

Christian Ullenboom

Aktuell
zu C23



C programmieren

Das umfassende Handbuch

- ▶ Moderne C-Programmierung für die professionelle Praxis
- ▶ Inkl. Sprachgrundlagen, Speicherverwaltung, Präprozessor, Metaprogrammierung u. v. m.
- ▶ Ideal für Studium und Beruf, mit Aufgaben und Lösungen



Alle Beispielprojekte zum Download



Rheinwerk
Computing

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 Ausgabeprecision, 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	<code>int</code> , <code>short</code> , <code>long</code> , <code>long long</code>	ganze Zahlen ohne Nachkommastellen, z. B. Zähler, Altersangaben, Stückzahlen
Gleitkommazahlen	<code>float</code> , <code>double</code> , <code>long double</code>	Zahlen mit Nachkommastellen, z. B. Längen, Preise, physikalische Größen
Zeichen	<code>char</code>	einzelne Zeichen wie 'A', 'z' oder '#'; intern als Zahl gespeichert
Wahrheitswerte	<code>bool</code>	logische Werte: wahr oder falsch; praktisch für Bedingungen in Programmen
Spezieller Typ	<code>void</code>	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.

```
// 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 Variablen¹ 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;          // Beispieleradius

    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(NULL));
    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)
<code>short</code>	16 Bit	16 Bit	16 Bit	16 Bit
<code>int</code>	16 Bit	32 Bit	32 Bit	16 Bit
<code>long</code>	32 Bit	64 Bit	32 Bit	32 Bit
<code>long long</code>	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.

Inhalt

Materialien zum Buch	31
Vorwort	33
1 Die Philosophie von C	41
1.1 Was ist C, und was ist es nicht?	41
1.1.1 Ursprung und Motivation	41
1.1.2 Was C ausmacht	43
1.1.3 Was C nicht ist	43
1.1.4 C im Vergleich zu C++	44
1.2 Der C-Standard	44
1.2.1 Versionen von C	45
1.2.2 Die C-Standardbibliothek und ihre Header	47
1.2.3 Ein Kommen und Gehen	50
1.2.4 Drafts	50
1.2.5 Gemeinsame Wurzeln und unterschiedliche Entwicklung von C und C++	50
1.2.6 Abseits des Standards	51
1.3 Die Bedeutung von C heute	52
1.3.1 Warum C immer noch so wichtig ist	53
1.3.2 C als Fundament der digitalen Welt	53
1.3.3 Aktuelle Popularität	54
1.4 Rückblick und Ausblick	55
2 Werkzeugkette und das erste C-Programm	57
2.1 Minimales Beispiel main.c	57
2.1.1 Ein kleiner Exkurs zur Programmsyntax	59
2.1.2 Host-Umgebungen und Freestanding-Umgebungen	60
2.2 Compiler im Alltag: GCC und Clang	60
2.2.1 GNU Compiler Collection (GCC)	62
2.2.2 LLVM und Clang	65
2.2.3 Compiler in Containern	66
2.2.4 Der Microsoft-Compiler (MSVC/cl)	66
2.3 C-Programm auf der Kommandozeile kompilieren und ausführen	68
2.3.1 Sprachunterstützung der Compiler	69
2.3.2 Sprachstandard explizit festlegen (ISO C)	71

2.3.3	Warnungen einschalten	71
2.3.4	GNU-Standard	72
2.4	Präprozessor, Compiler und Linker	73
2.4.1	Übersetzungsphasen explizit ausführen (GCC/Clang)	74
2.5	Editoren, IDEs	74
2.5.1	IDEs und Editoren im Überblick	76
2.5.2	Webbasierte IDEs	77
2.5.3	Compiler Explorer	78
2.6	Debugger und Laufzeitanalyse	80
2.6.1	Debug vs. Release	80
2.6.2	Breakpoints, Stepping, Watchpoints	81
2.6.3	Remote Debugging	82
2.7	Dokumentation zu C-Funktionen finden	82
2.8	Aufgabe	82
2.9	Rückblick und Ausblick	83
3	Syntax und Struktur von C-Programmen	85
3.1	Vom Quelltext zum Programm	85
3.1.1	Translation Units und Header-Dateien	85
3.2	Lexikalische Grundlagen	86
3.2.1	Schlüsselwörter	86
3.2.2	Bezeichner	87
3.2.3	Unicode und UTF-8	87
3.2.4	Konstanten und String-Literale	87
3.2.5	Punctuators und Operatoren	88
3.2.6	Kommentare	88
3.2.7	Aufgabe	90
3.3	Lesbarkeit und Stil	90
3.4	Grammatik und Programmbausteine	91
3.4.1	Deklarationen und Definitionen	91
3.4.2	Anweisungen und Blöcke	92
3.4.3	Das Semikolon	92
3.5	Rückblick und Ausblick	93

4	Variablen, Datentypen und Typsicherheit	95
4.1	Konstanten und String-Literale: feste Angaben im Quelltext	95
4.1.1	Ganzzahlkonstanten, Gleitkommakonstanten, Zeichenkonstanten und String-Literale	95
4.1.2	Literalverkettung	96
4.2	Werte mit der Funktion zur formatierten Ausgabe printf() ausgeben	97
4.2.1	Nicht passende Konvertierungsspezifikationen	98
4.2.2	Dezimale Gleitkommakonstanten mit Exponent	99
4.2.3	Aufgabe	99
4.3	Grundtypen	99
4.4	Variablen	100
4.4.1	Deklaration von Variablen	100
4.4.2	Deklaration mit Initialisierung	104
4.4.3	Werte einlesen (scanf)	105
4.4.4	Typableitung (Type Inference) (auto)	110
4.5	Konstante Werte	112
4.5.1	Konstante Werte (const)	112
4.5.2	Konstant berechenbare Ausdrücke (constexpr)	114
4.5.3	West gegen Ost const	115
4.5.4	Gültigkeitsbereich und Lebensdauer	115
4.6	Ganzzahlen (int, short, long, long long)	117
4.6.1	Ganzzahltypen mit implementierungsdefinierter Bitbreite	117
4.6.2	Datenmodelle	119
4.6.3	Ganzzahltypen mit fester, exakt definierter Breite	120
4.7	Gleitkommatypen (float, double, long double)	120
4.7.1	Spezielle Werte: $\pm\infty$, NaN und vorzeichenbehaftete Nullen	121
4.7.2	Stillschweigend gerundet	123
4.8	Zeichentyp (char)	123
4.8.1	Escape-Sequenzen	125
4.8.2	Hexadezimale Escape-Sequenzen	126
4.8.3	Mehrzeichen-Zeichenkonstante	127
4.8.4	Wie der C-Standard ein Byte bestimmt	127
4.9	Wahrheitswerte (bool)	128
4.9.1	Wie C lernte, wahr und falsch zu unterscheiden	129
4.10	Der Nicht-Wertetyp (void)	129
4.11	Vorzeichenbehaftete und vorzeichenlose Ganzzahlen	130
4.11.1	Einführung in vorzeichenbehaftete und vorzeichenlose Typen	130
4.11.2	Sonderfall char	132

4.11.3	Wertebereiche und Darstellung	133
4.11.4	Zweierkomplementdarstellung	135
4.11.5	Überlaufverhalten	136
4.11.6	Mischen von vorzeichenbehafteten und -losen Datentypen	138
4.12	Zahlendarstellungen und Schreibweisen	140
4.12.1	Oktal, Hexadezimal und Binär	140
4.12.2	Suffixe für Ganzzahl- und Gleitkommakonstanten (U, L, f ...)	141
4.12.3	Zifferntrenner	144
4.12.4	Aufgabe	145
4.13	Typkategorien und Grundbegriffe	145
4.14	Rückblick und Ausblick	145
5	Operatoren und Ausdrücke	147
5.1	Primäre Ausdrücke	148
5.2	Zuweisungsoperatoren	149
5.2.1	Einfache Zuweisung	149
5.2.2	L-Values und R-Values	150
5.2.3	Zuweisung als Ausdruck	151
5.3	Arithmetische Operatoren (+, -, *, /, %)	152
5.3.1	Unäre und binäre Operatoren	152
5.3.2	Grundlegende arithmetische Operatoren	153
5.3.3	Restwertberechnung	155
5.3.4	Aufgaben	157
5.4	Kurzschreibweisen für Zuweisungen und Änderungen	157
5.4.1	Erweiterte Zuweisungen	157
5.4.2	Präfixformen und Postfixformen	158
5.5	Vergleichsoperatoren	160
5.5.1	Gleichheitsoperatoren (==, !=)	161
5.5.2	Relationale Operatoren (<, <=, >, >=)	164
5.5.3	Vergleiche mit Gleitkommazahlen	165
5.6	Logische Operatoren (!, &&,)	165
5.6.1	Der Negationsoperator (!)	166
5.6.2	Der UND-Operator (&&)	167
5.6.3	Der ODER-Operator ()	167
5.6.4	Zusammengesetzte logische Ausdrücke	168
5.6.5	Abgeleitete logische Operatoren	169
5.6.6	Kurzschlussverhalten	170

5.6.7	Arithmetische Tricks mit booleschen Werten	172
5.6.8	Aufgabe	173
5.7	Typumwandlungen (Conversions) und ihre Regeln	174
5.7.1	Integer-Promotion (Ganzzahlpromotion)	174
5.7.2	Implizite Konvertierungen (Automatic Conversions)	175
5.7.3	Explizite Konvertierungen (Casts)	178
