



Xpert.press

Richard Kaiser

C++ mit Visual Studio 2017

Ein Fach- und Lehrbuch
für Standard-C++

EBOOK INSIDE

 Springer Vieweg

Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Weitere Bände in der Reihe <http://www.springer.com/series/4393>

Richard Kaiser

C++ mit Visual Studio 2017

Ein Fach- und Lehrbuch für Standard-C++

Richard Kaiser
Fakultät Technik
Duale Hochschule Baden-Württemberg
Lörrach, Deutschland

ISSN 1439-5428 ISSN 2522-0667 (electronic)
Xpert.press
ISBN 978-3-662-49792-0 ISBN 978-3-662-49793-7 (eBook)
<https://doi.org/10.1007/978-3-662-49793-7>

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

Springer Vieweg

© Springer-Verlag GmbH Deutschland 2018, korrigierte Publikation 2018

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer-Verlag GmbH, DE und ist ein Teil von Springer Nature

Die Anschrift der Gesellschaft ist: Heidelberger Platz 3, 14197 Berlin, Germany

Für

Daniel, Alex, Kathy

Emelie, Jonathan und Maximilian

Vorwort

Dieses Buch erscheint in zwei weitgehend identischen Ausgaben:

- **In der vorliegenden Ausgabe** werden reine Standard-C++-Programme geschrieben, d.h. ohne graphische Benutzeroberfläche. Alle Ein- und Ausgaben erfolgen über die Konsole.
- **In der anderen Ausgabe** „C++ mit Visual Studio 2017 und Windows-Forms-Anwendungen“ werden Programme mit einer grafischen Benutzeroberfläche geschrieben. Alle Ein- und Ausgaben erfolgen über eine Windows-Benutzeroberfläche.

Beide Ausgaben sind sowohl ein Lehrbuch, das keine Vorkenntnisse voraussetzt, als auch ein Fachbuch, das alle C++-Themen behandelt, die in der professionellen Software-Entwicklung eingesetzt werden. Die Themen bauen schrittweise aufeinander auf. Der Aufbau hat sich in zahlreichen Vorlesungen und Seminaren für Firmen bewährt.

Der Inhalt der beiden Ausgaben ist im Wesentlichen identisch. Das vorliegende Buch enthält aber einige weitere Themen, die etwas spezieller sind, sowie ein Kapitel über Standard-C++ Multithreading, das in Windows Forms Anwendungen nicht verfügbar ist. Die andere Ausgabe enthält dagegen Ausführungen zur Ein- und Ausgabe von Daten über die graphische Benutzeroberfläche.

Der Unterschied zwischen den beiden Ausgaben ist oft nur, dass in „C++ mit Visual Studio 2017 und Windows Forms Anwendungen“ Ein- und Ausgaben über ein Windows-Steuer-element (meist eine TextBox) erfolgen

```
textBox1->AppendText("Hello World");
```

während in „C++ mit Visual Studio 2017“ die Konsole verwendet wird:

```
cout << "Hallo world" << endl;
```

Die vorliegende Ausgabe ist eine umfassende Überarbeitung meines Buchs „C++ mit Visual Studio 2008“. Da der damals gültige C++-Standard stark erweitert wurde (zu C++11 und C++14), unterscheidet sie sich in den folgenden Punkten von dem Buch über Visual C++ 2008:

- Die Sprachelemente von C++11 und C++14 werden von Anfang an eingesetzt.

- Da ausschließlich Standard-C++ behandelt wird, ist dieses Buch nicht auf Visual Studio beschränkt. Praktisch alle Ausführungen gelten für jeden standardkonformen C++-Compiler (wie gcc, Intel, Embarcadero usw.).
- Der Umfang wurde auf ca. 800 Seiten begrenzt, um Leser nicht schon allein durch das Gewicht und die Fülle des Stoffs zu abzuschrecken. Die Erfahrungen aus meinen Firmenseminaren geben mir aber die Hoffnung, dass die allermeisten Themen abgedeckt sind, die im industriellen Einsatz notwendig sind.
- Viele Rückmeldungen aus meinen Vorlesungen (vor allem für Elektrotechnik-Studenten an der Dualen Hochschule Lörrach) und Firmen-Seminaren wurden eingearbeitet.

Die Lernziele sind dieselben wie schon in dem Buch über Visual Studio 2008:

- Die wichtigsten Sprachelemente von C/C++ kennenlernen. C++ ist nach wie vor eine der am häufigsten eingesetzten Programmiersprachen.
- Programmieren lernen, d.h. Programme zu schreiben, die konkrete, vorgegebene Aufgaben lösen. Das ist nur mit viel Übung möglich. Deshalb enthält dieses Buch auch viele Aufgaben. Es ist unerlässlich, zahlreiche Übungsaufgaben selbständig zu lösen.
- Eine moderne Entwicklungsumgebung kennenlernen. Insbesondere zu lernen, wie man sie effektiv einsetzt. Visual Studio ist das in der Industrie wohl am häufigsten eingesetzte Werkzeug zur Software-Entwicklung.

Ich habe versucht, bei allen Konzepten nicht nur die Sprachelemente und ihre Syntax zu beschreiben, sondern auch Kriterien dafür anzugeben, wann und wie man sie sinnvoll einsetzen kann.

Man hört immer wieder die Meinung, dass C++ zu schwierig ist, um es als einführende Programmiersprache einzusetzen. Dieses Buch soll ein in vielen Jahren erprobtes Gegenargument zu dieser Meinung sein. Damit will ich aber die Komplexität von C++ überhaupt nicht abstreiten. Wer C++ kann, findet sich leicht mit C#, Java usw. zurecht. Der umgekehrte Weg ist meist schwieriger.

Zahlreiche Übungsaufgaben geben dem Leser die Möglichkeit, die Inhalte praktisch anzuwenden und so zu vertiefen. Da man Programmieren nur lernt, indem man es tut, möchte ich ausdrücklich dazu ermuntern, zumindest einen Teil der Aufgaben zu lösen und sich dann selbst neue Aufgaben zu stellen. Der Schwierigkeitsgrad der Aufgaben reicht von einfachen Wiederholungen des Textes bis zu kleinen Projektchen, die ein gewisses Maß an selbständiger Arbeit erfordern. Die Lösungen der meisten Aufgaben findet man auf meiner Internetseite <http://www.rkaiser.de>.

Anregungen, Korrekturhinweise und Verbesserungsvorschläge sind willkommen. Bitte senden Sie diese an die Mail-Adresse auf meiner Internetseite.

Bei meinen Schulungskunden und Studenten bedanke ich mich für die zahlreichen Anregungen. Herrn Engesser und seinem Team vom Springer-Verlag danke ich für die Unterstützung und Geduld.

