

C programmieren

Table of Contents

1. Vorwort	22
1.1. Warum dieses Buch?	22
1.2. Warum C heute lernen?	23
1.3. Philosophie und Besonderheiten des Buches	24
1.4. Zielgruppe	25
1.5. Voraussetzungen	25
1.6. Aufbau und Lesetipps	26
1.7. Konventionen in diesem Buch	26
1.8. Über den Autor	28
1.9. Danksagung	28
2. Philosophie von C	28
2.1. Was ist C und was ist es nicht?	29
2.1.1. Ursprung und Motivation	29
2.1.2. Was C ausmacht	30
2.1.3. Was C nicht ist	31
2.1.4. C im Vergleich zu C++	31
2.2. Der C-Standard	33
2.2.1. Versionen von C	34
2.2.2. Die C-Standardbibliothek und ihre Header	36
2.2.3. Ein Kommen und Gehen	38
2.2.4. Drafts	39
2.2.5. Gemeinsame Wurzeln und unterschiedliche Entwicklung von C und C++	39
2.2.6. Abseits des Standards	40
2.3. Die Bedeutung von C heute	40
2.3.1. Warum C immer noch so wichtig ist	40
2.3.2. C als Fundament der digitalen Welt	41
2.3.3. Aktuelle Popularität	42
2.4. Rückblick und Ausblick	43
3. Werkzeugkette und das erste C-Programm	43
3.1. Minimales Beispiel main.c	44
3.1.1. Ein kleiner Exkurs zur Programmsyntax	47
3.1.2. Host-Umgebungen und Freestanding-Umgebungen {#host-freestanding}	47
3.2. Compiler im Alltag: GCC und Clang	48
3.2.1. GNU Compiler Collection (GCC)	49
3.2.2. LLVM und Clang	51
3.2.3. Compiler in Containern	52

3.2.4. Microsoft Visual C Compiler (MSVC)	53
3.3. C-Programm auf der Kommandozeile übersetzen und ausführen	54
3.3.1. Sprachunterstützung der Compiler	56
3.3.2. Sprachstandard explizit festlegen (ISO-C).....	57
3.3.3. GNU-Standard	57
3.4. Präprozessor, Compiler und Linker {#Übersetzungsphasen}	58
3.4.1. Übersetzungsphasen explizit ausführen (GCC/Clang)	59
3.5. Editoren, IDEs	60
3.5.1. IDEs und Editoren im Überblick	60
3.5.2. Webbasierte IDEs	62
3.5.3. Compiler Explorer	62
3.6. Debugger und Laufzeitanalyse	65
3.6.1. Debug vs. Release	66
3.6.2. Breakpoints, Stepping, Watchpoints	66
3.6.3. Remote Debugging	67
3.7. Dokumentation zu C-Funktionen finden	67
3.8. Rückblick und Ausblick	68
4. Syntax und Struktur von C-Programmen	68
4.1. Vom Quelltext zum Programm	68
4.1.1. Translation Units und Header-Dateien	69
4.2. Lexikalische Grundlagen	70
4.2.1. Schlüsselwörter	70
4.2.2. Bezeichner	70
4.2.3. Unicode und UTF-8	71
4.2.4. Literale	71
4.2.5. Punctuators und Operatoren	72
4.2.6. Kommentare	72
4.3. Lesbarkeit und Stil	74
4.4. Grammatik und Programmbausteine	74
4.4.1. Deklarationen und Definitionen	74
4.4.2. Anweisungen und Blöcke	75
4.4.3. Das Semikolon	75
4.5. Rückblick und Ausblick	76
5. Variablen, Datentypen und Typsicherheit	77
5.1. Literale: Erste feste Werte im Programm {#Literale}	77
5.1.1. Ganze Zahlen, Gleitkommazahlen, Zeichen, Strings	77
5.1.2. Literalverkettung {#Literalverkettung}.....	77
5.2. Werte ausgeben mit der Formatfunktion (printf())	78
5.2.1. Falsche Formatangaben	79
5.2.2. Gleitkommazahlen in wissenschaftlicher Notation	80
5.3. Grundtypen	81

5.4. Variablen	81
5.4.1. Deklaration von Variablen	81
5.4.2. Deklaration mit Initialisierung	85
5.4.3. Werte einlesen (scanf) <code>{#scanf}</code>	87
5.4.4. Typableitung (Type Inference) (auto) <code>{#auto}</code>	91
5.5. Konstante Werte	93
5.5.1. Konstante Werte (const)	94
5.5.2. Konstant berechenbare Ausdrücke (constexpr) <code>{#constexpr}</code>	95
5.5.3. West gegen Ost const <code>{#west-east-const}</code>	97
5.5.4. Gültigkeitsbereich und Lebensdauer	97
5.6. Ganzzahlen (int, short, long, long long)	99
5.6.1. Ganzzahltypen mit implementierungsabhängiger Bitbreite	99
5.6.2. Datenmodelle <code>{#Datenmodelle}</code>	100
5.6.3. Ganzzahltypen mit fester, exakt definierter Breite	101
5.7. Gleitkommatypen (float, double, long double)	102
5.7.1. Spezielle Werte: $\pm\infty$, NaN und vorzeichenbehaftete Nullen	103
5.7.2. Stillschweigend gerundet	105
5.8. Zeichentyp (char) <code>{#char}</code>	105
5.8.1. Escape-Sequenzen	106
5.8.2. Hexadezimale Escape-Sequenzen	107
5.8.3. Mehrzeichen-Char-Literal	108
5.8.4. Wie der C-Standard ein Byte bestimmt	108
5.9. Wahrheitswerte (bool)	109
5.9.1. Wie C lernte, wahr und falsch zu unterscheiden	110
5.10. Der Nicht-Wertetyp (void)	110
5.11. Vorzeichenbehaftete und vorzeichenlose Ganzzahlen	111
5.11.1. Einführung in vorzeichenbehaftete und vorzeichenlose Typen	111
5.11.2. Sonderfall char	113
5.11.3. Wertebereiche und Darstellung	114
5.11.4. Zweierkomplementdarstellung	115
5.11.5. Überlaufverhalten	117
5.11.6. Mischen von vorzeichenbehafteten und -zeichenlosen Datentypen <code>{#Wconversion}</code>	118
5.12. Zahlendarstellungen und Schreibweisen	120
5.12.1. Oktal, Hexadezimal und Binär	120
5.12.2. Literal-Suffixe für Ganzzahlen und Gleitkommazahlen (U, L, f ...)	121
5.12.3. Zifferntrenner	124
5.13. Typkategorien und Grundbegriffe	125
5.14. Rückblick und Ausblick	125
6. Operatoren und Ausdrücke	125
6.1. Primäre Ausdrücke	126
6.2. Zuweisungsoperatoren	127