5.7.4	Das Signed/Unsigned-Problem	179
5.7.5	Aufgaben	181
5.8	Kommaoperator, Vorrang, Assoziativität und Auswertungs- reihenfolge	181
5.8.1	Kommaoperator	182
5.8.2	Operatorpräzedenz	183
5.8.3	Assoziativität	183
5.8.4	Auswertungsreihenfolge (Sequenzierung)	185
5.9	Mathematisches aus den Standard-Bibliotheken	186
5.9.1	Mathematische Funktionen in <stdlib.h>	187
5.10	Rückblick und Ausblick	188
6	Kontrollstrukturen	189
6.1	Bedingte Anweisungen (Fallunterscheidungen: if, else)	189
6.1.1	Komplexe Bedingungen	191
6.1.2	Typische Fehler	192
6.1.3	if-else-Anweisung	197
6.1.4	else-if-Ketten für mehrere Bedingungen	199
6.1.5	Aufgaben	200
6.2	Bedingungsoperator (?:)	201
6.2.1	Verschachtelter Bedingungsoperator	203
6.2.2	Beispiel: Werte einklemmen	204
6.2.3	Bedingte Ausdrücke mit ausgelassenem Mittelteil (GCC)	205
6.3	switch-Anweisung	205
6.3.1	Default-Label	208
6.3.2	switch oder if?	209
6.3.3	Fall-through	209
6.3.4	Gültigkeitsbereich von Variablen	214
6.3.5	Aufgabe	215
6.4	while-Schleife	216
6.4.1	Endlosschleifen	217

6.4.2	Zuweisungen in der Bedingung	217
6.4.3	Aufgaben	221
6.5	do-while-Schleife	222
6.5.1	Konvertierung zwischen while und do-while	223
6.5.2	Aufgaben	224
6.6	for-Schleife	225
6.6.1	Varianten und leere Teile der for-Schleife	227
6.6.2	Mehrere Operationen mit dem Kommaoperator	228
6.6.3	Geschachtelte Schleifen	229
6.6.4	Aufgabe	231
6.7	Schleifen mit break und continue steuern	231
6.7.1	break	232
6.7.2	continue	232
6.7.3	Grenzen von break und continue	233
6.7.4	JSON-Dokumente formatieren (Pretty Printing)	234
6.7.5	Aufgabe	238
6.8	Best Practices, Tipps und Fehlerquellen bei Schleifen	239
6.8.1	Defensive Programmierung	239
6.8.2	Häufiger Fehler: Off-by-one-Fehler	240
6.8.3	Wahl der Schleifenart	242
6.8.4	Schleifenvariablen und Invarianten	242
6.8.5	Performance-Überlegungen	243
6.8.6	Abschlusstipps	245
6.9	Programmsprünge (goto)	246
6.9.1	Warum man bei goto aufpassen muss	247
6.9.2	goto zur Fehlerbehandlung	249
6.9.3	Aus verschachtelten Schleifen ausbrechen	249
6.9.4	Doppeldeutigkeit von break bei switch und Schleifen	250
6.9.5	Zustandsautomaten	251
6.10	Rückblick und Ausblick	253
7	Funktionen	255
7.1	Warum Funktionen?	255
7.1.1	Vorteile eigener Funktionen	256
7.2	Funktionsgrundlagen: Definition, Deklaration und Aufruf	257
7.2.1	Erstes Beispiel: eine Banner-Funktion	257
7.2.2	Benennung von Funktionen	258

7.2.3	Definition, Deklaration und Funktionsprototyp	259
7.2.4	Eine Definition pro Funktion	262
7.3	Parameterübergabe	263
7.3.1	Parameter und Argumente	264
7.3.2	Call by Value	266
7.3.3	Übliche Typkonvertierungen	266
7.4	Funktionen mit return verlassen	267
7.5	Funktionen mit Rückgabewerten	267
7.5.1	Kompakte Rümpfe bei boolean-Rückgaben	269
7.5.2	Was passiert, wenn ein Rückgabewert fehlt?	269
7.6	Überladen von Funktionen in C?	270
7.7	Rekursive Funktionen	272
7.7.1	Das Stack-Modell	273
7.7.2	Endlosrekursion	274
7.7.3	Weitere Rekursionsprobleme	274
7.7.4	Endrekursion (Tail Recursion) und Optimierung	275
7.7.5	Rekursion oder Iteration?	275
7.7.6	Aufgaben	275
7.8	Variadische Funktionen (variable Argumentlisten)	276
7.8.1	Syntax mit	277
7.8.2	Makros aus <stdarg.h>	277
7.8.3	Typinformation und Default Promotions	279
7.8.4	Anzahl und Typen der variablen Argumente erkennen	279
7.8.5	Stellenwert variadischer Funktionen	279
7.8.6	Aufgabe	279
7.9	Inline-Funktionen	280
7.9.1	Das inline-Schlüsselwort als Compiler-Hinweis	280
7.9.2	Lokale Hilfsfunktionen mit static inline	281
7.10	Attribute	282
7.10.1	Attribut-Syntax	282
7.10.2	Standard-Attribute	283
7.11	Fehlerbehandlung in Funktionen	287
7.11.1	Rückgabewerte als Fehlerindikatoren	288
7.11.2	Programm mit Fehlercode beenden	290
7.11.3	Fehlerbehandlung in C mit errno	290
7.11.4	Ein-/Ausgabe-Kanäle und Fehlermeldungen	292
7.11.5	Fehlermeldungen mit perror() und strerror()	293
7.11.6	Das Makro assert	295
7.12	Rückblick und Ausblick	298

8	Scope, Linkage, Storage Duration und Modularisierung	299
8.1	Scope, Linkage und Storage Duration von Variablen und Funktionen ...	300
8.1.1	Globale Variablen	301
8.1.2	Statische globale Variablen	305
8.1.3	Statische lokale Variablen	306
8.1.4	Globale Funktionen	307
8.1.5	Zusammenfassung: Sichtbarkeit (Scope) und Linkage	309
8.1.6	Storage Duration	310
8.1.7	Aufgabe	311
8.2	Storage Classes im Überblick	312
8.2.1	extern bei Funktionen	313
8.2.2	extern bei Variablen	315
8.2.3	auto	318
8.2.4	register	319
8.2.5	Aufgabe	321
8.3	Header-Dateien und Modularisierung	321
8.3.1	Module bilden	322
8.3.2	Funktionsprototyp im Header, Implementierung in .c-Dateien	323
8.3.3	#include und Präprozessor	325
8.3.4	#include <...> vs. #include "..."	327
8.3.5	Include-Pfade und -I	328
8.3.6	Header Best Practices	329
8.3.7	Aufgabe	343
8.4	Strukturierung größerer C-Anwendungen	344
8.4.1	Vom Problem zur Struktur	344
8.4.2	Module und Kapselung	344
8.4.3	Datenorientierter Entwurf	344
8.4.4	Abhängigkeiten und Architektur	345
8.4.5	Funktionen und Benennung	346
8.4.6	Evolution der Struktur	346
8.5	Rückblick und Ausblick	347
9	Der C-Präprozessor	349
9.1	Historische Entwicklung und Motivation	349
9.1.1	Rolle in den Übersetzungsphasen	350
9.1.2	Erstes Makro	350

9.1.3	Verwendung außerhalb von C	355
9.1.4	C++ und der Präprozessor	355
9.2	Makros definieren: #define	356
9.2.1	Wenn möglich, dann constexpr	356
9.2.2	Syntax-Makros	357
9.2.3	Konstanten-Makros	360
9.2.4	Funktionsmakros	361
9.2.5	Funktionsmakros mit Parametern	366
9.2.6	Makros entfernen: #undef	382
9.2.7	Funktionen per Makro umleiten	383
9.2.8	X-Makros: Listen einmal definieren, mehrfach verwenden	387
9.3	Bedingte Kompilierung	388
9.3.1	#if, #elif, #else und #endif	388
9.3.2	Der defined-Operator	390
9.3.3	#ifdef und #ifndef	391
9.3.4	#elifdef und #elifndef	392
9.4	Vordefinierte Makros	394
9.4.1	Datei- und Zeileninformationen	394
9.4.2	Compile-Datum und -Zeit	396
9.4.3	Standard- und Umgebungsinformationen	397
9.4.4	Unterstützung optionaler Sprach- und Bibliotheks- erweiterungen	398
9.5	Compilerabhängige Erweiterungen	399
9.5.1	Statement-Expressions	399
9.5.2	Compilerspezifische und plattformabhängige Makros	401
9.5.3	Beispiel: portabler Einsatz von Attributen mit bedingter Kompilation	402
9.5.4	Plattformspezifische Kompilierung	403
9.5.5	Unterscheidung zwischen C und C++	404
9.6	Rückblick und Ausblick	405
10	Benutzerdefinierte und erweiterte Datentypen	407
10.1	Typ-Alias definieren (typedef)	407
10.2	Ganzzahltypen aus <stdint.h>	409
10.2.1	Ganzzahlen mit fester Breite (intN_t und uintN_t)	410
10.2.2	Minimalbreiten-Ganzzahlen (int_leastN_t und uint_leastN_t)	415
10.2.3	Schnellste Ganzzahlen (int_fastN_t und uint_fastN_t)	417
10.2.4	Ganzzahltypen größter Breite (intmax_t und uintmax_t)	418

10.2.5	Portable Format-Makros für Ein- und Ausgabe	419
10.2.6	Zusammenfassung	423
10.3	Typen aus <stddef.h>	424
10.3.1	size_t	424
10.4	Optionale Gleitkomma-Datentypen	425
10.4.1	Warum es optionale Datentypen gibt	426
10.4.2	FloatN-Typen: feste IEEE-754-Formate für Portabilität	429
10.4.3	_FloatNx-Typen: erweiterte Formate für höhere Präzision	432
10.4.4	Dezimale Gleitkommazahlen (_Decimal32, _Decimal64, _Decimal128)	433
