edv-buchversand.de

C programmieren

Das umfassende Handbuch

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

Kapitel 4

Variablen, Datentypen und Typsicherheit

Programme verarbeiten Daten. Der Compiler muss wissen, wie Werte im Speicher abgelegt werden, und dafür benötigt jeder Wert einen Typ. Das Kapitel beginnt mit Konstanten und Literalen, den festen Werten im Quelltext: ganze Zahlen, Gleitkommazahlen, Zeichen und Zeichenketten. Danach folgen die Grundtypen sowie Variablen und Konstanten, ihre Deklaration und Initialisierung sowie die Typbestimmung durch den Compiler.

4.1 Konstanten und String-Literale: feste Angaben im Quelltext

Zahlen, Zeichen und Zeichenketten lassen sich unmittelbar im Quelltext notieren. Der C-Standard unterscheidet dabei zwischen **Konstanten** und **String-Literalen**.

Jede Konstante besitzt einen Typ und einen Wert. Der Typ wird anhand ihrer Form und ihres Wertes bestimmt. Der Wert muss im darstellbaren Wertebereich dieses Typs liegen.

String-Literale bilden eine eigene Kategorie. Ein gewöhnliches String-Literal beschreibt eine Folge von Zeichen.

4.1.1 Ganzzahlkonstanten, Gleitkommakonstanten, Zeichenkonstanten und String-Literale

Zu den zunächst behandelten Schreibweisen gehören Ganzzahlkonstanten, Gleitkommakonstanten, Zeichenkonstanten und String-Literale. Jede Kategorie besitzt eine eigene Schreibweise, an der der Compiler sie erkennt.

Kategorie	Beispiel(e)	Schreibweise
Ganzzahlkonstante	`0`, `69`, `1982`	hier in dezimaler Schreibweise, ohne Dezimalpunkt oder Exponent
Gleitkommakonstante	`3.14`, `0.5`, `2.`, `1e3`	enthält einen Dezimalpunkt oder einen Exponenten

Tabelle 4.1: Grundlegende Konstanten und String-Literale in C

Kategorie	Beispiel(e)	Schreibweise
Zeichenkonstante	`'A'`, `'\n'`	Zeichen oder Escape-Sequenz in einfachen Anführungszeichen
String-Literal	`"Hallo"`	Zeichenketten in doppelten Anführungszeichen

Tabelle 4.1: Grundlegende Konstanten und String-Literale in C (Forts.)

Tipp

Zu Beginn des C-Studiums werden Zeichenkonstanten und Strings (`'A'` ist nicht dasselbe wie `"A"`) gerne verwechselt.

4.1.2 Literalverkettung

C erlaubt es, mehrere String-Literale direkt hintereinanderzuschreiben; der Compiler fügt sie automatisch zu einem einzigen String zusammen. Zwischen den einzelnen Literalen darf beliebiger Leerraum stehen, also Leerzeichen, Tabulatoren oder auch ein Zeilenumbruch. Der C-Standard nennt dieses Vorgehen **Verkettung benachbarter String-Literale**, kurz **Literalverkettung**. Dies ist nützlich, um sehr lange Strings übersichtlich auf mehrere Zeilen zu verteilen oder um Escape-Sequenzen klar abzugrenzen.

Beispiel:

```
// GNfN.c
#include <stdio.h>
int main() {
  // Verkettung über Zeilenumbruch
  puts("Das ist "
       "ein langer "
       "String.");    // → Das ist ein langer String.


  // Verkettung mit Tabulator dazwischen
  puts("Hallo"     "Welt"); // → HalloWelt
}
```

Hinweis

- Andere Tokens dürfen nicht zwischen den String-Literalen stehen. Kommentare sind dagegen zulässig, weil sie bereits vor der Literalverkettung durch Leerraum ersetzt werden.

- In Sprachen wie Java, Python oder JavaScript werden Zeichenketten mit dem Plus-Operator (`+`) verknüpft. In C ist das nicht möglich: Der Plus-Operator ist ausschließlich für arithmetische Berechnungen vorgesehen.

4.2 Werte mit der Funktion zur formatierten Ausgabe printf() ausgeben

`puts()` gibt eine Zeichenkette aus und fügt anschließend einen Zeilenumbruch hinzu. Für die formatierte Ausgabe von Zahlen, Zeichen und Zeichenketten steht die in `<stdio.h>` deklarierte Funktion `printf()` zur Verfügung.

Das erste Argument von `printf()` ist ein **Formatstring** genannt. Sie legt fest, wie die nachfolgenden Argumente ausgegeben werden.

Ein Formatstring kann **Konvertierungsspezifikationen** enthalten. Sie beginnen mit `%` und enden mit einem **Konvertierungsspezifizierer**. So ist beispielsweise `%d` eine Konvertierungsspezifikation und `d` der darin enthaltene Konvertierungsspezifizierer.

Im folgenden Beispiel werden Werte ausgegeben, die durch verschiedene Konstanten und ein String-Literal angegeben werden:

```
// 2Qz7b.c
#include <stdio.h>

int main() {
  printf("%d\n", 69);      // Ganzzahlkonstante    → 69
  printf("%f\n", 3.14);    // Gleitkommakonstante  → 3.140000
  printf("%c\n", 'A');     // Zeichenkonstante     → A
  printf("%s\n", "Hallo"); // String-Literal       → Hallo
}
```

Die verwendeten Konvertierungsspezifikationen sind:

- `%d` für eine ganze Zahl
- `%f` für eine Gleitkommazahl
- `%c` für ein Zeichen
- `%s` für eine Zeichenkette

In den String-Literalen steht die Escape-Sequenz `\n` für ein Zeilenumbruchzeichen. `printf()` fügt im Gegensatz zu `puts()` nicht selbstständig einen Zeilenumbruch hinzu.

Nach dem Formatstring kann `printf()` eine variable Anzahl weiterer Argumente erhalten. Daher ist `printf()` eine variadische Funktion. Anzahl und Typen der Argumente müssen zu den Konvertierungsspezifikationen passen.

Hinweis

Beim Programmstart verwendet C die Locale `"C"`. In dieser Locale wird bei der formatierten Ausgabe von Gleitkommazahlen ein Punkt als Dezimaltrennzeichen verwendet. Ein Programm kann mit `setlocale()` eine andere Locale aktivieren. Das Dezimaltrennzeichen kann dann von der Kategorie `LC_NUMERIC` der gewählten Locale abhängen.

4.2.1 Nicht passende Konvertierungsspezifikationen

Ein häufiger Fehler besteht darin, dass eine Konvertierungsspezifikation nicht zum Typ des zugehörigen Arguments passt. Im folgenden Aufruf hat die Gleitkommakonstante `1.1` den Typ `double`, während `%d` ein Argument des Typs `int` erwartet. Das Verhalten des Programms ist daher undefiniert.

```
// 2MPou.c
#include <stdio.h>

int main() {
  printf("%d\n", 1.1); // falsch: %d erwartet int
}
```

Ein Compiler kann den Aufruf trotzdem übersetzen. GCC prüft mit der Option `-Wformat`, ob die Argumenttypen zu den Konvertierungsspezifikationen passen. Die Option `-Wall` aktiviert diese Prüfung. Eine mögliche GCC-Warnung lautet:

```
<source>:4:14: warning: format '%d' expects argument of type 'int', ↩
but argument 2 has type 'double' [-Wformat=]
    4 |     printf("%d\n", 1.1);
      |             ~^     ~~~
      |              |     |
      |              int   double
      |             %f
```

Hinweis: Undefiniertes Verhalten

Undefiniert bedeutet: Die Sprache legt nicht fest, was in diesem Fall passieren muss. Der Compiler darf also Annahmen treffen, die zu überraschenden Ergebnissen führen können. Das Programm kann scheinbar funktionieren, abstürzen oder sich nach einer kleinen Änderung plötzlich ganz anders verhalten. Merksatz: Normalerweise hat der Compiler immer recht. Außer bei undefiniertem Verhalten, denn dann hat niemand mehr recht.

