Grundkurs C

DIE LESEPROBE

Kapitel 3

Basisdatentypen in C

In diesem Kapitel geht es um die Basisdatentypen, ohne die Sie kein vernünftiges Programm schreiben und auch kaum eine sinnvolle Berechnung ausführen können. Außerdem werden sämtliche komplexen Datenstrukturen von den einfachen Datentypen abgeleitet.

Ebenso wie Sie nicht ohne Kochtopf kochen können, funktioniert eine Programmiersprache nicht ohne grundlegende Datentypen, mit deren Hilfe Sie Zahlen in einer Variablen ablegen können. Ebenso wenig würde die Mathematik ohne Variablen funktionieren, von denen die bekannteste wahrscheinlich *x* ist. Auf ähnliche Weise funktionieren die Variablen in C: Man kann für Variablen jederzeit den entsprechenden Wert einsetzen. Während Sie allerdings in einen Kochtopf alles Mögliche hineinschneiden können, müssen Sie bei den Basisdatentypen genau auf die »Zutaten« achten. Dies ist auch in der Mathematik so, wo Sie auch nicht mit allem alles tun dürfen. In C sind hierfür von Haus aus Datentypen für Ganzzahlen, Fließkommazahlen und Zeichen vorhanden. Diese sogenannten *primitiven Datentypen* müssen Sie nicht extra definieren, denn sie sind dem Compiler von vornherein bekannt und funktionieren, ohne dass Sie zusätzliche Bibliotheken einbinden müssen.

3.1 Variablen

Bevor wir Ihnen Variablen näherbringen, müssen wir ein paar Worte über Speicheradressen verlieren, weil diese in C sehr wichtig sind. Sicherlich wissen Sie, dass Ihr Computer Speicher besitzt, und möglicherweise haben Sie schon einmal welchen in Ihren PC eingebaut.

Aber wie legt Ihr PC die Daten im Speicher ab, und woher weiß er, wo er diese vorher abgelegt hat? Die Antwort ist, dass Ihr Prozessor eine interne

Schaltung besitzt – den sogenannten *Adressbus* – der Ihren Speicherbaustein anspricht und die Positionen der einzelnen Zeichen, die in diesem Speicherbaustein stehen, auf Zahlen abbildet. Diese Zahlen geben dann z. B. an, dass sich ein bestimmtes Zeichen an der 1000. Position befindet. Genau diese Positionsangabe ist die *Speicheradresse*. Da ein Zeichen des Ausführungszeichensatzes, das oft im ASCII-Code codiert ist, ein Byte (also 8 Bits) umfasst, gibt die Speicheradresse die Position des entsprechenden Bytes im Speicher an. Der Prozessor hat nun spezielle Befehle, um diese Bytes aus dem Speicher zu lesen und an einer bestimmten Position in diesem abzulegen. An dieser Position kann zum Beispiel der Inhalt einer Variablen stehen.

Eine Variable ist damit im Grunde genommen nichts anderes als eine Adresse im Hauptspeicher. Dort legen Sie die Daten ab und greifen später, wenn Sie den Inhalt wieder benötigen, auf sie zurück. Auch dies erledigen spezielle Prozessorbefehle und einige interne Speicherstellen, die *Register* genannt werden. Wie dies im Einzelnen funktioniert, brauchen Sie nicht im Detail zu verstehen. Sie müssen nur wissen, dass alle Variablen eine konkrete Speicheradresse besitzen, die der Compiler zusammen mit dem Namen in einer Variablentabelle ablegt.

Um nun programmtechnisch ohne kryptische Adressangaben und Prozessorbefehle wie `MOV EAX,[DS:ESI]` auf diese Adressen im Arbeitsspeicher zurückgreifen zu können, benötigen Sie einen eindeutigen Namen. Genau dies ist der *Bezeichner*. Der Compiler wandelt diesen Namen dann später in eine Adresse um und erzeugt auch die Variablentabelle automatisch. Natürlich belegt jede dieser Variablen einen gewissen Speicherplatz. Wie viel das ist, hängt davon ab, welchen Datentyp Sie verwendet haben, wie viel Platz dieser auf einem bestimmten System beansprucht und mit welchen Werten er implementiert wurde. Der Standard schreibt hier nur eine Mindestgröße für die einzelnen Typen vor.

3.2 Deklaration und Definition

Bevor Sie eine Variable als Platzhalter für einen Wert verwenden können, müssen Sie dem Compiler den Datentyp und den Bezeichner mitteilen.

Nur so kann die Variablentabelle korrekt gefüllt werden. Dieser Vorgang der Bekanntmachung wird als *Deklaration* bezeichnet. Was ein gültiger Bezeichner ist, haben Sie bereits in Abschnitt 2.4.1, »Bezeichner«, erfahren. Die Datentypen lernen Sie in diesem Kapitel kennen.

Damit eine Variable also verwendet werden kann, muss Speicherplatz für sie reserviert werden. Für das konkrete Speicherobjekt der Variablen im Programm bzw. im ausführbaren Code wird die *Definition* vereinbart.

Eine Definition, also eine Wertzuweisung, darf selbstverständlich mehrmals im Code vorkommen, eine Deklaration hingegen nur einmal innerhalb der aktuellen Funktion. Sie können beispielsweise eine Ganzzahlvariable `ivar` wie folgt vereinbaren:

```
int ivar;
```

Hier deklarieren Sie eine Variable vom Datentyp `int` mit dem Bezeichner `ivar` und definieren diese damit auch gleichzeitig. Somit ist es in diesem Beispiel nicht falsch zu sagen, dass eine Definition gleichzeitig auch eine Deklaration ist.

Mehrere Bezeichner desselben Datentyps lassen sich auch in einer Zeile vereinbaren. Sie werden dann durch Kommata getrennt:

```
int ivar1, ivar2, ivar3;
```

Wenn Sie eine Variable nur deklarieren wollen, müssen Sie das Schlüsselwort `extern` vor sie setzen:

```
// datei-01.c
extern int ivar;  // Deklaration
```

Mit diesem Schlüsselwort bestimmen Sie, dass kein Speicherplatz für `ivar` reserviert wird und dass Sie die Definition dieser Variablen (gewöhnlich) in einem anderen Modul vornehmen. Die Definition und die Speicherplatzreservierung erfolgen jetzt beispielsweise in einem anderen Quelltextmodul, z. B. durch:

```
// datei-02.c
int ivar;
```

Wozu überhaupt zwischen Deklaration und Definition unterscheiden?

An dieser Stelle mag Ihnen der Unterschied zwischen einer Deklaration und einer Definition noch etwas trivial erscheinen. Aber wenn Sie Ihren Quellcode auf mehrere Quelltextmodule und auf mehrere Funktionen aufteilen, ist es essenziell, diesen Unterschied zu kennen und Deklarationen und Definitionen voneinander trennen zu können. Denn irgendwann haben Sie Code geschrieben, den Sie mehrfach verwenden wollen, um das Rad nicht immer wieder neu zu erfinden. Und schon lagern Sie einige Teile Ihrer Deklarationen in eine eigene Header-Datei aus und müssen dann sauber zwischen Deklaration und Definition unterscheiden.

3.3 Initialisierung und Zuweisung von Werten

Nachdem Sie eine Variable definiert haben, besitzt diese noch keinen festen Wert. (Ausnahmen, die noch behandelt werden, sind globale Variablen, die im gesamten Programm gelten, sowie mit `static` ausgezeichnete Variablen.) Aber wenn die Variable noch keinen festen Wert hat, was steht dann in ihrer Speicheradresse? Meistens steht dort ein zufälliger Wert, der sich einfach vorher schon darin befunden hat (genauer gesagt, ein undefinierter Wert).

Einen Wert müssen Sie der Variablen erst noch zuweisen. Dies können Sie beispielsweise mit dem Zuweisungsoperator (`=`) erledigen:

```
int ivar;          // Definition
ivar = 12345;      // Zuweisung
```

Oder Sie verwenden eine Eingabefunktion wie etwa `scanf()`, nachdem Sie das entsprechende Kapitel gelesen haben. Sie können aber auch gleich bei der Definition der Variablen einen Initial- bzw. Anfangswert zuweisen. Dieser Vorgang wird als *Initialisierung* bezeichnet. Im folgenden Beispiel

wird einer Variablen mit dem Bezeichner `ivar` vom Datentyp `int` direkt bei der Definition der Wert 12345 zugewiesen:

```
int ivar = 12345;   // Initialisierung
```

Eine Initialisierung findet somit ausschließlich bei der Definition einer Variablen statt, wohingegen eine *Zuweisung* jederzeit und auch mehrmals notiert werden kann.