*Die Original-Version des Buches wurde
korrigiert. Ein Erratum finden Sie unter
https://doi.org/10.1007/978-3-662-49793-7_19*

Inhalt

1 Die Entwicklungsumgebung	1
1.1 Installation von Visual Studio für C++ Projekte	1
1.2 Ein erstes C++-Projekt	2
1.2.1 Ein Projekt für ein Standard-C++-Programm anlegen	2
1.2.2 Ein- und Ausgaben über die Konsole	4
1.2.3 Fehler beim Kompilieren	6
1.2.4 Den Quelltext auf Header-Dateien aufteilen	8
1.2.5 Ein Projekt für die Lösung der Übungsaufgaben	9
1.2.6 An einem Projekt weiterarbeiten	10
1.2.7 Der Start des Compilers von der Kommandozeile Θ	10
1.3 Der Quelltexteditor	11
1.3.1 Tastenkombinationen	11
1.3.2 Intellisense	13
1.3.3 Die Formatierung des Quelltexts	13
1.3.4 Definitionen einsehen	14
1.3.5 Symbole suchen	15
1.3.6 Namen umbenennen	16
1.3.7 Zeichenfolgen suchen und ersetzen	17
1.4 Kontextmenüs und Symbolleisten	19
1.5 Die Online-Hilfe (MSDN Dokumentation)	20
1.5.1 Hilfe mit F1 in Visual Studio	21
1.5.2 Die MSDN-Dokumentation im Internet	21
1.6 Projekte und der Projektmappen-Explorer	23
1.6.1 Projekte, Projektdateien und Projektoptionen	23
1.6.2 Projektmappen und der Projektmappen-Explorer	24
1.7 Weiterführende Möglichkeiten Θ	26
1.7.1 Navigieren	26
1.7.2 Code-Ausschnitte	28
1.7.3 Aufgabenliste	28
1.7.4 Der Objektkatalog und die Klassenansicht Θ	29
1.7.5 Die Fenster von Visual Studio anordnen Θ	29
1.7.6 Einstellungen für den Editor Θ	30
1.8 Bereitstellung (Deployment) Θ	31

2	Elementare Datentypen und Anweisungen	33
2.1	Syntaxregeln	33
2.2	Variablen und Bezeichner	37
2.3	Ganzzahldatentypen	40
2.3.1	Die interne Darstellung von Ganzzahlwerten	43
2.3.2	Ganzzahlliterale und ihr Datentyp	45
2.3.3	Typ-Inferenz: Implizite Typzuweisungen mit auto	48
2.3.4	Zuweisungen und Standardkonversionen bei Ganzzahlausdrücken	49
2.3.5	Operatoren und die „üblichen arithmetischen Konversionen“	52
2.3.6	Die Datentypen char und wchar_t	57
2.3.7	Der Datentyp bool	62
2.4	Kontrollstrukturen und Funktionen	66
2.4.1	Die if- und die Verbundanweisung	66
2.4.2	Die for-, die while- und die do-Schleife	70
2.4.3	Funktionen und der Datentyp void	74
2.4.4	Eine kleine Anleitung zum Erarbeiten der Lösungen	77
2.4.5	Werte- und Referenzparameter	81
2.4.6	Die Verwendung von Bibliotheken und Namensbereichen	82
2.4.7	Zufallszahlen	83
2.4.8	Default-Argumente	85
2.4.9	Programmierstil für Funktionen	88
2.4.10	Rekursive Funktionen	94
2.4.11	Die switch-Anweisung Θ	100
2.4.12	Die Sprunganweisungen goto, break und continue Θ	103
2.4.13	Assembler-Anweisungen Θ	105
2.5	Gleitkommatypen	105
2.5.1	Die interne Darstellung von Gleitkommawerten	106
2.5.2	Der Datentyp von Gleitkommaliteralen	109
2.5.3	Standardkonversionen	110
2.5.4	Mathematische Funktionen	115
2.6	Der Debugger, Tests und Ablaufprotokolle	119
2.6.1	Der Debugger	120
2.6.2	Der Debugger – Weitere Möglichkeiten Θ	123
2.6.3	Systematisches Testen	127
2.6.4	Unittests: Funktionen, die Funktionen testen	133
2.6.5	Ablaufprotokolle	137
2.6.6	Symbolische Ablaufprotokolle	141
2.7	Konstanten	146
2.7.1	Laufzeitkonstanten mit const	146
2.7.2	Compilezeit-Konstanten mit constexpr	149
2.7.3	constexpr Funktionen Θ	149
2.8	Kommentare	151
2.8.1	Kommentare zur internen Dokumentation	152
2.8.2	Kommentare und Intellisense	154
2.8.3	Dokumentationskommentare für externe Programme Θ	155
2.9	Exception-Handling Grundlagen: try, catch und throw	156
2.10	Namensbereiche – Grundlagen	160
2.11	Präprozessoranweisungen	162

2.11.1	Die #include-Anweisung	162
2.11.2	Makros Θ	164
2.11.3	Bedingte Kompilation	166
2.11.4	Pragmas Θ	170
3	Die Stringklassen: string, wstring usw.	173
3.1	Die Definition von Variablen eines Klassentyps	174
3.2	Einige Elementfunktionen der Klasse string.....	176
3.3	Raw-String-Literale (Rohzeichenfolgen).....	186
3.4	Konversionen zwischen string/wstring und elementaren Datentypen.....	188
3.5	Konversionen zwischen string und Klassen mit Stringstreams Θ	191
3.6	Unicode-Strings Θ	193
3.7	Landespezifische Einstellungen Θ	193
3.8	Reguläre Ausdrücke Θ	196
4	Arrays und Container	207
4.1	Synonyme für Datentypen	208
4.1.1	Einfache typedef-Deklarationen	208
4.1.2	Synonyme für Datentypen mit using.....	208
4.2	Eindimensionale Arrays.....	209
4.3	Die Initialisierung von Arrays bei ihrer Definition	215
4.4	Arrays als Container	216
4.5	Mehrdimensionale Arrays Θ	220
4.6	Dynamische Programmierung Θ	221
5	Einfache selbstdefinierte Datentypen	223
5.1	Mit struct definierte Klassen	223
5.2	Aufzählungstypen	229
5.2.1	Schwach typisierte Aufzählungstypen (C/C++03).....	229
5.2.2	enum Konstanten und Konversionen Θ	231
5.2.3	Stark typisierte Aufzählungstypen (C++11)	232
6	Zeiger, Strings und dynamisch erzeugte Variablen	235
6.1	Die Definition von Zeigervariablen	237
6.2	Der Adressoperator, Zuweisungen und generische Zeiger	239
6.3	Ablaufprotokolle für Zeigervariable	243
6.4	Dynamisch erzeugte Variablen	244
6.4.1	new und delete	245
6.4.2	Der Unterschied zu „gewöhnlichen“ Variablen	248
6.4.3	Memory Leaks in Visual C++ finden Θ	251
6.5	Dynamische erzeugte eindimensionale Arrays	253
6.6	Arrays, Zeiger und Zeigerarithmetik	255

