C++ für Spieleprogrammierer

Bearbeitet von
Heiko Kalista

5., aktualisierte und erweiterte Auflage 2016. Buch. 515 S. Hardcover
ISBN 978 3 446 44644 1
Format (B x L): 18,2 x 24,5 cm
Gewicht: 1057 g

HANSER

Leseprobe

Heiko Kalista

C++ für Spieleprogrammierer

ISBN (Buch): 978-3-446-44644-1

ISBN (E-Book): 978-3-446-44805-6

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44644-1

sowie im Buchhandel.

Inhalt

Vorwort

Hallo und herzlich willkommen! Du möchtest also gerne das Spieleprogrammieren lernen und hast Dich für dieses Buch entschieden. Genau jetzt, wenn Du diese Zeilen liest, befindest Du Dich auf den ersten Metern eines langen Weges, den es sich jedoch definitiv lohnt zu gehen.

Vor über zwei Jahrzehnten brach eine neue Ära an, als die Homecomputer die heimischen Wohnzimmer eroberten. Recht schnell fanden viele Leute Gefallen daran, nicht nur fertige Computerspiele zu spielen, sondern selbst herauszufinden, wie man solche eigenständig entwickeln kann. „Herauszufinden“ trifft dabei den Nagel auf den Kopf, denn Medien wie das Internet waren zu dieser Zeit nicht vorhanden und Buchmaterial war kaum bis gar nicht in den Regalen der Buchhandlungen zu finden. Aus diesem Grund trennte sich schnell die Spreu vom Weizen, und manche gaben auf. Manche klemmten sich jedoch so lange dahinter, bis die gewünschten Resultate über den Bildschirm flackerten. Genau dieser „Forschergeist“ ist es, der einen Spieleprogrammierer ausmacht. Man muss Geduld haben, bereit sein zu lernen und einen unermüdlichen Drang besitzen, sein Vorhaben in die Tat umzusetzen. Dazu könnte es keine bessere Zeit geben als die heutige. Im Internet gibt es eine Fülle von Informationen, Hilfestellungen und sogar ganze Communities, die sich mit dem Thema Spieleprogrammierung befassen. Früher bestanden die Quelltexte (Programmcodes) von Spielen noch aus einer endlosen Reihe von kryptischen, meist aus drei Buchstaben bestehenden Befehlen und unübersichtlichen Sprunganweisungen. Heute hingegen hat man mit C++ eine mächtige Hochsprache in der Hand, die leicht zu erlernen ist und viele Dinge einfacher und logischer macht. Informationen sind leichter zugänglich, und Spieleprogrammierer sind längst keine kleine Gruppe mehr, die sich gut gehütete Geheimnisse teilt. Heute ist es jedem möglich, sich selbst mit dieser Thematik zu befassen, solange er bereit ist, sich intensiv damit auseinander zu setzen.

Sicherlich ist mit der wachsenden Leistung der Computer und den immer höher werdenden Anforderungen an das System auch die Komplexität gestiegen. Dabei bleiben jedoch bestimmte Grundlagen immer gleich. Wer schon eine Programmiersprache beherrscht oder sich generell bereits mit der Thematik des Programmierens befasst hat, wird hier einige Dinge antreffen, die ihm bekannt vorkommen werden. Trotzdem ist Stillstand der größte Feind eines Spieleprogrammierers. Deshalb ist es enorm wichtig, die mittlerweile unglaublich ergiebige Informationsflut im Internet zu nutzen und immer auf dem neuesten Stand zu bleiben. Schneller als je zuvor kann man sich heute über das Internet die gewünschten Informationen beschaffen. Jedoch birgt diese Möglichkeit der Informationssuche auch eine

Gefahr: Man lässt sich schnell verleiten, den eigenen Kopf zugunsten einer Suchmaschine auszuschalten. Deshalb möchte ich an dieser Stelle unbedingt darauf hinweisen, dass man zuerst selbst versuchen sollte, ein Problem zu lösen, und erst dann auf Diskussionsforen oder Ähnliches zugreift, wenn man selbst wirklich nicht mehr anders weiterkommt. Diese Vorgehensweise ist enorm wichtig, um auf lange Sicht Erfolg zu haben. Oftmals findet man selbst eine bessere oder effektivere Lösung, wenn man darüber nachdenkt, statt andere für sich denken zu lassen. Nutze deshalb das Internet als Werkzeug und nicht, um den eigenen Kopf zu ersetzen.

Viele Bücher über die Spieleprogrammierung setzen an einem Punkt an, an dem schon weitreichende Vorkenntnisse der Programmiersprache C++ gefordert sind, oder bieten nur einen kleinen Crash-Kurs im Programmieren an. Andere Bücher befassen sich zwar mit der Programmiersprache C++, ohne dabei jedoch einen Bezug zur Spieleprogrammierung herzustellen. Dieses Buch versucht nun, diese Lücke zu schließen und den Grundstein zu legen, der auf dem Weg zum Spieleprogrammierer nötig ist: Das Erlernen der Programmiersprache C++. Nach dem Durcharbeiten dieses Buches solltest Du also über das essenzielle Grundlagenwissen verfügen, das nötig ist, um erfolgreich Spiele zu entwickeln.

In diesem Sinne: Let's code!

Anmerkungen zur fünften Auflage

Als ich vor über zehn Jahren die Idee hatte, ein Buch über C++ mit Bezug zur Spieleprogrammierung zu schreiben, hätte ich nie gedacht, dass dieses Buch einmal in die fünfte Auflage gehen würde. Doch nun ist es tatsächlich so weit und ich blicke zugegebenermaßen ein wenig erstaunt auf den kleinen Stapel Bücher, der gerade neben mir liegt. Erstaunt darüber, wie viel Zeit seit der ersten Auflage vergangen ist und welchen Wandel das Buch hinter sich hat. Beinhaltete die erste Auflage lediglich zehn Kapitel, wurde die zweite Auflage um drei Kapitel erweitert. Grundlagen der Windows-Programmierung, weitere fortgeschrittene Themen sowie ein kleines, aber vollständiges Spiel mit der SDL kamen hinzu. Dies war durch das rege Feedback und die tollen Vorschläge der Leser der ersten Auflage möglich.

In der dritten Auflage wurde das Buch hinsichtlich der verwendeten Entwicklungsumgebungen aktualisiert. Von Visual Studio 6.0 und Visual Studio .net 2003 fand ein Umstieg auf die Visual Studio 2008 Express Edition statt, die wesentlich moderner und benutzerfreundlicher war. Zudem gab es diese Entwicklungsumgebung kostenlos – eine feine Sache für Einsteiger und Hobby-Entwickler.

Mit der vierten Auflage wurde neben der Aktualisierung auf Visual Studio 12 eine weitere Entwicklungsumgebung auf einem anderen Betriebssystem vorgestellt: Xcode unter Mac OS X. Somit konnten auch Leser, die lieber auf Apple-Computern arbeiten, das Buch verwenden. Lediglich bei den Beispielen zur Windows-Programmierung gab es naturgemäß Einschnitte. Da der Quellcode zu den restlichen Beispielen fast komplett plattformunabhängig war, musste nur an wenigen Stellen auf systemspezifische Unterschiede eingegangen werden. Doch gerade in der heutigen Zeit, in der es eine größere Plattformvielfalt gibt als je zuvor (nicht zu vergessen die ganze Mobil-Sparte), ist es besonders wichtig, auch einmal über den Tellerrand hinauszuschauen und nicht nur auf einem einzigen System zu entwickeln.

In dieser fünften Auflage wurden die verwendeten Entwicklungsumgebungen ebenfalls wieder auf den neuesten Stand gebracht. Zudem hielt ich es für angebracht, die bestehenden Kapitel um einige Themen zu ergänzen, die bisher außen vor blieben. Dazu gehören Unions, Namensbereiche, der ternäre Operator, bitweise Operatoren und einiges mehr. Außerdem wurde das gesamte Kapitel 12 („Ein Spiel mit der SDL“) auf Version 2 der SDL aktualisiert.

Alle diese Auflagen wären ohne das Feedback und die konstruktiven Vorschläge der Leser und der Community nicht möglich gewesen. Aus diesem Grunde möchte ich mich hier noch einmal ganz besonders bei allen Lesern bedanken!

Danksagung

Bevor es nun losgeht, möchte ich mich an dieser Stelle bei allen Leuten bedanken, die mich beim Schreiben dieses Buches unterstützt haben und mithalfen, es zu verwirklichen. Marc Kamradt und Jörg Winterstein haben sich mühsam durch alle Kapitel geackert und mir überall dort auf die Finger geklopft, wo es angebracht war. Jörg hat es dabei immer wieder geschafft, mich mit seinem unverwechselbaren Humor in seinen Kommentaren zum Lachen zu bringen. Ohne Eure Hilfe wäre dieses Buch nicht zustande gekommen. Danke, Jungs!

Für die Grafiken des kleinen Demo-Spiels möchte ich mich bei Thomas Schreiter und seiner gestalterischen Zauberhand bedanken, die mich immer wieder verblüfft.

Matthias Gall und Mathias Ricken hatten immer dann ein offenes Ohr, wenn eine Frage auftauchte und beantwortet werden musste.

Peter Schraut und David Scherfgen haben sich ebenfalls einige Kapitel vorgenommen und mit vielen nützlichen Hinweisen und Vorschlägen zu diesem Buch beigetragen. Euch beiden ein dickes Dankeschön hierfür. Und David: Öle deine Maus und mach die Tastatur startklar … bald geht ein gewisses Spiel in eine neue Runde. Ach ja, Peter: Warum liegt denn da Stroh rum?

Fernando Schneider vom Hanser Verlag hat mir mit Rat und Tat zur Seite gestanden, wenn es um dieses Buchprojekt ging, und uns auf der Dusmania hilfreich unter die Arme gegriffen. Ich hoffe, dass es Dir gut geht und wir uns mal wieder sehen!

Weiterhin möchte ich mich bei Sieglinde Schärl, Julia Stepp und Irene Weilhart sowie allen anderen Mitarbeitern des Hanser Verlages für die gute Zusammenarbeit bedanken. Bei Sieglinde Schärl möchte ich mich vor allem für ihre große Geduld bedanken, denn mein ständiges „Ja, nein, vielleicht, vielleicht doch nicht“ bei Neuauflagen ist sicherlich nicht einfach.

Der größte Dank gilt meiner Mutter und allen Freunden, die daran geglaubt haben, dass die Idee dieses Buches auch in die Tat umgesetzt wird.

Spezieller Dank geht an Marc Grimm, dafür, dass er da ist!

Dreieich, im Dezember 2015

Heiko Kalista


geloest@icloud.com

1 Grundlagen

1.1 Einleitung

1.1.1 An wen richtet sich dieses Buch?

Dieses Buch ist für alle gedacht, die sich für die Thematik der Spieleprogrammierung interessieren und noch keinerlei Vorkenntnisse haben. Der Schwierigkeitsgrad beginnt bei null und steigert sich langsam, aber stetig im Verlauf der einzelnen Kapitel. Weiterführende Bereiche wie etwa die Grafikprogrammierung werden in diesem Buch zwar auch behandelt, aber für dieses Thema ist „nur“ ein Kapitel vorgesehen, das Dich ein wenig tiefer in die Materie einführen soll. Das Buch bezieht sich ausschließlich auf die Programmiersprache C++ und fokussiert dabei die Spieleprogrammierung.

Im Grunde spricht jedoch nichts dagegen, dieses Buch als Nachschlagewerk zu verwenden. Generell ist es für jeden nützlich, der die Programmiersprache C++ erlernen oder vertiefen will.

1.1.2 Welche Vorkenntnisse werden benötigt?

Um mit diesem Buch arbeiten zu können, werden keinerlei Programmierkenntnisse benötigt. Das Einzige, was man beherrschen sollte, ist der Umgang mit Windows oder Mac OS X. Wer weiß, wie man einen Computer einschaltet, Programme startet und mit Dateien arbeitet, ist fast schon überqualifiziert. Es wird erklärt, welche Programme man benötigt und wie man mit ihnen Quelltexte (Programmcode) erstellt, diese kompiliert (in eine für den Computer verständliche Sprache verwandelt) und schließlich lauffähige Programme erzeugt.

Solltest Du bereits über Programmierkenntnisse einer beliebigen anderen Sprache verfügen, so wird dies sicherlich nützlich sein, ist aber – wie schon erwähnt – nicht notwendig.

1.1.3 Wie arbeitet man am effektivsten mit diesem Buch?

Das Buch ist so gestaltet, dass die einzelnen Kapitel aufeinander aufbauen. Jedes Kapitel erfordert es, dass man die Themen aus den vorangegangenen Kapiteln durchgearbeitet und auch verstanden hat.

In den meisten Kapiteln wird es ein paar Aufgabenstellungen geben, die helfen sollen, das bisher Gelernte zu festigen. Dabei werden zu jeder einzelnen Aufgabe verschiedene Tipps gegeben, wie man sich am besten an die Lösung heranmacht. Es soll keinesfalls der Eindruck entstehen, dass es sich wie in der Schule um Hausaufgaben handelt. Vielmehr soll Dir hier die Möglichkeit gegeben werden, selbst zu kontrollieren, ob Du die bisher erklärten Themen wirklich verstanden hast. Praxis ist und bleibt eben ein wichtiger Aspekt, wenn nicht gar der wichtigste.

Am besten schaust Du Dir die jeweilige Aufgabe an und liest die dazugehörigen Tipps durch. Danach überlegst Du, wie man am besten diese Aufgabe lösen könnte. Gib nicht gleich auf, sondern versuche wirklich, zum Ziel zu kommen. Der Schwierigkeitsgrad ist so gewählt, dass man ohne Weiteres zur Lösung kommt, wenn man die vorhergehenden Kapitel gelesen und verstanden hat.