Variablen sofort mit einem Wert initialisieren

Damit Sie nicht versehentlich mit einer nicht initialisierten Variablen arbeiten, was zu undefinierten Ergebnissen führen kann, sollten Sie es sich zur Gewohnheit machen, Variablen schon bei der Definition mit einem Anfangswert zu initialisieren, zum Beispiel so:

```
int ivar = 0; // ivar mit 0 initialisiert
```

Wenn Ihr Compiler hier meckert, z. B. mit der Meldung VARIABLE IVAR IS NOT USED, dann nutzen Sie einen älteren C99-Compiler und können nicht Initialisierung und Definition in einer einzigen Zeile erledigen. In diesem Fall müssen Sie zwei getrennte Zeilen verwenden:

```
int ivar;
ivar=0;
```

3.4 Datentypen für Ganzzahlen

Für die Speicherung von vorzeichenbehafteten Ganzzahlen (hier zunächst: *standard signed integer types*) bietet C die in Tabelle 3.1 aufgelisteten Datentypen. Zusätzlich finden Sie in der folgenden Tabelle die *Mindestgrößen von Werten* und das *Formatzeichen*, um den Typ mit den Funktionen der `printf()`-Familien formatiert auszugeben oder mit Funktionen der `scanf()`-Familie einzulesen. Die tatsächlichen Wertebereiche sind besonders beim Typ `int` aber meistens größer.

Datentyp	Wertebereich (mindestens)	Formatzeichen
`signed char`	–128 +127	`%hhd` (für Dezimal) `%c` (für Zeichen)
`short`	–32.768 +32.767	`%hd` oder `%hi`
`int`	–32.768 +32.767	`%d` oder `%i`
`long`	–2.147.483.648 +2.147.483.647	`%ld` oder `%li`
`long long` (seit C99)	–9.223.372.036.854.775.808 +9.223.372.036.854.775.807	`%lld` oder `%lli`

Tabelle 3.1 Grundlegende Datentypen für Ganzzahlen

Neben diesen vorzeichenbehafteten fünf `signed`-Ganzzahltypen (engl. *standard signed integer types*) kann es abhängig von der Implementation auch erweiterte `signed`-Ganzzahltypen (engl. *extended signed integer types*) wie etwa `__int128` geben. Der neue C23-Standard enthält allerdings nur `__int128` und `__int256` als neuen primitiven Datentyp, und das auch nur optional. Die Arbeit mit beliebig langen Zahlen muss deshalb auch separat behandelt werden (siehe Bonus-Kapitel »Mit beliebig langen Zahlen arbeiten«, *www.rheinwerk-verlag.de/5984*).

Der bevorzugte Datentyp für Ganzzahlen lautet gewöhnlich `int`, weil per Definition die meisten Systeme mit ihm am natürlichsten umgehen und häufig auch am schnellsten rechnen können. Benötigen Sie mehr Bits (bis zu 128), steht Ihnen `long` oder `long long` zur Verfügung. Bei kleineren Werten können Sie hingegen `short` verwenden. Eigentlich heißen die korrekten Typnamen `short int`, `int`, `long int` und `long long int`. In der Praxis werden normalerweise nur `short`, `long` und `long long` verwendet (aber nicht immer).

Überlauf vorzeichenbehafteter signed-Ganzzahlwerte

Bei `signed`-Ganzzahltypen kann es zu einem Werteüberlauf kommen. Dies bedeutet, dass z. B. bei einer Multiplikation mehr Bits entstehen, als die verwendete Variable noch aufnehmen kann. Der Standard schreibt nicht vor, wie das System auf einen Überlauf von `signed`-Ganzzahltypen reagieren soll. Deshalb lässt sich nicht voraussagen, wie sich das Programm weiter verhält (engl. *undefined behavior*). Wenn Sie beispielsweise zum maximalen Wert einer positiven Ganzzahl einen positiven Wert hinzuaddieren, befinden Sie sich unter Umständen plötzlich im negativen Bereich und haben einen Überlauf (engl. *overflow*) erzeugt. Ein Beispiel:

```
int iVal = INT_MAX; // INT_MAX enthält max. Wert für int
iVal += 2; // Überlauf (undefined behavior)
```

Beim Programmieren sind Sie selbst dafür verantwortlich, dass es nicht so weit kommt. Beziehungsweise müssen Sie diesen Zustand separat in Ihrem Programm feststellen. Leider können Sie jedoch in Standard-C nicht auf die Prozessor-Flags zugreifen, die z. B. einen Overflow direkt anzeigen (sogenannte *Overflow-Flags*). Am besten benutzen Sie in C möglichst Datentypen von der doppelten Bitbreite, die Sie eigentlich benötigen.

Die Größe der Typen `int`, `short` und `long` ist nicht festgelegt. `int` ist mindestens so groß wie `short`. Der Datentyp `long` hingegen hat mindestens die Ausdehnung eines `int`. Daraus können Sie auch folgern, dass nicht hundertprozentig gesagt werden kann, wie viele Bits ein bestimmter Typ verwendet. Allerdings können Sie sich mit Sicherheit auf folgende Reihenfolge verlassen:

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

`<=` bedeutet hier: ist kleiner oder gleich.

Ähnlich sieht dies beim Abspeichern von einzelnen Zeichen aus. Auch hier hängt die Anzahl der Bits pro Zeichen von Ihrem System ab. Der Standard schreibt hier nur vor, dass `signed char` der kleinste und `long long` der größte

Typ für Ganzzahlen ist. Die anderen eingebauten Typen `short`, `int` und `long` liegen irgendwo dazwischen. Auf vielen modernen Betriebssystemen ist heute ein `char` 16 Bit und ein `long long` 64 Bit breit. Dies liegt daran, dass für Zeichen mittlerweile nicht mehr der ASCII-Code, sondern der internationale Unicode verwendet wird, der einem Zeichen statt einem Byte nun zwei Bytes zuordnet.

Das folgende Beispiel zeigt die einfache Ausgabe von vorzeichenbehafteten Ganzzahltypen mit `printf()`:

```
00 // Kapitel3/printf_beispiel.c
01 #include <stdio.h>

02 int main(void) {
03   signed char cVal = 100;
04   short sVal = 10000;
05   int iVal = 123456;
06   long lVal = 123456;
07   long long llVal = 123456;

08   printf("%hhd\n", cVal);
09   printf("%hd\n", sVal);
10   printf("%d\n", iVal);
11   printf("%ld\n", lVal);
12   printf("%lld\n", llVal);
13   return 0;
14 }
```

3.4.1 Vorzeichenlos und vorzeichenbehaftet

Für jeden der eben vorgestellten `signed`-Ganzzahltypen steht ein entsprechender `unsigned`-Ganzzahltyp (engl. *standard unsigned integer type*) zur Verfügung, der die Zahlen vorzeichenlos betrachtet. Wenn Sie beispielsweise eine Integer-Variable vereinbaren, ist diese (wenn auch implementationsabhängig) meistens vorzeichenbehaftet. Nehmen wir also an, Sie vereinbaren die folgende Variable:

```
int var;
```

Hier beträgt der Wertebereich von `int` abhängig von der Implementierung (siehe `INT_MIN` und `INT_MAX`) beispielsweise –2.147.483.648 bis +2.147.483.647. Mit dem Schlüsselwort `unsigned` können Sie jetzt eine ganzzahlige Variable ohne Vorzeichen vereinbaren. Dies sähe dann beispielsweise so aus:

```
unsigned int var;
```

In diesem Fall können Sie jedoch keine negativen Werte mehr speichern. Dafür wird der positive Wertebereich von `int` (abhängig von der Implementierung von `UINT_MAX` in der Header-Datei *limits.h*) natürlich größer.

Größer ist nicht gleich größer

Damit Sie das jetzt nicht falsch verstehen: Der Datentyp `int` bleibt natürlich weiterhin gleich breit. Mit `unsigned` verschiebt sich lediglich der Wertebereich (abhängig von der Implementierung) von beispielsweise –2.147.483.647 bis +2.147.483.647 auf mindestens 0 bis 4.294.967.295 (siehe Abbildung 3.1). Man kann auch sagen, dass es in diesem Fall kein Vorzeichenbit (dies ist oft das oberste Bit) mehr gibt und dass sich dadurch der Wertebereich verdoppelt.