10.4.5	Fallback-Strategien	434
10.4.6	Gibt es Float-Aliase (<stdfloat.h>) in C23?	434
10.5	Größen, Grenzen und Eigenschaften der Typen	434
10.5.1	Messwerkzeuge zur Ermittlung von Typ-Eigenschaften	435
10.5.2	Grenzen für Ganzzahltypen	436
10.5.3	Portabel definierte Ganzzahltypen	437
10.6	Rückblick und Ausblick	438
11	Komplexe Datentypen	439
11.1	Aufzählungen (enum)	439
11.1.1	Definition von Aufzählungen	439
11.1.2	Variablen eines Aufzählungstyps deklarieren und verwenden	440
11.1.3	Aufzählungen in switch-case	441
11.1.4	Namensräume und Sichtbarkeit von Aufzählungskonstanten	441
11.1.5	Verwendung als Parameter und Rückgabewert	443
11.1.6	Automatische Wertezuweisung und Vergleich	444
11.1.7	Explizite Wertzuweisungen	446
11.1.8	Sentinel-Werte in Aufzählungen	448
11.1.9	Anonyme enum-Deklarationen	450
11.1.10	Aufzählungskonstanten als konstante Ausdrücke und frühere Nutzungsmuster	451
11.1.11	Implizite und feste Basistypen	452
11.1.12	Vorwärtsdeklaration von Aufzählungen	454
11.1.13	Vereinfachung von enum-Nutzung durch typedef	455
11.2	Strukturen (struct)	457
11.2.1	Deklaration von Strukturen	458
11.2.2	Anlegen von Variablen vom Typ einer Struktur	460
11.2.3	Zugriff auf Strukturelemente	463
11.2.4	Wertweise Kopiersemantik	463

11.2.5	Initialisierung von Strukturen	469
11.2.6	Anonyme Strukturen und verschachtelte Typen	474
11.2.7	Vereinfachung von Strukturnutzung durch typedef	476
11.2.8	Strukturtypen in verschiedenen Translation Units	482
11.2.9	Aufgabe	483
11.3	Unions (union)	484
11.3.1	Aufbau und Definition einer Union	485
11.3.2	Anlegen von Variablen vom Typ einer Union	486
11.3.3	Zugriff und Übergabe an Funktionen	486
11.3.4	Anonyme Union	488
11.3.5	Union mit typedef	488
11.3.6	Variantentypen	489
11.3.7	Strukturen und Unions kombinieren	492
11.3.8	Tagged Unions	496
11.3.9	Variantentypen für Rückgabewerte (Result-Muster)	499
11.3.10	Type Punning mit Strukturen und Unions	502
11.3.11	Anonyme Unions und Kombination mit Strukturen	506
11.3.12	Aufgabe	510
11.4	Rückblick und Ausblick	511
12	Arrays	513
12.1	Einführung und Motivation für Arrays	513
12.2	Deklaration von Arrays	513
12.2.1	Arrays mit fester Größe (Compile-Time-Größe)	514
12.2.2	Arrays mit variabler Größe (Variable-Length Arrays, VLAs)	516
12.3	Zugriff auf Array-Elemente	519
12.3.1	Indexnotation	519
12.3.2	Über Arrays iterieren	520
12.3.3	Out-of-Bounds-Zugriffe und Undefined Behavior	521
12.3.4	Sichere Praktiken und Debugging	522
12.3.5	Grenzchecks mit AddressSanitizer	522
12.4	Initialisierung von Arrays	523
12.4.1	Listeninitialisierung	524
12.4.2	Teilinitialisierung und implizite Nullen	529
12.4.3	Standardinitialisierung mit {0}	530
12.4.4	Zeichenketten als char-Arrays	532
12.4.5	Designated Initializers	534
12.4.6	const Arrays	537

12.5	Arrays bei der Parameterübergabe und Rückgabe von Funktionen	540
12.5.1	Arrays an Funktionen übergeben	540
12.5.2	Variable Length Arrays in Funktionsparametern	541
12.5.3	Grenzen von sizeof bei Funktionsparametern	542
12.5.4	Mit const qualifizierte Parameter	543
12.5.5	Minimale Größengarantie mit static	543
12.5.6	Arrays als Rückgabewert: Einschränkungen und Alternativen	547
12.6	Aufgaben	550
12.7	Mehrdimensionale Arrays	554
12.7.1	Deklaration mehrdimensionaler Arrays	555
12.7.2	Variable Length Arrays (VLA) mit mehreren Dimensionen	555
12.7.3	Zugriff mit mehreren Indizes	555
12.7.4	Mehrdimensionale Arrays sind Arrays von Arrays	556
12.7.5	Mehrdimensionale Arrays liegen im Speicher hintereinander	557
12.7.6	sizeof bei mehrdimensionalen Arrays	558
12.7.7	Initialisierung von mehrdimensionalen Arrays	561
12.7.8	Mehrdimensionale Arrays an Funktionen übergeben	566
12.7.9	Praktische Beispiele mit mehrdimensionalen Arrays: Tabellen, Brettspiele und mehr	569
12.8	Einbinden externer Inhalte	576
12.9	Rückblick und Ausblick	578
13	Zeiger	579
13.1	Datenzeiger und Adressen	579
13.1.1	Speicher und Variablen	579
13.1.2	Adressoperator (&)	582
13.1.3	Zeigervariablen: Adressen speichern	583
13.1.4	Weitere zulässige Anwendungen des Adressoperators	585
13.1.5	Dereferenzierungsoperator (*)	586
13.1.6	Beispiel: Menüsystem mit Umschalten per Zeiger	588
13.1.7	Deklaration mehrerer Zeigervariablen	589
13.1.8	Typumwandlungen zwischen Zeigertypen	591
13.1.9	Das Schlüsselwort volatile	593
13.1.10	Zugriff auf Speicherstellen: Memory-Mapped I/O	593
13.1.11	Zeiger als Ganzzahl interpretieren sowie der Typ uintptr_t	596
13.1.12	Pointer Tagging	597
13.1.13	Aufgabe	599
13.2	Der Null-Pointer nullptr	599
13.2.1	Dereferenzierung einer Null-Pointer-Variablen	600

13.2.2	Zeigervariablen auf nullptr prüfen	600
13.2.3	Typ nullptr_t	601
13.2.4	Das Makro NULL	602
13.3	Zeiger und const: Unveränderlichkeit deklarieren	602
13.3.1	Zeiger auf konstante Daten	603
13.3.2	Konstanter Zeiger	605
13.3.3	Konstanter Zeiger auf konstante Daten	605
13.3.4	Zusammenfassung und Merkregel: const und Zeiger	606
13.4	Zeiger in Funktionen	607
13.4.1	Wertübergabe und Adressübergabe	607
13.4.2	Beispiel: swap()-Funktion mit Zeigern	608
13.4.3	Mehrere Rückgabewerte über Out-Parameter	609
13.4.4	Terminologie: In-, Out- und In-Out-Parameter	610
13.4.5	Ergebnisse über Out-Parameter und Status über Rückgabewerte	610
13.4.6	Grund für Zeigerübergabe bei scanf()	611
13.4.7	Zeiger als Rückgabewert	612
13.4.8	Häufige Fehler: Dangling-Pointers	614
13.4.9	Zeiger auf Funktionsparameter zurückgeben	615
13.4.10	Der Qualifizierer restrict: Aliasfreiheit und Optimierung	616
13.4.11	Exkurs: Referenzen in C++	618
13.4.12	Best Practices für Zeigerparameter	619
13.5	Void-Zeiger: typenloses Zeigerwerkzeug	620
13.5.1	Die Idee des void-Zeigers	621
13.5.2	Implizite und explizite Konvertierung	622
13.5.3	Hexadezimale Ausgabe beliebiger Datentypen	623
13.5.4	Einschränkungen von void *	625
13.6	Zeigerbeziehungen und -operationen	626
13.6.1	Vergleichsarten: Gleichheit und Ordnung	627
13.6.2	Gleichheitsvergleich: identische Adressen	627
13.6.3	Vergleich über unterschiedliche Zeigertypen und void *	629
13.6.4	Ordnungsvergleiche: relative Position im Speicher	630
13.6.5	Zeigerdifferenzen und ptrdiff_t	632
13.6.6	Zeigerprüfungen in Bedingungen	632
13.7	Array-Decay und Zeigerarithmetik	634
13.7.1	Array-Decay: implizite Umwandlung	634
13.7.2	Unterschiede zwischen Array-Namen und echten Zeigern	636
13.7.3	Zeigerarithmetik: Rechnen mit Adressen	637
13.7.4	Index-Notation vs. Zeiger-Notation	639
13.7.5	Iteration mit Zeigern	641

13.7.6	Array-Größe in Funktionen und der Effekt des Array-Decay	645
13.7.7	Mehrdimensionale Arrays und Zeiger	646
13.7.8	Aufgabe	652
13.8	Zugriff auf Strukturen über Zeiger	652
13.8.1	Der Pfeiloperator (->)	654
13.8.2	Funktionen mit Strukturzeigern	656
13.8.3	Verschachtelte Strukturen und mehrfache Pfeile	657
13.8.4	Zeiger auf Strukturen und verkettete Listen	658
13.8.5	Doppelt verkettete Listen	660
13.8.6	Aufgabe	661
13.9	Arrays von Zeigern	661
13.9.1	Deklaration und Initialisierung	662
13.9.2	Arrays von Zeichenketten	664
13.9.3	Verwendung mit Null-Pointer als Sentinel	666
13.9.4	Beispiel: Nachrichtenverwaltung mit Arrays von Zeigern auf Strukturen	668
13.9.5	Beispiel: Verlaufsliste mit Array von Zeigern	669