6.7	Arrays als Funktionsparameter Θ	258
6.8	Funktionszeiger und Datentypen für Funktionen Θ	260
6.9	Konstante Zeiger	261
6.10	Stringlitterale, nullterminierte Strings und <code>char*</code> -Zeiger	263
6.11	Verkettete Listen	268
6.12	Binärbäume Θ	278
6.13	Zeiger als Parameter Θ	283
6.14	C-Bibliotheksfunktionen in <code>string.h</code> für nullterminierte Strings Θ	284
7	Überladene Funktionen und Operatoren	289
7.1	Inline-Funktionen Θ	289
7.2	Überladene Funktionen	291
7.2.1	Funktionen, die nicht überladen werden können	293
7.2.2	Regeln für die Auswahl einer passenden Funktion	294
7.3	Überladene Operatoren mit globalen Operatorfunktionen	300
7.3.1	Globale Operatorfunktionen	302
7.3.2	Die Ein- und Ausgabe von selbst definierten Datentypen	305
7.4	Referenztypen, Werte- und Referenzparameter	307
7.4.1	Werteparameter	307
7.4.2	Referenztypen	307
7.4.3	Referenzparameter	309
7.4.4	Referenzen als Rückgabetypen	311
7.4.5	Konstante Referenzparameter	313
8	Objektorientierte Programmierung	317
8.1	Klassen	318
8.1.1	Datenelemente und Elementfunktionen	319
8.1.2	Der Gültigkeitsbereich von Klasselementen	323
8.1.3	Datenkapselung: Die Zugriffsrechte <code>private</code> und <code>public</code>	326
8.1.4	Der Aufruf von Elementfunktionen und der <code>this</code> -Zeiger	332
8.1.5	Konstruktoren und Destruktoren	333
8.1.6	OO Analyse und Design: Der Entwurf von Klassen	345
8.1.7	Klassendiagramme	349
8.2	Klassen als Datentypen	350
8.2.1	Der Standardkonstruktor	351
8.2.2	Objekte als Klasselemente und Elementinitialisierer	353
8.2.3	Initialisiererlisten	358
8.2.4	friend-Funktionen und <code>-</code> Klassen	363
8.2.5	Überladene Operatoren mit Elementfunktionen	366
8.2.6	Der Kopierkonstruktor	370
8.2.7	Der Zuweisungsoperator <code>=</code> für Klassen	375
8.2.8	Die Angaben <code>=delete</code> und <code>=default</code>	380
8.2.9	Konvertierende und explizite Konstruktoren Θ	382
8.2.10	Konversionsfunktionen mit und ohne <code>explicit</code> Θ	386
8.2.11	Statische Klasselemente	387

8.2.12	Konstante Objekte und Elementfunktionen	391
8.2.13	Funktionen als Objekte und Parameter mit <code>std::function</code>	393
8.2.14	Delegierende Konstruktoren Θ	397
8.2.15	Klassen und Header-Dateien	399
8.3	Vererbung und Komposition	401
8.3.1	Die Elemente von abgeleiteten Klassen	401
8.3.2	Zugriffsrechte auf die Elemente von Basisklassen	403
8.3.3	Verdeckte Elemente	405
8.3.4	Konstruktoren, Destruktoren und implizit erzeugte Funktionen	408
8.3.5	OO Design: public Vererbung und „ist ein“-Beziehungen	414
8.3.6	OO Design: Komposition und „hat ein“-Beziehungen	419
8.3.7	Konversionen zwischen public abgeleiteten Klassen	420
8.3.8	Mehrfachvererbung und virtuelle Basisklassen	423
8.4	Virtuelle Funktionen, späte Bindung und Polymorphie	428
8.4.1	Der statische und der dynamische Datentyp	428
8.4.2	Virtuelle Funktionen in C++03	429
8.4.3	Virtuelle Funktionen mit <code>override</code> in C++11	430
8.4.4	Die Implementierung von virtuellen Funktionen: <code>vptr</code> und <code>vtbl</code>	438
8.4.5	Virtuelle Konstruktoren und Destruktoren	444
8.4.6	Virtuelle Funktionen in Konstruktoren und Destruktoren	446
8.4.7	OO-Design: Einsatzbereich und Test von virtuellen Funktionen	447
8.4.8	OO-Design und Erweiterbarkeit	449
8.4.9	Rein virtuelle Funktionen und abstrakte Basisklassen	452
8.4.10	OO-Design: Virtuelle Funktionen und abstrakte Basisklassen	456
8.4.11	Objektorientierte Programmierung: Zusammenfassung	458
8.5	R-Wert Referenzen und Move-Semantik	460
8.5.1	R-Werte und R-Wert Referenzen	461
8.5.2	<code>move</code> -Semantik und <code>std::move</code>	463
8.5.3	Move-Semantik in der C++11 Standardbibliothek	469
8.5.4	Move-Semantik für eigene Klassen	470
9	Namensbereiche	473
9.1	Die Definition von Namensbereichen	474
9.2	Die Verwendung von Namen aus Namensbereichen	477
9.3	Header-Dateien und Namensbereiche	480
9.4	Aliasnamen für Namensbereiche Θ	483
10	Exception-Handling	485
10.1	Die <code>try</code> -Anweisung	486
10.2	Exception-Handler und Exceptions der Standardbibliothek	491
10.3	<code>throw</code> -Ausdrücke und selbst definierte Exceptions	494
10.4	Fehler und Exceptions	500
10.5	Die Freigabe von Ressourcen bei Exceptions: <code>RAII</code>	503
10.6	Exceptions in Konstruktoren und Destruktoren	505

10.7	noexcept.....	511
10.8	Die Exception-Klasse <code>system_error</code> Θ	512
11 Containerklassen der C++-Standardbibliothek		515
11.1	Sequenzielle Container der Standardbibliothek	515
11.1.1	Die Container-Klasse <code>vector</code>	515
11.1.2	Iteratoren	520
11.1.3	Geprüfte Iteratoren (Checked Iterators)	524
11.1.4	Die bereichsbasierte <code>for</code> -Schleife	525
11.1.5	Iteratoren und die Algorithmen der Standardbibliothek.....	528
11.1.6	Die Speicherverwaltung bei Vektoren Θ	531
11.1.7	Mehrdimensionale Vektoren Θ	533
11.1.8	Die Container-Klassen <code>list</code> und <code>deque</code>	534
11.1.9	Gemeinsamkeiten und Unterschiede der sequenziellen Container	535
11.1.10	Die Container-Adapter <code>stack</code> , <code>queue</code> und <code>priority_queue</code> Θ	537
11.1.11	Container mit Zeigern	539
11.1.12	<code>std::array</code> - Array Container fester Größe Θ	539
11.2	Assoziative Container	540
11.2.1	Die Container <code>set</code> und <code>multiset</code>	541
11.2.2	Die Container <code>map</code> und <code>multimap</code>	542
11.2.3	Iteratoren der assoziativen Container.....	544
11.2.4	Ungeordnete Assoziative Container (Hash-Container).....	546
12 Dateibearbeitung mit den Stream-Klassen		551
12.1	Stream-Variablen, ihre Verbindung mit Dateien und ihr Zustand	551
12.2	Fehler und der Zustand von Stream-Variablen	555
12.3	Lesen und Schreiben von Binärdaten mit <code>read</code> und <code>write</code>	557
12.4	Lesen und Schreiben mit den Operatoren <code><<</code> und <code>>></code>	562
12.5	Dateibearbeitung im Direktzugriff Θ	570
12.6	Manipulatoren und Funktionen zur Formatierung von Texten Θ	572
13 Funktoren, Funktionsobjekte und Lambda-Ausdrücke.....		575
13.1	Der Aufrufoperator <code>()</code>	575
13.2	Prädikate und Vergleichsfunktionen	579
13.3	Binder Θ	584
13.4	Lambda-Ausdrücke.....	587
13.5	Lambda-Ausdrücke – Weitere Konzepte Θ	596
13.5.1	Lambda-Ausdrücke werden zu Funktionsobjekten	596
13.5.2	Nachstehende Rückgabetypen	597
13.5.3	Generische Lambda-Ausdrücke	598
13.5.4	Lambda-Ausdrücke höherer Ordnung Θ	598
13.6	Kompatible function-Typen: Kovarianz und Kontravarianz Θ	599

