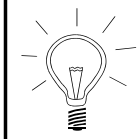
7 Datentypen und Variablen in C

Die Programmiersprache C verwendet eine sogenannt strikte **Typisierung**, was bedeutet, dass alle Variablen mit einen genau definierten Typ festgelegt werden müssen.

Ein **Datentyp** oder kurz **Typ** definiert, wieviele Bits für eine Variable im Speicher benötigt werden, und wie diese interpretiert werden müssen. Dadurch bestimmt der Typ, welche Werte eine Variable annehmen kann und welche nicht. Dies wird als **Wertebereich** bezeichnet.



Einen Datentyp in einen anderen umzuwandeln ist grundsätzlich nicht trivial. Die Sprache C hat jedoch viele automatische Umwandlungen – gerade für arithmetische Typen – eingebaut.

Die Programmiersprache C verfolgt auch heute noch keine strenge **Typumwandlung**, sondern erlaubt es, Werte eines Typs mehr oder weniger beliebig in Variablen eines anderen Typs zu speichern.

Wie großzügig in C eine solche Typumwandlung erfolgt, zeigt der folgende Programmausschnitt:

```
float  x = 3.9;
int    y = x;
```

Diese Zuweisung ist möglich und die Umwandlung von einem `float`-Typ in einen `int`-Typ erfolgt implizit durch festgelegte Regeln des Compilers.

Moderne Compiler haben jedoch häufig Warnungen eingebaut, um anzuzeigen, dass problematische Zuweisungen mit inkompatiblen Typen zu unerwünschten Effekten führen können. Obiges Beispiel würde ein Compiler mit eingeschalteten Warnungen in der zweiten Zeile anmerken mit dem Hinweis „Implicit conversion turns floating-point number into integer“. Solche Warnungen können mittels sogenannter „expliziter Casts“ entfernt werden. Siehe dazu Kapitel → 9.9.5.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-45209-4_7.

J. Goll und T. Stamm, *C als erste Programmiersprache*,
https://doi.org/10.1007/978-3-658-45209-4_7

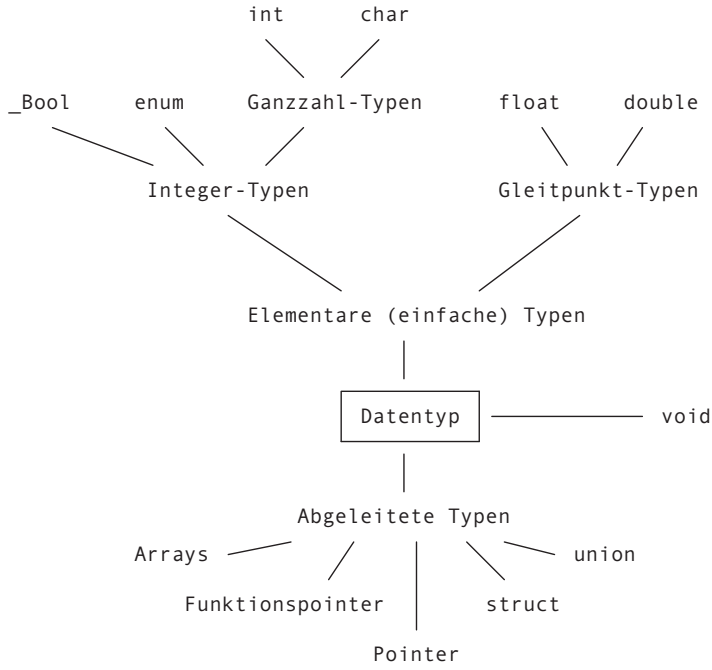
7.1 Typkategorien

In C kann grob zwischen elementaren und abgeleiteten Typen unterschieden werden.

Zu den **elementaren Typen** zählen die Integer-Typen, die Gleitpunkt-Typen sowie Typen, welche intern ebenfalls als Integer gespeichert werden wie Enumerationen und der Typ `_Bool`. Diese Typen werden in diesem Kapitel behandelt.

Abgeleitete Typen sind Typen, welche eine Referenz oder Zusammenstellung von elementaren oder abgeleiteten Typen darstellen. Dazu gehören Pointer (Zeiger), Funktionszeiger, Arrays, Strukturen und Unionen. Diese Typen werden in anderen Kapiteln besprochen.

Des Weiteren gibt es noch den speziellen Typ `void`, der in keine Kategorie passt. Er wird in Kapitel → 7.2.8 behandelt.



Ganzzahl- und Gleitpunkt-Typen werden auch „**arithmetische Typen**“ genannt. Arithmetische (elementare) Typen und Pointer-Typen werden auch als „**skalare Typen**“, Array-Typen und Struktur-Typen als „**zusammengesetzte Typen**“ (oder „Aggregat-Typen“) bezeichnet.

Es sei hier vermerkt, dass diese Auflistung bei weitem nicht vollständig ist. Einige Datentypen wurden erst durch neuere Standards eingeführt wie beispielsweise `_Complex`, andere wie beispielsweise `short` werden nur noch in Anhang → E aufgeführt und nochmals andere sind umgebungsspezifisch wie beispielsweise der `wchar_t`-Typ, wie er in Kapitel → 6.5.6 besprochen wurde.

Die heutzutage wichtigen elementaren Datentypen für die Programmiersprache C werden im folgenden Unterkapitel genauer vorgestellt.

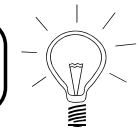
7.2 Elementare Datentypen in C

In diesem Kapitel werden die wichtigsten (elementaren) Datentypen in C genauer erläutert:

- `char`
- `int`
- `float`
- `double`
- Aufzählungs-Typen
- Boolescher Typ
- `void` Typ

7.2.1 Der Datentyp char

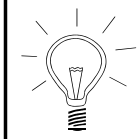
Die Größe einer Variablen vom Typ **char** ist 1 Byte.