Denken Sie aber daran, dass z. B. die Zahl -1 nicht einfach durch den Hex-Wert 0x80000001 dargestellt wird (+1 mit gesetztem Vorzeichenbit), wie ein Leser einmal anmerkte, da sein `printf()` stets falsche Werte ausgab. Die Zahl -1 entsteht dadurch, dass von dem Hex-Wert 0x00000000 der Wert 1 abgezogen wird, und dies ergibt dann für -1 den Hex-Wert 0xffffffff, bei dem dann auch das Vorzeichenbit gesetzt ist. Im weiteren Verlauf werden Sie noch viel mehr über die Hexadezimal-Schreibweise und die verschiedenen Zahlenformate erfahren.

Gleiches wie für `int` gilt auch für die Datentypen `short`, `long` und `long long`. Bei dem Datentyp `char` ist es etwas komplizierter. `char` kann entweder `signed char` oder `unsigned char` sein. Der Datentyp `char` wird gesondert in Abschnitt 3.5.1 behandelt.

Mit `signed` gibt es auch ein Schlüsselwort, um eine Variable explizit als vorzeichenbehaftet zu vereinbaren. Allerdings entspricht die Schreibweise von

```
signed int var;
```

der von

```
int var;
```

Somit ist die Verwendung des Schlüsselwortes `signed` überflüssig (außer bei `char`), weil ganzzahlige Datentypen ohne Verwendung von `unsigned` immer vorzeichenbehaftet sind. Sie können aber ein explizites `signed` dazu benutzen, Ihren Quellcode besser verständlich zu machen.

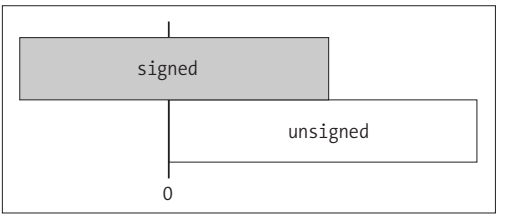

Abbildung 3.1 Mit »unsigned« ändert sich nicht die Bitbreite eines Datentyps, sondern es wird lediglich der Wertebereich verschoben.

Der Vollständigkeit halber finden Sie in Tabelle 3.2 einen Überblick über die `unsigned`-Gegenstücke bei den Ganzzahlen. Zusätzlich finden Sie den Datentyp `_Bool`, der ebenfalls zu den `unsigned`-Ganzzahltypen gehört. Die Wertebereiche in der Tabelle entsprechen auch hier wieder dem Mindestwert. Der tatsächliche Wert hängt von der Implementierung ab.

Datentyp	Wertebereich (mindestens)	Formatzeichen
`_Bool` (seit C99)	0 und 1	`%u`
`unsigned char`	0 bis 255	`%hhu` (für Dezimal) `%c` (für Zeichen)
`unsigned short`	0 bis 65.535	`%hu`

Tabelle 3.2 Grunddatentypen für vorzeichenlose Ganzzahlen

Datentyp	Wertebereich (mindestens)	Formatzeichen
`unsigned int`	0 bis 65.535	`%u`
`unsigned long`	0 bis 4.294.967.295	`%lu`
`unsigned long long` (seit C99)	0 bis 18.446.744.073.709.551.615	`%llu`

Tabelle 3.2 Grunddatentypen für vorzeichenlose Ganzzahlen (Forts.)

Überlauf bei »unsigned«-Ganzzahlwerten

Bei `unsigned`-Ganzzahlwerten kann es im Falle eines Überlaufs nicht zu einem undefinierten Verhalten kommen. Wenn hier der Wertebereich überschritten wird, wird mit einer sogenannten Modulo-Arithmetik weitergerechnet. Addieren Sie also zu einem maximalen `unsigned`-Ganzzahltyp z. B. 2 hinzu, wird mit 1 weitergerechnet:

```
unsigned int uVal = UINT_MAX; // UINT_MAX enthält max. Wert
uVal += 1; // = 0 (kein Überlauf)
```

Am Rande sei noch erwähnt, dass es auch hier – abhängig von der Implementation – erweiterte `unsigned`-Ganzzahltypen (engl. *extended unsigned integer types*) geben kann (zum Beispiel `__uint128`).

Ganzzahltypen mit fester Breite

Seit C99 werden diese Datentypen über die plattformunabhängige Header-Datei *stdint.h* um Ganzzahltypen wie etwa `int8_t`, `int16_t`, `int32_t`, `int64_t` usw. mit fester Länge ergänzt.

3.4.2 Suffixe für Ganzzahlen

Wenn Sie den Compiler informieren müssen, wie er ein bestimmtes Literal interpretieren oder von welchem Typ dieses Literal sein soll, dann können

Sie ein dem Typ entsprechendes Präfix oder Suffix zum Literal hinzufügen. Das Suffix `u` oder `U` deutet beispielsweise an, dass es sich um eine vorzeichenlose (`unsigned`) Zahl handelt. `l` bzw. `L` gibt an, dass es sich um eine `long`-Zahl handelt. Den Datentyp `long long` zeigen Sie mit `ll` bzw. `LL` an. Das können Sie auch mit `unsigned` kombinieren, indem Sie ein `ul` oder `UL` für `unsigned long` und `ull` oder `ULL` für `unsigned long long` verwenden. Verzichten Sie auf das Suffix, verwendet der Compiler `int`, sofern der Wert passt. Einige Beispiele wären:

```
unsigned int uVal = 1000u;
long lVal = 100000L;
unsigned long ulVal = 2222222UL;
long long llVal = 1234LL;
unsigned long long ullVal = 12341234323ULL;
unsigned int uHexVal = 0X42U;
```

3.5 Datentypen für Zeichen

In diesem Abschnitt erfahren Sie etwas zu den Zeichentypen in C. Zwar werden Sie in diesem Buch vorwiegend mit dem Datentyp `char` arbeiten, trotzdem sollen Sie auch kurz erfahren, welche Möglichkeiten es gibt, erweiterte bzw. umfangreichere Zeichensätze zu verwenden. Beachten Sie, dass die Verwendung von erweiterten Zeichensätzen (wie Unicode) jenseits von `char` kein triviales Thema mehr ist.

Für Sie ist es daher zunächst einmal wichtig, dass Sie sich mit dem Datentyp `char` vertraut machen. Wenn Sie anfangen, internationale Programme zu schreiben, dann werden Sie sich auch intensiver mit dem Thema »Unicode« befassen müssen, aber dafür benötigen Sie auf jeden Fall den aktuellen Unicode-Standard und separate Literatur. Wir nehmen nun erst einmal an, dass ein Zeichen von Typ `char` ein Byte umfasst. Wir wissen, dass dies eine sehr vereinfachte Sichtweise ist, und einige Leser und Leserinnen haben uns auch zu Recht geschrieben, dass sie stets die folgende korrektere Typendefinition benutzen

```
typedef unsigned char byte;
byte MyMem[0x10000];
```

und nicht

```
unsigned char MyMem[0x10000];
```

Dies ist korrekt, und Sie werden später auch noch erkennen warum das so ist, wenn Sie mehr über Typendefinitionen mit `typedef` gelernt haben. Zunächst verwenden wir jedoch die primitive Betrachtungsweise.