4.2.2 Dezimale Gleitkommakonstanten mit Exponent

Dezimale Gleitkommakonstanten können einen Exponententeil enthalten. Dieser wird durch `e` oder `E` eingeleitet. Der angegebene Exponent bestimmt die Zehnerpotenz, mit der der vorangestellte Wert multipliziert wird.

```
// 2SCeS.c
#include <stdio.h>

int main() {
  printf("%f\n", 3.14);     // → 3.140000
  printf("%f\n", 1.23e4);   // → 12300.000000
}
```

`1.23e4` bedeutet $1{,}23 \times 10^4 = 12300$. Diese Schreibweise vereinfacht die Angabe sehr großer und sehr kleiner Werte.

Hinweis

Wer aus Umgebungen wie MATLAB oder Excel kommt, kennt das Zeichen `^` als Potenzoperator. In C ist `^` kein Potenzoperator, sondern der **bitweise exklusive Oder-Operator XOR**. Beide Operanden müssen einen Ganzzahltyp haben. Ein Bit des Ergebnisses wird genau dann auf `1` gesetzt, wenn genau eines der entsprechenden Operandenbits den Wert `1` hat.

4.2.3 Aufgabe

- Gib Werte verschiedener Typen mit `printf()` aus, etwa `int`, `double` und `char`.
- Gib denselben Gleitkommawert mit `%f` und `%e` aus und vergleiche die Darstellungen.
- Verändere die Ausgabepräzision, beispielsweise mit `%.2f` und `%.2e`.
- Verwende absichtlich eine nicht passende Konvertierungsspezifikation und lasse den Aufruf mit `-Wall` oder `-Wformat` vom Compiler prüfen.
- Erkläre, warum aus dem Laufzeitverhalten eines Programms mit undefiniertem Verhalten keine allgemeingültige Aussage abgeleitet werden kann.

4.3 Grundtypen

In C bestimmt ein **Datentyp**, welche Art von Werten ein bestimmter Speicherbereich aufnehmen kann und wie der im Speicher abgelegte Wert zu interpretieren ist. Er legt auch dessen Größe in Bits fest. Mit anderen Worten: Der Datentyp legt etwa fest, ob dort eine ganze Zahl, eine Zahl mit Nachkommastellen oder nur ein einfacher Wahrheitswert stehen darf.

C definiert eine Reihe von **Grundtypen** (engl. **basic types**), die die Grundlage für alle Programme bilden. Diese Typen beeinflussen sowohl die Korrektheit als auch die Effizienz des Codes. Die Norm beschreibt diese Typen detailliert. Wer in C programmiert, muss sie kennen, um portable und zuverlässige Programme zu schreiben.

Die grundlegenden Kategorien sind Ganzzahltypen, Gleitkommatypen, Zeichentypen, Wahrheitswerte, bitgenaue Ganzzahlen und der spezielle Typ `void`. Die folgende Tabelle gibt einen Überblick:

Kategorie	Typbeispiele	Verwendung
Ganzzahlen	`int`, `short`, `long`, `long long`	ganze Zahlen ohne Nachkommastellen, z. B. Zähler, Altersangaben, Stückzahlen
Gleitkommazahlen	`float`, `double`, `long double`	Zahlen mit Nachkommastellen, z. B. Längen, Preise, physikalische Größen
Zeichen	`char`	einzelne Zeichen wie `'A'`, `'z'` oder `'#'`; intern als Zahl gespeichert
Wahrheitswerte	`bool`	logische Werte: wahr oder falsch; praktisch für Bedingungen in Programmen
Spezieller Typ	`void`	bedeutet »kein Wert«; wird z. B. für Funktionen genutzt, die nichts zurückgeben

Tabelle 4.2: Grundtypen in C und ihre typische Verwendung

4.4 Variablen

Nach der Betrachtung der Grundtypen folgt die Handhabung von Variablen, die Werte speichern und manipulieren.

4.4.1 Deklaration von Variablen

Eine Variable ist ein benannter Speicherbereich, in dem Werte eines bestimmten Typs abgelegt werden. Bevor wir eine Variable verwenden können, muss sie deklariert sein. Für lokale Variablen ist dies in der Regel eine Definition, die Speicher reserviert.

- Eine **Deklaration** macht einen Bezeichner bekannt und legt den Typ fest.
- Eine **Definition** ist eine spezielle Art der Deklaration, die zusätzlich Speicher reserviert.

Beispiel: Zuerst wird eine Variable `year` definiert, und in einer nachfolgenden Anweisung erhält sie mit dem Gleichheitszeichen einen Wert zugewiesen.

```c
// 2TXZF.c
#include <stdio.h>

int main() {
  int year;          // Definition (nicht nur Deklaration)
  year = 2027;       // Zuweisung eines Werts
  printf("year=%d\n", year); // → year=2027
}
```

Über den Typ (im Beispiel `int`) kann der Compiler die Speichergröße, die Ausrichtung und die Typkompatibilität bei Operationen bestimmen.

Namenskonventionen für Variablen

Variablennamen sind technisch fast beliebig wählbar, doch für die Lesbarkeit und Wartbarkeit eines Programms ist eine einheitliche und nachvollziehbare Benennung entscheidend. Grundregeln:

- **Beschreibend statt kryptisch:** Verwende sprechende Namen wie `year` oder `user_age` statt kurzer Kürzel wie `a` oder `x`. So erkennt man sofort den Zweck einer Variablen. Für grundlegende Beispiele nutzt dieses Buch teilweise bewusst kurze Namen wie `x` oder `n`, damit der Fokus auf der Struktur des Programms liegt und nicht auf der Bedeutung der Variablen.
- **Englische Begriffe:** Variablen werden in der Praxis fast immer auf Englisch benannt. Das sorgt für Konsistenz mit Bibliotheken, Dokumentation und internationaler Zusammenarbeit. In Ausnahmefällen können auch nicht-englische Bezeichner sinnvoll sein, etwa bei fachlich etablierten Begriffen (z. B. aus Recht oder Buchhaltung), die sich nicht präzise übersetzen lassen oder die direkt aus externen Datenquellen stammen. Solche Ausnahmen sollten jedoch bewusst gewählt und konsistent verwendet werden.
- **Keine unklaren Abkürzungen:** Abgekürzte Namen sind nur sinnvoll, wenn sie allgemein verständlich sind (`max`, `idx`).