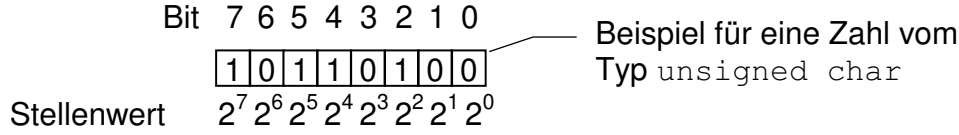
Das Wort char ist die Abkürzung von „character“ (Schriftzeichen). Gewöhnlich wird der Datentyp char dafür verwendet, um einzelne Zeichen aus dem Zeichensatz zu verarbeiten, wie beispielsweise `'c'`, oder Steuerzeichen wie `'\n'`. Der Wert eines Zeichens ist ein Integer – entsprechend dem Ausführungszeichensatz auf der Maschine. Mehr darüber kann in Kapitel → 6.1 nachgelesen werden.

Ein gespeichertes Zeichen kann als Zeichen ausgegeben werden, sein Wert kann aber auch zu Berechnungen herangezogen werden. Genauso kann man einer char-Variablen ein Zeichen oder eine kleine Zahl zuweisen, wie beispielsweise `char var = 48`. Damit eignet sich der Datentyp char zur Darstellung beziehungsweise zur Verarbeitung von Integer mit einem kleinen Wertebereich.

Die Interpretation, ob Zahl oder Zeichen, hat durch den Anwender zu erfolgen. Will man beispielsweise den Wert 48 einer char-Variablen als Zahl in Dezimalnotation ausgeben, gibt man bei `printf()` das Formatelement %d an. Will man den Wert 48 als Zeichen ausgeben, so gibt man das Formatelement %c an.



Der vorzeichenlose Datentyp `unsigned char` wird durch 1 Byte ohne Vorzeichenbit dargestellt. Alle Bits stehen für die Darstellung des Wertes einer positiven ganzen Zahl zur Verfügung. Das folgende Bild zeigt ein Beispiel für eine Zahl vom Typ `unsigned char` in Form einer Stellenwerttabelle, die für jedes der 8 Bits den entsprechenden Stellenwert anzeigt:



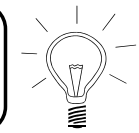
Der Wert der Zahl in diesem Beispiel berechnet sich zu:

$1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 128 + 32 + 16 + 4 = 180$

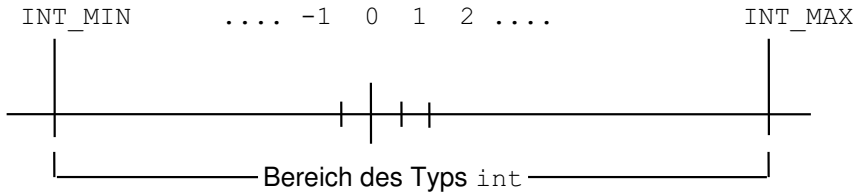
Es sei hier angemerkt, dass heutzutage angenommen werden kann, dass ein Byte 8 Bits besitzt. Der ursprüngliche Standard von C schrieb dies jedoch nicht vor. Somit hat ein `char` eigentlich keine genau vordefinierte Größe. Im Kapitel → 7.2.3 wird auf eine Lösung für dieses Versäumnis eingegangen.

7.2.2 Der Datentyp `int`

Der Datentyp **`int`** ist der Standard-Typ für die ganzen Zahlen (**Integer-Zahlen**).

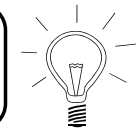


Die `int`-Zahlen umfassen auf dem Computer einen endlichen Zahlenbereich, der nicht überschritten beziehungsweise unterschritten werden kann. Dieser Bereich ist in folgendem Bild dargestellt:



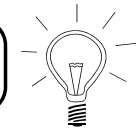
`INT_MIN` und `INT_MAX` stellen dabei die Grenzen der `int`-Werte auf einem Rechner dar. Somit gilt für jede beliebige Zahl `x` vom Typ `int`: $\texttt{INT\_MIN} \leq \texttt{x} \leq \texttt{INT\_MAX}$

Die Variablen vom Typ `int` haben als Werte ganze Zahlen im Bereich von `INT_MIN` bis `INT_MAX`.



Hierbei ist zu beachten, dass unterschiedliche Rechner den Typ `int` mit einer unterschiedlichen Anzahl Bits definieren. Entsprechend sind die Werte `INT_MIN` und `INT_MAX` auch unterschiedlich definiert.

Die Darstellung des Typs `int` ist implementierungsabhängig.



Früher entsprach die Größe eines `int` der prozessortypischen Repräsentation eines Integers. Dies bedeutet, dass je nach Prozessor ein `int` mit 16, 32 oder gar 64 Bits definiert sein konnte.

> Heutzutage jedoch hat sich der Gebrauch des `int`-Typs geändert und wurde für moderne Architekturen mehr oder weniger fixiert: Selbst auf 64-Bit-Rechnern besteht ein `int` häufig nur aus 32 Bits.

Umfasst die interne Darstellung von `int`-Zahlen 32 Bit, so entspricht dies üblicherweise einem Zahlenbereich von -2^{31} bis $+2^{31} - 1$. Wird eine `int`-Zahl durch 64 Bit dargestellt, so wird ein Wertebereich von -2^{63} bis $+2^{63} - 1$ aufgespannt. Die Entscheidung, wie viele Bits letztendlich für die Darstellung von `int`-Zahlen genommen werden, hängt vom Compilerhersteller und vom Prozessor ab, auf dem das C-Programm ablaufen soll.

Der Einfachheit halber wird beim Programmieren häufig schlicht `int` geschrieben und soll auch hier bis auf weiteres der Standard-Typ für einen Integer darstellen.