6.2.1. Einfache Zuweisung	127
6.2.2. L-Values und R-Values	128
6.2.3. Zuweisung als Ausdruck	129
6.3. Arithmetische Operatoren (+, -, *, /, %)	131
6.3.1. Unäre und binäre Operatoren	131
6.3.2. Grundlegende arithmetische Operatoren	131
6.3.3. Restwertberechnung	134
6.4. Kurzschreibweisen für Zuweisungen und Änderungen	136
6.4.1. Erweiterte Zuweisungen	136
6.4.2. Präfixformen und Postfixformen	137
6.5. Vergleichsoperatoren	139
6.5.1. Gleichheitsoperatoren (==, !=)	140
6.5.2. Relationale Operatoren (<, <=, >, >=)	142
6.5.3. Vergleiche mit Gleitkommazahlen	143
6.6. Logische Operatoren (!, &&,)	144
6.6.1. Der Negationsoperator (!)	144
6.6.2. Der UND-Operator (&&)	145
6.6.3. Der ODER-Operator ()	145
6.6.4. Zusammengesetzte logische Ausdrücke	146
6.6.5. Abgeleitete logische Operatoren	148
6.6.6. Kurzschlussverhalten	148
6.6.7. Arithmetische Tricks mit booleschen Werten	150
6.7. Typumwandlungen (Conversions) und ihre Regeln {#Typumwandlungen}	151
6.7.1. Integer-Promotion (Ganzzahlpromotion)	152
6.7.2. Implizite Konvertierungen (Automatic Conversions)	152
6.7.3. Explizite Konvertierungen (Casts)	155
6.7.4. Das Signed/Unsigned-Problem {#Signed_Unsigned}	156
6.8. Kommaoperator, Vorrang, Assoziativität und Auswertungsreihenfolge	158
6.8.1. Kommaoperator {#Kommaoperator}	158
6.8.2. Operatorpräzedenz {#Operatorpräzedenz}	159
6.8.3. Assoziativität	160
6.8.4. Auswertungsreihenfolge (Sequenzierung)	161
6.9. Mathematisches aus den Standard-Bibliotheken	163
6.9.1. Mathematische Funktionen in <stdlib.h>	164
6.10. Rückblick und Ausblick	165
7. Kontrollstrukturen	165
7.1. Bedingte Anweisungen (Fallunterscheidungen) (if, else)	166
7.1.1. Komplexe Bedingungen	167
7.1.2. Typische Fehler	169
7.1.3. if-else-Anweisung	173
7.1.4. else-if-Ketten für mehrere Bedingungen	175

7.2. Bedingungsoperator (?) {#Bedingungsoperator}	177
7.2.1. Verschachtelter Bedingungsoperator	179
7.2.2. Beispiel: Werte einklemmen	180
7.2.3. Bedingte Ausdrücke mit ausgelassenem Mittelteil (GCC)	181
7.3. switch-Anweisung	182
7.3.1. Default-Label	184
7.3.2. switch oder if?	185
7.3.3. Fall-through	185
7.3.4. Gültigkeitsbereich von Variablen	190
7.4. while-Schleife	191
7.4.1. Endlosschleifen	192
7.4.2. Zuweisungen in der Bedingung	193
7.4.3. Aufgaben	196
7.5. do-while-Schleife	197
7.5.1. Konvertierung zwischen while und do-while	198
7.5.2. Aufgaben	199
7.6. for-Schleife	200
7.6.1. Varianten und leere Teile der for-Schleife	202
7.6.2. Mehrere Operationen mit dem Kommaoperator	203
7.6.3. Geschachtelte Schleifen	204
7.7. Schleifen mit break und continue steuern	206
7.7.1. break	207
7.7.2. continue	207
7.7.3. Grenzen von break und continue	207
7.7.4. JSON-Dokumente formatieren (Pretty Printing)	208
7.8. Best Practices, Tipps und Fehlerquellen bei Schleifen	212
7.8.1. Ungewollte Endlosschleife	213
7.8.2. Häufiger Fehler: Off-by-one-Fehler	214
7.8.3. Wahl der Schleifenart	216
7.8.4. Schleifenvariablen und Invarianten {#Invarianten}	216
7.8.5. Performance-Überlegungen	217
7.8.6. Abschlusstipps	219
7.9. Programmsprünge (goto) {#goto}	219
7.9.1. Warum man bei goto aufpassen muss	220
7.9.2. goto zur Fehlerbehandlung {#goto-Fehlerbehandlung}	222
7.9.3. Aus verschachtelten Schleifen ausbrechen	222
7.9.4. Doppeldeutigkeit von break bei switch und Schleifen	223
7.9.5. Zustandsautomaten	224
7.10. Rückblick und Ausblick	226
8. Funktionen	226
8.1. Warum Funktionen?	227

8.1.1. Vorteile eigener Funktionen	228
8.2. Funktionsgrundlagen: Definition, Deklaration und Aufruf	229
8.2.1. Erstes Beispiel: eine Banner-Funktion {#void}	229
8.2.2. Benennung von Funktionen	231
8.2.3. Definition, Deklaration und Funktionsprototyp	232
8.3. Parameterübergabe	234
8.3.1. Parameter und Argumente	235
8.3.2. Call by Value	237
8.3.3. Übliche Typkonvertierungen	237
8.4. Funktionen mit return verlassen	238
8.5. Funktionen mit Rückgabewerten	239
8.5.1. Kompakte Rümpfe bei boolean-Rückgaben	240
8.5.2. Was passiert, wenn ein Rückgabewert fehlt?	240
8.6. Überladen von Funktionen in C?	242
8.7. Rekursive Funktionen	243
8.7.1. Das Stack-Modell	244
8.7.2. Endlosrekursion	245
8.7.3. Weitere Rekursionsprobleme	246
8.7.4. Endrekursion (Tail Recursion) und Optimierung	246
8.7.5. Rekursion oder Iteration?	247
8.8. Variadische Funktionen (variable Argumentlisten)	247
8.8.1. Syntax mit	248
8.8.2. Makros aus <stdarg.h>	248
8.8.3. Typinformation und Default-Promotions	249
8.8.4. Anzahl und Typen der variablen Argumente erkennen	250
8.8.5. Stellenwert variadischer Funktionen	250
8.9. Inline-Funktionen	250
8.9.1. Das inline-Schlüsselwort als Compiler-Hinweis	251
8.9.2. Lokale Hilfsfunktionen mit static inline	252
8.10. Attribute {#Attribute}	252
8.10.1. Attribut-Syntax	252
8.10.2. Standard-Attribute	253
8.11. Fehlerbehandlung in Funktionen	259
8.11.1. Rückgabewerte als Fehlerindikatoren	259
8.11.2. Programm mit Fehlercode beenden	260
8.11.3. Fehlerbehandlung in C mit errno {#errno}	261
8.11.4. Ein-/Ausgabe-Kanäle und Fehlermeldungen	262
8.11.5. Fehlermeldungen mit perror und strerror	264
8.11.6. Das Makro assert {#assert}	265
8.12. Rückblick und Ausblick	269
9. Scope, Linkage, Storage Duration und Modularisierung	269