3.5.1 Der Datentyp char

Der grundlegende Datentyp für Zeichen lautet `char` und belegt gewöhnlich ein Byte Speicherplatz, was somit meistens (aber nicht immer) $2^8 = 256$ Ausprägungen ermöglicht. Der Datentyp `char` ist zumindest groß genug, dass alle Zeichen des Basis-Ausführungszeichensatzes in ihm gespeichert werden können. Wird außerdem ein Zeichen in einem `char` gespeichert, dann ist es garantiert, dass der gespeicherte Wert der nichtnegativen Kodierung im Zeichensatz entspricht. Ein *Zeichensatz* wiederum ist dazu da, einem Zeichen einen bestimmten Wert zuzuweisen, weil ein Rechner letztendlich nur Dualzahlen speichern kann. Dazu ein Beispiel:

```
char ch = 'A';  // Dezimal 65 im ASCII-Zeichensatz
```

Bei dem weit verbreiteten ASCII-Zeichensatz entspricht die Zeichenkonstante `'A'` dem Dezimalwert 65. So wäre es beispielsweise auch möglich, stattdessen Folgendes anzugeben:

```
char ch = 65;  // Entspricht dem Zeichen 'A' (ASCII)
```

Das ist ohne Probleme möglich, weil `char` ja auch ein Integertyp ist. In der Praxis ist die Verwendung eines Dezimalwerts anstelle einer Zeichenkonstante allerdings nicht zu empfehlen, wenn Sie `char` als Zeichentyp und nicht als Integer-Typ verwenden wollen: Zum einen lässt sich nicht sofort erkennen, welches Zeichen hier dahintersteckt, und zum anderen schreibt der Standard nicht vor, welcher Zeichensatz verwendet werden soll. Zwar dürfte zu 99,9 % die ASCII-Zeichensatztabelle zum Einsatz kommen, aber wenn dann doch einmal ein anderer Zeichensatz verwendet wird, sind Sie mit der Zeichenkonstante immer auf der sicheren Seite. Hierzu folgt ein einfaches Beispiel:

```
00  // Kapitel3/char_beispiel.c
01  #include <stdio.h>

02  int main(void) {
03    char ch01 = 'A'; // bei ASCII 65
04    char ch02 = 66;  // besser wäre 'B'
05    printf("Dezimal: %d; Zeichen: %c\n", ch01, ch01);
06    printf("Dezimal: %d; Zeichen: %c\n", ch02, ch02);
07    return 0;
08  }
```

Das Programm gibt bei der Ausführung Folgendes aus:

```
Dezimal: 65; Zeichen: A
Dezimal: 66; Zeichen: B
```

`char` ist auch ein Ganzzahldatentyp, mit dem Sie rechnen können. Aufgrund des kleineren Wertebereichs wird dieser Typ dazu jedoch relativ selten genutzt. Sie werden allerdings noch sehen, dass es hier Ausnahmen gibt, z. B. wenn es darum geht, mit beliebig langen Zahlen zu rechnen. Viele moderne Kryptografie-Verfahren tun genau dies: Sie betrachten z. B. einen zu verschlüsselnden Text (oder auch den Schlüssel selbst) als sehr lange Zahl und führen mit dieser Zahl komplexe Operationen aus.

Vorzeichen bei »char«

Es hängt von der Compiler-Einstellung ab, ob `char` auch negative Zahlen aufnehmen kann – ob `char` also als `signed char` oder `unsigned char` implementiert ist. Dies ist unter anderem dann wichtig zu wissen, wenn Sie `char` als Typ für Ganzzahlen verwenden wollen.

3.5.2 Der Datentyp wchar_t

Für die Zeichensätze mancher Sprachen – wie etwa der chinesischen mit über tausend Zeichen – ist der Datentyp `char` zu klein. Für die Darstellung beliebiger landesspezifischer Zeichensätze kann daher der Breitzeichen-

typ wchar_t (*wide char* = breite Zeichen) aus der Header-Datei *stddef.h* verwendet werden. Bei der Verwendung eines solchen Zeichens muss vor die einzelnen Hochkommata noch das Präfix L gestellt werden:

```
wchar_t ch = L'Z';
```

Entsprechend wird auch bei dem Formatzeichen für die Ausgabe oder Eingabe eines wchar_t ein l vor das c gesetzt (%lc):

```
wprintf("%lc", ch);
```

Auch die üblichen Funktionen, die mit Zeichenketten arbeiten, können Sie mit wchar_t nicht mehr verwenden, und Sie müssen stattdessen auf die entsprechenden w-Versionen (wie beispielsweise wprintf()) zurückgreifen.

Die Größe von wchar_t lässt sich nicht exakt beschreiben, meistens beträgt sie jedoch 2 oder 4 Bytes. Es lässt sich lediglich mit Sicherheit sagen, dass wchar_t mindestens so groß wie char und höchstens so groß wie long ist. wchar_t muss auf jeden Fall mindestens groß genug sein, um alle Werte des größten unterstützten Zeichensatzes aufnehmen zu können.

Zeichen und Zeichensatz

Egal welchen Zeichentyp Sie verwenden, Sie sollten sich immer vor Augen halten, dass char und wchar_t selbst keine Zeichen speichern, sondern letztendlich nur Ganzzahlen, die ihre Bedeutung erst durch den Zeichensatz erhalten, der sich auf dem Rechner befindet. Dies gilt selbstverständlich auch für Ihren Quellcode und die in ihm enthaltenen Kommentare, die auf fremden Rechnern unter Umständen nicht lesbar sind. (Das betrifft z. B. die deutschen Umlaute oder französische Akzentzeichen.)

3.5.3 Unicode-Unterstützung

Der Unicode-Standard definiert mit UTF-8, UTF-16 und UTF-32 drei Zeichenkodierungsformate. Jedes Format hat seine Vor- und Nachteile. Bisher haben Programmierer char verwendet, um UTF-8 zu nutzen, unsigned

short oder wchar_t für UTF-16 und unsigned long oder wchar_t für UTF-32. Ab dem C11-Standard können Sie zwei Typen mit einer plattformunabhängigen Breite mit char16_t und char32_t für UTF-16 und UTF-32 aus der Header-Datei *uchar.h* nutzen.

```
#include <uchar.h>

char utf8ch = u8'Z';        // UTF-8
char16_t utf16ch = u'Z';    // UTF-16
char32_t utf32ch = U'Z';    // UTF-32
```

Für UTF-8 können Sie nach wie vor char verwenden. C11 hat außerdem die Präfixe u und U für UTF-16- bzw. UTF-32-Literale und das Präfix u8 für UTF-8-Literale eingeführt. Auch Unicode-Konvertierungsfunktionen sind in *uchar.h* deklariert.

3.6 Datentypen für Fließkommazahlen

Fließkommaliterale sind Werte mit einer Nachkommastelle und enthalten ein Dezimaltrennzeichen in Form eines Punktes (beispielsweise 3.1415, .25, 33. usw.). Auch eine wissenschaftliche Schreibweise des Exponenten als Zehnerpotenz ist möglich (22.44e–3 beispielsweise entspricht 0.02244). Im Standard finden Sie die Fließkommatypen aus Tabelle 3.3:

Datentyp	Wertebereich	Formatzeichen	Genauigkeit
float	1.2E–38 3.4E+38	%f	einfach
double	2.3E–308 1.7E+308	%f (%lf für scanf())	doppelt
long double	3.4E–4932 1.1E+4932	%Lf	zusätzlich

Tabelle 3.3 Datentypen für Fließkommazahlen

Beachten Sie, dass die Genauigkeit und die Wertebereiche dieser Typen komplett implementierungsabhängig sind. Es ist lediglich gewährleistet, dass bei `float`, `double` und `long double` (hier von links nach rechts) jeder Typ den Wert des vorherigen aufnehmen kann. Auf den meisten Architekturen entsprechen `float` und `double` den IEEE-Datentypen. Die Norm IEEE 754 definiert dabei die Darstellungen für binäre Gleitkommazahlen im Computer.

Bevorzugter Fließkommatyp

In der Praxis empfiehlt es sich, immer den Fließkommatyp `double` zu verwenden, weil der Compiler den Typ `float` intern häufig ohnehin in den Typ `double` umwandelt.

Im Gegensatz zu den Ganzzahlen gibt es bei den Fließkommazahlen keine Unterschiede zwischen vorzeichenbehafteten und vorzeichenlosen Zahlen. In C++, einer Variante von Standard-C, sind alle Fließkommazahlen vorzeichenbehaftet.

Beachten Sie außerdem, dass die Trennzeichen bei den Fließkommazahlen in US-amerikanischer Schreibweise angegeben werden. Anstatt eines Kommas zwischen den Zahlen müssen Sie also einen Punkt setzen (man spricht im Englischen von *floating* **point** *variables*):

```
double pi = 3,14159265;   // FALSCH
double pi = 3.14159265;   // RICHTIG
```

Wenn einer der Werte vor oder nach dem Dezimalpunkt 0 ist, beispielsweise 0.5 oder 1.0, können Sie die Ziffer 0 auch weglassen. Der Compiler ergänzt die 0 automatisch.

```
double c = .5;  // entspricht 0.5
double d = 5.;  // entspricht 5.0
```


»float_t« und »double_t«

Erwähnt werden sollten an dieser Stelle noch die Gleitkommatypen `float_t` und `double_t` aus der Header-Datei *math.h*, die seit C99 vorhan-

den sind. Mit ihnen wird bei arithmetischen Operationen intern gerechnet, und durch sie entfallen die zuvor notwendigen Konvertierungen. Welcher Typ dabei verwendet wird, hängt vom Wert des Makros `FLT_EVAL_METHOD` ab.