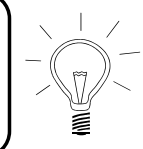
Während des Programmierens sollte man sich jedoch Gedanken darüber machen, wie groß ein Integer tatsächlich sein kann und wo immer möglich anstelle des Typs `int` Integer-Typen mit genau definierter Bitanzahl verwenden.

7.2.3 Integer-Typen mit genau definierter Bitanzahl

Da unterschiedliche Prozessoren unterschiedliche Registergrößen besitzen, sind auch die Größen des `int`-Typs je nach Prozessor unterschiedlich. Die Verwendung des Typs `int` ist somit abhängig vom Prozessor.

Um vom Prozessor unabhängig programmieren zu können, wurden im Standard C99 neue Integer-Typen definiert, um die benötigte Anzahl Bits exakt festzulegen.

> In der `<stdint.h>`-Bibliothek befinden sich Typ-Definitionen wie **`int32_t`**, welche einen Integer-Typ mit genau festgelegter Bitbreite definieren.



Es werden folgende Typen definiert:

Datentyp	Bits	Wertebereich
`int8_t`	8	-128 bis +127
`int16_t`	16	-32768 bis +32767
`int32_t`	32	-2147483648 bis +2147483647
`int64_t`	64	-9223372036854775808 bis +9223372036854775807
`uint8_t`	8	0 bis +255
`uint16_t`	16	0 bis +65535
`uint32_t`	32	0 bis +4294967295
`uint64_t`	64	0 bis +18446744073709551615

Die minimalen und maximalen Werte sind definiert durch folgende Makros:

```
int8_t:   INT8_MIN bis INT8_MAX
int16_t:  INT16_MIN bis INT16_MAX
int32_t:  INT32_MIN bis INT32_MAX
int64_t:  INT64_MIN bis INT64_MAX
uint8_t:  0 bis UINT8_MAX
uint16_t: 0 bis UINT16_MAX
uint32_t: 0 bis UINT32_MAX
uint64_t: 0 bis UINT64_MAX
```

Die `<stdint.h>`-Bibliothek definiert noch einige weitere spezielle Bitbreiten, worauf hier in diesem Buch jedoch nicht eingegangen wird.

Im Standard C11 wurden dann auch noch Typen definiert, die den Typ `char` ablösen sollen: `char16_t` und `char32_t`. Ab dem Standard C23 gibt es zusätzlich auch noch den Typ `char8_t`. Mehr dazu kann in Kapitel → 6.5.6 nachgelesen werden.

7.2.4 Die Datentypen `float` und `double`

Um Zahlen mit Nachkommastellen darzustellen, gibt es in C die sogenannten „**Gleitpunkt-Zahlen**“ in zwei Genauigkeiten: **`float`** und **`double`**.

> `float` und `double`-Zahlen entsprechen den rationalen und reellen Zahlen der Mathematik.

Man verwendet in C nicht den mathematischen Begriff der reellen Zahlen, sondern spricht von Gleitpunkt-Zahlen. Sie werden anhand der Exponentialdarstellung codiert:

$$\text{Zahl} = \text{Signifikante} \cdot \text{Basis}^{\text{Exponent}}$$

Der Typ `float` wird mit 32 Bits und der Typ `double` mit 64 Bits gespeichert. Der Typ `double` hat somit doppelt so viele Bits zur Verfügung, um Gleitpunkt-Zahlen zu speichern. Hierbei ist für beide Typen `float` und `double` genau festgelegt, wieviele Bits für die **Signifikante** und wieviele für den **Exponenten** verwendet werden. Folgende Tabelle zeigt, welche Wertebereiche möglich sind:

	`float`	`double`
Kleinstmögliche Zahl	`1.175494e-38`	`2.225074e-308`
Grösstmögliche Zahl	`3.402823e+38`	`1.797693e+308`
Anzahl genaue Dezimalstellen	6	15

Dies bedeutet beispielsweise, dass die Zahl π mit dem Typ `float` nur als die Zahl `3.141593` gespeichert werden kann, mit dem Typ `double` hingegen als die Zahl `3.141592653589793`.

> Die Codierung anhand der Exponentialdarstellung dient zur näherungsweisen Darstellung von reellen Zahlen auf Rechenanlagen, denn im Gegensatz zur Mathematik ist auf dem Rechner der Wertebereich nicht unendlich darstellbar und somit die Genauigkeit einer Zahl mit Nachkommastellen begrenzt.

Auf eine genaue Behandlung über die Codierung der Zahlen wird in Anhang → E.5 eingegangen.

Während in der Mathematik die reellen Zahlen unendlich dicht auf dem Zahlenstrahl liegen, haben Gleitpunkt-Zahlen tatsächlich diskrete Abstände zueinander. Es ist im Allgemeinen also nicht möglich, Brüche, Dezimalzahlen, transzendente Zahlen oder die übrigen nicht rationalen Zahlen wie beispielsweise die Quadratwurzel aus 2 exakt darzustellen.

Werden `float` oder `double`-Zahlen benutzt, so kommt es in der Regel zu **Rundungsfehlern**.

Wegen der Exponentialdarstellung werden die Rundungsfehler für große Zahlen größer, da die Abstände zwischen den im Rechner darstellbaren `float`-Zahlen zunehmen. Addiert man beispielsweise eine kleine Zahl `y` zu einer großen Zahl `x` und zieht anschließend die große Zahl `x` wieder ab, so erhält man meist nicht mehr den ursprünglichen Wert von `y`.

Auf Englisch heißen Gleitpunkt-Zahlen „floating point numbers“, woher auch das Schlüsselwort `float` kommt.