Für die Schreibweise mehrteiliger Namen haben sich in der Programmierung verschiedene Stile etabliert:

- **snake_case:** Alle Wörter werden kleingeschrieben und mit Unterstrichen verbunden (z. B. `user_age`). Der Name rührt daher, dass die Unterstriche wie eine »Schlange« durch den Text laufen. In der C-Programmierung ist dies die gängige Konvention für Variablen- und Funktionsnamen.
- **camelCase:** Die Wörter werden ohne Unterstriche aneinandergereiht, das erste Wort klein, jedes weitere beginnt mit einem Großbuchstaben (z. B. `userAge`). Die Großbuchstaben »ragen wie Höcker« aus dem Wort, daher der Name. Dieser Stil ist in C eher unüblich, gilt dagegen etwa in Java und JavaScript als Standard.

- **PascalCase**: Wie camelCase, jedoch beginnt auch das erste Wort mit einem Großbuchstaben (z. B. `UserAge`). Dieser Stil wird oft für Typen, Klassen, Namespaces oder Module genutzt und ist etwa in C#, .NET oder Delphi verbreitet.
- **SCREAMING_SNAKE_CASE**: Eine Variante von snake_case in Großbuchstaben (beispielsweise `MAX_VALUE`). Sie ist die traditionelle Konvention für Konstanten und Makros in C und C++.

Jede Sprache und jedes Projekt pflegen eigene Konventionen. Entscheidend ist, dass man sich innerhalb eines Projekts an den vorgegebenen Stil hält und ihn konsequent anwendet. Wer sich am Erscheinungsbild der Standardbibliotheken in C orientieren möchte, sollte snake_case verwenden.

Kein Zugriff auf unbekannte Variablen möglich

Greift ein Programm auf eine Variable zu, die zuvor nicht deklariert wurde, meldet der Compiler einen Fehler. Beispiel:

```
// YtqP.c
#include <stdio.h>

int main() {
  printf("x=%d\n", x);
}
```

GCC meldet:

```
<source>: In function 'main':
<source>:4:23: error: 'x' undeclared (first use in this function)
    4 |     printf("x=%d\n", x);
      |                       ^
<source>:4:23: note: each undeclared identifier is reported only once for ↩
each function it appears in
```

Die Meldung `error: 'x' undeclared` bedeutet, dass der Bezeichner `x` vor seiner Verwendung nicht deklariert wurde. Der Zusatz `(first use in this function)` kennzeichnet den ersten entsprechenden Zugriff innerhalb von `main()`.

Tipp

Deklariere Variablen nur dann, wenn sie tatsächlich benötigt werden. Entferne ungenutzte Variablen (die beispielsweise bei Änderungen von Programmcode »übrig« bleiben) wieder. Standardmäßig warnt der C-Compiler nicht vor ungenutzten lokalen Variablen. Mit der Option `-Wall` lässt sich diese Warnung bei GCC (und Clang) jedoch aktivieren. Im Miniprogramm `int main() { int x; }` gibt GCC mit `-Wall` folgende Warnung aus:

```
<source>:2:9: warning: unused variable 'x' [-Wunused-variable]
         2 |     int x;
           |         ^
```

Auch wenn eine lokale Variable nur geschrieben, aber nicht gelesen wird, erscheint die gleiche Warnung. Das Programm `int main() { int x = 1; }` erzeugt dieselbe Meldung wie oben.

Uninitialisierte Variablen

Eine häufige Fehlerquelle ist der Lesezugriff auf uninitialisierte Variablen. Die sogenannten automatischen Variablen ohne Anfangswert besitzen zunächst einen nicht festgelegten Speicherinhalt (oft als **indeterminierte Repräsentation** bezeichnet). Das heißt: Die Variable existiert zwar, enthält aber einen Restwert aus dem Speicher. Solange man sie nicht liest, ist das erlaubt, doch beim Zugriff legt der Standard fest, dass das Verhalten undefiniert ist. Die Folgen reichen von scheinbar zufälligen Zahlen über Programmabstürze bis zu schwer reproduzierbaren Seiteneffekten.

Ein Beispiel:

```
// 2Boa1.c
#include <stdio.h>

int main() {
  int x;       // Deklaration ohne Initialisierung
  printf("x=%d\n", x);  // → x=<nicht definiert>
}
```

Der Compiler kann dieses Programm übersetzen, warnt bei aktivierten Warnoptionen jedoch vor dem uninitialisierten Zugriff.

Mit der Option `-Wall` weist GCC auf das Problem hin:

```
<source>:5:5: warning: 'x' is used uninitialized [-Wuninitialized]
    5 |     printf("x=%d\n", x);
      |     ^~~~~~~~~~~~~~~~~~~
<source>:4:9: note: 'x' was declared here
    4 |     int x;
      |         ^
```

Clang warnt und macht gleich einen Vorschlag, dass wir die Variable `x` mit `0` initialisieren können:

```
<source>:5:23: warning: variable 'x' is uninitialized when used ↩
here [-Wuninitialized]
    5 |     printf("x=%d\n", x);
      |                       ^
```

```
<source>:4:10: note: initialize the variable 'x' to silence this warning
    4 |     int x;
      |          ^
      |           = 0
```

4.4.2 Deklaration mit Initialisierung

Bei der **Initialisierung** erhält eine Variable bereits bei ihrer Deklaration einen definierten Anfangswert. Dies geschieht in C, indem hinter dem Variablennamen ein Gleichheitszeichen (`=`) folgt.

```
// muF1.c
#include <stdio.h>

int main() {
  double y = 3.14;      // Deklaration mit Initialisierung
  printf("y=%f\n", y);  // → y=3.140000
}
```

Lokale Variablen sollten möglichst direkt initialisiert werden. Das verbessert die Lesbarkeit und verhindert Zugriffe auf unbestimmte Werte. Eine Ausnahme ist sinnvoll, wenn der Wert unmittelbar danach durch eine Zuweisung oder Funktion gesetzt wird.

> **Hinweis**
> Seit C99 dürfen in C lokale Variablen mitten im Block deklariert werden. In C90 mussten Deklarationen noch am Blockanfang stehen, deshalb findet man diese Schreibweise in sehr frühen C-Programmen und historischem Code häufig noch.

Konvertierung mit implementierungsdefiniertem Ergebnis

Wird einer `int`-Variable beispielsweise eine Ganzzahlkonstante zugewiesen, deren Wert nicht durch `int` dargestellt werden kann, muss der Wert nach `int` konvertiert werden. Ein Beispiel:

```
// 25KU1.c
int main() {
  int i = 3000000000; // bei 32-Bit-int nicht darstellbar
}
```

GCC warnt bei `-Wall` nicht vor dem Problem, wohl aber mit `-pedantic` (oder präziser `-Woverflow`):

```
<source>:2:13: warning: overflow in conversion from 'long int' to ↩
'int' changes value from '3000000000' to '-1294967296' [-Woverflow]
    2 |     int i = 3000000000;
      |             ^~~~~~~~~~
```

Die Ganzzahlkonstante `3000000000` passt bei einem 32-Bit nicht in `int`. Daher erhält sie gemäß den Regeln für Ganzzahlkonstanten einen ausreichend großen Typ, etwa `long` oder `long long`. Bei der anschließenden Konvertierung nach int ist der Wert nicht darstellbar. Das Ergebnis dieser Konvertierung ist bei einem vorzeichenbehafteten Zieltyp implementierungsdefiniert.