13.10	Mehrstufige Indirektion mit Zeigern auf Zeigern	671
13.10.1	Zeiger auf Zeiger deklarieren	671
13.10.2	Konstantheit bei mehrstufiger Indirektion	673
13.10.3	Warum mehrstufige Indirektion?	674
13.10.4	Unterschied: Array von Zeigern versus Zeiger auf Zeiger	675
13.10.5	Funktionen mit Arrays von Zeigern	676
13.10.6	Darstellung von Matrizen: zweidimensionale Arrays und Zeiger auf Zeiger	677
13.10.7	Zeiger in Funktionen verändern	679
13.10.8	Beispiel: verkettete Listen und Zeiger auf Zeiger	681
13.10.9	Mehrdimensionale Zeiger-Strukturen	683
13.11	Kommandozeilenargumente als Array von Zeigern	685
13.11.1	Programmargumente auswerten	688
13.12	Funktionszeiger	688
13.12.1	Deklaration und Syntax	690
13.12.2	Adresse einer Funktion speichern	692
13.12.3	Aufruf über Funktionszeiger	693
13.12.4	Vereinfachte Syntax durch Function-to-Pointer-Decay	695
13.12.5	Typedef für bessere Lesbarkeit	696
13.12.6	Namensräume durch Strukturen	697
13.12.7	Funktion höherer Ordnung	702
13.12.8	Arrays von Funktionszeigern für Sprungtabellen	707

13.12.9	Vergleichsfunktionen	709
13.12.10	Suchen und Sortieren über die Standardbibliothek	714
13.13	Rückblick und Ausblick	718
14	Speicherverwaltung und Fehlersicherheit	721
14.1	Einführung in die dynamische Speicherverwaltung	721
14.1.1	Stack vs. Heap: Eigenschaften, Lebensdauer und Anwendungsfälle	722
14.2	Grundlegende Allokationsfunktionen und Anwendungen	723
14.2.1	Speicher allozieren (malloc())	724
14.2.2	Freigabe von Speicher (free())	725
14.2.3	Speicher für Arrays allozieren	728
14.2.4	Speicher für Strukturen allozieren	729
14.2.5	Opake Typen: Strukturen hinter einer Schnittstelle verbergen	732
14.2.6	Speicher für Arrays von Strukturen allozieren	735
14.2.7	Dynamisch verkettete Listen implementieren	736
14.2.8	Allokation mit Initialisierung auf null (calloc())	748
14.2.9	Erweiterung und Verkleinerung von Speicher- bereichen (realloc())	750
14.2.10	Sichere Größenberechnung	756
14.3	Arbeiten mit Speicherbereichen	757
14.3.1	Speicher kopieren	758
14.3.2	Speicher vergleichen	760
14.3.3	Strukturen vergleichen	761
14.3.4	Speicher setzen	763
14.3.5	Einzelne Bytes suchen (memchr())	766
14.3.6	Aufgabe	768
14.4	Flexible Array-Mitglieder (FAM)	771
14.4.1	Syntax und Verwendung flexibler Array-Mitglieder	772
14.4.2	Fazit und Zusammenfassung	773
14.5	Speicherlebenszyklen und Ownership-Modelle	773
14.5.1	Ownership in C: Verantwortlichkeiten und Konventionen	774
14.5.2	Referenzzählung	776
14.5.3	Scope-basierte Ressourcenverwaltung mit Cleanup-Attributen	779
14.6	Typische Speicherfehler und ihre Vermeidung	781
14.6.1	Null-Pointer-Checks	781
14.6.2	Buffer-Overflows und Underflows bei dynamischem Speicher	786
14.6.3	Memory Leaks (Speicherlecks)	789
14.6.4	Dangling-Pointers und Use-after-free	796

14.6.5	Double-free und Invalid-free	800
14.6.6	Übersicht und Zusammenfassung über Undefined Behavior im Speichermanagement	803
14.7	Fragmentierung und Out-of-Memory-Handling	804
14.8	Alignment und Padding	804
14.9	Debugging und Analyse von Speicherfehlern	807
14.9.1	AddressSanitizer	808
14.9.2	MemorySanitizer (MSan)	810
14.9.3	Vertrauen gut, Valgrind ist besser	811
14.9.4	Core Dumps und Post-Mortem-Debugging	813
14.9.5	Wann welches Tool einsetzen?	813
14.9.6	Weitere Tools	814
14.9.7	Entwicklung und heutiger Stand	815
14.10	Allokatoren	816
14.10.1	Wie malloc() sich Speicher holt	816
14.10.2	Implementierungen von malloc()/free() in modernen Systemen	818
14.10.3	Eigene Allokatoren schreiben	820
14.10.4	Custom Allocators parametrisieren	826
14.10.5	Laufzeitüberlegungen: Heap vermeiden (Stack-Allokatoren, VLAs)	827
14.11	Rückblick und Ausblick	828
15	Metaprogrammierung	831
15.1	Das Problem: generischer Code ohne zusätzlichen Laufzeitaufwand	831
15.1.1	Was ist Metaprogrammierung?	832
15.1.2	Die drei Phasen der Kompilierung	833
15.2	Präprozessor-Metaprogrammierung	833
15.2.1	Portable Konfiguration und Plattformerkenung	834
15.2.2	Feature-Erkennung	835
15.2.3	Standard-Makros und Metainformationen	836
15.3	Compilerbasierte Metaprogrammierung	837
15.3.1	Typprüfungen mit static_assert	837
15.3.2	Generische Makros mit typeof und typeof_unqual	841
15.3.3	Das container_of-Idiom	845
15.3.4	Typabhängige Auswahl mit _Generic	848
15.3.5	Optimierungshinweis unreachable()	860
15.4	Zusammenfassung und Ausblick	862

16	Zeichen, Zeichenketten und Textverarbeitung	865
16.1	Historische Grundlagen der Textrepräsentation	865
16.1.1	char, signed char, unsigned char	865
16.1.2	Codepages und das klassische C-Zeichenmodell	867
16.2	Von einzelnen Zeichen zu Zeichenketten	867
16.2.1	Null-Terminierung von Strings	867
16.2.2	Klassische String-Literale	869
16.2.3	Die Transformation von String-Literalen in C	869
16.2.4	Zeiger vs. Array	871
16.2.5	Speicherorte für nicht-literale Strings	873
16.2.6	Umgang mit konstanten Strings	873
16.3	Moderne Textrepräsentation	874
16.3.1	Unicode als abstraktes Zeichensystem	874
16.3.2	Codierungen	874
16.3.3	Anzeigeebene: wie Zeichen erscheinen	876
16.4	Moderne Texttypen und Literale	877
16.4.1	char32_t	877
16.4.2	char8_t	878
16.4.3	char16_t	878
16.4.4	Historischer Typ wchar_t	879
16.4.5	Fazit	879
16.4.6	Unicode-Präfixe für String-Literale und Zeichenkonstanten	879
16.4.7	Universal Character Names (UCN)	882
16.5	Moderne String-Repräsentationen	884
16.5.1	String mit explizit gespeicherter Länge	884
16.5.2	Small-String-Konzepte	886
16.5.3	String-View	887
16.6	Memory-Management-Techniken	889
16.6.1	String-Pools	889
16.6.2	Globales Interning	891
16.6.3	Flyweight-Pattern	892
16.7	Dynamische Strings	893
16.7.1	String mit Kapazität	893
16.7.2	Wachstumsstrategien und Kostenmodell	897
16.7.3	Arena-basierte dynamische Strings	897
16.7.4	Vergleich mit *sprintf()-Funktionen	898
16.8	Locale: Einfluss auf Zeichen- und String-Funktionen	898
16.8.1	Locale-Kategorien	899
16.8.2	Locale setzen	900

16.8.3	Probleme des globalen Locale-Modells	900
16.8.4	Formatierungsdaten abfragen	901
16.8.5	Praxis: Locale sparsam einsetzen	901
16.9	Formatierte Ausgabe in Strings	902
16.9.1	Formatfunktionen [v]snprintf()/s[n]printf_s()	902
16.9.2	sprintf() und vsprintf()	904
16.9.3	String-Konkatenation mit Formatfunktionen	905
16.9.4	Automatische Speicherallokation mit GNU-Funktionen asprintf() und vasprintf()	907
16.9.5	Format-Spezifizierer	908
16.9.6	Einzelne Werte in Strings konvertieren	910
16.9.7	Compilerprüfungen bei Formatstrings	910
16.9.8	Formatstring-Prüfung für eigene Funktionen in GCC/Clang	913
16.9.9	Formatstring-Schwachstellen	914
16.9.10	Sichere Nutzung der printf()-Familie	915
16.10	Parsing und Konvertierung	916
16.10.1	Tokenisierung: Text in Einheiten zerlegen	916
16.10.2	Tokenisierung mit Standardfunktionen (strtok*)	916
16.10.3	Parser selbst entwickeln	918
16.10.4	Zeichenketten in Zahlen umwandeln (strto*)	918
16.10.5	Verkürzte Varianten atoi(), atol(), atoll() vermeiden	922
16.10.6	Strukturierte Eingaben auslesen (sscanf*)	923
16.11	Klassische Charakterklassifikation (is*) und weitere)	926
16.12	Unicode in C23	927
16.12.1	UTF-8 selbst verarbeiten	928
16.12.2	Die Scheinlösung: Wide Characters	930
16.12.3	Externe Bibliotheken	930