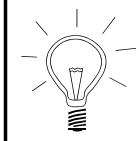
7.2.5 Aufzählungs-Typen

Aufzählungs-Typen definieren Integer, welche jedoch anhand eines Namens aufgezählt werden können. Hierfür wird das Schlüsselwort **`enum`** verwendet. Beispielsweise:

```
enum FileError {
  NO_ERROR,        // Konstante mit Wert 0
  FILE_NOT_FOUND,  // Konstante mit Wert 1
  FILE_LOCKED      // Konstante mit Wert 2
};
```

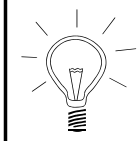
Definiert werden hier sowohl der neue Datentyp `enum FileError` als auch dessen Aufzählungs-Konstanten. Der Wertebereich des neuen Typs ist durch die Liste der Aufzählungs-Konstanten festgelegt.

Aufzählungs-Konstanten haben einen konstanten Integer-Wert. Der Typ einer Aufzählungs-Konstanten ist normalerweise `int`. Ab dem Standard C23 kann der Typ auch explizit festgelegt werden.



Der Typname des soeben definierten Typs ist `enum FileError`. Dabei ist „`FileError`“ das sogenannte „**Etikett**“ (auf Englisch „**enumeration tag**“), welches frei vergeben werden kann.

Zulässige Werte für Variablen eines Aufzählungs-Typs sind die Werte der Aufzählungs-Konstanten in der Liste der Definition des Aufzählungs-Typs. Es ist üblich, dass Aufzählungs-Konstanten groß geschrieben werden.



Auch wenn Aufzählungs-Konstanten groß geschrieben werden, sind sie doch nicht dasselbe wie symbolische Konstanten (siehe Kapitel → 21.2).

Aufzählungs-Konstanten dürfen nicht in Präprozessor-Bedingungen verwendet werden.

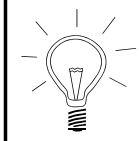
Die erste Aufzählungs-Konstante in der Liste hat standardmäßig den Wert 0, die zweite den Wert 1 usw. Es ist aber auch möglich, für jede Aufzählungs-Konstante einen Wert explizit anzugeben. Werden einige Werte in der Liste nicht explizit belegt, so wird der Wert ausgehend vom letzten explizit belegten Wert jeweils um 1 bis zum nächsten explizit angegebenen Wert hochgezählt. Dies ist in den folgenden Beispielen zu sehen:

```
enum test {ALPHA, BETA, GAMMA};                // 0, 1, 2
enum test {ALPHA = 5, BETA = 3, GAMMA = 7};    // 5, 3, 7
enum test {ALPHA = 4, BETA, GAMMA = 3};        // 4, 5, 3
```

Es dürfen jedoch keine zwei Aufzählungs-Konstanten innerhalb eines Aufzählungs-Typs mit demselben Wert existieren!

Aufzählungs-Konstanten in verschiedenen Aufzählungs-Typen müssen voneinander verschiedene Namen haben, wenn sie im selben Gültigkeitsbereich verwendet werden.

> Aufzählungs-Typen sind geeignet, um Konstanten zu definieren. Sie stellen damit in vielen Situationen eine Alternative zu der Definition von Konstanten mit Hilfe der Präprozessor-Anweisung `#define` dar.



Zu `#define` siehe Kapitel → 21.2. Für eine Beschreibung von Gültigkeitsbereichen, siehe Kapitel → 11.2.

Geschickt ist, dass bei der Definition von Aufzählungs-Konstanten Werte implizit generiert werden können, wie im folgenden Beispiel:

```
enum Monate {
  JAN = 1, FEB, MAR, APR, MAI, JUN, JUL, AUG, SEP, OKT, NOV, DEZ
};
```

Hier wird vollkommen automatisch der Februar (`FEB`) zum Monat 2, der März (`MAR`) zum Monat 3, usw.

Das Etikett kann auch weggelassen werden. Dann jedoch kann der Typ nicht mehr per Name angesprochen werden, nur die Aufzählungs-Konstanten sind noch ansprechbar. Um ihn dennoch verwenden zu können, kann er mittels `typedef` in einen selbst definierten Typ umgewandelt werden (siehe Kapitel → 14.2):

```
typedef enum {NO_ERROR, FILE_NOT_FOUND, FILE_LOCKED} ErrorId;
```

Oder aber er kann direkt für die Definition einer Variablen verwendet werden, hier beispielsweise die Variable `error`:

```
enum {NO_ERROR, FILE_NOT_FOUND, FILE_LOCKED} error;
```

Man kann in C zwar Variablen eines Aufzählungs-Typs definieren. Dennoch wird von einem C-Compiler nicht verlangt, zu prüfen, ob einer Variablen eines Aufzählungs-Typs eine passende Aufzählungs-Konstante zugewiesen wird. Beispielsweise erzeugt folgender Code keinen Fehler:

```
error = 1234;
```

Die Zuweisung beliebiger Integer-Zahlen zu einer `enum`-Variablen wird durch den Compiler nicht verboten. Man ist selbst verantwortlich dafür, dass der Wertebereich eines Aufzählungs-Typs nicht verletzt wird.

7.2.6 Der Datentyp `_Bool`

Ursprünglich hatte C im Gegensatz zu anderen Programmiersprachen keinen booleschen Typ. Anstelle dessen wurden boolesche Werte mittels der Werte `0` und nicht-`0` angegeben und als speichernde Typen werden üblicherweise Integer-Typen verwendet oder Pointer-Typen.

In C99 wurde die Standard-Bibliothek `<stdbool.h>` eingeführt, welche einen booleschen Typ für C deklariert namens `_Bool` mit den beiden **booleschen Werten** `false` und `true` als Aufzählungs-Konstanten. Häufig wird auch gleichzeitig der Typ `bool` definiert, welcher genau dasselbe darstellt wie `_Bool`. Da dieser Typ jedoch als Aufzählungs-Typ deklariert ist, ist er nach wie vor ein selbst definierter Typ und damit kein Standard-Datentyp des Compilers.