3.6.1 Suffixe für Fließkommazahlen

Wie den Ganzzahlen können Sie den Fließkommazahlen ebenfalls ein Suffix hinzufügen. Mit dem Suffix `f` oder `F` kennzeichnen Sie eine Fließkommazahl mit einer einfachen Genauigkeit (`float`). Das Suffix `l` oder `L` hingegen deutet auf eine Fließkommazahl mit erhöhter Genauigkeit hin (`long double`). Wenn Sie bei Fließkommazahlen keine Angabe machen, wird der Typ `double` verwendet.

Die Verwendung des Exponenten, mit dem die Größe der Zahl als Zehnerpotenz angegeben wird, kann mit `e` oder `E` (beide Buchstaben haben dieselbe Bedeutung), gefolgt von einem optionalen `+` oder `-` und einer Ziffernsequenz, notiert werden (22.45e0 entspricht 22.45, 22.45e1 entspricht 224.5, und 22.45e–3 ist 0.02245). Hierzu einige Beispiele:

```
float fVal = 3.33f;
long double ldVal = 32.32L;
double eVal1 = 4.3e10;
double eVal2 = 3.4e-5;
long double eVal3 = 8.4e123L;
```

3.6.2 Komplexe Gleitkommatypen

Seit dem C99-Standard werden auch komplexe Gleitkommatypen in der Header-Datei *complex.h* implementiert. Eine komplexe Zahl wird hierbei als Paar aus Real- und Imaginärteil dargestellt, die auch mit den Funktionen `creal()` und `cimag()` ausgegeben werden können. Beide Teile der komplexen Zahl bestehen entweder aus den Typen `float`, `double` oder `long double`. Daher gibt es wie bei den reellen Gleitpunktzahlen die folgenden drei komplexen Gleitkommatypen:

```
float _Complex
double _Complex
long double _Complex
```

Um die umständliche Schreibweise mit dem Unterstrich `_Complex` zu vermeiden, ist in der Header-Datei *complex.h* das Makro `complex` definiert. Anstelle des Schlüsselwortes `_Complex` können Sie auch `complex` verwenden:

```
float complex        // gleich wie float _Complex
double complex       // gleich wie double _Complex
long double complex  // gleich wie long double _Complex
```

Da komplexe Zahlen einen Real- und einen Imaginärteil haben, beträgt die Größe des Datentyps in der Regel das Doppelte der Größe der grundlegenden Datentypen. Ein `float _Complex` benötigt somit 8 Bytes, weil im Grunde genommen zwei `float`-Elemente benötigt werden. Folgendes Listing soll das verdeutlichen:

```
00 // Kapitel3/komplexe_zahlen.c
01 #include <stdio.h>
02 #include <complex.h>

03 int main(void) {
04   float complex fc = 2.0 + 3.0*I;
05   // 4 Bytes
06   printf("sizeof(float) : %zu\n", sizeof(float));
07   // 8 Bytes (realer und imaginärer Teil)
08   printf("sizeof(float complex) : %zu\n",
                    sizeof(float complex));
09   // Ausgabe von Real- und Imaginärteil
10   printf("%f + %f\n", creal(fc), cimag(fc));
11   return 0;
12 }
```

Listing 3.1 »komplexe_zahlen.c« zeigt den Umgang mit komplexen Zahlen.

Des Weiteren ist in der Header-Datei das Makro `I` definiert, das die imaginäre Einheit mit dem Typ `const float complex` darstellt. Vielleicht ist hier

eine kurze Hintergrundinformation zu komplexen Gleitpunktzahlen nötig: Eine komplexe Zahl `zVal` wird beispielsweise folgendermaßen in einem kartesischen Koordinatensystem dargestellt:

```
zVal = xVal + yVal * I
```

`xVal` und `yVal` sind dabei reelle Zahlen, und `I` ist der imaginäre Teil. Die Zahl `xVal` wird hier als realer Teil betrachtet, und `yVal` ist der imaginäre Teil von `zVal`.

Ebenfalls in C99 eingeführt wurden Gleitkomma-Datentypen für rein imaginäre Zahlen mit `float _Imaginary`, `double _Imaginary` und `long double _Imaginary`.

Falls Sie noch nicht das nötige mathematische Hintergrundwissen zu komplexen Zahlen haben, macht dies an dieser Stelle nichts. Aber ohne die Erwähnung von *complex.h* wäre dieses Buch schlicht unvollständig.

3.7 Boolescher Datentyp

Im C99-Standard wurde mit `_Bool` ein boolescher Wahrheitswert eingeführt, der auf jeden Fall groß genug ist, um die Werte 0 (= falsch) und 1 (= wahr) aufzunehmen. Der Datentyp `_Bool` gehört zur Gruppe der `unsigned`-Ganzzahltypen.

Glücklicherweise existiert für den Typ `_Bool` (ebenfalls seit C99) in der Header-Datei *stdbool.h* das Makro `bool`, sodass Sie den Bezeichner `bool` wie in C++ verwenden können. Allerdings müssen Sie hierfür die Header-Datei *stdbool.h* inkludieren. Manche fremden Programme – vor allem Downloads aus dem Internet – enthalten auch folgende Definition, anstatt die entsprechende Header-Datei einzufügen:

```
typedef enum {false,true} bool;
```

Ebenfalls häufig im Internet anzutreffen ist folgende, leider falsche Definition:

```
typedef enum {true,false} bool;
```

Bei der letzten Definition bilden Sie `false` auf 1 und `true` auf 0 ab, was in C zu schwer auffindbaren Fehlern führt. Hier verwechselt so mancher Programmierer bei der letzten (falschen) Definition C mit dem sehr viel älteren BASIC, wo in der Tat -1 (alle Bits sind gesetzt) als wahr und 0 (kein Bit ist gesetzt) als falsch angesehen wird. Gehen Sie daher auf Nummer sicher und verwenden Sie *stdbool.h*!

Der neue C23-Standard zählt allerdings mittlerweile sämtliche Variablen vom booleschen Typ zu den primitiven Datentypen und sieht die Datei *stdbool.h* nur noch als Option an. Wundern Sie sich also bitte nicht, wenn es in Zukunft immer mehr Compiler gibt, die *stdbool.h* entweder ignorieren oder sogar eine Meldung ausgeben, dass es die entsprechende Datei nicht gibt. Keine Panik: Ihre Programme laufen dann trotzdem noch, auch ohne *stdbool.h*.

Diese unbekannten oder ungewohnten Definitionen in fremden Programmen sind sehr verbreitet, vor allem auf Systemen mit wenig Speicher. Boolesche Werte sind Elemente einer »booleschen Algebra«, die einen von zwei möglichen Werten annehmen können. Dieses Wertepaar hängt von der Anwendung ab und lautet entweder `wahr/falsch`, `true/false` oder eben `ungleich 0/gleich 0`. In C kann hierfür das Wertepaar `true` (für wahr = 1) und `false` (für falsch = 0) verwendet werden, das in der Header-Datei *stdbool.h* definiert ist.

Sie können auch das Paar `ungleich 0` (für wahr) und `gleich 0` (für falsch) als Dezimalwert verwenden. Hierzu ein kleines Beispiel:

```
#include <stdbool.h>
// ...
// Schalter auf wahr setzen
_Bool b1 = 1;
// Schalter auf unwahr setzen
_Bool b2 = 0;
// Benötigt vor C23 <stdbool.h>
bool b3 = true; // wahr
// Benötigt vor C23 <stdbool.h>
bool b4 = false; // unwahr
```

Um hier kein Durcheinander zu verursachen, muss noch erwähnt werden, dass bereits der C-Standard seit C99 den Typ `_Bool` als echten Datentyp implementiert. Das Makro `bool` und das Wertepaar `true` bzw. `false` (kleingeschriebene Variante) können Sie aber nur verwenden, wenn Sie die Header-Datei *stdbool.h* inkludieren. Es ist auch hier leider nicht abzusehen, wie nach C23 in Zukunft mit *stdbool.h* verfahren wird und wie die Compiler-Hersteller dies sehen. Unter Linux benötigen Sie auf jeden Fall *stdbool.h*, weil dort zusätzlich zu C23 noch ein anderer Standard wichtig ist, nämlich der POSIX-Standard.