9.1. Scope, Linkage und Storage Duration von Variablen und Funktionen	270
9.1.1. Globale Variablen	271
9.1.2. Statische globale Variablen	276
9.1.3. Statische lokale Variablen	276
9.1.4. Globale Funktionen	278
9.1.5. Zusammenfassung: Sichtbarkeit (Scope) und Linkage	279
9.1.6. Storage Duration	280
9.2. Storage Classes im Überblick	282
9.2.1. extern bei Funktionen {#extern_funktion}	282
9.2.2. extern bei Variablen	284
9.2.3. auto	286
9.3. Header-Dateien und Modularisierung {#eigene_Header-Dateien}	288
9.3.1. Module bilden	289
9.3.2. Funktionsprototyp im Header, Implementierung in .c-Dateien	290
9.3.3. #include und Präprozessor	291
9.3.4. #include <...> vs. #include "..." {#include_bracket_hash}	293
9.3.5. Include-Pfade und -I	294
9.3.6. Header Best Practices	295
9.4. Rückblick und Ausblick	307
10. Der C-Präprozessor {#Präprozessor}	307
10.1. Historische Entwicklung und Motivation	307
10.1.1. Rolle in den Übersetzungsphasen	308
10.1.2. Erstes Makro	308
10.1.3. Verwendung außerhalb von C	312
10.1.4. C++ und der Präprozessor	312
10.2. Makros definieren: #define	313
10.2.1. Wenn möglich, dann constexpr	313
10.2.2. Syntax-Makros	313
10.2.3. Konstanten-Makros	316
10.2.4. Funktions-Makros	317
10.2.5. Funktions-Makros mit Parametern	321
10.2.6. Makros entfernen: #undef	336
10.2.7. Funktionen per Makro umleiten	337
10.2.8. X-Makros: Listen einmal definieren, mehrfach verwenden	341
10.3. Bedingte Kompilierung	342
10.3.1. #if, #elif, #else und #endif	342
10.3.2. Der defined-Operator	343
10.3.3. #ifdef und #ifndef	344
10.3.4. #elifdef und #elifndef	345
10.4. Vordefinierte Makros	347
10.4.1. Datei- und Zeileninformationen	347

10.4.2. Compile-Datum und -Zeit	349
10.4.3. Standard- und Umgebungsinformationen	350
10.4.4. Unterstützung optionaler Sprach- und Bibliothekserweiterungen	351
10.5. Compilerabhängige Erweiterungen	352
10.5.1. Statement-Expressions {#StatementExpressions}	352
10.5.2. Compilerspezifische und plattformabhängige Makros	353
10.5.3. Beispiel: Portabler Einsatz von Attributen mit bedingter Kompilation	354
10.5.4. Plattformspezifische Kompilierung	355
10.5.5. Unterscheidung zwischen C und C++ {#Name_Mangling}	356
10.6. Rückblick und Ausblick	357
11. Benutzerdefinierte und erweiterte Datentypen	357
11.1. Typ-Alias definieren (typedef)	358
11.2. Ganzzahltypen aus <stdint.h>	359
11.2.1. Ganzzahlen mit fester Breite (intN_t und uintN_t) {#intN_t}	361
11.2.2. Minimalbreiten Ganzzahlen (int_leastN_t und uint_leastN_t)	365
11.2.3. Schnellste Ganzzahlen (int_fastN_t und uint_fastN_t)	366
11.2.4. Größte Ganzzahlbreite (intmax_t und uintmax_t)	368
11.2.5. Portable Format-Makros für Ein- und Ausgabe	369
11.2.6. Zusammenfassung	374
11.3. Typen aus <stddef.h>	374
11.3.1. size_t	375
11.4. Optionale Floating-Datentypen {#Optionale_Floating_Datentypen}	375
11.4.1. Warum es optionale Datentypen gibt	376
11.4.2. _FloatN-Typen: Feste IEEE-754-Formate für Portabilität	379
11.4.3. _FloatNx-Typen: Erweiterte Formate für höhere Präzision	382
11.4.4. Dezimale Gleitkommazahlen (_Decimal32, _Decimal64, _Decimal128)	383
11.4.5. Fallback-Strategien	384
11.4.6. Portierbare Float-Aliase (<stdfloat.h>)	385
11.5. Größen, Grenzen und Eigenschaften der Typen	385
11.5.1. Messwerkzeuge zur Ermittlung von Typ-Eigenschaften	386
11.5.2. Grenzen für Ganzzahltypen	386
11.5.3. Portabel definierte Ganzzahltypen	387
11.6. Rückblick und Ausblick	388
12. Komplexe Datentypen	388
12.1. Enumerationen (enum) {#enum}	389
12.1.1. Definition von Enumerationen	389
12.1.2. Variablen vom Typ einer Aufzählung deklarieren und verwenden	390
12.1.3. Enumerationen in switch-case	390
12.1.4. Namensräume und Sichtbarkeit von Enumerationskonstanten	391
12.1.5. Verwendung als Parameter und Rückgabewert	392
12.1.6. Automatische Wertezuweisung und Vergleich	393

12.1.7. Explizite Wertzuweisungen	394
12.1.8. Sentinel-Werte in Enumerationen	396
12.1.9. Anonyme Enum	397
12.1.10. Enumerationskonstanten als konstante Ausdrücke und frühere Nutzungsmuster	398
12.1.11. Implizite und feste Basistypen	400
12.1.12. Vereinfachung von enum-Nutzung durch typedef	401
12.2. Strukturen (struct) {#struct}	403
12.2.1. Deklaration von Strukturen	404
12.2.2. Anlegen von Variablen vom Typ einer Struktur	406
12.2.3. Zugriff auf Strukturelemente	409
12.2.4. Wertweise Kopiersemantik	409
12.2.5. Initialisierung von Strukturen	415
12.2.6. Anonyme Strukturen und verschachtelte Typen	419
12.2.7. Vereinfachung von Strukturnutzung durch typedef	421
12.2.8. Strukturtypen in verschiedenen Übersetzungseinheiten	427
12.3. Unions (union) {#union}	427
12.3.1. Aufbau und Definition einer Union	428
12.3.2. Anlegen von Variablen vom Typ einer Union	429
12.3.3. Zugriff und Übergabe an Funktionen	429
12.3.4. Anonyme Union	430
12.3.5. Union mit typedef	431
12.3.6. Variantentypen	431
12.3.7. Strukturen und Unions kombinieren	433
12.3.8. Tagged Unions	437
12.3.9. Variantentypen für Rückgabewerte (Result-Muster)	440
12.3.10. Type Punning mit Strukturen und Unions	442
12.3.11. Anonyme Unions und Kombination mit Strukturen	445
12.4. Rückblick und Ausblick	449
13. Arrays {#Arrays}	449
13.1. Einführung und Motivation für Arrays	450
13.2. Deklaration von Arrays	450
13.2.1. Arrays mit fester Größe (Compile-Time-Größe)	451
13.2.2. Arrays mit variabler Größe (Variable-Length Arrays, VLAs)	453
13.3. Zugriff auf Array-Elemente	455
13.3.1. Indexnotation	455
13.3.2. Über Arrays iterieren	456
13.3.3. Out-of-Bounds-Zugriffe und Undefined Behavior	457
13.3.4. Sichere Praktiken und Debugging	457
13.3.5. Grenzchecks mit AddressSanitizer	458
13.4. Initialisierung von Arrays	459
13.4.1. Listeninitialisierung	459