Direkt im Anschluss erfolgt eine Musterlösung, die Du Dir auch dann anschauen solltest, wenn Du die Aufgabe erfolgreich gelöst hast. Auf keinen Fall solltest Du Dir diese Lösung anschauen, bevor Du nicht selbst versucht hast, die Aufgabe zu lösen. Es ist ein gewaltiger Unterschied, ob man sagt „ja klar, so hätte ich das auch gemacht“ oder ob man es tatsächlich erst mal selbst versucht. Es ist zwar nicht zwingend erforderlich, sich mit diesen Aufgaben zu beschäftigen, um die nächsten Kapitel zu verstehen, dennoch ist Eigeninitiative der Schlüssel zum Erfolg.

Zu gegebener Zeit wird es auch sogenannte Fehlerquelltexte geben. Das sind kleine Programmbeispiele, die typische Fehler enthalten, wie sie häufig vorkommen. Das Ganze hat den Sinn, ein Gespür dafür zu entwickeln, wo überall etwas schiefgehen kann, welche Arten von Fehlern es gibt (Compilerfehler, Laufzeitfehler, Warnungen). Jeder wird früher oder später selbst mal die vertracktesten Fehler in seinen Quelltext einbauen und sich dann kopfkratzend auf die Suche danach machen. Da es gerade am Anfang nicht immer gleich ersichtlich ist, wo ein Fehler stecken könnte, halte ich es für sinnvoll, so früh wie möglich darauf einzugehen, wie man in einem solchen Fehlerfall am besten vorgeht. Die Fehlerquelltexte in diesem Buch enthalten diverse kleine Gemeinheiten und natürlich auch Hilfestellungen, wie man sich am besten auf die Suche begibt. Auch ein noch so erfahrener Programmierer ist nicht vor solchen Dingen gefeit. Allerdings kommt mit der Zeit die Erfahrung, solche Fehler schneller einzugrenzen.

Wie auch schon bei den Aufgabenstellungen wird es im Anschluss eine korrigierte Fassung des Quelltextes geben. Dabei wird erklärt, wie man den einzelnen Fehlern schnellstmöglich auf die Schliche kommt und wie man sie hätte vermeiden können. Diese Übungen sind fast noch wichtiger als die Aufgabenstellungen, da es sehr ärgerlich ist, einen Quelltext, den man in fünf Minuten geschrieben hat, zwei Stunden lang nach Fehlern zu durchforsten.

1.1.4 Geduld, Motivation und gelegentliche Tiefschläge

Ich möchte hier nicht um den heißen Brei herumreden, aber gelegentliche Tiefschläge gehören nun leider dazu. Früher oder später kommt jeder mal an den Punkt, an dem einfach nichts klappen will und das Programm nicht das macht, was es eigentlich soll. Genau an diesem Punkt gibt es zwei Wege, die angehende Programmierer einschlagen können: Die einen geben frustriert auf und werfen den Compiler samt PC und Monitor aus dem Fenster.

Danach wird mächtig über die Programmiersprache geschimpft: Sie ist zu schwer, zu kompliziert, man muss ja studiert haben, um das zu verstehen, und so weiter.

Die anderen jedoch werfen nicht gleich die Flinte ins Korn und bleiben stattdessen hartnäckig am Ball. Natürlich ist das nicht immer einfach, jedoch gibt es auch hier Tricks, wie man sich ganz schnell wieder motiviert. Manchmal ist es ganz nützlich, den Compiler auszuschalten und sich mit anderen Dingen zu beschäftigen, sei es nun an den See zu gehen oder ein paar coole Spiele mit bombastischen Effekten zu spielen. Gerade wenn man sich auf den Homepages von Hobbyentwicklern umschaut und sich deren Resultate ansieht, ist man schnell wieder motiviert und denkt sich: „Hey, die haben das doch auch hinbekommen, warum sollte mir das nicht auch gelingen?“

Man sollte sich auch vor Augen halten, dass die heute professionellen Entwickler auch nichts in den Schoß gelegt bekommen haben. Alle haben sie mit einem winzig kleinen Programm angefangen, das ein simples „Hallo Welt“ auf dem Bildschirm ausgibt. Damit hat es sogar etwas ganz Besonderes auf sich: Irgendwie ist es zur Gewohnheit geworden, dass viele Bücher über diverse Programmiersprachen mit einem Beispiel beginnen, das den Text „Hallo Welt“ auf dem Bildschirm ausgibt. Das geht nun sogar so weit, dass man mit Microsoft Visual C++-Arbeitsumgebungen automatisch ein solches Beispielprogramm erzeugen kann.

Wenn man also an einem solchen Tiefpunkt angelangt ist, sollte man definitiv nicht von „Scheitern“ reden. Es scheitert nur der, der aufgibt. Es gibt im Internet eine ganze Fülle von Diskussionsforen und Chats, in denen sich Programmierer austauschen und gegenseitig helfen können. Scheue Dich nicht davor, dort Fragen zu stellen. Schneller als Du denkst, wirst Du nicht nur Fragen stellen, sondern sie auch beantworten. Außerdem werden hier Tipps gegeben, wie man am schnellsten in die sogenannte „Szene“ einsteigen kann und was es zu beachten gibt. Es ist also keine schlechte Idee, Kapitel 13 schon etwas früher aufzuschlagen.

Mir ist es nun schon einige Male passiert, dass mich jemand darum gebeten hat, mal seinen Quelltext durchzuschauen und nachzusehen, wo der Fehler liegt. Oft war es so, dass ich den Fehler nach recht kurzer Zeit gefunden habe. Wenn man dann gefragt wird, wie man so schnell das Problem eingegrenzt hat, gibt es eigentlich nur eine einzige Antwort: Ich habe diesen Fehler oft genug selbst gemacht! Zwar habe ich zu diesem Zeitpunkt auch mit dem Gedanken gespielt, der CPU jedes Beinchen einzeln auszureißen, Kaffee in die Tastatur zu kippen oder die Aerodynamik meines Monitors im freien Fall zu testen. Da so etwas aber auf die Dauer recht teuer wird, ist es sinnvoller, sich mit dem Problem zu befassen, um es letzten Endes auch zu lösen.

Das alles hört sich schlimmer an, als es ist, aber ich finde es wichtig, auch auf diese Dinge hinzuweisen. Lass Dich auf keinen Fall dadurch entmutigen, sondern behalte das eben Gesagte einfach im Hinterkopf – es kann und wird sich als nützlich erweisen.

1.1.5 Das Begleitmaterial zum Buch

Unter *http://downloads.hanser.de* findest Du sämtliche Quelltexte aus dem Buch. Wie Du diese Beispiele mit Deiner Entwicklungsumgebung laden und verwenden kannst, erkläre ich Dir ab Abschnitt 1.4.3.

Alle Quelltexte wurden unter Windows 10 mit Microsoft Visual Studio 2015 Community Edition getestet und kompiliert. Auf dem Mac wurden Mac OS X 10.10.15 (Yosemite) und Xcode 6.4 verwendet. Die Beispiele 10.1 und 10.3 aus Kapitel 10 sind unter Mac OS X mit Xcode nicht lauffähig, da sie Windows-spezifische Funktionen verwenden. Diese speziellen Beispiele sind somit nur als Visual Studio 2015-Projekte, nicht jedoch als Xcode-Projekte vorhanden.

Die jeweils aktuellste Version von Visual Studio 2015 Community Edition findest Du unter *http://www.microsoft.com/visualstudio/deu/downloads*. Wie bereits erwähnt ist diese Version kostenlos. Nach Ablauf des Testzeitraums ist jedoch eine (ebenfalls kostenlose) Registrierung nötig.

Wenn Du unter Mac OS X arbeitest, dann kannst Du Dir Xcode kostenlos im App Store herunterladen.

Alle für das Buch notwendigen Materialien und Software-Tools findest Du auf den folgenden Internetseiten:

http://downloads.hanser.de

http://www.microsoft.com/visualstudio/deu/downloads

1.1.6 Fragen zum Buch

Wenn Du rund um das Buch Fragen, Kritik oder Anregungen hast, dann schaue auf der Seite *www.spieleprogrammierer.de* vorbei. Dort findest Du ein Diskussionsforum, Neuigkeiten über das Buch und vieles mehr.

1.2 Die Programmiersprache C++

Nach dieser kleinen Einleitung möchte ich Dir nun einen kurzen Einblick in die Entstehungsgeschichte der Programmiersprache C++ geben. Es soll klar werden, welche Idee hinter dieser Sprache steckt und was man unter objektorientiertem Programmieren versteht. Begriffe wie „OO“ oder „ANSI-Standard“ sollten nach diesem Kapitel zumindest keine böhmischen Dörfer mehr sein, auch wenn böhmische Dörfer durchaus sehr hübsch sind.

1.2.1 Von Lochkarten zu C++

C++ ist eine sogenannte Hochsprache. Um zu verstehen, was eine Hochsprache eigentlich ist, muss man ein klein wenig die Zeit zurückdrehen und sich die Programmiersprachen der früheren Tage anschauen. Zu den Zeiten des Commodore 64 und des Amigas (beide waren die bekanntesten und beliebtesten Homecomputer der 80er- und 90er-Jahre) war Assembler noch die vorherrschende Sprache. Assembler ist eine sogenannte „maschinennahe“ Sprache und zeichnet sich durch einen vergleichsweise geringen Befehlssatz aus. Das bedeutet, dass die zur Verfügung stehenden Befehle nicht gerade mächtig sind. Aus diesem Grund benötigt man sehr viele von ihnen, was zur Unübersichtlichkeit und Komplexität beiträgt. Man hat in der Regel nur Rechenoperationen zur Verfügung und Befehle, um Speicherstellen zu beschreiben oder auszulesen. Weiterhin sind solche Assembler-Programme mit Sprungbefehlen durchsetzt, was die Lesbarkeit noch um einiges verschlechtert. Der Assembler-Befehlssatz des Commodore C64 bot nicht einmal die Möglichkeit, zwei Zahlen einfach so zu multiplizieren oder zu dividieren.

Glücklicherweise haben sich die Zeiten geändert, und die Programmiersprachen sind weitaus komfortabler geworden. Während man früher den Computer mit Lochkarten gefüttert hat, um ihn zu programmieren, stehen einem heute mächtige und sehr leistungsfähige Programmiersprachen zur Verfügung. Eine der Etappen zwischen reiner Maschinensprache und C++ waren sogenannte Interpreter, die es bis heute gibt. Die Programmiersprache Basic vom Commodore C64 wäre hier als Beispiel zu nennen. Man verfügte über einen größeren Befehlssatz und konnte gewisse Aufgaben wesentlich einfacher und schneller bewältigen. Die Ausgabe von Grafik, Text und Musik war um einiges leichter. Dass es sich um einen Interpreter handelte, bedeutete allerdings, dass das Basic-Programm zur Laufzeit – also während das Programm ausgeführt wird – Zeile für Zeile in Maschinencode umgewandelt wurde (jede Hochsprache muss generell in Maschinencode umgewandelt werden). Dieser Vorgang kostete natürlich eine ganze Menge Zeit, und flüssig laufende Spiele konnte man mit Basic kaum erstellen.

C++ ist nun eine Sprache, die nicht mehr zur Laufzeit in Maschinencode umgewandelt (interpretiert) wird. Stattdessen erledigt ein sogenannter Compiler diese Arbeit schon vor dem Ausführen des eigentlichen Programms. Natürlich besteht das endgültige, ausführbare Programm immer noch aus einer schier endlos langen Folge von Nullen und Einsen. Jedoch brauchen wir nicht mit dem Locher vor dem PC zu sitzen und uns unsere Spiele aus Papierstreifen zu erstellen, der Compiler nimmt uns diese Arbeit ab. Der gesamte C++-Quelltext wird also in einen für den Computer verständlichen Maschinencode verwandelt, und eine .exe-Datei (ausführbare Datei) entsteht. Der Vorteil liegt auf der Hand: Man hat eine für den Menschen leicht lesbare, logisch aufgebaute Programmiersprache, die vom Compiler in rasanten Maschinencode umgewandelt wird. Wir brauchen uns keine Sorgen mehr zu machen, dass die Geschwindigkeit stark beeinträchtigt wird, und gleichzeitig sind wir davon entbunden, uns mit kryptischen Maschinenbefehlen herumzuschlagen.

Allerdings gab es noch einen Vorgänger der Programmiersprache C++, nämlich einfach nur „C“. C++ ist nun nicht eine völlig neue Sprache, sondern eine Erweiterung von C. Der eigentliche und wichtigste Unterschied liegt darin, dass C++ „objektorientiert“ ist. Was dies genau bedeutet, klären wir im nächsten Abschnitt.

Trotz all diesen tollen Vorteilen schreibt sich ein Programm natürlich nicht von selbst, und es wird schon gar nicht von alleine strukturiert, ordentlich und lesbar werden. Letztendlich liegen diese Dinge immer noch beim Programmierer selbst, und das wird wohl auch immer so bleiben. Natürlich kann man sämtliche Regeln des guten Programmierstils außer Acht lassen und weiterhin „Spaghetti-Code“, sprich unlesbaren, wirren Code, schreiben. Doch dies liegt dann am Programmierer und nicht an der Programmiersprache.

1.2.2 Objektorientiertes Programmieren

Um zu verstehen, was objektorientierte Programmiersprachen ausmachen, muss man noch mal eine kleine Zeitreise machen. Wie vorangehend schon erwähnt, bestanden Quelltexte früher meist aus Rechenoperationen, Vergleichsanweisungen und Sprungbefehlen. Wenn etwa geprüft werden sollte, ob ein gegnerisches Raumschiff noch über genügend Energie verfügt, mussten Speicherbereiche ausgelesen und ausgewertet werden. Je nach Inhalt dieser Speicherstellen wurde dann an andere Stellen im Programm verzweigt. In kleineren Programmen war dies nicht unbedingt dramatisch, jedoch raufte man sich bei großen Projekten meist die Haare, wenn man sämtliche Verzweigungen in einem Programm nachvollziehen musste. Wollte man nun eine Vielzahl von Gegnern verwalten, hatte man eine Menge Arbeit vor sich. Mit dem recht spartanischen Befehlssatz war es teilweise eine echte Qual, sämtliche Positionen, Lebensenergien und so weiter sinnvoll zu verwalten. Im Grunde mussten jeder Gegner und jedes im Spiel vorkommende Objekt getrennt behandelt werden.