3.8 Speicherbedarf mit sizeof ermitteln

Wenn Sie die Größe eines Typs benötigen, müssen Sie den `sizeof`-Operator verwenden. Dieser gibt in der Regel die Größe des Operanden in Bytes zurück und wird beispielsweise bei der dynamischen Speicherreservierung verwendet oder dann, wenn Sie Programme schreiben, die auf andere Plattformen portierbar sind. Als Rückgabetyp von `sizeof` ist `size_t` definiert. `size_t` ist ein implementierungsabhängiger `unsigned`-Ganzzahlentyp und in der Header-Datei *stddef.h* (und anderen Header-Dateien) definiert. Wie viel Speicherplatz ein Variablentyp letztlich benötigt, hängt von seiner Implementierung ab. Die Formatanweisung für `size_t` lautet `%zu`. Beachten Sie, dass `sizeof` ein echter Operator ist und dass der entsprechende Größenwert auch nicht durch eine Funktion ermittelt wird.

Listing 3.2 enthält ein einfaches Beispiel, in dem der `sizeof`-Operator verwendet wird:

```
00  // Kapitel3/sizeof_beispiel.c
01  #include <stdio.h>

02  int main(void) {
03    int ival = 0;
04    double dval = 0;
05    printf("sizeof(ival)  : %zu\n", sizeof(ival));
06    printf("sizeof(dval)  : %zu\n", sizeof(dval));
07    // So geht es auch
08    printf("sizeof(float) : %zu\n", sizeof(float));
```

```
09    size_t sz = sizeof(char);
10    printf("sizeof(char)  : %zu\n", sz);
11    return 0;
12  }
```

Listing 3.2 Das Listing demonstriert die Verwendung des »sizeof«-Operators mit Variablen unterschiedlichen Typs.

Das Programm sieht bei der Ausführung auf unserem System folgendermaßen aus:

```
sizeof(ival)  : 4
sizeof(dval)  : 8
sizeof(float) : 4
sizeof(char)  : 1
```

Speicherausrichtung eines Operanden ermitteln

Seit C11 gibt es den Operator `_Alignof`, mit dem Sie die Speicherausrichtung des Operanden ermitteln können. Dieser Operator ist auch als Makro mit `alignof` vorhanden, damit Sie ihn etwa mit `alignof(long double)` so komfortabel wie schon den `sizeof`-Operator verwenden können. Wie der `sizeof`-Operator liefert auch `alignof` die Speicherausrichtung des Operanden vom Typ `size_t` zurück.

Die *Speicherausrichtung* (das sogenannte *Alignment*) ist die Ausrichtung von Speicherobjekten an den Wortgrenzen des Prozessors. Die Breite eines *Datenwortes* ist meistens die Anzahl der Bits, die maximal in ein Prozessorregister passen. Wenn Sie also einen modernen 64-Bit-Prozessor haben, fangen sämtliche Speicherobjekte an Adressen an, die durch 8 teilbar sind. Hierdurch wird zwar Platz vergeudet, aber die Ausführungsgeschwindigkeit enorm erhöht.

Bei den meisten Compilern können Sie das Alignment auch abschalten und erhalten dadurch kompakteren Code. Allerdings verlangen manche Prozessoren, wie z. B. der im Amiga verbaute 68000-Prozessor, dass sämtliche Speicherobjekte an geraden Adressen anfangen. ARM-Prozessoren (z. B. beim Raspberry Pi) verlangen sogar, dass sämtliche Speicheradressen von Speicherobjekten durch 4 teilbar sind.

Im Endeffekt bedeutet dies: Lassen Sie die Finger von den Alignment-Einstellungen! So ersparen Sie sich böse Systemabstürze, für die sich einfach kein Grund finden lässt.

3.9 Die Wertebereiche der Datentypen ermitteln

Der C-Standard selbst legt den Wertbereich und die Größe der einzelnen Datentypen nicht fest, sondern schreibt nur die Relation zwischen den Größen der Datentypen vor. Für jeden Basisdatentyp werden lediglich Mindestgrößen gefordert. Dadurch ergeben sich für den Compiler verschiedene Gestaltungsmöglichkeiten und auch Kommandozeilen-Argumente (z. B. für den CGG unter Linux), die aber für einen Einsteiger / eine Einsteigerin zu umfangreich sind. Im Ernstfall sollten Sie immer zuerst z. B. mit `man gcc` die verschiedenen Optionen studieren.

Eine Möglichkeit unter vielen ist, dass der Datentyp `int` so festgelegt wird, dass seine Größe der Datenwortgröße des Prozessors entspricht (wie wir bereits gesagt haben, muss dies bei manchen Prozessoren sogar so festgelegt werden). Das handhaben aber nicht alle Compiler so, und es gibt auch andere Schemata. Die Größe des Zeigertyps richtet sich wiederum häufig nach der Größe des Speicherbereichs, der vom Programm aus erreichbar (adressierbar) sein muss. Es ist daher durchaus möglich, dass der Speicherbereich kleiner oder größer ist, als die Prozessorarchitektur es zulässt. Auf modernen Betriebssystemen ist die Speicherverwaltung sowieso eine Sache für sich und derart komplex, dass man mindestens ein weiteres Buch benötigt, um diese zu erklären. Eines dieser Bücher ist z. B. »Moderne Betriebssysteme« von Andrew S. Tanenbaum. Aber Vorsicht: Sie sollten erst Experte bzw. Expertin werden, ehe Sie sich dieses Buch zulegen, sonst verstehen Sie keine einzige Zeile. Wenn Sie allerdings gerade mit dem Informatikstudium beginnen, dann sollten Sie sich das Buch von Tanenbaum auf jeden Fall anschaffen – Sie werden es früher oder später benötigen.

Wenn man jetzt davon ausgeht, dass ein Byte 8 Bit groß ist, was auf vielen Architekturen der Fall ist, dann sind die anderen Datentypengrößen alle ein Vielfaches von 8 Bit. Deshalb ergeben sich verschiedene sogenannte

Datenmodelle (auch *Programmiermodelle* genannt). Wir wollen an dieser Stelle nicht näher auf die verschiedenen Datenmodelle eingehen. Es geht uns vielmehr darum, dass Sie verstehen, warum die Datentypen auf unterschiedlichen Systemen unterschiedliche Wertebereiche haben können.

In Tabelle 3.4 finden Sie eine Übersicht über die gängigen Datenmodelle.

Modell	char	short	int	long	long long	void*
IP16	8	16	16	32	64	16
LP32	8	16	16	32	64	32
ILP32	8	16	32	32	64	32
LLP64	8	16	32	32	64	64
LP64	8	16	32	64	64	64
ILP64	8	16	64	64	64	64
SILP64	8	64	64	64	64	64

Tabelle 3.4 Bits von Datentypen verschiedener Datenmodelle

Wenn Sie wissen wollen, welche implementierungsabhängigen Wertebereiche die einzelnen Datentypen auf dem auszuführenden System haben, finden Sie die vom Compiler-Hersteller vergebenen Größen in der Header-Datei *limits.h* für Integer-Typen und in *float.h* für Gleitkommatypen. Benötigen Sie hingegen Integer-Typen mit einer festen Größe, dann bietet Ihnen der Standard entsprechende Typen wie `int8_t`, `int16_t` usw. an, die in der Header-Datei *stdint.h* definiert sind.

3.9.1 Limits von Integer-Typen

Möchten Sie erfahren, welchen Wertebereich `int` oder die anderen Ganzzahldatentypen auf Ihrem System haben, finden Sie in der Header-Datei *limits.h* entsprechende Konstanten dafür. Für den Datentyp `int` beispielsweise finden Sie die Konstanten `INT_MIN` für den minimalen und `INT_MAX` für den maximalen Wert.