4.4.3 Werte einlesen (scanf)

Um Tastatureingaben zu lesen, stellt C die Funktion `scanf()` bereit. Sie ist sozusagen das Gegenstück zu `printf()`: Während `printf()` Werte auf dem Bildschirm ausgibt, liest `scanf()` Werte von der Tastatur ein. Bei einem normalen Konsolenprogramm erfolgt diese Eingabe meist über die Tastatur.

Ein einfaches Beispiel zum Einlesen einer Ganzzahl:

```
// Q3oP.c
#include <stdio.h>

int main() {
  int number;
  printf("Bitte eine Zahl eingeben: ");   // Aufforderung
  scanf("%d", &number);                   // Wert mit Adresse (&) einlesen
  printf("Eingegeben: %d\n", number);     // Eingabe wieder ausgeben
}
```

Die Interaktion kann so aussehen:

```
Bitte eine Zahl eingeben: 123
Eingegeben: 123
```

Wichtig

Auf Online-Compiler-Seiten kann es sein, dass der Programmablauf angezeigt wird, aber eine echte Eingabe über `scanf()` nicht immer möglich ist.

Formatstrings und Konvertierungsspezifikationen

Wie bei `printf()` gibt es auch bei `scanf()` einen Formatstring. Er besteht aus Text, der wörtlich in der Eingabe erwartet wird, und Konvertierungsspezifikationen. Diese legen fest, wie die Eingabe interpretiert werden soll: `%d` liest eine Ganzzahl, `%f` eine Gleitkommazahl, `%c` ein einzelnes Zeichen und `%s` eine Zeichenkette.

Im Gegensatz zu `printf()` erwartet `scanf()` jedoch nicht die Werte selbst, sondern deren Adressen. Daher steht vor der Variablen der Operator `&`. Einlesen kann man nur in Variablen, also in Speicherstellen, die eine Adresse haben. Im Standard wird ein solcher Ausdruck, der ein Objekt im Speicher bezeichnet, als modifizierbarer **L-Value** bezeichnet. Nur

solche Ziele kann `scanf()` beschreiben. Mit Zeigern, Adressen und Speicherstellen beschäftigen wir uns intensiver in Kapitel 13.

Die nachfolgende Tabelle zeigt wichtige Konvertierungsspezifikationen für `scanf()`:

Format	Bedeutung (Eingabe)	Beispiel
`%d`	`int` als Dezimalzahl	`123` wird als `int` 123 eingelesen
`%u`	`unsigned int`	`123` wird als `unsigned int` 123 eingelesen
`%ld`	`long`	`123456789`
`%f`	`float`	`3.14`
`%lf`	`double`	`2.71828`
`%c`	einzelnes Zeichen	`A`
`%s`	String (bis zum nächsten Whitespace)	`Hello`

Tabelle 4.3: Konvertierungsspezifikationen für scanf()

Eine vollständige Übersicht aller Konvertierungsspezifikationen gibt Anhang D.

Wichtig bei Gleitkommazahlen: `scanf()` schreibt mit `%f` in eine `float`-Variable und mit `%lf` in eine `double`-Variable. Die Konvertierung erfolgt entsprechend dem angegebenen Format und dem Zieltyp der Variable.

Ein häufiger Fehler ist, das `&` bei skalaren Variablen[1] zu vergessen. Dann schreibt `scanf()` in einen zufälligen Speicherbereich und verursacht undefiniertes Verhalten. Beispiel:

```
// 2QKSy.c
#include <stdio.h>

int main() {
  int number;
  scanf("%d", number);  // × Fehler: &number ist richtig
}
```

Der GCC-Compiler warnt bei `-Wall`:

```
<source>:5:13: warning: format '%d' expects argument of type 'int *', ↩
but argument 2 has type 'int' [-Wformat=]
    5 |     scanf("%d", number);
      |            ~^   ~~~~~~
      |             |   |
```

1 Skalare Variablen sind elementare Datentypen ohne interne Unterteilung in mehrere Elemente, etwa Ganzzahlen, Gleitkommazahlen oder Zeichen. Sie stehen im Gegensatz zu zusammengesetzten Typen wie Arrays oder Strukturen, die mehrere Werte zusammenfassen.

```
      |             |   int
      |             int *
<source>:5:5: warning: 'number' is used uninitialized [-Wuninitialized]
    5 |     scanf("%d", number);
      |     ^~~~~~~~~~~~~~~~~~
<source>:4:9: note: 'number' was declared here
    4 |     int number;
      |         ^~~~~~
```

Der Clang-Compiler warnt bei `-Wall`:

```
<source>:5:17: warning: format specifies type 'int *' but the argument ↩
has type 'int' [-Wformat]
    5 |     scanf("%d", number);
      |            ~~   ^~~~~~
<source>:5:17: warning: variable 'number' is uninitialized when used ↩
here [-Wuninitialized]
    5 |     scanf("%d", number);
      |                 ^~~~~~
<source>:4:15: note: initialize the variable 'number' to silence this warning
    4 |     int number;
      |               ^
      |                = 0
```

Die Warnungen deuten auf Format- und Initialisierungsprobleme hin. In Wahrheit steckt die Ursache aber schlicht darin, dass das `&` vor der Variablen fehlt. `scanf()` erwartet die Adresse, wohin das Ergebnis geschrieben werden soll, bekommt aber stattdessen den aktuellen Wert der Variablen übergeben.

scanf() Rückgabe

Die Funktion `scanf()` liefert als Rückgabewert die Anzahl der erfolgreich zugewiesenen Eingabeelemente. Entscheidend ist also die Zuweisung, nicht das bloße Lesen. Konvertierungen mit unterdrückter Zuweisung (`%*d`) und der Spezifizierer `%n` werden nicht mitgezählt.

- **Wenn alles klappt:** Für jedes korrekt gelesene und gespeicherte Eingabeelement erhöht sich der Rückgabewert. Bei `scanf("%d %f", &i, &f)` wird beispielsweise `2` zurückgegeben, wenn sowohl die Ganzzahl als auch die Gleitkommazahl erfolgreich zugewiesen wurden.
- **Zuordnungsfehler (matching failure):** Passt die Eingabe nicht zur erwarteten Konvertierung, bricht `scanf()` an dieser Stelle ab. Der Rückgabewert gibt an, wie viele Elemente bis dahin erfolgreich zugewiesen wurden. Das kann auch `0` sein.
- **Eingabefehler (input failure):** Tritt ein Dateiende, ein Lesefehler oder ein Kodierungsfehler auf, bevor die erste Konvertierung abgeschlossen wurde, liefert `scanf()` die Endekennung `EOF`. Tritt das Dateiende dagegen erst nach einer abgeschlossenen

Konvertierung auf, wird die Anzahl der bereits zugewiesenen Elemente zurückgegeben.

`0` und `EOF` bedeuten also Verschiedenes: `0` bedeutet, dass die Eingabe nicht zur erwarteten Konvertierung passte. `EOF` bedeutet, dass bereits vor der ersten abgeschlossenen Konvertierung keine weitere Eingabe gelesen werden konnte.