14 Templates	601
14.1 Generische Funktionen: Funktions-Templates	602
14.1.1 Die Deklaration von Funktions-Templates mit Typ-Parametern	603
14.1.2 Spezialisierungen von Funktions-Templates	604
14.1.3 Funktions-Templates mit Nicht-Typ-Parametern	612
14.1.4 Explizit instanziierte Funktions-Templates Θ	614
14.1.5 Explizit spezialisierte und überladene Templates	614
14.1.6 Rekursive Funktions-Templates Θ	618
14.1.7 Variadische Templates	619
14.2 Generische Klassen: Klassen-Templates	623
14.2.1 Die Deklaration von Klassen-Templates mit Typ-Parametern	623
14.2.2 Spezialisierungen von Klassen-Templates	624
14.2.3 Klassen-Templates mit Nicht-Typ-Parametern	631
14.2.4 Explizit instanziierte Klassen-Templates Θ	632
14.2.5 Partielle und vollständige Spezialisierungen Θ	633
14.2.6 Vererbung mit Klassen-Templates Θ	639
14.2.7 Tupel mit <code><tuple></code> Θ	640
14.2.8 Alias Templates Θ	641
14.3 Type Traits	643
14.3.1 Prüfungen bei der Kompilation: <code>static_assert</code>	643
14.3.2 <code>type traits</code> und <code>static_assert</code>	644
14.3.3 Eine Konstruktion von <code>type traits</code>	647
14.3.4 Die <code>type traits</code> Kategorien	648
14.3.5 <code>type traits</code> zur Steuerung der Übersetzung und Optimierung	649
14.4 Typ-Inferenz	651
14.4.1 Implizite Typzuweisungen mit <code>auto</code>	651
14.4.2 Mit <code>decltype</code> den Datentyp eines Ausdrucks bestimmen	656
14.5 Kovarianz und Kontravarianz	659
15 STL-Algorithmen und Lambda-Ausdrücke	661
15.1 Iteratoren	661
15.1.1 Die verschiedenen Arten von Iteratoren	662
15.1.2 Umkehriteratoren	664
15.1.3 Einfügefunktionen und Einfügeiteratoren	665
15.1.4 Stream-Iteratoren	667
15.1.5 Container-Konstruktoren mit Iteratoren	669
15.1.6 Globale Iterator-Funktionen Θ	670
15.2 Lineares Suchen	671
15.3 Zählen	673
15.4 Der Vergleich von Bereichen	674
15.5 Suche nach Teilfolgen	675
15.6 Minimum und Maximum	676
15.7 Mit <code>all_of</code> , <code>any_of</code> , <code>none_of</code> alle Elemente in einem Bereich prüfen	677
15.8 Kopieren und Verschieben von Bereichen	678
15.9 Elemente transformieren und ersetzen	680
15.10 Elementen in einem Bereich Werte zuweisen Θ	682

15.11	Elemente entfernen – das erase-remove Idiom	683
15.12	Die Reihenfolge von Elementen vertauschen	686
15.12.1	Elemente vertauschen	686
15.12.2	Permutationen Θ	686
15.12.3	Die Reihenfolge umkehren und Elemente rotieren Θ	688
15.12.4	Elemente durcheinander mischen Θ	689
15.13	Algorithmen zum Sortieren und für sortierte Bereiche	689
15.13.1	Partitionen Θ	689
15.13.2	Bereiche sortieren	690
15.13.3	Binäres Suchen in sortierten Bereichen	693
15.13.4	Mischen von sortierten Bereichen	694
15.14	Numerische Berechnungen	696
15.14.1	Verallgemeinerte numerische Algorithmen	696
15.14.2	Valarrays Θ	699
15.14.3	Zufallszahlen mit <code><random></code> Θ	701
15.14.4	Komplexe Zahlen Θ	703
15.14.5	Numerische Bibliotheken neben dem C++-Standard Θ	706
16	Zeiten und Kalenderdaten mit chrono	707
16.1	Brüche als Datentypen: Das Klassen-Template ratio	707
16.2	Ein Datentyp für Zeiteinheiten: duration	709
16.3	Datentypen für Zeitpunkte: time_point	712
16.4	Uhren: system_clock und steady_clock	714
17	Multithreading	719
17.1	Funktionen als Threads starten	720
17.1.1	Funktionen mit async als Threads starten	721
17.1.2	Funktionen mit thread als Threads starten	725
17.1.3	Lambda-Ausdrücke als Threads starten	728
17.1.4	Zuweisungen und move für Threads	732
17.1.5	Die Klassen future und promise	733
17.1.6	Exceptions in Threads und ihre Weitergabe mit promise	737
17.1.7	Der Programmablauf mit async	741
17.1.8	Informationen über Threads	748
17.1.9	Sleep-Funktionen	752
17.1.10	Threads im Debugger	753
17.2	Kritische Abschnitte	754
17.2.1	Atomare Datentypen	757
17.2.2	Kritische Bereiche mit mutex und lock_guard sperren	759
17.2.3	Weitere Lock-Klassen: unique_lock und shared_lock	767
17.2.4	Weitere Mutex-Klassen	769
17.2.5	Deadlocks	771
17.2.6	call_once zur Initialisierung von Daten	774
17.2.7	Thread-lokale Daten	775
17.3	Bedingungsvariablen zur Synchronisation von Threads	776

17.4	Die „Parallel Patterns Library“ von Microsoft.....	779
18	C++11 Smart Pointer: shared_ptr, unique_ptr und weak_ptr	781
18.1	Gemeinsamkeiten von unique_ptr und shared_ptr.....	782
18.2	unique_ptr	787
18.3	shared_ptr	790
18.4	Deleter Θ	794
18.5	weak_ptr Θ	796
	Erratum zu: C++ mit Visual Studio 2017.....	E1
19	Literaturverzeichnis.....	801
	Index	803

Θ Angesichts des Umfangs dieses Buches habe ich einige Abschnitte mit dem Zeichen Θ in der Überschrift als „weniger wichtig“ gekennzeichnet. Damit will ich dem Anfänger eine kleine Orientierung durch die Fülle des Stoffes geben. Diese Kennzeichnung bedeutet aber keineswegs, dass dieser Teil unwichtig ist – vielleicht sind gerade diese Inhalte für Sie besonders relevant.