In C11 dann gilt der Typ `_Bool` als standardmäßig definiert, selbst wenn die `<stdbool.h>`-Bibliothek nicht eingebunden wird. Der Typ `bool` sowie die beiden Kontanten `false` und `true` hingegen sind nach wie vor nicht standardisiert und somit auch in C11 erst verfügbar, wenn die `<stdbool.h>`-Bibliothek eingebunden wird:

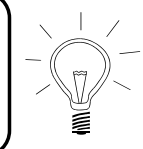
```
#include <stdbool.h>

int main(void) {
  bool a = true;
  bool b = false;
}
```

Ab dem kommenden Standard C23 dann werden `true` und `false` sowie der Typ `bool` endgültig in den Sprachumfang miteinbezogen.

7.2.7 Bit-Verarbeitung

In der Programmiersprache C gibt es keinen Typ für ein einzelnes Bit. Auch der boolesche Typ wird intern üblicherweise mit mehreren Bytes gespeichert.



Mit den in Kapitel → 9.8 beschriebenen Bit-Operatoren können jedoch Integer-Werte beliebig bitweise verknüpft werden. Dadurch ist es möglich, einzelne Bits manuell aus einem Integer auszulesen oder sie zu setzen.

Im aufkommenden Standard C23 werden zudem neue Funktionen in einer neuen Bibliothek `<stdbit.h>` eingeführt, welche das Bearbeiten von Bits vereinfacht. In diesem Buch wird nicht weiter darauf eingegangen.

7.2.8 Der Datentyp `void`

`void` ist ein Schlüsselwort und ein Datentyp. Dieser Typ bezeichnet eine leere Menge und wird beispielsweise verwendet, wenn eine Funktion keinen Rückgabewert oder keinen Übergabeparameter hat.

Pointer auf `void` werden in Kapitel → 8.2 beschrieben.

7.3 Modifikatoren und andere Besonderheiten

In diesem Kapitel wurden nur die wichtigsten elementaren Datentypen durchgenommen. Die Sprache C definiert in ihrem Kern jedoch insbesondere einige weitere arithmetische Typen, welche mittels sogenannter **Modifikatoren** angesprochen werden.

Zwei sehr bekannte Modifikatoren in der Programmiersprache C sind **`short`** und **`long`**, welche die Größe eines Integer-Typs modifizieren. Da solche Modifikatoren jedoch umgebungsabhängig, sprich von Compiler und System abhängig sind, wurden in neueren Standards präzisere, plattformunabhängige Typangaben eingeführt (siehe Kapitel → 7.2.3). In diesem Kapitel wird somit auf eine Behandlung der lange Zeit in Verwendung gewesenen `short` und `long`-Modifikatoren verzichtet. Diese werden im Anhang → E beschrieben.

Zudem sind heutzutage Speicherplatz und Rechengenauigkeit aufgrund der fortlaufenden Standardisierung und Verbesserung der Systeme nur noch in seltenen Fällen wirklich von großer Bedeutung, weswegen hier nur auf ein paar Besonderheiten aufmerksam gemacht werden soll.

7.3.1 Vorzeichenlose und vorzeichenbehaftete Datentypen

Für die elementaren Integer-Datentypen stehen die Typ-Modifikatoren **`signed`** und **`unsigned`** zur Verfügung. Durch diese beiden Modifikatoren wird festgelegt, ob ein Integer-Typ vorzeichenbehaftet ist oder nicht.

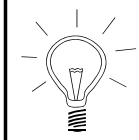
So können beispielsweise folgende Typen definiert werden:

```
signed char
unsigned char
signed int
unsigned int
```

Ob dabei der Modifikator vor oder nach dem Basis-Typ steht, spielt keine Rolle. Tatsächlich kann beim `int`-Typ der Basis-Typ gar weggelassen werden.

Werden nur die Schlüsselwörter `signed` oder `unsigned` ohne Angabe des Datentyps verwendet, so wird implizit der Datentyp `int` hinzugefügt, das heißt, es werden die Typen `signed int` beziehungsweise `unsigned int` benutzt.

Vorzeichenbehaftet, also von einem `signed`-Typ, sind standardmäßig alle Integer-Standard-Datentypen außer `char`. Der Typ `char` kann je nach Implementation oder Compilereinstellung vorzeichenbehaftet oder vorzeichenlos sein.



Mit dem Voranstellen des Modifikators `signed` vor einen Integer-Standard-Datentyp, beispielsweise `signed char`, wird erreicht, dass eine vorzeichenbehaftete Darstellung erzwungen wird. So wird bei `signed char` der Wertebereich von -128 bis +127 dargestellt.

Der Modifikator `unsigned` (nicht vorzeichenbehaftet) wird vor den Integer-Typ gestellt, wenn man erzwingen will, dass alle Werte des entsprechenden Wertebereichs größer gleich 0 sind. So kann beispielsweise bei `unsigned char` ein Wertebereich von 0 bis 255 dargestellt werden.

7.3.2 Zahlenüberlauf

Wie auch immer eine Variable schlussendlich definiert ist, sie kann bei Berechnungen nur Werte aus ihrem Wertebereich annehmen. Speichert beispielsweise eine Variable `x` vom Typ `int` einen sehr großen Wert, so kann es sein, dass das Resultat von `2 * x` größer als `INT_MAX` ist. Die Berechnung `2 * x` führt so zu einem mathematischen Fehler, dem sogenannten **Zahlenüberlauf** (auf Englisch: „**overflow**“). Auch für negative Zahlen kann dies passieren, wenn das Resultat kleiner als `INT_MIN` ist. Dann nennt man dies einen **Unterlauf** (auf Englisch: „**underflow**“).

Auf solche Zahlenüberläufe (oder -unterläufe) muss man beim Schreiben eines Programmes selbst achten. Der Zahlenüberlauf wird nämlich in C nicht durch einen Fehler oder eine Warnung angezeigt.