13.4.2. ARRAY_SIZE Makro	461
13.4.3. Teilinitialisierung und implizite Nullen	465
13.4.4. Standardinitialisierung mit {0}	466
13.4.5. Zeichenketten als char-Arrays	467
13.4.6. Designated Initializers	469
13.4.7. const Arrays {#const_arrays}	472
13.5. Arrays bei der Parameterübergabe und Rückgabe von Funktionen	474
13.5.1. Arrays an Funktionen übergeben	474
13.5.2. Variable Length Arrays in Funktionsparametern	476
13.5.3. Grenzen von sizeof bei Funktionsparametern	477
13.5.4. const qualifizierte Parameter	477
13.5.5. Minimale Größengarantie mit static	478
13.5.6. Arrays als Rückgabewert: Einschränkungen und Alternativen	481
13.6. Aufgaben	484
13.6.1. Zählen, wie viele Temperaturen einen Grenzwert überschreiten	484
13.6.2. Zufälliges Mischen (Fisher-Yates) einer Zahlenfolge	484
13.6.3. Binäre Suche nach Temperaturwerten (bei sortierten Daten)	485
13.6.4. Zusammenführen der Temperaturreihen zweier Messstationen in ein gemeinsames Array	485
13.6.5. Sortieren von Temperaturwerten mit Counting Sort	486
13.6.6. Analyse monatlicher Temperaturdaten mit Strukturen	487
13.7. Mehrdimensionale Arrays	487
13.7.1. Deklaration mehrdimensionaler Arrays	487
13.7.2. Variable Length Arrays (VLA) mit mehreren Dimensionen	487
13.7.3. Zugriff mit mehreren Indizes	488
13.7.4. Mehrdimensionale Arrays sind Arrays von Arrays	488
13.7.5. Mehrdimensionale Arrays liegen im Speicher hintereinander	490
13.7.6. sizeof bei mehrdimensionalen Arrays	491
13.7.7. Initialisierung von mehrdimensionalen Arrays	493
13.7.8. Mehrdimensionale Arrays an Funktionen übergeben	498
13.7.9. Praktische Beispiele mit mehrdimensionalen Arrays: Tabellen, Brettspiele und mehr	501
13.8. Einbinden externer Inhalte	507
13.9. Rückblick und Ausblick	509
14. Zeiger {#Zeiger}	509
14.1. Datenzeiger und Adressen	509
14.1.1. Speicher und Variablen	509
14.1.2. Adressoperator (&)	512
14.1.3. Zeigervariablen: Adressen speichern	513
14.1.4. Weitere zulässige Anwendungen des Adressoperators	515
14.1.5. Dereferenzierungsoperator (*)	516
14.1.6. Beispiel: Menüsystem mit Umschalten per Zeiger	518

14.1.7. Deklaration mehrerer Zeigervariablen	519
14.1.8. Typumwandlungen zwischen Zeigertypen	521
14.1.9. Das Schlüsselwort volatile {#volatile}	523
14.1.10. Zugriff auf Speicherstellen: Memory-Mapped I/O	524
14.1.11. Zeiger als Ganzzahl interpretieren sowie der Typ uintptr_t	526
14.1.12. Pointer Tagging	527
14.2. Der Null-Pointer nullptr	528
14.2.1. Dereferenzierung einer Null-Pointer-Variablen	529
14.2.2. Zeigervariablen auf nullptr prüfen	529
14.2.3. Typ nullptr_t	530
14.2.4. Das Makro NULL	530
14.3. Zeiger und const: Unveränderlichkeit deklarieren	531
14.3.1. Zeiger auf konstante Daten	531
14.3.2. Konstanter Zeiger	533
14.3.3. Konstanter Zeiger auf konstante Daten	533
14.3.4. Zusammenfassung und Merkregel: const und Zeiger	534
14.4. Zeiger in Funktionen	535
14.4.1. Wertübergabe und Adressübergabe	535
14.4.2. Beispiel: swap()-Funktion mit Zeigern	536
14.4.3. Mehrere Rückgabewerte über Out-Parameter {#out-parameter}	537
14.4.4. Terminologie: In-, Out- und In-Out-Parameter	538
14.4.5. Ergebnisse über Out-Parameter und Status über Rückgabewerte	538
14.4.6. Grund für Zeigerübergabe bei scanf()	539
14.4.7. Zeiger als Rückgabewert	540
14.4.8. Häufige Fehler: Dangling Pointers	541
14.4.9. Zeiger auf Funktionsparameter zurückgeben	542
14.4.10. Das Qualifizierer restrict: Aliasfreiheit und Optimierung {#restrict}	543
14.4.11. Exkurs: Referenzen in C++	544
14.4.12. Best Practices für Zeigerparameter	546
14.5. Void-Zeiger: Typenloses Zeigerwerkzeug {#void-Zeiger}	547
14.5.1. Die Idee des void-Zeigers	547
14.5.2. Implizite und explizite Konvertierung	548
14.5.3. Hexadezimale Ausgabe beliebiger Datentypen	549
14.5.4. Einschränkungen von void *	551
14.6. Zeigerbeziehungen und -operationen	552
14.6.1. Vergleichsarten: Gleichheit und Ordnung	553
14.6.2. Gleichheitsvergleich: Identische Adressen	553
14.6.3. Vergleich über unterschiedliche Zeigertypen und void *	554
14.6.4. Ordnungsvergleiche: Relative Position im Speicher	555
14.6.5. Anwendung: Array-Grenzprüfung	557
14.6.6. Zeigerdifferenzen und ptrdiff_t	557