16.13	Standard-Stringfunktionen für nullterminierte Zeichenketten	931
16.13.1	Die zwei grundlegenden Schwächen der klassischen C-Strings	932
16.13.2	Nur lesende Stringfunktionen	933
16.13.3	Schreibende Funktionen ohne Längenparameter	935
16.13.4	Schreibende Funktionen mit Längenparameter	935
16.13.5	Dynamische Stringfunktionen	937
16.13.6	Eigene Wrapper mit __builtin_object_size()	937
16.13.7	Beispiel: Dateiname und Dateiendung aus einem Pfad extrahieren	938
16.13.8	mem*()-Funktionen für String-Verarbeitung nutzen	940
16.13.9	Stringfunktionen mit verschiedenen String-Literal-Typen?	942
16.14	Vergleiche und Hashing	944
16.14.1	Grundformen des Stringvergleichs	944

16.14.2	Case-insensitive Vergleiche	945
16.14.3	Locale-abhängige Vergleiche	945
16.14.4	Natürliche Sortierung	947
16.14.5	Hashfunktionen für schnelle String-Vergleiche	948
16.15	Testing, Fuzzing und Verifikation von C-String-Code	951
16.15.1	Stringbezogene Fehler erkennen	951
16.15.2	Speicherbezogene Fehler erkennen	951
16.16	Aufgabe	951
16.17	Rückblick und Ausblick	956
17	Ein-/Ausgabe (I/O)	957
17.1	Grundlagen von Streams	957
17.2	Dateien öffnen und schließen	958
17.2.1	Eine Datei öffnen und schließen (fopen() und fclose())	958
17.3	Textdateien schreiben und lesen	960
17.3.1	Text schreiben (fprintf(), fputs(), fputc() usw.)	960
17.3.2	Text einlesen (fscanf(), fgets(), fgetc() usw.)	961
17.4	Standardstreams	962
17.4.1	Kurzformen für Ein- und Ausgabe	963
17.4.2	Einen Stream neu zuordnen (freopen())	963
17.5	Rohdaten verarbeiten	964
17.5.1	Strukturen binär schreiben und lesen	967
17.5.2	Aufgabe	970
17.6	Position im Stream steuern (fseek(), ftell() usw.)	972
17.7	Fehler erkennen und behandeln (ferror(), feof(), clearerr())	975
17.8	Puffer und Streameigenschaften steuern	976
17.8.1	Puffer einstellen (setvbuf())	976
17.8.2	Puffer leeren (fflush())	978
17.9	Dateien löschen und umbenennen	979
17.9.1	Eine Datei löschen (remove())	979
17.9.2	Eine Datei umbenennen (rename())	979
17.10	Temporäre Dateien	980
17.10.1	Eine temporäre Datei öffnen (tmpfile())	980
17.10.2	Sicheren temporären Dateinamen erzeugen (mkstemp())	981
17.11	Plattform- und POSIX-Erweiterungen	982
17.11.1	Umwandlung zwischen Deskriptoren und FILE *-Streams	982

17.12 POSIX-Erweiterungen rund um Datei- und Verzeichnis-I/O	983
17.12.1 Low-Level-I/O mit Dateideskriptoren	983
17.12.2 Dateisystem-Metadaten und Dateitypen	983
17.12.3 Rechte, Besitzer und Erstellungsmaske	984
17.12.4 Verzeichnisse lesen und verwalten	984
17.12.5 Links, Umbenennen und Löschen	985
17.12.6 Dateigröße ändern und Synchronisation	985
17.12.7 Dateideskriptor-Steuerung und Flags	985
17.12.8 Multiplexing und nicht-blockierendes I/O	986
17.12.9 Speicherabbildung von Dateien	986
17.12.10 Speicherbasierte Streams	986
17.12.11 Zeitstempel von Dateien setzen	986
17.13 Zusammenfassung und Ausblick	987
 18 Bitoperationen	 989
18.1 Bit- und Schiebeoperatoren	990
18.1.1 Bitweises UND (&)	991
18.1.2 Bitweises ODER ()	991
18.1.3 Bitweises XOR (^)	992
18.1.4 Bitweises NICHT (~)	992
18.1.5 Schiebeoperatoren (<<, >>)	993
18.1.6 Häufige Fehler bei bitweisen Operatoren	994
18.1.7 Aufgaben	995
18.2 Bitmasken, Flags und typische Muster	996
18.2.1 Bitmasken und ihre Konstruktion	996
18.2.2 Setzen einzelner Bits	997
18.2.3 Löschen einzelner Bits	998
18.2.4 Umschalten einzelner Bits	999
18.2.5 Prüfen einzelner Bits	999
18.2.6 Mehrere Bits gleichzeitig prüfen	1000
18.2.7 Aufzählungskonstanten als Bitmasken	1001
18.2.8 Häufige Fehler und Best Practices	1004
18.2.9 Zusammenfassung des Musters	1005
18.3 Standardisierte Bitfunktionen (<stdbit.h>)	1006
18.3.1 Übersicht der Funktionen	1006
18.4 Ganzzahlen mit fester Bitbreite (_BitInt)	1008
18.4.1 Beispiel und Überlaufverhalten	1008
18.4.2 printf() und Typecast	1010
18.4.3 _BitInt in Strukturen	1011

18.5	Bitfelder	1012
18.5.1	Bitfelder definieren	1012
18.5.2	Einfaches Flags-Beispiel	1013
18.5.3	IPv4-Beispiel	1015
18.5.4	Abbildung und Padding von Bitfeldern	1017
18.6	Rückblick und Ausblick	1019
19	Numerik und Mathematik	1021
19.1	Sichere Arithmetik (<stdckdint.h>)	1021
19.1.1	Wertebereiche und Überlaufprobleme	1021
19.1.2	Strategien zur Überlauferkennung	1022
19.1.3	Checked Integer Arithmetic	1023
19.1.4	Überlauf bei Gleitkommazahlen	1025
19.2	Besondere Gleitkommazahlen	1025
19.2.1	NaN	1026
19.2.2	Vorzeichenbehaftete Null (+0.0 und -0.0)	1026
19.2.3	Unendlichkeiten und Division durch null	1027
19.2.4	Subnormale Zahlen	1028
19.2.5	Klassifikation von Gleitkommazahlen	1028
19.2.6	Grenzen und Epsilon	1029
19.3	Konvertierungen zwischen Ganz- und Gleitkommazahlen	1032
19.3.1	Cast von Ganzzahlen in Gleitkommazahlen	1033
19.3.2	Casts von Gleitkommazahlen in Ganzzahltypen	1033
19.3.3	Rundung beim Cast von Gleitkommazahlen in Ganzzahlen	1034
19.4	Mathematische Funktionen (<math.h>)	1036
19.4.1	Demonstration ausgewählter Funktionen	1036
19.4.2	Übersicht der wichtigsten Funktionen	1037
19.4.3	Linken der Mathematik-Bibliothek (-lm)	1039
19.4.4	Aufgabe	1040
19.5	Floating-Point-Umgebung und Rundungsmodi (<fenv.h>)	1040
19.5.1	IEEE-754- und IEC-60559-Konformität in C23	1040
19.5.2	fesetround() und ihre Auswirkungen	1041
19.5.3	Floating-Point-Exceptions und Status-Flags	1044
19.5.4	Steuerung der FP-Umgebung in der Praxis	1045
19.6	Generische Mathematik (<tgmath.h>)	1046
19.6.1	Beispielnutzung mit Sinus	1046
19.6.2	Übersicht der typgenerischen Makros	1047

19.7	Zufallsfunktionen in C	1047
19.8	Komplexe Zahlen	1049
19.8.1	Datentypen float complex, double complex und long double complex	1049
19.8.2	Operatoren	1050
19.8.3	Funktionen	1050
19.8.4	Beispiel: Mandelbrot-Menge	1050
19.9	Rückblick und Ausblick	1051
20	Weitere Teile der C-Standardbibliothek	1053
20.1	Standard Utilities Header (<stdlib.h>)	1053
20.1.1	Überblick: bereits bekannte Teile	1053
20.1.2	Programm beenden	1054
20.1.3	Umgebungsvariablen lesen (getenv())	1055
20.1.4	POSIX-Erweiterungen für Umgebungsvariablen	1056
20.1.5	Externe Befehle ausführen (system())	1056
20.2	Signale (<signal.h>)	1058
20.3	Nichtlokale Sprünge (<setjmp.h>)	1059
20.3.1	Kontext speichern (jmp_buf)	1059
20.3.2	Rücksprungpunkt setzen (setjmp())	1059
20.3.3	Sprung ausführen (longjmp())	1060
20.3.4	Regeln und Fallstricke	1062
20.3.5	Try/Catch per Makros (setjmp()/longjmp())	1062
20.4	Zeit und Datum (<time.h>)	1067
20.4.1	Aktuelle Zeit abfragen: time_t und time()	1067
20.4.2	Zeitwerte in Kalenderzeit umwandeln	1069
20.4.3	Kalenderzeit formatieren: strftime()	1071
20.4.4	Kalenderdaten zurückwandeln: mktime() und timegm()	1074
20.4.5	Zeitspannen berechnen: difftime()	1076
20.4.6	Genauere Wandzeit: struct timespec und timespec_get()	1077
20.4.7	Auflösung abfragen: timespec_getres()	1079
20.4.8	POSIX am Rand: clock_gettime()	1079
20.4.9	CPU-Zeit messen: clock()	1080
20.4.10	Zeitzonen und Sommerzeit: tzset() als POSIX-Erweiterung	1081
20.5	Rückblick und Ausblick	1082

21	Nebenläufigkeit und Parallelität	1083
21.1	Einführung in Nebenläufigkeit	1083
21.1.1	Warum Nebenläufigkeit? Herausforderungen und Anwendungsfälle	1083
21.1.2	CPU-bound vs. I/O-bound Tasks	1085