Die Programmiersprache C++ stellt hingegen eine Menge Möglichkeiten zur Verfügung, um Objekte zu gruppieren, ihnen ihre eigenen individuellen Daten und Funktionen zuzuweisen und sie strukturierter zu verwalten. Später wirst Du lernen, was Klassen, Strukturen und Funktionen sind und wie man sinnvoll mit ihnen umgeht.

Ziel der objektorientierten Programmierung ist (nicht nur in meinen Augen) das Zusammenfassen von zusammengehörenden Dingen in immer wiederverwertbare Teile und Abschnitte. Das bedeutet, dass dem Programmierer die Arbeit enorm erleichtert wird, wenn er einmal gewisse grundlegende Eigenschaften festgelegt hat. Er kann beispielsweise festlegen, wie sich ein Raumschiff allgemein verhalten soll, und später davon ein spezielles Raumschiff ableiten, ohne viel im Quelltext zu ändern. In den späteren Kapiteln werde ich noch etwas genauer darauf eingehen, da es ohne pragmatische Beispiele nur schwer möglich ist, das Konzept der objektorientierten Programmierung sinnvoll zu verdeutlichen.

1.2.3 Der ANSI-Standard

Wie Du sicherlich weißt, haben Menschen den Drang, alles zu vereinheitlichen, allen Dingen einen Namen zu geben und alles Mögliche zu standardisieren, was ja auch einen Sinn hat (damit Raumschiffe an eine Station andocken können, müssen sie zum Beispiel eine einheitliche Andockvorrichtung besitzen). Hier bei uns in Deutschland ist DIN das bekannteste Beispiel (Deutsches Institut für Normung). In den vereinigten Staaten ist das Äquivalent dazu das „American National Standards Institute“ (ANSI).

Irgendwann kam der Zeitpunkt, an dem es jemand für sinnvoll hielt, auch die Programmiersprache C++ zu vereinheitlichen und gewisse Standards festzulegen. Im ersten Moment mag man sich fragen, wozu so etwas nötig ist. Denkt man etwas darüber nach, kommt man zu dem Schluss, dass es verschiedene Hersteller von Compilern (Microsoft, Borland u. v. m.) gibt, die sich noch dazu je nach Betriebssystem unterscheiden (Windows, Linux). Würde nun jeder Compilerhersteller einige Variationen einbauen, wäre das Chaos im wahrsten Sinne des Wortes schon vorprogrammiert, und das Portieren von Quellcode vom einen System auf das andere wäre mit sehr viel Aufwand verbunden. Unter Portierung versteht man das Umsetzen eines Programms auf ein anderes Betriebssystem. Wenn sich nun sowohl die Compilerhersteller als auch die Programmierer an den ANSI-Standard halten, ist die Portierbarkeit sichergestellt, und es gibt wesentlich weniger Probleme. Wie man sieht, eine sehr nützliche Sache.

Ein weiterer Punkt, der für den ANSI-Standard spricht, ist die sogenannte STL (Standard Template Library). Was genau diese STL ist, lässt sich an dieser Stelle noch nicht so einfach erklären. Allgemein kann man sagen, dass es sich um eine Art Funktionsbibliothek handelt, die einem viele nützliche Dinge bietet, die man nicht mehr selbst neu programmieren muss. Da gerade die STL ein sehr wichtiges Thema ist, wurde ihr in diesem Buch gleich ein ganzes Kapitel (Kapitel 9) gewidmet. Doch was hat diese STL nun mit dem ANSI-Standard zu tun? Nun, alle Funktionen in dieser Bibliothek sind streng nach dem ANSI-Standard programmiert und können somit von jedem Compiler fehlerlos verwendet werden (natürlich nur dann, wenn sich dieser auch an den ANSI-Standard hält).

Wer sich gerne genauer mit den Bestimmungen der ANSI-Norm beschäftigen möchte, sollte sich im Internet etwas umschauen. In diesem Buch werde ich nur so weit auf diese Norm eingehen, dass sichergestellt ist, dass sämtliche Code-Beispiele auf den gängigsten Compilern laufen.

1.2.4 Warum gerade C++?

Möglicherweise stellst Du Dir an dieser Stelle die Frage, warum man eigentlich ausgerechnet C++ für die Spieleprogrammierung verwenden sollte. Schließlich gibt es noch eine ganze Reihe anderer Programmiersprachen wie zum Beispiel Visual Basic, Java oder Delphi. Natürlich ist es möglich, auch mit diesen Sprachen ein funktionierendes Spiel zu programmieren. Die entscheidenden Punkte sind jedoch der Komfort der Sprache sowie die Systemnähe. Damit ist gemeint, wie weit es die Sprache zulässt, die Dinge selbst in die Hand zu nehmen. Nicht alle Programmiersprachen unterstützen sogenannte „Zeiger“, die für effektives Programmieren und schnelle Programme unerlässlich sind. Was es mit diesen Zeigern auf sich hat, werden wir allerdings erst später klären. (Ja, ja, ich weiß, ich verschiebe dauernd Themen nach hinten.) Leider lässt sich das aber nicht immer vermeiden, wie wir später noch sehen werden. An dieser Stelle möchte ich nur erwähnen, dass diese Zeiger sehr wichtig sind und man sie sich eigentlich nicht mehr aus der Spieleprogrammierung wegdenken kann. Man kann mit ihrer Hilfe zum Beispiel selbst Speicher reservieren, was eine ganze Menge Vorteile bringt.

Weiterhin sind viele SDKs (Software Development Kit) für C++ optimiert. Zwar kann man einige dieser SDKs auch mit Visual Basic oder Delphi verwenden, jedoch eben nicht alle. Als

Beispiel könnte man das DirectX-SDK nennen, das sogenannte Bibliotheksdateien und fertigen Quellcode zur Verfügung stellt, um Multimedia-Anwendungen zu programmieren (also die von uns heiß ersehnten eigenen Spiele).

Aus diesen Gründen ist C++ die vorherrschende Sprache bei der Spieleprogrammierung, und sie wird es wohl auch noch eine ganze Weile lang bleiben. Effektivität, eine Menge Freiheiten und die enorme Systemnähe machen C++ zu der Nummer eins der Programmiersprachen, besonders dann, wenn es um das Entwickeln von Spielen geht.

Es gibt eine Menge Leute, die der Meinung sind, dass „Neulinge“ erst einmal eine einfachere Sprache als C++ lernen sollen. Davon halte ich nicht viel, denn C++ ist auch nicht schwerer zu erlernen als andere Programmiersprachen. Wenn man sich mit der Thematik beschäftigt, braucht man nicht erst den Umweg über andere Sprachen zu gehen. Warum sollte man lernen, wie man eine Dampflok fährt, wenn der ICE direkt daneben steht? Wenn man C++ von Anfang an lernt und konsequent am Ball bleibt, trifft man in meinen Augen die richtige Entscheidung.

1.3 Jetzt geht es los … unser erstes Programm

So, nun ist es endlich so weit. Wir haben eine ganze Menge an theoretischem Geschwafel hinter uns gebracht und können uns jetzt endlich unserem ersten Programm widmen. Ich werde im gesamten Buch so verfahren, dass ich zuerst den kompletten Quelltext zeige und danach mit der Erklärung beginne. Alle neuen Befehle und die nötigen Erklärungen dazu erfolgen dann Schritt für Schritt und Zeile für Zeile. Das hat den Sinn, dass man den gesamten Quelltext auf einen Blick hat und nicht die einzelnen Bruchstücke im Kapitel zusammensuchen muss.

Nachdem der Sinn und die Funktionsweise des ersten Programms erklärt wurden, werde ich Dir zeigen, wie man sich einen neuen Arbeitsbereich mit Visual C++ anlegt, das Programm eingibt, kompiliert, linkt und ausführt. Generell ist dieses Buch so geschrieben, dass die darin enthaltenen Quelltexte compilerunabhängig sind, jedoch möchte ich zumindest die wichtigsten Punkte der am meisten verwendeten Compiler abdecken. Solltest Du mit einem anderen als dem hier behandelten Compiler arbeiten, schlage bitte im zugehörigen Handbuch nach oder lies die beigelegten Hilfedateien durch, um zu erfahren, wie Du einen neuen Arbeitsbereich beziehungsweise ein neues Projekt erstellst.

HINWEIS: Um für mehr Übersichtlichkeit zu sorgen, werden die einzelnen Programmzeilen mit Zeilennummern versehen. Diese dürfen nicht mit abgetippt werden, da es sonst zu Fehlern bei der späteren Kompilierung kommt. Natürlich dürfen die Doppelpunkte direkt nach den Zeilennummern auch nicht mit eingegeben werden.

So, nun ist es so weit. Hier kommt der erste Quelltext. Lies hier aber erst weiter, bevor Du Dich dranmachst, das Listing abzutippen. Weiter unten gibt es nämlich noch einige wichtige Erklärungen dazu.

Listing 1.1 Das erste Programm

```
01: // C++ für Spieleprogrammierer
02: // Listing 1.1
03: // Es wird ein Begrüßungstext ausgegeben
04: //
05: #include <iostream>
06:
07: using namespace std;
08:
09: // Hauptprogramm
10: //
11: int main ()
12: {
13:   cout << "Hier kommt die Konkurrenz!\n";
14:   return 0;
15: }
```

Bildschirmausgabe:

```
Hier kommt die Konkurrenz!
```

Tja, das schaut auf den ersten Blick doch ein bisschen verwirrend aus, oder nicht? Im Grunde gibt es hier jedoch nichts, was nicht einem immer wieder gleich ablaufenden Aufbau folgt. Dieses kleine Programm hat eigentlich nur die Aufgabe, einen kurzen Begrüßungstext auf dem Bildschirm auszugeben. Du magst Dich jetzt vielleicht fragen, warum man für eine so kleine Aufgabe denn so viele Zeilen benötigt. Diese Frage ist berechtigt, und es gibt eine recht kurze Antwort darauf: Nur sechs Zeilen dieses Listings sind für den Compiler wirklich wichtig! Die restlichen Zeilen sind sogenannte Kommentare oder Leerzeilen und dienen ausschließlich der Übersichtlichkeit und Lesbarkeit, ohne dabei das eigentliche Programm zu beeinflussen. Was Kommentare sind und wie man sie verwendet, werden wir gleich als Nächstes klären.

1.3.1 Kommentare im Quelltext

Wenn Du Dir den Quelltext genau anschaust, wirst Du feststellen, dass in den Zeilen 1, 2, 3 und 9 Sätze in Klartext stehen. Diese werden beim Kompilieren des Quelltextes einfach ignoriert und haben letztendlich keine Funktion. Der Einzige, den diese Kommentare interessieren, ist derjenige, der mit dem Quelltext arbeitet. In diesem kleinen Beispiel mag es etwas unsinnig erscheinen, alles so genau zu kommentieren. Doch es ist sehr wichtig, sich gleich zu Anfang gewisse Dinge anzugewöhnen. Gerade das vernünftige Kommentieren eines Quelltextes gehört zu den wichtigsten Dingen beim Programmieren überhaupt. Dabei wird man zu Beginn recht häufig denken, dass man dieses oder jenes ja nicht zu kommentieren braucht, da ja alles so wunderbar selbsterklärend ist. Allerdings wird man recht schnell die Erfahrung machen, dass ein Quelltext, den man seit einigen Wochen nicht mehr

angerührt hat, plötzlich die Eigenart aufweist, unverständlich zu erscheinen. Dann ist das Malheur passiert, und man muss sich wieder in Erinnerung rufen, warum man damals etwas so gemacht hat und was man hier und da eigentlich bewirken wollte. Durch sauberes Kommentieren erspart man sich letzten Endes eine Menge Arbeit.

Damit der Compiler „weiß", dass es sich um einen solchen Kommentar handelt, muss man vor den Klartext zwei Slashes stellen (//). Alles, was in dieser Zeile nach den Slashes steht, wird beim Kompilieren des Programms einfach ignoriert. Nun kann es ja auch vorkommen, dass ein Kommentar so ausführlich ist, dass er über mehrere Zeilen geht. Es wäre ja jetzt etwas mühsam, jede Kommentarzeile mit zwei Slashes zu beginnen. Aus diesem Grund gibt es noch eine zweite Möglichkeit, Kommentare zu erstellen. Man stellt einfach zu Beginn eines Kommentarblockes die Zeichenfolge /* voran, schreibt seinen Text und beendet den Block mit der Zeichenfolge */. Und so schaut das Ganze dann aus:

```
01: // Dies wäre eine sehr
02: // umständliche Art, lange
03: // Kommentare zu schreiben,
04: // oder etwa nicht?
01: /*
02: So ist das Ganze
03: doch schon um einiges
04: einfacher, oder?
05: */
```

Diese Art der Kommentierung kann noch auf eine andere Weise nützlich sein. Stell Dir vor, Du hast ein Spiel programmiert (ziemlich coole Vorstellung, oder?). Nun hast Du einen Programmteil, den Du aus irgendeinem Grund von der Kompilierung ausschließen möchtest. Du brauchst aber die betreffenden Programmzeilen nicht zu löschen, sondern Du kannst sie einfach mit der vorangehend gezeigten Methode „auskommentieren".

HINWEIS: Verwende Kommentare großzügig. Sie zu schreiben ist nicht halb so viel Arbeit, wie einen unkommentierten Quelltext mühsam zu entziffern. Versuche dabei jedoch, Dich auf sinnvolle Kommentare zu beschränken und diese aussagekräftig zu halten.