14.6.7. Zeigerprüfungen in Bedingungen	558
14.7. Array-Decay und Zeigerarithmetik {#Array_Decay}	559
14.7.1. Array-Decay: implizite Umwandlung	560
14.7.2. Unterschiede zwischen Array-Namen und echten Zeigern	562
14.7.3. Zeigerarithmetik: Rechnen mit Adressen	562
14.7.4. Index-Notation vs. Zeiger-Notation	565
14.7.5. Iteration mit Zeigern	566
14.7.6. Array-Größe in Funktionen und der Effekt des Array-Decay {#sizeof-array-decay}	569
14.7.7. Mehrdimensionale Arrays und Zeiger	570
14.8. Zugriff auf Strukturen über Zeiger	574
14.8.1. Der Pfeiloperator (->)	575
14.8.2. Funktionen mit Strukturzeigern	577
14.8.3. Verschachtelte Strukturen und mehrfache Pfeile	578
14.8.4. Zeiger auf Strukturen und verkettete Listen {#verkettete_Listen}	580
14.8.5. Doppelt verkettete Listen	582
14.9. Arrays von Zeigern	583
14.9.1. Deklaration und Initialisierung	584
14.9.2. Arrays von Zeichenketten	586
14.9.3. Verwendung mit Null-Pointer als Sentinel	588
14.9.4. Beispiel: Nachrichtenverwaltung mit Arrays von Zeigern auf Strukturen	589
14.9.5. Beispiel: Verlaufsliste mit Array von Zeigern	591
14.10. Mehrstufige Indirektion mit Zeigern auf Zeigern	592
14.10.1. Zeiger auf Zeiger deklarieren {#Zeiger_auf_Zeiger}	593
14.10.2. Konstanz bei mehrstufiger Indirektion	594
14.10.3. Warum mehrstufige Indirektion?	595
14.10.4. Unterschied: Array von Zeigern versus Zeiger auf Zeiger	595
14.10.5. Funktionen mit Arrays von Zeigern	596
14.10.6. Zweidimensionale Arrays als Zeiger auf Zeiger	597
14.10.7. Zeiger in Funktionen verändern	599
14.10.8. Beispiel: Verkettete Listen und Zeiger auf Zeiger	600
14.10.9. Mehrdimensionale Zeiger-Strukturen	603
14.11. Kommandozeilenargumente als Array von Zeigern	605
14.11.1. Programmargumente auswerten	607
14.12. Funktionszeiger {#Funktionszeiger}	608
14.12.1. Deklaration und Syntax	609
14.12.2. Adresse einer Funktion speichern	611
14.12.3. Aufruf über Funktionszeiger	613
14.12.4. Vereinfachte Syntax durch Function-to-Pointer Decay	614
14.12.5. Typedef für bessere Lesbarkeit	615
14.12.6. Namensräume durch Strukturen	616
14.12.7. Funktion höherer Ordnung	620

14.12.8. Arrays von Funktionszeigern für Sprungtabellen	625
14.12.9. Vergleichsfunktionen	626
14.12.10. Suchen und Sortieren über die Standardbibliothek	630
14.13. Rückblick und Ausblick	634
15. Speicherverwaltung und Fehlersicherheit {#Speicherverwaltung}	635
15.1. Einführung in dynamische Speicherverwaltung	635
15.1.1. Stack vs. Heap: Eigenschaften, Lebensdauer und Anwendungsfälle	636
15.2. Grundlegende Allokationsfunktionen und Anwendungen {#malloc}	637
15.2.1. Speicher allozieren (malloc()) {#malloc_sizeof}	637
15.2.2. Freigabe von Speicher (free())	639
15.2.3. Speicher für Arrays allozieren	641
15.2.4. Speicher für Strukturen allozieren	643
15.2.5. Speicher für Arrays von Strukturen allozieren	645
15.2.6. Dynamisch verkettete Listen implementieren {#dynamisch-verkette-liste}	646
15.2.7. Allokation mit Initialisierung auf Null (calloc())	657
15.2.8. Erweiterung und Verkleinerung von Speicherbereichen (realloc())	658
15.2.9. Sichere Größenberechnung	663
15.3. Arbeiten mit Speicherbereichen {#mem_Funktionen}	665
15.3.1. Speicher kopieren	665
15.3.2. Speicher vergleichen	667
15.3.3. Strukturen vergleichen	668
15.3.4. Speicher setzen	670
15.3.5. Einzelne Bytes suchen (memchr())	672
15.4. Flexible Array Members (FAM)	674
15.4.1. Syntax und Verwendung von Flexible Array Members	675
15.4.2. Fazit und Zusammenfassung	676
15.5. Speicherlebenszyklen und Ownership-Modelle {#Ownership}	676
15.5.1. Ownership in C: Verantwortlichkeiten und Konventionen	676
15.5.2. Referenzzählung	679
15.5.3. Scope-basierte Ressourcenverwaltung mit Cleanup-Attributen {#Cleanup-Funktion}	681
15.6. Typische Speicherfehler und ihre Vermeidung {#Typische_Speicherfehler}	683
15.6.1. Null-Pointer-Checks	684
15.6.2. Buffer Overflows und Underflows bei dynamischem Speicher	688
15.6.3. Memory Leaks (Speicherlecks)	690
15.6.4. Dangling Pointers und Use-after-Free	697
15.6.5. Double-Free und Invalid-Free	701
15.6.6. Übersicht und Zusammenfassung über Undefined Behavior im Speichermanagement	704
15.7. Fragmentierung und Out-of-Memory-Handling	705
15.8. Alignment und Padding {#Alignment}{#Padding}	705
15.9. Debugging und Analyse von Speicherfehlern	706

15.9.1. Address Sanitizer {#AddressSanitizer}	708
15.9.2. Memory Sanitizer (MSan) {#MemorySanitizer}	714
15.9.3. Valgrind	716
15.9.4. Core Dumps und Post-Mortem-Debugging	718
15.9.5. Wann welches Tool einsetzen?	718
15.9.6. Weitere Tools	719
15.9.7. Entwicklung und heutiger Stand	720
15.10. Allokatoren	720
15.10.1. Wie malloc() sich Speicher holt	721
15.10.2. Implementierungen von malloc()/free() in modernen Systemen	723
15.10.3. Eigene Allokatoren schreiben	724
15.10.4. Custom Allocatoren parametrisieren	729
15.10.5. Performance-Überlegungen: Heap vermeiden (Stack-Allokatoren, VLAs)	730
16. Metaprogrammierung {#Metaprogrammierung}	732
16.1. Das Problem: Generischer Code ohne Overhead	732
16.1.1. Was ist Metaprogrammierung?	733
16.1.2. Die drei Phasen der Übersetzung	733
16.2. Präprozessor-Metaprogrammierung	734
16.2.1. Portable Konfiguration und Plattformerkenung	734
16.2.2. Feature-Erkennung	735
16.2.3. Standard-Makros und Metainformationen	737
16.3. Compilerbasierte Metaprogrammierung {#Compilerbasierte_Metaprogrammierung}	737
16.3.1. Typprüfungen mit static_assert {#static_assert}	738
16.3.2. Generische Makros mit typeof und typeof_unqual	741
16.3.3. Das container_of-Idiom	744
16.3.4. Typabhängige Auswahl mit _Generic {#_Generic}	747
16.3.5. Optimierungshinweis unreachable()	756
16.4. Zusammenfassung und Ausblick	758
17. Zeichen, Zeichenketten und Textverarbeitung {#Strings}	759
17.1. Historische Grundlagen der Textrepräsentation	759
17.1.1. char, signed char, unsigned char	759
17.1.2. Codepages und das klassische C-Zeichenmodell	760
17.2. Von einzelnen Zeichen zu Zeichenketten	761
17.2.1. Null-Terminierung von Strings	761
17.2.2. Klassische String-Literale	762
17.2.3. Die Transformation von String-Literalen in C	763
17.2.4. Zeiger vs. Array	765
17.2.5. Speicherorte für nicht-literale Strings	766
17.2.6. Umgang mit konstanten Strings	766
17.3. Moderne Textrepräsentation {#Moderne_Stringrepräsentation}	767
17.3.1. Unicode als abstraktes Zeichensystem	767