21.1.3	Amdahl's Law und die Grenzen der Parallelisierung	1086
21.1.4	Threads vs. Prozesse	1088
21.2	Thread-Lebenszyklus: Erstellen, Ausführen, Beenden	1090
21.2.1	Thread-Bibliothek <threads.h>	1090
21.2.2	Erstellung eines Threads	1091
21.2.3	Freiwillig Rechenzeit abgeben	1094
21.2.4	Wer bin ich? Aktuellen Thread ermitteln	1095
21.2.5	Threads schlafen legen	1096
21.2.6	Natürliches und unnatürliches Ende eines Threads	1097
21.2.7	Der Hauptthread und Programmende	1101
21.2.8	Detached Threads	1104
21.2.9	Kooperatives Beenden von Threads durch Signalisierung	1106
21.2.10	Best Practices für den Thread-Lebenszyklus	1106
21.3	Das C-Memory-Modell und atomare Operationen	1106
21.3.1	Grundidee: Synchronisation schafft Ordnung	1107
21.3.2	Atomare und nicht-atomare Speicherzugriffe	1107
21.3.3	Data Race	1108
21.3.4	Ordnung und Korrektheit von Speicherzugriffen im C-Memory-Model	1110
21.4	Fortgeschrittene Konzepte kurz vorgestellt	1111
21.4.1	Atomare Typen und grundlegende Operationen (<stdatomic.h>) ...	1111
21.4.2	Synchronisation mit Mutex	1111
21.4.3	Condition Variables (cnd_t)	1111
21.4.4	Thread-lokaler Speicher (thread_local)	1112
21.5	Alternativen zu <threads.h>	1112
21.5.1	pthread als faktischer Referenzpunkt	1112
21.5.2	Ereignisschleifen statt Threads	1115
21.5.3	Konsequenzen für den Entwurf	1116
21.6	Rückblick und Ausblick	1116

22	Standardbibliothek, Implementierungen und Erweiterungen	1119
22.1	Implementierungen der Standardbibliothek	1119
22.1.1	Systemabhängige Implementierungen der libc	1120
22.1.2	Schlanke Implementierungen für eingebettete Systeme	1120
22.2	POSIX	1121
22.2.1	POSIX-Funktionsbereiche	1122
22.2.2	POSIX-Header	1124
22.2.3	POSIX aktivieren	1126
22.3	Plattformunabhängige Bibliotheken	1127
22.4	Plattformabstrahierende Bibliotheken	1128
22.4.1	Konkrete Beispiele plattformabstrahierender Bibliotheken	1129
22.5	Plattformspezifische APIs jenseits von C und POSIX	1130
22.5.1	Typische Funktionsbereiche plattformspezifischer APIs	1131
22.5.2	Header, Bibliotheken und ABI	1131
22.5.3	Konsequenzen für C-Programme	1131
22.6	Compiler- und Implementierungserweiterungen	1132
22.6.1	Typische Erweiterungen	1132
22.7	Zusammenfassung und Ausblick	1133
23	Build und Auslieferung	1135
23.1	Make	1136
23.1.1	Aufbau eines Makefiles	1136
23.1.2	Erstes Beispiel	1136
23.1.3	Ziele über die Kommandozeile ansprechen	1137
23.1.4	Abhängigkeiten	1137
23.1.5	Variablen und automatische Variablen	1138
23.1.6	Variablen beim Aufruf überschreiben	1140
23.1.7	Ausblick: pkg-config	1141
23.1.8	Ausblick: Autotools	1141
23.2	Projektstruktur in C	1144
23.2.1	Makefile unter Berücksichtigung der Verzeichnisstruktur	1145
23.3	CMake	1147
23.3.1	Ein erstes Beispiel	1148
23.4	Cross-Compiler	1149

23.5	Packaging und Distribution	1151
23.5.1	Statisches vs. dynamisches Linken	1151
23.5.2	Verteilungsformen	1151
23.5.3	Versionierung	1152
23.6	Continuous Integration	1152
23.7	Zusammenfassung und Ausblick	1154
24	Testen und Qualitätssicherung	1155
24.1	Compiler-Warnungen	1155
24.2	Statische Analyse	1156
24.3	Code-Formatter	1158
24.3.1	clang-format	1158
24.4	Tests	1159
24.4.1	Unity	1159
24.4.2	Regressionstests	1161
24.5	Code-Coverage	1161
24.6	Fuzzing	1162
24.7	Review-Prozesse	1163
24.8	Dokumentation	1164
24.8.1	Doxygen	1164
24.8.2	Was nicht dokumentiert wird	1165
24.9	Zusammenfassung und Ausblick	1166
25	Profiler und Optimierung	1167
25.1	Compiler-Optimierungen	1167
25.1.1	Optimierungsstufen: Steuerung durch Compiler-Schalter	1168
25.1.2	Optimierungen nach Reichweite	1169
25.1.3	Speicher und Aliasing als Optimierungsgrenze	1170
25.1.4	Fazit: Was bedeutet das für die Praxis?	1171
25.2	Grundlagen und Kennzahlen der Leistungsanalyse	1172
25.2.1	Laufzeit, Durchsatz, Latenz	1173
25.2.2	Messfehler, Störeinflüsse	1173
25.2.3	Warmup, Cache, Branch Prediction	1174
25.2.4	Messbare Kennzahlen und Messmethodik	1175
25.3	Werkzeuge	1176
25.3.1	Profiler	1176

25.3.2	Tracer	1177
25.3.3	Counter-basierte Werkzeuge	1178
25.3.4	Einordnung und Arbeitsweise	1179
25.4	Benchmarking und Reproduzierbarkeit	1180
25.4.1	Benchmarking als Experiment	1180
25.4.2	Micro- und Macro-Benchmarks als Messlinse	1180
25.4.3	Reproduzierbarkeit als Protokollfrage	1181
25.4.4	Vergleichbarkeit und Versuchsbedingungen	1181
25.4.5	Messreihen und Driftkontrolle	1182
25.4.6	Typische Benchmark-Artefakte	1182
25.4.7	Benchmark-Frameworks	1183
25.5	Optimierungstechniken im Code	1184
25.5.1	Algorithmische Optimierung	1184
25.5.2	Datenlayout und Cache-Lokalität	1185
25.5.3	Parallelisierung und SIMD	1187
25.5.4	Kontrollfluss und Mikrooptimierungen	1189
25.5.5	Die goldene Reihenfolge	1191
25.6	Methodik: systematisches Vorgehen bei Leistungsproblemen	1191
25.7	Rückblick und Ausblick	1193
26	Nachwort: der »C-Weg« – Philosophie und Kernprinzipien	1195
Anhang		1199
A	Standard-Header	1201
B	Die 8 kostbaren Übersetzungsphasen	1205
C	Operatorpräzedenz	1211
D	Konvertierungsspezifikationen sowie PRI- und SCN-Makros	1213
Index		1229

Index

__DATE__	396	Aggregattyp	145
__FILE__	394	Alias	407
__LINE__	394	Alias-Analyse	1170
__STDC_IEC_559__	1040	alignas	806
__STDC_IEC_559_COMPLEX__	1040	aligned_alloc	806
__STDC_IEC_60559_BFP__	398, 1040	Alignment	804
__STDC_IEC_60559_COMPLEX__	398, 1040	Alignment-Probleme	591
__STDC_IEC_60559_DFP__	398, 427	alignof	806, 837
__STDC_IEC_60559_TYPES__	398	Allocated Storage Duration	320
__STDC_UTF_16__	398	Anonyme Bitfelder	1019
__STDC_UTF_32__	398	Anonyme Union	506
__TIME__	396	ANSI C	46
_BitInt	1008	ANSI, American National Standards Institute	44
_Exit	1055	Anweisung	92
_Generic	848	argc	686
_GNU_SOURCE	908	Argument	264
.c-Datei	321	argv	686
/W4	1155	Arithmetischer Operator	147, 152
/Wall	1155	Arm Development Studio	77
/WX	1155	Array	513
\n	97	Array of Structs	1185
##-Operator	379	Array-Decay	634
#define	356	Array-to-Pointer Decay	695
#elif	388	ASan	808, 813, 1117
#elifdef	392	ASCII	867
#elifndef	392	ASLR	582
#else	388	asprintf	898
#endif	388	Assembler	41, 57
#if	388	Assoziativität	183
#ifdef	391	atexit	1054
#ifndef	391	atoi	922
#-Operator	376	Atomare Speicherzugriffe	1110
#undef	382	Attribut	282
%c, Konvertierungsspezifikation	97	Aufrufstapel	273
%d, Konvertierungsspezifikation	97	Auswahlstruktur	189
%f, Konvertierungsspezifikation	97	auto	110, 318
%s, Konvertierungsspezifikation	97	Automatic Storage Duration	320
		Autotools	1142

A

ABI	342, 804
abort	1055
access	984
Access violation	600
AddressSanitizer	808, 813, 1117
Adresse	661
Adressoperator	582

B

B	42
Backend	62
Backtrace	273
Barriere	1114
Basisfall	272

BCPL	42	calloc	748
Bedingte Kompilierung	388	camelCase	101
Bedingung	189	case	206
Bedingungsoperator	201	Case Range	211
Benchmark	1180	case-Label	207
Betriebssystem	42	Cast	174
Bezeichner	87, 148	cc65	62
Bibliotheken	1127	ceil	1037
Big-Endian	503	Celero	1183
Binär	140	CFLAGS	1139
Binäre Extended-Typen	425	Cfront	51
Binäre Interchange-Typen	425	char	100
Binärer Operator	152	Chocolatey	63
Binärmodus	957	Clamping	204, 255
Bitfeld	805, 1012	Clang	61
Bitgenaue Ganzzahlen	1008	clang-format	90, 1158
Bit-Operator	147, 990	clang-tidy	1156
Bjarne Stroustrup	50	class	51
Block	189	CLion	76