Wenn ein Überlauf (oder ein Unterlauf) stattfindet, wird bei modernen Prozessoren normalerweise ein „wrap-around“ durchgeführt. Das bedeutet, dass von der größtmöglichen Zahl plötzlich auf die kleinstmögliche Zahl gesprungen wird und umgekehrt.

Addiert man beispielsweise zum größten positiven Integer mit vorzeichenbehafteten Typ eine 1 dazu, so befindet man sich bei der heute üblichen Darstellung im Zweierkomplement (siehe Anhang → E.4) plötzlich im negativen Zahlenbereich.

Bei vorzeichenlosen Integer-Typen ist der Wertebereich stets eine Zahl modulo 2^n. Das bedeutet, dass von dem Resultat einer Berechnung der Rest bei der Integer-Division durch 2^n gebildet wird.

So ist beispielsweise bei einem Byte von 8 Bits die größte Zahl, die in den Typ `unsigned char` passt, die Zahl 255. Addiert man 2 dazu, so wird mit 1 weitergerechnet, da 257 modulo 256 den Wert 1 ergibt.

Zieht man eine große Zahl vom Typ `unsigned int` von einer kleinen `unsigned int`-Zahl ab, so muss das Ergebnis ebenfalls vom Typ `unsigned int` sein. Das Laufzeitsystem des Compilers hat damit kein Problem. Es rechnet einfach mit seiner Modulo-Arithmetik weiter.

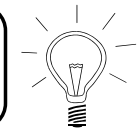
Das Laufzeitsystem des Compilers muss auf Überläufe des Wertebereichs nicht reagieren.

Meist wird in der Praxis so verfahren, dass ein Datentyp gewählt wird, der offensichtlich für die Anwendung einen standardisierten und ausreichend großen, aber nicht übermäßig großen Wertebereich hat.

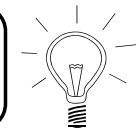
7.4 Variablen in C

In C gibt es zwei verschiedene Arten, wie **Variablen** vereinbart werden können: Definitionen und Deklarationen.

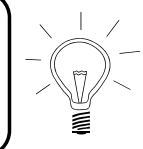
Der Begriff der **Vereinbarung** umfasst sowohl die Definition als auch die Deklaration.



Eine **Deklaration** legt die Art einer Variablen beziehungsweise die Schnittstelle einer Funktion fest.

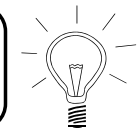


Eine **Definition** von Variablen und Funktionen dient dazu, selbige am gewünschten Ort im Speicher anzulegen. Hierbei ist automatisch eine Deklaration mit eingeschlossen.



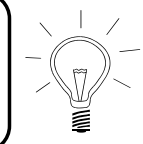
Kurz und bündig ausgedrückt bedeutet dies:

Definition = Deklaration + Reservierung des Speicherplatzes.



Reine Deklarationen dienen dazu, Datenobjekte beziehungsweise Funktionen dem Compiler bekanntzumachen, die in anderen Übersetzungseinheiten definiert werden oder in derselben Übersetzungseinheit erst nach ihrer Verwendung definiert werden.

Eine Deklaration umfasst stets den Namen eines Objektes und seinen Typ. Damit weiß der Compiler, mit welchem Typ er einen Namen verbinden muss.



Reine Deklarationen werden in den folgenden Kapiteln genauer behandelt:

- Deklaration von Variablen: → 15.3
- Definition eines neuen Typnamens mit `typedef`: → 14.2
- Vorwärtsdeklaration von Funktionen: → 11.6

Bei einer Definition von Variablen werden nebst dem Typ die sogenannte „Speicherklasse“ sowie „Qualifikatoren“ festgelegt.

7.4.1 Definition einfacher Variablen

Eine einzige Variable wird definiert durch eine Vereinbarung der Form

```
datentyp name;
```

also beispielsweise durch

```
int x;
```

Vom selben Typ können mehrere Variablen in einer einzigen Vereinbarung definiert werden, indem man die Variablennamen durch Kommas trennt wie in folgendem Beispiel:

```
int x, y, z;
```

Auf diese Schreibweise sollte jedoch nur in Ausnahmefällen zurückgegriffen werden, da sie schwieriger zu lesen ist und bei Pointern und Arrays (siehe dazu Kapitel → 8.1.2) zudem zu Fehlinterpretationen führen kann.

Die Namen der Variablen müssen den Namenskonventionen genügen (siehe Kapitel → 6.4). Natürlich darf ein Variablenname nicht identisch mit einem Schlüsselwort sein. Jede Variable in einer Folge von Definitionen von Variablen muss selbstverständlich ihren eigenen, eindeutigen Namen erhalten.

7.4.2 Externe, interne, lokale und globale Variablen und deren Bindung

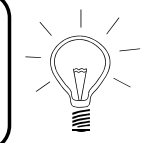
Im Folgenden werden folgende vier Begriffe beschrieben:

- Interne Bezeichner
- Externe Bezeichner
- Intern gebundene Bezeichner
- Extern gebundene Bezeichner

Ein C-Programm besteht aus Funktionen. Etwas, was sich innerhalb einer Funktion befindet, wird als „Interner Bezeichner" bezeichnet. Etwas, was sich außerhalb befindet als „externer Bezeichner".

Funktionen sind zueinander extern, da in C keine Funktion innerhalb einer Funktion definiert werden kann. Variablen können entweder innerhalb oder außerhalb von Funktionen definiert werden.

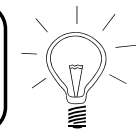
Interne Variablen werden umgangssprachlich als **lokale Variablen** bezeichnet. **Externe Variablen** als **globale Variablen**.



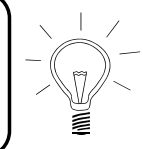
Die beiden Begriffe „lokal" und „global" sind umgangssprachliche Begriffe, welche im eigentlichen Standard nicht verwendet werden. Dennoch werden sie auch in diesem Buch gerne genutzt, um sie von den sogenannten „**Bindungen**" (auf Englisch „**Linkage**") zu unterscheiden:

Des Weiteren unterscheidet ein Compiler nämlich auch zwischen sogenannten „intern gebundenen" und „extern gebundenen" Bezeichnern.