17.3.2. Codierungen	767
17.3.3. Anzeigeebene: Wie Zeichen erscheinen	768
17.4. Moderne Texttypen und Literale	769
17.4.1. char32_t	769
17.4.2. char8_t	770
17.4.3. char16_t	770
17.4.4. Historischer Typ wchar_t	771
17.4.5. Fazit	771
17.4.6. Unicode-Literale in C23 {#Unicode-Literale}	771
17.4.7. Universal Character Names (UCN)	773
17.5. Moderne String-Repräsentationen	775
17.5.1. Strings mit Längenpräfix	775
17.5.2. Small-String-Konzepte	776
17.5.3. String-View	777
17.6. Memory-Management-Techniken	779
17.6.1. String-Pools	779
17.6.2. Globales Interning	781
17.6.3. Flyweight-Pattern	781
17.7. Dynamische Strings	783
17.7.1. String mit Kapazität	783
17.7.2. Wachstumsstrategien und Kostenmodell	785
17.7.3. Arena-basierte dynamische Strings	786
17.7.4. Vergleich mit *sprintf()-Funktionen	786
17.8. Locale: Einfluss auf Zeichen- und String-Funktionen	787
17.8.1. Locale-Kategorien	787
17.8.2. Locale setzen	788
17.8.3. Beispiel: Byte 0xE4 in verschiedenen Locales	788
17.8.4. Probleme des globalen Locale-Modells	789
17.8.5. Formatierungsdaten abfragen	790
17.8.6. Praxis: Locale sparsam einsetzen	790
17.9. Formatierte Ausgabe in Strings	791
17.9.1. Formatfunktionen snprintf()/vsprintf()/sprintf_s()/snprintf_s() {#Formatfunktionen}	791
17.9.2. sprintf und vsprintf	792
17.9.3. String-Konkatenation mit Formatfunktionen	793
17.9.4. Automatische Speicherallokation mit GNU-Funktionen asprintf() und vasprintf()	794
17.9.5. Formatspezifizierer	795
17.9.6. Einzelne Werte in Strings konvertieren	796
17.9.7. Compilerprüfungen bei Formatstrings	797
17.9.8. Formatstring-Prüfung für eigene Funktionen in GCC/Clang	799
17.9.9. Format-String-Vulnerabilities	800
17.9.10. Sichere Nutzung der printf-Familie	801

17.10. Parsing und Konvertierung	802
17.10.1. Tokenisierung: Text in Einheiten zerlegen	802
17.10.2. Tokenisierung mit Standardfunktionen (strtok*())	803
17.10.3. Parser selbst entwickeln	804
17.10.4. Zeichenketten in Zahlen umwandeln (strto*())	804
17.10.5. Verkürzte Varianten atoi(), atol(), atoll() vermeiden	808
17.10.6. Strukturierte Eingaben auslesen (sscanf*())	808
17.11. Klassische Charakterklassifikation (is*()) und weitere)	811
17.12. Unicode in C23	811
17.12.1. UTF-8 selbst verarbeiten	812
17.12.2. Die Scheinlösung: Wide Characters	814
17.12.3. Externe Bibliotheken	814
17.13. Standard-Stringfunktionen für nullterminierte Zeichenketten	815
17.13.1. Die zwei grundlegenden Schwächen der klassischen C-Strings	816
17.13.2. Nur lesende Stringfunktionen	817
17.13.3. Schreibende Funktionen ohne Längenparameter	818
17.13.4. Schreibende Funktionen mit Längenparameter	819
17.13.5. Dynamische Stringfunktionen	820
17.13.6. Eigene Wrapper mit __builtin_object_size	820
17.13.7. Beispiel: Dateiname und Dateiendung aus einem Pfad extrahieren	822
17.13.8. mem*()-Funktionen für String-Verarbeitung nutzen	823
17.13.9. Stringfunktionen mit verschiedenen String-Literal-Typen?	825
17.14. Vergleiche und Hashing	826
17.14.1. Grundformen des Stringvergleichs	826
17.14.2. Case-insensitive Vergleiche	827
17.14.3. Locale-abhängige Vergleiche	828
17.14.4. Natürliche Sortierung	829
17.14.5. Hashfunktionen für schnelle String-Vergleiche	830
17.15. Testing, Fuzzing und Verifikation von C-String-Code	832
17.15.1. Stringbezogene Fehler erkennen	832
17.15.2. Speicherbezogene Fehler erkennen	833
17.16. Rückblick und Ausblick	833
18. Ein-/Ausgabe (I/O)	834
18.1. Grundlagen von Streams	834
18.2. Dateien öffnen und schließen	835
18.2.1. Eine Datei öffnen und schließen (fopen() und fclose())	835
18.3. Textdateien schreiben und lesen	837
18.3.1. Text schreiben (fprintf(), fputs(), putc() usw.)	837
18.3.2. Text einlesen (fscanf(), fgets(), getc() usw.)	837
18.4. Standardstreams	839
18.4.1. Kurzformen für Ein- und Ausgabe	839