Blockkommentar	89	CMake	1135, 1147
BOM	350	CMakeLists.txt	1147
bool	100, 128	Code Units	875
Borland C	77	Code-Coverage	1161
Borrowing	775	Code-Formatter	1158
Branchless Programming	1190	Codepoint	874
break	232	Command Injection	1057
Breakpoint	81	Compilation	833
bsearch	714	Compiler	57
Buffer Underflow	786	Compiler-Optimierung	1167
Buffer Overflow	786	Compiler-Option	1155
BUFSIZ	978	Compiler-spezifische Header	1204
Build-System-Generator	1135	Compiler-Warnung	1155
Build-Tool	1135	Compound assignments	158
Byte	127	Conforming program	52
Byte Order Mark	350	const	112, 603
C		constexpr	114
C with Classes	50	Contiguous Memory	514
C++	44	continue	233
C11	46	Continuous Delivery	1153
C17	46	Continuous Integration	1153
C23	46	Core Dump	813, 814
C89	46	cos	1037
C90	46	cppreference	82
C95	46	CPU-Zeit	1080, 1173
C99	46	Cross-Compiler	1149
Call by Value	266	C-Standard	69
Callback	702	CSV	220
		ctype.h	926
		C-Weg	1195

D

Dangling else	199
Dangling-Pointer	614, 796
Data Race	1084, 1108
Datei	957
Datei-I/O	957
Datenmodelle	119
Datentyp	99
Debug-Build	80
Debugger	80
Debug-Symbole	80
default	208
Default-Label	208
Definition	113
Definition, Variable	100
Deklaration, Variable	100
Dekrement-Operator	158
Demand Paging	817
deprecated	284
Deprecation-Strategie	340
Dereferenzierungsoperator	586
Designated Initializers	471
Dezimal	140
Dezimale Typen	425
difftime	1076
Dmalloc	815
Docker	66
double	100
Double-free	800
do-while	222
Doxygen	1164
Drafts, Standard	50
Drei-Wege-Vergleich	709
Durchsatz	1173
DWARF	80

E

EABI	1150
Eclipse CDT	76
Einerkomplement	135
Einrückung	194
Einstiegspunkt	58
else	197
--enable-decimal-float	434
Endlosschleife	217
Endrekursion	275
enum	439
EOF	218, 975
Escape-Sequenz	125

exit	1054
exp	1037
Exponent	990
extern	313
External Linkage	309
Externe Definition	262

F

fabs	163
fabsf	163
faccessat	984
Fall-through	209
fallthrough	286
Fallunterscheidung	189
false	129
FAM	772
Fastbins	819
fclose	959
-fcommon	304
Feature	
<i>Obsolescent</i>	50
<i>Removed</i>	50
Fehlausrichtung	805
Fehlerbehandlung in I/O	975
feof	975
ferror	975
-ffast-math	1042, 1169
fflush	978
fgetc	961
fgets	961
FILE	958
File Scope	301
Fisher-Yates-Algorithmus	552
Fixed-Point-Arithmetik	1052
Fixed-Point-Darstellung	434
Flexible Array Members	772
float	100
Floating-Point-Kontraktionen	1038
floor	1037
FLT128_MAX	427
FLT16_MAX	427
FLT32_MAX	427
FLT64_MAX	427
fopen	958
fopen_s	959
for	225
Format-Makros	419
Formatstring	97
Fortschrittsausdruck	225
Forward-Deklaration	260

freed	964
free	725
free_sized	806
Freestanding-Umgebung	60
Frontend	62
-frounding-math	1042
-fsanitize	137
fscanf	961
fseek	972
ftell	972
-ftrapping-math	1042
-ftrapv	137
Füllbytes	805
Function-to-Pointer Decay	695
Funktion	43
Funktionsaufruf	257
Funktionsdefinition	257, 262, 313
Funktionsdeklaration	262, 313
Funktionsmakro	361
Funktionsprototyp	58, 260
Funktionsstyp	145
Funktionszeiger	688
-funsafe-math-optimizations	1169
Fuzzing	951, 1162
fwrite	964

G

Ganzzahlkonstante	88
Ganzzahlpromotion	174
Garbage Collection	43
GCC	47, 61
GCC Decimal-Float Library Routines	434
gcov	1162
gdb	81
Geheimnisprinzip	734
Geprüfte Arithmetik	137
Geschachtelte Schleifen	229
getc	962
getline	962
Gleitkommakonstante	88
GLFW	1129
GLib	1130
glibc	70, 1120
Globale Variable	301
GNU C Library	1120
GNU-Dialekt	72
Godbolt	78
Google Benchmark	1183
goto	246
gprof	1176

Grammatik von C	91
Grundtyp	100
GTK	1130
Gültigkeitsbereich	115

H

Hallo-Welt-Programm	57
Haltepunkt	81
Hauptarena	818
Header-Datei	47, 58, 321
Header-Guard	328
Heap	722
Helgrind	1117
Hexadezimal	140
Hochsprache	42
Homebrew	63
Host-Umgebung	60
htonl	969
htons	969

I

ICU	879
IDE	75
Identifizier	87, 148
IEEE 754	120
if	189
Implementation-defined behavior	52
in place	541
indeterminierte Repräsentation	103
Index	519
Indexnotation	519
INFINITY	122
Information Hiding	734
Initialisierung	104
Initialisierungsausdruck	225
Inkrement-Operator	158
inline	280
Instructions per Cycle	1176
int	100
INT_MAX	134
INT_MIN	134
int16_t	410
int32_t	410
int64_t	410
int8_t	410
Integer Promotion	174
Internal Linkage	309
Invalid-free	801
Invariante	242

IPC	1176
isalpha	926
isdigit	926
ISO C	46, 71
ISO/IEC 60559	120
ISO/IEC JTC 1/SC 22/WG 14	45
isspace	926
Iteration	189

J

Jahr-2038-Problem	1068
JSON	234
Jump Table	707

K

K&R	44
Keil μ Vision	77
Kernel	42
Kettenvergleich	165
Klammern	168
Klein- und Großschreibung	59
Kommaoperator	182, 228
Kommentar	89
Konstante	87, 95
Konstanter Zeiger	605
Kontrollstruktur	43, 189
Konvertierungsspezifikation	97
Konvertierungsspezifizierer	97
Kurzschlussverhalten	170

L

Label	246
Largebins	819
Latenz	1173
Laufvariable	226
Laufzeit	1173
LDFLAGS	1139
LDLIBS	1139
LeakSanitizer	810
Lebensdauer	116
libFuzzer	1162
Lifetime	116
LIKWID	1179
limits.h	134
Line-Marker	354
Linkage	299, 319
Linker	57
Linking	833

Linus Torvalds	44
Linux-Kernel	53
Literal	148
Literal-Encoding	870
Little-Endian	503
LLP64-Modell	119, 412
LLVM	61
Locale	867
log	1037
Logische Speicheradresse	581
Logischer Operator	147
Lokale Variable	301
long	100
long double	100
long long	100
longjmp	1059
LP64-Modell	119, 412
LSan	810
lstat	984
L-Value	105, 150

M

Macro Bleeding	336
Macro-Benchmarks	1180
main	686
main-Funktion	57
Make	1135, 1136
Makefile	1136
Makro	356
malloc	724
Man-Pages	82
Mantisse	990
Maschinencode	57
math.h	1036
max_align_t	806
maybe_unused	286
Mehrfachinitialisierung	151
Memcheck	812, 814, 951
memcmp	757
memcpy	757
memmove	757
Memory Leak	789
Memory-Mapped I/O	593
MemorySanitizer	810
memset	757
memwatch	815
Meson	1135
Metaprogrammierung	832
Micro-Benchmarks	1180
Microsoft Visual Studio	76

Migration	1173
Minimalbreiten-Typen	415
MISRA-C	47
mkstemp	980, 981
mktime	1074
Modularisierung	321
Modulo	155
Modulo-Bias	1048
MSan	810
MSVC	61
MSYS2	63
Multics	41
Mutex	1111

N

Name Mangling	404
Namenskollision	328
NaN	122, 1026
Narrowing Conversions	177
Nebenläufigkeit	1083
Negative Unendlichkeit	121
NetBeans	77
Nicht-atomare Speicherzugriffe	1110
Nichtlokaler Sprung	1059
Ninja	1135
No Linkage	309
nodiscard	283
noreturn	285
Not a Number	1026
ntohl	969
ntohs	969
NULL-Makro	602
Null-Pointer	600
nullptr	599
nullptr_t	601

O

Objektdatei	73
Objekttyp	145
OBOE	241
Obsolescent	50
ODR	262, 305
-Ofast	1169
Off-by-one error	240, 241
offsetof	806, 837
Oktal	140
One Definition Rule	262, 305
Opaque type	732
Operator	88

Operatorpräzedenz	183
Optimierungsstufe	1167
Optional installieren	1144
Origin-Tracking	810
Ownership	774

P

packed	806
Padding	805
Padding-Bits	411, 1019
PAPI	1179
Parallelität	1083
Parameter	264
PascalCase	102
PCH	341
PCRE	918
PDP-Maschine	41
perf	1176
Performance Engineering	1192
Phony Target	1136, 1137
Pipeline-Flush	1190
pkg-config	1141
Pointer	579
Pointer Tagging	597
Portabilität	42
Positive Unendlichkeit	121
POSIX	49, 1121
POSIX-Header	1204
Postfixform	159
Post-Mortem	814
Post-Mortem-Debugging	808, 813
pow	1037
Präfixform	158
pragma pack	806
Präprozessor	73, 349
Präzedenz	149, 183
Precompiled Headers	341
Preprocessing	833
Preprocessing Token	378
Primärer Ausdruck	148
printf	97
Profiler	1176
Programmierparadigma, imperativ	41
Promotion	174
Property-based Testing	951