Intern gebundene Bezeichner sind Namen, die innerhalb einer Übersetzungseinheit verwendet werden.

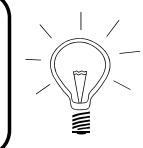


Extern gebundene Bezeichner sind Namen, die für mehrere Übersetzungseinheiten gültig sind. Sie haben also auch eine Bedeutung außerhalb der betrachteten Datei.



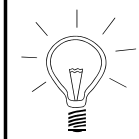
Extern gebundene Namen haben insbesondere eine Bedeutung für den Linker, der unter anderem die Bindungen zwischen Namen in separat bearbeiteten Übersetzungseinheiten herstellen muss. Siehe dazu auch Kapitel → 15.1.2.

Namen mit interner Bindung existieren eindeutig für jede Übersetzungseinheit. Namen mit externer Bindung existieren eindeutig für das ganze Programm.

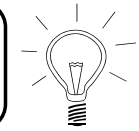


7.4.3 Globale Variablen

Funktionsinterne Variablen sind lokal zu einer Funktion. Sie haben nur innerhalb ihrer Funktion eine Bedeutung. Sie werden üblicherweise als „lokale Variablen“ bezeichnet. Funktionsexterne Variablen haben die Bedeutung von globalen Variablen für diejenigen Funktionen, die in einer Datei nach ihnen definiert werden.



Globale Variablen können grundsätzlich allen Funktionen eines Programms zur Verfügung stehen.



Um zu verstehen, warum funktionsexterne Variablen globale Variablen sind, muss man zuerst den Begriff der **Sichtbarkeit** verstehen. Dieser sagt: Die Sichtbarkeit einer Variablen bedeutet, dass man von einer Programmstelle aus die Variable sieht, das heißt, dass man auf sie über ihren Namen zugreifen kann. In Kapitel → 11.2 wird genauer auf dieses Thema eingegangen.

Da globale Variablen für alle Funktionen, die nach ihnen in einer Datei definiert werden, sichtbar sind, können sie von all diesen Funktionen verwendet und gelesen oder beschrieben werden. Die externen Variablen stehen also diesen Funktionen gemeinsam, in anderen Worten global, zur Verfügung. Daher auch der Name globale Variablen.

Globale Variablen sollte man so wenig wie möglich verwenden, da sie leicht zu schwer erkennbaren Fehlern führen können.

> Die Verwendung von globalen Variablen wird aus dem Blickwinkel des Software Engineering nicht gerne gesehen, da bei der Verwendung globaler Variablen leicht die Übersicht verloren geht und es unter Umständen zu schwer zu findenden Fehlern kommen kann. Arbeiten mehrere Funktionen auf einer globalen Variablen, so ist der Verursacher eines Fehlers schwer zu finden.

Dennoch kann es – vor allem bei der hardwarenahen Programmierung – zu Situationen kommen, wo man aus Performance-Gründen gezwungen ist, globale Variablen zu verwenden.

Deshalb sollten globale Variablen nur in zwingenden Fällen verwendet werden, beispielsweise wenn es aus Performance-Gründen nicht mehr ratsam ist, mit Übergabeparametern zu arbeiten.

Im folgenden Beispiel wird eine globale Variable `alpha` eingeführt:

globvar.c

```
#include <stdio.h>

int alpha = 3;

void f(void) {
  int a;
  a = alpha;
  printf("a hat den Wert %d\n", a);
}

int main(void) {
  int b = 4;
  printf("alpha hat den Wert %d\n", alpha);
  alpha = b;
  printf("alpha hat den Wert %d\n", alpha);
  f();

  return 0;
}
```

Die Ausgabe des Programms ist:

```
alpha hat den Wert 3
alpha hat den Wert 4
a hat den Wert 4
```

Mit der Vereinbarung `int alpha = 3;` wird die globale Variable `alpha` zu Programmbeginn angelegt. Damit ist die Variable `alpha` in den Funktionen `f()` und `main()` sichtbar. Man kann in diesen beiden Funktionen auf sie lesend oder schreibend zugreifen. Die Lebensdauer der externen Variablen `alpha` erstreckt sich bis zum Programmende. Erst nach Ablauf des Programms wird der Speicherplatz dieser Variablen freigegeben.

7.4.4 Initialisierung von Variablen

Jede einfache Variable kann bei ihrer Definition **initialisiert** werden, indem man einfach ein Gleichheitszeichen = gefolgt von einem Ausdruck des passenden Typs an den Namen der Variablen anhängt. Im folgenden Beispiel werden einige `int`- und `char`-Variablen bei ihrer Definition initialisiert:

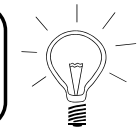
```
int main (void) {
  int a = 9;
  int b = (4 + 5) * 6;
  int c;                    // c ist nicht initialisiert

  char c1 = 'x';
  char c2 = 'y';
}
```

Da diese Initialisierung von Hand durch Zuweisung eines Wertes vorgenommen wird, wird sie auch „**manuelle Initialisierung**“ genannt.

Wenn eine lokale Variable nicht manuell initialisiert wird, so ist ihr Inhalt unbestimmt. Der Compiler nimmt für lokale Variablen keine automatische Initialisierung vor.

Globale Variablen werden zu Beginn eines Programms automatisch mit 0 initialisiert.



Mehr Informationen zu manuellen und automatischen Initialisierungen können in Kapitel → 15 nachgelesen werden.