18.4.2. Einen Stream neu zuordnen (freopen())	840
18.5. Rohdaten verarbeiten	840
18.5.1. Strukturen binär schreiben und lesen	842
18.6. Position im Stream steuern (fseek(), ftell() usw.)	844
18.7. Fehler erkennen und behandeln (ferror(), feof(), clearerr())	847
18.8. Puffer und Streameigenschaften steuern	848
18.8.1. Puffer einstellen (setvbuf())	849
18.8.2. Puffer leeren (fflush())	851
18.9. Dateien löschen und umbenennen	851
18.9.1. Eine Datei löschen (remove())	851
18.9.2. Eine Datei umbenennen (rename())	852
18.10. Temporäre Dateien	852
18.10.1. Eine temporäre Datei öffnen (tmpfile())	853
18.10.2. Sicheren temporären Dateinamen erzeugen (mkstemp())	854
18.11. Plattform- und POSIX-Erweiterungen	854
18.11.1. Low-Level Ein-/Ausgabe mit Dateideskriptoren	855
18.11.2. Umwandlung zwischen Deskriptoren und FILE *-Streams	855
18.12. POSIX-Erweiterung für eine Ausgabe-Funktionalität	855
18.12.1. Dateisystem-Metadaten und Dateitypen	856
18.12.2. Rechte, Besitzer und Erstellungsmaske	856
18.12.3. Verzeichnisse lesen und verwalten	856
18.12.4. Links, Umbenennen und Löschen	857
18.12.5. Low-Level-I/O mit Dateideskriptoren	857
18.12.6. Dateigröße ändern und Synchronisation	857
18.12.7. Dateideskriptor-Steuerung und Flags	858
18.12.8. Multiplexing und nicht-blockierendes I/O	858
18.12.9. Speicherabbildung von Dateien	858
18.12.10. Speicher-basierte Streams	858
18.12.11. Zeitstempel von Dateien setzen	859
18.13. Zusammenfassung und Ausblick	859
19. Bitoperationen {#Bit-Manipulation}	859
19.1. Bit- und Schiebeoperatoren	861
19.1.1. Bitweises UND (&)	861
19.1.2. Bitweises ODER ()	862
19.1.3. Bitweises XOR (^) {#XOR}	862
19.1.4. Bitweises NICHT (~)	862
19.1.5. Schiebeoperatoren (<<, >>) {#Schiebeoperatoren}	863
19.1.6. Häufige Fehler bei bitweisen Operatoren	864
19.2. Bitmasken, Flags und typische Muster	864
19.2.1. Bitmasken und ihre Konstruktion	865
19.2.2. Setzen einzelner Bits	866

19.2.3. Löschen einzelner Bits	866
19.2.4. Umschalten einzelner Bits	867
19.2.5. Prüfen einzelner Bits	868
19.2.6. Mehrere Bits gleichzeitig prüfen	869
19.2.7. Enum als Bitmasken	869
19.2.8. Häufige Fehler und Best Practices	873
19.2.9. Zusammenfassung des Musters	874
19.3. Standardisierte Bitfunktionen (<stdbit.h>)	874
19.3.1. Verfügbarkeit des Headers prüfen	874
19.3.2. Übersicht der Funktionen	875
19.4. Ganzzahlen mit fester Bitbreite (_BitInt) {#_BitInt}	876
19.4.1. Beispiel und Überlaufverhalten	877
19.4.2. printf() und Typcast	877
19.4.3. _BitInt in Strukturen	879
19.5. Bitfelder {#Bitfelder}	879
19.5.1. Bitfelder definieren	880
19.5.2. Beispiel: IPv4-Header mit Bitfeld	882
19.5.3. Abbildung und Padding von Bitfeldern	883
19.6. Rückblick und Ausblick	885
20. Numerik und Mathematik {#Numerik}	886
20.1. Sichere Arithmetik (<stdckdint.h>) {#stdckdint}	886
20.1.1. Wertebereiche und Überlaufprobleme	886
20.1.2. Strategien zur Überlauferkennung	887
20.1.3. Checked Integer Arithmetic	888
20.1.4. Überlauf bei Gleitkommazahlen	889
20.2. Besondere Gleitkommazahlen	890
20.2.1. NaN	890
20.2.2. Vorzeichenbehaftete Null (+0.0 und -0.0)	891
20.2.3. Unendlichkeiten und Division durch null	892
20.2.4. Subnormale Zahlen	892
20.2.5. Klassifikation von Gleitkommazahlen {#isnan}	893
20.2.6. Grenzen und Epsilon	894
20.3. Konvertierungen zwischen Ganz- und Gleitkommazahlen	896
20.3.1. Cast von Ganzzahlen in Gleitkommazahlen	896
20.3.2. Casts von Gleitkommazahlen in Ganzzahltypen	897
20.3.3. Rundung beim Cast von Gleitkommazahlen in Ganzzahlen	897
20.4. Mathematische Funktionen (<math.h>)	899
20.4.1. Demonstration ausgewählter Funktionen	899
20.4.2. Übersicht der wichtigsten Funktionen	900
20.4.3. Linken der Mathematik-Bibliothek (-lm)	901
20.4.4. Aufgabe	902

20.5. Floating-Point-Umgebung und Rundungsmodi (<fenv.h>) {#fenv}	902
20.5.1. IEEE-754- und IEC-60559-Konformität in C23	902
20.5.2. fesetround() und ihre Auswirkungen {#fesetround}	903
20.5.3. Floating-Point-Exceptions und Status-Flags	904
20.5.4. Steuerung der FP-Umgebung in der Praxis	906
20.6. Generische Mathematik (<tgmath.h>) {#tgmath}	906
20.6.1. Beispielnutzung mit Sinus	906
20.6.2. Übersicht der typgenerischen Makros	907
20.7. Zufallsfunktionen in C	908
20.8. Komplexe Zahlen	909
20.8.1. Datentypen float complex, double complex und long double complex	910
20.8.2. Operatoren	910
20.8.3. Funktionen	910
20.8.4. Beispiel: Mandelbrot-Menge	910
20.9. Rückblick und Ausblick	911
21. Weitere Teile der C-Standardbibliothek	912
21.1. Standard Utilities Header (<stdlib.h>)	912
21.1.1. Überblick: bereits bekannte Teile	913
21.1.2. Programm beenden	913
21.1.3. Umgebungsvariablen lesen (getenv())	914
21.1.4. POSIX-Erweiterungen für Umgebungsvariablen	915
21.1.5. Externe Befehle ausführen (system())	915
21.2. Signale (<signal.h>)	916
21.3. Nichtlokale Sprünge (<setjmp.h>) {#Nichtlokale_Sprünge}	917
21.3.1. Kontext speichern (jmp_buf)	917
21.3.2. Rücksprungpunkt setzen (setjmp())	918
21.3.3. Sprung ausführen (longjmp())	918
21.3.4. Regeln und Fallstricke	919
21.3.5. Try/Catch per Makros (setjmp()/longjmp())	920
21.4. Zeit und Datum (<time.h>)	923
21.4.1. Zeitdarstellung und grundlegende Typen	923
21.4.2. Umwandlung zwischen time_t und struct tm	924
21.4.3. Zeitspannen messen (Wandzeit)	926
21.4.4. Prozessorzeit messen (CPU-Zeit)	929
21.4.5. Zeit formatieren (strftime())	930
21.4.6. Zeitzonen und Sommerzeit (POSIX-Erweiterung)	931
21.5. Rückblick und Ausblick	932
22. Nebenläufigkeit und Parallelität	933
22.1. Einführung in Nebenläufigkeit	934
22.1.1. Warum Nebenläufigkeit? Herausforderungen und Anwendungsfälle	934
22.1.2. CPU-bound vs. I/O-bound Tasks	936