1.3.2 Die #include-Anweisung

Nachdem die Kommentarzeilen nun abgehakt sind, können wir uns den Programmzeilen widmen, die für die eigentliche Funktion des Programms wichtig sind. In der Zeile 5 gibt es auch schon gleich eine kleine Besonderheit, nämlich das Doppelkreuz (#). Dieses Zeichen ist für den Compiler von besonderer Bedeutung. Jeder Befehl, der mit einem solchen Doppelkreuz beginnt, ist ein sogenannter Präprozessor-Befehl, auch Präprozessor-Direktive genannt. Wenn Dein Quelltext kompiliert wird, dann passiert das nicht in einem Rutsch, sondern in mehreren Schritten. Der erste Schritt ist dabei die Behandlung aller Präprozessor-Befehle. Davon gibt es eine ganze Reihe, die wir im Verlauf des Buches noch kennenlernen werden. Diese Befehle haben im Grunde nichts mit dem fertig kompilierten Pro-

gramm zu tun. Die `#include`-Anweisung in der Zeilen 5 dient nun dazu, bereits vorhandene Quelltextdateien zu Deinem Quelltext hinzuzufügen (einzubinden). Die Datei „iostream“ gehört dabei zum Lieferumfang Deines Compilers und enthält alles, was nötig ist, um einfache Textausgaben zu realisieren. Was genau diese Datei beinhaltet, lässt sich aus dem Namen herauslesen. Das „`io`“ steht für „*input/output*“, und „`stream`“ bedeutet übersetzt etwa so viel wie „Strom“, wobei damit der Datenstrom gemeint ist. Dir stehen mit dem Einbinden dieser Datei somit viele Möglichkeiten zur Verfügung, um Datenströme zu verwalten. Das beinhaltet sowohl die Ausgabe von Text auf dem Bildschirm als auch die Eingabe von Text über die Tastatur. Würde man diese Datei nicht einbinden, wüsste der Compiler mit dem Befehl in Zeile 12 nichts anzufangen.

Es gibt eine ganze Reihe dieser Dateien, zum Beispiel für verschiedene Mathematikfunktionen oder auch zum Zeichnen von Grafiken. Im Verlauf des Buches wirst Du auch lernen, wie Du Dir solche Dateien selbst erstellen und verwenden kannst.

Nun noch ein Wort zu den spitzen Klammern (`< >`), in die der Dateiname eingeschlossen ist. Gibt man den Dateinamen in Anführungszeichen ein, dann sucht der Compiler die Datei im aktuellen Arbeitsverzeichnis. Bei den spitzen Klammern hingegen sucht er in einem speziellen Verzeichnis, das in der Regel im Installationsordner des Compilers zu finden ist. Dadurch erspart man es sich, die gewünschte Datei extra ins angelegte Arbeitsverzeichnis kopieren zu müssen.

HINWEIS: Alle Dateien, die mit `#include` eingebunden werden und nicht von uns selbst stammen oder separat installiert wurden, müssen in spitzen Klammern (<>) stehen.

Jetzt noch ein paar Worte zu Zeile 7. In C++ gibt es die Möglichkeit, sogenannte Namensbereiche festzulegen. Dies wird zum Beispiel dann gemacht, wenn man mit mehreren Quellcode-Dateien arbeitet. Da man Funktionen und Variablen Namen geben kann, kann es dabei recht leicht zu Konflikten kommen, wenn zwei verschiedene Dinge aus verschiedenen Dateien den gleichen Namen haben. Gerade dann, wenn mehrere Leute an einem Projekt arbeiten, kann dies zum Problem werden. Aus diesem Grund kann man bestimmte Teile eines Quelltextes in einen Namensbereich gruppieren. Möchte man nun Funktionen oder Variablen aus diesem Namensbereich verwenden, so muss man dies dem Compiler mitteilen. Und genau das tun wir hier in Zeile 7. Wir sagen dem Compiler, dass wir den Namensbereich **std** (was für „Standard“ steht) verwenden möchten. Die in den eingebundenen Dateien enthaltenen Funktionen und Variablen befinden sich allesamt in diesem Namensbereich. Weiter möchte ich an dieser Stelle nicht auf das Thema Namensbereiche eingehen, denn im Augenblick spielen sie noch keine große Rolle für uns. Falls Du dennoch mehr zu Namensbereichen wissen möchtest, dann kannst Du gerne einen Blick in Kapitel 11, Abschnitt 11, werfen.

Es gibt sicherlich noch eine ganze Menge zu den Namensbereichen zu sagen, jedoch müsste man dazu einfach zu weit ausholen. Deshalb belassen wir es hier einfach mal bei der Tatsache, dass diese Zeile benötigt wird.

1.3.3 Die main-Funktion

An dieser Stelle muss ich leider etwas vorgreifen, da Funktionen erst im dritten Kapitel behandelt werden. Um die Zeilen 11 bis 15 verstehen zu können, muss man jedoch wissen, was eine Funktion in etwa ist. Mit C++ hat man die Möglichkeit, häufig verwendete Programmteile in einer sogenannten Funktion zusammenzufassen. Man kann einer solchen Funktion einen Namen geben und sie dann bei Bedarf aufrufen. Ein Beispiel wäre eine Funktion, die eine komplexe Rechenoperation durchführt. Dazu kann man der Funktion Werte (Parameter) übergeben (zum Beispiel zwei Zahlen, mit denen man irgendeine komplexe Berechnung durchführen möchte). Außerdem kann eine Funktion auch einen Wert zurückgeben (in diesem Fall wäre es das Ergebnis der Berechnung).

Nun benötigt jedes C++-Programm zumindest **eine** Funktion, nämlich die `main`-Funktion. Diese dient als Einsprungspunkt des Programms. Das bedeutet einfach, dass beim Programmstart zuerst die `main`-Funktion aufgerufen wird. Deshalb werden wir auch unseren Quellcode erst einmal komplett innerhalb dieser Funktion schreiben. Wenn diese Funktion nicht vorhanden ist, wird es schon beim Kompilieren eine Fehlermeldung geben. Die Schreibweise ist hier ebenfalls sehr wichtig. C++ ist case-sensitive, was bedeutet, dass nach Groß- und Kleinbuchstaben unterschieden wird. Schreibt man etwa `Main`, `mAin` oder `MAIN`, so wird es ebenfalls zu einer Fehlermeldung kommen, da der Compiler die erwartete `main`-Funktion nicht finden kann.

Wie vorangehend bereits erwähnt, kann eine Funktion Werte übernehmen und zurückgeben. Das `int` bedeutet nun, dass die Funktion einen Wert des Typs `int` zurückliefert. Was genau es mit diesem Datentyp auf sich hat, werden wir in Kapitel 2 klären. Bis dahin sei nur gesagt, dass die Funktion dadurch angeben kann, ob alles richtig funktioniert hat. Hat alles ohne Probleme geklappt, liefert die Funktion eine 0 (Null) zurück, andernfalls einen anderen Wert. Dies zu entscheiden liegt damit in unserer Hand. Aber wir sind einfach so dreist und sagen immer, dass alles glattging (indem wir eine 0 zurückliefern).

Diese Rückgabe findet in Zeile 14 statt. Einen Wert (Parameter) gibt man immer mit dem Schlüsselwort `return` zurück. Ich werde später noch etwas genauer darauf eingehen. Wichtig ist allerdings noch, dass `return` nicht nur die Rückgabe eines Parameters bewirkt, sondern auch die Funktion beendet, in der das `return` steht. Würden nun also nach Zeile 13 noch weitere Befehle folgen, so würden diese nicht ausgeführt.

Das Klammernpaar hinter `main` enthält – wie man sieht – keine Werte. Das bedeutet einfach, dass die Funktion keine Parameter (Werte) erwartet. Wenn an dieser Stelle noch etwas unklar ist, was eine Funktion ist und wie sie arbeitet, dann ist das nicht weiter tragisch. Im dritten Kapitel werden wir noch genau genug darauf eingehen.

Bleibt noch zu klären, was es mit den beiden geschweiften Klammern auf sich hat. Jede Funktion besteht aus einem Funktionskopf (in diesem Fall `int main()`) und einem Funktionsrumpf. Im Funktionsrumpf steht der gesamte Code, der zu der Funktion gehört. Dieser wird dabei mit geschweiften Klammern begonnen und beendet (Zeilen 12 und 15).

Dir ist sicherlich aufgefallen, dass die Befehle in den Zeilen 13 und 14 etwas eingerückt sind. Dies hat ausschließlich den Grund, den Quelltext übersichtlicher zu gestalten und lesbarer zu machen. An dieser Stelle wird das noch nicht so deutlich, da es sich nur um ein sehr kleines Programm handelt. Spätestens bei den größeren Beispielen wird jedoch deut-

lich, dass es Sinn macht, den Quellcode an gewissen Stellen einzurücken. Zu welchem Zeitpunkt man wie einrücken sollte, wird später noch genauer erklärt.

HINWEIS: Bei C++ kannst Du so viele Leerzeichen und Leerzeilen verwenden, wie Du möchtest. Dadurch hast Du die Möglichkeit, deinen Code lesbar zu gestalten. Das gilt natürlich nicht innerhalb von Befehlen. Schau Dir andere Quelltexte an, und such Dir einen Aufbau heraus, der für Dich am besten lesbar ist.

1.3.4 „cout“ und einige mögliche Escape-Zeichen

Jetzt kommen wir endlich zu dem „Befehl“, der auch mal etwas auf den Bildschirm bringt. Der Befehl `cout` dient dazu, Text oder Zahlen auf dem Bildschirm auszugeben oder in eine Datei zu schreiben. Schaut man etwas genauer hin, erkennt man, dass das Wort „out“ in diesem Befehl steckt. Das „c“ steht dabei für „*console*“, was für die Textkonsole steht. Im Ganzen heißt das also „*console out*“. Um diesen Befehl verwenden zu können, haben wir ja in der Zeile 5 die Datei **iostream** eingebunden.

Eigentlich hat sich hier ein kleiner Fehler in die Erklärung eingeschlichen, denn `cout` ist in diesem Sinne kein wirklicher Befehl. Um jedoch beim Thema zu bleiben, werden wir `cout` erst einmal wie einen Befehl behandeln.

Die Syntax, die hier verwendet wird, ist recht simpel. Direkt nach `cout` folgt der sogenannte Umleitungsoperator `<<`. Wenn Du nun also schreibst `cout << „Text“;`, dann sagt der Umleitungsoperator aus, dass die Zeichenfolge „`Text`“ direkt an `cout` geleitet werden soll (die Klammern zeigen in Richtung des Befehls). Vorhin habe ich ja etwas über Datenströme erzählt, und genau so einen Datenstrom haben wir hier nun. Es wird ein Text über `cout` auf dem Bildschirm ausgegeben. Dieser Text muss in Anführungszeichen stehen.

Wie Du siehst, steht am Ende dieser Befehlszeile ein Semikolon. Jede Befehlszeile muss mit einem Semikolon abgeschlossen werden, damit der Compiler erkennt, dass die Befehlszeile zu Ende ist. Bei Präprozessor-Direktiven (also alles, was mit # beginnt) und Funktionsköpfen darf allerdings kein Semikolon stehen. Solltest Du mal vergessen, das Semikolon zu setzen (und das wird garantiert passieren, es passiert einfach **jedem**), dann wird Dich der Compiler höflich, aber bestimmt darauf hinweisen, dass etwas nicht stimmt.

HINWEIS: Jede Befehlszeile muss mit einem Semikolon abgeschlossen werden. Dies gilt jedoch nicht für Präprozessor-Direktiven und Funktionsköpfe.

So, was ist das jetzt für eine seltsame Zeichenfolge am Ende des Textes in Zeile 13? Die Zeichenfolge `\n` ist ein sogenanntes Escape-Zeichen. Diese Escape-Zeichen werden immer dann verwendet, wenn man etwas ausgeben möchte, was kein darstellbares Zeichen ist. Dazu gehören zum Beispiel Zeilenumbrüche, Tabulatoren oder Backspace. Die Zeichenfolge `\n` bewirkt hier ein sogenanntes Carriage Return, was nichts anderes ist als ein Zeilenumbruch.

Man kann einen Zeilenumbruch auch noch auf eine andere Weise erzeugen, und zwar durch den sogenannten Manipulator `endl`. Dabei ist `endl` die Kurzform von „end of line“, also im Deutschen „Ende der Zeile“. Zeile 13 könnte man folglich auch so schreiben (wie man sieht, kann der Umleitungsoperator `<<` auch mehrfach verwendet werden):

```
cout << "Hier kommt die Konkurrenz! " << endl;
```

Es gibt eine ganze Menge solcher Escape-Zeichen. Gerade wenn es darum geht, Text zu formatieren, sind sie unerlässlich. Die wichtigsten von ihnen findest Du in der folgenden Tabelle. Wenn Du nachher den Quellcode kompilierst, teste einfach einige von ihnen, um ihre Wirkung zu sehen.

Tabelle 1.1 Escape-Zeichen

Escape-Zeichen	Bedeutung
\n	Carriage Return (Zeilenumbruch)
\t	Horizontaler Tabulator
\v	Vertikaler Tabulator
\r	Zum Zeilenanfang zurückkehren
\b	Ein Zeichen zurück (Backspace)
\“	Anführungszeichen einfügen
\\	Backslash einfügen
\a	Beep (Signalton)

1.4 Die Entwicklungsumgebung Visual Studio 2015 Community Edition

Im Folgenden werde ich Dir zeigen, wie man sich mit Visual Studio 2015 Community Edition einen neuen Arbeitsbereich erstellt, das Programm eingibt, kompiliert und ausführt. Die entsprechende Erklärung für die Verwendung von Xcode 6.4 findest Du ab Abschnitt 1.5.

Solltest Du eine andere Entwicklungsumgebung verwenden, schlage bitte im zugehörigen Handbuch nach, um zu erfahren, wie man einen Arbeitsbereich beziehungsweise ein Projekt erstellt.

