Xpert.press

Moderne C++ Programmierung

Klassen, Templates, Design Patterns

Bearbeitet von
Ralf Schneeweiß

1. Auflage 2012. Buch. xiii, 393 S. Hardcover
ISBN 978 3 642 21428 8
Format (B x L): 15,5 x 23,5 cm
Gewicht: 771 g

Kapitel 2
Die Sprache C++

C++ ist inzwischen eine relativ alte Programmiersprache, die viele Brüche in ihrer Geschichte erlebt hat. Vieles, was man heute in C++ findet, ist älteren Programmiermethoden und Ideen zu verdanken, die heute nicht mehr unbedingt aktuell sind. Modernere objektorientierte Sprachen, wie zum Beispiel JAVA, sind in ihrer syntaktischen Struktur weit weniger komplex, da sie weniger unterschiedlichen Paradigmen[1] folgen müssen. In den folgenden Abschnitten soll ein Überblick über die verschiedenen Phasen der Entwicklung von C++ gegeben werden. Damit sollen auch die unterschiedlichen Ideen umrissen werden, die bis heute in der Sprache C++ stecken, ihre Grundlage bilden und sie einerseits mächtig und andererseits an manchen Stellen auch unübersichtlich machen. Für den C++-Entwickler stellt sich immer die schwierige Frage der Auswahl geeigneter Methoden für seine aktuelle Arbeit. Dabei muss zwischen konkurrierenden Leitideen ausgewählt werden und es müssen solche verworfen werden, die nicht mehr aktuell sind.

2.1 Geschichte und Paradigmenwandel

C++ entstand in den frühen 80er Jahren aus der Programmiersprache C heraus. Die Sprache wurde in den Bell Laboratories, die später von AT&T übernommen wurden, entwickelt. Federführender Ideengeber und Entwickler war Bjarne Stroustrup, der diese Rolle über lange Jahre beibehielt. Aus C wurde zunächst ein „C mit Klassen“ entwickelt, um die damals neuen Methoden der Objektorientierung den C-Programmierern zugänglich zu machen. Später, 1983, wurde erstmals der Name „C++“ für die neue Sprache verwendet. Er setzt sich zusammen aus dem Namen „C“ und dem Inkrementoperator „++“ aus C, um anzudeuten, dass C++ über C quasi einen Schritt hinausgeht. Die Objektorientierte Programmierung steckte zu damaliger Zeit noch in den Kinderschuhen. Die Methodenausstattung dieser neuen Leitidee war nach heutigem Maßstab noch etwas dürftig und erst

[1] Paradigma: Leitidee.

R. Schneeweiß, *Moderne C++ Programmierung*, Xpert.press,
DOI 10.1007/978-3-642-21429-5_2, © Springer-Verlag Berlin Heidelberg 2012

in Entwicklung begriffen. Die Sprachkonstrukte von C++ wurden natürlich an den Stand der damaligen Sofwaretechnik angepasst. Mit der fortschreitenden Technik der OO-Programmierung bekam C++ auch neue sprachliche Strukturen. Die Firma AT&T, die die Bell Laboratories übernahm, standardisierte C++ mit einem eigenen Firmenstandard, der von fremden Compilerherstellern übernommen werden konnte. Die meisten Hersteller von C++-Entwicklungssystemen machten sich diese AT&T-Standards zunutze und unterstützten sie. Natürlich boten die verschiedenen Hersteller auch spezifische Features ihrer Compiler an, die über den jeweils gültigen Stand der AT&T-Quasi-Standards hinausgingen. Die wesentlichen Standards von AT&T waren durch die Compiler 1.0, 1.1, 2.0, 2.1 und 3.0 vorgegeben. Die jeweilige aktuelle Standardimplementierung wird bis heute *cfront* genannt. Ein cfront-Compiler ist also konform zum aktuellen Standard. Seit Anfang der 90er Jahre wurde die Standardisierung durch das American National Standardisation Institute angestrebt.

Das für C++ zuständige Komitee brauchte allerdings bis 1998, um einen Standard zu verabschieden. In der Folge der ANSI-Standardisierung wurde der Standard auch zur internationalen ISO-Norm erklärt. Was war nun in den verschiedenen Standardisierungsschritten mit C++ passiert?

In den AT&T Standards der 1er Versionen bekam C++ vor allem die Sprachmerkmale, die zur Kapselung von Daten in Klassen notwendig sind. In der objektorientierten Entwicklung der damaligen Zeit bis 1986 standen in besonderer Weise die Gedanken der Datenkapselung und der Funktionsüberladung im Vordergrund. Vererbung wurde eher zur Anhäufung von Daten und Funktionalität verwendet, um eine Art Wiederverwendung gemeinsamer Funktionalität zu erreichen. Die dafür geschaffenen Sprachstrukturen bestehen in C++ bis heute, auch wenn wir die Vererbung nicht mehr in dem damaligen Sinne anwenden, denn sie hat sich als Sackgasse erwiesen. Dass die Vererbung in C++ eine Sichtbarkeit besitzt[2] und diese auch noch privat ist, hat den beschriebenen Hintergrund.

Mit der AT&T-Version 2.0 fanden die virtuellen Funktionen Einzug in die Sprache C++ und mit ihnen das Konzept der Polymorphie. Man kann daher sagen, dass im Grunde genommen erst diese Version der Sprache dem ähnlich wurde, was wir heute als C++ kennen. Mit der Polymorphie als zentralem Konzept der Objektorientierung wurde C++ überhaupt erst zu einer OO-Sprache. Vorher war das, was man C++ nannte, ein erweitertes C mit etwas besserer Typenkontrolle, anderen Kommentarzeichen und der Möglichkeit Daten zu verstecken. Jetzt erst konnten OO-Entwürfe mit C++ verwirklicht werden, die auf die späte Bindung von Methoden aufbauten und damit den Funktionsfokus zugunsten des Objektfokus aufgaben. OO war in C++ angekommen, wenngleich es an der einen oder anderen Stelle noch etwas hakte. So konnten virtuelle Methoden in Basisklassen deklariert werden, um die Polymorphiebasis für abgeleitete Klassen bereitzustellen. Diese virtuellen Methoden mussten formal aber immer implementiert werden, was inhaltlich natürlich wenig sinnvoll war und den einen oder anderen Entwickler noch

[2] Siehe Abschn. 2.2.21 auf S. 73

zu Fehlentscheidungen verführte („Was die Sprache erzwingt, kann ja so falsch nicht sein! Oder?“).