Wichtig

Den Rückgabewert von `scanf()` sollte man normalerweise überprüfen. Dabei vergleicht man mit der erwarteten Anzahl erfolgreicher Zuweisungen. Nach einem Zuordnungsfehler bleibt das Zeichen, an dem die Konvertierung gescheitert ist, im Eingabestrom. Soll der nächste Eingabeversuch mit einer neuen Zeile beginnen, kann man den Rest der aktuellen Zeile verwerfen. Im Buch verzichten wir häufig auf solche Prüfungen, um die Beispiele kompakt und lesbar zu halten. In echten Programmen sollte der Rückgabewert jedoch ausgewertet werden.

Mehrere Werte gleichzeitig einlesen

`scanf()` kann mehrere Werte in einem einzigen Aufruf einlesen. Dazu enthält der Formatstring mehrere Konvertierungsspezifikationen, zu denen die jeweiligen Variablenadressen übergeben werden. Die Eingaben werden dabei in der angegebenen Reihenfolge erwartet.

Beispiel:

```
// 3ktU.c
#include <stdio.h>

int main() {
  int a, b;
  float x;
  printf("Bitte zwei ganze Zahlen und eine Kommazahl eingeben: ");
  scanf("%d %d %f", &a, &b, &x);
  printf("Eingegeben: a=%d, b=%d, x=%.2f\n", a, b, x);
}
```

Eine mögliche Interaktion:

```
Bitte zwei ganze Zahlen und eine Kommazahl eingeben: 10 20 3.5
Eingegeben: a=10, b=20, x=3.50
```

Ein Leerzeichen im Formatstring steht nicht für genau ein Leerzeichen in der Eingabe, sondern für beliebig viele – auch gar keine – Whitespace-Zeichen. Benutzer können die Werte also durch Leerzeichen, Tabs oder Zeilenumbrüche trennen, und `scanf()` erkennt sie automatisch.

Die Leerzeichen in `"%d %d %f"` sind allerdings entbehrlich: Die numerischen Konvertierungen sowie `%s` überspringen führenden Whitespace ohnehin von selbst, `"%d%d%f"` verhält sich also identisch. Explizites Whitespace im Formatstring wird erst bei den Konvertierungen wichtig, die das nicht tun – `%c`, `%[...]` und `%n`.

Zu beachten ist außerdem das Dezimaltrennzeichen: In der Standard-Locale von C ist das der Punkt. Bei der Eingabe `3,5` liest `%f` daher nur die `3` ein; `,5` bleibt ungelesen im Eingabepuffer stehen.

Im Formatstring können neben Konvertierungsspezifikationen wie `%d` oder `%f` auch feste Zeichen vorkommen. Diese müssen exakt so in der Eingabe auftauchen, einschließlich Groß- und Kleinschreibung. Für Leerzeichen gilt dabei weiterhin die eben beschriebene Regel: Sie sind auch hier kein Literal, sondern passen auf beliebig viel – auch gar keinen – Whitespace.

Fester Text im Formatstring ist praktisch, wenn die Eingabe streng strukturiert ist, etwa bei Daten aus Dateien, Protokollen oder Schnittstellen zwischen Programmen. Damit kann geprüft werden, ob die Eingabe einem bestimmten Muster entspricht.

Beispiel:

```
// 219ns.c
#include <stdio.h>

int main() {
  int width, height;
  printf("Bitte im Format width=<Zahl> height=<Zahl> eingeben: ");
  scanf("width=%d height=%d", &width, &height);
  printf("Breite: %d, Höhe: %d\n", width, height);
}
```

Eine gültige Eingabe könnte lauten:

```
width=1920 height=1080
```

Da das Leerzeichen im Formatstring beliebig viel Whitespace zulässt, werden ebenso `width=1920height=1080`, `width=1920 height=1080` oder eine Eingabe mit Zeilenumbruch zwischen den beiden Angaben akzeptiert.

Weicht die Eingabe dagegen bei den festen Zeichen ab, scheitert das Einlesen teilweise oder vollständig. Bei `width=1920xheight=1080` wird `width` noch zugewiesen, das `x` passt aber nicht zum erwarteten `h`, und `scanf()` liefert `1`. Bei `Width=1920 height=1080` scheitert bereits das erste Zeichen, es wird nichts zugewiesen, und der Rückgabewert ist `0`. Für Benutzereingaben ist dieses Verhalten unflexibel.

Hinweis

Variadische Funktionen wie `printf()` können mit unterschiedlich vielen Argumenten aufgerufen werden. Für die Argumente im variadischen Teil greifen die soge-

nannten Default Argument Promotions: `char` und `short` werden zu `int` hochgestuft, `float` wird zu `double`. Das gilt jedoch nur für Werte, nicht für Zeiger. Darum funktioniert bei `printf()` die Konvertierungsspezifikation `%f` sowohl für einen ursprünglich als `float` als auch für einen als `double` übergebenen Wert – ein `float` ist bei der Übergabe bereits zu `double` konvertiert worden. (Seit C99 ist bei `printf()` auch `%lf` erlaubt und bedeutet dort dasselbe wie `%f`.) Bei `scanf()` ist das anders: Hier werden Zeiger übergeben, und die werden nicht angepasst. Deshalb muss die Konvertierungsspezifikation genau zum Typ des Arguments passen: Für `float *` verwendet man `%f`, für `double *` `%lf`.

Aufgabe

- Lies zwei Ganzzahlwerte in einem Aufruf von `scanf()` ein, und gib das Produkt als Ergebnis aus.
- Teste, was passiert, wenn nur ein Wert eingegeben wird.
- Nimm eine falsche Eingabe vor, und beobachte das Verhalten. Prüfe auch den Rückgabewert von `scanf()`.

4.4.4 Typableitung (Type Inference) (auto)

Dem Compiler muss zu jedem Zeitpunkt klar sein, welchen Typ eine Variable oder ein Ausdruck hat. Normalerweise schreiben wir diesen Typ explizit. Häufig kann der Compiler den Typ jedoch aus dem Initialisierer ableiten, zum Beispiel aus einer Ganzzahl, einem String-Literal oder dem Ergebnis einer Operation. Dieses automatische Ableiten von Typen nennt man **Type Inference** (Typableitung). Der abgeleitete Typ ist statisch und bleibt unveränderlich; die Typsicherheit bleibt damit erhalten.

In C ist eine Typdeklaration grundsätzlich zwingend notwendig. Allerdings können Typangaben schnell lang und unübersichtlich werden. Für solche Fälle stellt der Compiler seit C23 mit dem Schlüsselwort `auto` eine Möglichkeit zur Typableitung bereit.[2] Der Typ wird dabei aus dem Initialisierer abgeleitet, sodass die Deklaration kürzer bleibt, ohne die Typsicherheit zu verlieren. Ein Beispiel:

```
// pjJt.c
#include <stdio.h>

int main() {
  auto z = 69;       // Typ wird zu int
```

2 Das Schlüsselwort `auto` existiert zwar bereits seit frühen C-Versionen, hatte dort jedoch nur die Bedeutung einer Speicherklasse für lokale Variablen, die ohne Angabe ohnehin automatisch gilt und daher praktisch nie explizit notiert wird. Erst mit C23 wurde es um die Fähigkeit zur automatischen Typableitung erweitert.

```
  auto pi = 3.14159; // Typ wird zu double
  printf("z=%d, pi=%f\n", z, pi); // → z=69, pi=3.141590
}
```

In diesem Beispiel erkennt der Compiler den Typ von `z` als `int`, weil die Ganzzahlkonstante `69` den Typ `int` hat, und den Typ von `pi` als `double`, weil die Gleitkommakonstante ohne Suffix den Typ `double` hat.