1 Die Entwicklungsumgebung

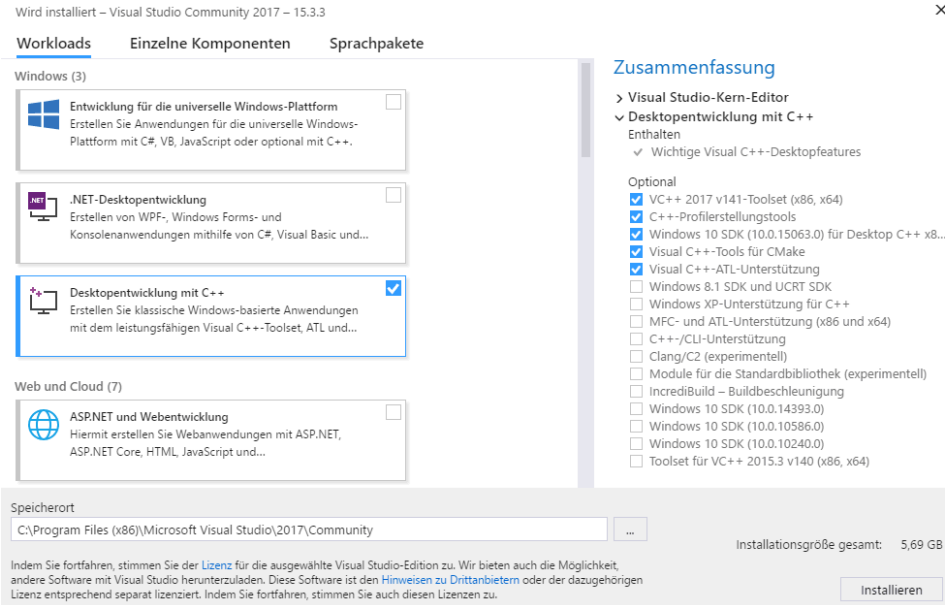
Visual Studio besteht aus verschiedenen Werkzeugen (Tools), die einen Programmierer bei der Entwicklung von Software unterstützen. Eine solche Zusammenstellung von Werkzeugen zur Softwareentwicklung bezeichnet man auch als Programmier- oder **Entwicklungsumgebung**.

Einfache Entwicklungsumgebungen bestehen nur aus einem Editor und einem Compiler. Für eine effiziente Entwicklung von komplexeren Anwendungen sind aber oft weitere Werkzeuge notwendig. Wenn diese wie in Visual Studio in einem einzigen Programm integriert sind, spricht man auch von einer **integrierten Entwicklungsumgebung** (engl.: „integrated development environment“, **IDE**).

In diesem Kapitel wird zunächst an einfachen Beispielen gezeigt, wie man mit Visual Studio 2017 (und früheren Versionen) C++-Programme entwickeln kann. Anschließend (ab Abschnitt 1.3) werden dann die wichtigsten Werkzeuge von Visual Studio ausführlicher vorgestellt. Für viele einfache Anwendungen (wie z.B. die Übungsaufgaben) reichen die Abschnitte bis 1.5. Die folgenden Abschnitte sind nur für anspruchsvollere oder spezielle Anwendungen notwendig. Sie sind deshalb mit dem Zeichen Θ (siehe Seite xvii) gekennzeichnet und können übergangen werden. Weitere Elemente der Entwicklungsumgebung werden später beschrieben, wenn sie dann auch eingesetzt werden können.

1.1 Installation von Visual Studio für C++ Projekte

Damit mit Visual Studio 2017 C++-Programme entwickelt werden können, muss bei der Installation von Visual Studio die „Desktopentwicklung mit C++“ markiert werden:



1.2 Ein erstes C++-Projekt

Im Folgenden wird an einem einfachen Beispiel gezeigt, wie man ein Projekt für ein Standard-C++ Programm anlegt. In dieses Programm werden dann einige einfache Anweisungen aufgenommen.

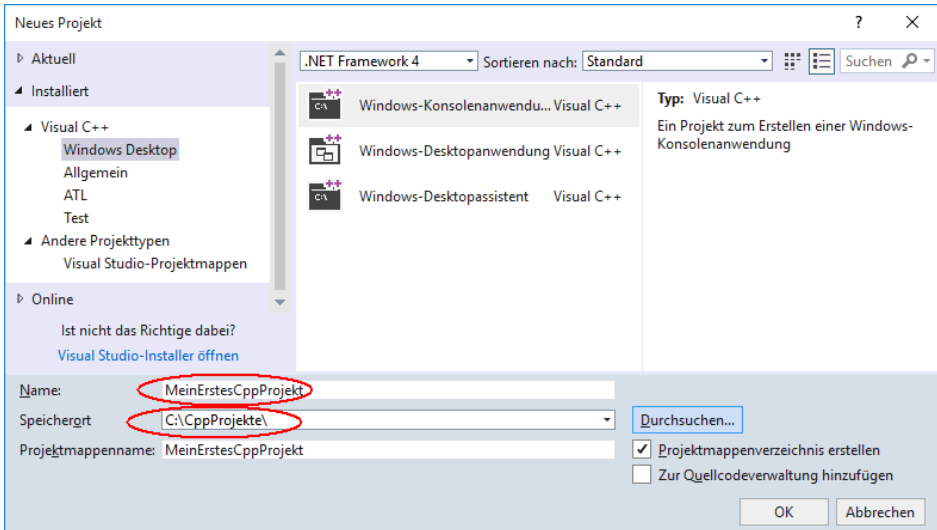
Wenn dabei Begriffe wie „Variable“, „Funktion“ usw. verwendet werden und diese für Sie neu sind, lesen Sie Sie trotzdem weiter – aus dem Zusammenhang erhalten Sie sicherlich eine intuitive Vorstellung, die zunächst ausreicht. Später werden diese Begriffe dann genauer erklärt.

1.2.1 Ein Projekt für ein Standard-C++-Programm anlegen

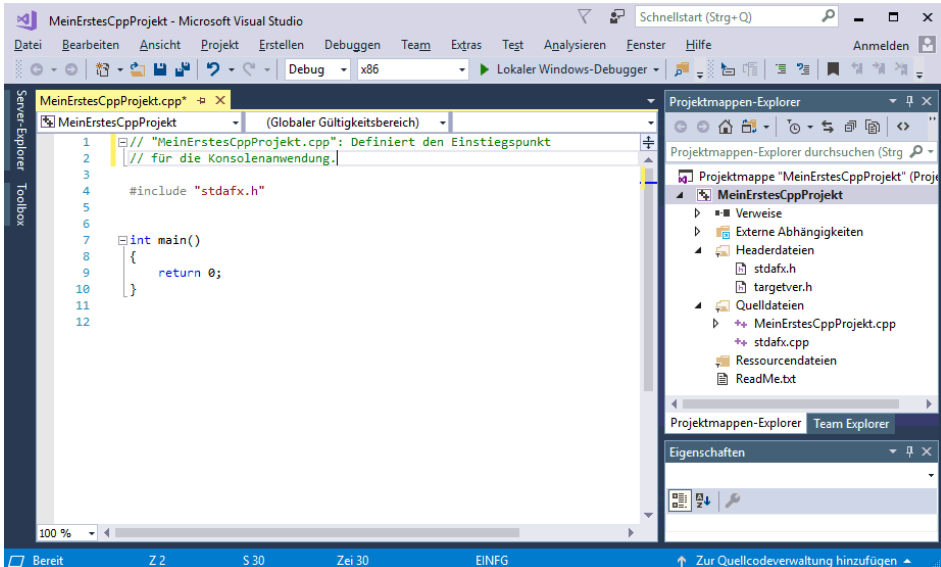
Ein Projekt für ein Standard-C++ Programm erhält man in Visual Studio mit einer Windows-Konsolenanwendung. Eine Konsolenanwendung verwendet wie ein DOS-Programm ein Textfenster für Ein- und Ausgaben. Ein Textfenster ist im Vergleich zu einer grafischen Benutzeroberfläche sehr spartanisch: Es ist nicht möglich, über Buttons oder Menüs verschiedene Aktionen auszuwählen. Obwohl eine Konsolen-Anwendung wie ein DOS-Programm aussieht, kann man es nicht unter MS-DOS, sondern nur unter Windows starten.