1.4.1 Anlegen eines neuen Projektes

Starte Microsoft Visual Studio 2015 Community Edition, wähle im Menü den Punkt DATEI und klicke dann auf NEU → PROJEKT... Im nun erscheinenden Fenster wählst Du zunächst in der linken Liste (*Projekttypen*) den Punkt VORLAGEN → VISUAL C++ und dort den Unterpunkt WIN32. Im mittleren Teil des Fensters wählst Du den Punkt WIN32-KONSOLENANWENDUNG.

Nachdem Du diese Einstellungen vorgenommen hast, kannst Du dem Projekt einen Namen geben. Tippe dazu im Feld *„Name“* den gewünschten Projektnamen ein (beispielsweise *„Listing_1“*. Im Eingabefeld direkt darunter wird der Speicherort für das neue Projekt gewählt. Wo Du Deine künftigen Projekte speicherst, bleibt Dir selbst überlassen. Allerdings solltest Du darauf achten, dass Du als Speicherort immer den gleichen Ordner wählst, damit Deine Projekte nicht quer auf der Festplatte verteilt sind. Ich habe es mir angewöhnt, direkt im Root-Verzeichnis ein Verzeichnis namens *Projekte* anzulegen, in das sämtliche Projekte wandern. Selbstverständlich sorgt Microsoft Visual Studio 2015 Community Edition automatisch dafür, dass für jedes Projekt ein neuer Unterordner angelegt wird.

Deaktiviere nun noch das Häkchen bei PROJEKTMAPPENVERZEICHNIS ERSTELLEN, damit nicht noch ein zusätzlicher zweiter Ordner gleichen Namens angelegt wird, und bestätige Deine Einstellungen mit dem Button OK. Sollte ein Häkchen bei ZUR QUELLCODEVERWALTUNG HINZUFÜGEN gesetzt sein, so deaktiviere auch dieses.

Wähle nun im nächsten Fenster den Menüpunkt ANWENDUNGSEINSTELLUNGEN auf der linken Seite. Achte darauf, dass in der Kategorie ANWENDUNGSTYP der Punkt KONSOLENANWENDUNG angewählt ist. Nun solltest Du noch in der Kategorie ZUSÄTZLICHE OPTIONEN das Häkchen LEERES PROJEKT aktivieren und das Häkchen SECURITY DEVELOPMENT LIFECYCLE (SDL)-PRÜFUNGEN deaktivieren. Dies sorgt dafür, dass wir nur das absolute Grundgerüst eines neuen Projekts angelegt bekommen, ohne dass weitere für uns erst einmal uninteressante Dateien erzeugt werden. Durch einen Klick auf FERTIGSTELLEN wird nun das vorhin gewählte Verzeichnis angelegt, und alle für das Projekt notwendigen Dateien werden erzeugt. Diese beinhalten unter anderem die gewählten Projekteigenschaften und weitere von der Entwicklungsumgebung benötigten Informationen. Diese Dateien sollten nicht außerhalb von Visual Studio verändert werden. Bild 1.1 und Bild 1.2 zeigen noch einmal alle nötigen Einstellungen

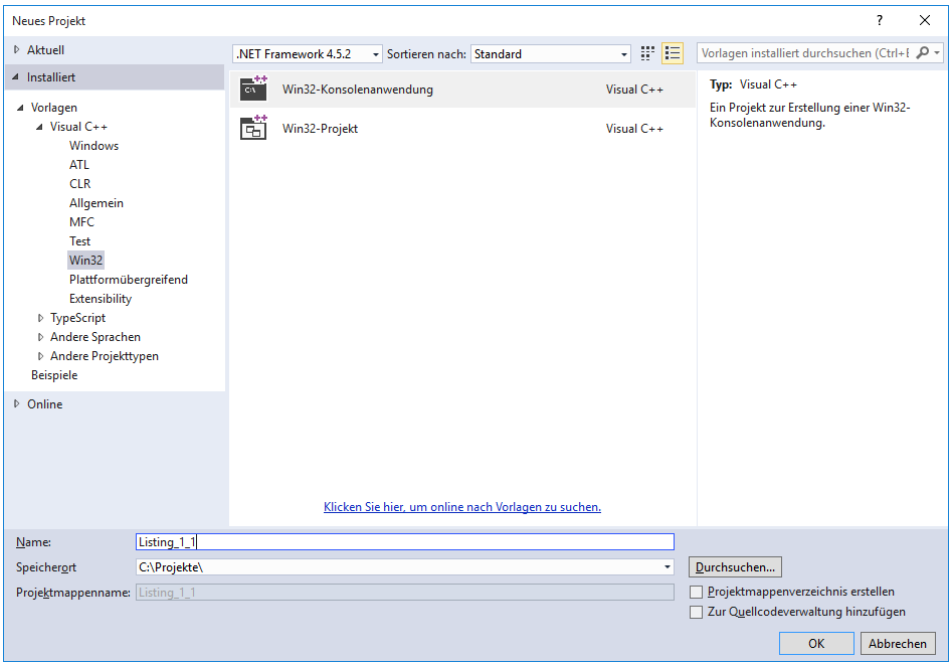

Bild 1.1 Anlegen eines neuen Projektes in Visual Studio 2015

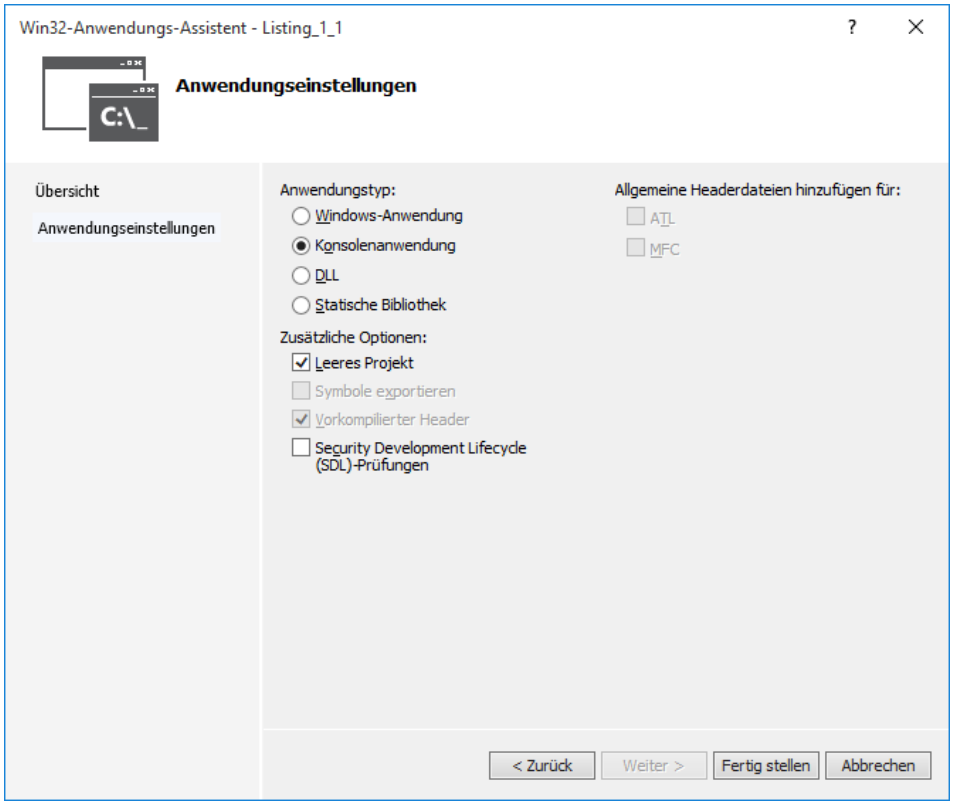

Bild 1.2 Anwendungseinstellungen des Projektes in Visual Studio 2015

Was jetzt noch fehlt, ist die eigentliche Quellcodedatei, in der später der Quellcode stehen wird. Klicke dazu im Projektmappen-Explorer mit der rechten Maustaste auf den Ordner QUELLDATEIEN und wähle den Punkt HINZUFÜGEN → NEUES ELEMENT. Es erscheint wieder ein neues Fenster, in dessen linkem Auswahlmenü (*Kategorien*) der Punkt CODE ausgewählt sein muss. Wähle nun im mittleren Bereich den Punkt C++-DATEI(.CPP) und gib im Eingabefeld nun als Dateinamen beispielsweise *Listing_1* ein. Den im zweiten Eingabefeld gewählten Pfad solltest Du nicht mehr ändern, damit alles schön seine Ordnung behält. Klicke nun auf den Button HINZUFÜGEN, damit die Quellcodedatei erzeugt und zum Projekt hinzugefügt wird. Bild 1.3 zeigt das noch einmal im Detail.

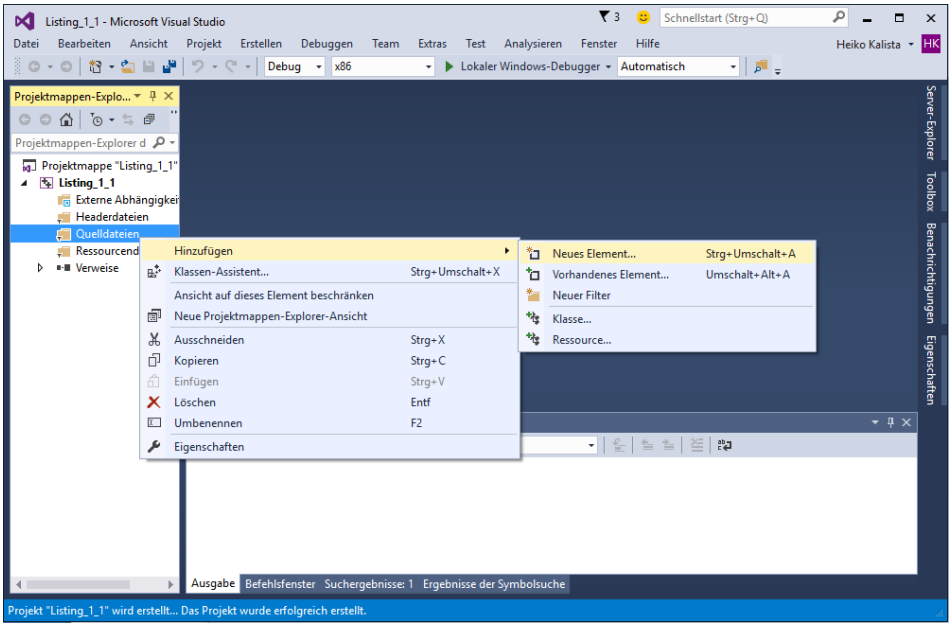

Bild 1.3 Hinzufügen einer neuen Quellcode-Datei in Visual Studio 2015

Damit sind alle nötigen Vorbereitungen für unser erstes eigenes Programm getroffen, und Du kannst gleich damit loslegen, den Quelltext abzutippen. Das funktioniert folgendermaßen:

1.4.2 Das Programm mithilfe des Quelltexteditors eingeben

Jetzt ist es so weit: Du kannst nun das Beispiel Zeile für Zeile abtippen. Denk bitte daran, dass Du die Zeilennummern nicht mit eingeben darfst. Der Quelltexteditor lässt sich im Grunde genauso einfach bedienen wie ein gewöhnliches Textverarbeitungsprogramm, weshalb es relativ wenige Schwierigkeiten beim Eingeben des Quellcodes geben sollte.

Was allerdings etwas anders im Gegensatz zu einem „normalen“ Textverarbeitungsprogramm ist, ist die Tatsache, dass gewisse Textabschnitte automatisch eingefärbt werden. Kommentare werden automatisch grün, Schlüsselwörter (feste Bestandteile der Programmiersprache C++) blau eingefärbt. Das sorgt nicht nur für mehr Übersichtlichkeit, sondern es hilft auch dabei, Fehler zu vermeiden. Wenn Du einen Kommentar oder ein Schlüsselwort eingibst und es sich nicht einfärbt, so hast Du definitiv etwas falsch gemacht. So sieht man auf den ersten Blick, ob etwas nicht stimmt, und muss nicht erst warten, bis man beim Kompilieren eine Fehlermeldung erhält.

Wenn Du die Klammer in Zeile 9 eingegeben und Return gedrückt hast, wirst Du feststellen, dass die nächste Zeile automatisch eingerückt wird. Dies ist auch ein Feature, das Dir der Quelltexteditor bietet. Sobald man eine öffnende, geschweifte Klammer eingibt, werden die folgenden Codezeilen automatisch um eine Tabulatorbreite eingerückt. Wenn man die

Klammer schließt, geht der Cursor in der nächsten Zeile um eine Tabulatorbreite zurück. Solltest Du eigene Einrückungen bevorzugen, dann erzeuge diese mit der Tab-Taste und nicht mit Leerzeichen, denn das spart Arbeit.

In einigen Dingen mag sich der Quelltexteditor von normalen Textverarbeitungsprogrammen unterscheiden, jedoch gewöhnt man sich recht schnell daran und lernt die zusätzlichen Features zu schätzen. Grundsätzliche Dinge wie Suchen und Ersetzen funktionieren ähnlich wie bei herkömmlichen Programmen.

Nimm Dir einfach etwas Zeit und experimentiere ein wenig mit dieser neuen Umgebung. Versuche das Beispiel genau so einzugeben, wie in Listing 1.1 dargestellt.

1.4.3 Laden der Programmbeispiele

Zum Begleitmaterial dieses Buches, das Du unter *http://downloads.hanser.de* herunterladen kannst, gehören selbstverständlich sämtliche Programmbeispiele, damit Du nicht alles von Hand abtippen musst. Trotzdem ist es ratsam, wenigstens die ersten Quelltexte von Hand einzugeben und den Arbeitsbereich selbst zu erstellen, um etwas Übung im Umgang mit der Arbeitsumgebung zu gewinnen.