Prozess	722, 1088
Pseudozufallszahl	1047
Punctuator	88
puts	58

Q

qsort	714
Qt Creator	76
Quelldatei	321
Quellzeichensatz	869
quick_exit	1055

R

Race Condition	1108
raise	1059
rand	1048
Rangfolge	168
Raw-String-Literale	869
realloc	750
rechtsassoziativ	152
Redzone	809
Referenzzählung	776
Rekursion	272
Rekursionsschritt	272
Release-Build	80
Relokation	73
Remote Debugging	82
removed	50
restrict	616
Restwertoperator	155
Result-Muster	500
return	250, 267
Returning Ownership	774
Review	1163
rewind	972
round	1037
Rückgabetyt	257
R-Value	150

S

Sanitizer	951
scanf	105
Schiebeoperator	990
Schleife	189
Schleifeninvariante	242
Schleifenkörper	216
Schleifenvariable	226
Schlüsselwort	86
Scope	115, 319
SCREAMING_SNAKE_CASE	102
Segmentation fault	600

Semantic Versioning	339, 1152
Semikolon	59, 92
SemVer	1152
Sequenzielle Initialisierung	471
Sequenzielle Konsistenz	1110
Sequenzierung	185
Sequenzpunkte	185
setjmp	1059
setjmp.h	1059
setvbuf	977
Shadow Memory	808
Shared Libraries	342
short	100
Short-Circuit Evaluation	170
SIGFPE	154
SIGINT	1058
signal	1058
signal.h	1058
signed	130
SIGSEGV	600, 1058
SIGTERM	1058
sin	1037
Sinking Ownership	775
size_t	424
sizeof	143, 837
Skid	1179
Smallbins	819
snake_case	101
snprintf	902
Spaghetticode	246
Speicheradresse	581
Speicherklasse	299
Speicherleck	789
Speichermanager	723
Speicherpool	817
Speicherverwaltung	43
sprintf	904
Sprungmarke	246
Sprungstruktur	189
Sprungtabelle	707
SQLite	53
sqrt	1037
srand	1048
Stack	273, 722
Stack Overflow	274
Stack Trace	273
Standardausgabe	292
Standardbibliothek	47, 58
Standardeingabe	292

Standardfehlerausgabe	292
stat	983
Statement-Expressions	400
static	306
Static Storage Duration	320
static_assert	837
Statische Analyse	1156
-std	71
stdalign.h	806
stdarg.h	277
stdbit.h	1006
stdbool.h	129
stdc_bit_width	566
stddef.h	424
stderr	292
stdin	292
stdint.h	409
stdout	292
Stepping	81
Storage Class	319
Storage Duration	319
Storage-Class-Specifier	114
strcat	931
strchr	931
strcmp	931
strcpy	931
Stream	957
strftime	1071
Strict-Aliasing-Regel	591, 1170
Strictly conforming program	52
String	865
string.h	868
Stringifizierung	376
Stringisierung	376
String-Literal	87, 95
Strippen von Symbolen	81
strlen	868
strncat	931
strncmp	931
strncpy	931
strptime	1074
strstr	931
strtoul	922
struct	457
Struct of Arrays	1186
Struktur	457
Subnormale Zahlen	1028
Surrogate-Paare	875
switch	206
Synchsafes Integers	995
Syntax	59

T

Tagged Unions	497
Tagname einer Struktur	458
Tags einer Struktur	479
Tail Recursion	275
Tail-Call-Elimination	275
tan	1037
tentative Definition	317
Ternärer Operator	203
Test	1159
Textmodus	957
Textsubstitution	326
The C Programming Language	44
thr_create	1090
thr_join	1090
Thread	722, 1083
Thread Storage Duration	320
Thread-Local Storage	1112
threads.h	1090
ThreadSanitizer	810, 1117
time_t	1067
timegm	1075
TinyCC	62
TLS, siehe Thread-Local Storage	1112
tmpfile	980
tmpnam	980
tm-Struktur	1070
Token	86
Token-Pasting	379
Token-Verkettung	379
tolower	926
toupper	926
trailing padding	805
Translation phases	1205
Translation Unit	86, 299
true	129
TSan	810, 1117
Tupel	468
Turbo C	77
Typableitung	110
Typalias einer Struktur	480
Type Inference	110
Type Punning	484
typedef	407, 476
typeof	842
typeof_unqual	844
Typkonvertierung	174
Typqualifizierer	112
Typumwandlung	174

U

UB	52
Überlauf	136, 179
Übersetzungseinheit	86
Übersetzungsphasen	73, 1205
UBSan	810
UINT_MAX	134, 136
UINT16_C	413
uint16_t	410
UINT32_C	413
uint32_t	410
uint64_t	410
UINT8_C	413
uint8_t	410
uintptr_t	596
Umbrella-Header	331
Unärer Operator	152
undefined behavior	52
UndefinedBehaviorSanitizer	810
undefiniert	98
undefiniertes Verhalten	805
Unendlichkeit	122
Unicode-Präfix	880
UNICS	41
Union	485
Unit-Test	1159
Unity	1159
Unix	41
unsigned	130
Unsorted Bin	819
unspecified behavior	52
unvollständiger Typ	129, 733
Use-after-free	797
uselocale	901
UTF-8-Format	87, 882

V

va_arg	277
va_end	277
va_list	277
va_start	277
Valgrind	1176
Variable	43, 100
Variable-Length Array	516
Variadische Funktion	97, 276
Variantentypen	484

vasprintf	898
Verbundanweisung	92
Vergleichsoperator	147
Versteckte Aliasse	1171
Visual Studio Code	76
VLA	516
VLA-Typen	517
void	58, 100, 129
volatile	593
vordefinierte Makros	394
vorzeichenbehaftet	130
vorzeichenlos	130
Vorzeichenverlust	179
vsprintf	902
vsprintf	904

W

-Wabsolute-value	853
-Wall	71, 1155
Wall-Clock-Time	1173
Warnung	1155
-Wassign-enum	445
Watchpoint	81
-Wcast-qual	604
Werkzeugkette	57
-Werror	72, 1155
-Wextra	71, 1155
-Wfloat-equal	163
-Wformat	911
-Wformat=2	1155
while	216
Wide Literal Encoding	870
Wiederholungsstruktur	189
-Wimplicit-fallthrough	212
-Wincompatible-pointer-types	674
-Wpedantic	71, 1155
Wrap-Around	136
Wrapper-Allokatoren	821
-Wsequence-point	186
-Wshadow	1155
WSL	64
-Wswitch	441
-Wswitch-default	208
-Wswitch-enum	441
-Wunsequenced	186
-Wuse-after-free	797

X

XOR 99

Y

Yoda-Stil 194

Z

Zeichenfolge 59

Zeichenkette 59, 865

Zeichenkonstante 88

Zeiger 43, 579

Zeiger auf Zeiger 671

Zeigerarithmetik 634, 637

Zeilenkommentar 89

Zusammengesetzte Zuweisung 158

Zusammenhängender Speicher 514

Zustandsautomat 251

Zuweisung 149

Zuweisungsoperator 149

Zweierkomplement 135

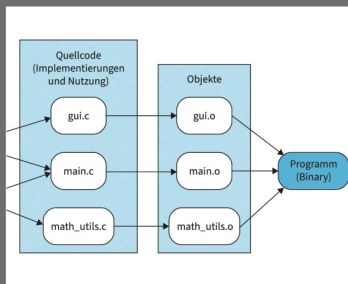
Modernes C von Anfang an

C ist die Lingua Franca der Systemprogrammierung. In diesem Handbuch lernen Sie C23 von Anfang an – egal, welchen Wissensstand Sie mitbringen. Von Variablen, Datentypen und Funktionen bis hin zu Zeigern und Metaprogrammierung erfahren Sie anhand vieler Übungen, wie Sie die Sprache professionell einsetzen. So schreiben Sie sauberen, sicheren und portablen Code.

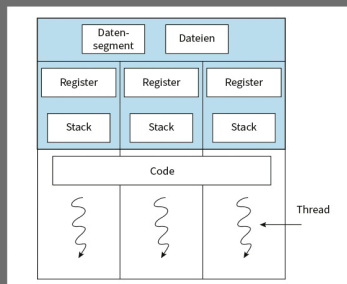
```
#include <stdio.h>

int main() {
    puts("Hello, world!");
}
```

Grundlagen und Konzepte



Viele Beispielprogramme



Fortgeschrittene Techniken

Ein stabiles Fundament

Syntax, Datentypen, Arrays und Zeiger: Erschließen Sie sich Schritt für Schritt die Welt der C-Programmierung. Oder frischen Sie bereits vorhandenes Wissen auf, bevor Sie mit den komplexen Themen starten.

C23 – professionell umgesetzt

Lernen Sie C-Programmierung, wie sie heute im Einsatz ist – mit allen neuen Features und Blick auf Kompatibilität. Ob für die Ausbildung oder im Beruf, zur Einführung oder zum Nachschlagen: Hier werden Sie fündig.

Von der Theorie zur Praxis

Nach jedem Kapitel lädt Sie der Autor dazu ein, selbst zu experimentieren und zu coden. Anhand vieler Übungen und Best Practices verknüpfen Sie direkt die Theorie mit der Praxis.



Alle Beispielprojekte zum Download



Christian Ullenboom ist Diplom-Informatiker und seit Jahrzehnten in der Welt der C-Programmierung zu Hause. Er ist Trainer und Berater für verschiedene Technologien und teilt seit Jahren sein Wissen auf seiner Webseite.

Aus dem Inhalt

Grundlagen

Variablen, Datentypen und Typsicherheit
Operatoren und Ausdrücke
Kontrollstrukturen
Funktionen
Arrays

Fortgeschrittene Themen

Zeiger
Speicherverwaltung
Metaprogrammierung
Zeichen und Textverarbeitung
Nebenläufigkeit und Parallelität
Die Standardbibliothek
Testen und Qualitätssicherung
Profiler und Optimierung