`auto` steht allein als Spezifizierer und erfordert eine Initialisierung. Fehlt der Initialisierer, meldet der Compiler einen Constraint-Verstoß. Damit ist sichergestellt, dass jede mit `auto` deklarierte Variable stets einen wohldefinierten statischen Typ besitzt.

Beispiel:

```
// bkDD.c
int main() {
  auto x;   // Fehler: Es fehlt ein Initialisierer
  x = 12;   // hilft nicht mehr
}
```

Der GCC-Compiler meldet:

```
<source>:2:5: error: 'auto' requires an initialized data declaration
    2 |     auto x;
      |     ^~~~
<source>:3:5: error: 'x' undeclared (first use in this function)
    3 |     x = 12;
      |     ^
<source>:3:5: note: each undeclared identifier is reported only once for ↩
each function it appears in
```

`auto` ist vor allem bei langen oder verschachtelten Typen sinnvoll, sofern der abgeleitete Typ aus dem Initialisierer klar hervorgeht. Bei einfachen Typen wie `int` oder `double` bringt es kaum einen Vorteil. Ist der Initialisierer kompliziert, kann `auto` die Lesbarkeit sogar mindern, weil der Typ nicht unmittelbar sichtbar ist. Bei Funktionsparametern und Rückgabewerten kann `auto` in C23 nicht zur Typinferenz verwendet werden; ihre Typen müssen explizit angegeben werden.

Statische und dynamische Typisierung

C gehört zur Gruppe der **statisch typisierten** Sprachen: Der Typ einer Variablen ist beim Übersetzen festgelegt und bleibt unveränderlich. Damit überprüft der Compiler alle Typen bereits zur Übersetzungszeit. Zu den statisch typisierten Sprachen zählen auch C++, Java, Rust oder Go.

Davon unterscheiden sich **dynamisch typisierte** Sprachen wie Python, JavaScript, Ruby oder PHP. Hier wird der Typ einer Variablen erst zur Laufzeit bestimmt und kann sich während der Programmausführung ändern.

4.5 Konstante Werte

Beim Programmieren stößt man oft auf Werte, die sich während der gesamten Programmausführung nicht ändern sollen. Solche unveränderlichen Größen nennt man **Konstanten**. Typische Beispiele sind die Kreiszahl π, die Anzahl der Tage in einer Woche oder eine feste Feldgröße.

Der Vorteil von Konstanten liegt auf der Hand: Sie machen den Code robuster und verständlicher. Ein Wert, der als konstant markiert ist, kann nicht versehentlich überschrieben werden. Außerdem können Compiler mit Konstanten besser optimieren.

Bevor wir uns die Schreibweise für Konstanten anschauen, müssen wir zwei Begriffe unterscheiden:

- **Compilezeit (Übersetzungszeit):** Das ist der Moment, in dem der Compiler den Quellcode analysiert und in Maschinencode übersetzt. Alles, was er schon zu diesem Zeitpunkt berechnen kann, zählt zur Compilezeit. Heißt es etwa `int x = 3 + 5`, dann kann der Compiler `x` direkt auf `8` setzen.
- **Runtime (Laufzeit:)** Damit ist die Phase gemeint, in der das fertige Programm ausgeführt wird. Werte, die erst zur Laufzeit berechnet werden, können nicht zur Compilezeit bestimmt werden. Dazu gehören etwa Benutzereingaben, die Inhalte einer Datei oder Werte, die sich erst beim Programmstart aus der Umgebung ergeben, wie Zufallswerte.

In C kennzeichnen `const` und `constexpr` unveränderliche Werte. `const` schützt ein Objekt nach der Initialisierung vor Änderungen; der Initialwert darf auch erst zur Laufzeit entstehen. `constexpr` verlangt zusätzlich, dass der Initialisierer bereits zur Übersetzungszeit ausgewertet werden kann.

4.5.1 Konstante Werte (const)

Das Schlüsselwort `const` (nicht ganz überraschend von *constant*) ist ein sogenannter **Typqualifizierer** (engl. **type qualifier**). Es markiert einen Typ als schreibgeschützt: Eine `const`-Variable besitzt weiterhin eine Speicheradresse und kann normal gelesen, aber nach der Initialisierung nicht mehr verändert werden. Der Initialwert kann dabei auch erst zur Laufzeit bestimmt werden.

Der C-Standard bezeichnet solche Objekte als **konstante Variablen** (engl. **constant variables**). Dieser Begriff wirkt auf den ersten Blick widersprüchlich, ist aber normgerecht: Es handelt sich weiterhin um Variablen, die Speicher belegen, deren Wert jedoch nicht mehr verändert werden darf.

```
// 239o3.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
int main() {
  // Kreisberechnung
  const double pi = 3.14159;          // Konstante für Pi
  const double radius = 5.0;          // Beispielradius

  const double circumference = 2 * pi * radius; // Umfangsformel
  const double area = pi * radius * radius;     // Flächenformel

  printf("Radius = %.2f\n", radius);          // → Radius = 5.00
  printf("Umfang = %.2f\n", circumference);   // → Umfang = 31.42
  printf("Fläche = %.2f\n", area);            // → Fläche = 78.54

  // Initialisiert den Zufallsgenerator mit der aktuellen Zeit
  srand(time(nullptr));
  const int randomValue = rand(); // Zufallswert, schreibgeschützt
  printf("Zufallswert = %d\n", randomValue);
  // → Zufallswert = <zufälliger Wert>
}
```

In diesem Beispiel ist `pi` nach der Initialisierung unveränderlich und wird für die Berechnung von Umfang und Fläche eines Kreises verwendet. Der Radius ist als Konstante vorgegeben, und aus ihm werden `circumference` und `area` berechnet und ausgegeben. Zusätzlich wird eine Zufallszahl erzeugt: Auch `randomValue` ist nach der Initialisierung schreibgeschützt und kann nicht geändert werden, obwohl sein Wert erst während der Programmausführung bestimmt wird.

Ein versuchter Schreibzugriff wird vom Compiler abgeblockt. Ein Beispiel:

```
// Yt5z.c
int main() {
  const double pi = 3.14159;
  pi = 3.0;       // Fehler: Versuch, eine const-Variable neu zuzuweisen
}
```

GCC meldet:

```
<source>:3:8: error: assignment of read-only variable 'pi'
    3 |     pi = 3.0;
      |        ^
```

Wichtig

Bei `const`-Variablen muss der gewünschte Wert bei der **Definition** gesetzt werden:

```
int main() {
  const int ok = 69; // korrekt
  const int bad;     // zwar erlaubt, aber ...
```

```
  bad = 69;          // × Fehler: Zuweisung an Read-only-Variable
}
```

In dem Beispiel hat die lokale Variable `const int bad;` im Block einen unbestimmten Wert, den man nicht mehr ändern darf; damit ist sie faktisch nutzlos.

