Leseprobe

Mit dieser Leseprobe machen Sie sich mit den »Basisdatentypen von C++« (Kapitel 2), der »Modularisierung« (Kapitel 6) und der »Template-Programmierung« (Kapitel 11) vertraut. Außerdem erhalten Sie das vollständige Inhalts- und Stichwortverzeichnis aus dem Buch.

Jürgen Wolf

C++ – Das umfassende Handbuch

1.062 Seiten, gebunden, mit CD, 3. Auflage 2014
39,90 Euro, ISBN 978-3-8362-2021-7

www.rheinwerk-verlag.de/3278

Kapitel 2

Die Basisdatentypen in C++

Bevor Sie sich höheren und komplexeren Themen der C++-Programmierung nähern können, müssen Sie zunächst die eingebauten Basisdatentypen von C++ und den Umgang damit kennenlernen.

Datentypen sind grundlegend für die Verwendung von Variablen und Konstanten. Hierbei wird zwischen Typen für Ganzzahlen, Fließkommazahlen, Zeichen und Wahrheitswerten unterschieden. Neben den grundlegenden Datentypen lernen Sie in diesem Kapitel auch den Umgang mit denselben, z.B. für arithmetische Berechnungen, kennen. Ebenfalls erklärt wird, was passiert, wenn diese Typen miteinander vermischt werden: hierbei wird eine Konvertierung durchgeführt. Die verschiedenen Verfahren von Typenumwandlungen werden am Ende des Kapitels erläutert.

2.1 Grundlegendes zu Datentypen

Zum Speichern von Werten benötigen Sie einen **Datentyp** in C++. Es stehen Ihnen mehrere eingebaute Typen (auch primitive Datentypen genannt) zur Verfügung. Hierbei unterscheidet man zwischen arithmetischen Ganzzahltypen (`short int`, `int`, `long int` und `long long int`), Fließkommatypen (oder auch Gleitpunkttypen, `float`, `double` und `long double`) und Typen für Zeichen (`char`, `wchar_t`, `char16_t` und `char32_t`). Mit der Nennung des Datentyps legen Sie auch gleich fest, was Sie damit machen können.

Selbst wenn Sie bspw. nur eine einfache Addition durchführen wollen, muss der Datentyp eine Addition auch unterstützen. Bei den eingebauten Typen mag sich dies etwas trivial und selbstverständlich anhören, aber wenn Sie später eigene Typen (bspw. mit Klassen) erstellen, sind Sie dafür verantwortlich, was bei der Verwendung des +-Operators passieren soll, wenn dieser zwischen zwei selbst erstellten Typen steht (Stichwort Operator-Überladung). Aber das geht an dieser Stelle etwas zu weit.

2.1.1 Erlaubte Bezeichner für den Zugriff auf Variablen

Zum Datentyp benötigen Sie einen gültigen Bezeichner (einen Namen für die Variable), um überhaupt auf die Daten im Speicher zugreifen zu können. Hierfür können

Sie beliebige Buchstaben (aber keine landesspezifischen Zeichen wie Umlaute), Ziffern oder das Unterstrichzeichen (bspw. `_bezeichner`) verwenden (Tabelle 2.1). Allerdings ist es nicht erlaubt, dass das erste Zeichen mit einer Ziffer beginnt (somit falsch: `4me`). Auf manchen Compilern kann auch das Dollarzeichen `$` im Bezeichner verwendet werden. Bei den Bezeichnern wird zwischen Groß- und Kleinschreibung unterschieden.

Gültige Bezeichner	Ungültige Bezeichner
`_8tung`	**8**`tung`
`_789_`	`w`**ö**`rter`
`val_4_you`	`paragraph`**§**
`NeBeL`	`ung`**%**`ltig`

Tabelle 2.1 Beispiele von gültigen und ungültigen Bezeichnern

Reservierte Bezeichner

Auf Bezeichner, die mit zwei sequenziellen Unterstrichen oder einem Unterstrich, gefolgt von einem Großbuchstaben, beginnen, sollten Sie verzichten, weil diese für die C++-Implementierung reserviert sind. Bezeichner wie `value` sind gewöhnlich für Compilerzwecke und `Value` für Betriebssystem- und Bibliothekszwecke gedacht.

Umlaute als Bezeichner?

Zwar können nicht direkt Umlaute wie *üöäÜÖÄ* in einem Bezeichner verwendet werden, aber der Standard erlaubt Universalcharakternamen wie `\UXXXXXXXX` oder `\uXXXX`. Folgender Code wäre somit standardkonform:

```
int sch\u00f6n = 5;  // nach C++-Standard erlaubt
std::cout << sch\u00f6n << std::endl;
```

Nicht standardkonform hingegen ist die Verwendung von Umlauten direkt im Bezeichner, wie:

```
int schön = 5;   // Fehler! Nicht erlaubt nach C++-Standard
std::cout << schön << std::endl;
```

Allerdings hat dieses Features mit den Universalcharakternamen den Nachteil, dass es nur sehr wenige Compilerhersteller implementiert haben. g++ hat etwa das Beispiel mit dem Universalcharakternamen ohne Beanstandungen übersetzt. Die meisten

anderen Compiler haben den Universalcharakternamen aber nur »kopfschüttelnd« abgelehnt.

Guter Bezeichner vs. schlechter Bezeichner

Sie wissen zwar nun, wie ein gültiger Bezeichner aussehen darf, und können unter Einhaltung dieser Regeln den Bezeichner nennen, wie Sie wollen. Trotzdem möchte ich Ihnen hier einige Ratschläge ans Herz legen:

- Verwenden Sie **konsistente Namen**. Verwenden Sie bspw. für eine Zählvariable entweder `cn`, `counter`, `cntr`, `nr` oder `n`, aber nicht mehrere nebeneinander. Ebenso sollten Sie die Bezeichner entweder auf Englisch (geläufiger) oder Deutsch schreiben. Ein Mischen der Sprachen wirkt sehr inkonsistent. Achten Sie außerdem möglichst auf Rechtschreibfehler, und vermeiden Sie Bezeichner, die ähnlich lauten (bspw. `color` und `colour`).
- Verzichten Sie auf unnötig lange Bezeichner, weil sich diese schwer merken lassen und zu Fehlern einladen. `einlangerundgueltigerBezeichner` ist für meinen Geschmack schon etwas zu lang.
- Bezeichner wie `O` (Buchstabe O) und `l` (Buchstabe L) sollten Sie vermeiden, weil diese leicht mit `0` (Null) und `1` (Eins) zu verwechseln sind.

2.1.2 Deklaration und Definition

Die Begriffe *Deklaration* und *Definition* werden in der Informatik im Allgemeinen gerne in einem Atemzug genannt und irrtümlicherweise gleichgesetzt. In diesem Abschnitt sollen diese beiden Begriffe etwas voneinander differenziert werden. Dieses Missverständnis rührt leider oft daher, weil man beim »Erstellen« einer Variablen wie folgt eben von einer Deklaration und Definition spricht (was wohlgemerkt nicht falsch ist):

```
// eine Variable erstellen
Datentyp Bezeichner1;
// mehrere Variablen erstellen
Datentyp Bezeichner2, Bezeichner3, Bezeichner4;
```

Von einer **Deklaration** ist gewöhnlich die Rede, wenn Größe, Bezeichner, Datentyp und weitere Attribute einer Variablen (oder – später im Buch – einer Funktion) festgelegt werden. Mit einer Deklaration machen Sie quasi zunächst nur die Variable (oder – später im Buch – auch die Funktion) mit dem Compiler bzw. Linker bekannt. Damit ist es möglich, dass Sie diese Variable oder Funktion an einer anderen Stelle im Quelltext verwenden können.

Die **Definition** ist eigentlich nur ein spezieller Fall einer Deklaration. Bei einer Variablen ist die Rede von einer Definition, wenn der Compiler bzw. Linker einen Code erzeugen und für die Variable(n) einen Speicherplatz reservieren soll, womit die Variable (bzw. Funktion) eine eindeutige Speicheradresse besitzt. Ob der Speicherplatz hier vom Datensegment (statisch) oder zur Laufzeit (dynamisch) reserviert wird, ist dabei egal. Bei Funktionen ist erst die Rede von einer Definition, wenn der Quelltext der Funktion (zwischen den geschweiften Klammern) geschrieben wird.

Eine Variable wirklich nur deklarieren

Bei einer Funktion haben Sie eine Nur-Deklaration, wenn Sie eben nur den Prototyp (Funktionskopf) ohne den eigentlichen Code verwenden (dazu mehr in Abschnitt 5.3, »Funktionen deklarieren (Vorausdeklaration)«). Aber auch bei einer Variablen können Sie eine Nur-Deklaration erstellen, indem Sie das Schlüsselwort **extern** davor setzen:

```
// Reine Deklaration
extern Datentyp Bezeichner;
```

In der Regel bedeutet dies dann auch, dass diese Variable irgendwo definiert werden muss, was man gewöhnlich in einer anderen Übersetzungseinheit macht. Diese Nur-Deklaration von Variablen sei aber hier jetzt nur am Rande erwähnt.

2.1.3 Initialisierung von Variablen

An dieser Stelle geht es jetzt noch nicht um die Initialisierung von Variablen mit dem =-Operator (Zuweisungsoperator) oder dem Stream-Objekt `cin`, sondern vielmehr darum, was nach der Deklaration und Definition für ein Wert im Speicherbereich steht. Gewöhnlich steht nämlich ohne weiteres Hinzutun ein beliebiger **undefinierter Zufallswert** in der Variablen. Hier sollten Sie es sich also gleich zur Regel machen, eine Variable **sofort** mit einem Wert zu **initialisieren**. Nur so können Sie vermeiden, dass mit einem undefinierten Zufallswert weitergearbeitet wird, was auch zu undefinierten Ergebnissen führen kann. Gehen Sie daher immer auf Nummer sicher, und initialisieren Sie die Variable gleich bei der Erstellung:

```
Datentyp Bezeichner = 0;
Datentyp Bezeichner{0};   // Neu in C++11
Datentyp Bezeichner = {0};
```

Empfehlung für besseren Code: Variablen sofort initialisieren

Um sich vor nicht vorhersehbaren Ergebnissen zu schützen, sollten Sie Variablen immer schon beim Erstellen mit einem gültigen Wert initialisieren.

2.1.4 Vereinheitlichte Initialisierung mit C++11

Mit dem neuen C++11-Standard wurde jetzt (endlich) auch eine vereinheitlichte Initialisierung mit geschweiften Klammern `{...}` eingeführt. Diese Initialisierung kann neben grundlegenden Datentypen später auch bei komplexeren Typen wie Arrays, Strukturen, Klassen oder Behälter-Klassen (Container-Klassen) verwendet werden. Vereinheitlichte Initialisierungen sind frei von Mehrdeutigkeiten und bedeuten bei gleichem Aussehen auch immer dasselbe.

Der Grund für eine vereinheitlichte Initialisierung in C++11 war, dass es zu viele verschiedene Formen von Initialsierungen gegeben hat (und immer noch gibt). Und zu viele unterschiedliche Dinge für denselben Zweck macht eine Programmiersprache eher komplexer. Für Einsteiger in C++ bedeutet diese einheitliche Verwendung jetzt auch eine wesentlich einfachere Verwendung der Sprache, weil kein Gedanke mehr daran verschwendet werden muss, wie eine Initialisierung für bspw. eingebaute Datentypen, Strukturen, Arrays oder Klassen mit Konstruktoren usw. auszusehen hat. Jetzt kann man überall einfach und einheitlich die `{...}`-Syntax verwenden.

Vereinheitlichte Syntax im Buch?

Als Autor steht man vor dem Dilemma, was man für das Buch verwenden soll. Würde ich durchgehend die neue vereinheitlichte Syntax verwenden, könnte es sein, dass die Leser außen vor bleiben, die keinen Compiler mit dem neuen C+11-Standard verwenden. Außerdem gibt es ja auch noch viel mehr Code ohne vereinheitlichte Syntax auf der Welt. Auf der anderen Seite könnte der Aufschrei laut sein, weil ich nicht den neuen C++11-Standard durchgehend behandle. So habe ich mich dafür entschieden, die entsprechenden Codezeilen im Buch mit einem Kommentar zu versehen

Hierzu nochmals ein theoretischer Code zu vereinheitlichten Initialisierung mit C++11:

```
Datentyp Bezeichner1{Wert};        // Vereinheitlichte Initialisierung
Datentyp Bezeichner2{Bezeichner1}; // auch erlaubt
Datentyp Bezeichner3 = {Wert};     // das = ist hier optional
```

Empfehlung für besseren Code: Vereinheitlichte Initialisierung verwenden

Wenn es möglich ist und Sie sich sicher sein können, dass Sie den C++11-Standard verwenden und verwenden können, sollten Sie die vereinheitlichte Syntax bevorzugen.

2.2 Ganzzahldatentypen (Integer-Datentypen)

C++ bietet eingebaute Ganzzahldatentypen (auch Integer-Datentypen oder integrale Typen genannt) in verschiedenen Größen an. In aufsteigender Größe sind diese

`short int`, `int`, `long int` und `long long int`. Die Versionen `short`, `long` und `long long` können Sie auch ohne den Namen `int` verwenden.

Gültige (Ganzzahl-)Literale für Integer-Typen, können wie folgt aussehen:

```
-1234    -99    0    456    39543    +2333    +1001
```

Zu diesen Zahlen muss noch Folgendes angemerkt werden:

- Positive Integer-Typen benötigen kein +-Zeichen davor. Das heißt, diese sind auch ohne das +-Zeichen positiv.

long long int (C++11)

Der extra lange Ganzzahltyp `long long int` wurde offiziell erst mit dem C++11-Standard eingeführt.

2.2.1 Regeln für gültige Ganzzahlen

Bevor Sie Ganzzahlen an Variablen übergeben können, müssen hier noch kurz die Regeln beschrieben werden, welche Zeichenfolgen Sie verwenden müssen, damit die Ganzzahl auch für den Compiler gültig ist. Neben der gängigeren dezimalen Schreibweise (Basis 10) können Sie auch eine oktale (Basis 8) und hexadezimale (Basis 16) Schreibweise verwenden. In Tabelle 2.2 finden Sie einige Beispiele gültiger Schreibweisen (auch Literale genannt) aufgelistet.

Dezimal	Oktal	Hexadezimal
0	00	0x0
1	01	0x1
10	12	0xa bzw. 0xA
98	0142	0x62

Tabelle 2.2 Einige dezimale Beispiele mit oktalen und hexadezimalen Gegenstücken

Oktal und hexadezimal

Oktale Zahlen beginnen mit **0** und die hexadezimale Schreibweise mit **0x** am Anfang. Bei der hexadezimalen Schreibweise werden die Buchstaben a, b, c d, e, f (bzw. die gleichen Großbuchstaben) verwendet, um 10, 11, 12 13, 14 und 15 darzustellen.

2.2.2 Ganzzahlen mit Werten initialisieren

Ganzzahlen lassen sich ganz einfach mit der einheitlichen `{...}`-Syntax oder dem Zuweisungsoperator `=` initialisieren. Ebenso können Sie die Werte mithilfe des Stream-Objekts `cin` und den Eingabeoperatoren in die Variable schieben. Hierzu einige Beispiele:

```
00  // kapitel02/ganzzahlen.cpp
01  short sval1{0};      // C++11
02  short sval2;
03  int ival1 = {0144};  // 0144 = 100
04  int ival2{ival1};    // C++11
05  long lval1 = 0xff;   // 0xff = 255
06  long lval2 = lval1;
07  long long llval{0};  // C++11
08  std::cout << "Einen long long-Wert bitte: ";
09  std::cin >> llval;
```

Fehler beim Übersetzen (C++11): Zuweisung via {} oder = ?

Sollten Sie Probleme bei der Übersetzung von Listing *ganzzahlen.cpp* haben, kann es sein, dass Ihr Compiler den C+11-Standard leider noch nicht unterstützt (oder er nicht aktiviert ist). In dem Fall müssen Sie die Zuweisungen mit der neuen vereinheitlichten `{...}`-Syntax zurück auf den klassischen Zuweisungsoperator `=` stellen. Für Zeile **01** würde dies daher bspw. bedeuten, dass Sie diese wie folgt ändern müssten:

```
short sval1 = 0;
```

Gleiches müssten Sie dann hier auch mit den Zeilen **03**, **04** und **07** machen. Die Entwicklungsumgebung von Microsoft (ab Visual Studio 2013) unterstützt die vereinheitlichte Initialisierung, ebenso Entwicklungsumgebungen wie NetBeans, Eclipse, Qt Creator oder selbst Dev-C++ 5.3.x (von Orwell), mit dem MinGW ab 4.6 können Sie dort die neueren C++11-Features nutzen, indem Sie gegebenenfalls bei den Compiler-Flags die Option `-std=c++11` setzen. Mit dem GCC (getestet wurde hier mit den Versionen 4.7 und 4.8) und Clang klappte es ebenfalls ohne Probleme.

In Zeile **01** wird die Variable `sval1` vom Typ `short` mit dem Wert `0` initialisiert. Die Zeile **02** zeigt, dass Variablen nicht unbedingt sofort initialisiert werden müssen, aber ich empfehle Ihnen, dies gewöhnlich immer zu tun (siehe Abschnitt 2.1.3, »Initialisierung von Variablen«). In Zeile **03** sehen Sie, wie Sie einen oktalen Wert an die Variable `ival1` vom Typ `int` zuweisen können. Hier verwenden wir außerdem zusätzlich noch das `=` vor `{...}`, was der einheitlichen Initialisierungssyntax entspricht und so ebenfalls möglich ist. Gleiches machen wir dann auch in Zeile **04**, wo wir die Vari-

able ival2 mit dem Wert von ival1 initialisieren. Selbiges geschieht auch in den Zeilen **05** und **06**, nur mit einem hexadezimalen Wert und hier mit dem klassischen Zuweisungsoperator. In Zeile **09** sehen Sie außerdem, wie Sie über das Stream-Objekt cin selbst einen Wert an die Variable llval von Zeile **07** über die Tastatur interaktiv eingeben können.

2.2.3 Positive oder negative Ganzzahlen

Ohne weitere Angabe können die Ganzzahldatentypen sowohl positive als auch negative Werte bis zu einer für den Typ bestimmten Größe speichern. Benötigen Sie allerdings einen vorzeichenlosen Ganzzahltyp, müssen Sie jeweils das Schlüsselwort unsigned vor den Typ stellen.

```
00  //kapitel02/vorzeichen.cpp
01  unsigned int ival1={0};
02  int ival2={0};
03  signed int ival3={0};
```

Mit dem Voransetzen von unsigned in Zeile **01** kann die Variable ival1 vom Typ int keine negativen Werte mehr speichern. Standardmäßig ohne besondere Angaben wie in Zeile **02** können positive und negative Werte gespeichert werden. Es ist zwar auch möglich, mit dem Schlüsselwort signed den Typ explizit als vorzeichenbehafteten Typ zu kennzeichnen, wie dies in Zeile **03** gemacht wurde, aber darauf können Sie auch meistens **verzichten** (abgesehen von char, siehe Abschnitt »Vorzeichen vom Datentyp char« in Abschnitt 2.4.1), weil ganzzahlige Typen ohne Verwendung von unsigned immer vorzeichenbehaftet wie in Zeile **02** sind.

Konstanten (Literale) mit den Suffixen U und L kennzeichnen

Eine ganzzahlige Konstante wie 123456 ohne weitere Angaben ist stets vom Typ int. Wenn Sie am Ende der Ganzzahl ein u bzw. U anhängen, gilt dieses Literal explizit als unsigned int. Hängen Sie hingegen ein l bzw. L ans Ende, wird diese Konstante explizit als long int gekennzeichnet. Kombinieren Sie beide miteinander (ul bzw. UL), wir diese Zahl als unsigned int long betrachtet:

```
unsigned int uval = {123456U};    // unsigned int Wert
long lval = {0x7FFFFFL};          // long int Wert
unsigned long ulval = {0777ul};   // unsigned long int Wert
```

Auch für den neu in C++11 eingeführten Datentyp long long gibt es mit ll bzw. LL ein entsprechendes Suffix. Für die unsigned-Versionen von unsigned long long lautet das Suffix ull bzw. ULL.

2.2.4 Boolescher Datentyp für die Wahrheit

Eigentlich kein Ganzzahldatentyp im eigentlichen Sinne, aber ich habe diesen hier trotzdem aufgenommen. Der Datentyp bool ist darauf spezialisiert, zu prüfen, ob ein Ausdruck wahr (= true) oder unwahr (= false) ist. Damit werden in der Praxis verschiedene Werte bzw. (bedingte) Ausdrücke auf Wahrheit oder Rückgabewerte getestet.

```
00  //kapitel02/wahrheitswert.cpp
01  int ival1 = 1234;
02  int ival2 = 0;
03  int ival3 = -1;
04  bool bval1 = ival1;  // = true
05  bool bval2 = ival2;  // = false
06  bool bval3 =  ival1 == ival2;  // = false
07  bool bval4 =  ival1 != ival2;  // = true
08  bool bval5 =  ival3;  // = true
```

In Zeile **04** wird der Ganzzahlwert von ival1 dem booleschen Typ bval1 zugewiesen, wodurch der Wert nach true konvertiert wurde, weil ein Wert ungleich 0 immer true ist. Das Gegenteil wird mit Zeile **05** bewiesen, wo Sie den Typ ival2 mit dem Wert 0 bval2 zuweisen. Hierbei wird nach false konvertiert, weil 0 immer false ist. In Zeile **06** wird der Ausdruck ival1==ival2 an bval3 zurückgegeben. Hierbei wird quasi behauptet, dass ival1 gleich ival2 ist, was ja nicht stimmt, und somit bekommt beval3 den Wert false zugewiesen. Ähnliches geschieht in Zeile **07**, nur bekommt bval4 hier tatsächlich ein true zugewiesen, weil ival1!=ival2 (ival1 nicht gleich ival2) stimmt. In Zeile **08** bekommt bval5 den negativen Wert von ival3 (= –1) zugewiesen. bval5 erhält trotzdem true zugewiesen. Es ist nämlich ein geläufiger Irrtum, dass negative Werte false sind. Dem ist nicht so, alle Werte ungleich 0 sind true, und nur 0 wird in ein false konvertiert.

Wozu bool?

Die Interaktion von bool und int mag dem Einsteiger an dieser Stelle noch etwas unlogisch erscheinen, aber spätestens wenn Sie mit Logikanweisungen oder Funktionen arbeiten, bekommen diese ihre Daseinsberechtigung.

++boolval bzw. boolval++ – bool inkrementieren (deprecated)

In der Vergangenheit wurde der Operator ++ (Präfix- und Postfixschreibweise) gerne mal zum Inkrementieren auf dem Typ bool anwendet, um den Wert so auf true zu setzen. Andersherum mit dem -- (Präfix- und Postfixschreibweise) war dies nicht möglich. Der -- zum Dekrementieren eines bool war unzulässig. Die Version mit Operator ++ wurde mit dem C++11-Standard als *deprecated* markiert und sollte in neuen Projekten nicht mehr verwendet werden.

2.3 Typen für Gleitkommazahlen

Auch bei den Gleitkommatypen (bzw. Gleitpunkttypen oder Fließkommazahlen, engl. *floating point numbers*) stehen Ihnen verschiedene Größen zur Verfügung. Für die einfache Genauigkeit wird `float`, für die doppelte Genauigkeit `double` und für eine erweiterte Genauigkeit `long double` verwendet. Die exakte Genauigkeit hängt vom Compilerhersteller ab.

Komma oder Punkt?

In C++ wird die US-Schreibweise für die Gleitkommazahlen verwendet, und daher wird ein Punkt statt eines Kommas gesetzt.

Hierzu einige Beispiele mit gültigen Initialsierungen von Gleitkommazahlen in C++:

```
00  //kapitel02/gleitkommazahlen.cpp
01  float fval1{123.456};   // C++11
02  float fval2{fval1};     // C++11
03  float fval3 = .336;     // 0.336
04  double dval1 = 2.22e-15;
05  long double ldval1 = 1.9e-3f;
06  long double ldval2{0.0};  // C++11
07  std::cout << "Bitte eine Gleitkommazahl: ";
08  std::cin >> ldval2;
09  std::cout << "Ihre Eingabe lautete " << ldval2 << std::endl;
```

In den Zeilen **01** und **02** sehen Sie zwei Möglichkeiten der vereinheitlichten C++11-Initialisierung. Wird der Wert vor dem Komma (Vorkommateil) leergelassen, wird automatisch 0 dafür verwendet, wie Zeile **03** beweisen soll. Sie könnten auch auf den Nachkommateil verzichten, wenn Sie einen Vorkommateil verwenden.

Zuweisung ohne einen Punkt

Würden Sie auf das Komma (bzw. genauer den Punkt) verzichten, würde intern dem Gleitpunkttyp ein `int`-Literal zugewiesen. Dabei würde eine implizite Umwandlung von `int` zum entsprechenden Gleitkommatyp stattfinden.

In den Zeilen **04** und **05** wird eine Exponential-Zuweisung (bzw. Schreibweise) demonstriert, welche Sie in der Regel bei besonders kleinen (0.0000000123) oder großen (12345000000000.0) Zahlen verwenden sollten. Die wissenschaftliche Exponential-Schreibweise besteht aus der Mantisse, einer Folge von Ziffern und dem Zehner-Exponent. Zwischen der Mantisse und Exponent muss ein `E` oder `e` stehen. In Zeile **08** wird dann noch eine Gleitkommazahl über das Stream-Objekt `std::cin` interaktiv über die Tastatur eingelesen und der Variablen `ldval2` zugewiesen.

Ausgabe mit und ohne Exponential-Schreibweise

Bei sehr kleinen und großen Zahlen wird häufig empfohlen, die wissenschaftliche Exponential-Schreibweise zu verwenden, weil es häufig einfacher ist, diese (fehlerfrei) einzutippen. Wer allerdings bei der Ausgabe nicht die Exponential-Schreibweise haben will, für den gibt es einen einfachen Manipulator. Auf die Manipulatoren wird im Detail noch gesondert in Abschnitt 13.5.1, »Manipulatoren«, eingegangen. Für die Ausgabe über das Stream-Objekt `cout` können Sie für diese Zwecke die Manipulatoren `fixed` und `scientific` verwenden. Diese Manipulatoren befinden sich ebenfalls im Namensraum `std`. Mit `scientific` erzwingen Sie die Ausgabe der Gleitpunktzahlen in der Exponential-Schreibweise. Das Gegenteil erreichen Sie hingegen mit dem Manipulator `fixed`. Hier ein Beispiel dazu:

```
00  //kapitel02/exponent.cpp
01  double dval1 = {3.14159265358979323};
02  double dval2 = {6.0234567e17};

03  std::cout << std::scientific;
04  std::cout << dval1 << std::endl; //=3.141593e+00
05  std::cout << std::fixed;
06  std::cout << dval2 << std::endl; //=602345670000000000.00...
```

In Zeile **01** wurde die Variable `dval1` mit einer sehr langen Gleitkommazahl initialisiert. Wollen Sie diesen Wert jetzt in der kürzeren Exponential-Schreibweise betrachten, müssen Sie nur den Manipulator `scientific` über das Stream-Objekt schicken, wie dies in Zeile **03** geschieht. Die nächste Ausgabe in Zeile **04** wird dann in der Exponential-Schreibweise durchgeführt.

Umgekehrt ist dies auch möglich. In Zeile **02** wurde bspw. eine Gleitkommazahl in der Exponential-Schreibweise angegeben. Um diesen Wert in einer Gleitkommazahl ohne Exponential-Schreibweise auszugeben, brauchen Sie nur den Manipulator `fixed`, wie in Zeile **05** zu sehen, an das Stream-Objekt `cout` zu schicken, und in der nächsten Zeile (**06**) wird die lange Schreibweise ohne die Exponential-Darstellung verwendet.

Konstanten (Literale) mit den Suffixen F und L kennzeichnen

Der bevorzugte Typ vom Compiler bei Gleitkommaliteralen ist immer `double`. Aber auch hier können Sie durch Anhängen eines Buchstabens explizit den Typ festlegen. Fügen Sie `f` bzw. `F` ans Ende der Konstante, wird das Literal explizit als `float` verwendet. Wollen Sie hingegen eine Konstante als `long double` kennzeichnen, müssen Sie `l` bzw. `L` an das Ende hängen:

```
float fval1 = {3.141592653589F};        // float Wert
long double ldval2 = {6.0234567e17L};   // long double Wert
```

2.4 Typ(en) für Zeichen

Gleich vorneweg: Die Sache mit der Darstellung von Zeichen auf einem Rechner ist gar nicht so trivial, wie man dies vielleicht auf den ersten Blick meinen würde. Also seien Sie gewarnt. Auch will ich hier noch erwähnen, dass in diesem Abschnitt vorerst nur die Darstellung einzelner Zeichen behandelt wird. Sind Sie auf der Suche nach Strings, dann sollten Sie zu Abschnitt 4.2, »Strings (Zeichenketten)«, blättern.

2.4.1 Der Datentyp char

Der Typ `char` (abgeleitet von engl. *character*, »Zeichen«) ist ein fundamentaler Datentyp, der Zeichen repräsentieren kann. Auch wenn der Typ fähig ist, Ganzzahlen (allerdings in einem kleinen Bereich) zu speichern, wird dieser eigentlich vorwiegend nur zur Zeichendarstellung verwendet. In den meisten Fällen hat ein `char` 8 Bits, womit sich rein theoretisch 256 verschiedene Zeichen ($2^8 = 256$) darstellen lassen.

Welcher Wert dann welchem Zeichen entspricht, hängt vom verwendeten Zeichensatz (Character Sets; Charsets) ab. Wenn Sie bspw. eine `char`-Variable mit dem Wert `65` initialisiert haben, wird dieser Wert anhand des verwendeten Zeichensatzes auf dem System codiert. Wird hierbei der ASCII-Zeichensatz verwendet (meistens der Fall), wäre dieser Wert anhand dieses Zeichensatzes das Zeichen `A`. Glücklicherweise können Sie das Zeichen auch als Literal (genauer Zeichenliteral) verwenden. Ein solches Zeichenliteral wird zwischen einzelnen Anführungszeichen gesetzt (bspw. `'A'`, `'C'`, `'X'`).

Zeichensatz

In den meisten Fällen können Sie sich zumindest darauf verlassen, dass in den ersten 128 Zeichen (0 bis 127) im (US-)ASCII-Code die Dezimalziffern, die 26 Buchstaben des englischen Alphabets (Groß- und Kleinschreibung) und andere grundlegenden (auch nicht darstellbaren) Zeichen enthalten sind. Leider wird es dann häufig etwas schwieriger, wenn Sie die landestypischen Zeichen darstellen wollen, wie bspw. hierzulande die Umlaute. Hier müssen Sie sich dann mit dem verwendeten Zeichensatz auf Ihrem System auseinandersetzen, oder aber Sie verwenden die universelle Lösung mit UTF-8-Literalen, indem Sie bspw. `locale` verwenden (siehe den Abschnitt »Die Crux mit den Umlauten unter Windows« in Abschnitt 1.6.1).

Hier ein erster Codeausschnitt dazu:

```
00  //kapitel02/zeichen.cpp
01  char cval1 = {'A'};
02  char cval2 = {65};
03  char cval3 = {10};
```



```
04  std::cout << cval1 << cval3;    // = A
05  std::cout << cval2 << cval3;    // = A
06  std::cout << "Bitte ein Zeichen: ";
07  std::cin >> cval3;
08  std::cout << cval3 << " = "
              << static_cast<int>(cval3) << std::endl;
```

In Zeile **01** wird die Variable `cval1` mit dem Zeichenliteral `'A'` initialisiert. Dass dies auch mit Ganzzahlen erlaubt ist, zeigt Zeile **02** mit dem Wert `65`, welcher der Variablen `cval2` zugewiesen wird. Gemäß der ASCII-Tabelle entspricht dieser Wert ebenfalls dem Zeichen `'A'`. In Zeile **03** wird der Wert `10` der Variablen `cval3` zugewiesen. Mehr dazu gleich.

Mit Zeile **04** wird das Zeichen `'A'` auf dem Bildschirm ausgegeben. Gleiches wird in den meisten Fällen auch bei der Ausgabe in Zeile **05** passieren. Hier sehen Sie, dass auch im Falle einer Initialisierung mit einer Ganzzahl bei der Ausgabe über `cout` das Zeichen ausgegeben wird, welches dem Wert im Zeichensatz entspricht.

Dass außerdem in den Zeilen **04** und **05** mit der Variablen `cval3` eine neue Zeile ausgelöst wird, liegt daran, dass im ASCII-Zeichensatz der Wert `10` für das Newline-Zeichen `'\n'` steht. In Zeile **07** können Sie zur Demonstration der Variablen `cval3` interaktiv ein Zeichen mit der Tastatur übergeben.

Wollen Sie den tatsächlichen Ganzzahlwert eines `char` ausgeben, sollten Sie eine Typumwandlung mit `static_cast<>` durchführen, wie dies in Zeile **08** demonstriert wurde. Mehr zur Konvertierung von Datentypen erfahren Sie in Abschnitt 2.9, »Konvertieren von Typen«. Statt des `static_cast<>` in diesem Beispiel würde sich auch ein implizite Typumwandlung mit `int ival3=cval;` anbieten.

Zeichenliteral oder Ganzzahlliteral?

Wenn Sie sich jetzt fragen sollten, ob Sie als Literal eine Ganzzahl (bspw. `65`) oder eine Zeichen (bspw. `'A'`) verwenden sollen, so würde ich Ihnen die Zeichenliterale empfehlen, weil diese einen Tick portabler und somit zuverlässiger sind. Wenn Sie erwarten, dass sich hinter der Ganzzahl 65 das Zeichen `'A'` verbirgt, dann gehen Sie davon aus, dass auf dem Rechner der ASCII-Zeichensatz läuft, was vielleicht auch in 99,99 % der Fälle richtig sein dürfte.

Aber für den Fall, dass eben das 0,01 % zutrifft und eben nicht der ASCII-Zeichensatz verwendet wird, sind Sie mit dem Zeichenliteral auf der sichereren Seite. Würde nämlich bspw. der EBCDIC-Zeichensatz verwendet, wäre dort der Wert `65` gar nicht (!) belegt.

Vorzeichen vom Datentyp char

Hier muss noch eine wichtige Anmerkung gemacht werden. Da `char` im Grunde auch ein Ganzzahltyp ist, hängt es bei diesem konkreten Typ von den Compilerherstellern ab, ob `char` als `signed` (-128 bis 127) oder als `unsigned` (0 bis 255) implementiert wurde. Somit ein weiterer Grund, `char` nicht für Dezimalzahlen zu verwenden.

Nicht darstellbare Steuerzeichen

An dieser Stelle sollten auch gleich noch die nicht darstellbaren Steuerzeichen beschrieben werden, wovon Sie bspw. das Zeichen `'\n'` zum Ausgeben einer neuen Zeile bereits mehrmals in diesem Buch verwendet haben. Solche Steuerzeichen werden mit einem umgekehrten Schrägstrich eingeleitet (auch *Backslash* genannt), gefolgt von einem Zeichen mit einer festen Bedeutung. Die Steuerzeichen sind auch in der Zeichencode-Tabelle des entsprechenden Zeichensatzes enthalten. Im weitverbreiteten ASCII-Code sind diese Steuerzeichen die Zeichen mit dem Dezimalwert 0 bis 31 und das Zeichen 127. Davon haben aber in der heutigen Zeit nur noch wenige Steuerzeichen eine wirkliche Bedeutung. In Tabelle 2.3 finden Sie eine Übersicht zu verschiedenen nicht darstellbaren Steuerzeichen, welche Sie in C++ verwenden können.

Dezimal	C++	Bedeutung
0	`\0`	Nullzeichen. Zeichen hat keinen Inhalt. Wird bspw. verwendet, um das Ende einer Zeichenkette zu markieren.
7	`\a`	Erzeugt ein Tonsignal (Beep).
8	`\b`	Bewegt den Cursor eine Position zurück.
9	`\t`	Horizontaler Tabulator, bewegt den Cursor zur nächsten Position (Tabstopp).
10	`\n`	Zeilenvorschub, bewegt den Cursor zur nächsten Zeile.
11	`\v`	Vertikaler Tabulator, bewegt den Cursor zu nächsten vordefinierten Zeile.
12	`\f`	Seitenvorschub, bewegt den Cursor zur nächsten Seite.
13	`\r`	Wagenrücklauf, bewegt den Cursor zum Anfang der aktuellen Zeile.

Tabelle 2.3 Nicht darstellbare Steuerzeichen im Überblick. Der Dezimalwert entspricht dem Wert in der ASCII-Code-Tabelle.

2.4.2 Unicode-Unterstützung

Jetzt soll auch das Unicode-Thema gesondert behandelt werden. Das Thema Unicode ist sehr umfangreich und kann hier daher nur angerissen werden, weil hierzu auch Kenntnisse jenseits der Programmierung nötig sind. Ich will Ihnen hier nur die C++-Seite und somit die Verwendung von Unicode mit C++ näherbringen.

Wie Sie jetzt wissen, gibt es mit den ersten 128 Zeichen dank des weitverbreiteten ASCII-Zeichensatzes selten Probleme mit den Zeichen. Sobald aber landestypische Zeichen wie Umlaute ins Spiel kommen, gibt es immer wieder Dinge, die fortan nicht klar sind. Zwar wurde hierfür der ASCII-Zeichensatz von 7 auf 8 Bits erweitert, und dieser Zeichensatz wurde dann unter den Bezeichnungen ISO-8859-1, ISO-8859-2 etc. bis ISO-8859-15 zusammengefasst. Die Umlaute finden Sie hierzulande in ISO-Latin-1 zusammengefasst. Allerdings hilft es meistens gar nichts, einfach die Eingabeaufforderung auf ISO-Latin-1 zu stellen, weil es so zu Problemen mit den mittlerweile viel weiter verbreiteten UTF-8-Codierungen für Unicode kommt, welche viele moderne Systeme verwenden. Gerade Einsteiger unter Windows verzweifeln dann bei den ersten Schritten in C++, wenn Sie noch Konsolenprogramme schreiben, dass dort noch aus Kompatibilitätsgründen am alten IBM-PC-Zeichensatz festgehalten wird, in dem die Dezimalwerte der Zeichen wieder einen andere Wert haben als dies mit ISO-Latin-1 der Fall ist.

Erst mit dem neuen C++11-Standard bietet C++ eigentlich eine richtig gute Unicode-Unterstützung an. Zwar war bis dato schon der Datentyp `wchar_t` vorhanden, aber dieser Typ hatte auf manchen Plattformen eine unterschiedliche Breite (mal 16 Bits und mal wieder 32 Bits).

Anwendung von wchar_t, bzw. wchar_t überhaupt noch verwenden?

Für die Verwendung von Zeichenliteralen für `wchar_t` müssen Sie ein `L` vor das Zeichen stellen. Da `cout` nur `char` als Zeichen ausgeben kann, müssen Sie außerdem noch das Stream-Objekt `wcout` für die Ausgabe bzw. `wcin` für das Einlesen von breiten Zeichen verwenden, bspw.:

```
wchar_t wcval={L'A'};
std::wcout << wcval << std::endl;
std::cout << "Bitte ein Zeichen: ";
std::wcin >> wcval;
std::wcout << wcval << std::endl;
```

Da es aber nun mal leider so ist, dass die Breite von `wchar_t` compilerspezifisch ist, wäre meine Empfehlung, diesen Typ gar nicht erst zu verwenden, um Unicode-Zeichen zu speichern. Ganz besonders dann nicht, wenn Ihr Programm zwischen verschiedenen Compilern portabel bleiben soll. `wchar_t` ist dafür geeignet, compilerdefinierte breite Zeichen zu speichern, welche auf manchen Compilern dann auch tatsächlich Unicode-Zeichen sein **könnten**. Leider hat man sich in der Praxis gerne darauf verlassen, dass schon alles in ein `wchar_t` reinpassen wird, ohne sich wirklich sicher zu sein, dass das Programm auch tatsächlich Unicode-kompatibel ist. Letztendlich führte dies dann nur noch dazu, dass **Encodings** weniger geläufig waren und dann einfach gar nicht mehr verwendet wurden.

Exkurs: Encoding/Codierung

Der Begriff *Encoding* (oder auch *Codierung*) in Verbindung mit Zeichen wird häufig missverstanden. Hierbei geht es ganz einfach darum, dass der Rechner Buchstaben, Zahlen und Sonderzeichen irgendwie abspeichern muss. Der Computer speichert Informationen wie Zeichen, Zahlen oder Sonderzeichen letztlich auch nur mit 0en und 1en. Wenn Sie das Zeichen »a« als solches sehen, dann nur deshalb, weil eine Zeichentabelle bzw. ein Zeichensatz (auch *Character Sets/Charsets*) auf Ihrem System aus einer Bitfolge dieses Zeichen darstellt. Als Encoding wird somit die Regel bezeichnet, nach der die Bits einem Zeichen (mithilfe des Zeichensatzes) zugeordnet werden können.

Einer der ersten und der wohl bekannteste Zeichensatz dürfte ASCII sein. Da ASCII nicht für landestypische Zeichen ausreichte, wurden 15 weitere Zeichensätze unter der Norm ISO 8859 standardisiert und erweitert (bspw. ISO 8859-1, ISO 8859-2 usw.). Die ersten 128 hat man identisch aus ASCII herauskopiert, um kompatibel zu bleiben. Die deutschen Umlaute und die meisten anderen mittel- und westeuropäischen Sonderzeichen waren in den ersten 10 der 15 ISO 8859 vorhanden. Für fernöstliche Sprachen war aber kein Platz mehr in ISO 5589, was maximal mit 256 Zeichen belegt werden konnte. Dafür gibt es jetzt Unicode, womit es möglich ist, alle Zeichen der Welt in eine Zeichentabelle zu stecken. Natürlich bleibt Unicode kompatibel zu ASCII mit den ersten 128 Zeichen. Allerdings gibt es auch bei Unicode wieder verschiedene Encodings, wovon allerdings UTF-8 und UTF-16 die wichtigsten sind.

Die neuen Unicode-Typen char16_t und char32_t (C++11)

Da man ja nicht 100 %ig vorhersagen konnte, in welchem Format auf welchem System denn nun genau ein Zeichen in `wchar_t` gespeichert wurde (hängt von der eingestellten `locale` ab), unterstützt C++ mit dem neuen C++11-Standard jetzt Unicode mit den neuen Zeichentypen `char16_t` und `char32_t` mit einer fixen Breite. Der Vorteil

gegenüber `wchar_t` liegt ganz klar auf der Hand, indem Sie genau wissen, dass mit `char16_t` das Zeichen (mindestens) 16 Bits breit und im UTF-16-Format codiert ist. Ebenso ist es mit `char32_t`, wo das Zeichen (mindestens) 32 Bits breit ist und als UTF-32-Format vorliegt (Tabelle 2.4).

Unicode-Strings

Aus den neuen Zeichentypen `char16_t` und `char32_t` werden auch die neuen Stringtypen `u16string` und `u32string` (siehe Abschnitt 4.2.3, »Unterstützung von Unicode (C++11)«) gebildet.

Codierung	Datentyp	Stringliteral
UTF-8	`char`	`u8"String ist UTF-8 codiert"`
UTF-16	`char16_t`	`u"String ist UTF-8 codiert"`
UTF-32	`char32_t`	`U"String ist UTF-8 codiert"`

Tabelle 2.4 Codierung der Datentypen

Die Verwendung der Unicode-Zeichen lässt sich ebenfalls mit einer Art Steuerzeichen realisieren. Für ein Unicodezeichen in einem String können Sie entweder `\unnnn` (für UTF-16) oder `\Unnnnnnnn` (für UTF-32) verwenden. Für `nnnn` bzw. `nnnnnnnn` müssen Sie die entsprechende 16- bzw. 32-Bit-Hexadezimalzahl für das gewünschte Unicode-Zeichen angeben. Hierzu ein einfacher Codeausschnitt:

```
00  //kapitel02/unicode.cpp
01  char16_t c16val{u'A'}; // (mind.) 16 Bits breites Zeichen
02  char32_t c32val{U'Z'}; // (mind.) 32 Bits breites Zeichen
03  std::cout << sizeof(c16val) << std::endl;
04  std::cout << sizeof(c32val) << std::endl;
05  std::cout << u8"\u00a9 J\u00fcrgen Wolf" << std::endl;
```

In Zeile **01** können Sie sicher sein, dass hier das Zeichen (mindestens) als 16 Bits breites UTF-Zeichen `c16val` zugewiesen wird. Die Ausgabe in Zeile **03** sollte daher mindestens 2 (Bytes) zurückgeben. Selbiges finden Sie in Zeile **02**, nur handelt es sich hier um (mindestens) ein 32 Bits breites UTF-32-Zeichen. Die Ausgabe von Zeile **04** sollte daher mindestens 4 (Bytes) sein. Mit dem `sizeof`-Operator, der in den Zeilen **03** und **04** verwendet wird, wird die Byte-Größe für den Typ zurückgegeben. In Zeile **05** wird zur Demonstration ein UTF-8-codierter String mit dem Inhalt `"© Jürgen Wolf"` ausgegeben.

Fortsetzung folgt ...

Das Thema Unicode mag Ihnen vielleicht hier noch etwas unvollständig erscheinen, und damit haben Sie auch Recht, aber spätestens wenn Sie später in Abschnitt 4.2, »Strings (Zeichenketten)«, die Strings kennenlernen, werden Sie die neue Unterstützung von Unicode als Ganzes besser verstehen. Dennoch ist es unerlässlich, sich selbst zunächst mit dem Thema Unicode im Allgemeinen auseinanderzusetzen. Leider funktioniert hierbei noch nicht alles auf allen Compilern so, wie man es sich wünscht.

UTF-8-Tabellen

Tabellen mit Unicode-Zeichen finden Sie sehr viele im Internet. Eine hilfreiche Tabelle, welche ich gerne verwende, finden Sie auf *http://www.utf8-zeichentabelle.de*. Wenn Sie aber weit mehr als die landestypischen Unicode-Zeichen suchen, dann empfehle ich Ihnen folgende Tabelle: *http://www.isthisthingon.org/unicode/index.php*.

2.5 Typ auto (C++11)

An dieser Stelle will ich Sie schon mal mit dem neuen Schlüsselwort `auto` vertraut machen, welchem Sie noch des Öfteren in diesem Buch begegnen werden.

Altes Schlüsselwort mit neuer Verwendung

`auto` ist nicht wirklich eine neues Schlüsselwort von C++11. In der Vergangenheit (C++03-Standard) konnten Sie das Schlüsselwort noch verwenden, um eine Speicherklasse zu beschreiben. Allerdings wurde es in der Vergangenheit fast nie verwendet, weil Variablen innerhalb von Funktionen ohnehin per Standard `auto` und außerhalb von Funktionen gar nicht erlaubt waren. Wozu also das Schlüsselwort `auto` verwenden, wenn eine Variable bereits `auto` war? So hat auch das Komitee entschieden, dieses Schlüsselwort neu zu definieren. Natürlich bedeutet dies auch, dass `auto` **nicht abwärtskompatibel** zu C++03 ist. Daher müssen Sie, falls Sie in einem älteren Code tatsächlich `auto` verwendet haben oder vorfinden, um die Speicherklasse einer Variablen zu beschreiben, dieses `auto` aus dem Quellcode entfernen.

Ab C++11 wird das Schlüsselwort verwendet, damit bei der Initialisierung automatisch der passende Datentyp vom Compiler ausgewählt wird. Wohlgemerkt ist die Rede hier nicht von einer dynamischen Typisierung, sondern von einer Typinferenz. Bei einer Typinferenz muss der Typ des Objekts nicht mehr explizit festgelegt werden, weil dieser ohnehin implizit schon festgelegt ist. In der Praxis kann eine solche Typinferenz nur dann funktionieren bzw. verwendet werden, wenn Sie die Variable

gleich bei der Erzeugung mit einem Wert initialisieren. Der Compiler identifiziert den Typ von `auto` also anhand des Initialisierers. Hierzu sei folgender Codeausschnitt gegeben:

```
00  //kapitel02/typinferenz.cpp
01  auto aval1 = 1234;    // = int
02  auto aval2 = 1234L;   // = long int
03  auto aval3 = 12.34;   // = double
04  auto aval4 = 12.34F;  // = float
05  auto aval5 = 'A';     // = char
```

Anhand des Initialisierers erkennt der Compiler, welchen Typ er für das Schlüsselwort `auto` einsetzen muss. In Zeile **01** wird `aval1` ein `int`, weil das der bevorzugte Typ vom Compiler ist, wenn ein Ganzzahlliteral ohne weitere Angaben (wie hier mit 1234) verwendet wird. In Zeile **02** haben wir den Zusatz `L` an das Ende des Literals gehängt, wodurch der Compiler daraus jetzt ein `long int` macht. In Zeile **03** wird der Compiler für `aval3` ein `double` verwenden, weil dies bei Gleitkommaliteralen der bevorzugte Typ des Compilers ist. In Zeile **04** wurde das Suffix `F` hinzugefügt, wodurch der Compiler daraus jetzt ein `float` macht. In Zeile **05** wird `char` als Typ verwendet.

»auto« macht das Leben einfacher

Bei den fundamentalen Datentypen mag Ihnen diese Typinferenz mit `auto` vielleicht ein wenig banal vorkommen. Aber spätestens wenn Sie Dinge wie Templates oder die Standardcontainer mit den Doppelpunkten und spitzen Klammern verwenden, werden Sie das kleine, aber feine Schlüsselwort `auto` nicht mehr missen wollen. Mit `auto` wird das Programm zwar nicht besser, aber der Code ist wesentlich einfacher zu lesen und auch zu schreiben.

Folgendes ist allerdings bei einer Typinferenz mit `auto` nicht erlaubt:

```
01  auto aval1 = 1234;   // = int
02  auto aval2 = aval1;  // Ok! = int
03  aval2 = 123.123;     // Auch Ok
04  aval1 = "string";    // Fehler!!!
```

Die Zuweisung von Zeile **02** geht noch in Ordnung, weil ja der Compiler den Typ in Zeile **01** mit `int` schon festgelegt hat und somit aus Zeile **02** auch ein `int` macht. Auch die Zuweisung in Zeile **03** wird anstandslos (wenn auch nicht schön) übernommen. Nur wird dabei der `double`-Wert 123.123 implizit vom Compiler in einen `int`-Wert konvertiert, so dass `aval2` nur noch den Wert 123 enthält. Diese Konvertierung, welche unabhängig von `auto` ist, wird auch *Narrowing* genannt. Die Zuweisung in Zeile **04** geht aber gar nicht mehr, weil dies hier eine Umwandlung von `const char*` zu `int` wäre und so etwas gar nicht mehr möglich ist.

2.6 Übersicht und Größe der Basisdatentypen

Nachdem Sie jetzt mit den fundamentalen Datentypen in C++ bekannt gemacht wurden, wird es Zeit, ein wenig auf die Details dieser Typen einzugehen – genauer auf deren Größe. Jeder Datentyp ist an eine bestimmte Grenze und Größe gebunden.

Der kleinste Datentyp `char` kann 256 verschiedene Werte speichern und darstellen (unabhängig vom Vorzeichen). Die Anzahl unterschiedlicher Werte ergibt sich aus den acht zusammenhängenden einzelnen Bits (2 hoch 8 = 256). Jede einzelne dieser acht Bitstellen kann den Wert 1 oder 0 darstellen. Und so ergeben sich 256 verschiedene Bitstellungen. Bei den anderen Datentypen ist dies genauso, nur ist hierbei die Anzahl der Bits häufig länger, wodurch sich viel mehr verschiedene Werte darstellen lassen. Wie viele Bits das sind und wie dieser Wert dann interpretiert wird, geben Sie mit dem Datentyp an.

Ein char mit einem Byte besteht nicht zwangsläufig aus 8 Bits

Hier muss noch ein weitläufiger Irrtum beseitigt werden: Und zwar muss ein `char` mit 1 Byte nicht zwangsläufig aus 8 Bits bestehen. Es gibt bspw. Rechner, auf denen ein `char` mit 32 Bits implementiert ist. Der Standard schreibt nur vor, dass 1 Byte eine zusammenhängende Folge von **mindestens** 8 Bits sein muss.

Wie groß letztendlich die einzelnen fundamentalen Datentypen sind, kann gar nicht zu 100 % genau gesagt werden und hängt davon ab, wie diese implementiert sind. Aber es gibt doch gewisse Spielregeln, an die sich die Compilerhersteller halten müssen. Diese sollen hier jetzt erwähnt werden.

2.6.1 Ganzzahltypen

Der natürlichste Typ bei den Ganzzahlen ist `int`, weil dieser der Größe der Ausführungsumgebung angepasst ist. Bei (älteren) 16-Bit-Systemen waren dies 2 Bytes und bei 32-Bit-Systemen wiederum 4 Bytes. Leider lässt sich dies nicht bei einem 64-Bit-System weiterführen, wo das `int` ja 8 Bytes haben müsste. Auf 64-Bit-Systemen hat man das `int` auf einer Breite von 32 Bits belassen. Stattdessen hat dort jetzt der Datentyp `long int` eine Breite von 64 Bits. Leider trifft dies auch nicht immer zu, so bleibt bspw. auf einem 64-Bit-Windows-Rechner auch ein `long int` bei 32 Bits Breite, für 64 Bits müssen Sie dort auf `long long` zurückgreifen.

Wie breit also die Typen sind, hängt vom zugrunde liegenden Datenmodell ab, das verwendet wird. Sobald Sie anfangen, portablen Code zu schreiben, werden Sie sich wohl auch mit diesem Thema befassen müssen. In Tabelle 2.5 finden Sie einen Überblick über einige gängige Datenmodelle und deren Breite.

Type	LP64	ILP64	LLP64	ILP32	LP32
`char`	8	8	8	8	8
`short`	16	16	16	16	16
`int`	32	64	32	32	16
`long`	64	64	32	32	32
`long long`			64		
Zeiger	64	64	64	32	32

Tabelle 2.5 Zum besseren Verständnis: Das L steht für »long«, das P für Pointer (Zeiger), und I steht für »int«. Die Zahlen 32 und 64 sind selbsterklärend.

Nachdem nicht 100 %ig gesagt werden kann, wie breit ein bestimmter Typ implementiert wurde, da das ja vom zugrunde liegenden Datenmodell abhängt, können Sie sich aber auf folgende Reihenfolge verlassen:

```
char <= short <= int <= long <= long long
```

(<= bedeutet hier: ist kleiner oder gleich)

Datentypen für Zeichen

`char` ist in der Regel immer 1 Byte groß und kann daher 256 verschiedene Zeichen darstellen, was für den ASCII-Code und auch die Umlaute ausreicht. Für breitere Typen steht Ihnen auch der Typ `wchar_t` zur Verfügung, der abhängig von der Implementierung mal 2 und mal 4 Bytes groß ist. Beachten Sie, dass es sich bei diesen Typen im Prinzip trotzdem um Ganzzahltypen handelt. Besser als `wchar_t` sind die neuen in C++11 eingeführten Typen `char16_t` und `char32_t` für die Unicode-Zeichendarstellung von UTF-16 und UTF-32, weil hierbei eine Breite von mindestens 16 bzw. 32 Bits garantiert ist.

Wahrheitswert bool

Da `bool` nur zwei Zustände (`true` oder `false`) speichern kann, würde hierfür lediglich 1 Bit ausreichen. Allerdings ist die kleinste adressierbare Einheit 1 Byte, weshalb `bool` auch mit dieser Größe implementiert sein könnte. Auf manchen Systemen hat ein `bool` aber auch dieselbe Größe wie die Prozessorarchitektur (32 oder 64 Bits), was eine bessere Zugriffsgeschwindigkeit ermöglichen kann.

2.6.2 Gleitkommazahlen

Auch bei den Gleitkommazahlen lässt sich nicht mit 100 %iger Sicherheit vorhersagen, wie breit diese Typen auf den unterschiedlichen Systemen sind. `float` ist zwar häufig mit 4, `double` mit 8 und `long double` mit 10 bzw. 16 Bytes vorzufinden, aber eine Garantie gibt es hierfür auch nicht. Aber wie auch bei den Ganzzahlen können Sie sich auf folgende Reihenfolge verlassen:

```
float <= double <= long double
```

(<= bedeutet hier: ist kleiner oder gleich)

2.6.3 Byte-Größe ermitteln – sizeof-Operator

Die Größe der fundamentalen Datentypen auf Ihrem System können Sie mit dem `sizeof`-Operator ermitteln. Hierbei müssen Sie lediglich den gewünschten Datentyp zwischen den Klammern des `sizeof`-Operators (bspw. `sizeof(Datentyp)`) stellen, und es wird die Byte-Größe für den Datentyp zurückgegeben. Hier einige Beispiele dazu:

```
00  //kapitel02/sizeof.cpp
01  std::cout << sizeof(char) << std::endl;  // = immer 1 Byte
02  std::cout << sizeof(wchar_t) << std::endl;
03  std::cout << sizeof(int) << std::endl;
04  std::cout << sizeof(double) << std::endl;
05  std::cout << sizeof(long double) << std::endl;
```

Ausgeführt wurden diese Beispiele auf unterschiedlichen Systemen und besonders bei `wchar_t` in Zeile **02** und `long double` in Zeile **05** gab es häufig unterschiedliche Größen. `wchar_t` war auf dem einen System 2 und auf anderen wiederum 4 Bytes breit. `long double` hatte hingegen 8, 10 und 16 Bytes auf Testsystemen.

2.6.4 Sicherheit beim Kompilieren mit static_assert (C++11)

Erfreulicherweise ist dieses neue C++11-Feature bei fast alle gängigen neueren Compilern implementiert. Mit `static_assert()` überprüfen Sie einen konstanten Ausdruck zwischen den Klammern zur Übersetzungszeit. Gibt die Auswertung dieses Ausdrucks nicht `true` zurück, bricht der Compiler die Übersetzung mit der Fehlermeldung ab, welche Sie ebenfalls angeben können. Daher passt dieses `static_assert()` recht gut in dieses Kapitel hier rein. Ein einfaches Beispiel hierzu:

```
00  //kapitel02/static_assert.cpp
01  static_assert( sizeof(long double) == 16,
                   "Need 16 byte long double" );
```


Hier fordern wir den Compiler auf, den Ausdruck `sizeof(long double)==16` zu überprüfen. Unsere Anwendung erfordert 16 Bytes für ein `long double` auf dem System, auf dem der Quellcode übersetzt wird. Ist der Ausdruck `true`, wird der Quellcode weiter übersetzt. Gibt der Ausdruck `false` zurück, bricht der Compiler die Übersetzung ab und gibt die dahinter geschriebene Fehlermeldung (hier »Need 16 byte long double«) aus (Abbildung 2.1).

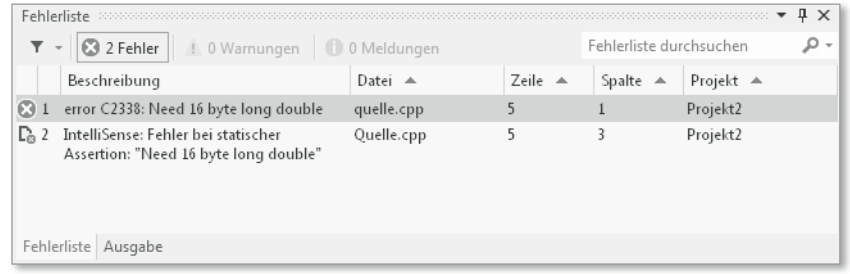


Abbildung 2.1 Beim Microsoft Visual Studio ist auf diesem Testsystem ein »long double« nicht 16 Bytes breit, und daher bricht der Compiler die Übersetzung dank unseres static_assert() ab.

Machen Sie regen Gebrauch von static_assert()

Die Verwendung von `static_assert()` kann nur empfohlen werden. Sie kostet auch überhaupt keine Laufzeit der Anwendung, weil diese Sicherheitschecks nur vom Compiler benutzt werden. Lediglich die Übersetzungszeit nimmt logischerweise zu, aber ich denke, damit kann man leben, weil man hiermit auf so manche Sicherheitsüberprüfung während der Laufzeit des Programms verzichten kann. Und dadurch muss man ja eigentlich auch weniger Code schreiben, wodurch wiederum die Laufzeit des Programms verbessert wird. Die Überprüfungen mit `static_assert()` sind nicht nur auf fundamentale Datentypen beschränkt, sondern machen u.a. auch später Sinn, etwa um auf falsche Template-Parameter zu testen. Auch können Sie diese Prüfungen an jeder Stelle (in einem Namensraum, global, lokal) im Code einbringen.

2.6.5 <limits> und std::numeric_limits

Wenn Sie auf der Suche nach mehr Informationen zu den fundamentalen Datentypen sind, dann werden Sie mit der Headerdatei `<limits>` und der darin enthaltenen Spezialisierung des Templates `numeric_limits` fündig. Die Spezialisierungen befinden sich im Namensraum `std`. Die Verwendung ist im Grunde denkbar einfach, auch wenn die Templates hier noch nicht behandelt wurden:

```
numeric_limits<DATENTYP>::member_constant_or_function;
```

Zwischen den spitzen Klammern geben Sie für DATENTYP den fundamentalen Datentyp an, über den Sie Informationen haben möchten. Hinter den beiden Doppelpunkten geben Sie die Member-Funktion oder Member-Konstante an, wovon in der Headerdatei <limits> eine Menge enthalten sind. So lautet bspw. die Member-Funktion für den maximalen Wert eines Datentyps max() oder eine Member-Konstante, um herauszufinden, ob der Typ ein Vorzeichen hat, is_signed. Hierzu ein einfaches Beispiel, wie Sie Member-Funktion und Member-Konstanten anwenden können:

```
00  //kapitel02/limits.cpp
01  #include <limits>  // benötigte Headerdatei
...
02  std::cout << std::numeric_limits<int>::min()
              << std::endl;
03  std::cout << std::numeric_limits<int>::max()
              << std::endl;
04  std::cout << std::numeric_limits<int>::digits
              << std::endl;
05  std::cout << std::numeric_limits<int>::is_signed
              << std::endl;

06  static_assert(
        (std::numeric_limits<unsigned char>::digits == 8),
        "!!! unsigned char hat hier keine 8 Bits !!!"
               );

07  std::cout << std::numeric_limits<long>::min()
              << std::endl;
08  std::cout << std::numeric_limits<long>::max()
              << std::endl;
09  std::cout << std::numeric_limits<long>::digits
              << std::endl;
10  std::cout << std::numeric_limits<long>::is_signed
              << std::endl;
```

Der Header in Zeile **01** ist nötig, um die Spezialisierungen von numeric_limits verwenden zu können. In den Zeilen **02** bis **03** ermitteln Sie den kleinstmöglichen (min()) und größtmöglichen (max()) Wert, der in einem int gespeichert werden kann.

Die Anzahl der Bits eines Datentyps **ohne** das Vorzeichenbit (!) können Sie mit der Member-Konstante digits ermitteln. Das ist auch der Grund, warum in Zeile **04** bspw. 31 (Bits) statt der vielleicht eher erwarteten 32 zurückgegeben wird. Testen Sie es einfach aus, indem Sie ein unsigned int stattdessen verwenden, wobei dann auch

32 (Bits) zurückgegeben wird (ich gehe jetzt davon aus, dass Sie vor keinem ILP64-System sitzen; siehe Abschnitt 2.6.1, »Ganzzahltypen«).

Ob ein Typ mit einem Vorzeichen implementiert wurde, können Sie mit is_signed (siehe Zeile **05**) ermitteln. Zurückgegeben wird entweder true (mit Vorzeichen) oder false ohne Vorzeichen. In dem Fall wird 1 für true und 0 für false ausgegeben.

Selbstverständlich können Sie die Spezialisierungen auch mit static_assert() verwenden (siehe Abschnitt 2.6.4, »Sicherheit beim Kompilieren mit static_assert (C++11)«). In Zeile **06** machen wir dies, indem unser Programm voraussetzt, dass unsigned char auf dem System 8 Bits breit ist, um übersetzt werden zu können. In den meisten Fällen wird das Beispiel wohl ohne Anstand übersetzt. Wenn nicht, dann haben Sie eine Maschine vor sich, auf der char größer als 8 Bits ist.

In den Zeilen **07** bis **10** wird nochmals mit dem Typ long alles wiederholt, was eben zuvor mit int bei den Zeilen **02** bis **05** gemacht wurde.

Die Member in <limits>

In der Headerdatei <limits> sind noch viele weiterer Mitglieder vorhanden. Hierfür lohnt sich ein Blick in eine Referenz Ihrer Wahl (bspw. *http://en.cppreference.com*).

2.6.6 <climits> und <cfloat>

Die beiden alten Headerdateien <climits> (für Ganzzahltypen) und <cfloat> (für Gleitkommatypen) sind in der C++-Praxis nicht mehr nötig, seit die Spezialisierungen in Form von Templates mit <limits> eingeführt wurden. Beide Headerdateien sind Relikte aus der C-Programmierung und sollten, wenn möglich, in künftigen C++-Projekten nicht mehr verwendet werden.

Kleine Randnotiz: Anzahl der Bits eines char

Zwar wird empfohlen, nicht auf Konstanten von <climits> oder <cfloat> zurückzugreifen, aber früher oder später kommt ein Anwender einmal damit in Berührung, die Anzahl der Bits für ein char ermitteln zu müssen. Ein beliebter Fehler hierbei ist es, std::numeric_limits<char>::digits zu verwenden. Da nicht 100 %ig gesagt werden kann, ob char als signed oder unsigned implementiert ist, könnte der Rückgabewert 7 (wenn signed) oder eben 8 (wenn unsigned) zurückliefern. Es wird leider immer wieder vergessen, dass digits die Anzahl von Bits **ohne** Vorzeichenbit zurückgibt. Entweder greifen Sie hierbei auf die alte Konstante CHAR_BIT aus <climits> zurück, oder Sie verwenden hierfür die modernere (bessere) und gleichwertige Alternative mit std::numeric_limits<unsigned char>::digits.

2.6.7 Übersicht zu den fundamentalen Datentypen

An dieser Stelle finden Sie noch eine kurze Übersicht zu den fundamentalen Datentypen. Die Werte für die Größe oder den Wertebereich sind implementierungsabhängig und entsprechen in den folgenden Tabellen lediglich Werten, wie sie recht häufig anzutreffen sind. Wie groß ein Typ auf Ihrem System tatsächlich ist oder welchen Wertebereich er hat, können Sie mit `sizeof(T)` (für die Speichergröße in Bytes) oder `std::numeric_limits<T>::min()` bzw. `std::numeric_limits<T>::max()` (für den minimalen bzw. maximalen Wertebereich) ermitteln. Für `T` geben Sie den entsprechenden Datentyp an.

Integrale Typen (Ganzzahltypen)

Der grundlegende integrale Datentyp ist `int` mit allen dazugehörenden Verwandten. Auch der Wahrheitswert `bool` und der Zeichentyp `char` gehören zur Gruppe der integralen Typen. In Tabelle 2.6 finden Sie eine Übersicht zu diesen Typen.

Datentyp	Speicher	Gängiger Wertebereich
`bool`	1 Byte	`true` (1) oder `false` (0)
`char`	1 Byte	–128 bis +127 bzw. 0 bis 255
`signed char`	1 Byte	–128 bis +127
`unsigned char`	1 Byte	0 bis 255
`wchar_t`	2 oder 4 Bytes	implementierungsabhängig
`short`	2 Bytes	–32.768 bis +32.767
`unsigned short`	2 Bytes	0 bis 65.535
`int`	4 Bytes	–2.147.483.648 bis +2.147.483.647
`unsigned int`	4 Bytes	0 bis 4.294.967.295
`long`	4 oder 8 Bytes	–9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807
`unsigned long`	4 oder 8 Bytes	wie `unsigned int` oder 0 bis 18.446.744.073.709.551.615

Tabelle 2.6 Ganzzahltypen

Neue Typen mit C++11

Ergänzend zu Tabelle 2.6 mit den Ganzzahltypen, wurden mit dem C++11-Standard noch folgende Datentypen offiziell hinzugefügt (Tabelle 2.7).

Datentyp	Speicher	Gängiger Wertebereich
`char16_t`	2 Bytes	implementierungsabhängig
`char32_t`	4 Bytes	implementierungsabhängig
`long long`	8 Bytes	–9.223.372.036.854.775.808 bis +9.223.372.036.854.775.807
`unsigned long long`	8 Bytes	0 bis 18.446.744.073.709.551.615

Tabelle 2.7 Neue C++11-Datentypen

Gleitkommatypen

Zu guter Letzt fehlt Ihnen nur noch der Überblick über die Gleitkommatypen von C++ (Tabelle 2.8).

Datentyp	Speicher	Gängiger Wertebereich
`float`	4 Bytes	1.2E-38 bis 3.4E+38
`double`	8 Bytes	2.3E-308 bis 1.7E+308
`long double`	10 (oder 16) Bytes	3.4E-4.932 bis 1.1E+4.932

Tabelle 2.8 Gleitkommatypen

2.7 Rechnen mit C++

Nachdem Sie nun die fundamentalen Datentypen kennen, ist es an der Zeit, Ihnen zu zeigen, wie Sie damit einfache arithmetische Operationen ausführen können. An dieser Stelle werden Sie auch bemerken, dass es hier doch einige Tücken gibt, die beachtet werden müssen. Wir beschränken uns bei diesem Buch auf grundlegende und einfache Rechenoperationen.

Unterschiedliche Typen

Falls Sie bei arithmetischen Berechnungen unbedingt Datentypen mischen müssen, sollten Sie 100 % darüber Bescheid wissen und nichts dem Zufall überlassen. Hierbei besteht die Möglichkeit, alles dem Compiler zu überlassen oder die Fäden selbst in die Hand zu nehmen.

Seit C++11 besteht endlich auch die Möglichkeit, die automatische Konvertierung einzuschränken. Auf das Thema wird in Abschnitt 2.9, »Konvertieren von Typen«, eingegangen.

2.7.1 Arithmetische Operatoren

An dieser Stelle kann ich davon ausgehen, dass Sie mit den grundlegenden Bedeutungen arithmetischer Berechnungen (aus Grundschulzeiten) vertraut sind. In Tabelle 2.9 finden Sie die Darstellung der verwendeten arithmetischen Operatoren in C++.

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo

Tabelle 2.9 Arithmetische Operatoren in C++

Alle Operatoren können Sie auf integralen Ganzzahltypen- und Gleitkommatypen anwenden, mit Ausnahme des %-Operators (=Modulo). Dieser darf nur für Ganzzahlen verwendet werden.

%-Operator (Modulo)

Mit dem Modulo-Operator wird eine Division von zwei Ganzzahlen durchgeführt. Als Ergebnis erhalten Sie dann den Rest der Division zurück. So würde eine Berechnung von 10 % 3 das Ergebnis 1 ergeben, weil der Rest von 10/3 gleich 1 ist.

Bei den arithmetischen Operatoren wird auch von *binären Operatoren* gesprochen, weil diese zwei Operanden benötigen (bi = zwei). Operanden können dabei Variablen oder Werte sein. Da eine arithmetische Berechnung, wie bspw. `val1+val2`, häufig keinen Sinn macht (außer bei Überprüfungen oder als Argument bei einem Funktionsaufruf bzw. Rückgabewert), werden Sie sicherlich den Wert dieser Berechnung irgendwo speichern wollen. Wenig überraschend dürfte es sein, dass Sie hierfür den Zuweisungsoperator (=) verwenden können.

Folgender Codeausschnitt soll die arithmetischen Operatoren in der Praxis demonstrieren:

```
00  //kapitel02/calc.cpp
01  int result={0};
02  int val1={100};
```



```
03  int val2={30};
04  int rest{val1 % val2}; // C++11 only !!!

05  val1 + val2;  // Nutzloser Ausdruck
06  result = val1 * val2;
07  std::cout << result << std::endl; //=3000
08  std::cout << rest << std::endl;   //=10
09  std::cout << val1 + val2 << std::endl;  //=130

10  result = val1 + val2 * 2;
11  std::cout << result << std::endl;    //=160
12  result = (val1 + val2) * 2;
13  std::cout << result << std::endl;    //=260
14  std::cout << 102.2 / 33.2 << std::endl; //=3.07831

15  std::cout << "Eine einfache Addition!" << std::endl;
16  std::cout << "1. Summand: ";
17  std::cin >> val1;
18  std::cout << "2. Summand : ";
19  std::cin >> val2;
20  std::cout << "Summe = " << val1 + val2 << std::endl;
```

In Zeile **04** wurde eine arithmetische Berechnung direkt der Variablen `rest` zugewiesen. In diesem Fall enthält `rest` den Rest der Division von `val1/val2`. Diese Zuweisung zwischen geschweiften Klammern funktioniert allerdings so nur mit dem neuen C++11-Standard. Bei einem Nicht-C++11-Compiler verwenden Sie stattdessen bitte `int rest=val1%val2;`. Das Ergebnis in `rest` wird in Zeile **08** ausgegeben.

Die Addition in Zeile **05** ist zwar kein Fehler, aber in dieser Zeile recht nutzlos, was der Compiler gewöhnlich mit einer Warnmeldung monieren sollte. In Zeile **06** wird der Wert von `val1` mit `val2` multipliziert und `result` zugewiesen und das Ergebnis in Zeile **07** ausgegeben. In Zeile **09** können Sie sehen, wie Sie eine Berechnung als Ausdruck direkt über den `<<`-Operator nach `cout` schicken und ausgeben können, ohne dass der Wert zwischengespeichert wird.

Natürlich können Sie Zahlen und Variablen bei den arithmetischen Berechnungen beliebig mischen, wie dies in den Zeilen **10** und **12** demonstriert wird. Dass hierbei bei gleichen Werten in den Zeilen **11** und **13** unterschiedliche Ergebnisse ausgegeben werden, liegt daran, dass sich C++ auch an die Punkt-vor-Strich-Regelung hält.

Punkt- vor Strichrechnung

Bei der Regel der Punkt- vor Strichrechnung werden Multiplikationen und Divisionen vor Additionen und Subtraktionen durchgeführt. Daher ergibt eine Berechnung von 10 + 10 * 2 den Wert 30, weil hier zuerst 10 * 2 durchgeführt und dann die 10 hinzuaddiert wird. Wollen Sie zuerst die Strichrechnung durchführen, müssen Sie Klammerungen verwenden (siehe Zeile **12**), weil geklammerte Operatoren einen höheren Rang bei der Operatorrangfolge haben. Damit würde (10 + 10) * 2 den Wert 40 ergeben, weil nach dieser Schreibweise zuerst die Berechnung in den Klammern 10 + 10 durchgeführt und dieser Wert erst anschließend mit 2 multipliziert wird.

Sinngemäß können Sie auch nur konstante Zahlen für eine Berechnung verwenden, wie Sie dies in Zeile **14** sehen. Da nicht anders angegeben, werden diese beiden Gleitkommazahlen als `double`-Werte behandelt. In den Zeilen **15** bis **20** wird noch eine interaktive Addition durchgeführt.

Erweiterter arithmetischer Zuweisungsoperator

Für alle eben gezeigten arithmetischen Rechenoperatoren +, -, *, / und % gibt es noch eine erweiterte Zuweisungsform mit +=, -=, *=, /= und %=. In Tabelle 2.10 finden Sie die Verwendung dieser Form.

Arithmetische Berechnung	Kürzere Schreibweise
`val1 = val1 * 100;`	`val1 *= 2;`
`val1 = val1 + val2;`	`val1 += val2;`
`val1 = val1 – val2;`	`val1 -= val2;`
`val1 = val1 / 2;`	`val1 /= 2;`
`val1 = val1 % 3;`	`val1 %= 3;`

Tabelle 2.10 Arithmetische Berechnungen und gleichwertige kürzere Gegenstücke

2.7.2 Unäre Gegenstücke

Neben den binären Operatoren haben die Operatoren +, - und * noch jeweils ein unäres Gegenstück. Im Gegensatz zum binären Operator haben die unären Operatoren nur eine einseitige Beziehung zu einem Operanden (Tabelle 2.11).

Operator	Binär	Unär
+	Addition (10 + 10)	Liefert das Positive einer Zahl zurück (+10).
-	Subtraktion (10 – 5)	Liefert das Negative einer Zahl zurück (–10)
*	Multiplikation (10 * 10)	Dereferenzierungsoperator (siehe Abschnitt 4.3, »Zeiger«)

Tabelle 2.11 Unäre Doppelgänger zum binären Gegenstück

Da die unären Operatoren einen höheren Rang als die binären haben, gibt es auch keine Probleme, wenn Sie diese bei arithmetischen Berechnungen verwenden. Hierzu ein kleines Beispiel:

```
00  //kapitel02/unaer.cpp
01  int negval={-123};
02  int posval={+123};

03  std::cout << negval + posval << std::endl;  //=0
04  std::cout << -50 + -100 << std::endl;       //=-150
05  std::cout << -150 + +200 << std::endl;      //=50
```

In Zeile **01** erhält `negval` dank des unären Minus einen negativen Wert. Auf das unäre Plus in Zeile **02** hingegen könnten Sie verzichten, weil der Wert ohne weitere Angaben automatisch positiv ist. Die Addition von `negval` und `posval` ergibt gleich den Wert 0, wie Zeile **03** ausgibt. Die Berechnungen in den Zeilen **04** und **05** sollen demonstrieren, dass die unären Operatoren eine höhere Priorität in der Operatorrangfolge haben. Dabei wird zuerst der Wert mit dem Vorzeichen ausgewertet und dann die arithmetische Berechnung ausgeführt. Auch hier hätte man in Zeile **05** auf das Plus vor dem Wert 200 verzichten können.

Weitere unäre Operatoren

Die hier gezeigten unären Operatoren sind nicht die einzigen in C++. Im Verlauf des Buches werden Sie noch auf weitere unäre Operatoren stoßen.

2.7.3 Wenn Wertebereiche überschritten werden

Wenn Sie Datentypen für arithmetische Berechnungen verwenden, müssen Sie beachten, dass es hierbei auch einen Wertüberlauf geben kann. Im Beispiel vom Datentyp `int` würde mit `std::numeric_limits<int>::max()+1` ein Überlauf stattfinden. Gerade wenn Sie eine Anwendung erstellen, in der ein Anwender unterschiedli-

che Werte eingeben muss, sollten Sie sich niemals darauf verlassen, dass der Anwender schon das Richtige eingeben wird. Der folgende Codeausschnitt soll Ihnen einen solchen Überlauf demonstrieren:

```
00  //kapitel02/ueberlauf.cpp
01  int val1 = {2147483647+1};
02  int val2 = {2147483647};
03  int val3 = {0};

04  std::cout << "Bitte eine Ganzzahl: ";
05  std::cin >> val3;
06  val2 += val3;
07  std::cout << val1 << std::endl;  // -2147483648
08  std::cout << val2 << std::endl;
```

In Zeile **01** findet ein Integer-Überlauf statt, wie die Ausgabe in Zeile **07** auch bestätigt. Hierbei könnte sich der Compiler mit einer Warnmeldung beschweren. Anders sieht es allerdings schon bei der Berechnung in Zeile **06** aus. Wenn der Anwender in Zeile **05** einen Wert größer als 0 eingibt, findet ebenfalls ein Überlauf statt, welcher aber logischerweise nicht mehr zur Übersetzungszeit vom Compiler beanstandet werden kann.

Betrachten Sie am einfachsten die interne Bitdarstellung bei der Berechnung, und Sie werden den Überlauf verstehen:

```
  0111 1111 1111 1111 1111 1111 1111 1111        2147483647
+ 0000 0000 0000 0000 0000 0000 0000 0001      +         1
--------------------------------------       -----------
  1000 0000 0000 0000 0000 0000 0000 0000      -2147483648
```

Genau genommen handelt es sich bei der Bitdarstellung des Ergebnisses ja tatsächlich um den Wert 2.147.483.648. Aber für die Darstellung von `signed int` stehen Ihnen gewöhnlich 31 Bits (`std::numeric_limits<int>::digits`) zur Verfügung. Das höchste Bit (hier das 32.) ist für das Vorzeichen verantwortlich, weshalb hier ein Vorzeichenwechsel stattfindet und somit der Wert eben -2.147.483.648 lautet. Hätten Sie hier `unsigned int` verwendet, würde der Wert 2.147.483.648 lauten. Aber auch mit `unsigned int` würde bei `std::numeric_limits<int>::max()+1` ein Überlauf stattfinden, wie folgende Bitdarstellung zeigen soll:

```
  1111 1111 1111 1111 1111 1111 1111 1111        4294967295
  0000 0000 0000 0000 0000 0000 0000 0001      +         1
  --------------------------------------       -----------
1 0000 0000 0000 0000 0000 0000 0000 0000                 0
```


Da Ihnen für `unsigned int` hier 32 Bits zur Darstellung des Wertes zur Verfügung stehen, wird aus 4.294.967.295+1 der Wert 0 und nicht 4.294.967.296. Hätte `unsigned int` 33 Bits, würde der Wert sogar richtig dargestellt werden. Aber in diesem Fall hat `unsigned int` nun mal nur 32 Bits, und somit wird das 33. Bit (hier durchgestrichen) einfach abgeschnitten.

Bereichsüberschreitung ist ein Fehler!

Auch wenn es hier den Anschein hat, dass ja bei einem Wertüber- bzw. Unterlauf alles wieder von Neuem beginnt (ähnlich wie bei einer Uhr) und auch das Programm weiterhin seinen Dienst tut, sollten Sie trotzdem beachten, dass ein solcher Über- bzw. Unterlauf eigentlich ein Fehler ist. Es ist nicht garantiert, dass alles wieder von vorn beginnt.

Ansätze zum Prüfen von Über- bzw. Unterläufen

An dieser Stelle angekommen, muss leider erwähnt werden, dass der C++-Standard selbst keine Lösung anbietet, einen Über- bzw. Unterlauf zu überprüfen. Es ist Aufgabe des Programmierers, sicherzustellen, dass keine Wertebereichsüberschreitung stattfindet. Einen möglichen Ansatz, die Überschreitung eines `unsigned`-Typs zu überprüfen, soll das folgende Listing demonstrieren:

```
00  //kapitel02/unsigned_ueberlauf.cpp
01  unsigned int val1 = {0};
02  unsigned int val2 = {0};
03  unsigned int result = {0};

04  std::cout << "Bitte eine Ganzzahl: ";
05  std::cin >> val1;
06  std::cout << "Noch eine Ganzzahl  : ";
07  std::cin >> val2;
08  result = val1 + val2;
09  if( result < val1 ) {
10    std::cout << "Achtung, der Wert wurde überschritten"
                << std::endl;
11  }
12  std::cout << result << std::endl;
```

Bei diesem Beispiel wird eine einfache Addition durchgeführt. Die Überprüfung eines Überlaufs führen wir in Zeile **09** durch. Das Prinzip ist recht einfach: Wenn das Ergebnis der Addition in `result` kleiner als einer der eingegebenen Summanden ist, dann kann etwas nicht stimmen, und es muss wohl ein Überlauf stattgefunden haben. Zur Überprüfung wird hier vorausgreifend eine `if`-Bedingung (siehe

Abschnitt 3.1, »Bedingte Anweisung«) verwendet. Das Programm bei der Ausführung mit und ohne Überlauf zeigt Ihnen Abbildung 2.2.

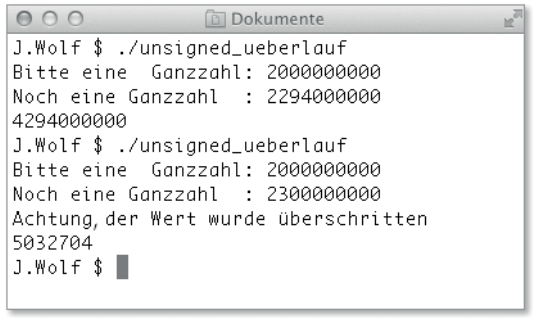


Abbildung 2.2 Das Programm bei der Ausführung: einmal ohne und einmal mit absichtlichem Überlauf

Beispiel nicht ganz »wasserdicht«

Es sollte klar sein, dass eine solche manuelle Überprüfung nur so lange funktioniert, wie zwei gleiche Typen mit ebenfalls erlaubten Werten verwendet werden. Sobald Sie einen konstanten bzw. größeren Wert verwenden, als der Zieltyp aufnehmen kann, ist auch hier der weitere Verlauf wieder unvorhersehbar.

Wesentlich einfacher geht da schon die Überprüfung von vorzeichenbehafteten Typen (`signed`) von Hand, weil Sie hier lediglich anhand des höchsten Bits überprüfen müssen, ob das Vorzeichen gewechselt wurde. Zwar muss hierbei ein etwas komplexerer Code verwendet werden, aber einen möglichen Lösungsansatz will ich Ihnen auch hierzu nicht vorenthalten:

```
00  //kapitel02/signed_ueberlauf.cpp
01  #include <limits>
...
02  int val1 = {0};
03  int val2 = {0};
04  int result = {0};
05  std::cout << "Bitte eine Ganzzahl: ";
06  std::cin >> val1;
07  std::cout << "Noch eine Ganzzahl  : ";
08  std::cin >> val2;
09  result = val1 + val2;

10  if( (val1&(1<<std::numeric_limits<int>::digits)) !=
        (result&(1<<std::numeric_limits<int>::digits)) ) {
```



```
11    std::cout << "Achtung! Vorzeichen wurde gewechselt"
                << std::endl;
12  }
13  std::cout << result << std::endl;
```

Die Überprüfung auf einen Vorzeichenwechsel führen wir in Zeile **10** durch. Mit den beiden kryptischen Zeilen überprüfen wir lediglich das höchste Bit von `val1` und `result` mit jeweils einem booleschen Ausdruck. Ist hierbei das höchste Bit von `val1` ungleich (`!=`) dem höchsten Bit von `result`, wurde wohl (logischerweise) das Vorzeichen gewechselt.

Weitere Informationen

Zugegeben, hier wurden keine ultimativen Lösungen zu diesem Thema geliefert. Das war aber auch gar nicht meine Absicht. Es war mir nur wichtig, Sie als Leser auf das Thema hinzuweisen, so dass Sie sich über Dinge wie *Integer Overflow* von Anfang an im Klaren sind und beim Codedesign darauf achten. Mir ist auch bewusst, dass dieses Thema hier vielleicht zunächst zu viel des Guten war. Für Einsteiger war dieser Abschnitt wohl definitiv zu komplex. Trotzdem wäre ich zufrieden, wenn hier zumindest hängengeblieben ist, dass man nicht einfach arithmetische Berechnungen durchführen und sich darauf verlassen kann, dass der Anwender schon das Richtige eingeben wird.

Wenn Sie auf der Suche nach einer fertigen Bibliothek sind, dann kann ich Ihnen wärmstens die Bibliothek *SafeInt* (*http://safeint.codeplex.com*) ans Herz legen. Die Bibliothek ist dermaßen beliebt, dass sie bei Microsoft Visual C++ 2012 von Haus aus mit der Headerdatei `<safeint.h>` dabei ist.

2.7.4 Rundungsfehler bei Gleitkommazahlen

Nach einem etwas ungemütlicheren Thema muss ich Ihnen leider noch ein weiteres recht ähnliches zumuten. Und zwar betrifft es die Gleitkommazahlen, die nicht immer exakt dargestellt werden können – sprich, es kann zu Rundungsfehlern kommen.

Interna von Gleitkommazahlen

Die Interna von Gleitkommazahlen sind eher Sache der theoretischen Informatik, worüber Sie sich bei Bedarf bitte selbst informieren. In diesem Abschnitt werden nur kurz grundlegende und programmierspezifische Dinge zu den Gleitkommazahlen behandelt. Das Thema Gleitkommazahlen ist sehr umfangreich und kann hier nicht in einem Lehrbuch zu C++ abgehandelt werden.

Gleitkommazahlen können binär nicht immer exakt dargestellt werden. Ursache dafür ist die Darstellungsweise von Gleitkommazahlen durch den Prozessor. Daher können Genauigkeitsverluste und resultierend bei arithmetischen Berechnungen fehlerhafte Ergebnisse auftreten. Damit Sie in C++ nicht über diese Rundungsfehler stolpern, sollten Sie folgende Ratschläge beherzigen:

- Bei Überprüfungen sollten Sie Gleitkommazahlen niemals auf Gleichheit (== bzw. !=) testen. Selbst eine Gleitkommazahl wie 1.333 kann Rundungsfehler enthalten. Verwenden Sie dafür immer die Operatoren für größer gleich (>=) oder kleiner gleich (<=).
- Entwickeln Sie Programme, in denen Geldbeträge verwaltet werden, sollten Sie ganz auf Gleitkommazahlen verzichten. Verwenden Sie stattdessen Ganzzahlen. Einen Geldbetrag wie 1,99 Euro sollten Sie dann mit 199 Eurocents berechnen.
- Müssen Sie trotzdem Gleitkommazahlen bei Ihren Projekten verwenden, müssen Sie sich unbedingt umfassend damit auseinandersetzen und gegebenenfalls externe Alternativen (Bibliotheken) in Erwägung ziehen.
- Entgegen der Intuition können auch Addition und Subtraktion gerne mal Rundungsfehler verursachen. Daher sollten Sie + und - genauso kritisch betrachten wie * und /.

Probleme bei der Genauigkeit von Gleitkommazahlen

Neben den Rundungsfehlern müssen Sie auch die Genauigkeit von Gleitkommazahlen im Auge behalten, welchen auch nur endlich ist. Ohne mich in Details zu verstricken: Ein Gleitkommatyp muss das Vorzeichen, den Exponenten und die Mantisse speichern. Dadurch, dass die Mantisse auch begrenzt ist, entstehen irgendwann Ungenauigkeiten. In der Regel kommt das genormte Gleitkommasystem IEEE 754 in C++ zum Einsatz, wo die Nachkommastelle in der Mantisse gespeichert wird. Bei einer einfachen Genauigkeit wie dem Typen `float` stehen hierfür 23 Bits zur Verfügung. Bei der doppelten Genauigkeit wie bei einem `double` hat der Nachkommateil gewöhnlich 52 Bits. Nicht exakt definiert ist die Breite für Typen (`long double`) mit erweiterter Genauigkeit. Das folgende Beispiel soll Ihnen diesen Sachverhalt zur Genauigkeit von Gleitkommazahlen näher demonstrieren:

```
00  // kapitel02/genauigkeit.cpp
01  float fval = {1234.12 / 1.23};
02  double dval = {1234.12 / 1.23};

03  std::cout.precision(10);
04  std::cout << fval << std::endl;  // 1003.349609
05  std::cout << dval << std::endl;  // 1003.349593
```


In den Zeilen **01** und **02** wird jeweils dieselbe Berechnung durchgeführt und als Ergebnis einmal in ein `float` und einmal in `double` gespeichert. In Zeile **03** stellen wir für die Genauigkeit der Ausgabe von `cout` ein, wie viele Stellen wir sehen wollen. Diese Einstellung hat keinen Einfluss auf den Inhalt der Werte von `float` bzw. `double`. Die Ausgabe in den Zeilen **04** und **05** zeigen jetzt das Problem mit der Genauigkeit von Gleitkommazahlen. `float` ist nach sieben Stellen am Ende und könnte quasi nicht mehr zwischen 1003.349609 und 1003.349593 unterscheiden. Bei `double` ist der Spielraum mit 15 Stellen noch etwas größer.

Genauigkeit reicht mir nicht aus

Wenn Ihnen die Genauigkeit von `double` bzw. `long double` auch nicht mehr ausreicht, bleibt Ihnen nichts anderes übrig, als nach Alternativen zu suchen. Hier würde sich bspw. *Arbitrary Length Integer* für Fließkommazahlen (*http://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic*) anbieten. Boost bietet hierfür bspw. eine Bibliothek mit an, welche *Arbitrary Length Integer* verwendet. Genauso würde sich die *GNU MP Bignum Library* für Integer anbieten (*http://gmplib.org*) oder *GNU MPFR Library* (*http://www.mpfr.org*) für Gleitkommazahlen.

Wozu ist float überhaupt gut?

Da ja `double` der bevorzugte Typ vom Compiler ist und auch wesentlich genauer ist als `float`, stellt sich die Frage, wozu und wann man denn überhaupt `float` verwenden sollte. Die wenigen Beweggründe, die mir hierzu einfallen, sind bspw. eine Schnittstelle (was ja auch eine Hardware sein kann), welche `float` verlangt, und auch der geringere Speicherverbrauch (was allerdings erst bei extrem großen Arrays ins Gewicht fallen dürfte).

2.7.5 Komplexe Zahlen – <complex>

Wenn Sie auf der Suche nach komplexen Zahlen mit Real- und Imaginärteil sind, werden Sie mit der Standard-Template-Klasse `complex` im Namensraum `std` fündig. Komplexe Zahlen können Sie mit allen drei Gleitkommatypen `float`, `double` und `long double` verwenden. Hierfür müssen Sie den gewünschten Typ lediglich zwischen spitze Klammern (<Typ>) stellen, da es sich um ein Template handelt.

```
01  #include <complex>  // benötigte Headerdatei
02  std::complex<double> dcompval{1.0, 2.0};  // C++11-Sytle
03  std::complex<float> fcompval(-1.6, -1.3);
```

Die Headerdatei <complex> in Zeile **01** ist immer nötig, wenn Sie komplexe Zahlen verwenden wollen. In Zeile **02** wurde eine komplexe Zahl vom Typ `double` mit einem

Realteil von 1.0 und einen Imaginärteil von 2.0 initialisiert. Hier wurde auch gleich die neue C++11-Initialisierungssyntax verwendet. Klappt diese bei Ihnen nicht, müssen Sie die geschweiften Klammern durch runde austauschen, wie dies mit der komplexen Zahl in Zeile **03** vom Typ `float` gemacht wurde.

Templates (Schablonen)

Das Thema Templates ist auch ein ganz wichtiges Thema in C++ und bekommt selbstverständlich ein extra Kapitel (Kapitel 11, »Template-Programmierung«) in diesem Buch.

Für komplexe Zahlen lassen sich alle arithmetischen Grundrechenoperatoren +, -, * und / wie gewohnt verwenden. Auch die Operatoren für Gleichheit oder Ungleichheit lassen sich benutzen. Wollen Sie bei den komplexen Zahlen nur den Real- oder den Imaginärteil, finden Sie hierfür die Methoden `real()` und `imag()` vor. Daneben stehen Ihnen viele weitere Methoden zur Verfügung. Hierzu ein Anwendungsbeispiel mit den komplexen Zahlen:

```
00  //kapitel02/komplexeZahlen1.cpp
01  #include <complex>
02  #include <cmath>  // für atan()

03  std::complex<double> cdval1(1.33, 1.66);
04  std::complex<double> cdval2(0.66, 1.99);
05  std::complex<double> cdval3 = cdval1 + cdval2;
06  std::cout << cdval3 << std::endl;

07  std::cout << "Realteil     : " << cdval3.real()
              << std::endl;
08  std::cout << "Imaginärteil : " << cdval3.imag()
              << std::endl;

09  double pi = 4.0 * atan(1.0);  // Pi berechnen lassen
10  double phase = pi/4.0; // 45°
11  double magnitude={0.0};
12  std::cout << "Magnitude eingeben: ";
13  std::cin >> magnitude;
14  cdval3 = std::polar(magnitude, phase);
15  std::cout << "Kartesische Zahl : " << cdval3
              << std::endl;
```

In den Zeilen **03** und **04** werden die komplexen Zahlen gleich mit einem Wert initialisiert. Würden Sie bei den komplexen Zahlen auf eine Initialisierung verzichten, so

würden diese automatisch mit (0.0 + 0.0 i) initialisiert. In Zeile **05** wird demonstriert, dass sich hiermit auch einfache arithmetische Berechnungen durchführen lassen. In den Zeilen **07** und **08** sehen Sie, wie Sie an den Real- und Imaginärteil des komplexen Typs kommen. Nach den vorbereiteten Werten in den Zeilen **10** bis **13** lassen wir in Zeile **14** eine komplexe kartesische Zahl aus den Polarkoordinaten berechnen. `std::polar()` von Zeile **14** ist eine Funktion aus `<complex>`, und `atan()` aus Zeile **09** gehört zu `<cmath>`.

Eine komplexe Zahl können Sie auch über die Tastatur und das Stream-Objekt `cin` einlesen:

```
00  //kapitel02/komplexeZahlen2.cpp
01  #include <complex>

02  std::complex<double> cdval;
03  std::cout << "Ein komplexe Zahl bitte (n.n, n.n):";
04  std::cin >> cdval;
05  std::cout << "Ihre Eingabe war : " << cdval << std::endl;
06  std::cout << cdval << std::endl;
```

Bei der Eingabe in Zeile **04** müssen Sie die komplexe Zahl in der Form (1.2, 3.4) eingeben, also mit (!) den runden Klammern. Wenn die Eingabe für den »normalen« Anwender einfacher gehalten werden muss, können Sie stattdessen auch zunächst zwei `double`-Werte einlesen und dann den entsprechenden Real- und Imaginärteil einer komplexen Zahl übergeben.

Weitere Methoden

Wie bereits erwähnt, gibt es auch hier weitere Methoden, um mit komplexen Zahlen aus der Headerdatei `<complex>` arbeiten zu können. Hierfür empfehle ich Ihnen, bei Bedarf wieder in einer Referenz Ihrer Wahl nachzublättern (bspw. *http://en.cppreference.com*).

2.7.6 Nützliche mathematische Funktionen – <cmath>

Auf der Suche nach soliden trigonometrischen, hyperbolischen, exponentialen und weiteren mathematischen Funktionen dürften Sie in der Bibliothek von `<cmath>` fündig werden. `<cmath>` stammt noch aus klassischen C-Zeiten, aber trotzdem erfreuen sich die Funktionen darin immer noch großer Beliebtheit. Alle Funktionen können mit den Gleitkommatypen `float`, `double` und `long double` verwendet werden. Ein einfaches Beispiel hierzu:

```
01  #include <cmath>
02  double pi = 4.0 * atan(1.0);
03  std::cout << pi << std::endl;
```

Die Headerdatei in Zeile **01** ist unbedingt nötigt, wenn Sie Funktionen aus `<cmath>` verwenden wollen. In Zeile **02** verwenden wir die Funktion `atan()` aus der Bibliothek von `<cmath>` und lassen in diesem Beispiel den Wert der Kreiszahl Pi berechnen.

Echte C++-Mathefunktionen

Sind Sie auf der Suche nach echten mathematischen C++-Funktionalitäten, so kann ich Ihnen empfehlen, sich die **Boost-Bibliothek** anzusehen. Dort sind eine Menge an Funktionalitäten zur Mathematik und Numerik enthalten. Mehr können Sie in den Dokumenten unter *http://www.boost.org/doc/libs/?view=category_Math* nachlesen.

Zwar waren für den C++11-Standard auch einige mathematische Funktionen geplant (siehe C++ Technical Report 1, 2011 oder kurz TR1 C++11; *http://en.wikipedia.org/wiki/C%2B%2B_Technical_Report_1#Mathematical_special_functions*), aber diese haben es nicht mehr rechtzeitig in den Standard geschafft. Das wird sich wohl bis zum nächsten Standard (im Augenblick **C++14** genannt) ändern, mit dem sie wohl eingeführt werden.

Vektorberechnungen

Sind Sie auf der Suche nach Dingen wie Vektorberechnungen, so sollten Sie sich die Template-Klasse `valarray` aus der gleichnamigen Headerdatei ansehen. Diese Klasse wird allerdings in diesem Buch nicht behandelt.

2.8 Zufallszahlen (neu in C++11)

Mit dem C++11-Standard wurde endlich eine eigene Bibliothek für Zufallszahlen eingeführt. Für das Erzeugen von Zufallszahlen mit der neuen Bibliothek sind zwei Dinge nötig:

1. Generator: Dieser Generator (engl. *Engine*) oder auch Zufallszahlenerzeuger erzeugt einen Zufallswert (hier ist auch die Rede von einem Strom von Zufallswerten) zwischen einem minimalen und maximalen Wert. Hierfür stehen einem mehrere vordefinierte Zufallswerte und Generatoren zu Verfügung.
2. Zufallszahlverteilung: Damit können Sie Zufallszahlen mithilfe des Generators mit unterschiedlichen Wahrscheinlichkeitsverteilungen erzeugen.

Das folgende Listing demonstriert Ihnen die einfachste Anwendung von Zufallszahlen. Einmal erzeugen wir Ganzzahlen und einmal Gleitkommazahlen:


```
00  // kapitel02/zufall.cpp
01  #include <random>
...
02  std::random_device rn;
03  std::mt19937 engine(rn());

04  std::uniform_int_distribution<int> dice(1, 6);
05  std::cout << "Yahtzee: ";
06  for(auto i=1; i<=5; ++i)
07  {
08    std::cout << dice(engine) << " ";
09  }
10  std::cout << std::endl;

11  std::uniform_real_distribution<double> randdouble(1.0, 10.0);
12  for(auto i=1; i<=5; ++i)
13  {
14    std::cout << randdouble(engine) << " ";
15  }
16  std::cout << std::endl;
```

Die Headerdatei `<random>` in Zeile **01** ist nötig, um die neuen Zufallszahlen mit C++11 verwenden zu können. Damit der Generator auch unterschiedliche Zufallszahlen zurückliefert, benötigen wir einen Startwert (engl. *seed*), welchen wir in Zeile **02** anlegen. In Zeile **03** wird der Generator mit diesem Startwert initialisiert. `mt19937` ist einer von mehreren vordefinierten Generatoren. Nachdem der Generator initialisiert wurde, richten wir die Zufallszahlenverteilung in Zeile **04** ein. Hier verwenden wir einen diskreten Verteiler für Ganzzahlen zwischen 1 und 6. In den Zeilen **06** bis **10** spielen wir das Würfelspiel *Yahtzee* (oder auch Kniffel), wobei in Zeile **08** der eingerichtete Zufallsgenerator in einer `for`-Schleife, die fünfmal durchlaufen wird, die Zufallszahlen produziert. Dasselbe wird nochmals in den Zeilen **11** bis **16** demonstriert, nur dass hier in Zeile **11** ein kontinuierlicher Verteiler für Gleitkommazahlen zwischen 1.0 und 10.0 eingerichtet wurde. Abbildung 2.3 zeigt das Programm bei der Ausführung.

Kurze Einführung

Das Thema Zufallszahlen wurde an dieser Stelle nur kurz angerissen und vereinfacht dargestellt, um Ihnen ein häufiges Rezept für zufällige Ganzzahlen zu liefern. Auch wurde hier wieder ein wenig auf Ding wie u.a. Schleifen vorgegriffen, weshalb der Code gerade für Einsteiger noch recht komplex erscheinen dürfte.

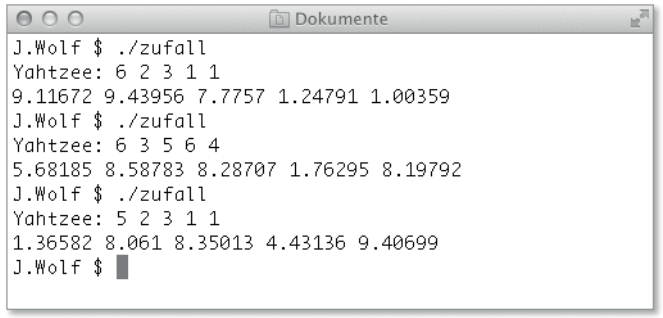


Abbildung 2.3 Hier werden bei jedem Aufruf neue Zufallszahlen generiert.

2.9 Konvertieren von Typen

Jetzt kommen wir zu einem sehr wichtigen Thema. Und zwar geht es um das Mischen verschiedener Datentypen. Zwangsläufig muss beim Mischen verschiedener Datentypen einer der Operanden in einen neuen Zieltyp konvertiert werden. Bei solchen Konvertierungen kann es (zwangsläufig) zur Veränderung des Wertes kommen, wie dies bspw. der Fall ist, wenn Sie einen `double`-Wert nach `int` konvertieren, wo logischerweise die Nachkommastelle verworfen wird. Am idealsten und einfachsten wäre es, wenn Sie erst gar keine Konvertierung durchführen müssten und einfach dieselben Operanden verwenden könnten. Aber dies ist leider nicht immer machbar. Daher haben Sie die Möglichkeit, die Konvertierung entweder dem Compiler zu überlassen, der diese dann nach seinen Regeln durchführt, oder Sie greifen selbst ein und führen eine explizite Umwandlung des Datentyps durch. Mit C++11 ist es neuerdings auch möglich, die automatische Konvertierung zu beschränken, um ein versehentliches Konvertieren von bspw. `double` nach `int` zu unterbinden.

2.9.1 Automatische (implizite) Typumwandlung

Wenn Sie dem Compiler eine Typumwandlung überlassen, geht dieser wie folgt vor (Abbildung 2.4):

- Verwenden Sie eine Zuweisung, wird automatisch der Wert, der auf der rechten Seite des Zuweisungsoperators steht, in den Wert umgewandelt, der auf der linken Seite steht.
- Bei arithmetischen Berechnungen bzw. Vergleichen von zwei verschiedenen Typen wird automatisch immer der kleinere Datentyp in den größeren Datentyp umgewandelt.
- Wird an einer Funktion (siehe Abschnitt 5.5, »Funktionsparameter«) ein anderer Typ als Argument übergeben, wie in der Funktion vereinbart wurde, so wird automatisch in den Typ der Funktion konvertiert.

Verlust von Informationen (Narrowing, implizite Verengung)

Natürlich sollten Sie sich immer bewusst sein, dass es bei einer Typumwandlung zu Verlusten von Informationen kommen kann, umso ärgerlicher, wenn die Typumwandlung unbedacht oder versehentlich erfolgt ist. Dass der Nachkommateil einer Gleitkommazahl bei der Zuweisung an eine Ganzzahl nicht mehr vorhanden ist, sollte jedem klar sein. Schlimmer wird es dann schon, wenn der Quelltyp größer als der Zieltyp ist. Dies ist bspw. gegeben, wenn Sie ein `int` als Quelltyp einem `short` als Zieltyp zuweisen. Hier kann es zu einem Überschreiten des Wertebereichs kommen. Ebenfalls sehr problematisch ist das Mischen von Typen mit unterschiedlichen Vorzeichen (`signed`, `unsigned`), wobei es zu undefinierten Problemen kommen kann. Der Compiler warnt Sie gewöhnlich (hängt von der Einstellung ab), wenn Sie `signed` und `unsigned` bspw. bei Vergleichen mischen, da es hierbei passieren kann, dass ein Vergleich von einem `unsigned int`-Wert mit 4.294.967.295 und einem `signed int`-Wert von –1 (`unsigned==signed`) gleich `true` zurückgibt, weil beide Bitdarstellungen identisch sind!

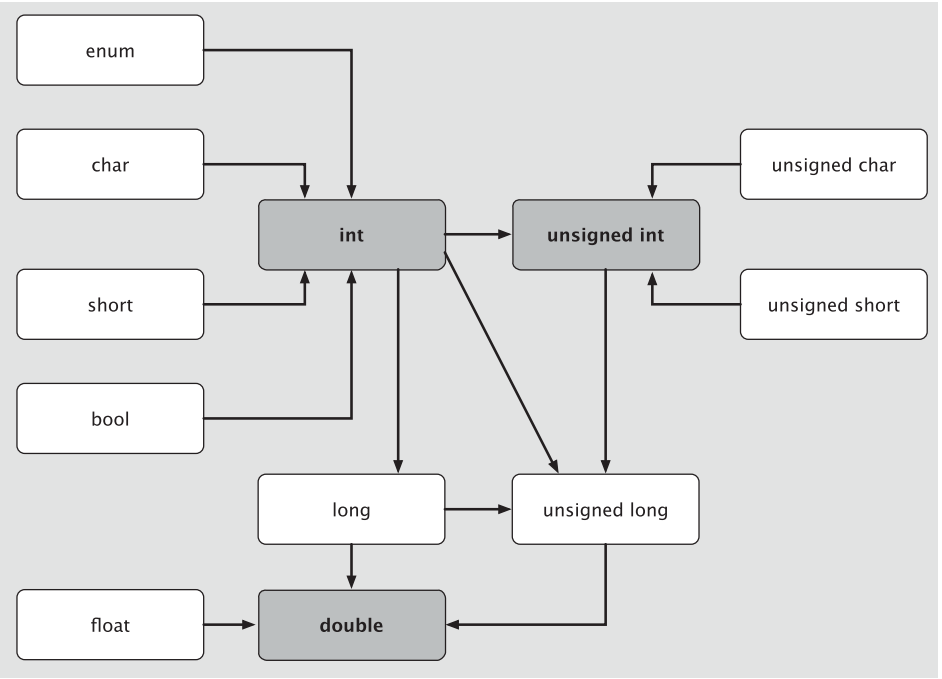


Abbildung 2.4 Hier wurden zwar nicht alle Typen aufgelistet, aber die Grafik soll zeigen, dass der Compiler bei der Verwendung von unterschiedlichen Typen immer versucht, in den nächsthöheren Typ umzuwandeln.

Hierzu nun ein einfaches Beispiel, welches diese impliziten Typumwandlungen in der Praxis zeigen soll:

```
00  // kapitel02/implizitkonvert.cpp
01  void funktion( int ival ) {
02    std::cout << "funktion(): " << ival << std::endl;
03  }
...
04  double dval = 123.123;
05  funktion(dval); //=123

06  int ival = dval; //=123
07  std::cout << "ival: " << ival << std::endl;

08  short sval1 = 32767;
09  std::cout << "sval1 + 1 = " << sval1+ 1 << std::endl;
10  short sval2 = sval1 + 1;  // Bereichsüberschreitung
11  std::cout << "sval2 = " << sval2 << std::endl;

12  long lval = sval1 + 1;
13  std::cout << "lval = " << lval << std::endl;
```

In Zeile **05** wird ein `double`-Wert als Argument an die Funktion (Zeile **01** bis **03**) übergeben, welche allerdings als Argument einen `int`-Typ erwartet (siehe Zeile **01**). Der Compiler führt in der Funktion eine implizite Konvertierung durch, so dass in der Funktion nur noch der Integer-Wert 123 (Zeile **02**) ausgegeben wird. Der Nachkommateil wird verworfen. Selbiges geschieht in Zeile **06**, wo eine `double`-Wert 123.123 einem `int`-Typ zugewiesen wird. Der Compiler konvertiert auch hier implizit den `double`-Wert in einen `int`-Wert, und der Nachkommateil wird verworfen. Diese implizite Verengung des Typs wird auch als *Narrowing* bezeichnet.

In Zeile **08** erhält `sval1` den maximal möglichen Wert, den ein `short` speichern kann. Dass die arithmetische Berechnung in Zeile **09** trotzdem korrekterweise 32768 ausgibt und kein Überlauf stattfindet, liegt an der automatischen Promotion (= Ausweitung). Mehr zur automatischen Promotion folgt nach der Beschreibung des Listings. Zur Demonstration übergeben wir in Zeile **10** den maximalen Wert von `short` +1 der `short`-Variablen `sval`, was in diesem Fall dann, wie die Ausgabe in Zeile **11** zeigt, zu einem Wertüberlauf führt und ein Fehler im Programm ist. In Zeile **12** machen wir nochmals dasselbe, nur ist der Zieltyp hier `long`, und da `long` wesentlich breiter als `short` ist, gibt es mit dieser Typumwandlung überhaupt keine Probleme.

Narrowing vermeiden

Um diesen Abschnitt nicht falsch zu verstehen, folgt noch ein Hinweis. Sofern es irgendwie möglich ist, sollten Sie immer versuchen, ganz auf eine implizite Verengung (*Narrowing*) zu verzichten.

Vom Compiler bevorzugt ...

Wenn Sie Abbildung 2.4 betrachten, fragen Sie sich sicherlich, warum hier die Typen `int`, `unsigned int` und `double` hervorgehoben wurden. Es wurde zwar schon das eine oder andere Mal indirekt erwähnt, aber hierbei handelt es sich um eine spezielle Form der Umwandlung, die Sie als Programmierer gar nicht mitbekommen (und deren Ergebnis Sie in Listing *implizitkonvert.cpp* in Zeile **09** sehen konnten). Die Rede ist von einer automatischen Promotion (eingedeutscht so viel wie »Aufweitung«). Wenn Sie quasi eine arithmetische Operation durchführen, wird eine Promotion auf `int`, `unsigned int` oder `double` durchgeführt.

Hierbei gibt es eine integrale Promotion, bei der ein Typ wie `bool`, `char`, `signed char` oder `short` auf ein `int` bzw. `unsigned char`, `unsigned short` auf ein `unsigned int` »aufgeweitet« werden. Bei einer Gleitkommapromotion wird dann ein `float` nach `double` aufgeweitet. Dies hat den Vorteil, dass im Grunde mit dem natürlichsten Typ auf dem System gearbeitet wird, womit die arithmetischen Berechnungen eben am schnellsten sind.

Diese Typen werden nicht »aufgeweitet«

Keine Promotion findet statt, wenn einer der Operanden einen größeren Wertebereich wie `int`, `unsigned int` oder `double` besitzt, wie dies bspw. bei `long int`, `long long int` oder `long double` der Fall wäre.

2.9.2 Automatische Typumwandlung beschränken (C++11)

Wollen Sie eine automatische Verengung (*Narrowing*) von Datentypen einschränken, dann brauchen Sie einfach nur die neu in C++11 eingeführte einheitliche Initialisierung mit den geschweiften Klammern zu verwenden. Damit können Sie quasi versehentliche Konvertierungen abfangen. Folgendes Listing zeigt Ihnen, wie Sie das *Narrowing* im neuen C++11-Standard einschränken können:

```
00  // kapitel02/nonarrowing.cpp
01  double dval = 123.123;
02  int ival={dval};  // Fehler !!! Narrowing

03  short sval1 = {32767};
04  short sval2 = {sval1 + 1};  // Fehler !!! Narrowing

05  long lval = {sval1 + 1};  // OK!
```

In Zeile **02** findet ein *Narrowing* von `double` nach `int` statt. Die Daten von `dval` passen aber nicht komplett in `ival` und da hier die neue einheitliche Initialisierungsliste verwendet wurde, ist es jetzt ein Fehler. Selbiges gilt in Zeile **04**, wo der Ausdruck `sval1+1`

zwischen den geschweiften Klammern nicht mehr in `sval2` passt. Die Konvertierung in Zeile **05** hingegen ist problemlos möglich.

Einheitliche Initialisierung bevorzugen

Mit der einheitlichen Initialisierung zwischen den geschweiften Klammern werden versehentliche und unschöne Verengungen von Typen vermieden, daher empfehle ich Ihnen, diese künftig auch regelmäßig zu verwenden. Und wenn Sie trotzdem mal ein *Narrowing* benötigen, dann sollten Sie den Typ explizit konvertieren, um zu zeigen, dass Sie sich hier im Klaren sind, was Sie tun. Die explizite Typumwandlung ist Thema des nächsten Abschnitts. Zu diesem Thema muss allerdings noch hinzugefügt werden, dass viele Compiler das *Narrowing* zwischen den geschweiften Klammern noch nicht (richtig) beherrschen. Aber viele Compiler geben mindestens eine Warnmeldung aus, dass ein *Narrowing* stattfindet.

2.9.3 Explizite Typumwandlung

Da Sie jetzt wissen, wie der Compiler implizit vorgeht, um Typen zu konvertieren, was passiert, wenn Sie Typen mischen, und wie Sie eine (versehentliche) implizite Verengung (*Narrowing*) vermeiden können, kommen wir jetzt zur expliziten Typumwandlung. Mit ihr können Sie als Programmierer zeigen, dass Sie genau wissen, was Sie tun, und dass es sich nicht um eine versehentliche Konvertierung handelt, bei der es eventuell zu einem Datenverlust kommen kann.

Durch vorausschauende Planung eine Typumwandlung vermeiden

Es kann nicht oft genug erwähnt werden: In den meisten Fällen lässt sich eine Typumwandlung mit ein wenig vorausschauender Planung des Codes vermeiden. Auch wenn C++ spezielle Operatoren für die Typumwandlung anbietet, ist es meistens trotzdem mehr eine Lösung »mit Gewalt« als »mit Gefühl«.

Hierzu nun eine Übersicht der verschiedenen Cast-Operatoren, welche C++ zur Verfügung stellt. Dabei wird immer der Wert von `ausdruck` in den `TYP` umgewandelt:

- `static_cast<TYP>(ausdruck)`
  Das ist der Cast-Operator, den Sie für die typischen Standard-Typumwandlungen verwenden sollten. Dabei wird empfohlen, diesen Operator bei jeder Typumwandlung zu verwenden, damit deutlich wird, dass hier eine (gewollte) Typumwandlung stattfindet.
- `dynamic_cast<TYP>(ausdruck)`
  Dieser Operator ist ähnlich wie `static_cast`, nur findet eine Überprüfung zur Laufzeit statt, wenn der Compiler bei der Übersetzung den Typ noch nicht kennen sollte.

  Der Operator kann verwendet werden, um Zeiger oder Referenzen zwischen abgeleiteten Klassen umzuwandeln. Für Details dazu ist es an dieser Stelle noch zu früh.
- `const_cast<TYP>(ausdruck)`
  Mit diesem Operator können Objekte, die mit `const` gekennzeichnet wurden und eigentlich nicht mehr geändert werden können, vorübergehend außer Kraft gesetzt werden. In der Praxis kann dies bspw. »nützlich« sein, wenn Sie einen konstanten Zeiger oder Referenzen an eine Funktion übergeben, welche keine konstanten Daten erwartet. Dieser Operator ist mit Vorsicht zu genießen, weil es keine Garantie gibt, dass die konstanten Daten dann tatsächlich auch geändert werden können.
- `reinterpret_cast<TYP>(ausdruck)`
  Dieser Cast-Operator ist wohl eher ein Exot und kann auf einer sehr tiefen Ebene die unterschiedlichsten Typumwandlungen durchführen. Allerdings hängt das Ergebnis auch häufig von der Implementierung des Operators ab.

Hierzu noch ein paar einfache Beispiele mit dem `static_cast`-Operator im Einsatz:

```
00  // kapitel02/staticcast.cpp
...
01  void funktion( int ival ) {
02    std::cout << "funktion(): " << ival << std::endl;
03  }
...
04  double dval = {123.123};
05  funktion(static_cast<int>(dval));
06  int ival=static_cast<int>(dval); //=123
07  std::cout << "ival: " << ival << std::endl;

08  short sval1 = {32767};
09  long lval = static_cast<long>(sval1 + 1);
10  std::cout << "lval = " << lval << std::endl;

11  char ch1 = 'Z';
12  std::cout << static_cast<int>(ch1) << std::endl;

13  unsigned int val1 = {4294967295};
14  unsigned short val2 = static_cast<short>(val1);
```

In Zeile **05** wird der `static_cast`-Operator verwendet, um den `double`-Wert in ein `int` zu konvertieren und als Argument der Funktion (Zeile **01** bis **03**) zu übergeben. Auch wenn der Operator nicht unbedingt nötig wäre, sollten Sie sich angewöhnen, diesen trotzdem zu verwenden, um deutlich zu machen, dass ein Typ konvertiert wird. Dieselbe Typumwandlung wurde auch noch bei der Zuweisung in Zeile **06** durchgeführt.

In Zeile **09** wird ebenfalls der `static_cast`-Operator verwendet, obwohl es auch hier nicht nötig gewesen wäre. Aber auch hier soll deutlich gemacht werden, dass eine Typumwandlung hier explizit gewollt wurde und es kein Versehen war. In Zeile **12** wird dank der Konvertierung eines `char` in ein `int` der dezimale Wert des Buchstabens `'Z'` ausgegeben. Diese Konvertierung von `char` nach `int` dürfte wohl auch die einzige sinnvolle in diesem Beispiel gewesen sein. Bei allen anderen hätte man sich wohl eher auf einen einheitlichen Typ festlegen können.

Trotz aller Technik ist man nicht vor Informationsverlusten geschützt, wie Zeile **14** zeigen soll. Der Wert von `unsigned int` hat nun mal keinen Platz in `unsigned short`, und der Wert in `unsigned short` wird ein ganz anderer sein. Gegebenenfalls kann man hier noch den `static_cast` in einer einheitlichen C++11-Initialisierung zwischen geschweifte Klammern stellen, wo sich der Compiler dann über das *Narrowing* beschweren sollte.

Vorteile der C++-Cast-Operatoren

Wenn Sie schon unbedingt eine einfache Typumwandlung durchführen müssen, dann sollten Sie stets den `static_cast`-Operator verwenden und dies nicht dem Compiler mit der impliziten Umwandlung überlassen. Zum einen zeigen Sie damit an, dass Sie hier bewusst eine Umwandlung vornehmen wollen. Im Falle eines Fehlers mit seltsamen Ergebnissen werden Sie so auch schneller fündig, weil Sie nur nach `static_cast` im Code suchen müssen. Das hilft enorm bei der Lesbarkeit des Codes (auch wenn die Verwendung von `operator<TYP>(ausdruck)` zunächst etwas kryptisch wirken mag).

Alter C++-Cast im Funktionsstil

Der Vollständigkeit halber soll hier auch noch der alte C++-Cast im Funktionsstil erwähnt werden. Er ähnelt ein wenig dem alten C-Cast, nur dass dieser eher an einen Funktionsaufruf erinnert:

```
01  double dval = {123.123};
02  int ival(dval);  // C++-Cast im Funktionsstil
03  int ival2 = (int)dval; // C-Style-Cast
```

Hier wird in Zeile **02** ein `double`-Wert in ein `int` umgewandelt. In der Praxis wird aber empfohlen, stattdessen den neueren `static_cast` zu verwenden. In der letzten Zeile wird der C-Style-Cast verwendet.

Kapitel 6
Modularisierung

In diesem Kapitel werden wichtige Schlüsselwörter für Typen, Namensbereiche, die Präprozessor-Direktiven und das Organisieren von Quelldateien behandelt.

Auf den folgenden Seiten werden verschiedene Themen abgehandelt, die sich recht gut in Verbindung mit der Organisierung (Modularisierung) von mehreren Quelldateien beschreiben lassen. Hierbei werden Sie die Namensräume kennenlernen. Bisher haben Sie in Ihren Beispielen `std::` geschrieben, ohne dass dies etwas genauer erklärt wurde. Ebenso haben Sie verschiedene Schlüsselwörter für Speicherklassenattribute oder Typqualifikatoren verwendet, ohne Genaueres darüber zu erfahren. Daher werden diese Schlüsselwörter ebenfalls in diesem Kapitel behandelt. Zusätzlich werden hier auch noch die Präprozessoranweisungen erläutert (bspw. `#include` usw.). Am Ende möchte ich Ihnen noch ein paar Ratschläge mitgeben, wie Sie Ihren Code organisieren können.

6.1 Namensräume

In diesem Abschnitt werden Sie eigene Namensräume erstellen und verwenden. Am Ende soll noch kurz erläutert werden, warum in den Programmen bisher immer `std::` verwendet wurde und welche Alternativen es dafür gibt.

Mit einem Namensraum (engl. *Namespace*) erzeugen Sie einen eigenen Gültigkeitsbereich, worin Sie beliebige Bezeichner wie Klassen, Variablen, Funktionen, Typen oder gegebenenfalls sogar weitere Namensräume deklarieren können. Besonders bei umfangreichen Projekten mit mehreren Personen und mehreren Modulen (Quelldateien) und Klassenbibliotheken wird zur Verwendung von Namensräumen geraten, um eventuelle Konflikte mit gleichnamigen Bezeichnern zu vermeiden.

Um den Sinn und Zweck von Namensbereichen zu verstehen, finden Sie hier ein völlig offensichtliches Beispiel dazu. Betrachten Sie folgenden einfachen Code:

```
00  // kapitel06/namespace01.cpp
01  #include <iostream>
02  using namespace std;

03  void cout( int data );
```

```
04  int main() {
05    cout << "Das gibt Probleme\n";
06  }

07  void cout( int data ) {
08    // Code für sort();
09  }
```

Dieses kleine Beispiel enthält zwei unschöne Dinge, weshalb sich der Code nicht übersetzen lässt. In Zeile **03** deklarieren und in den Zeilen **07** bis **09** definieren wir eine Funktion `cout()`, was zu Problemen mit dem Standard-Stream `cout` führt. Dass die Funktion `cout()` zu einer Fehlermeldung bei der Übersetzung führt, ist allerdings das globale Einbinden vom Namensraum `std` in Zeile **02** Schuld, wodurch es jetzt zwei Bezeichner `cout` im Gültigkeitsbereich gibt. Das Problem könnten Sie jetzt hier beheben, indem Sie Zeile **02** entfernen und in Zeile **05** wieder wie gehabt über `std::cout` auf den Stream zugreifen würden.

Oder aber Sie könnten das Problem beheben, indem Sie die Funktion `cout()` in einen eigenen Namensraum verpacken:

```
00  // kapitel06/namespace02.cpp
01  #include <iostream>
02  using namespace std;  // Nicht gut! Siehe Abschnitt 6.1.4

03  namespace mySpace {     // Neuer Namensraum - Anfang
04    void cout( int data ) {
05      // Anweisungen
06    }
07  }  // Neuer Namensraum - Ende

08  int main() {
09    cout << "Jetzt gibt es keine Probleme\n";
10    mySpace::cout(1234);
11  }
```

Jetzt wurde die Funktion `cout()` zwischen den Zeilen **03** bis **07** in einen Namensraum `mySpace` gepackt. So viel zur Theorie. Mehr dazu erfahren Sie in den folgenden Abschnitten. Ich wollte Sie hier nur schon mal für das Thema Namensräume sensibilisieren. Auch wenn das Problem hier mit `cout` ziemlich offensichtlich war, sollte das Beispiel zeigen, zu welchen Problemen es bei zwei gleichen Bezeichnern im selben Gültigkeitsbereich kommen kann. In solch trivialen Beispielen hat man das Problem vielleicht im Überblick, aber bei mehreren tausend Zeilen Code und ein paar Dutzend Modulen und vielen verschiedenen Entwicklern kann es da schon mal zu Überschneidungen kommen.

Der Namensraum std

Dem Namensraum `std` ist ein extra Abschnitt gewidmet (Abschnitt 6.1.4, »Der Namensraum std«).

6.1.1 Einen neuen Namensraum erstellen

Um zu veranschaulichen, wie Sie einen neuen Namensraum erstellen, soll ein möglichst einfaches Fallbeispiel herhalten, in dem es zu Konflikten kommt. Betrachten Sie die beiden Codeausschnitte von *datei1.cpp* und *eineHeaderdatei.h*. Hier kommt es ganz offensichtlich zu Konflikten:

```
// kapitel06/0/datei1.cpp
#include <iostream>
#include <string>
#include "eineHeaderdatei.h"

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void function() {/*...*/}

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

```
// kapitel06/0/eineHeaderdatei.h
#include <string>

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void function() {/*...*/}
class aClass { /*...*/};

// ...
```

Was sofort auffallen sollte, ist, dass es hier gleichnamige Bezeichner im globalen Gültigkeitsbereich gibt, weil die Headerdatei *eineHeaderdatei.h* auch noch in *datei1.cpp* mit eingebunden wurde. Eine einfache Lösung bei solchen Minibeispielen könnte es sein, einfach andere Bezeichner zu verwenden. Aber bei ein paar tausend Zeilen bedeutet dies dann schon einen ziemlichen Aufwand. Allerdings liefert C++ für solche Zwecke ja eigene Namensräume.

Einen neuen Namensraum können Sie mit dem Schlüsselwort `namespace` wie folgt anlegen:

```
namespace NAME_FUER_NAMENSBEREICH
{
  // Deklarationen und Defintionen
}
```

Bezogen auf unsere Beispiele stehen Ihnen jetzt zwei Möglichkeiten zur Verfügung.

Möglichkeit 1

Der Header wird in den neuen Namensbereich gesetzt:

```
// kapitel06/1/datei1.cpp
#include <iostream>
#include <string>
namespace myspace {
  #include "eineHeaderdatei.h"
}

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void function() {/*...*/}

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

```
// kapitel06/1/eineHeaderdatei.h
#include <string>

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void function() {/*...*/}
class aClass { /*...*/};

// ...
```

Möglichkeit 2

Wir erstellen einen neuen Namensraum in der Quelldatei.

```
// kapitel06/2/datei1.cpp
#include <iostream>
#include <string>
#include "eineHeaderdatei.h"

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void function() {/*...*/}

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

```
// kapitel06/2/eineHeaderdatei.h
#include <string>

namespace myspace {
  std::string nname;
  std::string vname;
  unsigned short alter;
  struct Datensatz { /*...*/};
  void function() {/*...*/}
  class aClass { /*...*/};
}

// ...
```

Es spricht auch nichts dagegen, viele weitere Namensbereiche zu verwenden. Die Anzahl der Namensbereiche ist nicht beschränkt. Im Beispiel wäre es praktisch recht sinnvoll, die globalen Dinge von *datei1.cpp* auch noch in einen Namensraum zu packen:


```
// kapitel06/3/datei1.cpp
#include <iostream>
#include <string>
#include "eineHeaderdatei.h"

namespace anotherspace {
  std::string nname;
  std::string vname;
  unsigned short alter;
  struct Datensatz { /*...*/};
  void function() {/*...*/}
}

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

```
// kapitel06/3/eineHeaderdatei.h
#include <string>

namespace myspace {
  std::string nname;
  std::string vname;
  unsigned short alter;
  struct Datensatz { /*...*/};
  void function() {/*...*/}
  class aClass { /*...*/};
}

// ...
```

Die Anzahl der Namensbereiche ist nicht beschränkt.

Deklarationen und Definitionen trennen

In der Praxis wird auch hier gewöhnlich die Deklaration von Funktionen, Strukturen, Klassen usw. von der Definition getrennt. Die Definition erstellt man gewöhnlich in einer anderen Quelldatei. Auch hierbei bieten sich zwei verschiedene Möglichkeiten an, die Deklaration und Definition eines Namensbereichs zu trennen.

Möglichkeit 1: Deklarationen und Definitionen voneinander trennen

Deklarationen

```
#include <string>
...
namespace myspace {
  std::string nname;
  std::string vname;
  unsigned short alter;
  struct Datensatz;
  void funktion();
  class aClass;
}

// ...
```

Definitionen

```
...
// Definition von Datensatz
struct myspace::Datensatz {
  // Typen ...
};
// Definition von funktion()
void myspace::funktion() {
  // Code für funktion
}
// Definition von aClass
class myspace::aClass {
  // Code für Klasse
};

// ...
```

Möglichkeit 2: Deklarationen und Definitionen voneinander trennen, indem einfach erneut der Namensraum geöffnet wird

Deklarationen

```
#include <string>
...
namespace myspace {
  std::string nname;
  std::string vname;
  unsigned short alter;
  struct Datensatz;
  void funktion();
  class aClass;
}

// ...
```

Definitionen

```
...
// Namensbereich erneut öffnen
namespace myspace {
  // Definition von Datensatz
  struct Datensatz {
    // Typen für Datensatz
  };
  // Definition von funktion()
  void funktion() {
    // Code für funktion()
  }
  // Definition von aClass
  class aClass {
    // Code für aClass
  };
}

// ...
```

6.1.2 Namensraum verwenden

Namensräume sind eigene Gültigkeitsbereiche, wie Sie diese in Abschnitt 5.4, »Exkurs: Gültigkeitsbereiche«, kennengelernt haben. Es gibt jetzt drei Möglichkeiten, wie Sie nun auf einen Namensbereich zugreifen können. Als Beispiel soll folgender, sehr einfach gehaltener Code verwendet werden.

```
// kapitel06/datei1-0.cpp
#include <iostream>
#include <string>
#include "eineHeaderdatei.h"

std::string nname;
std::string vname;
unsigned short alter;
struct Datensatz { /*...*/};
void function() {/*...*/}

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

```
// kapitel06/eineHeaderdatei.h
#include <string>

namespace myspace {
  std::string nname;
  std::string vname;
  unsigned short alter;
  struct Datensatz { /*...*/};
  void function() {/*...*/}
  class aClass { /*...*/};
}

// ...
```


Mithilfe des Quellcodes sollen drei Möglichkeiten demonstriert werden, um auf einen Namensraum zuzugreifen.

Möglichkeit 1: Mit Bereichsoperator (Scope)

Der Zugriff mithilfe des Scope-Operators `::` (oder auch Bereichsoperators) dürfte Ihnen ja relativ geläufig sein, weil Sie diesen Zugriff bereits bei sämtlichen Beispielen mit `std::` verwendet haben. Um einzelne Bezeichner aus einem Namensraum verwenden zu können, müssen Sie den Namensbereich, gefolgt vom Scope-Operator, und dann den Bezeichner angeben:

```
Namensbereich::Bezeichner
```

Bezogen auf unser Beispiel könnte dies jetzt wie folgt aussehen:

```
00  // kapitel06/datei-1-1.cpp
...
01  #include "eineHeaderdatei.h"
...
02  int main() {
03    myspace::nname = "Mustermann";
04    myspace::vname = "Max";
05    nname = "Musterfrau";
06    vname = "Maxima";
07    std::cout << myspace::nname << std::endl;
08    std::cout << nname << std::endl;
09    myspace::aClass testClass;
...
10  }
```

Um auf Namensbereiche zuzugreifen, müssen Sie dem Projekt natürlich entsprechende Dateien hinzufügen. Im Beispiel wurde alles recht einfach in eine Headerdatei verpackt, welche wir hier auch in Zeile **01** hinzugefügt haben. In den Zeilen **03** und **04** erfolgt ein Zugriff auf die Variablen `nname` und `vname` aus dem Namensbereich `myspace`, welche in der Headerdatei *eineHeaderdatei.h* deklariert und definiert wurden. Der Zugriff in den Zeilen **05** und **06** auf die gleichnamigen Bezeichner `nname` und `vname` erfolgt dann auf die globalen Variablen aus der Quelldatei *datei1.cpp*. In Zeile **09** wird ein Objekt vom Typ `aClass` aus dem Namensraum `myspace` definiert.

Möglichkeit 2: Einzelne Bezeichner importieren

Die zweite Möglichkeit ist es, nur einzelne Bezeichner aus einem Namensraum mit dem Schlüsselwort `using` zu importieren, das wird auch als `using`-Deklaration bezeichnet:

```
using Namensraum::Bezeichner;
```

In der Praxis in unserem Quellcode lässt sich dies bspw. wie folgt realisieren:

```
00  // kapitel06/datei-1-2.cpp
...
01  #include "eineHeaderdatei.h"
...
02  int main() {
03    using myspace::nname;
04    using myspace::vname;
05    using myspace::aClass;
06    nname = "Mustermann";
07    vname = "Max";
08    ::nname = "Musterfrau";
09    ::vname = "Maxima";
10    aClass testClass;
11    std::cout << nname << std::endl;
12    std::cout << ::nname << std::endl;
13  }
```

Mithilfe der drei `using`-Deklarationen in den Zeilen **03** bis **05** innerhalb der `main`-Funktion stehen die Bezeichner `nname`, `vname` und `aClass` innerhalb der `main`-Funktion lokal zu Verfügung und können ohne Scope-Operator verwendet werden. Da es allerdings globale Funktionen mit denselben Bezeichnern gibt, würde es zu einem Konflikt kommen. Hier können wir uns in den Zeilen **08** und **09** nochmals mit dem alleinigen Scope-Operator (`::Bezeichner`) behelfen, um explizit auf die globalen Bezeichner zuzugreifen.

Globale using-Deklaration

Natürlich ist es auch möglich, die `using`-Deklaration global außerhalb einer Funktion zu machen. Allerdings würde es in diesem Fall dann tatsächlich zu Problemen mit den gleichnamigen globalen Bezeichnern kommen. Hier könnten Sie sich dann auch nicht mehr mit einem alleinigen Scope-Operator (`::Bezeichner`) retten. Ich denke, spätestens jetzt dürfte es Ihnen dämmern, dass es mit globalen Variablen irgendwann immer zu Problemen kommt und warum es besser ist, wenn möglich auf globale Sachen zu verzichten. Daher wäre es hier auch ideal, die globalen Bezeichner wiederum in einen Namensbereich zu packen.

Möglichkeit 3: Alles importieren

Die dritte Möglichkeit wäre es, den kompletten Namensraum zur Verfügung zu stellen. Damit wäre es möglich, ohne Umwege direkt den Bezeichner zu verwenden – also explizite Anwendung über den Scope-Operator:


```
using namespace Namensraum;
```

Bezogen auf unser Beispiel wird es schwer, den Namensraum komplett mit `using namespace myspace;` zur Verfügung zu stellen, weil das zwangsläufig zu Konflikten mit den globalen Bezeichnern führen würde. Hier kommen Sie daher nicht mehr um den Scope-Operator herum, um Mehrdeutigkeiten zu vermeiden:

```
00  // kapitel06/datei-1-3.cpp
...
01  #include "eineHeaderdatei.h"
...
02  int main() {
03    using namespace myspace;
04    myspace::nname = "Mustermann";
05    myspace::vname = "Max";
06    ::nname = "Musterfrau";
07    ::vname = "Maxima";
08    aClass testClass;

09    std::cout << myspace::nname << std::endl;
10    std::cout << ::nname << std::endl;
11  }
```

In Zeile **03** wird der komplette Namensraum von `myspace` importiert und innerhalb der `main`-Funktion zur Verfügung gestellt. Damit es hierbei nicht zu Namenskonflikten mit den gleichnamigen globalen Bezeichnern kommt, mussten in den Zeilen **04** und **05** trotzdem die Zugriffsoperatoren verwendet werden. Nur die Klasse in Zeile **08** konnte problemlos ohne einen Scope-Zugriff verwendet werden, weil es da keinen gleichnamigen globalen Bezeichner gibt.

Namensbereich ist ein echter Gültigkeitsbereich

An dieser Stelle muss nochmals deutlich gesagt werden, dass ein Namensbereich ein echter Gültigkeitsbereich ist. Die Gültigkeit einer `using`-Deklaration oder eines kompletten `using`-Namensraumes beginnt und endet immer innerhalb eines Anweisungsblocks ab `{` bis `}`. Deshalb ist es von enormer Bedeutung, wo Sie die `using`-Anweisungen platzieren, um einen eindeutigen Namen innerhalb eines Gültigkeitsbereichs zu erhalten. Wenn Sie `using`-Anweisungen außerhalb einer Funktion global platzieren, werden alle Bezeichner eines Namensraumes im kompletten Programm sichtbar. Hierbei sollten Sie sich schon Ihre Gedanken, machen! Warum zunächst etwas in einen Namensraum stecken, um es dann wieder öffentlich zur Verfügung zu stellen?

Zugriff auf verschachtelte Namensräume

Vollständigkeitshalber soll hier auch noch erwähnt werden, dass es durchaus möglich ist, Namensbereiche zu verschachteln, wie bspw.:

```
namespace outerSpace {
  // ...
  namespace innerSpace {
    void funktion();
  }
}
```

Der Zugriff auf `funktion()`, welche sich innerhalb vom Namensbereich `innerSpace` befindet, der sich wiederum innerhalb von `outerSpace` befindet, muss dann vollständig qualifiziert über `outerSpace::innerSpace::funktion()` erfolgen.

6.1.3 Aliases für Namensräume

Zu lange und umständliche Namen eines Namensraumes können Sie auch mit einem anderen Namen (um-)benennen oder verbergen. Ein einfaches Beispiel dazu:

```
01  namespace my_own_namespace {
02     unsigned int ival;
03     // ...
04  }

05  namespace mon = my_own_namespace;
...
06  using namespace mon;
...
07  mon::ival = 1234;
08  std::cout << my_own_namespace::ival << std::endl;
...
```

In den Zeilen **01** bis **04** wird ein Namensbereich mit dem Namen `my_own_namespace` erstellt. In Zeile **05** erstellen wir mit `mon` einen Aliasnamen für `my_own_namespace`. Sie können jetzt quasi `mon` und `my_own_namespace` verwenden. In Zeile **06** würden Sie alle Bezeichner aus `mon`, alias `my_own_namespace`, importieren. Der Zugriff auf die einzelnen Bezeichner des Namensbereichs kann jederzeit über `mon` (siehe Zeile **07**), aber auch über das ursprüngliche `my_own_namespace` (siehe Zeile **08**) erfolgen.

6.1.4 Der Namensraum std

An dieser Stelle muss noch auf den Namensraum `std` eingegangen werden, welchen wir in diesem Buch in fast jedem Beispiel mit `std::Bezeichner` verwendet haben.

Sehr häufig anzutreffen ist immer noch die Form, wo alles global importiert und zur Verfügung gestellt wird. In der Praxis sollten Sie sich aber bereits vorher darüber Gedanken machen, ob Sie den kompletten Inhalt von `std` verfügbar machen wollen, weil in der echten Praxis die Beispiele eher nicht so trivial wie in Büchern oder Tutorials sind. Gerne werden in eigenen Modulen die Bezeichner wie `count`, `sort`, `find`, `swap` oder ähnlich verwendet, und schnell hat man einen Konflikt, weil der Namensraum `std` ebenfalls solche Bezeichner enthält. Zugegeben, wenn man nur die Standardbibliothek verwendet, hat man alles noch im Griff. Aber spätestens wenn Sie eine von den unzähligen Bibliotheken verwenden, welche von anderen Programmierern oder bei einem Projekt zur Verfügung gestellt werden, kann (meistens wird) es immer wieder zu Namenskonflikten kommen.

Wie Sie bereits erfahren haben, stehen in der Praxis drei verschiedene Möglichkeiten zur Verfügung, die Bezeichner aus anderen Namensbereichen zu verwenden. Hier nochmals die drei Möglichkeiten in Bezug auf `std`:

1. Sie verwenden den Namensraum mit dem Bezeichner direkt im Quellcode. Statt bspw. `cout` für die Ausgabe verwenden Sie `std::cout`. Der Vorteil ist hierbei, dass es niemals mehr zu einem Konflikt gleicher Bezeichner kommen kann. In der Praxis sieht dies dann wie folgt aus:

   ```
   std::cout << "Namensbereich::" << std::endl;
   ```

2. Sie verwenden eine `using`-Deklaration, womit Sie nur einzelne Bezeichner eines Namensraumes öffentlich machen. Schreiben Sie diese `using`-Deklaration dann auch noch schön brav in einen Anweisungsblock, dann sind diese Bezeichner nur in diesem Anweisungsblock gültig. In der Praxis sieht dies wie folgt aus:

   ```
   {
     using std::cout;
     using std::endl;
     cout << "Namensbereich::" << endl;
   ...
   }
   cout << "Namensbereich::" << endl; // Fehler!!!
   ```

   Durch die die beiden `using`-Direktiven in diesem Beispiel sind `cout` und `endl` innerhalb des Anweisungsblocks sehr komfortabel ohne den Namensbereich und Zugriffsoperator erreichbar.

3. Sie verwenden die bereits bekannte `using`-Direktive mit `namespace std` für den gesamten Namensraum `std`. Setzen Sie diese Direktive an einem globalen Bereich, sind alle Bezeichner überall verfügbar, und die Gefahr von Namenskonflikten ist am größten. Sie können dies aber auch ein wenig eingrenzen, indem Sie die `using`-Direktive innerhalb eines Anweisungsblocks schreiben. Dann sind die Bezeichner nur innerhalb des Anweisungsblocks sichtbar, wie folgendes Beispiel zeigen soll:

```
{
    using namespace std;
    cout << "Namensbereich::" << endl;  // OK
}
cout << "Namensbereich::" << endl; // Fehler!!!
```

Innerhalb des Anweisungsblocks sind alle Bezeichner des Namensbereichs `std` bei diesem Beispiel verwendbar. Aber außerhalb des Anweisungsblocks sind die Bezeichner wieder unbekannt, weil der Namensraum `std` hier nicht mehr gültig ist.

Was ist jetzt die beste Lösung?

Das ist auch eine Frage des persönlichen Stils. Auf keinen Fall empfehlen kann ich eine `using`-Deklaration oder den kompletten Namensbereich mit `using` global zu stellen. Es macht einfach keinen Sinn, erst alles in einen Namensbereich zu stecken, um anschließend alles wieder global zur Verfügung zu stellen. Sicherer ist es da schon, einzelne Bezeichner oder gar den kompletten Namensraum innerhalb eines Anweisungsblocks zu importieren. Oder aber Sie verwenden gleich den Namensraum mit Bezeichner direkt im Code (`Namensraum::Bezeichner`). Dies bedeutet natürlich das geringste Namenskonfliktrisiko, aber die meiste »Schreibarbeit«.

Für manche wiederkehrenden Aufgaben hat sich die Form `using std::BEZEICHNER` innerhalb von Funktionen durchgesetzt, so z.B., wenn man `swap()` für eigene Objekte implementieren möchte – ein `using std::swap;` zu Beginn der eigenen Funktion macht den Bezeichner standardmäßig innerhalb der eigenen Funktion bekannt.

6.2 Speicherklassenattribute

Mit den Schlüsselwörtern `extern` und `static` legen Sie fest, wo die Typen im Speicher abgelegt werden und/oder wie lange diese dort gültig sind. Hier gibt es noch weitere Schlüsselwörter wie `mutable` und `thread_local`, deren Bedeutungen aber erst im entsprechenden Kapitel erläutert werden.

6.2.1 Das Schlüsselwort extern

Mit dem Schlüsselwort `extern` weiß der Compiler, dass der Linker das Speicherobjekt in einer anderen Übersetzungseinheit (Quell- oder Headerdatei) auflösen muss. Wenn Sie bspw. eine globale Variable `ival` in der Quelldatei *extern01.cpp* definieren und auch noch in der Datei *extern02.cpp* verwenden wollen, können Sie diese Variablen in *extern02.cpp* nicht erneut definieren, weil sich der Compiler sonst beschweren würde. Mithilfe des Schlüsselwortes `extern` vor der Variablen `ival` in *extern02.cpp*

weiß der Compiler jetzt, dass sich der Linker um die Auflösung kümmert. Die zwei Dateien, welche dies demonstrieren sollen, sehen Sie hier:

```
// kapitel06/extern01.cpp
#include <iostream>

int var=0;    // Definition
void print(); // Deklaration

int main() {
  var = 123456;
  print();
}
```

```
// kapitel06/extern02.cpp
#include <iostream>

extern int var; // Deklaration

void print() {
  std::cout << var
            << std::endl;
}
```

Zugegeben, das ist kein schönes Beispiel, aber es demonstriert das Schlüsselwort `extern` sehr gut. Würden Sie `extern` in der Quelldatei *extern02.cpp* entfernen, würde der Linker beim Auflösen der Symbole die Übersetzung verweigern, weil zwei gleichnamige Bezeichner definiert werden müssten. Probieren Sie es selbst aus.

Beide Dateien beim Übersetzen angeben

Da Sie hier mit *extern01.cpp* und *extern02.cpp* zwei Quelldateien haben, müssen Sie beide Quelldateien bei der Übersetzung mit angeben, um daraus eine ausführbare Datei zu machen. Auf das Übersetzen mehrerer Quelldateien wird noch gesondert in Abschnitt 6.6, »Modulare Programmierung – Code organisieren«, eingegangen.

Im Beispiel konnten Sie sehr schön erkennen, dass Sie mit dem Schlüsselwort `extern` die Möglichkeit haben, eine Variable nur zu deklarieren. Dies entspricht in etwa der Deklaration des Funktionsprototyps in der Datei *extern01.cpp* mit `void print();`. Theoretisch **könnten** Sie hier vor der Deklaration des Funktionsprototyps auch das Schlüsselwort `extern` stellen, um die Lesbarkeit des Programms zu erhöhen – nämlich um dem Leser mitzuteilen, dass die Definition woanders erfolgt.

»extern« bei der Funktionsdeklaration nicht nötig

Bei Funktionsdeklarationen muss das Schlüsselwort `extern` nicht zwangsläufig verwendet werden (ist aber auch nicht falsch). Deklarationen von Funktionsprototypen werden vom Compiler implizit automatisch als `extern` behandelt, wenn nicht noch die Definition in der gleichen Datei folgt.

Natürlich könnten Sie es sich jetzt auch ganz einfach machen und statt *extern02.cpp* eine Headerdatei verwenden, in der Sie die Definition der Variablen und der Funk-

tion schreiben, und diese Headerdatei dann in *extern01.cpp* inkludieren. Genau genommen teilt man auch hier wiederum die Deklaration im Header von der Definition, aber das soll jetzt hier noch nicht behandelt werden, sondern erst – dann ausführlicher – in Abschnitt 6.6.

Tipp: Auf globale Variablen wenn möglich verzichten

Wenn es möglich ist, sollten Sie auf globale Variablen (und Objekte) verzichten. Damit ersparen Sie sich seine Menge an Zeit, die Sie in die Suche nach schwer auffindbaren Fehlern investieren müssten, wenn der schlechteste Fall eintritt.

6.2.2 Das Schlüsselwort static

Das Schlüsselwort `static` kann auf zwei unterschiedlichen Wege verwendet werden. Je nachdem, ob das Schlüsselwort global oder lokal verwendet wird, ändert sich die Bedeutung oder auch hier der Gültigkeitsbereich.

Globales static

Wenn Sie `static` vor einer globalen Variablen oder Funktion setzen, beschränkt sich die Gültigkeit auf das Modul (die Datei). Damit könnten Sie Namenskonflikte zwischen gleichnamigen Bezeichnern in unterschiedlichen Dateien vermeiden. Eine mit `static` gekennzeichnete Variable wird außerdem auch automatisch mit 0 initialisiert. Betrachten Sie hier folgendes fast bekannte Listing:

```
// kapitel06/static01.cpp
#include <iostream>

int var=0;    // Definition
void print(); // Deklaration

int main() {
  var = 123456;
  print();
}
```

```
// kapitel06/static02.cpp
#include <iostream>

static int var;  // Definition

void print() {
  std::cout << var << std::endl;
}
```

Das Beispiel zeigt, wie es mit `static` für globale Bezeichner schnell zu Doppeldeutigkeiten kommen kann und wie es häufig eher verwirrt, als etwas deutlich zu machen.

Im Gegensatz zu den Beispielen von *extern01.cpp* und *extern02.cpp* aus dem vorherigen Abschnitt wurde in diesem Beispiel das Schlüsselwort `static` nur bei *static02.cpp* vor die Variable `var` gesetzt (statt des Schlüsselwortes `extern`). Damit haben Sie jetzt

in *static02.cpp* eine Variable `var` definiert (welche auch automatisch mit 0 initialisiert wurde), die jetzt nur innerhalb der Datei *static02.cpp* gültig ist. Die andere globale Variable `var` aus der Datei *static01.cpp* weiß nichts von dieser mit `static` gekennzeichneten Variablen, und es kommt somit auch zu keinem Konflikt mehr.

Leider hat das Beispiel eine Doppeldeutigkeit, und es ist nicht zu erkennen, was der Programmierer hier erreichen wollte. Die Wertzuweisung von `var=123456` in *static01.cpp* hat keinerlei Effekt mehr auf die folgende Ausgabe mit `print()`, weil die Funktion `print()` in *static02.cpp* definiert wurde und dort eben das (Datei-)lokalere `var` – hier also das `static int var` – verwendet. Mit `var=123456` weisen Sie allerdings den Wert der globalen Variablen `var` von *static01.cpp* zu.

Tipp: Verzichten Sie auf globales static

Aufgrund der Doppeldeutigkeiten, welche beim globalen `static` auftreten können, empfehle ich Ihnen, ganz darauf zu verzichten. Wenn Sie den Gültigkeitsbereich innerhalb einer Datei beschränken müssen, können Sie stattdessen auch auf einen anonymen Namensraum zurückgreifen.

Den anonymen Namensbereich habe ich Ihnen noch vorenthalten. Das Thema um die Namensbereiche wurde ja bereits in Abschnitt 6.1, »Namensräume«, beschrieben. Hierzu sehen Sie eine gleichwertige Alternative zum globalen `static`, um mithilfe von anonymen Namensräumen eine Variable ebenfalls nur innerhalb einer Datei gültig zu machen.

```
// kapitel06/static03.cpp
#include <iostream>

static int file_local;

int main() {
  file_local = 123456;
}
```

```
// kapitel06/static04.cpp
#include <iostream>

namespace {
  int file_local;
}

int main() {
  file_local = 123456;
}
```

Im linken Beispiel (*static03.cpp*) wird das Schlüsselwort `static` verwendet, um die Gültigkeit der Variablen `file_local` auf die Datei zu beschränken. Dasselbe erreichen Sie jetzt auch (siehe *static04.cpp*), wenn Sie eine Variable bzw. Funktion innerhalb eines leeren `namespace {}` schreiben. Alles, was zwischen `{` und `}` im anonymen Namensraum steht, ist ebenfalls nur innerhalb der Datei gültig.

Global, static global oder anonym

Zugegeben in diesem Abschnitt wurde das Thema recht umfassend und vielseitig behandelt. Daher soll noch erwähnt werden, dass im Fall der Fälle eine dateilokale `static`-Variable einer globalen Variablen vorzuziehen ist, weil sich die Fehlersuche hiermit zumindest von allen Modulen auf eine Datei reduziert. Trotzdem soll dies hier nicht heißen, dass dateiglobale Sachen so viel besser sind. Und da `static` (als lokales `static`) noch eine weitere Funktion hat, wäre es vielleicht besser, für dateilokale Sachen einen anonymen Namensraum zu verwenden.

Lokales static

Wie bereits eben erwähnt, hat das Schlüsselwort `static` noch eine weitere, recht andere Funktion, welche sich zwar auch wieder auf das Thema Gültigkeitsbereich bezieht, aber eben häufig für Verwirrung sorgt, weil es eben eine andere Bedeutung bekommt. Wenn Sie eine `static`-Variable innerhalb eines Anweisungsblocks (bspw. Funktion) verwenden, bekommt diese Variable einen festen Platz im Datensegment und bleibt somit zur Laufzeit des Programms dort enthalten. Innerhalb einer Funktion bleibt eine mit `static` gekennzeichnete Variable für immer gültig und behält sogar ihren Wert nach dem Ende der Funktion – so dass bei erneutem Funktionsaufruf der beim letzten Funktionsaufruf gesetzte Wert vorhanden ist. In der Praxis ist damit quasi auch möglich, ein mit `static` gekennzeichnetes (lokales) Speicherobjekt aus einer Funktion zurückzugeben, das ja mit einer gewöhnlichen lokalen Variablen auf dem Stapelspeicher (Stack/-Segment) nicht erlaubt ist, weil dieses nach dem Ende der Funktion zerstört würde.

Lokaler Gültigkeitsbereich bleibt allerdings

Hier sollte noch erwähnt werden, dass eine mit `static` gekennzeichnete lokale Variable trotzdem nach wie vor nur innerhalb des Gültigkeitsbereichs (Anweisungblock) gültig ist, wo diese definiert wurde.

Hierzu ein einfaches Beispiel, welches das lokale `static` als Gedächtnis einer Funktion verwendet:

```
00  // kapitel06/static05.cpp
01  #include <iostream>

02  int pseudozufallszahl(int x);

03  int main() {
04    std::cout << pseudozufallszahl(11111) <<  std::endl;
```



```
05    std::cout << pseudozufallszahl(11111) <<  std::endl;
06    std::cout << pseudozufallszahl(11111) <<  std::endl;
07  }

08  int pseudozufallszahl(int x) {
09    static int startwert = 246532;
10    if( x % 2)
11      startwert+=x;
12    else
13      startwert-=x;
14    return startwert;
15  }
```

Wenn Sie das Beispiel *static05.cpp* ausführen, werden Sie vielleicht überrascht sein, dass in den Zeilen **04** bis **06** dreimal unterschiedliche Werte ausgegeben werden, obwohl die Funktion `pseudozufallszahl()` immer mit demselben Wert aufgerufen wird. Diesen Umstand verdanken Sie der statischen Variablen `startwert` in Zeile **09**, welche sich ja bei jedem erneuten Funktionsaufruf den Wert des vorherigen Funktionsaufrufes gemerkt hat und somit immer mit dem zuvor verwendeten Wert weiterarbeitet. Dies ist nur dadurch möglich, weil sich diese Variable im Datensegment und nicht wie lokale Variablen ohne `static` auf dem Stacksegment befindet. Die Zeilen **10** bis **13** sind nur ein Beiwerk, um den übergebenen Wert `x`, wenn dieser ungerade ist, zum `startwert` hinzuzuaddieren oder bei ungeradem Wert zu subtrahieren.

Genauer Zeitpunkt der Initialisierung

Lokale `static`-Variablen werden genau dann initialisiert, wenn der Code, in dem diese stehen, das erste Mal ausgeführt wird. Globale Variablen hingegeben sind (oder werden) alle schon beim Programmstart initialisiert.

6.2.3 Aus Alt mach Neu und »deprecated«

In Abschnitt 2.5, »Typ auto (C++11)«, haben Sie ja bereits die neue Bedeutung des alten Schlüsselwortes `auto` im neuen C++11-Standard kennengelernt. Die alte Bedeutung von `auto` als Speicherklasse war ohnehin ziemlich überflüssig, weil Variablen innerhalb von Funktionen schon `auto` waren und außerhalb nicht erlaubt sind. Daher wurde die alte Verwendung von `auto` entfernt (C++03) und einer neuen Bedeutung zugeordnet (C++11). Daher ist das alte `auto` eines der wenigen Dinge in C++, welche im neuen C++11-Standard nicht abwärtskompatibel sind.

Deprecated: Schlüsselwort register

Früher konnte man noch mit dem Schlüsselwort `register` dem Compiler vorschlagen, diese Variable in das schnellere Prozessorregister statt den langsameren Arbeitsspeicher zu legen. Aber da der Compiler letztendlich selbst entscheidet, was er in das Prozessorregister legt, war auch dieses Schlüsselwort recht überflüssig und wurde mit C++11 zu »deprecated« erklärt, was bedeutet, dass Sie dieses Schlüsselwort für künftige Projekte nicht mehr verwenden sollten.

6.3 Typqualifikatoren

Es gibt mit `const` und `volatile` zwei Typqualifikatoren, mit denen Sie bei einer Deklaration die Typen weiter modifizieren können. Wobei `volatile` in der Praxis wohl eher seltener benötigt wird.

6.3.1 Das Schlüsselwort const

Speicherobjekte, welche Sie mit `const` gekennzeichnet haben, können nach ihrer Definition im Speicher nicht mehr verändert werden. Das Objekt wird damit mit einem Schreibschutz versehen. Jeglicher Versuch, ein mit `const` geschütztes Objekt zu verändern, führt zu einer Fehlermeldung des Compilers.

Doch keine echte Kontante?

Ein Objekt, welches mit `const` gekennzeichnet wird, legt, um es recht genau zu nehmen, nicht wirklich eine echte »physische« Konstante an. Mit `const` wird ein Objekt lediglich als nicht mehr änderbar (*read-only*) markiert und darf somit nicht mehr auf der linken Seite einer Zuweisung stehen. Versuchte Änderungen des Speicherobjekts, welches mit `const` markiert wurde, würde der Compiler sofort bemäkeln. Es ist mit `const_cast<>` trotzdem möglich, u. U. diesen Schreibschutz des Compilers zu »überkleben«.

Besonders häufig wird `const` bei Parametern von Funktionen (und später auch Methoden) verwendet, um ein versehentliches Verändern eines Objekts innerhalb einer Funktion zu verhindern (siehe auch Abschnitt 5.5, »Funktionsparameter«). Mit einem Funktionskopf wie `void funktion(const int& val);` stellen Sie praktisch sicher, dass `val` in der Funktion nicht mehr versehentlich geändert werden kann.

Hierzu ein kurzes Beispiel mit `const`:


```
00 // kapitel06/const01.cpp
01 #include <iostream>
02 #include <string>

03 int quad(const int x);

04 int main() {
05   const std::string r_str = "Text mit Schreibschutz";
06   r_str = "Fehler!!!";  // !!! Fehler !!!
07   std::cout << quad(10) << std::endl;
08 }

09 int quad(const int x) {
10   x*=x;  // !!! Fehler !!!
11   return x;
12 }
```

In Zeile **05** kennzeichnen Sie `r_str` mit `const` und versehen den String damit mit einem Schreibschutz. Der Versuch, dieses Objekt in Zeile **06** trotzdem auf die linke Seite einer Zuweisung zu stellen, wird der Compiler mit einem deutlichen »Kopfschütteln« quittieren. Ebenso sieht es in der Funktion `quad()` (Zeile **09** bis **12**) aus, wo in Zeile **10** versucht wird, den Wert `x` zu ändern, der im Funktionskopf als `const` qualifiziert wurde. Das `*=` stellt ja auch eine erweiterte Form der Zuweisung dar. Somit wird sich der Compiler über diese Zeile (**10**) beschweren. Den Fehler können Sie ganz einfach beheben, indem Sie Zeile **10** entfernen und in Zeile **11** einfach `return x*x;` tippen.

C++11: Stärkeres const mit constexpr

Mit C++11 wurde mit `constexpr` ein noch stärkeres `const` eingeführt. Allerdings ist `constexpr` schon ein bisschen mehr als nur ein einfaches `const`, weshalb dies gesondert im Abschnitt »constexpr« in Abschnitt 6.4.3 behandelt wird.

extern bei const

Wenn Sie `const`-Objekte global definieren, sind diese, anders als bei gewöhnlichen globalen Objekten, nur in der Quelldatei sichtbar, in der diese definiert wurden. Wenn Sie daher ein globales mit `const` qualifiziertes Speicherobjekt überall sichtbar machen wollen, müssen Sie es zusätzlich noch mit dem Schlüsselwort `extern` kennzeichnen. Betrachten Sie hierzu das folgende Beispiel:

```
// kapitel06/extconst01.cpp
#include <iostream>

extern const double pi=3.1415;
void print(); // Deklaration

int main() {
  print();
}
```

```
// kapitel06/extconst02.cpp
#include <iostream>

extern const double pi;

void print() {
  std::cout << pi
            << std::endl;
}
```

Mit `const` qualifizierte Objekte sind nur in der Quelldatei gültig, in der diese definiert wurden. Sollen diese auch in anderen Dateien sichtbar sein, müssen Sie sie zusätzlich noch mit `extern` deklarieren.

In der Datei *extconst01.cpp* wird `pi` deklariert und definiert. Wenn diese globale `const`-Variable künftig auch in anderen Dateien zur Verfügung stehen soll, muss das Schlüsselwort `extern` davorgestellt werden. In der Datei *exconst02.cpp* finden Sie hingegen nur die Deklaration (ohne Definition), welche nötig ist, damit der Linker einen Verweis auf die globale `const`-Variable erhält und sie in der Definition der Funktion `pi` verwendet werden kann.

6.3.2 Das Schüsselwort volatile

`volatile` wird für eine tiefere Ebene benötigt und wohl von den wenigsten je benutzt werden. Eine mit `volatile` qualifizierte Variable kann von außerhalb des laufenden Programms einen neuen Wert erhalten. Bei der Erstellung von Maschinencode verhindert diese Qualifizierung einer Variablen mit `volatile` eine für die Funktion beeinträchtigende Optimierung, so dass im Programm immer auf den in der Hardware tatsächlich vorhandenen Wert zugegriffen wird.

In normalen Programmen sind solche Optimierungsgründe durchaus berechtigt und sinnvoll. Da legt der Prozessor schon mal gerne eine Variable in seinen Zwischenspeicher (Cache). Bei der Hardwareprogrammierung (wie Treibern) kann ein derartiges Verhalten aber katastrophal sein, weil sich im Cache noch die alten Zustände der Hardware befinden könnten. Da Sie sich nicht darauf verlassen können, dass die zwischengespeicherte Variable noch aktuell ist, erzwingen Sie mit `volatile`, dass die Variable frisch aus dem Hauptspeicher eingelesen wird.

Ein Beispiel zu `volatile` zu schreiben, bringt an dieser Stelle jetzt nicht viel. Daher soll folgender theoretische Code den Sachverhalt kurz demonstrieren:

```
volatile int *status;
...
for(volatile int i = 0; i < port_n; i++ ) {
```



```
   ...
   *status = aktuell[i];
}
```

Ohne `volatile` könnte der Compiler hier bspw. die Zählvariable `i` und auch `port` in den Zwischenspeicher (Cache) des Prozessors legen, um die Laufzeit des Programms zu optimieren. Wenn Sie allerdings die Werte zur Laufzeit der Schleife ändern, wäre das für einen Hardwaretreiber eine Katastrophe. Dank `volatile` stellen Sie sicher, dass alle Zugriffe auf `port` so verlaufen, wie Sie es wollen, und es wird keine Optimierungen geben, so dass die Werte immer frisch aus dem Hauptspeicher gelesen werden.

6.4 Spezielle Schlüsselwörter

C++ hat noch weitere Schüsselwörter anzubieten. Einige dieser Schlüsselwörter bekommen erst im Zusammenhang mit Klassen (ab Kapitel 7, »Grundlagen zu den Klassen«) einen Sinn und werden daher auch zu gegebener Zeit etwas umfassender beschrieben. Andere Schlüsselwörter wurden neu mit dem C++11-Standard eingeführt.

6.4.1 Das Schlüsselwort inline für Funktionen

Mit dem Schlüsselwort `inline` können Sie die Eigenschaft einer Funktion ändern. Allerdings wurde das Schlüsselwort bereits in Abschnitt 5.8.1, »Inline-Funktionen«, umfassend beschrieben, weshalb Sie bei Bedarf bitte dort nachschlagen.

6.4.2 Schlüsselwörter für Klassen (teilweise C++11)

Auch für Klassen gibt es mehrere Schlüsselwörter. Da wären zum einen die Schlüsselwörter `virtual` und `explicit`, welche im Zusammenhang mit Klassen und Methoden stehen. Die Beschreibung dazu finden Sie daher an passender Stelle im Buch. Das Schlüsselwort `virtual` wird ab Abschnitt 9.7, »Polymorphie mithilfe von virtuellen Methoden«, verwendet und beschrieben. Die Bedeutung von `explicit` lernen Sie im Abschnitt »Implizite Konvertierung vermeiden – explicit« in Abschnitt 7.4.2 kennen.

Neue Schlüsselwörter mit C++11

Neu in C++11 hinzugekommen sind die Schlüsselwörter `override` und `final`, welche im Zusammenhang mit `virtual` verwendet werden. Beide Schlüsselwörter werden in den Abschnitten 9.9.5, »Überschreiben erzwingen mit override (C++11)«, bzw. 9.9.6, »Nicht mehr überschreiben mit final (C++11)«, umfassender beschrieben.

Ebenso mit C++11 sind die Schlüsselwörter `delete` und `default` eingeführt worden. Genau genommen sind diese Schlüsselwörter nicht echt neu, sondern nur zusätzlich seit C++11 belegt worden. Beide Schlüsselwörter werden in Abschnitt 7.9, »Erzeugen von Methoden steuern (C++11)«, erläutert.

6.4.3 Neue Schlüsselwörter mit C++11

In diesem Abschnitt werden Sie die neu in C++11 eingeführten Schlüsselwörter `auto`, `decltype` und `constexpr` etwas besser kennenlernen.

auto

In Abschnitt 2.5, »Typ auto (C++11)«, wurde dieses Schlüsselwort bereits beschrieben, nämlich wo es dem Programmierer als Typinferenz das Arbeiten vereinfachen kann. Später, wenn Sie außerdem komplexere Typen, wie bspw. umständliche lange Iteratoren (siehe Abschnitt 12.4, »Iteratoren«), verwenden, werden Sie das neue `auto` schätzen, wenn Sie statt immer wieder `std::map<std::string, int>::const_iterator` einfach nur `auto` hinschreiben können. In Abschnitt 5.9, »Die neue Funktionssyntax (C++11)«, haben Sie dann auch noch ein weiteres Anwendungsgebiet von `auto` kennengelernt, nämlich als neue Syntax für Funktionsdeklarationen mit nachgestelltem Rückgabetyp.

decltype

Mit `decltype` können Sie sich die Typinformationen aus einem Ausdruck extrahieren lassen. Auf diese grundlegende Funktion wurde bereits in Abschnitt 5.9.2, »decltype«, eingegangen.

constexpr

Mit `constexpr` (kurz für *const expression*, also konstanter Ausdruck) wurde ein Schlüsselwort eingeführt, mit dem es möglich ist, konstante Werte berechnen zu lassen. Im Gegensatz zu `const` funktioniert dies auch, wenn der Wert aus einer Funktion oder gar einem Konstruktor kommt (siehe Abschnitt 7.4.2, »Konstruktoren«). Betrachten Sie kurz folgendes Beispiel:

```
...
01  unsigned int plus1(unsigned int val=0) {
02    return (val + 1);
03  }
...
04  double rawDarray[plus1(10)]; // Fehler !!!
```


In Zeile **04** wird sich der Compiler beschweren, dass in C++03 ein rohes Array mit variabler Länge nicht erlaubt ist. Wenn Sie sich allerdings den Code der Funktion genauer ansehen und den Funktionsaufruf `plus1(10)`, können Sie sofort erkennen, dass die Länge hier ja schon längst feststeht (hier 11).

Hier könnten Sie jetzt einwenden, warum dann nicht gleich 11 zwischen `[]` schreiben? Das ist richtig, aber manchmal gibt es Situationen, in denen man nicht einfach eine Zahl oder ein Literal verwenden kann, obwohl der Wert bereits feststeht. Das Beispiel mit `plus1()` ist zwar trivial, und hier schreibt man wohl besser ein Literal rein, aber es gibt häufig auch komplexere Berechnungen, die man dann eben einfach den Compiler durchführen und ihn einen entsprechenden Wert einsetzen lässt – und hier kommen konstante Ausdrücke mit dem neuen Schlüsselwort `constexpr` ins Spiel.

Hierzu nun die Funktion `plus1()` mit vorangestelltem `constexpr`:

```
...
01  constexpr unsigned int plus1(unsigned int val=0) {
02    return (val + 1);
03  }
...
04  double rawDarray[plus1(10)];
```

Durch das Voranstellen von `constexpr` in Zeile **01** prüft und berechnet der Compiler jetzt die Konstante für Funktionen und setzt den entsprechenden Wert in Zeile **04** zwischen `[]` ein.

Der Vorteil von `constexpr`-Ausdrücken ist, dass Sie damit sicherstellen können, dass die Daten vom Compiler schon berechnet werden können und somit in den ROM-Speicher (*Read-only-Memory*) gelegt und verwendet werden können. Damit dies auch möglich ist, werden einige Bedingungen vorausgesetzt. Zum Beispiel darf die Funktion nicht `void` als Rückgabetyp verwenden, es dürfen keine Variablen oder neue Typen deklariert werden, und es muss eine Rückgabewertanweisung vorhanden sein.

Neben einer Array-Größe können solche konstanten Ausdrücke auch bei Template-Parametern, `enum`-Elementen, `case`-Fallunterscheidungen oder `static_assert`-Ausdrücken nützlich sein.

constexpr bei einem Konstruktor

Auch wenn die Konstruktoren erst in Abschnitt 7.4.2 behandelt werden, soll hier schon mal der Hinweis folgen, falls Sie `constexpr` darauf anwenden wollen, dass der Konstruktor dann aus einer Elementinitialisierungsliste (siehe Abschnitt »Konstruktoren effektiver initialisieren mit Element-Initialisierer« in Abschnitt 7.4.2) und einem leeren Anweisungsblock bestehen muss.

Ein weiterer Vorteil ist auch, dass der Compiler einen Fehler meldet, wenn der Ausdruck nicht zur Kompilierzeit (fürs ROM) berechnet werden kann. Dies kann ein einfaches `const` nicht leisten (auch ein `static const` nicht).

constexpr, das stärkere const

Es wurde bereits im Buch erwähnt, dass `constexpr` in gewisser Hinsicht auch als stärkeres `const` für ganz gewöhnliche Variablen verwendet werden kann. Ein Vorteil von `constexpr` ist es schon, dass Sie `constexpr` im Gegensatz zu `const` gleich mit Daten initialisieren müssen. Der Wert für `constexpr` muss quasi feststehen (daher müssen Funktionen auch einen Wert zurückgeben). Der Zwang zur Initialisierung von `constexpr`-Variablen erspart einem die eine oder andere böse Überraschung, welche bei Versäumnissen mit `const` auftreten können.

Und da der Compiler die Werte zur Übersetzungszeit berechnen kann, kann er diesen Wert als Zahl dann überall dort einsetzen, wo dieser benötigt wird. Es wird also nicht zwangsläufig eine Variable mit einer Speicheradresse angelegt. Betrachten Sie hierzu folgendes Beispiel:

```
...
01  #include <cmath> // für atan()

02  #define PI_V1 atan(1)*4
03  static const double PI_V2 = atan(1)*4;
04  constexpr double PI_V3 = atan(1)*4;
...
05  std::cout << PI_V1 << std::endl;
06  std::cout << PI_V2 << std::endl;
07  std::cout << PI_V3 << std::endl;
...
```

Hier finden Sie in den Zeilen **02** bis **04** drei verschiedene Möglichkeiten, die Kreiszahl `PI` errechnen zu lassen. Die schlechteste (nicht sehr C++-like) ist per `define`-Makro in Zeile **02** (siehe Abschnitt 6.5.2, »#define und #undef«). Hierbei handelt es sich nur um eine textuelle Ersetzung, bei der der Präprozessor noch vor dem Compilerlauf alle im Code verwendeten `PI_V1` durch `atan(1)*4` ersetzt. Praktisch bedeutet dies, dass überall jetzt `atan(1)*4` im Speicher abgelegt werden muss und die Berechnung an Ort und Stelle ausführt werden muss, wo eben `PI_V1` geschrieben wurde.

Besser ist dann schon die zweite Möglichkeit in Zeile **03** mit `const`. Hier wird nur einmal ein fester Speicherplatz zugewiesen und auch die Berechnung nur einmal bei der Zuweisung in Zeile **03** ausgeführt.

Im Gegensatz zu `const` existiert mit `constexpr` in Zeile **04** hingegen nicht zwangsläufig eine Speicheradresse. Es wird praktisch der berechnete Wert überall direkt eingesetzt (wie bei einem Literal). Es wird nur eine Speicheradresse von `constexpr` verwendet, wenn Sie den Compiler zwingen, einen Speicher im Programm zu verwenden:

```
01  constexpr double PI_V3 = atan(1)*4;
02  const double* my_pi = &PI_V3;
```

Erst durch die explizite Verwendung des Adressoperators in Zeile **02** wird jetzt für `PI_V3` ein Speicher angelegt.

Auch in der Standardbibliothek von C++11 sind viele Dinge jetzt auch als `constexpr` implementiert. Das ist bspw. extrem hilfreich, wenn Sie ein Array mit der Größe eines bestimmten Datentyps mithilfe von `numeric_limits` anlegen wollen, wenn Sie bspw. folgendermaßen (vor dem C++11-Standard) ein Array anlegen wollten:

```
int char_feld[numeric_limits<char>::max()] = {0};
```

Da für die Array-Größe eine statische Größe nötig gewesen ist, ließ sich die hier gezeigte Verwendung nicht benutzen und wurde vom Compiler mit Fehlermeldungen quittiert. Im neuen C++11-Standard ist diese Verwendung allerdings jetzt möglich, weil hier `max()` und viele weitere Funktionen auch `constexpr` sind.

6.5 Präprozessor-Direktiven

Bevor der Compiler und gegebenenfalls danach der Linker etwas zum Übersetzen bekommen, wird vorher noch der Präprozessor ausgeführt (Abbildung 6.1). Der Präprozessor (manchmal auch Präcompiler) ist oftmals ein eigenes Programm, welches die Eingabedaten (den Quelltext) zur weiteren Verarbeitung für ein anderes Programm (hier den Compiler) aufbereitet (meistens eine textuelle Ersetzung durchführt). Den Präprozessorlauf bekommen Sie in der Regel nicht mit, weil dieser gewöhnlich automatisch vor dem Compilerlauf ausgeführt wird.

Präprozessorlauf

Abhängig vom Compiler können Sie selbstverständlich auch einen reinen Präprozessorlauf durchführen. Bei g++ oder clang können Sie diesen mit der Option `-E` durchführen. Aber bei Microsoft Visual C++ können Sie bspw. mit `/E` die Ausgabe des Präprozessors auf die Standardausgabe betrachten. Hierzu empfehle ich Ihnen, bei Bedarf die Dokumentation bzw. das Manual Ihres Compilers zu lesen.

Die Befehle für den Präprozessor werden Direktiven (oder Präprozessor-Direktiven) genannt, beginnen immer mit dem #-Zeichen (Hash-Zeichen) am Anfang einer Zeile und enden am Zeilenende. Im Gegensatz zu einer C++-Anweisung benötigt eine Prä-

prozessor-Direktive kein Semikolon am Ende. Im Gegenteil, ein Semikolon am Ende kann gar zu unerwarteten Ergebnisse führen. Längere Direktiven können Sie in der nächsten Zeile fortsetzen, wenn Sie ein Backslash-Zeichen an das Zeilenende setzen. Pro Zeile ist eine Direktive erlaubt.

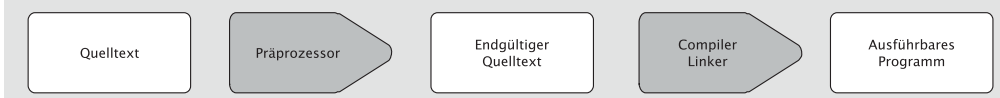


Abbildung 6.1 Der Compiler bzw. Linker bekommt den Quelltext erst nach dem Präprozessorlauf.

6.5.1 #include

Die Direktive `#include` haben Sie bereits in jedem Ihrer Programme verwendet. Mit `#include` wird der benötigte Quellcode der Headerdatei im Quelltext einkopiert. Hierbei gibt es folgende zwei Versionen:

```
01  #include <datei>
02  #include "datei"
```

Wenn Sie die Headerdatei wie in Zeile **01** zwischen die spitzen Klammern <> stellen, wird in einem voreingestellten Pfad nach der Datei gesucht. Der voreingestellte Pfad wird in der Regel von Ihrem Compiler oder der Entwicklungsumgebung festgelegt. In der Praxis können Sie weitere Pfade selbst hinzufügen. Wie dies geht, hängt vom System und dem verwendeten Compiler/der Entwicklungsumgebung ab.

Schreiben Sie hingegen die Datei zwischen Anführungszeichen "" wie in Zeile **02**, wird zunächst im lokalen Verzeichnis gesucht, wo Sie die Quelldatei gespeichert haben. Wird dort `"datei"` nicht gefunden, wird nochmals im voreingestellten Pfad gesucht, wie dies mit `<datei>` geschieht. In der Praxis wird die Version mit den Anführungszeichen für eigene Headerdateien verwendet.

Groß- und Kleinschreibung bei Headerdateien

Ein wichtiger Hinweis sollte hier noch angebracht werden, bevor Sie eigene Headerdateien schreiben. Beachten Sie, dass C++ nicht wissen kann, wie das Dateisystem mit Dateinamen umgeht. Daher kann nicht genau gesagt werden, ob beim Einkopieren zwischen Groß- und Kleinschreibung unterschieden wird. Persönlich sind meine Erfahrungen, dass Compiler unter Unix/Linux/Mac OS X ganz strikt auf Groß- und Kleinschreibung achten, während es bei Compilern unter MS-Windows in der Regel egal ist, ob hier `#include "datei"` oder `#include "Datei"` steht.

Relativer und absoluter Pfad

Zwischen den Anführungszeichen können Sie auch den relativen oder absoluten Pfad angeben:

```
#include "/home/wolf/cpp-examples/kap006/meinHeader.h"
#include "kap006/meinHeader.h"
#include "../kap006/meinHeader.h"
```

Dem Compiler das include-Verzeichnis mitteilen

Auch hier kann die Verwendung von relativen und absoluten Pfadangaben zu Problemen führen, wenn Sie den Code auf andere Systeme portieren wollen. Die klassische Fehlermeldung hierzu lautet: **meinheader.h: no such file or directory**. Da kann es dann bspw. nötig sein, ein Laufwerk anzugeben (bspw. unter MS-Windows), oder der Pfad muss komplett umgeändert werden. Auch hierfür bietet fast jeder Compiler eine Option an, den `include`-Pfad per Argument (bspw. `-I include-pfad` beim g++ bzw. clang oder `/I include-pfad` bei MS Visual C++) zur Übersetzung mit zu übergeben. Hierzu empfiehlt es sich, bei Bedarf wieder in der Dokumentation Ihres Compilers nachzulesen.

6.5.2 #define und #undef

Mit der `#define`-Direktive können Dinge wie symbolische Konstanten, Makros, bedingte Kompilierung und Fehlersuche realisiert werden. Mit `#define` können Sie quasi einen Text vom Präprozessor durch einen anderen Text ersetzen lassen. Allerdings, um es nochmals zu erwähnen, bleibt es trotzdem eine Direktive, die vom Präprozessor durchgeführt wird, welcher keine Ahnung von C++ hat. Daher sollten Sie auch echte C++-Dinge in der Praxis verwenden und symbolische Konstanten bzw. Makros mit echten C++-Techniken realisieren.

#define und C++

In der modernen C++-Praxis benötigen Sie die `#define`-Direktive eigentlich nur noch für Dinge wie die bedingte Kompilierung und vielleicht noch zur Fehlersuche (Debugging). Ansonsten gibt es für alles andere bessere Alternativen.

Symbolische Konstanten und Makros

Symbolische Konstanten können mit der `#define`-Direktive verwendet werden, wenn bestimmte konstante Werte in einem umfangreichen Programm mehrmals verwendet werden sollen, bspw.:

```
#define PI 3.1415f  // Symbolische Konstante
```

Der Präprozessor würde jetzt vor dem Compilerlauf alle im Quelltext vorhandenen Textfolgen `PI` durch die Textfolge `3.1415f` ersetzen. Hier ist bewusst noch die Rede von einer Textfolge, weil es für den Präprozessor nichts anderes ist.

Bei einem `#define`-Makro ist dies im Grunde nicht viel anders, nur dass Sie hier auch noch Parameter hinzufügen können, bspw.:

```
#define HOCH2(val) ( val * val )  // Makro
```

Steht jetzt im Quelltext die Textfolge `HOCH2(10)`, ersetzt der Präprozessor diesen Text durch die Textfolge `10 * 10`. Hierzu ein kurzes und einfaches Beispiel:

```
00  // kapitel06/define01.cpp
01  #include <iostream>
02  #include <cmath> // für atan()

03  #define PI atan(1)*4
04  #define DE_DE "Kreisfläche    :"
05  #define EN_EN "area of a disk :"
06  #define HOCH2(val) ( val * val )

07  int main() {
08    std::cout << DE_DE << HOCH2(10)*PI << std::endl;
09    double a = HOCH2(10+1)*PI;
10    std::cout << EN_EN << a << std::endl;
11  }
```

Der Präprozessor ersetzt jetzt die in den Zeilen **03** bis **06** verwendeten symbolischen Konstanten und Makros an den passenden Stellen im Quellcode in den Zeilen **07** bis **11**. Der Compiler bekommt dann, auf die Zeilen **07** und **11** beschränkt, folgenden Code nach der Textersetzung des Präprozessors zu sehen:

```
...
07  int main() {
08    std::cout << "Kreisfläche :" << 10*10*3.1415 << std::endl;
09    double a = 10+1*10+1*3.1415;
10    std::cout << "area of a disk :" << a << std::endl;
11  }
```

Leider enthält dieses Beispiel auch einen unangenehmen Seiteneffekt in der Berechnung in Zeile **09**, wenn Sie das Ergebnis aus Zeile **10** mit dem Ergebnis aus Zeile **08** vergleichen. Das Ergebnis aus Zeile **10** ist erheblich kleiner als das Ergebnis aus Zeile **08**, obwohl wir ja einen höheren Werte 10+1 verwendet haben. Der Seiteneffekt liegt

an der Punkt-vor-Strich-Regelung, welche hier mit (((10 + (1 * 10)) + (1* 3.1415)) rechnet. Den Fehler können Sie beheben, wenn Sie beim Makro `HOCH2` entsprechend die Klammern setzen:

```
06  #define HOCH2(val) ( (val) * (val) )
```

Allerdings sind es u.a. solche Seiteneffekte, wie hier mit den vergessenen Klammern, die immer wieder zu Problemen in Bezug auf Makros führen. Bedenken Sie, dass dieses Beispiel wieder recht trivial ist und der Fehler daher leicht zu finden war. Bei komplexeren Makros macht die Fehlersuche (Debugging) allerdings keinen Spaß mehr, wenn es sich um Makros handelt. Neben diesem Seiteneffekt wäre noch die mangelnde Typsicherheit zu bemängeln. Auch die Laufzeit wird negativ beeinflusst, wenn man Ausdrücke wie `#define PI atan(1)*4` verwendet, weil bei jedem Vorkommen von `PI` die Berechnung erneut durchgeführt werden muss. Eine bessere Alternative dazu haben Sie ja bereits im Abschnitt »constexpr, das stärkere const« in Abschnitt 6.4.3 kennengelernt. Auch der Gültigkeitsbereich von Makros ist ab der Position der `#define`-Direktive global sichtbar und lässt sich nicht in einen Namensraum einbetten.

Alternativen für #define-Makros und #define-Konstanten

`#define`-Direktiven sollen hier jetzt nicht ganz verteufelt werden. Bei Dingen wie der bedingten Übersetzung oder für die Fehlersuche sind Makros immer noch sehr wichtig. Aber für Makros und symbolische Konstanten sollten Sie echte C++-Techniken vorziehen. Statt symbolischer Konstanten bieten sich `const` (siehe Abschnitt 6.3.1, »Das Schlüsselwort const«) oder `enum` (siehe Abschnitt 4.7, »Aufzählungstypen (C++11-Version)«) an. Die besseren C++-Alternativen für Makros sind `inline`-Funktionen (siehe Abschnitt 5.8.1, »Inline-Funktionen«) und natürlich die Funktions-Templates (siehe Abschnitt 11.1, »Funktions-Templates«). Wenn Sie den neueren C++11-Standard verwenden können, dann würde ich Ihnen `constexpr` (siehe Abschnitt »constexpr« in Abschnitt 6.4.3) empfehlen.

Symbol und Makro aufheben

Soll eine symbolische Konstante oder ein Makro ab einer bestimmten Stelle im Programm nicht mehr gültig sein, können Sie diese mit der Direktiven `#undef` aufheben:

```
#undef SYMB_KONST
```

Ab der Position, wo Sie die dieses `#undef` setzen, führt der Präprozessor keine Ersetzung mehr von `SYMB_KONST` durch. Befindet sich hinter einer `#undef`-Direktiven trotzdem noch ein `SYMB_KONST`, hängt es vom Compiler ab, was als Nächstes geschieht – meistens wird hier wohl eine Fehlermeldung ausgegeben.

In der Praxis verwendet man `#undef`, um eine symbolische Konstante zunächst zu löschen und anschließend mit einem neuen Wert zu versehen. Auch lassen sich so symbolische Konstanten von Standard-Headerdateien aufheben und mit neuen Werten versehen. Hierzu ein kurzes Beispiel:

```
00  // kapitel06/define02.cpp
...
01  #define A "Kreisfläche : "
02  const double PI = atan(1)*4;

03  int main() {
04    std::cout << A << 10*10*PI << std::endl;
05  #undef A
06    double A = 20*20*PI;
07    std::cout << A << std::endl;
08  #define A "Andere Bedeutung :"
09    std::cout << A << std::endl;
10  }
```

Das Beispiel soll hier keine Schule machen. In Zeile **04** ersetzt der Präprozessor die symbolische Konstante `A` durch das Stringliteral `"Kreisfläche :"`. Diese symbolische Konstante wird in Zeile **05** aufgehoben. Der Präprozessor lässt daher den Bezeichner der Variablen `A` vom Typ `double` in den Zeilen **06** und **07** unberührt. Erst in Zeile **08** bekommt `A` als symbolische Konstante für den Präprozessor wieder eine Bedeutung, so dass in Zeile **09** das `A` durch das Stringliteral `"Andere Bedeutung :"` ersetzt wird.

6.5.3 Bedingte Kompilierung

Bei der bedingten Kompilierung können Sie beeinflussen, welchen Code der Compiler unter einer bestimmten Bedingung erhält und welcher ausgeblendet werden soll. Natürlich sind dies auch wieder nur Präprozessor-Direktiven, aber es kann recht nützlich sein, ganze Codeblöcke ein- bzw. auszublenden. Auch bei der Suche nach Fehlern (Debugging) kann eine solche bedingte Kompilierung sehr hilfreich sein.

Da es hierfür recht viele, aber trotzdem überschaubare Direktiven gibt, finden Sie hier alle in Tabelle 6.1 aufgelistet.

Direktive	Bedeutung
`#if AUSDRUCK`	Wenn `AUSDRUCK` ungleich 0 ist, bekommt der Compiler den dahinterstehenden Quellcode zum Übersetzen.

Tabelle 6.1 Präprozessor-Direktiven für die bedingte Kompilierung

Direktive	Bedeutung

`#ifdef SYMBOL`
`#if defined SYMBOL`
`#if defined (SYMBOL)`
	Ist das `SYMBOL` definiert (bspw. mit `#define`), bekommt der Compiler den darauffolgenden Quellcode.

`#ifndef SYMBOL`
`#if !defined SYMBOL`
`#if !defined (SYMBOL)`
	Ist das `SYMBOL` **nicht** definiert, bekommt der Compiler den darauffolgenden Quellcode.

`#elif SYMBOL` `#elif AUSDRUCK`	Ist das `SYMBOL` definiert (bspw. mit `#define`) oder der `AUSDRUCK` ungleich 0, erhält der Compiler den dahintergeschriebenen Programmteil. Einem `#elif` geht immer ein `#if` oder `#ifdef` voraus.
`#else`	Hier können Sie einen alternativen Programmteil dahinterschreiben, den der Compiler erhält, wenn alle vorangegangenen `#if`, `#ifdef` usw. nicht zutreffend waren.
`#endif`	Damit wird dem Präprozessor das Ende der bedingten Kompilierung mitgeteilt.

Tabelle 6.1 Präprozessor-Direktiven für die bedingte Kompilierung (Forts.)

Mehrfaches Inkludieren vermeiden

In fast jeder Headerdatei finden Sie ein Konstrukt der bedingten Kompilierung, welches überprüft, ob eine Headerdatei bereits inkludiert wurde oder nicht, damit diese nicht noch ein zweites Mal hinzugefügt wird. Folgendes Konstrukt sollten Sie also auch künftig in Ihrer Headerdatei hinzufügen:

```
00  // meinHeader.h
01  #ifndef MEIN_HEADER_H
02  #define MEIN_HEADER_H
03  // Hier kommt der Code für meinHeader.h
...
04  #endif
```

In Zeile **01** überprüfen Sie zunächst, ob das Symbol `MEIN_HEADER_H` noch **nicht** vorhanden ist. Trifft dies zu, wird das Symbol in Zeile **02** definiert. Anschließend folgt (ab Zeile **03**) der Code für die Headerdatei `meinHeader.h`, der vom Präprozessor für den Compiler in die Datei einkopiert werden soll. Das Ende der bedingten Kompilierung zeigen Sie dem Präprozessor mit der `#endif`-Direktiven in Zeile **04** an.

Falls die Bedingung in Zeile **01** nicht zutrifft und das Symbol `MEIN_HEADER_H` bereits definiert wurde, werden die Zeilen dahinter nicht mehr erneut ausgeführt, weil der Code der Headerdatei ja bereits einkopiert wurde, und somit wird die Arbeit des Präprozessors hinter `#endif` (Zeile **04**) weitergeführt.

Compiler- und systemspezifische bedingte Kompilierung

Das Thema geht über den C++-Standard hinaus und ist, wie Sie am Titel ablesen können, compiler- und systemspezifisch. Trotzdem kann es auf jedem System verwendet werden. Wenn ein Symbol nicht existiert, wird der darauffolgende Code ignoriert. Ein einfaches Beispiel soll Ihnen diese bedingte Kompilierung demonstrieren:

```
00  // kapitel06/define03.cpp
...
01  #ifdef __WIN32__
02    #define OS "MS Windows"
03  #elif __APPLE__
04    #define OS "Apple"
05  #elif __linux__
06    #define OS "Linux"
07  #elif __unix__
08    #define OS "Unix"
09  #else
10    #define OS "Unbekannt"
11  #endif

12  int main() {
13    std::cout << "Ihr System: " << OS << std::endl;
14  }
```

In den Zeilen **01**, **03**, **05** und **07** wird jeweils geprüft, ob in der Umgebung die symbolische Konstante `__WIN32__` (für MS-Windows), `__APPLE__` (für Macintosh Apple), `__linux__` (für Linux-Systeme) oder `__unix__` (für Unix-Systeme) definiert ist. Ist die entsprechende symbolische Konstante vorhanden, setzen wir in den Zeilen **02**, **04**, **06** oder **08** die symbolische Konstante `OS` mit einem entsprechenden Stringliteral. Ist keines der Symbole bekannt, wird die `#else`-Direktive aktiv, und wir setzen `OS` auf `"Unbekannt"`. Entsprechend der gesetzten symbolischen Konstante für `OS` ersetzt der Präprozessor in Zeile **13** die Konstante durch das Stringliteral und gibt im Idealfall das Betriebssystem aus, auf dem das Programm übersetzt wurde.

Weitere Symbole für Compiler und Systeme

Neben den hier verwendeten symbolischen Konstanten für Betriebssysteme gibt es noch eine Menge mehr – auch compilerspezifische. Allerdings geht dies, wie bereits eingangs erwähnt, über den normalen C++-Standard hinaus und ist daher auch wirklich compiler- und systemabhängig.

Hilfe bei der Fehlersuche

Ebenfalls recht hilfreich kann eine solche bedingte Kompilierung bei der Suche nach Fehlern sein. Betrachten Sie hierzu folgendes einfache Beispiel:

```
00  // kapitel06/define04.cpp
...
01  #define DEBUG 2

02  int main() {
03    double r=0.0, A=0.0, pi=atan(1)*4;
04    std::cout << "Bitte Radius eingeben: ";
05    std::cin >> r;
06  #if DEBUG > 0
07    std::cout << "DEBUG: " << r << std::endl;
08  #endif
09    A = r*=r*pi;
10  #if DEBUG > 1
11    std::cout << "DEBUG: " << r << std::endl;
12  #endif
13    std::cout << "Kreisfläche: " << A << std::endl;
14    // Weitere Berechnungen mit r ...
...
15  }
```

Am Anfang des Programms finden Sie in Zeile **01** eine symbolische Konstante `DEBUG` mit dem Wert `2` definiert. Den Wert können Sie jederzeit verändern. Dadurch dass `DEBUG` auf `2` gesetzt ist, werden die Zeilen **07** und **11** mit an den Compiler gegeben und entsprechende Debugging-Informationen auf dem Bildschirm ausgegeben. Im Beispiel wurde der Wert `r` (für Radius) überwacht, welcher irgendwo versehentlich nach der Eingabe geändert wird, womit die späteren Berechnungen zu falschen Ergebnissen führen. Setzen Sie den Wert der symbolischen Konstante auf `1`, wird nur noch die Debug-Information aus Zeile **07** ausgegeben, weil die Bedingung in den Zeilen **10** bis **12** nicht mehr zutreffend ist. Setzen Sie hingegen `DEBUG` gar auf `0`, erfolgt gar keine Debug-Ausgabe mehr auf dem Bildschirm.

6.5.4 Weitere Direktiven

Hier fehlen noch einige Präprozessor-Direktiven (Tabelle 6.2), welche recht nützlich sein können.

Direktive	Bedeutung
`#error "nachricht"`	Schreiben Sie diese Direktive in den Quellcode, wird die Übersetzung mit der Fehlermeldung `"nachricht"` abgebrochen. Dies kann hilfreich sein, wenn Sie bei einem Projekt an einem Quellcode arbeiten, der noch nicht funktioniert und Sie andere darauf hinweisen wollen.
`#line n "datei"`	Damit können Sie die Zeilennummer im Quelltext auf `n` und den Dateinamen auf `"datei"` setzen. Diese Direktive hat einen Einfluss auf die symbolischen Konstanten `__LINE__` und `__FILE__`.
`#pragma`	Dies sind compilerspezifische Direktiven. Wenn ein Compiler eine bestimmte `#pragma`-Direktive nicht kennt, wird diese ignoriert. Damit lassen sich Direktiven verwenden, ohne mit anderen Compilern in Konflikt zu geraten.

Tabelle 6.2 Weitere Präprozessor-Direktiven

Zusätzlich gibt es in C++ noch einige vordefinierte symbolische Konstanten, welche ich Ihnen hier auch nicht vorenthalten will (Tabelle 6.3).

Symbol	Bedeutung
`__LINE__`	Darin ist die aktuelle Zeilennummer des Quellcodes enthalten. Kann nützlich für Debugging-Zwecke sein.
`__FILE__`	Darin ist die Zeichenkette der aktuellen Datei enthalten.
`__DATE__`	eine Zeichenkette mit dem aktuellen Datum
`__TIME__`	eine Zeichenkette mit der aktuellen Uhrzeit

Tabelle 6.3 Vordefinierte symbolische Konstanten in C++

6.6 Modulare Programmierung – Code organisieren

Wenn der Quellcode umfangreicher wird und mehrere Entwickler an einem Projekt arbeiten, wird in der Regel nicht nur mit einer Quelldatei gearbeitet. Hierbei werden in der gängigen Praxis mehrere Quell- und Headerdateien – auch Module genannt – verwendet.

Nur eine kurze Einführung

Das Thema hier ist nur eine kurze Einführung und stellt auch keine ultimative Vorgehensweise dar – die gibt es nämlich nicht. Trotzdem gibt es einige Richtlinien, die Ihnen helfen, bei umfangreicheren Projekten den Überblick zu bewahren, wenn Sie sich daran orientieren. Das Thema hängt zudem auch von der verwendeten Entwicklungsumgebung, der Anzahl der mitwirkenden Programmierer und zum Teil auch vom verwendeten System ab. Das bedeutet letztendlich, dass Sie bei umfangreicheren Projekten nicht darum herumkommen, sich intensiver mit den Details zu befassen.

6.6.1 Module

Ein Modul ist ganz einfach eine Datei mit einer Sammlung von Funktionen oder Klassen. Idealerweise werden diese Dateien in sinnvolle Gruppen aufgeteilt. Funktionen und Klassen, welche bspw. die Netzwerkprogrammierung oder die Datenbankprogrammierung betreffen, steckt man gewöhnlich jeweils in ein extra Modul. Dass die einzelnen Module dann auch noch in eine Quelldatei (**.cpp*) und Headerdatei (**.h*) aufgeteilt werden, sei jetzt nur am Rande erwähnt.

Der Vorteil solcher logischen Strukturierungen ist, dass Sie solche Module jederzeit auch wieder in anderen Projekten verwenden können. Neben der Wiederverwendbarkeit von Modulen stellt ein Aufteilen von Quellcode die beste Möglichkeit dar, in einer Gruppe von mehreren Personen zu programmieren. Jeder Programmierer erhält eine Aufgabe für ein spezielles Modul. Natürlich ist es auch möglich, dass mehrere Entwickler an derselben Datei arbeiten. Allerdings wird hierfür in der Praxis dann noch eine Versionsverwaltungssoftware verwendet, damit es hier nicht zu einem Chaos kommt. Auch muss ein Projekt dank solcher Module nicht immer wieder komplett übersetzt werden.

Sie ahnen es, dass eine solche Projektverwaltung bei großen Projekten eine umfangreichere und aufwendige Sache sein kann. Es gibt hier zwar keine festen Regeln, wie Sie Ihre Module aufteilen können, aber Sie sollten immer versuchen, ein Modul so zu schreiben, dass es selbstständig arbeiten kann. Genauer: Die Notwendigkeit des Austauschs von Informationen zwischen den Modulen sollte möglichst gering sein.

6.6.2 Sinnvolle Quellcodeaufteilung

In der Praxis wird ein Modul in zwei Teile aufgeteilt, einen öffentlichen und einen versteckten Bereich (Abbildung 6.2):

- Öffentliche Schnittstelle: In der Regel ist es die Headerdatei (meistens mit der Endung **.h*), die die öffentliche Schnittstelle für Funktionen und Klassen anbietet

– genauer sämtliche öffentlichen Deklarationen. Mithilfe dieser Schnittstelle weiß der Benutzer, wie er die Funktionen und Klassen im Modul verwenden kann.

- Private Dateien: Darin finden Sie gewöhnlich die eigentliche Implementation (den eigentlichen arbeitenden Quellcode) der Schnittstellen und Klassen – also die Definitionen. Die privaten Dateien liegen dabei nicht immer als lesbare Quelldatei (mit der Endung **.cpp*) vor, sondern sind häufig auch nur als vorübersetzte Objektdatei (Endung: **.o* oder **.obj*) oder gar als Bibliotheksdatei vorhanden.

Objekt- und Bibliotheksdatei sind nicht portabel

Natürlich sollte Ihnen klar sein, dass Sie eine Objektdatei oder eine Bibliotheksdatei nicht mehr auf anderen Plattformen verwenden können. Eine für Windows erstellte Objekt- oder Bibliotheksdatei lässt sich nicht unter Mac OS X oder Linux verwenden.

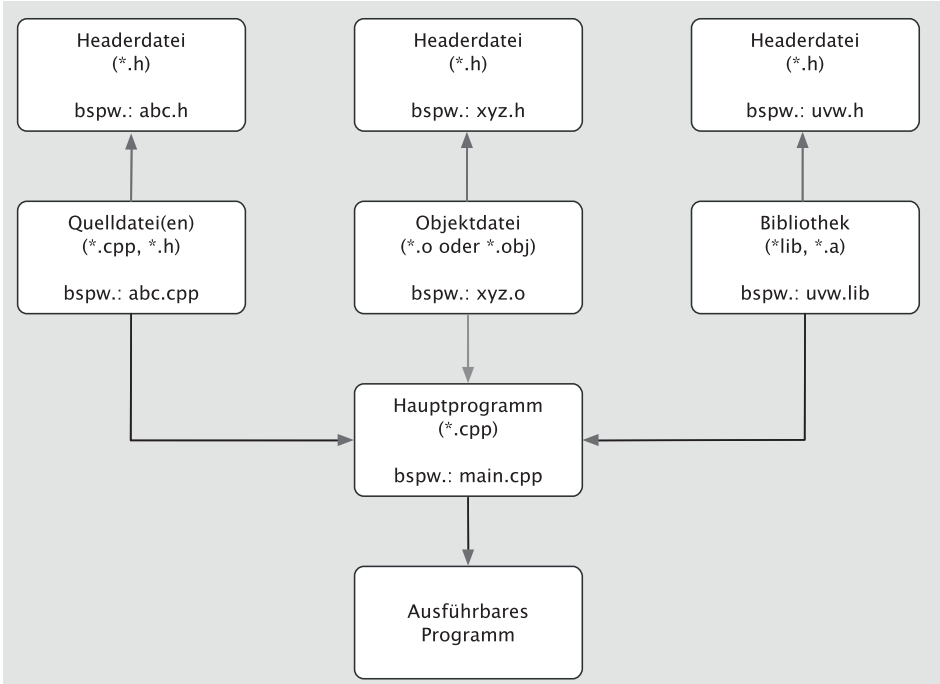


Abbildung 6.2 Eine theoretische Aufteilung von Modulen

Damit es mit der Übersetzung der einzelnen Module auch klappt, müssen Sie dem Compiler bzw. Linker mitteilen, wo er diese finden kann (Abbildung 6.3). Neben dem Inkludieren der Headerdatei(en) mittels `#include` müssen Sie bei Ihrem Projekt auch die Quelldatei (**.cpp)* mit angeben. Der Compiler bzw. Linker kann nicht wissen, was

alles zum Projekt gehört. Bei Entwicklungsumgebungen müssen Sie dem Projekt hierbei gewöhnlich nur eine neue Datei (in der Projektverwaltung) hinzufügen. In der Kommandozeile hingegen geben Sie einfach diese Quelldatei noch zusätzlich bei der Übersetzung mit an. Analog gilt dies auch, wenn es sich um eine Objekt- oder Bibliotheksdatei handelt.

Quelldatei inkludieren

Manchmal sieht man leider auch verzweifelte Inkludierungen von Quelldateien wie:

```
#include "abc.cpp"
```

Es ist zwar nicht falsch, so etwas zu machen, aber doch eher unüblich. Häufig kommt noch hinzu, dass man gar keine Quelldatei hat, sondern nur auf die Headerdatei mit Objekt- oder Bibliotheksdatei beschränkt ist.

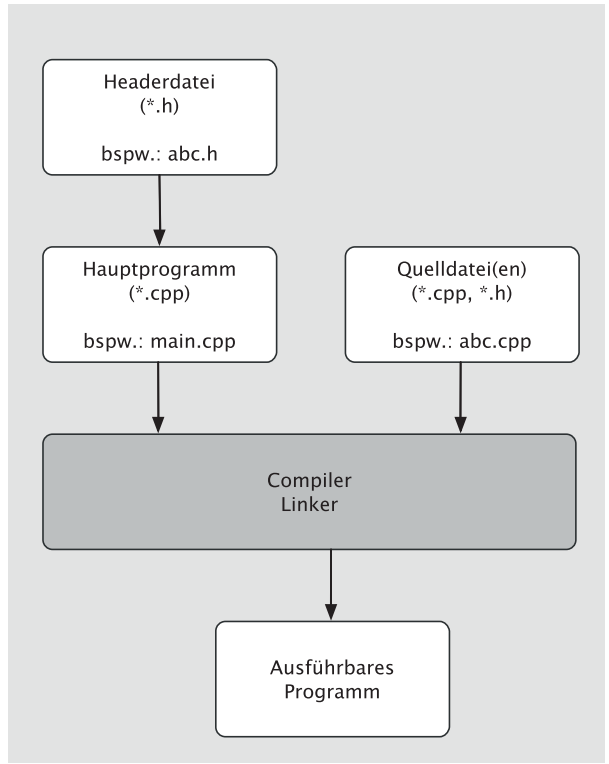


Abbildung 6.3 Damit es mit der Übersetzung der einzelnen Module klappt, müssen immer alle zu einem Projekt gehörenden Dateien angegeben bzw. inkludiert werden.

Wichtig bei einem Projekt mit Modulen ist natürlich vor allem auch, dass dem Compiler bzw. Linker die einzelnen Module auch bekannt sind. Während eine Header-

datei noch in den Quellcode geschrieben wird, muss die Quelldatei (bzw. auch Objekt- und/oder Bibliotheksdatei) dem Projekt bei der Übersetzung hinzugefügt werden. Wie dies genau funktioniert, hängt von der Entwicklungsumgebung oder dem verwendeten Compiler in der Kommandozeile ab.

6.6.3 Die öffentliche Schnittstelle (Headerdatei)

Die öffentliche Schnittstelle (Endung **.h* oder manchmal auch **.hpp*) ist für die Vorbereitung und auch für den Anwender des Moduls enorm wichtig. Gerade wenn die private Datei des Moduls nur als Objekt- oder Bibliotheksdatei mitgeliefert wird, erkennt der Benutzer anhand der Schnittstelle(n), wie er die Funktionen und Klassen des Moduls verwenden kann. Bevorzugt schreibt man in einer Headerdatei die Deklarationen und vereinzelt auch einige Definitionen. In Tabelle 6.4 finden Sie eine kurze Übersicht, was in einer solchen Headerdatei für gewöhnlich enthalten sein kann.

Element	Beispiel
Aufzählungen	`enum class Color { RED=1, GREEN, BLUE };`

Bedingte Kompilierung	`#ifndef __SYMOL__`
	`#define __SYMBOL__`
	`...`
	`#endif`

Bezeichner deklarieren	`class EineKlasse;`
Funktionen deklarieren	`int simple_function(const int v);`
`include`-Direktiven	`#include <headerdatei01>` `#include "headerdatei02"`

`inline`-Funktionen	`inline int foo(const int v) {`
	`  return v*v;`
	`}`

Kommentare	`// Kommentar in einer Zeile`
Konstanten definieren	`static const double PI = 3.1415;`
Symbole, Makros	`#define __SYMBOL__ xyz`

Namensbereiche	`namespace myNamespace {`
	`  int a, b, c;`
	`  int simple_func(const int val);`
	`};`

Tabelle 6.4 Dinge, die man gewöhnlich in einer Headerdatei findet

Element	Beispiel

Templates definieren	`template<class T1> class Cxyz {`
	`  ...`
	`};`

Templates deklarieren	`template<class T1> class Cxyz;`

Typdefinition	`struct bar{`
	`  std::string str01;`
	`  bar* next;`
	`};`

Tabelle 6.4 Dinge, die man gewöhnlich in einer Headerdatei findet (Forts.)

Hierzu eine sehr einfach gehaltene kurze Headerdatei *abc.h* zur Demonstration:

```
00  // kapitel06/abc/abc.h
01  #include <iostream>
02  #ifndef ABC_H
03  #define ABC_H

04  namespace myMath {
05    static const double PI=3.1415;
06    double area_of_a_disk( const double radius );
07  }
08  #endif
```

In der Headerdatei finden Sie eine Funktionsdeklaration (siehe Abschnitt 5.3, »Funktionen deklarieren (Vorausdeklaration)«) und die Definition einer Konstante (siehe Abschnitt 6.3.1, »Das Schlüsselwort const«) in einen Namensraum (siehe Abschnitt 6.1, »Namensräume«) verpackt. Natürlich wird hier auch der Präprozessor-Teil mit `#ifndef` für die bedingte Kompilierung verwendet (siehe Abschnitt 6.5.3, »Bedingte Kompilierung«).

6.6.4 Die privaten Dateien

Die eigentlichen Implementierungen der Schnittstellen (also Definitionen) von Funktionen, Klassen usw. werden gewöhnlich in der (privaten) Quelldatei mit der Endung **.cpp* geschrieben. Als Namen verwendet man hierfür meistens denselben Namen wie für die dazugehörende Headerdatei. In unserem Beispiel wäre dies somit *abc.cpp*:

```
00  // kapitel06/abc/abc.cpp
01  #include "abc.h"
```

```
02  namespace myMath {
03    double area_of_a_disk( const double radius ) {
04      return radius * radius * PI;
05    }
06  }
```

Hier finden Sie jetzt im erneut geöffneten Namensbereich die Definition der Funktion `area_of_a_disk()`. Auch die Konstante `PI` aus der Headerdatei *abc.h* wird hier verwendet. Die Headerdatei *abc.h* muss logischerweise auch eingebunden werden.

Randnotiz

Gewöhnlich werden in die private Datei auch anonyme Namensbereiche und exportierte Template-Definitionen reingeschrieben.

Objektdatei erstellen

An dieser Stelle sei angemerkt, dass Sie aus *abc.cpp* und *abc.h* bereits eine Objekt- oder gar Bibliotheksdatei machen könnten. Hierzu ist keine `main`-Funktion nötig. Um eine solche Objektdatei zu erstellen, wird *abc.cpp* und *abc.h* einfach nur kompiliert – ohne Linkerlauf.

Eine so erstellte Objektdatei *abc.o* bzw. *abc.obj* und die Headerdatei *abc.h* können Sie jetzt weitergeben. Den Quellcode von *abc.cpp* müssen Sie so nicht freigeben, und dieser bleibt verborgen. Beim Übersetzen eines ausführbaren Programms müssen Sie hierbei die Headerdatei *abc.h* mit einbinden und dem Linker mitteilen, wo dieser die Objektdatei finden kann. Fast anlog verläuft dies beim Erstellen von Bibliotheksdateien. Natürlich sollten Sie wissen, dass solche Objekt- bzw. Bibliotheksdateien nicht mehr portabel und somit systemabhängig sind. Eine auf Linux erstelle Objektdatei kann nicht mehr unter Windows verwendet werden.

6.6.5 Die Client-Datei

Die *Client-Datei* gibt es eigentlich nicht als Begriff, aber ich verwende diesen Begriff relativ gerne, weil es sich hierbei um die Datei handelt, welche auf das (oder die) Modul(e) zugreift – und meistens das Hauptprogramm `main()` enthält. In der Client-Datei müssen Sie dem Projekt nur noch die Headerdatei(en) und die Quelldatei(en) hinzufügen. Hier also die Client-Datei *main.cpp*:

```
00  // kapitel06/abc/main.cpp
01  #include <iostream>
02  #include "abc.h"
```



```
03  int main() {
04    double r=0.0;
05    std::cout << "Radius : ";
06    std::cin >> r;
07    double A = myMath::area_of_a_disk(r);
08    std::cout << "Kreisfläche : " << A << std::endl;
09  }
```

Wie bereits eingangs erwähnt, ist es wichtig, dass Sie dem Projekt die private Quelldatei hinzufügen oder sie in der Kommandozeile mit angeben. Beim Beispiel mit *abc.cpp* und *abc.h* sieht die Übersetzung bei mir in der Kommandozeile mit dem g++ (mit clang++ ist dies analog) wie folgt aus (alle Dateien liegen hier im selben Verzeichnis, in dem ich mich gerade auch befinde):

```
J.Wolf $ g++ -o main main.cpp abc.cpp -Wall -pedantic -std=c++11
J.Wolf $ ./main
Radius : 123.123
Kreisfläche : 47622.9
J.Wolf $
```

Bei einer Entwicklungsumgebung ist dies im Prinzip recht ähnlich, nur dass Sie hierbei eben die Datei(en) dem Projekt hinzufügen müssen (Abbildung 6.4). Es reicht also nicht nur aus, einfach eine neue Datei anzulegen.

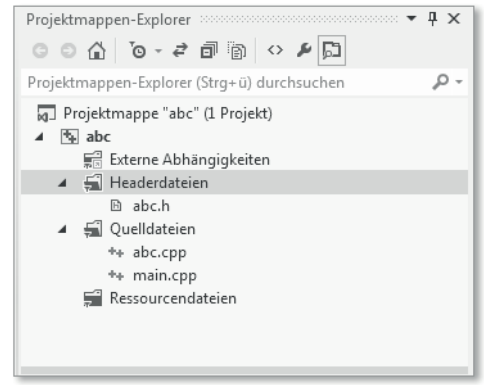


Abbildung 6.4 Dank der Projektverwaltung ist es in einer Entwicklungsumgebung wesentlich einfacher den Überblick zu behalten, wenn einzelne Dateien ordentlich zum Projekt hinzugefügt wurden.

Nur wenn alle Dateien dem Projekt hinzugefügt wurden, klappt es auch bei der Entwicklungsumgebung mit mehreren Modulen. In Abbildung 6.4 sehen Sie den Projektmappen-Explorer von Microsofts Visual C++.

Kapitel 11

Template-Programmierung

Mit den Templates (engl. für Schablonen) lernen Sie eine weitere fundamentale Technik zur Typparametrierung kennen. Diese Technik ermöglicht Ihnen eine generische Programmierung und typsichere Container. So verwendet bspw. die Standardbibliothek selbst Templates, um typsichere Container wie std::vector, std::list usw. zur Verfügung zu stellen. Auch viele generische Algorithmen arbeiten damit. Ein guter Grund also, einen Blick darauf zu werfen, um selbst Templates schreiben zu können.

Eigene Templates zu erstellen, kann sehr nützlich sein, wenn Sie häufig ähnliche Funktionen oder ähnliche Klassen nur mit unterschiedlichen Typen implementieren müssen. Sie haben das Prinzip bereits am Standardcontainer `std::vector` gesehen. Das Prinzip von `std::vector` bleibt ja im Grunde immer dasselbe, es unterscheidet sich nur der Typ. Daher wurde `std::vector` als **Klassen-Template** implementiert, womit Sie diesen Container jetzt mit jedem beliebigen Typ verwenden können, indem Sie den Typ zwischen die spitzen Klammern schreiben (`std::vector<int>`, `std::vector<double>`, `std::vector<KlassenXY>` usw.). Gleiches gilt auch für häufig verwendete Algorithmen wie das Suchen oder Sortieren. Auch hier macht es einem die Standardbibliothek leicht und bietet **Template-Funktionen** an, bei denen der Compiler den Typ anhand der Argumente selbst herausfindet, anstatt diese Algorithmen mehrfach für jeden Typ speziell zu schreiben.

Wie Sie hier jetzt herauslesen konnten, können Sie Klassen-Templates und Funktions-Templates erstellen. Der Vorteil ist zunächst, dass Sie eine Klasse oder Funktion nur einmal schreiben müssen. Auch die Instanzen solcher Templates werden nur angelegt, wenn diese tatsächlich verwendet werden. Wenn Sie ohne Templates eine `funktionXY()` mit Parameter für `int`, `double` und `long long` anlegen, wird für alle drei Versionen ein Maschinencode erzeugt. Erstellen Sie diese Funktion als Funktions-Template, wird nur ein Code von den Versionen erzeugt, die Sie tatsächlich im Programm verwenden. Das kann die ausführbare Datei schlanker halten. Auch die Überarbeitung von Code wird vereinfacht, denn anstatt alle Versionen einer bestimmten Funktionalität anzupassen, müssen Sie nur noch den Code des einen Templates

anpassen. Das spart Zeit und verhindert Fehler, welche bei der mehrfachen Codierung entstehen könnten.

Keine Reduzierung von Maschinencode

Wohlgemerkt, Templates reduzieren niemals den Umfang des Maschinencodes, sondern ersparen dem Programmierer lediglich, ähnliche Funktionen und Klassen für verschiedene Datentypen mehrmals schreiben zu müssen.

11.1 Funktions-Templates

Ein Funktions-Template sollte in der Lage sein, Argumente von verschiedenen Typen anzunehmen und auch, wenn nötig, verschiedene Rückgabewerte zurückzugeben. Hierzu soll eine einfache Funktion demonstriert werden, welche den kleinsten von zwei Werten zurückgibt:

```
01  int minVal(int v1, int v2) {
02    if( v1 < v2 ) {
03      return v1;
04    }
05    return v2;
06  }
```

Diese Funktion liefert den kleinsten von zwei übergebenen Werten an den Aufrufer zurück. Dank der impliziten Konvertierung des Compilers lässt sich diese Funktion auch mit `short` und `char` als Parametern verwenden. Wenn Sie jetzt aber eine Version für eine Gleitkommazahl verwenden wollen, müssen Sie eine weitere Version mit `float`, `double` und/oder `long double`, also Parameter und Rückgabewert implementieren:

```
01  double minVal(double v1, double v2) {
02    if( v1 < v2 ) {
03      return v1;
04    }
05    return v2;
06  }
```

Jetzt haben Sie zwei Versionen von `minVal()` und können diese Funktion mit Integern und Gleitpunktzahlen aufrufen. Wenn Sie diese Funktion auch mit `std::string` oder besser noch einer eigenen Klasse aufrufen wollen, müssen Sie weitere Versionen dafür implementieren.

std:min() und std::max()

Hier soll noch hinzugefügt werden, dass die C++-Standardbibliothek in der Headerdatei `<algorithm>` längst Funktions-Templates mit `std::min()` und `std::max()` anbietet, welche den kleinsten bzw. größten Wert von zwei übergebenen Werten zurückgibt. Mit Einführung von C++11 funktionieren `std::min()` und `std::max()` auch für mehrere Elemente, wenn man `{...}` verwendet.

Der Code hierfür ist eigentlich immer derselbe, nur die Typen sind anders. In solch einem Fall bietet sich jetzt ein Funktions-Template an, bei dem wir nur eine Version erstellen und den Compiler veranlassen, uns die benötigten weiteren Versionen zu generieren.

11.1.1 Funktions-Templates implementieren

Wenn Sie ein Funktions-Template definieren wollen, so müssen Sie dies mit folgendem Präfix einleiten:

```
template <typename T>
```

Der Parameter `T` wird hierbei für den formalen Datentyp verwendet, der bei der Definition der Funktion zum Einsatz kommt. `T` steht hierbei für einen beliebigen Datentyp, wie bspw. `int`, `double`, `long long` oder auch eine andere Klasse. Sie müssen hier übrigens nicht zwangsläufig `T` für den Namen des formalen Datentyps verwenden und können auch jeden beliebigen (gültigen) Bezeichner nehmen. Allerdings findet man in der gängigen Praxis meistens den Buchstaben `T` vor.

template <class T> oder template <typename T>

In einigen Programmen kann es sein, dass Sie hier auch `template <class T>` anstatt `template <typename T>` vorfinden, was ebenfalls richtig ist. Beide Varianten sind absolut gleichbedeutend. Nur die Version mit `typename` wurde etwas später in C++ eingeführt und ist somit etwas jüngeren Datums. Ich persönlich bevorzuge `typename`, weil es sich besser herauslesen lässt, dass `T` ein beliebiger Typ (wie `int`, `double` usw.) sein kann, der nicht nur als Klasse definiert wird. Bei der Verwendung des Schlüsselwortes `class` entsteht eher der Eindruck, dass es sich bei `T` um eine Klasse handeln muss. Häufig wird hierbei auch `class` verwendet, wenn erwartet wird, dass `T` eine Klasse ist, und `typename`, wenn `T` ein beliebiger eingebauter Typ (`int`, `double` etc.) sein sollte.

template template Parameter

So ganz ohne schlechtes Gewissen kann ich die Hinweisbox darüber nicht stehen lassen. Es gibt nämlich tatsächlich eine Gelegenheit, bei der es nicht egal ist, ob Sie `typename` oder `class` verwenden, und zwar bei `template-template`-Parameter – also wenn Sie innerhalb eines Template-Parameters ebenfalls einen Template-Parameter verwenden:

```
template <template <typename> class    T> class CLO1 { }; //Ok!!!
template <template <typename> typename T> class CLO2 { }; //Fehler!!!
```

Hier **muss** im fett hervorgehobenen Teil `class` statt `typename` verwendet werden. Auch wenn Sie diese (Klassen-)Templates jetzt noch nicht verstehen und wir auch gar nicht mehr näher darauf eingehen werden, behalten Sie dies vielleicht trotzdem im Hinterkopf, falls Sie irgendwann einmal in Ihrem Programmiererleben auf einen `template-template`-Parameter stoßen oder ihn schreiben müssen – was mir persönlich noch nie untergekommen ist. Aber es sollte halt immerhin erwähnt werden.

Somit sieht die Definition unseres Funktions-Templates für `minVal()` wie folgt aus:

```
00  // kapitel11/function_template/minval.cpp
...
01  template <typename T>
02  T minVal(T v1, T v2) {
03    if( v1 < v2 ) {
04      return v1;
05    }
06    return v2;
07  }
...
```

Vergleichen Sie das Funktions-Template mit unseren ursprünglichen Definitionen, so wurde hier lediglich statt eines Datentyps beim Rückgabewert und Parameter der Buchstabe `T` (für den Typ) verwendet. Den formalen Datentyp `T` können Sie auch mit lokalen Daten innerhalb der Funktion verwenden. Auch ersetzt der Compiler bei der Generierung den formalen Datentyp durch den tatsächlichen Datentyp. Ein einfaches Beispiel hierzu:

```
...
01  template <typename T>
02  void mySwap(T& v1, T& v2) {
03    T tmp = v1;
04    v1 = v2;
05    v2 = tmp;
06  }
...
```


Dieses Funktions-Template tauscht Daten vom Typ `T` miteinander. In Zeile **03** wurde hier eine lokale Variable vom Typ `T` verwendet, welche als temporärer Speicher für den Austausch zweier Daten vom Typ `T` dient. Die zu tauschenden Werte müssen ebenfalls vom selben Datentyp sein, was hier ja mit den Parametern vom selben Typ (auch `T`) der Fall ist.

Ebenso könnten Sie innerhalb von Funktionen, welche aus einem Template erstellt wurden, ein weiteres Funktions-Template aufrufen. Das folgende Beispiel erweitert die Funktion `mySwap()` um die Funktion, dass die Werte nur dann getauscht werden, wenn der Wert im linken Parameter größer ist als der Wert im rechten Parameter:

```
00  // kapitel11/function_template/greaterThan.cpp
...
01  template <typename T>
02  bool greaterThan(T v1, T v2) {
03    return v1 > v2;
04  }

05  template <typename T>
06  void mySwap(T& v1, T& v2) {
07    if( greaterThan(v1, v2) ) {
08      T tmp = v1;
09      v1 = v2;
10      v2 = tmp;
11    }
12  }
...
```

Jetzt werden mit `mySwap()` nur die Werte vertauscht, wenn `greaterThan()` in Zeile **07** gleich `true` zurückgibt.

Nun haben Sie zwar das Funktions-Template definiert, aber der Compiler erzeugt aufgrund der Definition eines Funktions-Templates im Gegensatz zu normalen Funktionen noch keinen Maschinencode, weil er noch gar nicht wissen kann, für welchen Datentyp er eine Version generieren soll. Da der Datentyp des formalen Parameters (hier `T`) noch nicht bekannt ist, wird hier auch von einer **unvollständigen Definition** gesprochen.

Funktions-Templates über mehrere Module

Da es sich bei einem Funktions-Template allein um eine unvollständige Definition handelt, stellt sich gleich die Frage, wohin damit bei umfangreicheren Projekten mit mehreren Modulen? Eine häufige Empfehlung hierzu lautet, dass man die Definition des Funktions-Templates in eine Headerdatei stellt, damit das Template in allen Modulen zur Verfügung steht, wo die Headerdatei eingebunden wird.

Der Grund dafür ist, dass die endgültige Funktion (Maschinencode) ja erst beim Aufruf einer durch das Funktions-Template vorgegebenen Funktion erstellt wird, und dafür benötigt der Compiler den Code der Funktion.

11.1.2 Aufruf des Funktions-Templates

Nachdem Sie ein Funktions-Template definiert haben, können Sie dieses Template wie eine ganz normale Funktion aufrufen. Der Compiler kümmert sich darum, dass die richtige Version verwendet oder eben entsprechender Maschinencode generiert wird. Zum Beispiel:

```
00  // kapitel11/function_template/minval.cpp
...
01  std::cout << minVal(100, 101) << std::endl;
02  std::cout << minVal(1.832, 1.831) << std::endl;
03  std::cout << minVal(std::string("aab"), std::string("aaa"));

04  int val1=100, val2=200;
05  mySwap(val1, val2);
06  std::cout << val1 << "/" << val2 << std::endl;

07  double dval1=3.142, dval2=3.141;
08  mySwap(dval1, dval2);
09  std::cout << dval1 << "/" << dval2 << std::endl;
...
```

Im Beispiel von `minVal()` sehen Sie hier, wie diese Funktion einmal mit Integer-Werten (Zeile **01**), Gleitkommazahlen (Zeile **02**) und mit `std::string` (Zeile **03**) aufgerufen wurde. Gleiches wurde auch mit `mySwap()` mit `int` und `double` gemacht (Zeilen **05** und **08**). Sicherlich fragen Sie sich, wie das im Compiler vor sich geht?

Der Compiler sucht bei der Übersetzung zuerst nach einer Funktion, die zu dem verwendeten Datentyp des Funktionsaufrufes passt, und verwendet diese. Findet der Compiler aber keine passende Funktion, sucht er nach einem Funktions-Template. Findet dieser ein Funktions-Template, ersetzt er den formalen Datentyp (hier `T`) durch den echten Datentyp und generiert eine Funktion daraus. In unserem Beispiel generiert der Compiler bspw. drei Versionen von `minVal()`, einmal für ein Integer (`int`), einmal für eine Gleitkommazahl (`double`) und einmal eine Version mit `std::string`. Die Funktion wird also erst bei einem Funktionsaufruf mit dem entsprechenden Datentyp mithilfe des Funktions-Templates **instanziiert**.

Von der Funktion `mySwap()` hingegen werden nur zwei Versionen generiert, weil diese Funktion nur mit `int` und `double` aufgerufen wurde.

Funktions-Template mit eigener Klasse als Parameter

Funktions-Templates arbeiten hervorragend mit den eingebauten Typen von C++ (bspw. `char`, `int`, `double`, `long long` usw.) und in der Regel auch mit fast allen Standardcontainer-Klassen von C++, wie Sie im Beispiel mit `std::string` gesehen haben. Wenn Sie allerdings eigene Klassen verwenden, gelten auch hier alle Regeln, wie bei der Verwendung eigener Klassen als Funktionsargument – letztendlich generiert der Compiler ja aus dem Funktions-Template eine gewöhnliche Funktion mit entsprechenden Parametern. Wenn Sie folgendes Beispiel ausprobieren, wird sich der Compiler bei Ihnen beschweren:

```
...
01  class C {
02    int val;
03  public:
04    C(int v=0) : val(v) {}
05    void print() const { std::cout << val << std::endl; }
06  };
...
07  C c01(101);
08  C c02(102);
09  C c03 = minVal(c01, c02);  // Fehler !!!
10  c03.print();
...
```

Die Compiler beschwert sich, dass er hier den Operator `operator<` vermisst, welcher in unserem Funktions-Template `minval()` verwendet wird. Standardcontainer-Klassen bieten ja schon den Komfort, dass sie `operator<` für alle eingebauten C++-Typen bereits von Haus aus implementiert haben. Da im Funktions-Template ein Vergleich mit dem `operator<` verwendet wird, müssen Sie daher in der Klasse `C` diesen Operator auch überladen und implementieren. Die Operatorüberladung wird in Kapitel 8, »Operatoren überladen«, behandelt.

Spezialisierung

Nicht immer lässt sich das Problem wie hier mit einer einfachen Operatorüberladung lösen. Ist eine Klasse zu speziell und erfordert sie eine besondere Behandlung, die sich mit dem vorhandenen Funktions-Template nicht vereinbaren lässt, weil kein sinnvolles Ergebnis zurückgeliefert wird, können Sie auch eine Spezialisierung für dieses Funktions-Template definieren. Hierbei wird das Funktions-Template durch eine separat definierte Funktion (die Spezialisierung) überladen. Mehr zur Spezialisierung von Funktions-Templates können Sie in Abschnitt 11.1.3, »Funktions-Template spezialisieren«, nachlesen.

Folgender Codeausschnitt implementiert jetzt `operator<` als `friend`-Funktion in der Klasse `C` und sorgt dafür, dass auch vom Funktions-Template `minVal()` eine Version für unsere Klasse `C` instanziiert werden kann:

```
00  // kapitel11/function_template/minval2.cpp
...
01  class C {
02    int val;
03  public:
04    C(int v=0) : val(v) {}
05    friend bool operator<(C &c1, C &c2);
06    void print() const { std::cout << val << std::endl; }
07  };

08  bool operator<(C &c1, C &c2) {
09    return (c1.val < c2.val);
10  }
...
```

Typübereinstimmung

Wenn Sie Funktions-Templates verwenden, sollten Sie noch wissen, dass beim Auflösen des formalen Datentyps keine automatische Typkonvertierung durchgeführt wird. Wenn Sie bspw. ein Funktions-Template mit zwei formalen Datentypen mit `T` definiert haben, dann müssen beim Aufruf des Templates beide Typen miteinander übereinstimmen. Selbst im Prinzip einfachste implizite Anpassungen von Integer-Typen wie `short`, `int` und `long` führen zu einem Compilerfehler – der Compiler weigert sich schlicht, aus dem Funktions-Template eine Funktion zu generieren. Ein Beispiel:

```
01  int ival=123;
02  short sval=321;
03  int mval01 = minVal(ival, sval);  // Fehler !!!
```

Der Compiler müsste hier eine Funktion `minVal(int, short)` generieren. Aber bei der Definition des Funktions-Templates für `minVal()` wurde dies nicht vereinbart und mit `T` eben nur ein formaler Datentyp vorgegeben. Wenn Sie ein Funktions-Template mit unterschiedlichen Datentypen als Parameter benötigen, so ist dies auch möglich, wie Sie im folgenden Abschnitt 11.1.3, »Funktions-Template spezialisieren«, noch erfahren werden.

In unserem Beispiel in Zeile **03** könnten Sie den Fehler nur beheben, indem Sie eine explizite Umwandlung des ersten oder zweiten Parameters wie folgt durchführen:

```
03  int mval01 = minVal(ival, static_cast<int>(sval));
```


Hiermit würde eine neue Funktion `minVal(int, int)` instanziiert werden. Natürlich könnten Sie stattdessen auch den ersten Parameter nach `short` umwandeln, um eine Funktion `minVal(short, short)` zu generieren, aber eine Umwandlung von `int` nach `short` ist doch immer etwas kritischer zu betrachten, weil dabei Informationen verloren gehen könnten.

Alternativ würde sich hier auch eine explizite Typangabe für das Template-Argument des formalen Datentyps (hier `T`) anbieten. Darauf wird in Abschnitt 11.1.5, »Explizite Template-Argumente«, noch gesondert eingegangen. Trotzdem will ich Ihnen die explizite Typangabe für das Template-Argument für unser Beispiel nicht vorenthalten:

```
03  int mval01 = minVal<int>(ival, sval);
```

Hiermit generiert uns der Compiler eine Funktion `minVal(int, int)`.

11.1.3 Funktions-Template spezialisieren

Nicht immer eignet sich ein Funktions-Template für Typen gleichermaßen. Bei dem einen oder anderen Typ kann es sein, dass einfach kein sinnvolles Ergebnis zurückgegeben wird, und bei einem anderen Typ (bspw. einer eigenen Klasse) kann es sein, dass bestimmte Anweisungen gar nicht mit dem Typ ausgeführt werden können. Betrachten Sie hierzu folgenden Codeausschnitt, der den Compiler veranlasst, einmal eine `std::string`-Version und einmal eine Version `const char*` aus dem Funktions-Template `minVal()` zu instanziieren:

```
00  // kapitel11/function_template/minval_spezial.cpp
...
01  std::string str01("abba");
02  std::string str02("aabaaaa");
03  std::cout << minVal(str01, str02) << std::endl;
...
04  std::cout << minVal("aaa", "abaaa")  << std::endl; // ???
```

Der Compiler generiert von beiden Versionen eine Funktion. Einmal `minVal(std::string,std::string)` und einmal `minVal(const char*, const char*)`. Im Beispiel von Zeile **03** mit `minVal(std::string,std::string)` wird, wenn Sie sich ein wenig mit der Stringbibliothek und dem darin implementierten `operator<()` befasst haben, der lexikografisch kleinere String zurückgegeben. Die Vergleichsoperatoren von `std::string` arbeiten lexikografisch, das heißt so sortiert, wie dies bspw. bei einem Lexikon üblicherweise der Fall ist. Was ist aber, wenn Sie dies hier gar nicht im Sinn gehabt hatten? Was ist, wenn Sie stattdessen wirklich den String zurückhaben wollen, der kleiner ist – also weniger Buchstaben enthält?

Auch die Generierung und Verwendung von `minVal(const char*,const char*)` in Zeile **04** ist eher fragwürdig, weil ein direkter Vergleich, wie er im Funktions-Template vorgenommen wird, nur die kleinere der beiden Adressen zurückgibt, in der die C-Strings gespeichert sind.

Hier haben Sie also zunächst einen Fall, in dem es nicht zum gewünschten Ergebnis kommt, und einen zweiten Fall, in dem kein sinnvolles Ergebnis geliefert wird. Für solche und weitere Fälle können Sie eine **Spezialisierung** erstellen. Hierfür müssen Sie lediglich dieselbe Funktion separat mit den entsprechenden Typen definieren und überladen.

Allerdings können (sollten) Sie nicht hergehen und eine ganz gewöhnliche Funktionsüberladung implementieren. Gerade bei umfangreicheren Projekten mit mehreren Modulen wirft dies ein Problem auf. Wird nämlich eine Spezialisierung in einem anderen Modul als Funktions-Template definiert, weiß der Compiler nicht mehr, ob hier eine Instanziierung des Funktions-Templates oder einer Spezialisierung vorliegt und meldet gewöhnlich einen Fehler, nämlich eine doppelte Definition.

template<>

Damit Spezialisierungen nicht mit den Funktions-Templates kollidieren, bietet der Standard eine eigene Syntax dafür an, wie Spezialisierungen eingeleitet werden müssen. Hierzu können Sie folgendes Präfix vor die Spezialisierung stellen:

```
template <>
```

Der Datentyp, für den dieses Funktions-Template jetzt verwendet werden soll, wird nach dem Funktionsnamen zwischen spitze Klammern geschrieben. Wenn sich der Datentyp aus den Parametern der Funktion von alleine ergibt, so können Sie diese Angabe zwischen den spitzen Klammern nach dem Funktionsnamen auch weglassen.

Bezogen auf unser Funktions-Template `minVal()` könnte die Spezialisierung für die Typen `std::string` und `const char*` jetzt wie folgt aussehen:

```
00  // kapitel11/function_template/minval_spezial.cpp
...
01  template <typename T>
02  T minVal(T v1, T v2) {
03    if( v1 < v2 ) {
04      return v1;
05    }
06    return v2;
07  }
```



```
08  template<>
09  std::string minVal <std::string>(std::string s1,
                                     std::string s2) {
10    if(s1.size() < s2.size()) {
11      return s1;
12    }
13    return s2;
14  }

15  template<>
16  const char* minVal <const char*> (const char *s1,
                                      const char *s2) {
17    int val1 = std::char_traits<char>::length(s1);
18    int val2 = std::char_traits<char>::length(s2);
19    if( val1 < val2 ) return s1;
20    return s2;
21  }
```

Die Spezialisierung von `std::string` in den Zeilen **08** bis **14** wird mit `template<>` eingeleitet. Der Datentyp für das Funktions-Template wird hier mit `<std::string>` nach dem Funktionsnamen angegeben. In der Funktion wird für den Vergleich die Methode `size()` in Zeile **10** verwendet. Diese Methode liefert die Anzahl der Zeichen zurück, welche sich im String befinden. Je nachdem, welcher der beiden Strings weniger Buchstaben enthält, der wird zurückgegeben.

Ähnlich sieht es auch mit unserer Spezialisierung für `const char*` in den Zeilen **15** bis **21** aus. Für die Länge des C-Strings verwenden wir hierzu in den Zeilen **17** und **18** die statische Methode `length()` aus `<string>`, welche die Länge eines nullterminierten C-Strings zurückgibt. Diese Funktion ist die Alternative zur Uralt C-Funktion `strlen()` aus der Headerdatei `<cstring>`. Auch hier wird entsprechend der C-String zurückgegeben, der weniger Buchstaben enthält – eben der kleinste C-String.

Spezialisierungen werden bevorzugt behandelt

Dank des Schlüsselwortes `template<>` vor einer Spezialisierung können Sie sich sicher sein, dass der Compiler immer erst die Spezialisierung aufruft. Haben Sie also ein Funktions-Template und eine Spezialisierung definiert, wird immer die Spezialisierung verwendet. Aber wenn eine globale Funktion mit dem entsprechenden Datentyp der Parameter definiert wurde, verwendet der Compiler die globale Funktion. Anstatt also eine Spezialisierung zu schreiben, könnten Sie auch einfach eine globale Funktion mit dem passenden Datentyp als Parameter schreiben.

Spezialisierung von Funktions-Template verwenden?

An dieser Stelle muss noch angemerkt werden, dass die Spezialisierung von Funktions-Templates auch zu vielen Kontroversen geführt hat, und auch Probleme mit sich bringt. Ich will hier nicht mehr näher darauf eingehen, aber der folgende Artikel von Herb Sutter beschreibt dies sehr gut: *http://www.gotw.ca/publications/mill17.htm*

11.1.4 Mehrere Template-Parameter verwenden

Natürlich sind Templates nicht nur auf einen formalen Datentyp beschränkt. Ganz im Gegenteil, sofern es Sinn macht, können beliebig viele formale Datentypen definiert werden. Logischerweise sollten sich die formalen Datentypen und die entsprechenden Bezeichner unterscheiden. Hierzu ein einfaches Funktions-Template mit zwei formalen Datentypen:

```
00  // kapitel11/function_template/moreformalT01.cpp
01  #include <typeinfo>
...
02  template <typename T, typename U>
03  void func(T t, U u) {
04    std::cout << "Inhalt : " << t << std::endl;
05    std::cout << "Inhalt : " << u << std::endl;
06    if( typeid(t).name() == typeid(u).name() ) {
07      std::cout << t << " und " << u
                  << " sind vom selben Typ\n";
08    }
09  }
...
10    func("Test", 123.123);   // func(const char*, double)
11    func('A', 12345L);       // func(char, long)
12    func(2000, 3.1415f);     // func(int, float)
13    func(987, 123);          // func(int, int)
14    func(123.123, 133.122);  // func(double, double)
...
```

Bei diesem Funktions-Template werden zwei formale Datentypen (`T` und `U`) verwendet. Häufig sieht man hier auch `T1` und `T2`, aber ich finde, dass es mit `T` und `U` deutlicher zu unterscheiden und besser zu lesen ist. Dieses Template können Sie jetzt mit beliebigen Typen aufrufen und instanziieren lassen, wie Sie in den Zeilen **09** bis **13** auch sehen können. Hinter dem Aufruf finden Sie als Kommentar, was für Funktionen der Compiler hier instanziieren muss. Die Überprüfung mit `typeid` in den Zeilen **05** bis **07** dient nur der Demonstration, um gegebenenfalls mitzuteilen, dass eine

Funktion mit zwei gleichen Typen (wie dies beim Aufruf der Zeilen **12** und **13** der Fall ist) instanziiert wurde – was logischerweise auch gültig ist.

Es sollte hierbei nicht unerwähnt bleiben, dass Sie selbstverständlich trotz formaler Datentypen im Funktions-Template auch eingebaute Datentypen mit den üblichen Features verwenden können:

```
template <typename T>
 void func(T t, double dval) {
...
}
```

Ebenso ist es möglich, eine bereits generische Funktion als Funktions-Template erneut zu überladen oder gar zu spezialisieren. Dies hört sich komplizierter an, als es ist, daher auch hierzu ein einfaches Beispiel:

```
00  // kapitel11/function_template/moreformalT02.cpp
...
01  template <typename T, typename U>
02  void func(T t, U u) {
03    std::cout << "func(T t, U u)\n";
04    std::cout << "Inhalt : " << t << std::endl;
05    std::cout << "Inhalt : " << u << std::endl;
06  }

07  template <typename T>
08  void func(T t, double dval) {
09    std::cout << "func(T t, double dval)\n";
10    std::cout << "Inhalt : " << t << std::endl;
11    std::cout << "Inhalt : " << dval << std::endl;
12  }

13  template <>
14  void func(double dval1, double dval2) {
15    std::cout << "func(double, double)\n";
16    std::cout << "Inhalt : " << dval1 << std::endl;
17    std::cout << "Inhalt : " << dval2 << std::endl;
18  }
...
19    func("Test", 123.123); // func(T t, double dval)
20    func('A', 12345L);     // func(T t, U u)
21    func(2000, 3.1415f);   // func(T t, U u)
22    func(987, 123);        // func(T t, U u)
23    func(123.123, 133.122);// func(double, double)
...
```

In den Zeilen **01** bis **06** sehen Sie unsere bereits bekannte generische Funktion mit mehreren Parametern. Dann, in den Zeilen **07** bis **12**, finden Sie eine weitere überladene Version mit einem zweiten festen Parameter. Zur Demonstration wurde diese Funktion nochmals in den Zeilen **13** bis **18** für `double` spezialisiert (es handelt sich aber immer noch um ein Template). Alternativ hätten Sie stattdessen auch einfach eine globale Funktion `void func(double, double)` definieren können.

Die Spezialisierung in den Zeilen **13** bis **18** wird auch nur dann aufgerufen, wenn `func()` mit zwei `double`-Werten aufgerufen wird, wie dies in Zeile **23** der Fall ist. Die überladene Version von den Zeilen **07** bis **12** wird instanziiert und verwendet, wenn der Funktionsaufruf mit einem `double` als zweitem Parameter erfolgt, wie dies in Zeile **19** der Fall ist. In Zeile **21** wird die Gleitkommazahl mit dem Suffix `f` als `float` gekennzeichnet, daher wird hier die generische Funktion aus den Zeilen **01** bis **06** verwendet. Bei allen anderen Funktionsaufrufen wird ebenfalls die Version mit zwei formalen Datentypen verwendet.

Templates mit variabler Argumentanzahl

Neu in C++11 wurde auch die Möglichkeit eingeführt, Templates mit einer variablen Anzahl an Argumenten zu verwenden. Solche Templates werden auch als *Variadic Templates* bezeichnet. Mehr zu dieser neuen Möglichkeit finden Sie in Abschnitt 11.3, »Templates mit variabler Argumentanzahl (C++11)«.

11.1.5 Explizite Template-Argumente

Die Instanziierungen der Funktions-Templates, wie Sie sie bisher aufgerufen haben, waren alle implizit. Das bedeutet, sobald Sie ein Funktions-Template mit einem bestimmten Typ aufgerufen haben, wurde der formale Datentyp des Templates durch den entsprechenden Typ ersetzt und ein Maschinencode dafür erzeugt. Rufen Sie bspw. unser bekanntes Funktions-Template mit `minVal(123, 456)` auf, erzeugt der Compiler daraus eine Funktion `minVal(int, int)`.

Sie haben aber auch die Möglichkeit, eine solche Instanziierung explizit zu machen – also dem Compiler vorzugeben, von welchem Typ er die formalen Datentypen ersetzen und einen Maschinencode erzeugen soll. Um jetzt explizit die Argumente anzugeben, müssen Sie nur die Template-Argumente in spitzen Klammern hinter dem Funktions-Template-Namen angeben. Rufen Sie das Funktions-Template jetzt mit `minVal<double>(123, 456)` auf, erzeugt der Compiler durch die explizite Angabe von `double` zwischen den spitzen Klammern tatsächlich auch eine Funktion `minVal(double, double)`.

Das folgende Listing demonstriert zwei Fälle, in denen die explizite Angabe von Template-Argumenten sinnvoll sein kann:


```
00 // kapitel11/function_template/explizitTemplate.cpp
...
01 template <typename T, typename U>
02 T half(U u) {
03   return u / 2;
04 }
05 template <typename T>
06 T minVal(T v1, T v2) {
07   if( v1 < v2 ) {
08     return v1;
09   }
10   return v2;
11 }
...
12   int ival = 3;
13   double dval = half (ival);  // Fehler !!!
14   std::cout << dval << std::endl;
15   std::cout << minVal(ival, dval) << std::endl;// Fehler !!!
...
```

Mit dem Aufruf in Zeile **13** kann der Compiler nichts anfangen und weiß nicht, was er für den Rückgabewert verwenden soll. Der Datentyp des Rückgabewertes kann hier nicht aus dem Funktionsaufruf abgeleitet werden. In diesem Beispiel können Sie anhand der Zuweisung sehen, dass es hier ein `double` sein soll. Aber der Compiler muss ja instanziieren und kann das mit der Zuweisung nicht erkennen. Hier müssen Sie also den Datentyp explizit angeben.

Den zweiten Aufruf in Zeile **15** kennen Sie bereits aus einem früheren Beispiel, wo die Typübereinstimmung beschrieben wurde. Auch hier meldet der Compiler einen Fehler, weil es hier nur einen formalen Datentyp mit `T` gibt und beim Funktionsaufruf zwei verschiedene Typen verwendet wurden. Bei Funktions-Templates werden keine impliziten Konvertierungen durchgeführt. Hier können Sie entweder einen `static_cast<>` innerhalb der Parameter verwenden oder eben auch den Datentyp explizit angeben.

Hier nun die Lösung, wie Sie die Typen für den oder die formalen Parameter explizit angeben, womit Sie den Compiler veranlassen, eine Funktion exakt mit diesem Typ zu instanziieren:

```
00 // kapitel11/function_template/explizitTemplate.cpp
...
12   int ival = 3;
13   double dval = half<double, int>(ival);
14   std::cout << dval << std::endl;
15   std::cout << minVal<double>(ival, dval) << std::endl;
...
```

Mit der expliziten Angabe der Template-Argumente in Zeile **13** veranlassen Sie den Compiler, eine Funktion mit `double` als Rückgabewert für `T` und `int` als Parameter für `U` zu instanziieren. In diesem Beispiel hätten Sie auch das `int` in den spitzen Klammern weglassen und nur `half<double>(ival)` verwenden können und so den Compiler den Datentyp für den Parameter wieder selbst ermitteln lassen können. Das `double` würde für den Rückgabewert verwendet. Auch hier werden die formalen Datentypen in derselben Reihenfolge abgearbeitet, wie diese im Funktionskopf des Templates stehen.

Beim zweiten Fall in Zeile **15** wird jetzt ebenfalls durch die explizite Angabe von `double` eine Funktion mit `double`-Parametern (`double minVal(double,double);`) erzeugt – egal, mit welchen Typen Sie die Funktion aufgerufen haben. Dies bedeutet auch, dass eine implizite Konvertierung nach den C++-Regeln möglich sein sollte, was hier mit `int` nach `double` kein Problem darstellt. Sie könnten aber jetzt nicht hergehen und die Funktion mit einem `const char*` aufrufen.

11.1.6 Methoden-Templates

Es ist auch möglich, Methoden-Templates ohne Klassen-Templates (siehe Abschnitt 11.2, »Klassen-Templates«) zu erstellen. Dies kann bspw. nützlich sein, wenn Sie eine Klasse geschrieben haben, in der viele überladene Methoden vorhanden sind, wie dies im folgenden Beispiel der Fall ist:

```
01  class Server {
...
02  public:
03    void send_data(int val) {...}
04    void send_data(double val) {...}
05    void send_data(long long val) {...}
06    void send_data(std::string val) {...}
07    void send_data(const char* val) {...}
08  };
...
09  Server s01;
10  s01.send_data(100);          // int
11  s01.send_data(123.123);      // double
12  s01.send_data(12341234LL);   // long long int
13  s01.send_data(std::string("Text")); // std::string
14  s01.send_data("Text");   // const char*
...
```

Hier finden Sie in der Klasse `Server` die Methode `send_data()` mehrfach überladen für verschiedene Datentypen vor. Wenn hier mehrfach derselbe Code für die überladenen Methoden verwendet wird, können Sie sich die Tipparbeit sparen indem Sie eine Methoden-Template für `send_data()` verwenden. Eine mögliche Implementierung könnte wie folgt aussehen:


```
01  class Server {
...
02  public:
03    template <typename T> void send_data(T val);
04  };
...
05  template <typename T> void Server::send_data(T val) { ... }
...
```

Hier sehen Sie die Deklaration des Methoden-Templates in Zeile **03** und die Definition in Zeile **05**.

11.2 Klassen-Templates

Die Template-Technik, wie Sie diese mit den Funktions-Templates eben kennengelernt haben, ist nicht nur auf Funktionen beschränkt, sondern lässt sich auch mit Klassen verwenden. Um genau zu sein, wäre C++ ohne Klassen-Templates wohl nicht so populär und beliebt wie es heute ist. Die Standardbibliothek definiert nämlich sehr viele Klassen-Templates, wovon Sie selbst (un-)wissentlich bereits `std::vector` oder `std::string` verwendet haben. Bei `std::vector` haben Sie den Typ ja in den spitzen Klammern angegeben. `std::string` hingegen ist eine Instanziierung aus dem Typ `char` und intern wie folgt definiert:

```
typedef basic_string<char> string;
```

Neben den Standardcontainer-Klassen wie `std::vector`, `std::list` usw. sind auch die Stream-Klassen für die Ein-/Ausgabe (siehe Kapitel 13, »Die Stream-Ein-/Ausgabeklassen von C++«) letztlich meist als Instanziierung von `char` implementiert.

Zum Weiterlesen

Bevor Sie sich nach diesem Kapitel hinsetzen und eigene Klassen-Templates wie einen Stapel, Listen usw. schreiben, empfehle ich Ihnen, erst einmal Kapitel 12, »Container, Iteratoren, Algorithmen und Hilfsmittel«, durchzusehen, wo wir auf die in C++ vorhandenen Standardcontainer eingehen werden. Ich bin mir sicher, hier ist fast immer das dabei, was Sie suchen. Wenn nicht, dann wissen Sie anschließend, wie Sie selbst ein Klassen-Template erstellen können.

11.2.1 Klassen-Templates implementieren

Zur Einleitung will ich Ihnen zeigen, wie Sie selbst einen, wenn auch ganz primitiven, Behälter (Containerklasse) erstellen können, welcher (fast) jeden beliebigen Typ aufnehmen kann. Die Einleitung eines Klassen-Templates erfolgt auch hier wieder mit dem aus Funktions-Templates bereits bekannten

```
template <typename T>
```

Auch hier steht `T` wieder für den formalen Datentyp, den der Compiler bei der Instanziierung, also der Erzeugung des Maschinencodes, durch den gewünschten Typ ersetzt. Der Name `T` kann wieder ein beliebiger gültiger Bezeichner sein – `T` ist aber auch hier eine sehr gängige Verwendung. Ebenfalls können Sie wieder `<class T>` statt `<typename T>` verwenden. Auf den Unterschied zwischen `class` und `typename` wurde bereits in Abschnitt 11.1.1, »Funktions-Templates implementieren«, eingegangen.

Hierzu nun die Definition eines kompletten Klassen-Templates:

```
00  // kapitel11/class_template/myContainer01.cpp
...
01  template <typename T>
02  class myContainer {
03    T data;
04  public:
05    void setData(const T& d) {data=d;}
06    T getData() const {return data;}
07  };
...
```

In den Zeilen **01** bis **07** haben Sie ein Klassen-Template `myContainer<T>` definiert, welches sich jetzt fast wie `std::vector<T>` verwenden lässt – natürlich fehlen hier noch viele Funktionalitäten, aber als theoretischer Vergleich taugt es schon. Würden Sie jetzt ein Objekt wie `myContainer<int>` anlegen, würde der Compiler alle formalen Datentypen (hier `T`) durch `int` ersetzen und einen Maschinencode dazu erstellen. Mehr zur Instanziierung von Objekten als Klassen-Template erfahren Sie in Kürze in Abschnitt 11.2.3, »Objekte aus Klassen-Templates erzeugen«.

Klassen-Templates mit struct und union

Klassen-Templates lassen sich neben `class` auch für und mit `struct` und `union` definieren.

11.2.2 Methoden von Klassen-Templates implementieren

Im eben erstellten Beispiel des Klassen-Templates `myContainer<T>` konnten Sie auch sehr schön sehen, dass es ohne Weiteres möglich ist, den formalen Datentyp bei den Parametern oder dem Rückgabewert von Methoden zu setzen.

Werteübergabe als Referenz oder Zeiger bevorzugen

Hier fehlt noch die Empfehlung, dass man die Übergabe von Werten, wie im Beispiel mit `setData(const T&)`, als Referenz (oder Zeiger) realisiert und nicht als Kopie übergibt. Denn wird das Klassen-Template mit Objekten verwendet, könnten doch relativ viele Daten zusammenkommen, die dann unnötigerweise auf das Stack gelegt und wieder abgebaut werden müssten. Allerdings sollten Sie diese Empfehlung unter Vorbehalt betrachten. Wenn Sie nämlich Multithreads verwenden sollten, gilt dies nicht mehr. Mehr dazu können Sie in Abschnitt 15.2.1, »Argumente für den Thread«, nachlesen.

In der Praxis dürfte wohl eher selten die Definition der Methode `inline` innerhalb der Klassen-Template-Definition geschrieben werden. Wie Sie eine Methode außerhalb einer Klasse definieren, wissen Sie ja bereits, aber im Fall einer Methode eines Klassen-Templates ist dies etwas spezieller, und daher muss hier gesondert darauf eingegangen werden. Die Syntax für eine Methode eines Klassen-Templates mit formalen Datentypen sieht demnach wie folgt aus:

```
template <typename T>
void Klassenname<T>::methode( parameter ) { ... }
```

Bezogen auf unser Klassen-Template `myContainer<>` sieht diese Definition der Methode außerhalb der Klassen-Template-Definition jetzt wie folgt aus:

```
00  // kapitel11/class_template/myContainer02.cpp
...
01  template <typename T>
02  class myContainer {
03    T data;
04  public:
05    void setData(T& d);
06    T getData() const;
07  };

08  template <typename T>
09  void myContainer<T>::setData(T& d) {
10   data=d;
11  }
```

```
12  template <typename T>
13  T myContainer<T>::getData() const {
14   return data;
15  }
...
```

Es ist wichtig, dass Sie bei Methoden für Klassen-Templates zusätzlich noch den formalen Datentyp in spitzen Klammern (hier `<T>`) hinter dem Klassennamen angeben, wie hier in den Zeilen **09** und **13** zu sehen ist. Eine Angabe wie `myContainer<T>` entspricht der Angabe eines Datentyps. Ein leeres `myContainer` wäre lediglich ein Template-Name.

Vorläufige Definition

Auch hier gilt wie schon bei den Funktions-Templates, dass die Definition einer Methode auch nur eine vorläufige Definition darstellt, weil der Typ ja immer noch unvollständig ist, da hier ja nur ein formaler Datentyp enthalten ist. Die eigentlichen Methoden werden auch hier erst dann erzeugt, wenn Sie ein Objekt des Klassen-Templates instanziieren.

Methoden überschreiben (spezialisieren)

Wie auch schon bei den Funktions-Templates (siehe Abschnitt 11.1.3, »Funktions-Template spezialisieren«) lassen sich Methoden von Klassen-Templates bei Bedarf mit `template<>` und dem gewünschten Datentyp überschreiben (spezialisieren). Statt `myContainer<T>` muss hier dann der tatsächliche Datentyp für den formalen Datentyp `T` angegeben werden. Eine solche Spezialisierung für `std::string` für die Methoden `setData()` und `getData()` lässt sich bspw. so realisieren:

```
00  // kapitel11/class_template/myContainer02.cpp
...
01  template <typename T>
02  void myContainer<T>::setData(T& d) {
03    data=d;
04  }

05  template <typename T>
06  T myContainer<T>::getData() const {
07    return data;
08  }

09  template <>
10  void myContainer<std::string>::setData(std::string& str){
```



```
11    std::cout << "Spezialisierung von std::string\n";
12    data = str;
13  }

14  template <>
15  std::string myContainer<std::string>::getData() const {
16    std::cout << "Spezialisierung von std::string\n";
17    return data;
18  }
...
```

In den Zeilen **09** bis **13** und **14** bis **18** finden Sie zwei Spezialisierungen für `myContainer<std::string>`, welche verwendet und erzeugt würden, wenn Sie ein Objekt `myContainer<std::string>` instanziieren würden.

11.2.3 Objekte aus Klassen-Templates erzeugen

Wie auch schon bei den Funktions-Templates werden auch Klassen-Templates erst bei der ersten Verwendung mit dem entsprechenden Typ instanziiert. Das Instanzieren bei Klassen-Templates haben Sie ja im Grunde schon des Öfteren in diesem Buch verwendet, bspw. `std::vector<T>`. Neben dem Klassennamen müssen Sie zwischen den spitzen Klammern den Typ für den formalen Datentyp angeben, von dem der Compiler eine Klasse generieren soll.

Hierzu nun ein paar klassische Anwendungsbeispiele unserer Minimal-Beispielklasse `myContainer<T>`:

```
00  // kapitel11/class_template/myContainer02.cpp
...
01  class Simple {
02    int x;
03  public:
04    Simple(int _x=0) : x(_x) {}
05    int get_x() const {return x;}
06  };
...
07  double dval1 = 123.123;
08  double dval2 = 234.234;
09  myContainer<double> Dcont[3];
10  Dcont[0].setData(dval1);
11  Dcont[1].setData(dval2);
12  std::cout << Dcont[0].getData() << std::endl;
13  std::cout << Dcont[1].getData() << std::endl;
```

```
14  Simple Sval(100000);
15  myContainer<Simple> Scont;
16  Scont.setData(Sval);
17  std::cout << Scont.getData().get_x() << std::endl;

18  std::string str("Text");
19  myContainer<std::string> StrCont;
20  StrCont.setData(str);
21  std::cout << StrCont.getData() << std::endl;
...
```

In Zeile **09** wird der Compiler eine Klasse für `myContainer<double>` generieren und dann drei Objekte `Dcont` als C-Style-Array anlegen. Der Compiler ersetzt hierbei den formalen Datentyp `T` im Klassen-Template durch `double`. In den Zeilen **10** bis **13** übergeben wir unserem Container einige Werte und geben diese wieder aus.

Natürlich funktioniert dies auch mit eigenen Klassen, wie dies mit der Instanziierung in Zeile **15** demonstriert werden soll. Hier wird der Compiler eine Klasse für `myContainer<Simple>` generieren. Der Compiler ersetzt hier den formalen Datentyp `T` durch die Klasse `Simple` (Zeile **01** bis **06**). Anschließend wird die Übergabe und Ausgabe eines einzelnen Wertes demonstriert.

Bei der letzten Instanziierung in Zeile **19** wird eine weitere Form der Klasse von unserem Compiler generiert. Hier wird der formale Datentyp durch `std::string` ersetzt. Als Methoden kommen hier in den Zeilen **20** und **21** außerdem die von uns erstellten Spezialisierungen für `std::string` zum Einsatz.

Doppelte Fehlerüberprüfung

Ein weiterer Vorteil beim Instanziieren von Templates im Allgemeinen ist es, dass der Compiler hier zweimal auf Fehler überprüfen muss. Einmal beim Kompilieren der Template-Definition und einmal bei der Generierung von Maschinencode.

11.2.4 Klassen-Templates mit mehreren formalen Datentypen

Wie auch Funktions-Templates können Sie bei Klassen-Templates im Grunde beliebig viele weitere formale Datentypen verwenden. Sie müssen lediglich die formalen Datentypen bei der Definition des Klassen-Templates zwischen den spitzen Klammern, getrennt durch ein Komma, mit dem Schlüsselwort `typename` (bzw. `class`) auflisten. Im folgenden Beispiel sehen Sie eine primitive Version von `std::pair` als eigenes Klassen-Template mit mehreren formalen Datentypen:


```
00  // kapitel11/class_template/myPair01.cpp
...
01  template <typename T, typename U>
02  class myPair {
03    T data01;
04    U data02;
05  public:
06    myPair(const T& t,const U& u) : data01(t), data02(u) {}
07    // ...
08    void print() const;
09  };

10  template <typename T, typename U>
11  void myPair<T, U>::print() const {
12    std::cout << data01 << " : " << data02 << std::endl;
13  }
...
14  std::string month("Januar");
15  int temp = -5;
16  myPair<std::string, int> temperatur(month, temp);
17  temperatur.print();
...
```

Mit diesem Klassen-Template `myPair<T, U>` können Sie eine Klasse erstellen, welche zwei (fast) beliebige Typen verwalten kann. Im Beispiel erzeugen wir eine Instanz mit `std::string` und `int` in Zeile **16**. Zunächst veranlassen wir den Compiler, entsprechenden Maschinencode dafür zu generieren, und dann erzeugen wir das Objekt `temperatur`. Zur Demonstration wurde auch eine Methode in den Zeilen **10** bis **13** geschrieben, weil Sie hier, wenn außerhalb der Klasse definiert, immer beide formalen Datentypen hinter dem Klassennamen in spitzen Klammern angeben müssen.

Hierzu noch ein Tipp, falls Sie ein Klassen-Template, welches mehrere formale Datentypen enthält, nicht mit bspw. `myPair<double, double>` instanziieren wollen, wo der Compiler `T` durch `double` und `U` durch `double` ersetzt. Wenn Sie hier eine Instanz der Klasse mit `myPair<double>` generieren wollen, müssen Sie lediglich den Template-Kopf wie folgt anpassen:

```
00  // kapitel11/class_template/myPair02.cpp
...
01  template <typename T, typename U=T>
02  class myPair {
03    T data01;
04    U data02;
05  public:
```

```
06    myPair(const T& t, const U& u) : data01(t), data02(u) {}
...
07  };
...
08  double x=1.5, y=2.5;
09  myPair<double> coordinates(x, y);
10  coordinates.print();
...
```

Hiermit setzen Sie in Zeile **01** `U` auf den Standardwert `T`, falls in den spitzen Klammern nur ein Datentyp geschrieben wurde und das zweite Argument fehlt, wie es hier in Zeile **09** der Fall ist. Durch die Instanziierung in Zeile **09** ersetzt der Compiler den formalen Datentyp `T` durch `double`, und da der zweite Typ für den zweiten formalen Datentyp nicht angegeben wurde, wird hier durch `U=T` das `U` ebenfalls durch `double` ersetzt. Mehr über das Setzen des Standardwertes von Template-Parametern können Sie in Abschnitt 11.2.6, »Klassen-Templates mit Default-Wert«, nachlesen.

Templates mit variabler Argumentanzahl

Neu in C++11 wurde auch die Möglichkeit eingeführt, Templates mit einer variablen Anzahl an Argumenten zu verwenden. Solche Templates werden auch als *Variadic Templates* bezeichnet. Mehr zu dieser neuen Möglichkeit finden Sie in Abschnitt 11.3, »Templates mit variabler Argumentanzahl (C++11)«.

11.2.5 Klassen-Templates mit Non-Type-Parameter

Ein Template-Parameter muss nicht zwangsläufig ein formaler Datentyp sein. Es dürfen auch echte Datentypen verwendet werden (auch Zeiger und Referenzen sind erlaubt). Solche Typen werden auch als Non-Type-Parameter (= Nicht-Typ-Parameter) bezeichnet. Hier ein solcher Non-Type-Parameter:

```
template<typename T, int val>
...
```

Hier sehen Sie mit `int val` einen solchen Non-Type-Parameter, welcher sich wie eine Art Konstante verwenden lässt. Dieser Non-Type-Parameter **muss** allerdings ein ganzzahliger Typ sein. In der Praxis wird diesem Non-Type-Parameter häufig auch noch ein Default-Wert (siehe Abschnitt 11.2.6, »Klassen-Templates mit Default-Wert«) mitgegeben, der verwendet wird, wenn bei der Instanziierung kein Wert für diesen Parameter angegeben wird:

```
template<typename T, int val=10>  // mit Default-Wert
...
```


Auch hier wollen wir zur Demonstration des Non-Type-Parameters wieder eine vereinfachte Version des neu in C++11 eingeführten fixen Arrays `std::array` erstellen. Hier der Code dazu mit anschließender Erläuterung:

```
00  // kapitel11/class_template/nonType.cpp
...
01  template <typename T, int n=1>
02  class fixedArray {
03    T data[n]{0}; // C++11
04  public:
05    T& operator[](int index);
06    int mySize() const {return n;}
07    void printAll() const {
08      for( auto it : data ) { // C++11
09        std::cout << it << std::endl;
10      }
11    }
12  };

13  template <typename T, int n>
14  T& fixedArray<T, n>::operator[](int index) {
15    if(index >= n) {
16      std::cout << "Zugriff auf Pos. " << index
                  << " von " << n << std::endl;
17      throw std::out_of_range("fixedArray:Pufferüberlauf");
18    }
19    else if(index < 0) {
20      std::cout << "Zugriff auf Pos. " << index
                  << " von " << n << std::endl;
21      throw
          std::out_of_range("fixedArray:Pufferunterlauf");
22    }
23    return data[index];
24  }
...
25  try {
26    fixedArray<int, 10> vals;
27    for(int i=0; i < vals.mySize(); ++i) {
28      vals[i] = i+i;
29    }
30    vals.printAll();
```

```
31    fixedArray<double> dvals;
32    for(int i=0; i < dvals.mySize(); ++i) {
33      dvals[i] = i+i;
34    }
35    dvals.printAll();
36    dvals[5] = 3.1415;  // Bereichsüberschreitung !!!
37  }
38  catch(std::out_of_range& e) {
39    std::cerr << e.what() << std::endl;
40  }
...
```

In Zeile **01** finden Sie neben dem formalen Datentyp `T` den Non-Type-Parameter `n`, welcher hier auch gleich den Default-Wert 1 erhält, weil ein Array mit 0 keinen Sinn macht und ohnehin nicht zulässig ist. In Zeile **03** wird dieser Non-Type-Parameter dann verwendet, um ein Array mit `n` Elementen zu erzeugen. Erzeugt wird das Klassen-Template hier auch erst, wenn es aufgerufen wird. In Zeile **26** wird bspw. der formale Datentyp durch `int` und der Non-Type-Parameter `n` durch `10` ersetzt. Der Compiler instanziiert somit daraus quasi ein `int`-Array mit zehn Elementen. Allerdings bietet unser fixes Array mehr als ein herkömmliches C-Array.

So liefert die Methode `mySize()` die Größe des Arrays (was immer der Non-Type-Parameter ist) zurück. Zusätzlich gibt die Methode `printAll()` alle Elemente auf dem Bildschirm aus. Und das Beste ist, dass unser Array eine Bereichsüber- bzw. Unterschreitung erkennt, weil wir hier den `operator[]` überladen haben, welcher den Bereich, auf den zugegriffen wird, überprüft und gegebenenfalls die Ausnahme `std::out_of_range` auswirft. Wichtig ist auch bei dem `operator[]` und den Methoden, die Sie außerhalb des Klassen-Templates definieren, dass Sie auch den Non-Type-Parameter neben dem formalen Datentyp in den spitzen Klammern mit angeben müssen, wie Sie in Zeile **13** sehen.

In Zeile **31** wird der formale Datentyp `T` durch `double` ersetzt. Da für den non-Type-Parameter keine weiteren Angaben in den spitzen Klammern gemacht wurden und wir einen Default-Wert mit `1` gesetzt haben, wird ein `double`-Array mit einem Element instanziiert. In Zeile **36** machen wir mit Absicht eine Bereichsüberschreitung, was von `catch` in den Zeilen **38** bis **40** abgefangen wird.

11.2.6 Klassen-Templates mit Default-Wert

Wie Sie bereits gesehen haben, können Sie Template-Parameter, wie Funktionsparameter, mit einem Default-Wert (Standardwert) versehen. Wenn bei der Instanziierung eines Templates dann das entsprechende Argument fehlt, wird der Default-Wert dafür verwendet. Die Default-Werte werden gewöhnlich bei der Definition des

Templates angegeben. Default-Werte bei Templates betreffen allerdings nicht nur literale Werte, sondern auch formale Datentypen können gleich mit einem Default-Typ vorbelegt werden, der verwendet wird, wenn bei der Instanziierung kein Typ angegeben wird.

Default-Werte auch für Funktions-Templates

Default-Werte können auch bei Funktions-Templates verwendet werden.

Um nochmals auf das Klassen-Template `fixedArray` zurückzukommen, so könnten Sie bspw. Folgendes verwenden:

```
01  template <typename T=int, int n=10>
02  class fixedArray {
...
03  };
```

Hier verwenden wir für den formalen Datentyp `T` den Default-Wert `int` und für den Non-Type-Parameter `n` den Default-Wert `10`. Diese Werte werden immer dann verwendet, wenn der Anwender des Klassen-Templates bei der Instanziierung keine Angaben macht. Hier gelten jedoch dieselben Regeln wie schon bei den Default-Werten von Funktionen (siehe Abschnitt 5.5.6, »Default-Parameter (Standardparameter)«). Wenn also ein Parameter einen Default-Wert bekommt, müssen alle anderen rechts nachfolgenden Parameter ebenfalls einen Default-Wert bekommen. Hier bedeutet dies konkret, wenn der formale Parameter `T` den Default-Wert `int` bekommen hat, muss der Non-Type-Parameter daneben auch einen Default-Wert bekommen.

Von dem so definierten Klassen-Template `fixedArray` mit den beiden Default-Werten lässt sich jetzt auf drei verschiedene Arten eine Klasse instanziieren:

```
01  fixedArray<> vals;           //fixedArray<int, 10>
02  fixedArray<double> dvals;    //fixedArray<double, 10>
03  fixedArray<char,8> bytes;    //fixedArray<char, 8>
04  fixedArray<100> whatever;    // Fehler !!!
```

In Zeile **01** wird der Compiler eine Klasse mit den Default-Werten generieren, welche ein `int`-Array mit zehn Werten als Eigenschaft aufnehmen kann. In Zeile **02** hingegen wird der formale Datentyp `T` vom Klassen-Template durch `double` ersetzt, und der Compiler generiert hier ein `double`-Array mit zehn (Default-Wert-)Elementen als Eigenschaft. In Zeile **03** hingegen verwenden wir einen Typ für den formalen Datentyp und auch einen Wert für den Non-Type-Parameter, wodurch der Compiler eine Klasse generiert, welche ein `char`-Array mit acht Elementen als Eigenschaft speichern kann.

Alias-Template (C++11)

Eine Instanziierung wie in Zeile **04** vom Beispiel zuvor mit `fixedArray<100>` ist nicht möglich, weil, wenn bei einer Instanziierung ein Argument weggelassen wird, so müssen alle nachfolgenden Argumente weggelassen werden. Es gibt trotzdem eine Möglichkeit, mithilfe von `using` solche Werte in spitzen Klammern zu verwenden:

```
01  template <int MAX>
02  using ifixedArray = fixedArray<int, MAX>;     // C++11
...
03  ifixedArray<100> bigArray01;
...
```

In den Zeilen **01** und **02** finden Sie ein sogenanntes Alias-Template, was neu mit C++11 eingeführt wurde. Hiermit können Sie jetzt, wie in Zeile **03** zu sehen ist, eine Klasse für ein `int`-Array als Eigenschaft generieren lassen, dessen Anzahl Sie beim Aufruf, wie hier mit 100, zwischen den spitzen Klammern festlegen können. Hierbei handelt es sich um eine echte Template-Technik, welche hier in Zeile **01** mit dem Non-Type-Parameter für `MAX` eingeleitet wurde und in Zeile **02** verwendet wird, um ein `fixedArray` vom Typ `int` mit `MAX` Elementen zu generieren. Dieses Klassen-Template kann jetzt mit dem Aliasnamen `ifixedArray` verwendet werden (siehe Zeile **03**). Wohlgemerkt, hiermit wird nach wie vor das Klassen-Template `fixedArray<T, n>` verwendet. Nur erfolgt der Aufruf mit einem Aliasnamen, um die Verwendung zu erleichtern.

Formale Datentypen

Für solche Alias-Templates lassen sich auch formale Datentypen verwenden.

Aliasnamen

Auf ähnliche Weise können Sie auch ein Alias mit `using` erstellen, um beim Aufruf zur Generierung einer Klasse gar keine spitzen Klammern mehr anzugeben:

```
01  using i10fixedArray = fixedArray<>;          // C++11
02  using d10fixedArray = fixedArray<double>;    // C++11
...
03  i10fixedArray intArray;      //fixedArray<int,10>
04  d10fixedArray doubleArray;   //fixedArray<double,10>
...
```

Hier verwenden Sie `using` in den Zeilen **01** und **02**, um einen Aliasnamen für `fixedArray<int, 10>` und `fixedArray<double, 10>` verwenden zu können, was Sie dann in den Zeilen **03** und **04** auch im Einsatz sehen. Solche einfachen Aliasnamen können

Sie aber auch noch mit der alten C++03-Technik mit Hilfe von `typedef` realisieren. Alias-Templates allerdings nicht mehr.

Standardbibliothek

Auch die Standardbibliothek erleichtert Ihnen das Leben mit solchen Aliasnamen. So verdanken Sie es einem `typedef`, dass Sie `std::string` nicht mit `std::basic_string<char>` verwenden müssen, weil hier folgender `typedef` benutzt wurde:

```
typedef basic_string<char> string;
```

11.2.7 Klassen-Templates spezialisieren

Wie auch schon mit Funktions-Templates können Sie Klassen-Templates spezialisieren, wenn ein Datentyp nicht so recht mit einem Klassen-Template funktionieren will oder kein vernünftiges Ergebnis zurückliefert. Im Gegensatz zu Funktions-Templates können Sie Klassen-Templates sogar nur teilweise spezialisieren (partielle Spezialisierung).

Klassen-Template partiell spezialisieren

Um das Thema hier nicht zu umfangreich werden zu lassen, soll an dieser Stelle nochmals das Klassen-Template `myPair` aus Abschnitt 11.2.4, »Klassen-Templates mit mehreren formalen Datentypen«, hierfür verwendet werden. Und zwar soll hierbei das Klassen-Template partiell spezialisiert werden, wenn beim Aufruf einer der beiden formalen Datentypen `std::string` enthält.

Folgende Aufrufe sollten also partiell spezialisiert werden:

```
01  myPair<int, std::string> data01(86153, "Augsburg");
02  myPair<std::string, int> data02("Friedberg", 86163);
```

Und so kann die partielle Spezialisierung dazu aussehen:

```
00  // kapitel11/class_template/specialization.cpp
...
01  template <typename T, typename U=T>
02  class myPair {
03    T data01;
04    U data02;
05  public:
06    myPair(const T& t, const U& u) : data01(t), data02(u) {}
...
07  };
```

```
08  template <typename T>
09  class myPair<T, std::string> {
10    T data01;
11    std::string data02;
12  public:
13    myPair(const T& t, const std::string& str) :
      data01(t), data02(str) {
14      std::cout << "myPair<T, std::string>\n";
15    }
...
16  };

17  template <typename T>
18  class myPair<std::string, T> {
19    std::string data01;
20    T data02;
21  public:
22    myPair(const std::string& str, const T& u) :
      data01(str), data02(u) {
23      std::cout << "myPair<std::string, T>\n";
24    }
...
25  };
...
26  myPair<int, std::string> data01(86153, "Augsburg");
27  myPair<std::string, int> data02("Friedberg", 86163);
...
```

In den Zeilen **01** bis **07** sehen Sie unser ursprüngliches Klassen-Template mit zwei formalen Datentypen. In den Zeilen **08** bis **16** finden Sie die partielle Spezialisierung, die der Compiler verwendet, wenn das Klassen-Template mit `myPair<T, std::string>` aufgerufen und daraus eine Klasse instanziiert werden soll, wie dies in Zeile **26** mit `myPair<int,std::string>` der Fall wäre.

Für die partielle Spezialisierung muss im Klassen-Template natürlich der formale Datentyp (hier wieder `T`) zwischen die spitzen Kammern (Zeile **08**) gestellt werden, der nicht spezialisiert wird und erst beim Aufruf seine Typen erhält. Nach dem Klassennamen (Zeile **09**) werden jetzt in den spitzen Klammern alle Template-Argumente (hier `T`) und die Datentypen (hier `std::string`) aufgelistet, für die diese partielle Spezialisierung gelten soll.

Dass es auch in einer anderen Reihenfolge möglich ist, soll die partielle Spezialisierung in den Zeilen **17** bis **25** zeigen, welche der Compiler für die Instanziierung verwendet, wenn `myPair<std::string, int>` aufgerufen wird, wie dies in Zeile **27** der Fall ist.

Unser Beispiel hat aber einen Schönheitsfehler, wenn Sie das Klassen-Template mit `myPair<std::string, std::string>` aufrufen wollen. Der Compiler kann jetzt keine Instanziierung machen, weil er nicht weiß, ob er `myPair<T, std::string>` oder `myPair<std::string, T>` verwenden soll. Gäbe es hier nur eine der beiden partiellen Spezialisierungen, so wäre das Problem gelöst, und es würde jetzt die eine partielle Spezialisierung verwendet werden. Würde dann die andere partielle Spezialisierung aufgerufen, würde der Compiler eine Instanz aus `myPair<T, U>` generieren.

Um im vorliegenden Fall keine der partiellen Spezialisierungen streichen zu müssen, würde sich eine vollständige Spezialisierung von `myPair<std::string, std::string>` anbieten.

Was kann nicht partiell spezialisiert werden?

Im Gegensatz zu einem Klassen-Template ist es nicht möglich, ein Funktions-Template partiell zu spezialisieren. Auch eine Methode eines Klassen-Templates kann nicht partiell, sondern muss vollständig spezialisiert werden.

Klassen-Template voll spezialisieren

Wenn bei Klassen-Templates keines der Template-Argumente so recht passen will, dann können Sie auch eine vollständige Spezialisierung des Klassen-Templates erstellen. Im Beispiel zuvor war ja die Rede von einer vollständigen Spezialisierung von `myPair<std::string,std::string>`. Realisieren können Sie eine vollständige Spezialisierung wie folgt:

```
00  // kapitel11/class_template/specialization.cpp
...
01  template <>
02  class myPair<std::string, std::string> {
03    std::string data01;
04    std::string data02;
05  public:
06    myPair(const std::string& t, const std::string& str) :
      data01(t), data02(str) {
07      std::cout << "myPair<std::string, std::string>\n";
08    }
...
09  };
...
10 myPair<std::string,std::string> data04("Augsburg","Friedberg");
```

Zum vollständigen Spezialisieren eines Klassen-Templates bleibt die spitze Klammer, wie auch bei den Funktions-Templates, leer (siehe Zeile **01**). Der oder (wie hier)

die Datentypen, für welche das Klassen-Template jetzt spezialisiert werden soll, muss nach dem Klassennamen in den spitzen Klammern geschrieben werden, wie hier in Zeile **02** mit `<std::string,std::string>` zu sehen ist. Durch die vollständige Spezialisierung des Klassen-Templates `myPair<T,U>` durch `myPair<std::string,std::string>` wird jetzt nach dem Aufruf in Zeile **10** vom Compiler die Spezialisierung der Zeilen **01** bis **09** für die Generierung verwendet.

11.2.8 Klassen-Template als Parameter an Funktion/Methode

Ein Thema habe ich bewusst bei den Funktions-Templates noch unterschlagen. Und zwar geht es um die Möglichkeit, ein Klassen-Template als Objekt einer Funktion bzw. Methode zu übergeben. Da der Compiler erst beim Aufruf eine entsprechende Klasse aus einem Klassen-Template generiert, muss logischerweise auch der Funktionsparameter ein Template-Argument sein. Da der Datentyp des Funktionsparameters wegen des formalen Datentyps des Klassen-Templates unterschiedlich sein kann, muss eine Funktion auch ein Funktions-Template sein.

Im Grunde ist die Definition dieses Funktions-Templates relativ einfach. Gewöhnlich verwendet man auch hier eine Referenz (bzw. auch `const`-Referenz) vom Typ des Klassen-Templates. Hierzu ein kurzer Codeausschnitt, der dies demonstrieren soll:

```
00  // kapitel11/class_template/func_class_param01.cpp
...
01  template <typename T>
02  class myContainer {
03    T data;
04  public:
05     myContainer(const T& t) : data(t) {};
06     void print() { std::cout << data << std::endl;}
07  };

08  template <typename T>
09  void function(myContainer<T>& myObj) {
...
10    myObj.print();
11  }
...
12    std::string str("Ein Text");
13    myContainer<std::string> data01(str);
14    function(data01);
...
```


In Zeile **08** bis **11** sehen Sie die Funktion, welche als Parameter ein Objekt des Klassen-Templates `myContainer<T>` erwartet. Das Template-Argument `T` entspricht hier dann dem Datentyp, mit dem Sie das Objekt des Klassen-Templates instanziiert haben. In Zeile **13** wird bspw. eine Klasse mit `myContainer<std::string>` instanziiert. Somit generiert auch der Compiler eine Funktion mit `function(myContainer<std::string>&)` für uns, dank des Funktionsaufrufes in Zeile **14**.

11.3 Templates mit variabler Argumentanzahl (C++11)

Neu in C++11 ist hinzugekommen, Templates mit beliebig vielen Argumenten zu erstellen. Diese neue Technik wird häufig auch als *Variadic Templates* bezeichnet. Die Syntax dieser neuen Technik sieht allerdings zunächst etwas ungewöhnlich aus, weil hier in der Typliste die drei Punkte `...` (auch Ellipse genannt) verwendet werden.

Anwendungsgebiet

In der Praxis dürfte wohl die Implementierung von solchen *Variadic Templates* ganz besonders für diejenigen interessant sein, die sich mit dem Erstellen von Bibliotheken beschäftigen. Wer jetzt anschließend allerdings hergeht und hiermit ein Klassen-Template erstellen will, welches eine variable Anzahl an Typen aufnehmen kann, dem möchte ich vorher noch den sequenziellen Datentyp `std::tuple` (siehe Abschnitt 12.6.2, »Tupel, das bessere pair (C++11)«) ans Herz legen, der ebenfalls mit C++11 eingeführt wurde, und auch meistens die Lösung sein sollte, nach der Sie suchen.

Hierbei werden mit `...` die Parameter zu einem *Parameter-Pack* zusammengepackt und auch wieder entpackt. Dies hängt davon ab, auf welcher Seite der Operator `...` steht. Das Ganze in der Theorie zu lesen, hilft hier nicht weiter, daher soll Ihnen ein Listing den Sachverhalt demonstrieren:

```
00  // kapitel11/variadic_template/variadic01.cpp
...
01  #include <typeinfo>       // operator typeid
02  #include <unordered_map>  // std::unordered_map
03  std::unordered_map <std::size_t,std::string> type_names;

04  template <typename T>
05  void output(T val) {
06   std::cout << type_names[typeid(val).hash_code()] << ":";
07   std::cout << val << std::endl;
08  }
```

```
09  template<typename First, typename ... Rest>
10  void output(First first, Rest ... rest) {
11    output(first);
12    output(rest ...);
13  }
...
14  type_names[typeid(const char*).hash_code()]="const char*";
15  type_names[typeid(char).hash_code()] = "char";
16  type_names[typeid(int).hash_code()] = "int";
17  type_names[typeid(double).hash_code()] = "double";
18  type_names[typeid(bool).hash_code()] = "bool";

19  output("ende", 2, 3.14, 'A', false);
20  output(3.1415);
...
```

Die Verwendung von `type_names` in den Zeilen **03**, **06** und **14** bis **18** können Sie ignorieren, sie dient nur der Informationsausgabe, um welchen Typ es sich hierbei handelt. Hierfür wurde die neu in C++11 eingeführte Funktion `hash_code()` aus `<typeinfo>` verwendet.

Betrachten Sie zunächst den Funktionsaufruf von `output()` in Zeile **19** mit beliebigen Parametern. Die Funktion wurde überladen und ist daher in doppelter Ausführung vorhanden. Aber dazu gleich mehr. Mit dem Funktionsaufruf wird das *Variadic Template* in den Zeilen **09** bis **13** aufgerufen.

Zuerst müssen wir die übergebene Liste in zwei Teilen zwischen dem ersten und den restlichen Elementen aufteilen. In Zeile **09** haben Sie den Typ `First` für das erste Element und `Rest` als Parameter-Pack für die restlichen Elemente festgelegt. Beachten Sie hier die Position des `...`-Operators, der links von `Rest` steht, was bedeutet, dass der Operator die einzelnen Parameter zu einem Parameter-Pack **verpackt** hat.

Im Funktionskopf von Zeile **10** finden Sie jetzt die einzelnen Parameter. In `First first` haben Sie immer jeweils das erste Element in der Liste, und in `Rest ... rest` finden Sie die restlichen Elemente. Hier sehen Sie auch gleich, dass jetzt der `...`-Operator rechts von `Rest` steht, und dies bedeutet, dass hier das Parameter-Pack wieder **entpackt** wird. In `first` findet sich jetzt das erste Element in der Liste, welches hier als `const char*` ermittelt wird und in Zeile **11** die überladene Version `output()` in den Zeilen **04** bis **08** mit dem C-String `"ende"` aufruft. Die überladene Version sollte hierbei ein Funktions-Template oder eine passende Überladung sein, damit der Compiler eine entsprechende Version generieren kann. In der Funktion geben wir in Zeile **06** den Typ und in Zeile **07** den Inhalt aus, ehe die Funktion wieder mit dem Aufrufer in Zeile **12** fortfährt.

In Zeile **12** starten Sie jetzt einen rekursiven Aufruf (Rekursion = Selbstaufruf), indem Sie das restliche Parameter-Pack ohne `First` erneut für die Funktion in den Zeilen **09** bis **13** aufrufen. Der Trick ist dabei, dass `rest` bei jedem erneuten Selbstaufruf um das erste Element `first` gekürzt wird. Der erneute Funktionsaufruf des Parameter-Packs würde somit wie folgt aussehen:

```
output(2, 3.14, 'A', false);
```

Jetzt beginnt das Spielchen wieder von vorne, nur dass jetzt `2` als `int`-Wert für `First` ermittelt wird, und der `Rest` als Parameter-Pack wäre `<double, char, bool>`. In jedem weiteren Selbstaufruf passiert jetzt dasselbe und die Funktion wird immer ohne das Element `first` erneut aufgerufen:

```
output(3.14, 'A', false);
output('A', false);
output(false);
```

Die Rekursion wird beendet, wenn `rest` nur noch einen Parameter enthält. In dem Fall wird nur noch das Funktions-Template aufgerufen. Ebenfalls nur das Funktions-Template wird aufgerufen, wenn Sie die Funktion `output()` mit nur mit einem Parameter aufrufen, wie dies in Zeile **20** mit dem `double`-Wert demonstriert wurde.

Schleifen nicht möglich

Die Frage werden Sie sich sicherlich hier gestellt haben, warum ich hier keine Schleifen wie `for` oder `while` verwendet haben? Das ist nicht möglich, weil das Parameter-Pack zur Übersetzungszeit behandelt werden muss.

Das Programm bei der Ausführung:

```
const char* : ende
int : 2
double : 3.14
char : A
bool : 0
double : 3.1415
```

operator<< vorhanden?

Dass unser Beispiel hier so glatt funktioniert, liegt auch daran, dass `operator<<` für alle Typen, die wir verwendet haben, definiert war. Ist dies nicht der Fall, müssen Sie natürlich den `operator<<` in Ihrer Klasse definieren. Ansonsten würde sich der Compiler bei Ihnen beschweren.

sizeof ...-Operator

Einen Operator gibt es dann mit `sizeof...()` für Parameter-Packs doch noch. Mit diesem Operator können Sie zur Übersetzungszeit die Anzahl der vorhandenen Elemente des Parameter-Packs herausfinden. Hier der Operator im Einsatz:

```
00  // kapitel11/variadic_template/variadic02.cpp
...
01  template <typename ... Args>
02  auto countArgs(Args ... args) -> int {
03    return (sizeof ... (args) );
04  }
...
05  std::cout << countArgs("eins", 2, 3.14) << std::endl;
```

Parameter-Pack in Tupel konvertieren

Früher oder später werden Sie auf den neuen sequenziellen Typ `std::tuple` stoßen und dessen komfortablen Zugriff auf die einzelnen Elemente mit `get<N>` zu schätzen wissen. Schließlich ist `std::tuple` auch das prominenteste Beispiel für den Einsatz von *Variadic Templates*. Auf `std::tuple` wird noch in Abschnitt 12.6.2, »Tupel, das bessere pair (C++11)«, etwas umfassender eingegangen. Wenn Sie bspw. ein Parameter-Pack haben und daraus ein `std::tuple` machen wollen, so ist dies im Grunde mit `std::make_tuple` keine große Sache. Folgender Codeausschnitt demonstriert Ihnen, wie Sie aus einem Parameter-Pack ein Tupel machen können:

```
00  // kapitel11/variadic_template/variadic03.cpp
...
01  #include <tuple>
...
02  template <typename ... Args>
03  auto conv2tuple(Args ... args) ->
          decltype(std::make_tuple(args...)) {
04    return std::make_tuple(args...);
05  }
...
06  auto myTuple = conv2tuple("ende", 2, 3.14, 'A', false);
07  std::cout << std::get<0>(myTuple) << std::endl;
08  std::cout << std::get<1>(myTuple) << std::endl;
09  std::cout << std::get<2>(myTuple) << std::endl;
10  std::cout << std::get<3>(myTuple) << std::endl;
11  std::cout << std::get<4>(myTuple) << std::endl;
```

Das Prinzip ist einfach. In Zeile **06** rufen Sie die Funktion `conv2tuple()` mit verschiedenen Parametern auf, wo diese zu einem Parameter-Pack verpackt werden. In der

Funktion wird dieses Paket wieder entpackt, wobei wir hier auch gleich den Rückgabewert mit `decltype` vom Compiler ermitteln lassen. In Zeile **04** machen wir aus der Liste von Parametern mithilfe von `std::make_tuple()` ein `std::tuple` und weisen den Rückgabewert in Zeile **06** `myTuple` zu. In den Zeilen **07** bis **11** verwenden wir die Funktion `get<>`, um die Elemente des Tupels auszugeben – und nein, auch bei einem Tupel kann nicht mit Schleifen wie `for` oder `while` durchlaufen werden, weil es sich letztendlich auch wieder um *Variadic Templates* handelt.

»Variadische« Klassen-Templates oder doch lieber ein Tupel?

In der Tat ist es durchaus möglich, »variadische« Klassen-Templates zu generieren. Hierfür müssen Sie allerdings die Klassen rekursiv implementieren, was ziemlich komplex werden kann. Hier bietet das neue `std::tuple`, welches ebenfalls in C++11 eingeführt wurde, viel mehr Flexibilität und Komfort. Wer sich trotzdem dafür interessiert, der kann sich ja mal dieses Beispiel auf der folgenden Webseite ansehen:

http://thenewcpp.wordpress.com/2012/02/15/variadic-templates-part-3-or-how-i-wrote-a-variant-class

Inhalt

Anhang

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

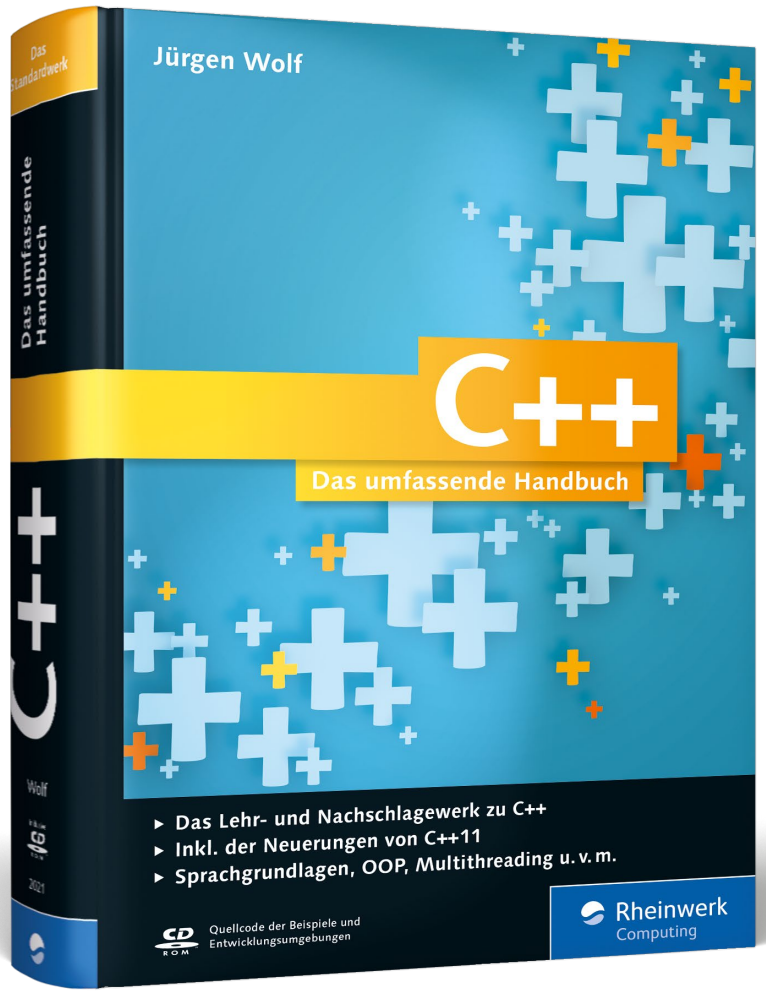
V

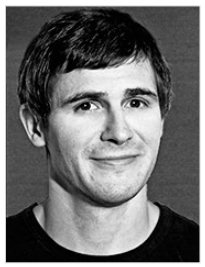
W

X

Y

Z





Jürgen Wolf ist Softwareentwickler, Digitalfotograf und Autor aus Leidenschaft. C/C++, Perl, Linux und die Bildbearbeitung mit Photoshop Elements und GIMP sind seine Themen. Sein Traum: ein ruhiges Leben im Westen Kanadas. Und Bücher auf der Veranda zu schreiben. Besuchen Sie seine umfangreiche Website: www.pronix.de.

Jürgen Wolf

C++ – Das umfassende Handbuch

1.062 Seiten, gebunden, mit CD, 3. Auflage 2014
39,90 Euro, ISBN 978-3-8362-2021-7

www.rheinwerk-verlag.de/3278