Zu jedem Beispiel gibt es einen separaten Ordner, in dem sich sämtliche benötigten Dateien befinden. Dies sind die *.cpp*- und *.hpp*-Dateien, die den eigentlichen Quelltext enthalten. Weiterhin findest Du pro Beispiel eine *.sln*-Datei, die den Arbeitsbereich für unsere Entwicklungsumgebung darstellt. Diese Dateiendung ist mit Visual Studio 2015 Community Edition verknüpft und kann somit durch einen Doppelklick geöffnet werden. Alternativ kannst Du auch im Menü den Punkt DATEI wählen und dort über ÖFFNEN → PROJEKT/PROJEKTMAPPE… zur entsprechenden *.sln*-Datei navigieren und diese öffnen.

Es kann sein, dass Du nach dem Öffnen des Projekts nur ein graues Fenster, nicht aber den gewünschten Quellcode siehst. In diesem Fall musst Du rechts im Projektmappen-Explorer zur entsprechenden Quellcode-Datei navigieren, indem Du einfach die Reiter der Reihe nach aufklappst.

Solltest Du eine andere Entwicklungsumgebung verwenden, kannst Du zuerst einmal versuchen, ob sich die *.sln*-Datei damit öffnen und gegebenenfalls automatisch konvertieren lässt. Ist dies nicht der Fall, dann schaue in der Dokumentation Deiner Entwicklungsumgebung nach, wie sich ein Arbeitsbereich einrichten lässt und wie Du Quellcode-Dateien hinzufügen kannst.

1.4.4 Das Programm kompilieren und linken

Wie entsteht nun aus einem Quelltext ein lauffähiges Programm? Dazu sind einige Schritte notwendig, um die wir uns zum Glück nicht selbst kümmern müssen. Alles, was dazu notwendig ist, um aus einem Quelltext ein lauffähiges Programm zu erzeugen, erledigt die Entwicklungsumgebung für uns. Doch vom Quellcode bis zur ausführbaren Datei ist es für unsere Entwicklungsumgebung ein langer Weg. Zuerst einmal werden sämtliche Präprozessor-Direktiven abgearbeitet. Es entsteht eine temporäre Quelltextdatei, in der sämtlicher

Quellcode (also auch die eingebundenen Dateien) vertreten ist. Diese wird dann kompiliert, und es entstehen eine oder mehrere Objektdateien. Beim Kompilieren wird der Quelltext in eine für den Computer verständliche Sprache umgewandelt. Welche das ist, kann man leicht erraten, wenn man vorhin alle Abschnitte gelesen und nicht übersprungen hat: Assembler. Diese Objektdateien sind natürlich noch keine ausführbaren Programme. Vielmehr handelt es sich um „Bruchstücke", die noch miteinander verbunden werden müssen. Diese Aufgabe erledigt der Linker. Er führt sozusagen sämtliche Bruchstücke zu der endgültigen ausführbaren Datei zusammen (.exe). Es werden gegebenenfalls diverse Libraries (Bibliotheksdateien) mit eingebunden. Diese Bibliotheksdateien stellen Sammlungen von Funktionen dar, die zum Teil dem Compiler beiliegen oder auch speziell von diversen Anbietern aus dem Internet heruntergeladen werden können. Es gibt zum Beispiel Bibliotheksdateien zum Abspielen von MP3-Dateien, die fertige Funktionen zum einfachen Gebrauch zur Verfügung stellen und vieles mehr.

Bei den hier im Buch vorgestellten Entwicklungsumgebungen braucht man sich keine Gedanken über diese einzelnen Schritte zu machen, da alles für uns erledigt wird. Durch Anwählen eines Menüpunktes oder durch Drücken eines Shortcuts (Tastenkürzel) wird aus unserem Quelltext ein ausführbares Programm (solange der Quelltext keine schweren Fehler enthält, die dies verhindern).

Du solltest jetzt einen Arbeitsbereich und den Quelltext aus dem ersten Beispiel vor Dir haben. Wie man darauf ein lauffähiges Programm erzeugt, wird nun erklärt.

Schau Dir mal im Menü den Punkt ERSTELLEN an. Dort findest Du gleich mehrere Punkte, die für uns interessant sind. Wie so oft, kommt man auch hier durch mehrere Wege ans Ziel. Man kann selbst bestimmen, ob zuerst nur kompiliert werden soll und dann die ausführbare Datei separat erstellt wird. Alternativ kann man auch alles auf einmal erstellen lassen (kompilieren und linken in einem Rutsch).

Fürs Erste wählen wir einfach den Punkt ERSTELLEN → PROJEKTMAPPE NEU ERSTELLEN. Nun beginnt der Compiler seine Arbeit. Wenn dieser fehlerlos die Objektdateien erstellen konnte, tritt der Linker in Aktion und erstellt endlich die lang ersehnte *.exe*-Datei.

HINWEIS: Wie der Name schon sagt, bewirkt ERSTELLEN > PROJEKTMAPPE NEU ERSTELLEN, dass sämtliche Quelltextdateien neu kompiliert werden, selbst wenn keine Änderungen darin vorgenommen wurden. Das mag bei unserem ersten Beispiel noch nicht dramatisch sein, jedoch wird es bei großen Projekten sehr lange dauern und unnötig Zeit rauben. Deshalb sollte man nur die Dateien kompilieren lassen, die man auch tatsächlich verändert hat. Dies geschieht über den Menüpunkt ERSTELLEN > PROJEKTMAPPE ERSTELLEN. Eine komplette Neuerstellung ist in der Regel nur selten nötig – etwa dann, wenn beim Kompiliervorgang etwas schief ging.

Möchte man diese Schritte einzeln anschauen, so kann man auch zuerst auf ERSTELLEN > KOMPILIEREN klicken (alternativ Strg+F7 drücken) und danach ERSTELLEN > NUR PROJEKT > NUR 'LISTING_1' ERSTELLEN wählen.

Wenn es nichts zu beanstanden gab und Du alle Schritte richtig ausgeführt hast, sollte unten im Ausgabefenster folgender Text stehen:

```
------ Neues Erstellen gestartet: Projekt: 1_1, Konfiguration: Debug Win32 ------
  Listing_1_1.cpp
  1_1.vcxproj -> C:\Projekte\Listing_1_1\Debug\1_1.exe
========= Alles neu erstellen: 1 erfolgreich, 0 fehlerhaft, 0 übersprungen ==========
```

Beim Neuerstellen werden sämtliche temporäre Dateien (etwa die Objektdateien und die ausführbare *.exe*-Datei) gelöscht. Diesen Schritt kann man auch durch den Menüpunkt ERSTELLEN > PROJEKTMAPPE BEREINIGEN manuell herbeiführen. Das macht etwa dann Sinn, wenn beim Kompilieren etwas unerwartet schiefgeht. Manchmal liegt es daran, dass veraltete Objektdateien verwendet werden. Durch Auswählen dieses Menüpunktes wird alles gelöscht und muss dann zwingend neu erstellt werden.

Nun wird angezeigt, dass alle im Projekt enthaltenen Quellcode-Dateien kompiliert werden. Bei uns ist das erst mal nur *Listing_1.cpp*. Sollte dabei ein Fehler auftauchen, würde man das direkt in der Ausgabe sehen.

Wenn alles geklappt hat, tritt direkt im Anschluss der Linker in Aktion. Auch hier sind Fehler möglich, selbst wenn der gesamte Quellcode ohne Probleme kompiliert werden konnte.

Die letzte Zeile ist nun wohl die wichtigste. Hier bekommt man zusammenfassend angezeigt, ob es Fehler oder Warnungen gab oder nicht. Bei einem Fehler wird das Erstellen unterbrochen, bei Warnungen hingegen nicht. (Man kann allerdings einstellen, dass Warnungen wie Fehler behandelt werden. In diesem Fall würde dann ebenfalls abgebrochen werden). Auf Warnungen kommen wir im nächsten Kapitel noch zu sprechen.

Wenn Deine Ausgabe so ausschaut wie vorangehend gezeigt, dann ging alles glatt, und Du hast Deine erste ausführbare Datei erstellt. Diese kannst Du durch Drücken von STRG+F5 starten.

1.4.5 Ausführen des Programms

Man kann das Programm immer direkt nach dem Kompilieren aus der Entwicklungsumgebung heraus starten, ohne die Datei extra aufrufen zu müssen. Dies bietet uns zumindest bei den Microsoft-Compilern einen Vorteil: Wenn man eine Konsolenanwendung aus einer Microsoft-Entwicklungsumgebung heraus aufruft, so wird automatisch nach Beenden des Programms eine „Press any key to continue“-Meldung im Konsolenfenster erzeugt. Diese Meldung bewirkt, dass das Konsolenfenster erst dann geschlossen wird, wenn man eine beliebige Taste drückt. Ruft man die erstellte .exe-Datei außerhalb des Compilers auf, so erscheint diese Meldung nicht. Das hat zur Folge, dass man bei der Ausführung des ersten Beispiels nichts weiter sieht als ein kurz aufblitzendes Fenster. Warum das so ist, lässt sich leicht erklären: Das Programm wird ja stur von Anfang bis Ende abgearbeitet. Nachdem nun unser kleiner Text ausgegeben wurde, folgt kein weiterer Befehl, und das Programm wird beendet. Somit wird das Konsolenfenster so schnell geschlossen, dass man nur ein kurzes Aufblitzen bemerkt. Somit muss in jedem Fall als Letztes etwas geschehen, was eine Tastatureingabe erfordert, damit das Programm weiterhin aktiv bleibt.

Wie man dieses kleine Problem umgeht, werden wir klären, wenn der `cin`-Befehl an die Reihe kommt. Bis dahin solltest Du die Beispiele direkt aus der Entwicklungsumgebung heraus starten. Achte hier allerdings darauf, dass Du das Programm mit Strg+F5 oder dem Menüpunkt DEBUGGEN > STARTEN OHNE DEBUGGING startest. Nur dann ist gewährleistet, dass beim Programmende noch auf eine Tastatureingabe gewartet wird.

Wenn Du das Programm startest, solltest Du ein typisches DOS-Fenster mit schwarzem Hintergrund sehen, auf dem folgende Ausgabe zu lesen ist:

```
Hier kommt die Konkurrenz!
Press any key to continue
```

Okay, zugegeben: Das ist nicht gerade das, was man spektakulär nennen würde. Trotzdem wird mit diesem kleinen Beispiel sozusagen ein Grundstein gelegt. Man ist nun in der Lage, ein eigenes Programm zu schreiben, und sei es noch so klein und unscheinbar. Fast alle, die sich mit dem Programmieren von Spielen oder Anwendungsprogrammen beschäftigen, haben einmal mit einem solchen kleinen Programm angefangen. Text auf dem Bildschirm auszugeben ist zu Beginn das Wesentlichste, was man beherrschen muss. Wie Du sehen wirst, ist diese Voraussetzung die Grundlage, die in den folgenden Kapiteln unerlässlich ist.

Jetzt ist es an der Zeit, etwas mit dem Beispiel herumzuexperimentieren. Füge weitere Zeilen hinzu, in denen mehr Text ausgegeben wird, und mache rege von den Escape-Zeichen aus Tabelle 1.1 Gebrauch. Denk daran, dass jede Zeile mit einem Semikolon abgeschlossen werden muss, um Fehler zu vermeiden.

HINWEIS: Es ist sehr wichtig, immer mit den Beispielen zu experimentieren. Einfach nur die Listings zu kompilieren und sich das Resultat anzuschauen, genügt eben nicht. Versuche den Quelltext etwas abzuändern, ohne jedoch da bei dem Thema vorzugreifen. Dadurch wird man schneller mit den erklärten Themen vertraut, und der Lerneffekt ist größer.

1.5 Die Entwicklungsumgebung Xcode

Wenn Du auf einem Macintosh arbeitest und Xcode6.4 oder höher verwendest, dann beachte bitte, dass die Beispiele 10.1 und 10.3 in Kapitel 10 nicht lauffähig sind. Dies liegt daran, dass diese Beispiele Windows-spezifische Funktionen verwenden.

Genau wie Microsoft Visual Studio 2015 Community Edition bietet auch Xcode eine Fülle mächtiger Funktionen und Einstellungen. Es versteht sich von selbst, dass ich nicht auf alle Features im Detail eingehen kann, da dies den Rahmen dieses Buches sprengen würde.

Wie jede andere Software wird auch Xcode ständig weiterentwickelt. Somit kann sich die Bedienung nach einem Update natürlich von der hier im Buch erläuterten Vorgehensweise unterscheiden. Allerdings gehören Veränderungen eben zum täglichen Brot eines Entwicklers, weshalb man sich immer mit den neuesten Features seiner gewählten Entwicklungsumgebung auseinander setzen sollte.

Beachte bitte, dass Xcode nur in einer englischen Fassung vorliegt. Man kann die Entwicklungsumgebung nicht auf Deutsch starten.

Natürlich gibt es auch auf Macintosh-Computern noch andere Entwicklungsumgebungen, mit denen Du arbeiten kannst. Es sollte jedoch klar sein, dass diese nicht alle in einem Buch vorgestellt werden können. Solltest Du eine Alternative zu Xcode suchen, dann schau Dich ein wenig im Internet um. Dort wirst Du viele Möglichkeiten finden.

1.5.1 Anlegen eines neuen Projekts

Starte Xcode und wähle in der linken Hälfte des Startfensters CREATE A NEW XCODE PROJECT. Wird das Startfenster nicht angezeigt, dann klicke im Menü auf FILE→NEW→PROJECT... Im daraufhin erscheinenden Fenster wählst Du ebenfalls in der linken Hälfte den Punkt APPLICATION innerhalb der Kategorie OS X. Im rechten Teil des Fensters stehen nun mehrere Projekt-Typen zur Verfügung. Wähle COMMAND LINE TOOL und klicke auf NEXT. Bild 1.4 zeigt die Einstellungen noch einmal.