4.5.2 Konstant berechenbare Ausdrücke (constexpr)

Manche Werte lassen sich schon vollständig zur Übersetzungszeit berechnen. Das kann der Compiler nutzen, um den Code effizienter zu machen und bestimmte Sprachkonstrukte überhaupt erst zu ermöglichen.

Um dem Compiler mitzuteilen, dass ein Ausdruck zwingend schon beim Übersetzen ausgewertet werden soll, setzt man zusätzlich zum Typ das Schlüsselwort `constexpr`. Dieses Schlüsselwort gehört zur Gruppe der **Storage-Class-Specifier**, also Schlüsselwörter, die nicht den Typ selbst ändern, sondern zusätzliche Vorgaben für die Speicherung oder Auswertung einer Variablen machen. `constexpr` war lange nur in C++ verfügbar und wurde erst mit dem C23-Standard nach C übernommen.

Im Fall von `constexpr` bedeutet das: Die Variable muss direkt bei der Definition initialisiert werden, und der Initialisierer muss ein **konstanter Ausdruck** sein, also ein Ausdruck, der vollständig zur Übersetzungszeit ausgewertet werden kann, etwa eine Ganzzahlkonstante, eine Gleitkommakonstante oder ein daraus gebildeter konstanter Ausdruck.

`const` bedeutet zunächst nur Schreibschutz und ist nicht automatisch gleichbedeutend mit einer zur Übersetzungszeit bekannten Konstante.

Das vorherige Beispiel lässt sich auf die entscheidenden Unterschiede reduzieren:

```
// reWU.c
constexpr double pi = 3.14159;      // zur Compilezeit festgelegt
constexpr double radius = 5.0;      // ebenfalls zur Compilezeit bekannt
constexpr double circumference = 2 * pi * radius;
constexpr double area = pi * radius * radius;

const int randomValue = rand();
```

Die ersten vier Werte können bereits zur Übersetzungszeit berechnet werden. `randomValue` bleibt dagegen `const`, weil `rand()` erst zur Laufzeit ausgewertet wird.

Tipp

- Verwende `const`, wenn ein Wert erst zur Laufzeit bestimmt wird, danach aber unveränderlich bleiben soll.

- Verwende `constexpr`, wenn ein Wert bereits zur Übersetzungszeit berechnet werden kann oder wenn er in Kontexten benötigt wird, die zwingend konstante Ausdrücke verlangen.

Häufig werden Konstanten in C mit **Großbuchstaben** geschrieben, oft mit Unterstrichen, beispielsweise:

```
constexpr int MAX_USERS = 100;
const double PI = 3.14159;
```

Diese Konvention ist nicht vom Standard vorgeschrieben, aber weitverbreitet und macht Konstanten im Quelltext auf den ersten Blick erkennbar.

4.5.3 West gegen Ost const

In einer C-Deklaration stehen vor dem eigentlichen **Deklarator** (also dem Teil der Deklaration, der den Namen einer Variablen oder Funktion und ihre genauere Form festlegt) verschiedene **Spezifizierer** und **Qualifizierer**. Dazu gehören etwa Typspezifizierer wie `int` oder `double`, Storage-Class-Specifier wie `static` oder `constexpr` sowie Typqualifizierer wie `const` oder `volatile`.

Die Reihenfolge dieser Spezifizierer und Qualifizierer ist in C weitgehend frei, solange sie vor dem Deklarator stehen. Deshalb sind die folgenden Paare jeweils gleichwertig:

- `static const int x = 69;`
- `int static constexpr x = 69;`
- `const long volatile unsigned y = 69;`
- `unsigned volatile long const y = 69;`

In C kann `const` sowohl vor als auch nach dem Grundtyp stehen. Diese beiden Varianten werden oft als **west const** und **east const** bezeichnet.[3] Letztlich sind Deklarationen der Form `const Typ name` und `Typ const name` gleichbedeutend. In diesem Buch wird durchgehend die Schreibweise `const Typ name` verwendet.

4.5.4 Gültigkeitsbereich und Lebensdauer

Wenn wir im Quelltext eine Variable deklarieren, ist sie nicht automatisch überall im Programm zugänglich. Zwei Begriffe sind hier grundlegend:

- Der **Gültigkeitsbereich** einer Variablen beschreibt vereinfacht gesagt, **wo im Quellcode** sie sichtbar ist und verwendet werden darf. Es ist also eine reine Frage der Textstellen: Innerhalb welcher geschweiften Klammern `{...}` darf man den Bezeichner benutzen?

3 Die Bezeichnungen stammen aus einer Analogie zur Leserichtung: Im **west-const**-Stil steht das Schlüsselwort links vom Typ, im **east-const**-Stil rechts davon.

- Die **Lebensdauer** einer Variablen beschreibt dagegen, **wann während der Programmausführung** die Variable tatsächlich existiert. Sie gibt an, wann Speicher für die Variable bereitgestellt wird und wann er wieder freigegeben wird.

In C gilt für lokale Variablen im Blockscope: Sichtbar sind sie nur im Block, in dem sie deklariert wurden, und ihre Lebensdauer beginnt beim Eintritt in diesen Block und endet beim Verlassen.

```
// 14XS.c
#include <stdio.h>

int main() {
  int x = 10; // sichtbar im gesamten Rumpf von main,
              // existiert, solange main läuft

  {
    int y = 20; // sichtbar nur in diesem inneren Block
                // existiert nur während der Ausführung dieses Blocks
    printf("x = %d, y = %d\n", x, y); // → x = 10, y = 20
  } // Ende des Gültigkeitsbereiches für y; y wird zerstört

  // printf("%d\n", y); // × Compilerfehler: y ist hier nicht sichtbar
}
```

In diesem Beispiel gilt:

- `y` ist nur im inneren Block sichtbar (**Gültigkeitsbereich**) und wird beim Verlassen des Blocks zerstört (**Lebensdauer**).
- `x` ist im gesamten Funktionsrumpf sichtbar (**Gültigkeitsbereich**) und existiert, solange `main()` läuft (**Lebensdauer**).

Blöcke treten nicht nur in frei gewählter Form auf, sondern regelmäßig auch durch Kontrollstrukturen wie Fallunterscheidungen und Schleifen. Dort ist es besonders wichtig, Variablen lokal im jeweiligen Block zu halten, damit sie nur während der betreffenden Ausführung existieren und nicht ungewollt außerhalb sichtbar sind.

Wichtig

Gültigkeitsbereich und Lebensdauer sind unterschiedliche Konzepte. Eine Variable kann außerhalb ihres Gültigkeitsbereichs existieren (Speicher ist noch reserviert), aber wir dürfen nicht mehr legal auf sie zugreifen. Umgekehrt kann eine Variable sichtbar sein, aber ihre Lebensdauer noch gar nicht begonnen haben. Wir werden darauf noch einmal zurückkommen, z. B. bei `extern`-Deklarationen.