Ab dem Standard C23 können Variablen mittels der aus C++ bekannten „Null-Initialisierung“ mittels öffnender und schließender geschweiter Klammer vollständig mit 0 gefüllt werden:

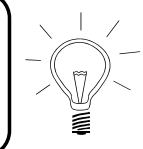
```
struct MyStruct s = {};
```

7.5 Qualifikatoren

Ein C-Compiler macht über die Verwendung von Variablen gewisse Annahmen. Beispielsweise, dass jede Variable innerhalb eines Codes grundsätzlich veränderbar ist oder dass die Einträge eines Arrays sich nicht mit einem zweiten Array überlappen.

Wenn bei der Erstellung eines Programmes jedoch klar ist, dass der Compiler diese Grundannahmen nicht machen soll, so kann mittels sogenannter „**Typ-Qualifikatoren**“ dem Compiler ein Hinweis gegeben werden, wie der Code zu behandeln ist.

Qualifikatoren sind an den Typ einer Variable gebunden. Sie verändern die Art und Weise, wie der Compiler auf die Speicherinhalte zugreift.



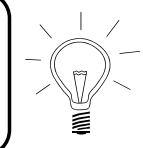
Dabei ist wichtig zu verstehen, dass Qualifikatoren an den Typ gebunden sind, und nicht an die Variable oder den Wert, der dahinter liegt. Auf Werte oder Speicheradressen wird oftmals im selben Programm an unterschliedlichen Stellen mittels unterschiedlich qualifizierter Typen zugegriffen. Die Angabe eines Qualifikators zum Typ ist für den Compiler somit ein Hinweis, wie das Ansprechen des Wertes während der Gültigkeit einer Variablen zu erfolgen hat.

Gerade bei Variablen mit einem Pointer- oder Array-Typ sind Hinweise für den Compiler wichtig, um falschen Zugriffen vorzubeugen und korrupte Daten zu vermeiden. Auf Pointer und Arrays wird in Kapitel → 8 eingegangen.

In C gibt es 4 Qualifikatoren: `const`, `restrict`, `volatile` und `_Atomic`. Der `restrict`-Qualifikator wird in Kapitel → 8.4 und der `_Atomic`-Qualifikator in Kapitel → 23.5.5 behandelt. Hier finden sich nur die Erläuterungen zu den Qualifikatoren `const` und `volatile`:

7.5.1 Der Qualifikator `const`

Mit dem Schlüsselwort **`const`** ist es möglich, eine Variable zu vereinbaren, welche nur gelesen werden kann. Der Compiler schützt diese Variable vor Schreibzugriffen.



Es ist somit beispielsweise möglich, anstatt mit `#define PI 3.1415927` eine klar typisierte, aber schreibgeschützte Variable zu definieren:

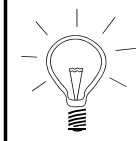
```
const double PI = 3.1415927;
```

Die sofortige Initialisierung der Variablen ist verbindlich, denn nach der Initialisierung ist die Variable für Schreibzugriffe gesperrt.

Sehr wichtig ist das `const`-Schlüsselwort insbesondere bei der Übergabe von Argumenten per Pointer. Wenn eine Funktion ein Argument mit dem Schlüsselwort `const` erwartet, so gibt dies den Hinweis, dass die Funktion dieses Argument nicht verändern wird. Mehr dazu kann in Kapitel → 11.5.2 nachgelesen werden.

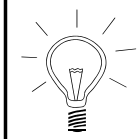
Der Compiler unterstützt das Schreiben von sicherem Code, indem er Verletzungen von `const` als Fehler meldet.

Durchgängige Programmierung mit dem Schlüsselwort `const` wird als „**const-safe-Programmierung**“ bezeichnet. Gerade bei der Übergabe von Pointern als Parameter kann diese Sicherheitsvorkehrung einem Fehlverhalten von Code vorbeugen.



7.5.2 Der Qualifikator `volatile` 

> Der Qualifier **`volatile`** wird gebraucht, wenn eine Variable implementierungsabhängige Eigenschaften hat und der Compiler keine **Optimierung** durchführen soll. Implementierungsabhängig bedeutet hier, dass sich der Wert einer Variablen durch andere Prozesse wie beispielsweise durch Interrupts ändern kann.



Dies ist beispielsweise bei Memory-Mapped-Input/Output der Fall. Hier werden Register durch Memory-Adressen angesprochen und nicht über Ports (Kanäle). Hier müssen die entsprechenden Variablen stets an denselben Adressen stehen und der Compiler darf die Variablen nicht aus Optimierungsgründen woandershin wie beispielsweise in Register verschieben.

Weiteres Beispiel: Ein optimierender Compiler kann feststellen, ob sich der Wert einer Variablen im einem bestimmten Programmfluss ändert oder nicht. Ändert sich der Wert nicht, könnte er sich den zu Beginn berechneten Wert „merken“ und diesen zwischengespeicherten Wert im weiteren Verlauf des Programms verwenden, statt mehrfach auf den Speicher der Variablen zugreifen.

Daher darf ein Compiler bei der Verwendung des Schlüsselwortes `volatile` nicht optimieren, beispielsweise weder die Variable verschieben noch zwischengespeicherte Werte verwenden.

Entsprechende Variablen sind heutzutage kaum mehr anzutreffen, da moderne Betriebssysteme Zugriffe auf solche Speicherstellen durch gesicherte Bibliotheksaufrufe anbieten und sie so verstecken.

7.6 Übungsaufgaben

Aufgabe 1: Initialisierung von lokalen Variablen

Erstellen Sie ein Programm, in welchem jeweils eine lokale Variable der Typen `int`, `float`, `double` und `char` definiert, aber nicht initialisiert wird. Geben Sie den Inhalt der Variablen aus. Beobachten Sie das Ergebnis und beantworten Sie, weshalb Variablen immer initialisiert werden sollten.

Aufgabe 2: Initialisierung von globalen Variablen

Ändern Sie das vorherige Programm, indem Sie die lokalen Variablen zu globalen machen. Beobachten Sie erneut das Ergebnis. Was ist festzustellen?