Um diese Werte zu ermitteln, müssen Sie selbstverständlich auch den Header *limits.h* in Ihr Programm inkludieren. *Inkludieren* ist das C-Fachwort für »eine Header-Datei mit `#include` einfügen« und kommt aus dem Lateinischen (*includere* = dt. *einfügen*). Das folgende Listing gibt Ihnen den tatsächlichen Wertebereich für die Datentypen `char`, `short`, `int`, `long` und `long long` auf Ihrem System aus:

```
// Kapitel3/limits.c
#include <stdio.h>
#include <limits.h>

int main(void) {
  printf("min. char-Wert     : %d\n", SCHAR_MIN);
  printf("max. char-Wert     : +%d\n", SCHAR_MAX);
  printf("min. short-Wert    : %d\n", SHRT_MIN);
  printf("max. short-Wert    : +%d\n", SHRT_MAX);
  printf("min. int-Wert      : %d\n", INT_MIN);
  printf("max. int-Wert      : +%d\n", INT_MAX);
  printf("min. long-Wert     : %ld\n", LONG_MIN);
  printf("max. long-Wert     : +%ld\n", LONG_MAX);
  printf("min. long long-Wert: %lld\n", LLONG_MIN);
  printf("max. long long-Wert: +%lld\n", LLONG_MAX);
  return 0;
}
```

Listing 3.3 Das Listing zeigt, dass Standardtypen auf unterschiedlichen Systemen unterschiedlich groß sein können.

Die Ausgabe des Programms hängt natürlich von der Implementierung ab. Bei uns sieht sie unter Windows 11 (AMD 64) und Pelles C wie folgt aus:

```
min. char-Wert     : -128
max. char-Wert     : +127
min. short-Wert    : -32768
max. short-Wert    : +32767
min. int-Wert      : -2147483648
max. int-Wert      : +2147483647
min. long-Wert     : -2147483648
```

```
max. long-Wert     : +2147483647
min. long long-Wert: -9223372036854775808
max. long long-Wert: +9223372036854775807
```

Einen Überblick über alle Konstanten in der Header-Datei *limits.h* und deren Bedeutungen finden Sie auf den entsprechenden Manpages, in der Online-Hilfe des Compilers oder im Web unter *http://en.cppreference.com/w/c/types/limits*. Und vergessen Sie auch hier ist nicht das Committee Draft des C11-Standard unter *http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf*. Dasselbe gilt auch für die gleich folgenden Limits von Gleitkommazahlen.

3.9.2 Limits von Fließkommazahlen

Auch bei Fließkommazahlen gibt es eine Header-Datei mit Konstanten, in der Sie die Wertebereiche (Limits) ermitteln können. Alle implementierungsabhängigen Wertebereiche für Fließkommazahlen sind in der Header-Datei *float.h* deklariert. Zur Demonstration zeigen wir Ihnen nachfolgend ein einfaches Listing. Dieses ermittelt die Genauigkeit der Dezimalziffern aus den Konstanten `FLT_DIG` (für `float`), `DBL_DIG` (für `double`) und `LDBL_DIG` (für `long double`), die im Header *float.h* deklariert sind:

```
00  // Kapitel3/float_digits.c
01  #include <stdio.h>
02  #include <float.h>

03  int main(void) {
04    printf("float Genauigkeit      : %d\n", FLT_DIG);
05    printf("double Genauigkeit     : %d\n", DBL_DIG);
06    printf("long double Genauigkeit: %d\n", LDBL_DIG);
07    return 0;
08  }
```

Auf unserem PC sieht die Ausgabe des Programms so aus:

```
float Genauigkeit      : 6
double Genauigkeit     : 15
long double Genauigkeit: 15
```

3.9.3 Integer-Typen mit fester Größe verwenden

Wenn Sie sich nicht auf die implementierungsabhängigen Größen der Basis-Integer-Typen auf den verschiedenen Systemen verlassen wollen oder können bzw. einen Integer-Typ mit einer *vorgegebenen Breite* benötigen, finden Sie seit C99 entsprechende Typen in der Header-Datei *stdint.h* definiert. Mit »vorgegebener Breite« ist die Anzahl der Bits zur Darstellung des Wertes gemeint. Die speziellen Formatierungsspezifizierer für die `printf`- und `scanf`-Familien hingegen finden Sie in der Header-Datei *inttypes.h*. Die einzelnen Typen können Sie hierbei in folgende Gruppen aufteilen (*N* steht für die Anzahl von Bits, und Typen mit `u` (`unsigned`) sind vorzeichenlos):

- **`intN_t`** und **`uintN_t`**: ein Integer-Typ mit einer Breite von exakt *N* Bits wie beispielsweise `int64_t` bzw. `uint64_t` für einen Integer-Typ mit 64 Bit Breite. Entsprechend den Typen finden Sie in der Header-Datei *stdint.h* auch die zugehörigen Limits für die minimalen und maximalen Werte mit `INTN_MIN` und `INTN_MAX` bzw. `UINTN_MAX`.
- **`int_leastN_t`** und **`uint_leastN_t`**: ein Integer-Typ mit einer Breite von mindestens *N* Bits. Er ist damit garantiert der kleinste Typ der Implementation. Auch dazu finden Sie mit `INT_LEASTN_MIN` und `INT_LEASTN_MAX` bzw. `UINT_LEASTN_MAX` entsprechende Limits für die minimalen bzw. maximalen Werte. *N* kann hierbei 8, 16, 32 oder 64 sein.
- **`int_fastN_t`** und **`uint_fastN_t`**: ein schneller Integer-Typ mit einer Breite von mindestens *N* Bits. Dieser Typ ist garantiert der schnellste Integer-Typ in der Implementation. Auch hierzu finden Sie mit `INT_FASTN_MIN` und `INT_FASTN_MAX` bzw. `UINT_FASTN_MAX` entsprechende Limits für die minimalen bzw. maximalen Werte. *N* kann hierbei 8, 16, 32 oder 64 sein.
- **`intmax_t`** und **`uintmax_t`**: der garantiert größtmögliche Integer-Typ der Implementation. Den minimalen und maximalen Wert können Sie mit `INTMAX_MIN` und `INTMAX_MAX` bzw. `UINTMAX_MAX` ermitteln.

Wenn Sie die Header-Datei *stdint.h* eingebunden haben, können Sie diese Integer-Typen mit fester Bitbreite genauso einsetzen und verwenden wie die Integer-Typen ohne feste Größe:

```
00 // Kapitel3/fixed_ints.c
01 #include <stdio.h>
02 #include <stdint.h>

03 int main(void) {
04   int64_t bigVar = 12345678;
05   printf("bigVar          : %lld\n", bigVar);
06   printf("sizeof(int64_t) : %zu\n", sizeof(int64_t));
07   printf("INT64_MAX       : %lld\n", INT64_MAX);
08   return 0;
09 }
```

Das Beispiel gibt bei der Ausführung Folgendes aus:

```
bigVar          : 12345678
sizeof(int64_t) : 8
INT64_MAX       : 9223372036854775807
```

3.9.4 Sicherheit beim Kompilieren mit static_assert

Mit der C11-Funktion `static_assert()` überprüfen Sie einen konstanten Ausdruck zwischen den Klammern zur Übersetzungszeit. Trifft die Auswertung des Ausdrucks nicht zu, bricht der Compiler mit einer Fehlermeldung ab, die Sie ebenfalls mit angeben können. Auch hier wurde mit `static_assert()` ein komfortables Makro definiert, um nicht die Schreibweise mit dem vorangestellten Unterstrich verwenden zu müssen. Damit Sie diese Funktion verwenden können, müssen Sie die Header-Datei *assert.h* einbinden. Ein einfaches Beispiel hierzu ist:

```
// Kapitel3/static_assert_1.c
#include <assert.h>
…
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

wahr, wird der Quellcode weiter übersetzt. Ist der Ausdruck hingegen falsch, bricht der Compiler die Übersetzung ab und gibt die dahinter angegebene Fehlermeldung aus (hier: `Need 16 byte long double`).

Wollen Sie beispielsweise sichergehen, dass Ihr Programm *nicht* auf einem System übersetzt wird, auf dem `unsigned char` *mehr* als 8 Bit hat, können Sie dies mit `static_assert()` folgendermaßen umsetzen:

```
// Kapitel3/static_assert_2.c
#include <assert.h>  // für static_assert
#include <limits.h>  // für CHAR_BIT
…
static_assert( CHAR_BIT == 8,
               "unsigned char hat hier nicht 8 Bit!" );
```

Die Verwendung von `static_assert()` empfehlen wir besonders für größere Projekte. Sie kostet überhaupt keine Laufzeit der Anwendung, weil in diesem Fall die Sicherheitschecks nur vom Compiler benutzt werden. Lediglich die Übersetzungszeit nimmt logischerweise zu. Aber wir denken, dass man damit leben kann, weil hiermit auf so manche Sicherheitsüberprüfung während der Laufzeit des Programms verzichtet werden kann.

Wahrscheinlich werden Sie als Einsteiger bzw. Einsteigerin kaum mit `static_assert()` in Berührung kommen, und wenn Sie an dieser Stelle nicht alles genau verstanden haben, können wir Sie beruhigen: Sie benötigen `static_assert()` nicht zwingend, ein guter und vor allem übersichtlicher Programmierstil tut es auch.

3.10 Konstanten erstellen

Benötigen Sie einen unveränderbaren Wert, können Sie eine Konstante verwenden. Der Sinn und Zweck einer solchen Konstanten ist es, dass der Wert zur Laufzeit des Programms nicht mehr verändert werden kann. Das Gegenstück zu einer Konstanten ist eine Variable. Eine Konstante definieren Sie, indem Sie vor den eigentlichen Datentyp das Schlüsselwort `const` setzen:

```
const int cIvar = 365;      // Integer-Konstante
const char dquote = '"';    // Zeichenkonstante
const float pi = 3.141592;  // Fließkommakonstante
```

Wenn Sie nun aus Versehen versuchen, den Wert der Konstanten zu verändern, gibt der Compiler zur Übersetzungszeit den Fehler aus. Somit können `const`-Werte auf gewöhnlichem Wege der direkten Zuweisung nicht mehr verändert werden. Wenn Sie in dem folgenden Beispiel versuchen, den Wert von `cIvar` zu ändern, wird sich der Compiler mit einer Fehlermeldung bei Ihnen beschweren:

```
const int cIvar = 365;  // Konstante
cIvar = 364;            // Fehler, da const
```

In der Praxis werden Konstanten mit `const` recht häufig bei Variablen, Zeigern, Parametern von Funktionen usw. verwendet – eben immer, wenn ein Wert nicht mehr verändert werden soll.

Typ-Qualifikatoren

Bei dem Schlüsselwort `const` handelt es sich um einen Typ-Qualifikator (engl. *type qualifier*). Über solche Qualifikatoren kann ein Datentyp bei der Vereinbarung modifiziert werden. Neben `const` gibt es noch weitere Qualifikatoren wie `volatile`, `restrict` (seit C99) oder `_Atomic` (seit C11).

3.11 Lebensdauer und Sichtbarkeit von Variablen

Die *Lebensdauer* einer Variablen gilt immer bis zum Ende des Anweisungsblocks, in dem sie definiert wurde. Dazu ein einfaches Beispiel:

```
00  // Kapitel3/lebensdauer_1.c
...
01  {  // Anweisungsblock - Anfang
02     int ivar = 1234;
03     printf("%d\n", ivar);
```

```
04  }  // Anweisungblock - Ende
05  printf("%d\n", ivar);  // Fehler!!!
…
```

Im Beispiel oben wurde innerhalb des Anweisungsblocks in Zeile **(02)** eine Integer-Variable mit dem Bezeichner `ivar` definiert. Die Ausgabe in Zeile **(03)** wäre noch in Ordnung, aber bei der Ausgabe in Zeile **(05)** wird sich der Compiler mit einer Fehlermeldung beschweren, weil hier die Variable `ivar` nicht mehr gültig ist. Die Variable `ivar` ist nur innerhalb des Anweisungsblocks der Zeilen **(01)** bis **(04)** gültig. Danach existiert sie nicht mehr. Das Problem in diesem Beispiel könnten Sie beheben, indem Sie die Variable `ivar` außerhalb, also vor dem Anweisungsblock in Zeile **(01)** notieren:

```
00  // Kapitel3/lebensdauer_2.c
…
01  int ivar = 1234;
02  {  // Anweisungsblock - Anfang
03    printf("%d\n", ivar);  // = 1234
04  }  // Anweisungblock - Ende
05  printf("%d\n", ivar);    // = 1234
…
```

Wie in Zeile **(01)** vereinbart, ist die *Sichtbarkeit* der Variablen `ivar` jetzt sowohl in Zeile **(03)** als auch in Zeile **(05)** gegeben. Existiert nun allerdings im Anweisungsblock eine weitere Variable `ivar`, greift der innere Block auf die nächstliegende gleichnamige Variable im inneren Block zu. Ein Beispiel hierzu wäre:

```
00  // Kapitel3/lebensdauer_3.c
…
01  int ivar = 1234;
02  {  // Anweisungsblock - Anfang
03    int ivar = 4321;
04    printf("%d\n", ivar);  // = 4321
05  }  // Anweisungblock - Ende
06  printf("%d\n", ivar);    // = 1234
…
```

Durch die Vereinbarung einer weiteren gleichnamigen Variablen `ivar` in Zeile **(03)** wird innerhalb des Anweisungsblocks in den Zeilen **(02)** bis **(05)** die äußere Variable `ivar` aus Zeile **(01)** überdeckt. Zugegeben, das Beispiel ist kein guter Stil, und die meisten Compiler geben hier auch eine Warnmeldung aus, aber es zeigt sehr schön die Sichtbarkeit von Variablen.

3.12 void – ein unvollständiger Typ

Nicht erwähnt wurde bisher `void` – ein leerer Datentyp, der keine Werte aufnehmen kann. `void` wird auch als unvollständiger Typ bezeichnet, weshalb auch keine Variable von diesem Typ erzeugt werden kann (die einzige, aber nicht ganz unwichtige Ausnahme ist `void*`). Wird der Typ `void` bei einer Funktion als Rückgabewert verwendet, so gibt die Funktion keinen Wert zurück.

3.13 Kontrollfragen und Aufgaben

1. Welche grundlegenden Datentypen für Ganzzahlen gibt es?
2. Womit können Sie eine Integer-Variable ohne Vorzeichen vereinbaren?
3. Aufgrund verschiedener Datenmodelle ist die Breite von Datentypen implementierungsabhängig. Was können Sie tun, wenn Sie einen Ganzzahltyp mit fester Breite benötigen?
4. Nennen Sie den grundlegenden Datentyp, der für Zeichen verwendet wird.
5. Ermitteln Sie die Größe in Byte der Datentypen `int` und `long long` auf Ihrem System.
6. Womit können Sie die implementierungsabhängigen minimalen und maximalen Wertbereiche für Ganz- und Gleitkommazahlen ermitteln?
7. Was bewirkt das Schlüsselwort `const` vor einem Datentyp?

Im nächsten Kapitel geht es darum, wie Sie in C mit Variablen so rechnen können, wie Sie es aus der Mathematik gewohnt sind – inklusive gängiger Regeln, z. B. Punkt-vor-Strichrechnung und Klammern.

```
04     printf("  for  F\n\t\tu\n\t\tn\n");
05     return 0;
06  }
```

B.2 Antworten und Lösungen zu Kapitel 3

1. `signed char`, `short`, `int`, `long` und `long long`
2. Mit dem Schlüsselwort `unsigned` kann eine ganzzahlige Variable ohne Vorzeichen vereinbart werden.
3. Für Ganzzahltypen mit fester Bitbreite gibt es seit C99 verschiedene plattformunabhängige Typen, die in der Header-Datei *<stdint.h>* definiert sind.
4. Der grundlegende Datentyp für Zeichen lautet `char`.
5. Eine einfache Musterlösung mit dem `sizeof`-Operator sieht wie folgt aus:

```
00  // kap003/loesung001.c
01  #include <stdio.h>

02  int main(void) {
03    printf("int: %zu Bytes\n", sizeof(int));
04    printf("long long: %zu Bytes\n", sizeof(long long));
05    return 0;
06  }
```

6. Die implementierungsabhängigen Wertebereiche für Integer-Typen sind in der Header-Datei *<limits.h>* und die für Gleitkommazahlen in der Header-Datei *<float.h>* definiert.
7. Mit dem Qualifikator `const` können Sie nicht mehr veränderbare Variablen (Konstanten) definieren.