22.1.3. Amdahl's Law und die Grenzen der Parallelisierung	938
22.1.4. Threads vs. Prozesse	939
22.2. Thread-Lebenszyklus: Erstellen, Ausführen, Beenden	941
22.2.1. Thread-Bibliothek <code><threads.h></code>	941
22.2.2. Erstellung eines Threads	942
22.2.3. Freiwillig Rechenzeit abgeben	944
22.2.4. Wer bin ich? Aktuellen Thread ermitteln	946
22.2.5. Threads schlafen legen	946
22.2.6. Natürliches und unnatürliches Ende eines Threads	947
22.2.7. Der Hauptthread und Programmende	951
22.2.8. Detached Threads	954
22.2.9. Kooperatives Beenden von Threads durch Signalisierung	955
22.2.10. Best Practices für den Thread-Lebenszyklus	955
22.3. Das C-Memory-Modell und atomare Operationen	956
22.3.1. Das C-Memory-Modell: Sequentielle Konsistenz und Happens-Before	956
22.3.2. Atomare und nicht-atomare Speicherzugriffe	956
22.3.3. Data Race	957
22.3.4. Ordnung und Korrektheit von Speicherzugriffen im C-Memory-Modell	959
22.4. Fortgeschrittene Konzepte — kurz vorgestellt	959
22.4.1. Atomare Typen und grundlegende Operationen (<code><stdatomic.h></code>)	960
22.4.2. Synchronisation mit Mutex	960
22.4.3. Condition Variables (<code>cond_t</code>)	960
22.4.4. Thread-lokaler Speicher (<code>thread_local</code>)	960
22.5. Alternativen zu <code><threads.h></code>	961
22.5.1. pthreads als faktischer Referenzpunkt	961
22.5.2. Plattformabhängige Alternativen	963
22.5.3. Konsequenzen für den Entwurf	963
22.6. Rückblick und Ausblick	963
23. Standardbibliothek, Implementierungen und Erweiterungen <code>{#weitere_libs}</code>	965
23.1. Implementierungen der Standardbibliothek	965
23.1.1. Systemabhängige Implementierungen der <code>libc</code>	965
23.1.2. Schlanke Implementierungen für eingebettete Systeme	966
23.2. POSIX	966
23.2.1. POSIX Funktionsbereiche	967
23.2.2. POSIX-Header	968
23.3. Plattformunabhängige Bibliotheken	970
23.4. Plattformabstrahierende Bibliotheken	972
23.4.1. Konkrete Beispiele plattformabstrahierender Bibliotheken	972
23.5. Plattformspezifische APIs jenseits von C und POSIX	973
23.5.1. Typische Funktionsbereiche plattformspezifischer APIs	974
23.5.2. Toolchains und Kompilierung	974

23.5.3. Konsequenzen für C-Programme	974
23.6. Compiler- und Implementierungserweiterungen	975
23.6.1. Typische Erweiterungen	975
23.7. Zusammenfassung und Ausblick	976
24. Build und Auslieferung	976
24.1. Make	978
24.1.1. Aufbau eines Makefiles	978
24.1.2. Erstes Beispiel	978
24.1.3. Ziele über die Kommandozeile ansprechen	979
24.1.4. Abhängigkeiten	980
24.1.5. Variablen und automatische Variablen	980
24.1.6. Parameterübergabe über die Shell	981
24.1.7. Ausblick: pkg-config	982
24.1.8. Ausblick: Autotools	983
24.2. Projektstruktur in C	985
24.2.1. Makefile unter Berücksichtigung der Verzeichnisstruktur	987
24.3. CMake	988
24.3.1. Ein erstes Beispiel	989
24.4. Cross-Compiler	990
24.5. Packaging und Distribution	991
24.5.1. Statisches vs. dynamisches Linken	991
24.5.2. Verteilungsformen	991
24.5.3. Versionierung	992
24.6. Continuous Integration	992
24.7. Zusammenfassung und Ausblick	993
25. Testen und Qualitätssicherung	993
25.1. Statische Analyse	993
25.2. Code-Formatter	994
25.2.1. clang-format	994
25.3. Tests	995
25.3.1. Unity	995
25.3.2. Regressionstests	997
25.4. Code Coverage	997
25.5. Fuzzing	998
25.6. Review-Prozesse	999
25.7. Dokumentation	1000
25.7.1. Doxygen	1000
25.7.2. Was nicht dokumentiert wird	1001
25.8. Zusammenfassung und Ausblick	1001
26. Profiler und Optimierung	1002
26.1. Compiler-Optimierungen {#Compiler-Optimierungen }	1002

26.1.1. Optimierungsstufen: Steuerung durch Compiler-Schalter	1003
26.1.2. Optimierungen nach Reichweite	1004
26.1.3. Speicher und Aliasing als Optimierungsgrenze	1004
26.1.4. Fazit: Was bedeutet das für die Praxis?	1005
26.2. Grundlagen und Kennzahlen der Leistungsanalyse	1006
26.2.1. Laufzeit, Durchsatz, Latenz	1007
26.2.2. Messfehler, Störeinflüsse	1007
26.2.3. Warmup, Cache, Branch Prediction	1008
26.2.4. Messbare Kennzahlen und Messmethodik	1008
26.3. Werkzeuge	1009
26.3.1. Profiler	1009
26.3.2. Tracer	1010
26.3.3. Counter-basierte Werkzeuge	1012
26.3.4. Einordnung und Arbeitsweise	1012
26.4. Benchmarking und Reproduzierbarkeit	1013
26.4.1. Benchmarking als Experiment	1013
26.4.2. Micro- und Macro-Benchmarks als Messlinse	1013
26.4.3. Reproduzierbarkeit als Protokollfrage	1014
26.4.4. Vergleichbarkeit und Versuchsbedingungen	1014
26.4.5. Messreihen und Driftkontrolle	1014
26.4.6. Typische Benchmark-Artefakte	1015
26.4.7. Benchmark-Frameworks	1015
26.5. Optimierungstechniken im Code	1016
26.5.1. Algorithmische Optimierung	1016
26.5.2. Datenlayout und Cache-Lokalität	1017
26.5.3. Parallelisierung und SIMD	1018
26.5.4. Kontrollfluss und Mikrooptimierungen	1020
26.5.5. Die goldene Reihenfolge	1021
26.6. Methodik: Systematisches Vorgehen bei Performance-Problemen	1022
26.7. Rückblick und Ausblick	1024
27. Nachwort: Der ›C-Way‹: Philosophie und Kernprinzipien	1024
Anhang A: Standard-Header	1026
Anhang B: Die 8 Übersetzungsphasen	1028
Anhang C: Operatorpräzedenz	1032

1. Vorwort

1.1. Warum dieses Buch?

Die Programmiersprache C ist über 50 Jahre alt, so alt wie ich. In der Geschichte der Informatik ist das eine enorme Zeitspanne. Und dennoch: Auch heute kommt man an C kaum vorbei. Viele