnen haben, besteht der nächste Schritt darin, dem Projekt eine Quelltextdatei hinzuzufügen.

5. Fügen Sie dem Projekt eine Quelltextdatei hinzu. Klicken Sie dazu mit der rechten Maustaste im Projektmappen-Explorer auf den Projektknoten (in unserem Beispielprojekt ist dies der Knoten mit dem fett dargestellten Namen *HalloWelt*) und rufen Sie im Kontextmenü den Befehl **Hinzufügen/Neues Element** auf.

Falls Sie das Fenster des Projektmappen-Explorers nicht sehen, können Sie es über den Menübefehl **Ansicht/Projektmappen-Explorer** einblenden lassen.

6. Wählen Sie im erscheinenden Dialogfeld die Vorlage *C++-Datei (.cpp)* aus, geben Sie einen Namen für die Datei an und klicken Sie auf **Hinzufügen**.

Die Beispielprogramme der ersten Teile bestehen meist nur aus einer Quelltextdatei, die dann der Einfachheit halber und zur leichteren Identifizierung den Namen des Projekts trägt.

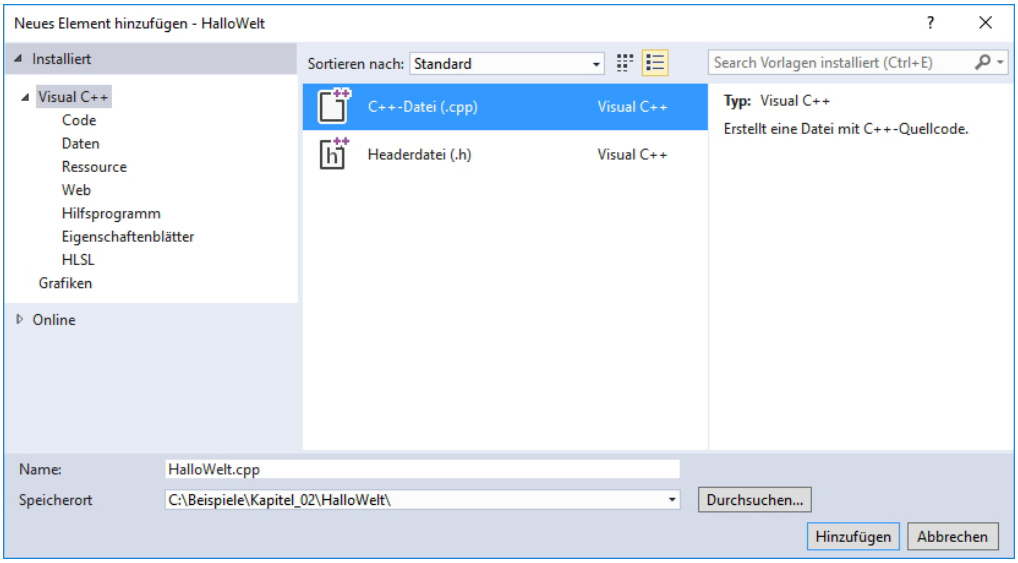

Bild 2.4 Dem Projekt eine Quelltextdatei hinzufügen