Gültigkeitsbereich und Lebensdauer lassen sich gezielt steuern. Mit `static` kann die Lebensdauer einer Variablen über mehrere Funktionsaufrufe hinweg erhalten bleiben, ob-

wohl ihr Gültigkeitsbereich lokal bleibt. Außerdem können Variablen auch außerhalb von Funktionen deklariert werden; sie sind dann global sichtbar und haben eine entsprechend längere Lebensdauer. Das ist sinnvoll, wenn Werte über Funktionsgrenzen hinweg erhalten bleiben oder von mehreren Programmteilen gemeinsam genutzt werden sollen. Beides behandeln wir später ausführlich in Kapitel 8.

4.6 Ganzzahlen (int, short, long, long long)

C stellt eine kleine, aber bewusst unterschiedliche Auswahl an Grundtypen bereit. Sie unterscheiden sich in Wertebereich, Darstellungsbreite und Speicherbedarf und damit darin, wofür sie sich eignen.

Die folgenden Unterkapitel ordnen die Typen ein, erklären die üblichen Datenmodelle der Plattformen und zeigen, wie sich Vorzeichen, Bitbreite und Darstellung (insbesondere das seit C23 verpflichtende Zweierkomplement) auf Verhalten, Grenzwerte und Portabilität auswirken.

Beginnen wir mit den Ganzzahltypen. Sie dienen dazu, ganze Zahlen wie `0`, `69` oder `-7654321234567` zu speichern, also Werte ohne Nachkommastellen. Damit lassen sich etwa Zähler für Schleifen, Altersangaben oder Stückzahlen darstellen.

C unterscheidet grundsätzlich zwei Familien von Ganzzahltypen: solche mit implementierungsdefinierter Breite und solche mit exakt festgelegter Breite.

4.6.1 Ganzzahltypen mit implementierungsdefinierter Bitbreite

Ganzzahltypen sind in verschiedenen Bitbreiten definiert, wobei die Norm Mindestanforderungen vorgibt. Standardmäßig sind Ganzzahldatentypen **vorzeichenbehaftet** (engl. signed), d. h., sie können sowohl positive als auch negative Werte darstellen. Es gibt aber auch **vorzeichenlose** (engl. unsigned) Varianten, die nur den Wert 0 und nichtnegative Werte abbilden.

Langform	Kurzform	Mindestbreite (Bits)	Garantierter Wertebereich (vorzeichenbehaftet)
`short int`	`short`	16	mindestens -32768 … +32767
`int`	–	16	mindestens -32768 … +32767
`long int`	`long`	32	mindestens -2147483648 … +2147483647
`long long int`	`long long`	64	mindestens -9223372036854775808 … +9223372036854775807

Tabelle 4.4: Standard-Ganzzahltypen in C mit garantierten Mindestgrößen

Die Kurzformen sind im C-Standard gleichbedeutend mit den Langformen. Es besteht keinerlei Unterschied in Bedeutung oder Verhalten, jedoch werden die Kurzformen häufiger verwendet.

`long long` wurde mit C99 eingeführt, um eine standardisierte Möglichkeit für Ganzzahlen mit mindestens 64 Bit Breite zu schaffen. Damit trug der Standard auch der wachsenden Bedeutung Rechnung, größere Wertebereiche und die auf modernen Maschinen zunehmend verfügbaren größeren Wortbreiten portabel abzubilden. Der Standard erlaubt keine zusätzlichen Erweiterungen wie `long long long time_ago`.

Das folgende Programm deklariert für verschiedene Ganzzahltypen eine Variable und gibt sie mit den jeweils passenden Formatzeichen von `printf()` aus.

```
// segt.c
#include <stdio.h>

int main() {
  short s = 100;
  int i = 100000;
  long l = 1000000000;
  long long ll = 1000000000000LL;

  printf("short: %hd\n", s);          // → short: 100
  printf("int: %d\n", i);             // → int: 100000
  printf("long: %ld\n", l);           // → long: 1000000000
  printf("long long: %lld\n", ll);    // → long long: 1000000000000
}
```

Das Suffix `LL` gibt an, dass die Ganzzahlkonstante als `long long int` typisiert werden soll. Ohne dieses Suffix wird ihr Typ anhand des Werts nach den Regeln für Ganzzahlkonstanten bestimmt.

Die tatsächliche Breite kann größer sein, abhängig von der Plattform. Wichtig ist, dass die oben genannten Mindestgrößen garantiert sind. Außerdem schreibt der Standard eine Größenhierarchie vor: `short` ist nie breiter als `int` und `int` nie breiter als `long`. Daraus folgt etwa, dass `int` theoretisch auch nur 16 Bit breit sein kann, nämlich genau dann, wenn die Plattform `short` und `int` gleich groß macht. In der Praxis ist `int` auf modernen Plattformen fast immer 32 Bit breit, aber portable Programme dürfen das nicht voraussetzen.

Mit `_BitInt` gibt es einen weiteren Ganzzahltyp mit frei wählbarer Bitbreite, den wir später in Abschnitt 18.4 gesondert betrachten.

Hinweis

Die Festlegung auf Mindestgrößen statt auf exakt bestimmte Bitbreiten ist aus heutiger Sicht ungewöhnlich, historisch aber erklärbar: C sollte auf sehr unterschiedlichen Rechnerarchitekturen effizient laufen.

Neuere Programmiersprachen setzen dagegen häufig stärker auf feste Breiten. Auch C bietet heute über `<stdint.h>` Typen mit Mindestbreite und, sofern unterstützt, exakt definierter Breite wie `int32_t`. Dazu gleich mehr.

Ausgeblendet aus den Beispielen ist der Datentyp `char`, den es vorzeichenlos und vorzeichenbehaftet gibt. Wir werden über diesen Datentyp im eigenen Abschnitt 4.8 sprechen.

4.6.2 Datenmodelle

Auf realen Plattformen werden die in der Norm geforderten Mindestgrößen durch konkrete **Datenmodelle** umgesetzt. Ein Datenmodell beschreibt, wie die elementaren Ganzzahltypen (`int`, `long`, `long long`) und Zeiger auf einer bestimmten Architektur konkrete Speicherbreiten erhalten.

Für 64-Bit-Systeme haben sich verschiedene Modelle etabliert. Diese Modelle werden mit Kürzeln bezeichnet. Dabei gilt:

- **L** steht für das Schlüsselwort `long`,
- **P** steht für das englische Wort pointer (Zeiger),
- die Zahlen 32 oder 64 geben die jeweilige Bitbreite an.

So spricht man vom **LP64-Modell**, wenn `long` und Zeiger 64 Bit breit sind, und vom **LLP64-Modell**, wenn `long long` und Zeiger 64 Bit breit sind.

Typ	Standard-Mindestgröße	LP64 (Linux/Unix, 64-Bit)	LLP64 (Windows, 64-Bit)	16-Bit-Mikrocontroller (z. B. AVR)
`short`	16 Bit	16 Bit	16 Bit	16 Bit
`int`	16 Bit	32 Bit	32 Bit	16 Bit
`long`	32 Bit	64 Bit	32 Bit	32 Bit
`long long`	64 Bit	64 Bit	64 Bit	64 Bit

Tabelle 4.5: Typische Breiten von Ganzzahltypen auf verschiedenen Plattformen

Diese Unterschiede zeigen, dass portable Programme in C keine festen Annahmen über die Größe bestimmter Grundtypen treffen sollten. Es gibt jedoch Möglichkeiten, Typen mit exakt festgelegter Bitbreite zu verwenden.