Nach der Installation von Visual Studio wie in Abschnitt 1.1 findet man unter **Datei** **Neu-Projekt** **Installiert Visual C++** eine Projektvorlage für Windows-Konsolenanwendungen. Hier gibt man nach *Name* einen Namen und nach *Speicherort* ein Verzeichnis für das Projekt ein und dann den OK-Button anklickt:

Ein Projekt für eine solche Anwendung erhält man, indem man nach *Name* einen Namen und nach *Speicherort* ein Verzeichnis für das Projekt eingibt:



Nach dem Anklicken des OK-Buttons wird links der Editor und rechts der Projektmappen-Explorer angezeigt:



Der Editor enthält eine Funktion mit dem Namen *main*:

```
#include "stdafx.h"

int main()
{
    return 0;
}
```

Diese Funktion wird beim Start des Konsolen-Programms aufgerufen. Die Anweisungen, die durch dieses Programm ausgeführt werden sollen, fügt man dann in diese Funktion vor *return* ein.

1.2.2 Ein- und Ausgaben über die Konsole

Ein- und Ausgaben erfolgen bei einem Konsolenprogramm vor allem über die nach

```
#include <iostream> // für cin und cout notwendig
using namespace std;
```

vordefinierten Streams

```
cin    // für die Eingabe von der Tastatur
cout  // für die Ausgabe am Bildschirm
```

Die Anmerkungen nach „//“ sind übrigens ein sogenannter **Kommentar** (siehe Abschnitt 2.8). Ein solcher Text zwischen „//“ und dem Zeilenende wird vom Compiler nicht übersetzt und dient vor allem der Erläuterung des Quelltextes.

Für *cout* ist der Ausgabeoperator „<<“ definiert. Damit kann man einen Ausdruck (z.B. einen Text oder den Wert einer Variablen) an der Konsole ausgegeben. Mit *endl* wird ein Zeilenvorschub eingefügt, so dass die nächste Ausgabe in der nächsten Zeile beginnt. Fügt man die Anweisung

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

in die *main*-Funktion vor *return 0* ein,

```
#include "stdafx.h"

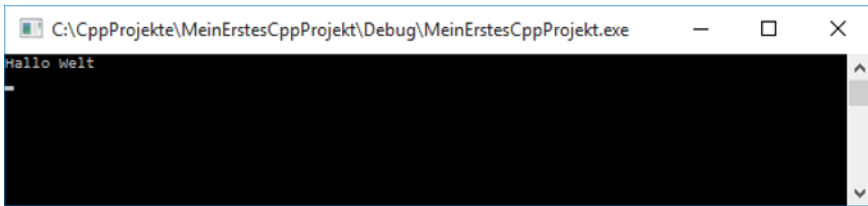
#include <iostream> // für cin und cout notwendig
using namespace std;

int main()
{
    cout << "Hallo Welt" << endl;
    return 0;
}
```

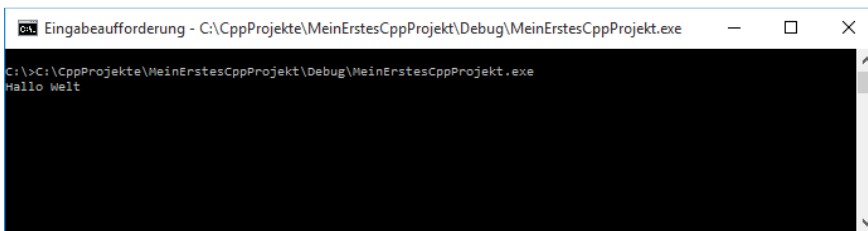
wird beim Start des Programms mit

- mit *Debuggen\Debugging starten* von der Menüleiste, oder
- mit *F5* von einem beliebigen Fenster in Visual Studio oder
- durch den Aufruf der vom Compiler erzeugten Exe-Datei.

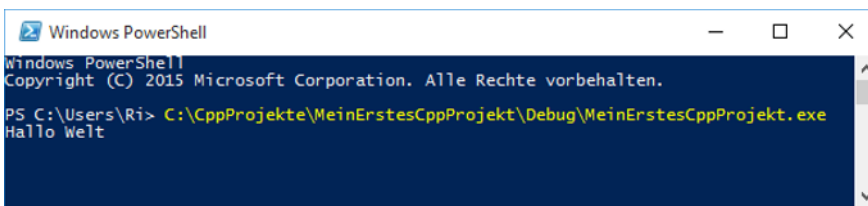
der Text „Hallo Welt“ an der Konsole ausgegeben:



Bitte stören Sie sich nicht daran, dass die Konsole beim Start mit *F5* gleich wieder verschwindet und der Text nur ganz kurz sichtbar ist. Wir werden diesen Schwachpunkt gleich anschließend beheben. Beim Start des Programms von einer Eingabeaufforderung (Windows StartButton *Windows-System*Eingabeaufforderung)



oder über eine PowerShell tritt dieses Problem nicht auf:



Für den Eingabestream *cin* ist der Eingabeoperator „>>“ definiert. Gibt man nach diesem Operator eine Variable an, wartet das Programm bei der Ausführung darauf, dass der Anwender einen Wert an der Konsole eintippt, und weist diesen dann der Variablen zu. Wie man Variablen definiert und verwendet, wird in Abschnitt 2.2 noch genauer beschrieben.

Durch das Programm

```
#include "stdafx.h"

#include <iostream> // für cin und cout notwendig
using namespace std;
```

```
int main()
{
    int x, y;
    cout << "x="; // der Anwender soll einen Wert eingeben
    cin >> x;     // den Wert einlesen
    cout << "y=";
    cin >> y;
    cout << "x+y=" << x + y << endl;
    return 0;
}
```

wird zunächst die Meldung „x=“ ausgegeben, und dann eine vom Benutzer eingegebene Zahl in der Variablen x gespeichert. Entsprechend auch für y. Danach wird die Summe der beiden Werte ausgegeben.

Mit dem Eingabeoperator kann man auch das Problem lösen, dass die Konsole nach dem Start mit F5 gleich wieder verschwindet: Man wartet einfach, dass der Benutzer einen Wert eingibt. Das erreicht man z.B. mit den Anweisungen vor return 0:

```
char c;
cin >> c;
return 0;
```

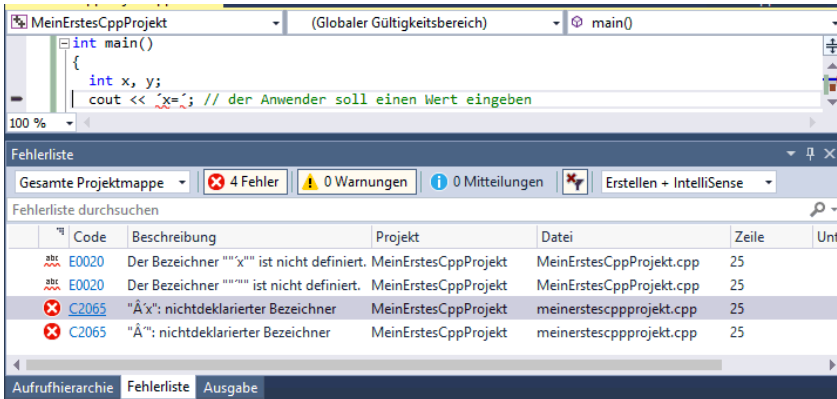
Zur Formatierung der Ausgabe kann man Manipulatoren (siehe Abschnitt 12.6) verwenden.