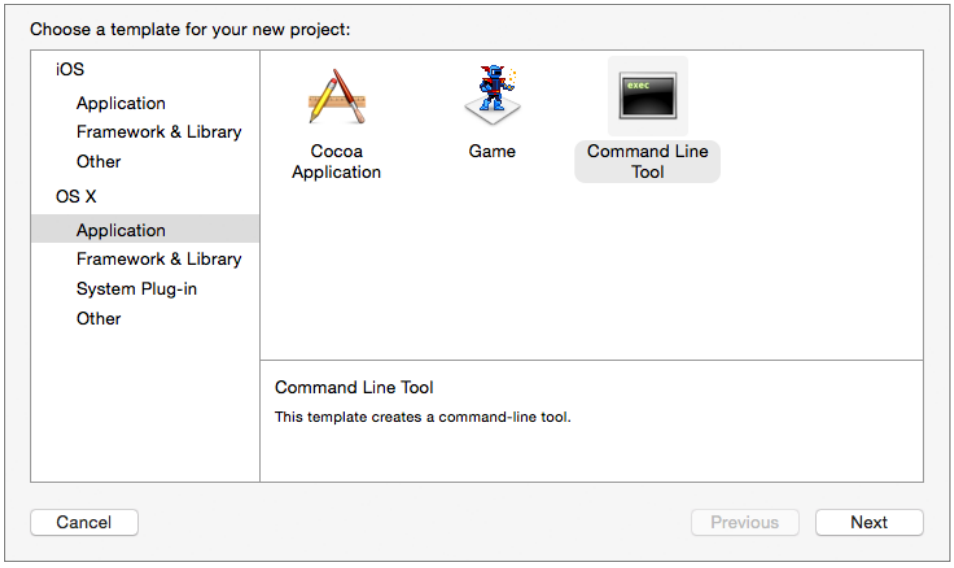

Bild 1.4 Anlegen eines neuen Projektes in Xcode 6.4

Nachdem nun ausgewählt ist, welcher Projekt-Typ angelegt werden soll, kannst Du dem Projekt einen Namen geben. Gib dazu im Feld PRODUCT NAME den gewünschten Projektnamen ein (beispielsweise „*Listing_1_1*“). In den Feldern ORGANIZATION NAME und ORGANIZATION IDENTIFIER kann man beispielweise den eigenen Namen und einen Firmennamen eingeben. Sobald später Quellcode-Dateien erzeugt werden, fügt Xcode diese Informationen automatisch den Kommentaren hinzu. Für unsere Zwecke sind diese Felder nebensächlich und können nach Belieben ausgefüllt werden. Wähle nun noch im Dropdown-Feld LANGUAGE den Eintrag C++ und bestätige Deine Einstellungen durch einen Klick auf NEXT. Deine Einstellungen sollten nun denen in Bild 1.5 entsprechen.

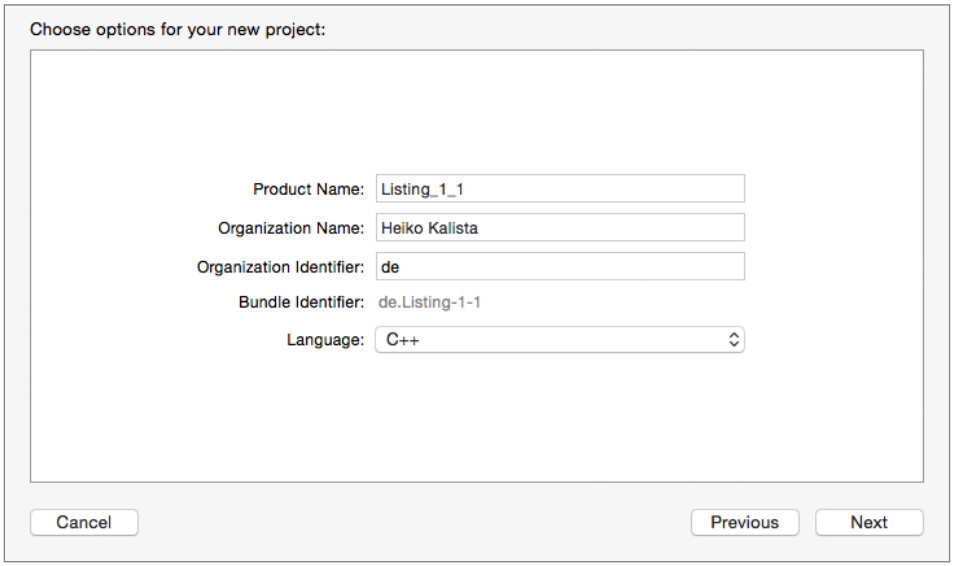

Bild 1.5 Optionen für das neue Projekt in Xcode 6.4

Im nächsten Fenster kannst Du den Speicherort Deines Projekts wählen. Wo Du Deine künftigen Projekte speicherst, bleibt Dir selbst überlassen. Allerdings solltest Du darauf achten, dass Du als Speicherort immer den gleichen Ordner wählst, damit Deine Projekte nicht quer auf der Festplatte verteilt sind. Es ist sinnvoll, sich einen Ordner namens *Projekte* anzulegen und in diesem alle neuen Projekte in separaten Ordnern zu speichern. So wahrt man eine gewisse Ordnung und findet seine Projekte schnell wieder.

Bevor Du durch einen Klick von CREATE das Projekt anlegst, solltest Du noch den Haken bei CREATE GIT REPOSITORY entfernen, falls dieser gesetzt ist. Diese Option kann zwar im Prinzip auch angewählt bleiben, ist für uns aber noch unwichtig. Wenn Du die Option aktivierst, wird Dein Quellcode unter eine Versionsverwaltung, ein sogenanntes Git Repository gestellt. Im Grunde bedeutet das, dass man nach Änderungen am Quellcode „Schnappschüsse" seines Quellcodes erzeugen kann. Man kann zu alten Versionen zurückkehren, alle Änderungen nachvollziehen und so weiter. Da dies jedoch ein Thema für sich ist, werde ich hier auf eine weitere Erklärung verzichten. Für unsere Zwecke ist eine solche Versionsverwaltung noch etwas zu viel des Guten. Allerdings schadet es nicht, wenn Du Dich im Web nach diesem Thema erkundigst und Dich mit der Materie befasst. Es gibt sehr viele gute Erklärungen bezüglich des Funktionsprinzips und Anleitungen, wie man dieses Feature benutzt.

Sind alle Schritte erledigt, legt Xcode ein neues Projektverzeichnis mit allen zugehörigen Dateien an. Diese beinhalten unter anderem die gewählten Projekteigenschaften und weitere von der Entwicklungsumgebung benötigten Informationen. Diese Dateien sollten nicht von Hand verändert werden.

1.5.2 Hinzufügen und Erstellen von neuen Dateien

Xcode erstellt für ein neues C++-Projekt automatisch eine Datei namens *main.cpp*. Es spricht nichts dagegen, diese Datei als Ausgangspunkt zu verwenden und dort die jeweiligen Quellcode-Beispiele einzugeben. Bei den herunterladbaren Beispielprojekten wird jedoch statt *main.cpp* in der Regel der Name des Beispiels verwendet, beispielsweise *Listing_1_1.cpp*. Welche Namensgebung Du bevorzugst, kannst Du natürlich selbst entscheiden. Damit das auch reibungslos klappt, möchte ich Dir kurz zeigen, wie Du neue und/oder bereits vorhandene Quellcode-Dateien zum Projekt hinzufügen kannst.

Um eine neue Quellcode-Datei zum Projekt hinzuzufügen, klickst Du mit der rechten Maustaste auf den Ordner LISTING_1_1 im Projekt-Navigator (Ordner-Symbol in der Symbolleiste oben links). Im darauf erscheinenden Kontextmenü wählst Du NEW FILE… und selektierst im darauf folgenden Dialog OS X und SOURCE in der linken Auswahlbox. Im rechten Teil des Fensters erscheinen nun verschiedene Dateitypen, von denen Du C++ FILE wählst. Durch einen Klick auf NEXT gelangst Du zum letzten Schritt. Gib den von Dir gewünschten Dateinamen ein und deaktiviere das Häkchen bei ALSO CREATE A HEADER FILE. Klicke nun noch auf NEXT und belasse im nächsten Dialog einfach alle Einstellungen so, wie sie sind. Ein Klick auf CREATE erzeugt nun die neue, leere .cpp-Datei.

Möchtest Du Deinem Projekt eine bereits vorhandene Quellcode-Datei hinzufügen, dann klicke ebenfalls mit der rechten Maustaste auf den Ordner LISTING_1_1 und wähle dieses Mal im Kontextmenü ADD FILES TO „LISTING_1_1“… Im darauf folgenden Dialog kannst du die gewünschte Datei auswählen und zum Projekt hinzufügen. Falls sich die Datei außerhalb Deines Projekts befindet, kannst Du diese durch das Aktivieren des Häkchens COPY ITEMS IF NEEDED automatisch in Dein Projektverzeichnis kopieren lassen.

Soll eine Datei gelöscht werden, dann klicke die betreffende Datei mit der rechten Maustaste an und wähle DELETE. Daraufhin kannst Du entscheiden, ob nur die Referenz (REMOVE REFERENCE) oder die tatsächliche Datei gelöscht werden soll (MOVE TO TRASH). Ersteres entfernt nur den Verweis im Projekt. Das bedeutet, dass die Datei im Ordner verbleibt, jedoch beim Kompilieren nicht mehr beachtet wird. Letzteres schickt nicht nur den Verweis, sondern auch die Datei selbst ins Nichts.

1.5.3 Das Programm mithilfe des Quelltexteditors eingeben

Falls Du bereits Erfahrung mit anderen Entwicklungsumgebungen hast, so wird Dir der Quelltexteditor von Xcode sicher keine Schwierigkeiten bereiten. Denke bitte daran, dass Du die Zeilennummern nicht mit eingeben darfst. Der Quelltexteditor lässt sich im Grunde genauso einfach bedienen wie ein gewöhnliches Textverarbeitungsprogramm, weshalb es relativ wenige Probleme beim Eingeben des Quellcodes geben sollte.

Was allerdings etwas anders im Gegensatz zu einem „normalen“ Textverarbeitungsprogramm ist, ist die Tatsache, dass gewisse Textabschnitte automatisch eingefärbt werden. Kommentare werden automatisch Grün, Schlüsselwörter (feste Bestandteile der Programmiersprache C++) Magenta eingefärbt. Das sorgt nicht nur für mehr Übersichtlichkeit, sondern es hilft auch dabei, Fehler zu vermeiden. Wenn Du einen Kommentar oder ein Schlüsselwort eingibst und es sich nicht einfärbt, so hast Du definitiv etwas falsch gemacht. So

siehst Du auf den ersten Blick, ob etwas nicht stimmt, und musst nicht erst warten, bis Du beim Kompilieren eine Fehlermeldung erhältst.

Wenn Du die Klammer in Zeile 9 eingegeben und Return gedrückt hast, wirst Du feststellen, dass die nächste Zeile automatisch eingerückt und auch die zugehörige, schließende Klammer eingefügt wird. Dies ist auch ein Feature, das Dir der Quelltexteditor bietet. Sobald man eine öffnende, geschweifte Klammer eingibt, werden die folgenden Codezeilen automatisch um eine Tabulatorbreite eingerückt. Solltest Du eigene Einrückungen bevorzugen, dann erzeuge diese mit der Tab-Taste und nicht mit Leerzeichen, denn das spart Arbeit.

In einigen Dingen mag sich der Quelltexteditor von normalen Textverarbeitungsprogrammen unterscheiden, jedoch gewöhnt man sich recht schnell daran und lernt die zusätzlichen Features zu schätzen. Grundsätzliche Dinge wie Suchen und Ersetzen funktionieren ähnlich wie bei herkömmlichen Programmen.

Nimm Dir einfach etwas Zeit und experimentiere ein wenig mit dieser neuen Umgebung. Versuche das Beispiel genau so einzugeben, wie in Listing 1.1 dargestellt.

1.5.4 Laden der Programmbeispiele

Alle Programmbeispiele, die mit Xcode respektive Mac OS X kompatibel sind, befinden sich selbstverständlich als fertige Projekte in dem zum Buch gehörenden Begleitmaterial, damit man nicht alles von Hand abtippen muss. Trotzdem ist es ratsam, wenigstens die ersten Quelltexte von Hand einzugeben und die Projekte selbst zu erstellen, um etwas Übung im Umgang mit der Arbeitsumgebung zu gewinnen.

Für jedes Beispiel im Buch findest Du jeweils in einem Unterordner sämtliche benötigten Dateien. Dies sind die *.cpp*- und *.hpp*-Dateien, die den eigentlichen Quelltext enthalten. Weiterhin findest Du pro Beispiel eine Datei mit der Endung *.xcodeproj*, die die Projektdatei für unsere Entwicklungsumgebung darstellt. Diese Dateiendung ist mit Xcode verknüpft und kann somit durch einen Doppelklick geöffnet werden. Alternativ kannst Du auch im Menü den Punkt FILE wählen und dort über OPEN... zum entsprechenden Projekt browsen. Dabei genügt es schon, den obersten Ordner zu wählen, Du musst nicht extra die Projektdatei suchen. Ein Klick auf OPEN öffnet das gewünschte Projekt.