Erst 1992 in der AT&T-Version 3.0 bekam C++ den OO-Rundschliff. Dazu gehörten in allererster Linie die rein virtuellen Funktionen ohne Implementierung. Dieses sehnlichst erwartete und in Newsgroups vorher schon diskutierte Sprachfeature machte nun allen Entwicklern deutlich, dass es Klassen ohne eigene Funktionalität geben kann. Klassen, die eine reine Schnittstellenfunktion haben. Es gab auch noch Versionen bis 3.3, die aber hauptsächlich Elemente einführten, die mit dem OO-Paradigma nichts mehr zu tun hatten. So wurden die Templates und das Exception Handling in den 3er Versionen von C++ aufgenommen. Anfang der 90er Jahre gab es in der Entwicklergemeinde eine Diskussion um die Anwendung der neuen Templatetechnik. So gab es schon Stimmen, die die Templates befürworteten, da sie doch den Compiler in die Lage versetzten, Code zu erzeugen, statt nur zu übersetzen. Andere befürchteten unbeherrschbare Nebeneffekte – z. B. Aufblähung des erzeugten Codes – und eine zunehmende Komplexität der Sprache. Man war schließlich zu einem gesicherten Wissensstand über die Objektorientierung gelangt, da sollte das Instrumentarium nicht durch völlig fremde Konzepte aufgeweicht oder gestört werden. Insbesondere am Design der für C++ längst überfälligen Containerbibliothek entzündete sich die Diskussion. Dynamische Datencontainer werden in C++ benötigt um 1-zu-n-Relationen zwischen Klassen und Objekten zu modellieren. In den C++-Standardbibliotheken der AT&T-Standards waren keine Containerbibliotheken enthalten, weshalb Compilerhersteller eigene herstellerspezifische Bibliotheken auslieferten. Software, die diese herstellerspezifischen Implementierungen nutzte, war natürlich nicht mehr portierbar, was dazu führte, dass häufig für jede spezielle Software extra Container implementiert wurden. In dieser Zeit entstanden sehr viele – sehr viele! – verschiedene, mehr oder weniger gut funktionierende Container (in jedem Softwareprojekt die eigene verkettete Liste). Solche Container können mit reinen OO-Methoden entwickelt werden, oder aber auch mit den generischen Techniken, die durch die Templates in die Sprache eingeführt wurden. Den Streit entschied A. Stephanov[3] mit seiner Standard Template Library zugunsten der generischen Technologie. Diese Bibliothek beruht ganz auf dem Einsatz der C++-Templates und war frei verfügbar. Die STL ist gegenüber entsprechenden OO-Containerbibliotheken von überzeugender Eleganz und Flexibilität. Ein weiterer Vorteil ist die gute Beherrschbarkeit der Laufzeitaspekte beim Einsatz der Bibliothek. Dies alles zusammengenommen führte zunächst dazu, dass die generischen Templatetechniken anerkannt wurden.

Eine weitere Folge war die Aufnahme der STL in den ANSI/ISO-Standard. Die Arbeiten um die Standardisierung veränderten die STL nicht wesentlich. Die STL ist seitdem in leicht angepasster Form Bestandteil der C++-Standardbibliothek. Außerdem verdrängten generische Technologien ihre OO-Pendants aus einigen Bereichen der Bibliothek. Mit gewisser Berechtigung kann man sagen, dass mit der Integration der Templatetechnologie in C++ nun Methoden zur Verfügung stehen,

[3] Der Autor der STL, Alexander Stephanov, wurde durch Hewlett Packard und SGI für deren Entwicklung unterstützt. Über Server bei HP stand die STL schon 1994 zum Download bereit.

die die lang versprochene Wiederverwendbarkeit Wirklichkeit werden lassen. Die OO-Methoden sind dazu nur marginal in der Lage. Es kam auch das Konzept der Namensräume mit in den Standard. Außerdem wurden neue Typenkonvertierungen definiert. Die Templatetechniken wurden mit dem ANSI/ISO-Standard weiter verfeinert und die syntaktischen Möglichkeiten nahmen zu. Sie haben mit dem Standard eine Komplexität erreicht, die das bloße Erlernen der Syntaxregeln enorm erschweren. Dazu kommt noch, dass mit der Verabschiedung des Standards praktisch kein Hersteller in der Lage war, einen Compiler mit korrekter Standardkonformität zu liefern. Lange Zeit stand praktisch nur ein Experimentalcompiler, der Comeau-C++-Compiler, zur Verfügung. Der erste nahezu standardkonforme Produktivcompiler war der Metrowerks Code Warrior, der vor allem für die Macintosh-Plattform verwendet wurde. Eine ähnlich gute Unterstützung des Standards erreichte der GCC-C++-Compiler mit seinen 3er Versionen. Erst mit den stabilisierten Versionen dieses Compilers 2005 stand eine standardkonforme Entwicklungsplattform in der Breite zur Verfügung. Es kommt noch eine weitere Schwierigkeit für die Anwendung des Standards hinzu: in den ANSI/ISO-Standard wurde Exception Handling als integraler Bestandteil aufgenommen. Dabei wurde der `new`-Operator zur Speicherallokation neu definiert. In den AT&T-Versionen von C++ liefert der `new`-Operator einen Nullzeiger zurück, wenn die Allokation fehlschlägt. In der ANSI-Version wirft `new` eine Exception, um einen Fehlschlag anzuzeigen. Diese Änderung ist derart tiefgreifend, dass eine Abwärtskompatibilität standardkonformer Compiler nicht mehr gegeben ist. Wenn Softwareprojekte alte Codeteile verwenden, kann damit der Standard nicht zur Anwendung kommen, da sonst die alte Fehlerbehandlung deaktiviert wird und es zu undefinierten Laufzeitzuständen kommen kann – fast schon zwangsläufig kommen muss. Alter und neuer Code vertragen sich nicht. Alter Code muss zunächst umgeschrieben werden, um mit neuem standardkonformen Code zu harmonisieren[4]. Viele erfolgreiche Bibliotheken sind nach einem AT&T- oder verwandten Standard geschrieben und werden weiter verwendet[5]. Ein weiteres Problem besteht in den Gewohnheiten der Entwickler. Um standardkonform zu entwickeln müssen sie Exception Handling als integralen Bestandteil ihrer Software begreifen, was einen Bruch mit alten Gewohnheiten und ganz neue Programmstrukturen erfordert. Diese Gründe zusammengenommen verhindern acht Jahre nach der Verabschiedung des Standards immer noch seine breite Akzeptanz.

Heute ist schon die nächste Version des ISO-Standards in Diskussion. Dabei wird auch die Aufnahme von freien Bibliotheken, wie z. B. der boost-Library diskutiert. Zu hoffen bleibt allerdings, dass Verhaltensweisen des Sprachkerns nicht mehr redefiniert werden, wie es bei der letzten Standardisierung geschah, da das einer zügigen Annahme eines Standards erneut im Wege stehen könnte.

---

[4] Es ist im ANSI/ISO-Standard zwar ein Migrationspfad für die alte Anwendung des new-Operators vorgesehen – die „nothrow"-Variante –, jedoch ist eine Portierung alten AT&T-konformen Codes deutlich aufwändiger als die reine Anpassung der Verwendungen des Operators new.

[5] Ein prominentes Beispiel dafür ist sicher die MFC.

2.2 Grundlagen

C++ ist eine reine Compilersprache. Der Quelltext muss durch einen Compiler in ausführbaren Code übersetzt werden, damit ein Programm ablaufen kann. Der Quelltext wird für den Ablaufvorgang der Software nicht mehr gebraucht. Dieses traditionelle Konzept teilt C++ mit älteren prozeduralen Sprachen wie Fortran, Pascal und natürlich auch C. Von C hat C++ die typische Zusammenarbeit zwischen Compiler und Linker geerbt. Nicht jede Compilersprache braucht einen separaten Linker. Das C-Konzept der zweistufigen Bearbeitung bringt jedoch eine sehr flexible Möglichkeit zur Modularisierung von Softwareprojekten mit sich. In der ersten Stufe übersetzt der Compiler ein Quelltextmodul zu einer Objektdatei. In der zweiten Stufe bindet der Linker verschiedene Objektdateien – und damit die Funktionen der verschiedenen Module – zur lauffähigen Software zusammen. C++ wird ebenso durch einen Compiler zu Objektdateien übersetzt, die später durch einen Linker gebunden werden. Dadurch lässt C++ große Freiheiten bei der Aufteilung eines Softwareprojektes in Module. Technisch ist die Flexibilität der Modularisierung eines Softwareprojektes sehr hoch. Es bleibt dem Entwickler überlassen, wie er mit dieser Flexibilität umgeht. In Abschn. 6.1 auf S. 331 wird auf die Modularisierung näher eingegangen. Zunächst sollen jedoch einfache Beispiele gegeben werden, bei welchen die Modulaufteilung noch keine Rolle spielt.

Ein einfaches erstes Beispiel zum Einsatz des Compilers

Fangen wir also mit einem ganz einfachen, für einen Anfang mit einer Programmiersprache sehr typischen „Hello world“ an:

```
#include <iostream>

int main()
{
  std::cout << "Hello world" << std::endl;
  return 0;
}
```

Listing 2.1 Erstes Beispiel

Wenn man mit einem beliebigen Texteditor diesen Text erfasst und unter einem beliebigen Namen mit der Endung `.cc` oder `.cpp` abspeichert, hat man schon einen echten C++-Quelltext. Schreiben Sie zum Beispiel „`prg1.cpp`“. Der Quelltext stellt ein einfaches Programm dar, das die Ausgabe „Hello world“ auf die Standardausgabe schreiben soll. Das Beispiel kann also unter einer textorientierten Shell zum Laufen gebracht werden. Sie brauchen eine beliebige Textshell unter UNIX, Linux oder MacOS. Wenn Sie mit Windows arbeiten, nehmen Sie einfach die DOS-Eingabeaufforderung. Natürlich muss ein C++-Compiler auf dem System installiert sein, damit das Programm übersetzt werden kann. Wichtig ist auch, dass

der Compiler aus der Kommandoumgebung, in der Sie arbeiten, erreichbar ist. Auch bei einem installierten Compilersystem müssen nicht immer alle wichtigen Pfade systemweit gesetzt sein. Manche Hersteller legen dazu eine kleine Batch-Datei ihrem Compiler bei, damit die Systempfade damit gesetzt werden können. Dazu muss die Batch natürlich innerhalb der Kommandoumgebung aufgerufen werden. Für manche Compiler muss man eine solche Batch-Datei auch selbst schreiben. Da es von Hersteller zu Hersteller sehr unterschiedlich sein kann, was vorhanden ist und was nicht, und da es darüber hinaus noch auf jedem Betriebssystem andere Verfahren der Batcherstellung gibt, kann dieses Installationsproblem an dieser Stelle nicht hinreichend behandelt werden. Dazu nur drei Regeln (wobei `compiler_root` für das Verzeichnis steht, in dem Ihr Entwicklungssystem installiert wurde):

1. Der Compiler und der Linker sind ausführbare Tools, die im Suchpfad der Umgebung gefunden werden müssen. Normalerweise `compiler_root/bin`.
2. Der Compiler muss das Verzeichnis mit den Includedateien finden. Normalerweise `compiler_root/include`.
3. Der Linker muss das Verzeichnis mit den Bibliotheksdateien finden. Normalerweise `compiler_root/lib`.

Manchmal gehen Compilerhersteller etwas unterschiedliche Wege, um ihren Tools die typischen Orte ihres Arbeitsmaterials mitzuteilen. Das können Umgebungsvariablen sein oder auch Konfigurationsdateien im Compilerverzeichnis. Wenn Sie noch nie mit einem C- oder C++-Compiler zu tun hatten, sollten Sie die erste Installation am besten von jemandem durchführen lassen, der es mindestens schon einmal gemacht hat. Gehen wir also von einem installierten und in Ihrer Kommandoumgebung bekannten Entwicklungssystem aus. Im Allgemeinen ruft man den Compiler auf und übergibt ihm den Quelltextnamen in der Kommandozeile. Also:

```
%% g++ prg1.cpp
```

für den GNU-Compiler unter Linux oder UNIX.

```
%% bcc32 prg1.cpp
```

für den Borland-C++-Compiler unter Windows.

```
%% cl /GX prg1.cpp
```

für den MS-Visual C++-Compiler unter Windows.

Auf manchen Systemen wie zum Beispiel QNX heißt der Aufruf des C++-Compilers `CC`. Also: `CC prg1.cpp`.

Dabei sollte nun auf den Unices eine Datei `a.out` entstanden sein. Bei den Windows-Compilern müsste eine Exedatei `prg1.exe` entstanden sein. Diese Dateien sind ausführbar und können direkt von der Kommandozeile aus gestartet werden.

Also:

```
%% ./a.out
```

für Unix und Linux bzw.

```
%% prg1
```

für Windows.

Wir haben den Aufruf des Compilers kennen gelernt, der eine ausführbare Datei produziert. Ganz ist dieser Satz nicht richtig, denn irgendwie muss ja noch der in den vorangegangenen Abschnitten beschriebene Linker mit von der Partie sein. In unserem ersten Beispiel haben wir den Linker mit dem Compiler aufgerufen. Die genannten Compilerimplementierungen vereinfachen den Einstieg dadurch, dass bei einem direkten Aufruf des Compilers nach dessen Durchlauf noch der Linker gestartet wird, um die Arbeit zu Ende zu bringen. Wenn man die beiden Arbeitsschritte voneinander entkoppeln möchte – oder muss –, kann man das dem Compiler mit einem Kommandozeilenargument mitteilen. Die entsprechenden Outputs des reinen Compileprozesses finden sich meistens auch bei einem integrierten Compiler-Linker-Lauf im aktuellen Verzeichnis. Sie tragen die Endung `.obj` bei den meisten Windows-Compilern und `.o` bei den Compilern der Unices. Auch diese Zwischenprodukte werden für den Ablauf des Programms nicht mehr gebraucht.

Um dieses erste Programmbeispiel nicht ganz unerklärt zu lassen, sollen die einzelnen Zeilen noch kurz besprochen und entsprechende Verweise auf spätere Kapitel eingefügt werden. Mit der Zeile „`#include <iostream>`“ haben wir eine typische Präprozessoranweisung, wie sie in Abschn. 2.2.2 erklärt ist. Die Includeanweisung fügt die Datei „`iostream`“ in den von uns geschriebenen Programmquelltext ein. Wo er die Datei `iostream` findet? Sie befindet sich im Allgemeinen im Includeverzeichnis des Compilers. Was es mit dieser Datei auf sich hat, wird in Abschn. 5.2.1 auf S. 268 beschrieben. An dieser Stelle sei nur gesagt, dass darin die Definitionen für `std::cout` und `std::endl` zu finden sind. Die Zeile „`int main()`“ und der dazugehörige Block, der durch die geschweiften Klammern begrenzt wird, definiert die Hauptfunktion des Programms. Die Funktion heißt `main`, das Wörtchen `int` zeigt an, dass die Hauptfunktion einen ganzzahligen Wert zurück gibt (in diesem Fall an das aufrufende System), und die einfachen Klammern zeigen an, dass es sich bei `main` überhaupt um eine Funktion handelt. Dazu mehr in Abschn. 2.2.16 auf S. 48. Jedenfalls springt die Programmausführung vom Betriebssystem aus zuallererst in die Funktion `main()` und führt die darin enthaltenen Anweisungen aus. Im Beispielprogramm sind das nur zwei. Die Ausgabe auf der Konsole „`std::cout << "Hello world" << std::endl;`“ und die Returnanweisung der Funktion „`return 0;`“, die vereinbarungsgemäß etwas – hier die Null – zurückliefert. Vereinbarungsgemäß deshalb, weil die Funktion `int main()` ja einen ganzzahligen Rückgabewert in ihrer Deklaration verspricht. Momentan sehen wir von dieser Werterückgabe an das Betriebssystem nichts.

Da jetzt die grundsätzliche Funktionsweise des Compiler-Linker-Gespanns geklärt ist, soll nun der Compilerlauf in seinen Einzelheiten genauer betrachtet werden. Das nächste Kapitel beschreibt die sprachlichen Möglichkeiten, den Quelltext selbst mit Hilfe des Compilers zu manipulieren.

2.2.1 Bezeichner in C++

Es gibt in C++ Schlüsselworte und Operatoren, die den Sprachumfang darstellen. Alles andere in einem Programm muss man selbst definieren und benennen. Auch wenn noch nicht eingeführt wurde, was die Dinge für eine Bedeutung haben, so sollen sie hier doch schon einmal aufgezählt werden: Bezeichnen muss man Konstanten, Variablen, Funktionen, Namensräume, Strukturen, Enums, Unions, Klassen und Klassenmethoden. Welche Namen dafür gewählt werden, ist weitgehend dem Programmierer überlassen. Natürlich sollte er solche Namen finden, die im jeweiligen Zusammenhang einen „Sinn" ergeben. Ganz frei ist er aber beim Erfinden der Bezeichner nicht. Es gibt ein paar Regeln, die für alle Bezeichner gelten, die in einem Programm eingeführt werden können:

1. Bei Bezeichnern in C++ werden Groß- und Kleinschreibung unterschieden. Das heißt, sie sind case-sensitiv. Die beiden Bezeichner `Prozentwert` und `prozentwert` bezeichnen also unterschiedliche Dinge.
2. Bezeichner bestehen aus Buchstaben und können auch Ziffern enthalten.
3. Beginnen muss ein Bezeichner immer mit einem Buchstaben des lateinischen ASCII-Zeichensatzes oder mit dem Unterstrich _. Der ASCII-Zeichensatz umfasst das Alphabet ohne nationale Sonderzeichen. Also `a`, `b`, `c`, `d`, `e`, `f`, `g`, `h`, `i`, `j`, `k`, `l`, `m`, `n`, `o`, `p`, `q`, `r`, `s`, `t`, `u`, `v`, `w`, `x`, `y`, `z` und deren groß geschriebene Varianten `A` – `Z`. Die deutschen Sonderzeichen `ä`, `ö`, `ü` und `ß` sind beispielsweise nicht erlaubt. Das gleiche gilt für die dänischen, die isländischen usw.
4. Erst nach dem ersten Zeichen dürfen auch Ziffern verwendet werden. Also `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` und `9`. Der Bezeichner `a42` ist ein gültiger Bezeichner. Dagegen ist `42a` nicht gültig.
5. Alle Zeichen in einem Bezeichner sind signifikant. Sie werden also zur Erkennung und Unterscheidung herangezogen. Es gibt keine Größenbegrenzung[6].
6. Schlüsselworte dürfen nicht als Bezeichner verwendet werden.
7. Es gibt noch einige andere durch den Compiler vordefinierte Bezeichner, die nicht verwendet werden dürfen. So sind beispielsweise einige Konstanten oder Makros durch den Compiler vordefiniert.

Bei der Wahl von geeigneten Bezeichnern müssen die vorangegangenen Regeln beachtet werden. Die Regeln sind allerdings so geartet, dass die Einhaltung einigermaßen intuitiv erfolgen kann. Es kommt selten vor, dass man mit ihnen in Konflikt gerät.

[6] Das war in älteren C++-Dialekten einmal anders. Dort galten 32 oder 256 Stellen im Bezeichner als signifikant. Alles was darüber hinausging wurde einfach abgeschnitten. Auch heute sind noch solche Compiler im breiten Einsatz, die nach diesem alten Verfahren arbeiten. Natürlich gelten auch bei Compilern, die dem ANSI/ISO-Standard entsprechen, technische Grenzen. Ein Bezeichner sollte also nicht den ganzen Hauptspeicher füllen.

2.2.2 Der Präprozessor

Der Präprozessor ist eine Phase des Compilers, die Textersetzungen am Quelltext durchführt. Diese erste Phase hat mit dem eigentlichen Parsen des C++-Codes noch nichts zu tun. Sie ist dem eigentlichen Compilevorgang zeitlich vorgeschaltet und man kann mit ihr den Quelltext für den eigentlichen Compiliervorgang vorbereiten. So kommt es beispielsweise oft vor, dass man Code portabel auf mehreren Plattformen lauffähig hält, indem man Teile, die nur für eine bestimmte Plattform Gültigkeit besitzen, durch Präprozessoranweisungen für eben diese Plattform inkludiert. Andere Teile können durch den Präprozessor herausgeschnitten werden. Es gibt viele Gründe dafür, warum man Textmanipulationen am Quelltext durch den Compiler durchführen lässt.

Nur die Präprozessorphase kann man für solche Textoperationen nutzen. Man programmiert sie mit speziellen Anweisungen, den sogenannten Präprozessordirektiven[7]. Die Präprozessorphase gibt es auch in der Sprache C und sie wurde von dort nach C++ übernommen (die Präprozessorbefehle unterscheiden sich auch zwischen den zwei Sprachen kaum). Man erkennt die Präprozessordirektiven an der Raute # vor dem Schlüsselwort. Es gibt nicht viele verschiedene Präprozessordirektiven. Sie lassen sich in der folgenden Liste zusammenfassen:

```
#define
#undef
#if
#ifdef
#ifndef
#elif
#else
#endif
#include
#line
#error
#pragma
```

Dazu kommen noch die beiden Operatoren # und ##, die für die Bildung von symbolischen Namen gebraucht werden.

Die Präprozessordirektive `#define`

Eine sehr einfache Anwendung des Präprozessors findet sich in der Definition von Konstanten durch die `#define`-Anweisung.

```
#define MAXIMUM    1000000
```

[7] In den meisten Fällen ist der Präprozessor ein eigenes Tool, das durch den Compiler selbst aufgerufen wird. Daher kann man einen solchen Präprozessorlauf auch ohne einen ganzen Compilerlauf anstoßen, um die Ergebnisse bestimmter Präprozessordirektiven zu überprüfen.

Diese Zeile stellt für den Präprozessor die Anweisung dar, alle Zeichenketten im nachfolgenden Text, die „MAXIMUM“ lauten, durch die Zeichenkette „1000000“ zu ersetzen. Damit wird eine symbolische Konstante geschaffen, die an beliebig vielen Stellen des Quelltextes verwendet werden kann. Vor dem eigentlichen Parsevorgang des Compilers ersetzt der Präprozessor die symbolische Konstante MAXIMUM durch die numerische 1000000. Die #define-Anweisung kann noch wesentlich mehr, als nur Konstanten zu definieren. Mit ihr lassen sich Makros definieren, die in Abschn. 2.2.19 auf S. 56 erklärt werden.

Die Präprozessordirektive #include ..

Ebenso leicht ist die #include-Anweisung zu verstehen. In fast allen C- und C++-Programmen findet man am Anfang der Quellcodedateien diese Art der Präprozessordirektiven:

```
#include <iostream>
```

Mit dieser Anweisung fügt der Präprozessor die Textdatei, die in den spitzen Klammern genannt ist, in den aktuellen Code ein. Im Folgenden wird diese Textdatei „Includedatei“ oder „Headerdatei“ genannt. Es wird dabei einfach der Text der Quellcodedatei um den Text der Includedatei[8] an der aktuellen Stelle erweitert. Es ist wirklich nichts Weiteres. Dem Compiler ist es an der Stelle der Includeanweisung vollkommen egal, was in der einzufügenden Includedatei steht. Die Wirkungsweise ist einfach auf die Einfügung des Textes beschränkt, der in der aufgeführten Datei enthalten ist. Spätere Compilephasen nach dem Präprozessorlauf überprüfen das Eingefügte dann syntaktisch. Die spitzen Klammern, die den Dateinamen umschließen, haben die Wirkung, dass der Compiler in einem vorher eingestellten Pfad nach der angegebenen Datei sucht. In diesem Pfad stehen für gewöhnlich die Bibliotheksheaderdateien. Beim Einsatz mehrerer Bibliotheken kann man den Pfad erweitern. Das geschieht in Abhängigkeit vom eingesetzten Entwicklungssystem in Konfigurationsdateien oder in Umgebungsvariablen. Dem Compiler selbst kann man auch einen Parameter übergeben, der den genannten Pfad setzt. Dieser Kommandozeilenparameter beginnt für gewöhnlich mit -I. Möchte man eine Headerdatei vom gleichen Verzeichnis einfügen, aus dem auch die Quellcodedatei stammt, in der die Includeanweisung steht, so verwendet man Anführungszeichen.

```
#include "Definitionen.h"
```

Von diesem Wurzelverzeichnis des aktuellen Compiliervorgangs kann man auch relative Pfadangaben machen.

```
#include "../../mylibs/include/Definitionen.h"
```

Die Headerdateien haben für gewöhnlich die Endung .h, .hpp oder .hxx. Die Endung .h war schon in der Sprache C üblich und wurde für C++ beibehalten.

[8] Das ist im Prinzip eine normale Quellcodedatei. Allerdings enthält sie für gewöhnlich nur bestimmte Definitionen. In folgenden Abschnitten wird näher darauf eingegangen werden.

Manchmal möchte man die Dateien als C++-Quelltexte kenntlich machen und verwendet dabei eine der beiden anderen Varianten. Notwendig ist das nicht, dem Compiler ist es egal.

In ANSI-C haben auch alle Bibliotheksheaderdateien die Endung `.h`. In den alten AT&T-Standards für C++ wurden die Dateiendungen beibehalten. Erst mit der ANSI/ISO-Standardisierung wurden die Dateiendungen für Bibliotheksheader gestrichen. Das hat den Grund, dass man C++ für so viele Systeme einsetzbar gestalten wollte wie möglich. Dabei kann man sich nicht immer auf das Vorhandensein von Dateisystemen verlassen, die Dateierweiterungen kennen.

Die Präprozessordirektiven `#if`, `#else`, `#elif` und `#endif`

Die Anweisungen `#if`, `#else`, `#elif` und `#endif` werden zur so genannten bedingten Compilierung verwendet. Mit ihnen findet ein „Zurechtschneiden" des Quelltextes im Präprozessorlauf statt. Bevor der Quelltext der eigentlichen syntaktischen Überprüfung unterzogen wird, werden Teile aktiviert bzw. deaktiviert, indem sie eingefügt oder ausgeschnitten werden. Dazu werden mit Hilfe dieser Präprozessordirektiven Konstanten abgefragt, die zur Compilezeit schon feststehen.

```
#if DEBUG == 1
  void TraceOut( char *txt )
  {
    std::cerr << txt << std::endl;
  }
#elif DEBUG == 2
  void TraceOut( char *txt )
  {
    writeToFile( "TRACES.TXT", txt );
  }
#else
  inline void TraceOut( char * )
  {
  }
#endif
```

Listing 2.2 Verzweigungen im Präprozessorlauf

In dem vorangegangenen Beispiel werden drei Quellcodeabschnitte zur Auswahl gegeben. Je nach dem Wert der Konstanten `DEBUG` wird einer der drei Abschnitte gewählt und übersetzt. Die anderen Abschnitte sondert der Präprozessor schon vor dem eigentlichen Übersetzungsvorgang aus.

Die Anwendung der Präprozessoranweisung `#if` zieht nicht zwingend `#else` oder `#elif` nach sich. Beide sind optional.

Die Präprozessordirektiven #ifdef und #ifndef

Die beiden Anweisungen #ifdef und #ifndef sind erweiterte Versionen der Anweisung #if. Sie fragen ab, ob ein Bezeichner durch den Präprozessor definiert wurde oder nicht. Man kann sie auch mit der Grundform umschreiben.

```
#ifdef BIGENDIAN
 #define HIGHBYTE RIGHTBYTE
 #define LOWBYTE  LEFTBYTE
#else // LITTLEENDIAN
 #define HIGHBYTE LEFTBYTE
 #define LOWBYTE  RIGHTBYTE
#endif
```

Listing 2.3 Verzweigungen im Präprozessorlauf anhand einer Definition

In dem Beispiel in Listing 2.3 werden zwei Definitionen in Abhängigkeit einer schon existierenden Definition unterschiedlich definiert. Solch ein Beispiel ist durchaus realistisch. Es bleibt aber noch ganz im Bereich der Präprozessordirektiven. Mit der Anweisung #if kann man das Beispiel folgendermaßen umformulieren:

```
#if defined(BIGENDIAN)
// ...
```

Listing 2.4 Die Anwendung des Präprozessoroperators defined

Das nächste Beispiel präpariert ein Stückchen Code unterschiedlich, je nachdem, ob es von einem C- oder einem C++-Compiler verarbeitet wird.

```
#ifdef __cplusplus
extern "C" {
#endif

int f()
{
  // ...
}

#ifdef __cplusplus
}
#endif
```

Listing 2.5 Beispiel: bedingte Compilierung

Auch dieses ist ein sehr realistisches Beispiel. Bei C++-Compilern ist die symbolische Konstante __cplusplus vordefiniert (siehe Abschn. 2.2.2 auf S. 19).

Das führt in dem Beispiel dazu, dass für einen C++-Compilevorgang zwei kleine Codeschnipselchen mehr compiliert werden als für C[9].

Etwas anders ausgedrückt, kann man den Effekt auch so erreichen:

```
#ifndef EXTC
 #ifdef __cplusplus
  #define EXTC extern "C"
 #else
  #define EXTC
 #endif
#endif // EXTC

EXTC int f()
{
    // ...
}
```

Listing 2.6 Beispiel: bedingte Compilierung

Die Anweisung `#ifndef` negiert die Bedeutung von `#ifdef` einfach. Das `n` steht für „not“. Das Beispiel in Listing 2.6 kann auch mit `#if` reformuliert werden:

```
#if !defined(EXTC)
 #if defined(__cplusplus)
  #define EXTC extern "C"
 #else
  #define EXTC
 #endif
#endif // EXTC
...
```

Listing 2.7 `#ifndef` anders ausgedrückt

Ein weiteres Beispiel demonstriert, wie man zur Compilezeit feststellen kann für welches System der Code compiliert wird. Visual C++ unter Windows definiert beispielsweise mit den Präprozessorkonstanten `_WIN32` und `_WIN64` ob die Applikation 32 oder 64-bittig übersetzt wird.

```
#if defined( _WIN64 )
 #define MAXBUFFERSIZE 128000
#else
 #define MAXBUFFERSIZE  32000
#endif
```

Listing 2.8 Detektion zur Compilezeit

[9] Die hier gezeigten bedingten Codefragmente `extern "C"` mit den dazugehörigen geschweiften Klammern werden für die Integration von C- und C++-Code gebraucht.

Die Direktive #undef macht eine Definition rückgängig. Man hebt mit der Anweisung #undef Dinge auf, die mit #define definiert wurden. Das sind symbolische Konstanten und Makros. Anzuwenden ist #undef wie eine #define-Anweisung.

```
#undef Bezeichner
```

Die Präprozessordirektive #error

Die Direktive #error führt zu einem sofortigen Abbruch des Compilevorgangs und gibt eine Fehlermeldung aus.

```
#if defined( _WIN64 )
 #define MAXBUFFERSIZE 128000
#elif defined( _WIN32 )
 #define MAXBUFFERSIZE  32000
#else
 #error Kein 32 oder 64-Bit System!
#endif
```

Listing 2.9 Abbruch des Compilerlaufs bei Fehlerfall

Diese #error-Direktive wird dazu verwendet, den Compilevorgang abzubrechen, wenn nicht die benötigten Bedingungen für eine erfolgreiche Übersetzung detektiert werden. Der Text in der Argumentenzeile der Anweisung wird dabei durch den Compiler auf der Standardausgabe ausgegeben.

Die Präprozessoranweisung #pragma

Die Anweisung #pragma ist zwar selbst eine Standarddirektive, ihre Funktion ist es jedoch, nichtstandardisierte Einstellungen spezifischer Compiler vorzunehmen. Die Argumente der Direktive sind also jeweils herstellerspezifisch. Wenn ein Compiler die Argumente einer #pragma-Direktive nicht erkennt, muss er sie ignorieren. Das ist eine Standardvorgabe. Welche Einstellungen durch Pragmas beeinflusst werden können, ist absolut abhängig von den technischen Notwendigkeiten eines Zielcompilers oder vom Erfindungsreichtum des Compilerherstellers. So erzwingt die folgende Pragmaanweisung beim MS-Visual C++ 6.0-Compiler, dass die Warnung mit der Nummer 164 als Fehler behandelt wird.

```
#pragma warning( error : 164 )
```

Für den gleichen Compiler hat die folgende Pragmaanweisung die Bedeutung, dass das Funktioninlining auf eine bestimmte Aufruftiefe begrenzt wird. Ruft also eine Inline-Funktion eine andere Inline-Funktion, und so weiter, werden diese Funktionen nur bis zur der Aufrufebene inline expandiert, die in der Pragma-

anweisung genannt wurde.

```
#pragma inline_depth( [0... 255] )
```

Diese Beispiele sollen zeigen, wie compilerabhängig Pragmaanweisungen sind. Will man sie für einen bestimmten Compiler verstehen, muss man seine Dokumentation zu Rate ziehen. Jeder Compiler hat einen großen Satz von möglichen Einstellungen und oft auch bestimmte Funktionen, die nur bei diesem anzutreffen sind.

So erzeugt beispielsweise die Quelltextzeile

```
#pragma keeka
```

beim Borland C++-Compiler Version 5.5.1 für Windows die folgende Ausgabe auf der Konsole:

```
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
keeka.cpp:
 /\__/\
| -  - |
* ^--^ *
 x=--=x
  |    |
|       |
| |  | |     .
| |__| |\___//
 vv  vv *---*

Keeka: Simply the best darn cat in the Universe.
```

Andere Compiler ignorieren dieses Pragma völlig.

Vordefinierte Makros und symbolische Konstanten

Bestimmte Makros und Konstanten sind durch den Compiler bereits vordefiniert. Sie können bei der Definition eigener Makros behilflich sein, ermöglichen die programmtechnische Aufnahme bestimmter Daten, die mit dem Compilevorgang in Verbindung stehen und unterstützen auch die Implementierung von Fehleranalyse- und Abbruchcodes.

Die standardmäßig vordefinierten Makros und Konstanten sind:

`__LINE__`	Zeilennummer als Dezimalkonstante.
`__FILE__`	Dateiname als Stringliteral.
`__DATE__`	Datum als Stringliteral im Format `mm dd yyyy`.
`__TIME__`	Zeit als Stringliteral im Format `hh:mm:ss`.
`__cplusplus`	In ANSI/ISO C++ von 1998 mit 199711L definiert. Wird in zukünftigen Standards wahrscheinlich entsprechend höhere Werte annehmen.

Neben den standardisierten Makros und Konstanten gibt es eine Unmenge von Vordefinitionen, die herstellerspezifisch sind. Jeder Compiler bringt einen Satz von Makros und Konstanten mit, der auf sein bestimmtes Einsatzgebiet und auf seine spezielle Technik abgestimmt ist. Diese Vordefinitionen sind nur in sehr seltenen Fällen kompatibel.

Die Präprozessoroperatoren # und

Die beiden Operatoren werden gebraucht um symbolische Konstanten oder Makronamen durch den Präprozessor generieren zu lassen. Dabei wandelt # eine Zahl in einen Text und ## fügt zwei Texte aneinander.

Mit dem Makro `#define STR(n) #n` lassen sich durch den Präprozessor Literalkonstanten, die Zahlen repräsentieren, in einen Text konvertieren. Der Ausdruck `STR(42)` wird durch den Compiler in den Text „42“ übersetzt.

```
#include <iostream>

#define SOURCE "Datei: "##__FILE__
#define TRACE( msg ) std::cerr << \
(SOURCE##" => "##msg)<<std::endl;

int main()
{
  TRACE( "Eine Debugmeldung" );

  return 0;
}
```

Listing 2.10 Die Anwendung des Präprozessoroperators ##

2.2.3 Variablen

Alle Daten, die in einem C++-Programm verwaltet werden und damit im Speicher gehalten werden, müssen in C++ typisiert sein. Das heißt, dass man sich vorher über die Art der Daten im Klaren sein muss, bevor man mit ihnen umgeht. Für die Daten werden dann Platzhalter definiert. Im einfachsten Fall – die komplizierteren folgen in den Absätzen weiter hinten in diesem Buch – sind es Variablen, die Daten aufnehmen können.

Sehen wir uns also hier die Variablendeklaration in C++ an. Es können in C++ nur Variablen verwendet werden, die vorher deklariert wurden. Variablen als Platzhalter für irgendwelche Daten, die erst zur Laufzeit feststehen, müssen dem Typ ihrer Daten entsprechen. Es gibt keine Variablen, die wie in BASIC verschiedene Datentypen, also zum Beispiel ganzzahlige Werte oder Texte, aufnehmen können. Die Datentypen, die

solchen Variablen zugrunde liegen, nennt man variante Typen[10]. Eine Variable in C++ wird also zuerst mit deklariert – mit dem nötigen Datentyp – und dann benutzt:

```
int i;
```

Die Variable `i` hat den Datentyp `int`, der ganzzahlige Werte in einem bestimmten Wertebereich erlaubt. Der Bezeichner `i` erlaubt nun mit der Variablen umzugehen. Man kann dieser Variablen zum Beispiel einen Wert zuweisen:

```
i = 7;
```

Der Begriff „Variable‘ steht wie in der Algebra dafür, dass der Platzhalter mit einer bestimmten Bezeichnung für verschiedene Werte steht. Allerdings gibt es einen großen Unterschied zur Algebra. In einem C++-Programm kann eine Variable zu einem Zeitpunkt immer nur genau einen Wert aufnehmen. In der Algebra kann eine Variable durchaus für mehrere mögliche Werte stehen. Die Mathematik steht außerhalb von Zeit und Raum. In einem Programm sieht es anders aus. Ein Programm läuft in der Zeit ab und ändert dabei seinen internen Zustand. Es werden also die Variablen während der Ablaufzeit des Programms geändert. Zu einem bestimmten Zeitpunkt hat eine Variable also immer genau einen bestimmten Wert.

Technisch gesehen ist eine Variable ein Speicherplatz, der Werte eines definierten Typs aufnehmen kann. Der Typ bestimmt die Größe des Speicherplatzes und auch die Weise, wie der Speicherplatz interpretiert wird. Letzteres muss man so verstehen, dass der Speicher, der ja nur eine Aneinanderreihung von Bytes darstellt, alle beliebigen Werte aufnehmen kann, die man in Bytes speichert. In jede Zelle (Byte) passen ganzzahlige Werte von 0 bis 255. Wenn beispielsweise Text oder Fließkommazahlen abgespeichert werden sollen, so muss es ein Verfahren geben, die zu speichernde Information in eben diesen Werten abzulegen, die man in Bytes speichern kann. Hinter einem Fließkommatyp, wie zum Beispiel `double`, steht also die Speichergröße, die ein solcher Zahlentyp im Speicher braucht (acht Bytes), und das Verfahren, wie eine solche Kommazahl in dem Speicher untergebracht wird. Eine Zuweisung, wie sie an der Variablen `i` demonstriert wurde, ordnet die Daten in einer dem Typ inneliegenden Weise und legt sie im Speicher ab.

Im Gegensatz zu C können in C++ fast an jeder Stelle Variablen deklariert werden[11]. Wenn man mehrere Variablen des gleichen Typs braucht, kann man die Variablenbezeichner hinter dem Typ durch Komma trennen.

```
int a, b, c;
```

Diese Deklaration legt drei Variablen des Typs `int` an. Außerdem können Variablen an der Stelle der Deklaration gleich mit einem Wert initialisiert werden.

```
int a = 1, b = 2, c = 3;
```

[10] So etwas kann zwar auch mit C++ erreicht werden. Dazu muss allerdings erst das Klassenkonzept verstanden werden. Deshalb stelle ich mich zunächst auf den Standpunkt, dass es variante Datentypen in C++ nicht gibt.

[11] In C müssen sie immer am Anfang eines Blocks vor den Anweisungen stehen.

Die Variablen, die innerhalb von Funktionen deklariert werden, liegen auf dem Stack. Das ist ein Speicherbereich, der unter anderem für diese lokalen Daten zur Verfügung steht. Der Stack und die möglichen Orte der Variablendeklaration sind in Abschn. 2.2.23 auf S. 91 beschrieben.

2.2.4 Standarddatentypen

C++ ist eine typisierende Programmiersprache. Das heißt, dass alle Daten, die in einem C++-Programm verwaltet werden, einen Datentyp besitzen. Ein Datentyp beschreibt, um welche Art von Daten es sich handelt. Diese Aussage geht durchaus über das hinaus, was in diesem Abschnitt besprochen wird. Hier werden die Grundbausteine der Datentypen besprochen: die einfachen numerischen Typen, die man auch als *Standarddatentypen* bezeichnet. In späteren Abschnitten und Kapiteln werden noch ganz andere Aspekte des Datentyps in der Objektorientierten Programmierung beschrieben.

Zunächst aber zu den Datentypen, die durch die Sprache C++ fertig mitgeliefert werden: die Standarddatentypen. Es gibt vier vorzeichenbehaftete und vier vorzeichenlose ganzzahlige Typen. Die Grundtypen `int`, `short`, `long` sind vorzeichenbehaftet. Man kann das Vorzeichen auch dadurch zum Ausdruck bringen, dass man ihnen das Schlüsselwort `signed` voranstellt. Der vierte vorzeichenbehaftete ganzzahlige Typ ist `signed char`. Von diesen Typen lassen sich die vorzeichenlosen Varianten bilden, indem man ihnen das Schlüsselwort `unsigned` voranstellt. Also `unsigned char`, `unsigned short`, `unsigned int` und `unsigned long`. Der Typ `char` ist je nach Voreinstellung des Compilers vorzeichenlos oder -behaftet.

Während die Typen `char`, `short` in ihren vorzeichenbehafteten sowie in ihren vorzeichenlosen Varianten üblicherweise eine feste Breite haben [12], richten sich die `int` und `long` Datentypen stärker nach den Vorgaben der Plattform und des Compilers.

Alle Standarddatentypen auf einem typischen 32-Bit System im Überblick:

Vorzeichenbehaftete Typen		**Vorzeichenlose Typen**	
	int		
oder	**signed int**		**unsigned int**
oder	**signed**	*oder*	**unsigned**
Bereich	$-(2^{31})$ .. $2^{31}-1$	*Bereich*	0 .. $2^{32}-1$

[12] „üblicherweise“ deshalb, da im Standard keine feste Breite definiert ist. Es ist lediglich folgendes definiert:
`sizeof(char)` ≤ `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)`.
Alle ganzzahligen Datentypen sind also plattformabhängig. Zum Vergleich dazu: [ISO-C++03] Abschnitte 3.9.1.x [basic.fundamental]

	long		
oder	**signed long**		**unsigned long**
Bereich	$-(2^{31}) .. 2^{31} - 1$	*Bereich*	$0 .. 2^{32} - 1$

	short		
oder	**signed short**		**unsigned short**
Bereich	$-(2^{15}) .. 2^{15} - 1$	*Bereich*	$0 .. 2^{16} - 1$

	char		
	Kann signed oder unsigned sein		
	signed char		**unsigned char**
Bereich	$-(2^{7}) .. 2^{7} - 1$	*Bereich*	$0 .. 2^{8} - 1$

float

Genauigkeit 32 Bit

$3,4e10^{-38} .. 3,4e10^{38}$

double

Genauigkeit 64 Bit

$1,7e10^{-308} .. 1,7e10^{308}$

bool

Bereich `true` und `false`

Auf einem 16-Bit System unterscheidet sich normalerweise der Bereich des Datentyps `int`, der sich dem `short` angleicht. Bei 64-Bit Systemen ist es üblicherweise der Datentyp `long`, der eine größere Breite bekommt.

Vorzeichenbehafteter Typ auf 64-Bit System		**Vorzeichenloser Typ auf 64-Bit System**	
	long		
oder	**signed long**		**unsigned long**
Bereich	$-(2^{63}) .. 2^{63} - 1$	*Bereich*	$0 .. 2^{64} - 1$

Der Operator `sizeof`

Mit dem Schlüsselwort `sizeof` kann die Größe eines Datentyps in Byte bestimmt werden. Der `sizeof()`-Ausdruck wird immer zur Compilezeit ausgewertet. Das heißt, dass `sizeof(short)` durch den Compiler meistens direkt in `2` übersetzt

wird[13]. Der Ausdruck `sizeof(int)` ist stark maschinenabhängig, weil der Datentyp `int` üblicherweise der Prozessorbreite entspricht[14]. Trotzdem ist er bei einem Compiledurchlauf immer konstant. Der Ausdruck `sizeof(long)` entspricht auf 32-Bit Maschinen üblicherweise 4, da der Datentyp `long` dort normalerweise 4 Bytes breit ist[15]. Bei 64-Bit Systemen verbreitert sich dieser Datentyp häufig auf 8 Byte[16]. Der Operator kann auch auf Variablen angewendet werden, liefert aber auch dann nur die Größe des zugrunde liegenden Typs. Er liefert also immer einen konstanten Wert, der nicht erst zur Laufzeit ermittelt werden muss. Das folgende Programm demonstriert die Wirkung des Operators `sizeof`:

```
#include <iostream>

int main()
{
    std::cout << sizeof(short) << std::endl;
    std::cout << sizeof(int) << std::endl;
    std::cout << sizeof(long) << std::endl;
    std::cout << sizeof(float) << std::endl;
    std::cout << sizeof(double) << std::endl;

    int x = 42;

    std::cout << "Der Wert " << x
        << " wird in einer Variablen mit der Größe "
        << sizeof(x) << " gehalten." << std::endl;

    return 0;
}
```

Listing 2.11 Anwendung des `sizeof`-Operators

Das Beispiel aus Listing 2.11 erzeugt folgende Ausgabe:

```
2
4
4
4
8
Der Wert 42 wird in einer Variablen mit der Größe 4 gehalten.
```

[13] Ich kenne noch keine Ausnahme.

[14] Auf 16-Bit Maschinen ist `sizeof(int)` üblicherweise 2, bei 32-Bit Maschinen ist es 4.

[15] Auch dazu kenne ich noch keine Ausnahme.

[16] Wie z. B. bei 64-Bit Linux oder 64-Bit MacOS X.