Aufgabe 1.2.2

Schreiben Sie eine einfache Windows Konsolen-Anwendung, die den Text „Hello world“ am Bildschirm ausgibt.

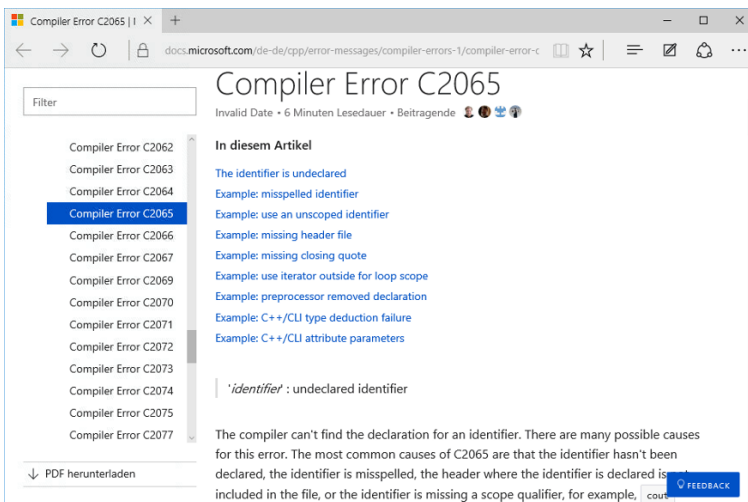
1.2.3 Fehler beim Kompilieren

Bei allen Anweisungen muss man die Sprachregeln von C++ genau einhalten. So muss man z.B. als Begrenzungszeichen für einen String das Zeichen " (*Umschalt+2*) verwenden und nicht eines der ähnlich aussehenden Akzentzeichen ` oder ´ bzw. das Hochkomma ' (*Umschalt+#*). Jedes dieser Zeichen führt bei der Übersetzung des Programms zu einer **Fehlermeldung des Compilers**:



Ein solcher Fehler bedeutet, dass der Compiler die angezeigte Anweisung nicht verstehen kann, weil sie die Sprachregeln von C++ nicht einhält. Ein Fehler zieht oft eine Reihe von Folgefehlern nach sich.

Mit einem Doppelklick auf eine Zeile der Fehlerliste springt der Cursor im Editor an die Stelle mit dem Fehler. Mit *FI* erhält man in der Fehlerliste noch eine ausführlichere Beschreibung des Fehlers.



Wenn Sie eine solche Fehlermeldung des Compilers erhalten, müssen Sie den Fehler im Quelltext beheben. Das kann vor allem für Anfänger eine mühselige Angelegenheit sein, insbesondere wenn die Fehlermeldung nicht so präzise auf den Fehler hinweist wie in diesem Beispiel. Da Fehler Folgefehler nach sich ziehen können, sollte man immer den Fehler zur ersten Fehlermeldung zuerst beheben.

Manchmal sind die **Fehlerdiagnosen** des Compilers sogar eher **irreführend** als hilfreich und schlagen eine falsche Therapie vor. Auch wenn Ihnen das kaum nützt: Betrachten Sie es als kleinen Trost, dass die Fehlermeldungen in anderen Programmiersprachen (z.B. in C) oft

noch viel irreführender sind und schon so manchen Anfänger völlig zur Verzweiflung gebracht haben.

1.2.4 Den Quelltext auf Header-Dateien aufteilen

Damit die Quelltextdateien nicht zu umfangreich und unübersichtlich werden, empfiehlt es sich, diese auf verschiedene Dateien aufzuteilen. Eine solche Aufteilung wird für alle größeren **Projekte** und insbesondere auch für die Lösungen der **Übungsaufgaben** in diesem Buch empfohlen.

Dazu kann man folgendermaßen vorgehen:

1. Mit *Datei\Neu\Projekt\Visual C++\Windows-Konsolenanwendung* (siehe Abschnitt 1.2.1) ein neues Projekt anlegen. Bei den folgenden Beispielen wird ein Projekt mit dem Namen *MeinProjekt* angenommen.
2. Die Funktionen, Deklarationen, Klassen usw. kommen in eine eigene Datei, die dem Projekt mit *Projekt\Neues Element hinzufügen\Visual C++\Code als Headerdatei(.h)* mit einem passenden Namen (z.B. *MeinHeader.h*) hinzugefügt wird. Diese Datei wird dann vor der *main*-Funktion und nach `#include "stdafx.h"` mit einer `#include`-Anweisung in *cpp*-Datei des Projekts aufgenommen:

```
#include "stdafx.h"
#include <iostream> // für cin und cout notwendig
#include "MeinHeader.h"
int main()
{
  ...
```

Im Prinzip hat die `#include`-Anweisung (siehe Abschnitt 2.11.1) der Header-Datei von 3. denselben Effekt, wie wenn man die Anweisungen der Header-Datei an der Stelle der `#include`-Anweisung in das Programm aufnimmt.

Damit auch in der Header-Datei Daten mit *cin* und *cout* ein- und ausgegeben werden können, wird auch in diese Datei am Anfang

```
#include <iostream> // für cin und cout notwendig
```

eingefügt.

Nimmt man in eine solche Header-Datei die Funktion

```
int plus1(int x)
{
  return x + 1;
}
```

auf, kann man diese in der *main*-Funktion aufrufen und das Ergebnis anzeigen:

```
int main()
{
    cout << "17+1=" << plus1(17) << endl; // 17+1=18
    ...
}
```

Bei großen Projekten ist oft eine differenziertere Aufteilung auf Dateien empfehlenswert: In die Header-Dateien werden nur die Deklarationen und in die cpp-Dateien nur die Definitionen aufgenommen. Diese Vorgehensweise hat aber zur Folge, dass man nach jeder Änderung einer Parameterliste in der cpp-Datei die Parameterliste in der Header-Datei anpassen muss. Das ist recht umständlich und führt zu Fehlermeldungen, wenn es vergessen wird. In Abschnitt 8.2.15 wird gezeigt, wie man eine solche Aufteilung mit Unterstützung von Visual Studio leicht durchführen kann.

1.2.5 Ein Projekt für die Lösung der Übungsaufgaben

Die in Abschnitt 1.2.4 beschriebene Aufteilung von Funktionen auf verschiedene Header-Dateien kann auch für die Lösungen der Übungsaufgaben in diesem Buch sinnvoll sein. Wenn man für jede Aufgabe ein eigenes Projekt anlegt, hat man bereits nach Kapitel 2 eine unübersichtlich große Anzahl von Projekten.

Deswegen wird empfohlen, einige oder sogar alle Lösungen eines Kapitels in eine einzige Header-Datei zu schreiben: Für jede Lösung einer Aufgabe werden eine oder mehrere Funktionen geschrieben und diese dann in der *main*-Funktion aufgerufen. Dann kann eine solche Datei zwar auch ziemlich groß werden, wenn Sie alle Aufgaben machen. Aber man kann leicht zu den Lösungen von anderen Aufgaben blättern und sich davon anregen lassen. Mit Namensbereichen (siehe Abschnitt 2.10) kann man eine Header-Datei gut gliedern und die Übersichtlichkeit steigern.

Auf diese Empfehlung wird bei den ersten Aufgaben hingewiesen, später dann nicht mehr.