1.5.5 Das Programm kompilieren und linken

Wie entsteht nun aus einem Quelltext ein lauffähiges Programm? Dazu sind einige Schritte notwendig, um die wir uns zum Glück nicht selbst kümmern müssen. Alles, was notwendig ist, um aus einem Quelltext ein lauffähiges Programm zu erzeugen, erledigt die Entwicklungsumgebung für uns. Doch vom Quellcode bis zur ausführbaren Datei ist es für unsere Entwicklungsumgebung ein langer Weg. Zuerst einmal werden sämtliche Präprozessor-Direktiven abgearbeitet. Es entsteht eine temporäre Quelltextdatei, in der sämtlicher Quellcode (also auch die eingebundenen Dateien) vertreten ist. Diese wird dann kompiliert, und es entstehen eine oder mehrere Objektdateien. Anders als bei Visual Studio 2015 Community Edition werden diese Objektdateien jedoch nicht per Default im Projektpfad gespeichert, sondern liegen unter */Library/Developer/Xcode/DerivedData/ProjektName/Build/*

Intermediates. Beim Kompilieren wird der Quelltext in eine für den Computer verständliche Sprache umgewandelt. Welche das ist, kann man leicht erraten, wenn man vorhin alle Abschnitte gelesen und nicht übersprungen hat: Assembler. Diese Objektdateien sind natürlich noch keine ausführbaren Programme. Vielmehr handelt es sich um „Bruchstücke“, die noch miteinander verbunden werden müssen. Diese Aufgabe erledigt der Linker. Er führt sozusagen sämtliche Bruchstücke zur endgültigen ausführbaren Datei zusammen. Es werden gegebenenfalls diverse Libraries (Bibliotheksdateien) mit eingebunden. Diese Bibliotheksdateien stellen Sammlungen von Funktionen dar, die zum Teil dem Compiler beiliegen oder auch speziell von diversen Anbietern aus dem Internet heruntergeladen werden können. Es gibt zum Beispiel Bibliotheksdateien zum Abspielen von MP3-Dateien, die fertige Funktionen zum einfachen Gebrauch zur Verfügung stellen und vieles mehr.

Bei den hier im Buch vorgestellten Entwicklungsumgebungen braucht man sich keine Gedanken über diese Schritte zu machen, da alles für uns erledigt wird. Durch Anwählen eines Menüpunkts oder durch Drücken eines Shortcuts (Tastenkürzel) wird aus unserem Quelltext ein ausführbares Programm (solange der Quelltext keine schweren Fehler enthält, die dies verhindern).

Die erzeugte, ausführbare Datei findet sich ebenfalls (mit den Standard-Einstellungen) in dem vorangehend beschriebenen Pfad in den jeweiligen Unterordnern *Build/Product/Debug* oder *Build/Product/Release*. Über ein Terminal kann die Datei auch direkt per `./Datei` ausgeführt werden. Wie das Programm wesentlich einfacher direkt aus der Entwicklungsumgebung heraus gestartet werden kann, erkläre ich Dir im nächsten Abschnitt.

Du solltest jetzt Xcode und den Quelltext aus dem ersten Beispiel vor Dir haben. Ein Klick auf PRODUCT > BUILD sorgt dafür, dass unser erstes Programm kompiliert und im Erfolgsfall auch gleich gelinkt wird.

HINWEIS: Klickt man erneut auf PRODUCT > BUILD, so wird nur dann neu kompiliert und gelinkt, wenn es auch Änderungen am Quellcode gab. Ansonsten wird die ausführbare Datei nicht erneut erstellt. Generell ist es so, dass immer nur die Quellcode-Dateien kompiliert werden, die seit dem letzten Build-Prozess eine Änderung erfahren haben. Das gilt dann natürlich auch für alle daraus entstehenden Abhängigkeiten.

Durch einen Klick auf PRODUCT > CLEAN werden sämtliche erzeugen Objektdateien gelöscht. Das führt dazu, dass das Projekt komplett neu gebaut werden muss, wenn man erneut auf PRODUCT > BUILD klickt. Das mag bei unserem ersten Beispiel noch nicht dramatisch sein, jedoch wird es bei großen Projekten sehr lange dauern und unnötig Zeit rauben. Ein „Clean“ist in der Regel nur dann nötig, wenn beim Kompiliervorgang etwas schief ging.

Wenn es nichts zu beanstanden gab und Du alle Schritte richtig ausgeführt hast, sollte im mittleren, oberen Bereich von Xcode der Status *Build Listing_1_1:* **Succeeded** | *Today at (Uhrzeit)* zu sehen sein.

Wenn Deine Ausgabe so ausschaut wie vorangehend gezeigt, dann ging alles glatt, und Du hast Deine erste ausführbare Datei erstellt. Diese kannst Du durch Drücken von CMD+R starten.

1.5.6 Ausführen des Programms

Du kannst das Programm immer direkt nach dem Kompilieren aus der Entwicklungsumgebung heraus starten, ohne die erzeugte Datei extra aufrufen zu müssen. Klicke dazu entweder im Menü auf PRODUCT→RUN oder drücke alternativ den Shortcut CMD+R. Wenn Du die Default-Einstellungen von Xcode nicht verändert hast, dann sollte jetzt unten rechts die nachfolgende Ausgabe erscheinen. Falls das nicht der Fall ist, dann klicke im Menü auf VIEW→DEBUG AREA→ACTIVATE CONSOLE oder verwende den Shortcut SHIFT+CMD+C.

```
Hier kommt die Konkurrenz!
Press any key to continue
```

Okay, zugegeben: Das ist nicht gerade das, was man spektakulär nennen würde. Trotzdem wird mit diesem kleinen Beispiel sozusagen ein Grundstein gelegt. Du bist nun in der Lage, ein eigenes Programm zu schreiben, und sei es noch so klein und unscheinbar. Fast alle, die sich mit dem Programmieren von Spielen oder Anwendungsprogrammen beschäftigen, haben einmal mit einem solchen kleinen Programm angefangen. Text auf dem Bildschirm auszugeben ist zu Beginn das Wesentlichste, was man beherrschen muss. Wie Du sehen wirst, ist diese Voraussetzung die Grundlage, die in den folgenden Kapiteln unerlässlich ist.

Später, wenn unsere Programme auch Eingaben vom Benutzer erwarten, kann ebenfalls das eben gesehene Ausgabefenster verwendet werden.

Jetzt ist es an der Zeit, etwas mit dem Beispiel herumzuexperimentieren. Füge weitere Zeilen hinzu, in denen mehr Text ausgegeben wird, und mache rege von den Escape-Zeichen aus Tabelle 1.1 Gebrauch. Denk daran, dass jede Zeile mit einem Semikolon abgeschlossen werden muss, um Fehler zu vermeiden.

HINWEIS: Es ist sehr wichtig, immer mit den Beispielen zu experimentieren. Einfach nur die Listings zu kompilieren und sich das Resultat anzuschauen, genügt eben nicht. Versuche den Quelltext etwas abzuändern, ohne jedoch bei diesem Thema vorzugreifen. Dadurch wirst Du schneller mit den erklärten Themen vertraut, und der Lerneffekt ist größer.

1.6 Umstellen der Build-Konfiguration

Vorangehend wurde ja bereits erwähnt, dass es verschiedene Konfigurationen gibt, um Deinen Quellcode zu kompilieren. Die Standardeinstellung, die wir bisher verwendet haben, nennt sich „Debug“. Es gibt jedoch noch eine weitere Konfiguration namens „Release“. Doch was ist nun der Unterschied zwischen beiden? Um dies herauszufinden, stellst Du jetzt am besten einfach mal die andere Konfiguration um. Auch hier gibt es wieder sowohl für Visual Studio 2015 Community Edition als auch für Xcode 6.4 eine Erklärung.

1.6.1 Build-Konfiguration in Visual Studio 2015 Community Edition umstellen

Die Build-Konfiguration kann in Visual Studio 2015 Community Edition auf zwei verschiedene Arten umgestellt werden. Die erste Möglichkeit besteht darin, einfach das entsprechende Dropdown-Feld oben in der Toolbar zu benutzen und dort RELEASE zu wählen. Alternativ kannst Du auch im Menü auf ERSTELLEN > KONFIGURATIONS-MANAGER klicken und im daraufhin erscheinenden Fenster unter KONFIGURATION DER AKTUELLEN PROJEKTMAPPE den Eintrag RELEASE wählen.

1.6.2 Build-Konfiguration in Xcode 6.4 umstellen

Klicke in der oberen linken Leiste rechts neben den Buttons RUN und STOP auf das Dropdown-Feld, das den gleichen Namen wie Dein Projekt hat (beispielsweise LISTING_1_1). Wähle in der nun erscheinenden Liste den Eintrag EDIT SCHEME… Im daraufhin erscheinenden Fenster wählst Du in der linken Liste den Eintrag RUN. In der rechten Hälfte des Fensters wählst Du in der oberen Auswahl die Kategorie INFO und stellst bei dem Eintrag BUILD CONFIGURATION den zugehörigen Eintrag auf RELEASE um.

1.6.3 Der Unterschied zwischen Debug und Release

Wenn Du diese Einstellung vorgenommen hast, musst Du das Projekt neu erstellen lassen. In den jeweiligen Ordnern (unter Visual Studio 2015 Community Edition ist es das Projektverzeichnis, unter Xcode */Library/Developer/Xcode/DerivedData/ProjektName/Build/Products*) wird nun ein Ordner namens „Release“ erzeugt, in dem die ausführbare Datei zu finden sein wird. Vergleichst Du nun die Größe der beiden ausführbaren Dateien (in den Ordnern „Debug“ und „Release“), dann wirst Du feststellen, dass die Release-Version um einiges kleiner ist. Der Grund dafür ist recht einfach: Die Debug-Version enthält eine Fülle von Informationen, die beim Debuggen des Programms nützlich sind. Unter „Debuggen“ versteht man die gezielte Fehlersuche in einem Programm, wenn man es beispielsweise im Einzelschrittmodus laufen lässt. In Abschnitt 8.8 werde ich Dir zeigen, wie man einen Quelltext debuggen kann.

Die Debug-Version ist auch um einiges absturzsicherer, da bestimmte Fehler einfach abgefangen werden (natürlich bekommt man auch recht einfach die Debug-Version zum Abstürzen, wenn man sich nur etwas Mühe gibt). Es kann also durchaus vorkommen, dass die Debug-Version fehlerfrei läuft und die Release-Version den gesamten Rechner zum Absturz bringt. Der Preis für diese etwas stabilere und mit Informationen gespickte Debug-Version sind sowohl die Dateigröße als auch der Geschwindigkeitsunterschied. Bezogen auf die Beispiele in diesem Buch wird sich wohl kaum ein Geschwindigkeitsunterschied messen lassen, da eigentlich nichts Rechenintensives passiert. Deshalb kann man das hier erst einmal vernachlässigen. Wenn man allerdings ein richtiges Spiel programmiert, dann kommt es auf jedes Fitzelchen Geschwindigkeit an, da bringt dann das Umstellen von Debug auf Release wirklich noch einen großen Geschwindigkeitsvorteil.

Generell sollte man während der Entwicklungsphase die meiste Zeit über die Debug-Konfiguration eingestellt lassen. Nur gelegentlich sollte man auf Release stellen und prüfen, ob das Programm weiterhin wie erwartet läuft. Sobald das Programm fertiggestellt ist und fehlerfrei funktioniert, sollte man auf die Release-Konfiguration umstellen. Wird das Programm zum Download bereitgestellt oder sogar verkauft, sollte immer nur die Release-Version verwendet werden, unter anderem auch, weil in der Debug-Version oft noch Teile des Quellcodes des Programms vorhanden sind, die dann einfach so gelesen werden könnten.

PRAXISTIPP: Wenn Du an einem Projekt arbeitest, das mehrere Tage, Wochen oder gar Monate Entwicklungszeit in Anspruch nimmt, dann teste dieses gelegentlich auch mit der Release-Version. So ersparst Du Dir das leidige Suchen, wenn Du einen Fehler eingebaut hast, der in der Debug-Version nicht zum Tragen kommt. Um ganz sicherzugehen, solltest Du jedoch so oft wie möglich beide Versionen testen.

So, damit ist dieses erste Kapitel beendet. Möglicherweise war es an einigen Stellen etwas trocken, aber wie so oft müssen gewisse Grundlagen einfach erstmal durchgearbeitet werden. Dafür wird das nächste Kapitel auch wesentlich praxisorientierter, versprochen!

Wenn Du noch nicht mit dem Quellcode aus Listing 1.1 experimentiert hast, solltest Du das jetzt nachholen. Wichtig ist, dass Du mit der Entwicklungsumgebung vertraut bist und die `cout`-Funktion beherrschst, bevor Du Dich ans nächste Kapitel machst.

1.7 Aufgabe

Die erste Aufgabe in diesem Buch unterscheidet sich ein wenig von den anderen Aufgaben, denn es gibt in diesem Sinne keine Lösung und auch keinen Quelltext dafür. Da wir bisher noch nicht sonderlich viel über das eigentliche Programmieren gelernt haben, besteht diese Aufgabe eigentlich nur aus der „Festigung“ der elementaren Grundlagen.

Es liegt natürlich an Dir, ob Du die Aufgaben in diesem Buch wahrnimmst und versuchst, sie zu lösen. Dennoch kann ich Dir nur empfehlen, das zu tun. Wenn man etwas liest, meint man schnell, alles verstanden zu haben, und ist der Meinung, Aufgaben wären an dieser Stelle unnötig. Schnell stellt man aber fest, dass es schon anders ausschaut, wenn man etwas selbstständig machen soll.

In diesem Kapitel ging es bisher ja darum, einen neuen Arbeitsbereich anzulegen und mithilfe von `cout` Text auf dem Bildschirm auszugeben. Dazu gehörten auch die `main`-Funktion und das Einbinden der nötigen Header-Dateien. Und genau darum geht es bei dieser Aufgabe. Versuche mal selbstständig, ohne viel nachzuschlagen, einen neuen Arbeitsbereich zu erstellen, die nötigen Header einzubinden und etwas Text auszugeben. Nutze dabei auch die in Tabelle 1.1 gezeigten Escape-Sequenzen. Programmieren ist zwar nicht das stumpfe Auswendiglernen von allen möglichen Dingen, jedoch sollte man die Grundlagen so weit beherrschen, dass man nicht immer wieder nachschlagen muss.

Index

D

E

F

G

H

I

K

L

M

N

O

T

U

V

W

X

Z