Der Aufwand, diese Funktionen mit Werten zu testen, die von einem Benutzer über die Konsole eingegeben werden, ist relativ groß. Deshalb reicht es bei vielen Übungsaufgaben aus, diese Funktionen mit hartkodierten Werten aufzurufen:

```
int main()
{
    cout << "plus1(1)=" << plus1(1) << endl;
    cout << "plus1(2)=" << plus1(2) << endl;
    cout << "plus1(3)=" << plus1(3) << endl;
}
```

Ein Programm ist aber flexibler, wenn die Eingabewerte eingelesen werden:

```
int main()
{
    int n;
    cout << "n=";
    cin >> n;
    cout << n << "+1=" << plus1(n) << endl;
}
```

Das Programm wird noch flexibler, wenn mehrere Optionen angeboten werden:

```

int main()
{
    int n, m;
    do {
        cout << "plus1 (1)" << endl;
        cout << "plus2 (2)" << endl;
        cout << "Programmende (0)" << endl;
        cin >> m;
        if (m == 1)
        {
            cout << "n=";
            cin >> n;
            cout << n << "+1=" << plus1(n) << endl; // 17+1=18
        }
        else if (m==2)
        {
            cout << "n=";
            cin >> n;
            cout << n << "+1=" << plus2(n) << endl; // 17+2=19
        }
    } while (m == 0);
}

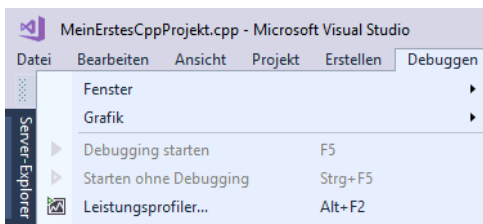
```

Da der Schreibaufwand für eine solche Menüstruktur relativ groß ist, kann in den Lösungen der Übungsaufgaben auch darauf verzichtet werden.

1.2.6 An einem Projekt weiterarbeiten

Wenn man an einem **Projekt weiterarbeiten** will, das man zu einem früheren Zeitpunkt begonnen hat, kann man im Windows Explorer die Projektdatei mit der Namensendung **.sln* oder **.vcproj* öffnen. Da das Projektverzeichnis aber auch noch viele andere Dateien enthält, besteht die Gefahr, dass man eine andere Datei anklickt und dann nicht das Projekt, sondern nur die Datei geöffnet wird.

Das hat dann zur Folge, dass z.B. die Option *Debuggen\Start Debugging* ausgegraut ist:



Diese Gefahr besteht mit *Datei\zuletzt geöffnete Projekte* oder *Datei\Öffnen\Projekt* nicht. Hier werden nur Projekte angeboten.

1.2.7 Der Start des Compilers von der Kommandozeile Ø

Der C++-Compiler von Visual Studio kann nach einem Aufruf von

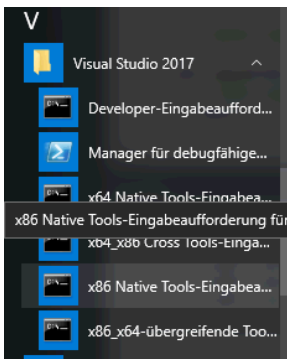
```
"C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\
Common7\Tools\vsdevcmd\ext\vcvars.bat"
```

unter dem Namen *cl* auch von einer Kommandozeile aus (z.B. *Start|Programme|Zubehör|-Eingabeaufforderung* oder *Take Command* (<http://www.jpsoft.com>)) gestartet werden:

```
cl test.cpp
```

Die zahlreichen Parameter erhält man mit der Option */?* (z.B. */D* für Präprozessor-Makros und */I* für *#include*-Suchpfade).

Eine solche Konsole kann man auch von Windows 10 aus mit einer „Developer Eingabeaufforderung“ starten:



Aufgabe 1.2

Erstellen Sie wie im Text ein Projekt mit dem Namen *MeinErstesCppProjekt*, das eine Header-Datei mit einer Funktion enthält (z.B. *plus1*), die in der *main*-Funktion mit einem Argument aufgerufen wird. Das Ergebnis von *plus1* soll dann auf der Konsole angezeigt werden.

Alle Lösungen der Übungsaufgaben in diesem Buch können in eine Header-Datei eines Projekts aufgenommen werden, das so aufgebaut ist.





1.3 Der Quelltexteditor

Der Quelltexteditor (kurz: **Editor**) ist das Werkzeug, mit dem die Quelltexte geschrieben werden. Er ist in die Entwicklungsumgebung integriert und kann z.B. über über *Ansicht|Code* aufgerufen werden.

1.3.1 Tastenkombinationen

Der Editor enthält über Tastenkombinationen zahlreiche Funktionen, mit denen sich nahezu alle Aufgaben effektiv durchführen lassen, die beim Schreiben von Programmen auftreten.

Die nächste Tabelle enthält Funktionen, die vor allem beim Programmieren nützlich sind, und die man in einer allgemeinen Textverarbeitung nur selten findet. Die meisten dieser Optionen werden auch unter *Bearbeiten*|*Erweitert* sowie auf der *Text Editor* Symbolleiste (unter *Ansicht*|*Symbolleisten*) angezeigt:

Tastenkürzel	Aktion oder Befehl
<i>F5</i> bzw. 	kompilieren und starten, wie <i>Debuggen-Debugging Starten</i>
<i>Umschalt+F5</i>	Laufendes Programm beenden, wie <i>Debuggen-Debugging beenden</i> . Damit können auch Programme beendet werden, die mit  nicht beendet werden können. Versuchen Sie immer zuerst diese Option wenn Sie meinen, Sie müssten Visual Studio mit dem Windows Task Manager beenden.
<i>F1</i>	kontextsensitive Hilfe (siehe Abschnitt 1.5)
<i>Strg + `</i> (` ist das Zeichen links von der Rücktaste)	setzt den Cursor vor die zugehörige Klammer, wenn er vor einer Klammer (z.B. (), {}, [] oder <>) steht
<i>Umschalt+Strg + `</i>	markiert den Bereich zwischen den Klammern außerdem noch
<i>Strg+M+M</i> bzw. unter <i>Bearbeiten</i> <i>Gliedern</i>	ganze Funktionen, Klassen usw. auf- oder zuklappen
<i>Alt+Maus</i> bewegen bzw. <i>Alt+Umschalt+Pfeiltaste</i> (←, →, ↑ oder ↓)	zum Markieren von Spalten, z.B. <pre> /// <summary> /// Clean up any /// </summary> /// <param name= </pre>
<i>Strg+K+C</i> oder <i>Strg+K+U</i> bzw.  oder 	einen markierten Block auskommentieren bzw. die Auskommentierung entfernen
Rechte Maustaste <i>Dokument öffnen</i>	öffnet in einer Zeile mit einer <i>#include</i> -Anweisung die angegebene Datei

Eine ausführliche Liste der Tastenkombinationen findet man in der MSDN-Dokumentation (siehe Abschnitt 1.5) mit dem Suchbegriff „Visual Studio Tastenkombinationen“.

Weitere Möglichkeiten des Editors, die gelegentlich nützlich sind:

- Die Schriftgröße kann man mit *Strg+Mausrad* vergrößern und verkleinern.
- Wenn man verschiedene Teile eines Textes in verschiedenen Fenstern anschauen (und z.B. vergleichen will), kann man die aktuelle Datei in verschiedenen Fenstern öffnen:

