

Dr. Helmut Herold

Dr. Jörg Arndt

C-Programmierung unter Linux / UNIX / Windows

**Beispiele, Anwendungen,
Programmiertechniken**



Alle in diesem Buch enthaltenen Programme, Darstellungen und Informationen wurden nach bestem Wissen erstellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund ist das in dem vorliegenden Buch enthaltene Programm-Material mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials, oder Teilen davon, oder durch Rechtsverletzungen Dritter entsteht.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann verwendet werden dürften.

Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Andere hier genannte Produkte können Warenzeichen des jeweiligen Herstellers sein.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Druck, Fotokopie, Microfilm oder einem anderen Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.
ISBN 3-938626-17-7

© 2010 Nicolaus Millin Verlag GmbH, Lohmar (<http://www.millin.de>)

Umschlaggestaltung: millin Verlag, <http://www.millin.de>

Gesamtlektorat: Nicolaus Millin

Fachlektorat: Dieter Bloms, Jörg Dippel, Klaas Freitag, Bruno Gerz, Bernhard Hoelcker, Björn Jacke, Andreas Jaeger, Dirk Pankonin, Wolfgang Rosenauer, Christian Steinrücken, Peter Varkoly

Satz: L^AT_EX

Druck: druckhaus köthen GmbH, Köthen (<http://www.koethen.de>)

Printed in Germany on acid free paper.

Inhaltsverzeichnis

| | | |
|----------|---|-----------|
| 1 | Einführendes Beispiel | 3 |
| 1.1 | Erste wesentliche C-Regeln | 3 |
| 1.2 | Zeilen-Kommentare mit // (neu in C99) | 5 |
| 1.3 | Gute Lesbarkeit von Programmen | 5 |
| 1.4 | Vermeiden von geschachtelten Kommentaren | 6 |
| 2 | Elementare Datentypen | 7 |
| 2.1 | Die Grunddatentypen in C | 7 |
| 2.2 | Wertebereiche für die einzelnen Datentypen | 9 |
| 2.3 | Fallgrube: Verlust von Bits bei zu großen Zahlen | 10 |
| 3 | Konstanten | 11 |
| 3.1 | char-Konstanten | 11 |
| 3.2 | Ganzzahlige Konstanten | 11 |
| 3.3 | Gleitpunktkonstanten | 12 |
| 4 | Variablen | 13 |
| 4.1 | Variablen und die C-Regeln für Variablennamen | 13 |
| 4.2 | Tipps zur Wahl der Variablennamen | 14 |
| 4.3 | Deklaration von Variablen | 14 |
| 4.4 | Tipp: Variablen bereits bei Deklaration dokumentieren | 16 |
| 5 | Ausdrücke und Operatoren | 17 |
| 5.1 | Der einfache Zuweisungsoperator | 17 |
| 5.1.1 | Allgemeines zum einfachen Zuweisungsoperator | 17 |
| 5.1.2 | Initialisierung von Variablen | 19 |
| 5.2 | Arithmetische Operatoren | 19 |
| 5.2.1 | Die arithmetischen Operatoren | 19 |

| | | |
|-------|--|----|
| 5.2.2 | Die C-Begriffe Ausdruck und Anweisung | 20 |
| 5.2.3 | Ausgabe von int-Variablen und -Ausdrücken | 21 |
| 5.2.4 | Ausgabe von Gleitpunkt-Variablen und -Ausdrücken | 21 |
| 5.2.5 | Fallgrube: Ganzzahl- statt Gleitpunktdivision | 22 |
| 5.3 | Vergleichsoperatoren | 23 |
| 5.3.1 | Die unterschiedlichen Vergleichsoperatoren | 23 |
| 5.3.2 | Die zwei Wahrheitswerte von Vergleichen | 23 |
| 5.3.3 | Prioritäten der Vergleichsoperatoren | 23 |
| 5.4 | Logische Operatoren | 24 |
| 5.4.1 | TRUE und FALSE in C | 24 |
| 5.4.2 | Der Datentyp _Bool (neu in C99) | 24 |
| 5.4.3 | Die C-Operatoren für NOT, AND und OR im Überblick | 25 |
| 5.4.4 | Der Negations-Operator ! | 25 |
| 5.4.5 | Der AND-Operator && | 26 |
| 5.4.6 | Der OR-Operator | 26 |
| 5.4.7 | Die Priorität der logischen Operatoren | 26 |
| 5.4.8 | Keine unnötige Auswertung rechts von &&und | 27 |
| 5.4.9 | Übung: Überprüfungen mit logischen Operatoren | 28 |
| 5.5 | Bit-Operatoren | 28 |
| 5.5.1 | Bitweise Invertierung mit ~ | 28 |
| 5.5.2 | Bitweise AND-Verknüpfung mit & | 30 |
| 5.5.3 | Bitweise OR-Verknüpfung mit | 31 |
| 5.5.4 | Bitweise XOR-Verknüpfung mit ^ | 33 |
| 5.5.5 | Bit-Operatoren nur für ganzzahlige Datentypen erlaubt | 34 |
| 5.5.6 | Fallgruben | 34 |
| 5.5.7 | Übung: Überprüfungen mit Bit-Operatoren | 36 |
| 5.6 | Shift-Operatoren | 37 |
| 5.6.1 | Die beiden Shift-Operatoren <<und >> | 37 |
| 5.6.2 | Shift-Operatoren nur für ganzzahlige Datentypen erlaubt | 38 |
| 5.6.3 | Priorität der Shift-Operatoren | 38 |
| 5.7 | Zusammengesetzte Zuweisungsoperatoren | 39 |
| 5.7.1 | Die zusammengesetzten Zuweisungsoperatoren | 39 |
| 5.8 | Inkrement- und Dekrement-Operatoren | 40 |
| 5.8.1 | Inkrementieren und Dekrementieren mit ++ und -- | 40 |
| 5.8.2 | Präfix- und Postfix-Schreibweise für ++ und -- | 40 |
| 5.8.3 | ++ und -- ist nur für Variablen erlaubt | 42 |
| 5.8.4 | ++ und -- ist nicht auf linken Seite einer Zuweisung erlaubt | 42 |

| | | |
|----------|--|-----------|
| 5.9 | Prioritätstabelle für Operatoren | 42 |
| 5.10 | Assoziativität der Operatoren | 43 |
| 5.11 | Erlaubte und unerlaubte Operationen für C-Datentypen | 43 |
| 5.12 | Priorität und Auswertungszeitpunkt bei ++ und -- | 43 |
| 5.13 | Fallgrube: Zugriff auf nicht vorbesetzte Variablen | 45 |
| 5.14 | Übungen | 46 |
| 6 | Symbolische Konstanten | 47 |
| 6.1 | Konstanten-Definition mit #define | 47 |
| 6.1.1 | Die Direktive #define | 47 |
| 6.1.2 | Regeln für Konstanten-Namen bei #define | 49 |
| 6.1.3 | Konstanten machen Programm leicht änderbar | 49 |
| 6.2 | Konstanten-Definition mit const | 49 |
| 7 | Ein- und Ausgabe | 51 |
| 7.1 | Headerdateien und #include | 51 |
| 7.1.1 | Bibliotheken und Headerdateien | 51 |
| 7.1.2 | Eigene Headerdateien | 52 |
| 7.2 | Ein- und Ausgabe eines Zeichens | 53 |
| 7.2.1 | getchar() und putchar() | 53 |
| 7.2.2 | Gepufferte Eingabe bei getchar() | 53 |
| 7.2.3 | Puffer-Bereinigung mit Dummy getchar() | 57 |
| 7.2.4 | Puffer-Bereinigung ist nicht immer notwendig | 58 |
| 7.2.5 | Fallgrube: Zahlen nicht mit getchar() einlesen | 60 |
| 7.2.6 | Die Headerdatei <ctype.h> | 60 |
| 7.2.7 | Einfache Makros | 62 |
| 7.3 | Die Ausgabe mit printf() | 67 |
| 7.3.1 | Die Funktion printf() | 67 |
| 7.3.2 | Fallgruben | 74 |
| 7.3.3 | Tipps | 75 |
| 7.4 | Die Eingabe mit scanf() | 77 |
| 7.4.1 | Die Funktion scanf() | 77 |
| 7.4.2 | Fallgruben | 83 |
| 7.4.3 | Die Headerdatei <math.h> | 86 |
| 7.4.4 | Fallgrube: Vergessen von #include <math.h> | 88 |
| 8 | Datentypumwandlungen | 91 |
| 8.1 | Implizite Datentypumwandlungen | 91 |

| | | |
|-----------|---|------------|
| 8.1.1 | Der sizeof-Operator | 91 |
| 8.1.2 | Implizite Datentypumwandlungen | 93 |
| 8.1.3 | Fallgrube: Zuweisen von Ganzzahlausdrücken an Gleitpunktvariablen | 97 |
| 8.2 | Explizite Datentypumwandlungen | 98 |
| 8.2.1 | Explizite Datentypumwandlungen mit cast-Operator | 98 |
| 8.2.2 | Fallgruben | 98 |
| 9 | Die Headerdateien <limits.h und <float.h> | 101 |
| 9.1 | <limits.h> - Grenzwerte von Ganzzahltypen | 101 |
| 9.2 | <float.h> – Grenzwerte von Gleitpunkt-Datentypen | 102 |
| 10 | Anweisungen und Blöcke | 103 |
| 11 | Die if-Anweisung | 105 |
| 11.1 | Die zweiseitige if-Anweisung | 105 |
| 11.2 | Die einseitige if-Anweisung | 110 |
| 11.3 | Verschachtelte if-Anweisungen | 112 |
| 11.4 | Tipp: Einrücken untergeordneter Programmteile | 113 |
| 11.5 | Fallgruben | 114 |
| 11.5.1 | Falsche Gleichheitsüberprüfung | 114 |
| 11.5.2 | Keine unnötige Auswertung rechts von && und | 115 |
| 11.5.3 | Vergleiche von negativen Zahlen mit unsigned-Variablen | 116 |
| 11.5.4 | Hohe Priorität des Negations-Operators ! | 116 |
| 11.6 | Programmiertechniken | 117 |
| 11.6.1 | if-Kaskaden | 117 |
| 12 | Die bedingte Bewertung ?: | 119 |
| 13 | Die switch-Anweisung | 121 |
| 13.1 | Die switch-Anweisung | 121 |
| 13.2 | Fallgrube: case-Marken müssen ganzzahlige Konstanten sein | 125 |
| 13.3 | Tipps | 126 |
| 13.3.1 | Alle case-Marken (auch letzte) mit break abschließen | 126 |
| 13.3.2 | default immer angeben | 126 |
| 14 | Der Komma-Operator | 127 |
| 14.1 | Der Komma-Operator | 127 |
| 14.2 | Komma-Operator hat die niedrigste Priorität | 128 |

| | |
|--|------------|
| 15 Die for-Anweisung | 129 |
| 15.1 Die for-Anweisung | 129 |
| 15.2 Die for-Schleife und der Komma-Operator | 133 |
| 15.3 Fallgrube: Semikolon am Ende des for-Schleifenkopfs | 136 |
| 15.4 Geschachtelte Schleifen | 137 |
| 15.5 Eine endlose for-Schleife | 143 |
| 15.6 for bei Durchläufen mit festen Schrittweiten | 143 |
| 15.7 Variablendeklaration im for-Schleifenkopf (neu in C99) | 146 |
| 15.8 Programmiertechniken | 146 |
| 15.8.1 Anhalten einer Bildschirmausgabe | 146 |
| 15.8.2 Zeilenvorschübe bei geschachtelten Schleifen | 149 |
| 15.8.3 Kombinieren mit for-Schleifen | 150 |
| 15.8.4 Zwischeninformationen bei rechenintensiven Programmen | 152 |
| 15.8.5 Merker in for-Schleifen bei Eintreten von Ereignissen | 153 |
| 15.9 Fallgruben | 154 |
| 15.9.1 Niemals die Laufvariable im Schleifenkörper ändern | 154 |
| 15.9.2 Gleitpunktzahlen niemals auf Gleichheit prüfen | 157 |
| 15.9.3 Laufvariable einer for-Schleife läuft über Endwert hinaus | 159 |
| | |
| 16 Die while-Anweisung | 161 |
| 16.1 Die while-Anweisung | 161 |
| 16.2 Programmiertechniken | 163 |
| 16.2.1 while bei unbekannter Zahl von Schleifendurchläufen | 163 |
| 16.2.2 Konsistenzprüfungen bei Eingaben | 165 |
| 16.2.3 Die Konstante EOF | 166 |
| 16.2.4 Minimum und Maximum in einer Zahlenfolge | 167 |
| 16.3 Zufallszahlen in C | 168 |
| | |
| 17 Die do... while-Anweisung | 175 |
| 17.1 Die do... while-Anweisung | 175 |
| 17.2 Programmiertechniken | 177 |
| 17.2.1 do... while-Schleifen nicht so oft wie while-Schleifen | 177 |
| 17.2.2 Abschließendes } while immer in einer Zeile | 178 |
| | |
| 18 Die break-Anweisung | 179 |
| 18.1 Die break-Anweisung | 179 |
| 18.2 break bewirkt Verlassen einer Schleifenebene | 180 |
| 18.3 Programmiertechniken | 180 |

| | | |
|-----------|---|------------|
| 18.3.1 | Sofortiges Verlassen von Schleifen und switch | 180 |
| 18.3.2 | Endlosschleifen und break | 182 |
| 19 | Die continue-Anweisung | 183 |
| 19.1 | Die continue-Anweisung | 183 |
| 19.2 | Programmiertechniken | 184 |
| 19.2.1 | continue nur im äußersten Notfall | 184 |
| 19.2.2 | Korrekte Programme müssen auch schnell sein | 185 |
| 19.3 | Datums- und Zeitangaben (<time.h>) | 189 |
| 19.3.1 | Konstanten und Datentypen | 189 |
| 19.3.2 | Funktionen | 189 |
| 19.3.3 | Beispiele zu Funktionen aus <time.h> | 192 |
| 20 | Marken und die goto-Anweisung | 197 |
| 20.1 | Marken und die goto-Anweisung | 197 |
| 20.2 | Programmiertechniken | 197 |
| 20.2.1 | goto nur im äußersten Notfall | 197 |
| 20.2.2 | Lesbarere und schnellere Programme mit goto | 198 |
| 21 | Grafikprogrammierung unter Linux | 199 |
| 21.1 | Benutzung von LCGI | 199 |
| 21.2 | Grafikmodus ein- und ausschalten | 200 |
| 21.3 | Eingaben im Grafikmodus | 200 |
| 21.4 | Bildschirm-, Farben- und Pixel-Operationen | 204 |
| 21.5 | Positionieren, Linien zeichnen und Farbe einstellen | 207 |
| 21.6 | Figuren zeichnen und ausfüllen | 209 |
| 21.7 | Einstellungen für Textausgaben | 215 |
| 21.8 | Bilder laden, Bildteile speichern und einblenden | 217 |
| 21.9 | Kuchenstücke malen | 220 |
| 21.10 | Grafikpaket neu bzw. anders einrichten | 222 |
| 21.11 | Arbeiten mit mehreren Zeichenfenstern | 223 |
| 21.12 | Programmierung der Maus | 223 |
| 21.13 | Transformation mathematischer Koordinaten | 226 |
| 22 | Funktionen | 231 |
| 22.1 | Allgemeines zu Funktionen | 231 |
| 22.1.1 | Allgemeines Beispiel zu Funktionen | 231 |
| 22.1.2 | Die Begriffe Parameter und Argumente | 233 |

| | | |
|-----------|---|------------|
| 22.1.3 | Bibliotheken und Headerdateien | 233 |
| 22.2 | Erstellen eigener Funktionen | 234 |
| 22.2.1 | Definition von Funktionen in C89/C99 | 234 |
| 22.2.2 | Definition von Funktionen in Alt-C | 237 |
| 22.2.3 | Die return-Anweisung | 238 |
| 22.2.4 | Funktionen ohne Rückgabewert | 238 |
| 22.2.5 | Forward-Deklarationen | 239 |
| 22.2.6 | Funktions-Prototypen | 242 |
| 22.2.7 | Implizite Datentypumwandlung beim Funktionsaufruf | 246 |
| 22.2.8 | Typische Anwendungsgebiete von Funktionen | 248 |
| 22.3 | Die Parameter von Funktionen | 253 |
| 22.3.1 | Leere Parameterliste durch Angabe von void | 253 |
| 22.3.2 | Bei Funktionsaufrufen findet nur Wertübergabe statt | 253 |
| 22.3.3 | Call by reference | 258 |
| 22.3.4 | Auswertung der Argumente findet vor Funktionsaufruf statt | 260 |
| 22.3.5 | Fallgruben | 261 |
| 22.4 | Ellipsen-Prototypen für Funktionen mit variabler Argumentzahl | 263 |
| 22.4.1 | Reihenfolge der Argument-Ablage im Stack | 263 |
| 22.4.2 | Ellipsen-Prototypen | 264 |
| 22.4.3 | Abarbeiten variabel langer Argumentlisten | 264 |
| 22.4.4 | Verfahren zum Abarbeiten variabel langer Argumentlisten | 265 |
| 22.4.5 | Fallgruben | 269 |
| 22.5 | Neuheiten in C99 | 270 |
| 22.5.1 | Inline-Funktionen | 270 |
| 22.5.2 | Der vordefinierte Name <code>__func__</code> | 271 |
| 22.5.3 | Keine Unterstützung von implizitem int | 272 |
| 22.5.4 | Keine impliziten Funktionsdeklarationen | 272 |
| 22.5.5 | Einschränkungen bei return | 272 |
| 22.6 | Rekursive Funktionen | 273 |
| 22.6.1 | Allgemeines zu rekursiven Funktionen | 273 |
| 22.6.2 | Einige typische Anwendungen für die Rekursion | 276 |
| 22.7 | Zeiger auf Funktionen | 280 |
| 22.7.1 | Zeiger auf Funktionen | 280 |
| 22.7.2 | Typische Anwendungen | 283 |
| 23 | Speicherklassen und Modultechnik | 287 |
| 23.1 | Gültigkeitsbereich, Lebensdauer, Speicherort | 287 |

| | | |
|-----------|---|------------|
| 23.1.1 | Gültigkeitsbereich | 287 |
| 23.1.2 | Lebensdauer | 291 |
| 23.1.3 | Speicherort | 292 |
| 23.1.4 | Gültigkeit, Lebensdauer und Speicherort im Überblick | 292 |
| 23.1.5 | Übung: Ausgabe des Programms <code>block3.c</code> | 293 |
| 23.2 | Schlüsselwörter <code>extern</code> , <code>auto</code> , <code>static</code> und <code>register</code> | 293 |
| 23.2.1 | Das Schlüsselwort <code>extern</code> | 293 |
| 23.2.2 | Das Schlüsselwort <code>auto</code> | 296 |
| 23.2.3 | Fallgrube: Niemals Adressen von <code>auto</code> -Variablen zurückgeben | 303 |
| 23.2.4 | Das Schlüsselwort <code>static</code> | 304 |
| 23.2.5 | Das Schlüsselwort <code>register</code> | 310 |
| 23.3 | Die Schlüsselwörter <code>const</code> und <code>volatile</code> | 310 |
| 23.3.1 | Das Schlüsselwort <code>const</code> | 310 |
| 23.3.2 | Das Schlüsselwort <code>volatile</code> | 312 |
| 23.3.3 | Kombination von <code>const</code> und <code>volatile</code> | 313 |
| 23.4 | Modultechnik und Information Hiding | 314 |
| 23.4.1 | Linker und Compiler | 317 |
| 23.4.2 | Beispiel: Simulation von Turingmaschinen | 319 |
| 24 | Präprozessor-Direktiven | 331 |
| 24.1 | Bedingte Kompilierung | 332 |
| 24.1.1 | Präprozessor-Direktiven zur bedingten Kompilierung | 332 |
| 24.1.2 | Typische Anwendungen | 334 |
| 24.1.3 | Testen mit Makro <code>assert()</code> aus Headerdatei <code><assert.h></code> | 339 |
| 24.2 | Einkopieren von anderen Headerdateien | 340 |
| 24.2.1 | Die Präprozessor-Direktive <code>#include</code> | 340 |
| 24.2.2 | Typische Anwendungen | 341 |
| 24.3 | Definition von Makros (<code>#define</code> und <code>#undef</code>) | 342 |
| 24.3.1 | Definition von Konstanten mit <code>#define</code> | 342 |
| 24.3.2 | Definition von Funktionsmakros mit <code>#define</code> | 343 |
| 24.3.3 | Operator <code>#</code> : Ersetzung von Makroparametern durch String | 345 |
| 24.3.4 | Operator <code>##</code> : Zusammensetzen neuer Namen | 346 |
| 24.3.5 | Rekursive Makrodefinitionen | 346 |
| 24.3.6 | Makros mit variabler Anzahl von Argumenten (neu in C99) | 348 |
| 24.3.7 | Makrodefinitionen mit <code>#undef</code> wieder aufheben | 349 |
| 24.3.8 | Unterschiede zwischen Funktionen und Makros | 349 |
| 24.4 | Vordefinierte Makronamen | 353 |

| | | |
|-----------|--|------------|
| 24.5 | Die restlichen Präprozessor-Direktiven | 354 |
| 24.5.1 | #line – Festlegen einer neuen Zeilennummerierung | 354 |
| 24.5.2 | #error – Ausgeben von Fehlermeldungen | 355 |
| 24.5.3 | #pragma – Festlegen von compilerspezifischem Verhalten | 355 |
| 24.5.4 | # – Die Null-Direktive | 355 |
| 25 | Zeiger und Arrays | 357 |
| 25.1 | Eindimensionale Arrays | 357 |
| 25.1.1 | Eindimensionale Arrays | 357 |
| 25.1.2 | Nur statische Arrays erlaubt (in C89) | 361 |
| 25.1.3 | Von Arrays belegter Speicherplatz | 362 |
| 25.1.4 | Fallgruben | 363 |
| 25.2 | Mehrdimensionale Arrays | 365 |
| 25.2.1 | Zweidimensionale Arrays | 365 |
| 25.2.2 | Drei-, vier-, fünf- und sonstige mehrdimensionale Arrays | 373 |
| 25.2.3 | sizeof liefert die Größe eines Arrays | 373 |
| 25.3 | Zusammenhänge zwischen Arrays und Zeigern | 374 |
| 25.3.1 | Arrayname ist konstanter Zeiger auf erstes Element | 374 |
| 25.3.2 | Zugriff auf Arrayelemente ist auch über Zeiger möglich | 377 |
| 25.3.3 | Unterschied zwischen Arraynamen und echtem Zeiger | 380 |
| 25.3.4 | Erlaubte Operationen mit Zeigern | 381 |
| 25.3.5 | Unerlaubte Operationen mit Zeigern | 381 |
| 25.3.6 | Übergabe eines Arrays an eine Funktion mittels Adresse | 384 |
| 25.3.7 | call by value für Arrays (Zeiger) | 388 |
| 25.3.8 | Nachlese zu Arrays und Zeiger | 389 |
| 25.3.9 | Algorithmus: Der Bubble-Sort | 389 |
| 25.3.10 | Verwendung der Bibliotheksfunktion qsort() | 390 |
| 25.4 | Strings und char-Zeiger | 392 |
| 25.4.1 | Besonderheiten von C-Strings | 392 |
| 25.4.2 | Das Schlüsselwort restrict für Zeiger (neu in C99) | 394 |
| 25.4.3 | Eigene Realisierung der Funktion strcpy() mit Arrays | 394 |
| 25.4.4 | Eigene Realisierung der Funktion strcpy() mit Zeigern | 395 |
| 25.4.5 | Die Headerdatei <string.h> | 398 |
| 25.4.6 | Umwandeln von Strings in numerische Werte | 413 |
| 25.4.7 | Umwandeln von numerischen Werten in Strings | 419 |
| 25.4.8 | Besonderheiten beim Einlesen von Strings mit scanf() | 421 |
| 25.4.9 | Ein- und Ausgabe von Strings mit gets() und puts() | 422 |

| | | |
|-----------|--|------------|
| 25.4.10 | Unterschied zwischen Zeiger- und Array-Deklaration | 423 |
| 25.4.11 | Direkter Zugriff auf Zeichen in einer String-Konstante | 425 |
| 25.5 | Array-Initialisierungen | 426 |
| 25.5.1 | Initialisierung von Arrays | 426 |
| 25.5.2 | Dimensionierungsangaben bei der Initialisierung | 428 |
| 25.5.3 | Zeiger auf unbenamte Arrays (neu in C99) | 429 |
| 25.5.4 | Implizite Initialisierung bei static-Variablen/Arrays | 430 |
| 25.5.5 | Initialisierung lokaler Variablen auch mit Nicht-Konstanten | 431 |
| 25.5.6 | Initialisierung von lokalen Arrays in C89/C99 | 432 |
| 25.5.7 | Initialisierung von lokalen Arrays mit variablen Werten (neu in C99) | 432 |
| 25.5.8 | Initialisierung von lokalen Arrays mit 0 oder NULL | 433 |
| 25.5.9 | Initialisierte Arrays mit const vor Überschreiben schützen | 434 |
| 25.6 | Lokale Arrays variabler Länge (neu in C99) | 435 |
| 25.7 | Zeigerarrays und Zeiger auf Zeiger | 436 |
| 25.7.1 | Einfache Zeigerarrays | 436 |
| 25.7.2 | Zeiger auf Arrays | 437 |
| 25.7.3 | Vertauschen von zwei Arrays über Zeiger | 438 |
| 25.7.4 | Übergabe von Arrays an Funktionen | 440 |
| 25.7.5 | Zeiger-Zeiger | 441 |
| 25.7.6 | Unterschiede bei zweidimensionalen Arrays und Zeigerarrays | 442 |
| 25.7.7 | Zugriff auf beliebige Elemente in einem Zeigerarray | 445 |
| 25.7.8 | Zeigerarrays mit Funktionsadressen | 451 |
| 26 | Argumente auf der Kommandozeile | 453 |
| 26.1 | Die Parameter argc und argv der Funktion main() | 453 |
| 26.2 | Optionen auf der Kommandozeile | 456 |
| 27 | Dynamische Speicher-Reservierung und -Freigabe | 463 |
| 27.1 | Nachteile von statischen Arrays | 463 |
| 27.1.1 | Gefahr der Speicherüberschreibung | 464 |
| 27.1.2 | Speicherplatzvergeudung | 465 |
| 27.2 | Speicher reservieren mit malloc() | 466 |
| 27.2.1 | Die Funktion malloc() | 466 |
| 27.2.2 | Dynamische Arrays für beliebige Datentypen | 470 |
| 27.2.3 | Konvertierung von void-Zeigern | 473 |
| 27.3 | Speicher reservieren und initialisieren mit calloc() | 474 |
| 27.4 | Größenänderung eines allozierten Speichers mit realloc() | 475 |

| | | |
|-----------|---|------------|
| 27.4.1 | Die Funktion realloc() | 475 |
| 27.4.2 | Besonderheiten der Funktion realloc() | 478 |
| 27.4.3 | Schnellere Programme mit größeren Speicherblöcken | 479 |
| 27.5 | Freigeben von dynamisch reservierten Speicherbereich | 481 |
| 27.5.1 | Die Funktion free() | 481 |
| 27.5.2 | Fallgrube: free() setzt übergebenen Zeiger nicht auf NULL | 481 |
| 27.5.3 | Tipp: Eigenes Makro zur Freigabe von dynamischen Speicher | 483 |
| 27.6 | Fallgruben | 484 |
| 27.6.1 | free() nur auf von malloc(), calloc() und realloc() gelieferte Zeiger | 484 |
| 27.6.2 | Allozieren von Speicherplatz in einer Funktion | 485 |
| 27.7 | Programmiertechnik: Dynamische Zeiger-Arrays | 488 |
| 27.8 | Fallgrube: free() bei Zeiger-Arrays | 489 |
| 28 | Strukturen | 491 |
| 28.1 | Deklaration und Definition von Strukturen | 491 |
| 28.1.1 | Deklaration von Strukturen | 491 |
| 28.1.2 | Wichtige Regeln und Hinweise für Strukturdeklarationen | 492 |
| 28.1.3 | Definition von Strukturvariablen | 493 |
| 28.1.4 | Zusammenfassung von Strukturdeklaration und -definition | 494 |
| 28.1.5 | Namenlose Strukturen | 495 |
| 28.2 | Operationen mit Strukturvariablen | 496 |
| 28.2.1 | Zugriff auf Strukturkomponenten mittels Punktoperator | 496 |
| 28.2.2 | Zuweisung zwischen Strukturkomponenten | 497 |
| 28.2.3 | Zuweisung ganzer Strukturvariablen | 502 |
| 28.2.4 | Vergleich von Strukturvariablen ist nicht möglich | 502 |
| 28.2.5 | Casting für komplette Strukturvariable ist nicht möglich | 503 |
| 28.2.6 | Adreß- und sizeof-Operator für Strukturvariablen erlaubt | 503 |
| 28.3 | Initialisierung von Strukturvariablen | 504 |
| 28.3.1 | Initialisierung von Strukturvariablen in C89 und C99 | 504 |
| 28.3.2 | Initialisierung von Strukturvariablen (nur in C99) | 505 |
| 28.4 | Strukturarrays | 507 |
| 28.5 | Strukturen als Funktionsparameter | 514 |
| 28.6 | Zeiger und Strukturen | 516 |
| 28.6.1 | Allgemeines zu Zeiger und Strukturen | 516 |
| 28.6.2 | Dynamische Strukturarrays | 522 |
| 28.6.3 | Rekursive Strukturen | 529 |
| 28.7 | Strukturen mit variabel langen Arrays (neu in C99) | 558 |

| | | |
|-----------|--|------------|
| 28.8 | Spezielle Strukturen (Unions und Bitfelder) | 559 |
| 28.8.1 | Unions | 559 |
| 28.8.2 | Bitfelder | 564 |
| 29 | Eigene Datentypen | 569 |
| 29.1 | Definition eigener Datentypnamen mit typedef | 569 |
| 29.1.1 | Vergabe neuer Namen an existierende Datentypen mit typedef | 569 |
| 29.1.2 | Höhere Portabilität und bessere Lesbarkeit durch typedef | 572 |
| 29.2 | Definition eigener Datentypen mit enum | 573 |
| 29.2.1 | Definition eigener Datentypen mit enum | 573 |
| 29.2.2 | Regeln für enum | 575 |
| 30 | Dateien | 577 |
| 30.1 | Höhere E/A-Funktionen | 577 |
| 30.1.1 | Vordefinierte Struktur FILE | 578 |
| 30.1.2 | Öffnen und Schließen von Dateien | 578 |
| 30.1.3 | Lesen und Schreiben in Dateien | 580 |
| 30.1.4 | Unterschied zwischen Text- und Binärmodus | 596 |
| 30.1.5 | Positionieren in Dateien | 598 |
| 30.1.6 | Öffnen einer Datei mit existierendem Stream | 601 |
| 30.1.7 | Löschen und Umbenennen von Dateien | 603 |
| 30.1.8 | Pufferung | 603 |
| 30.1.9 | Temporäre Dateien | 605 |
| 30.1.10 | Ausgabe von System-Fehlermeldungen | 608 |
| 31 | Anhang | 613 |
| 31.1 | Prioritätstabelle für die Operatoren | 613 |
| 31.2 | C-Schlüsselwörter | 613 |
| 31.3 | Wertebereiche für die einzelnen Datentypen | 614 |
| 31.4 | Die Funktion printf() | 615 |
| 31.5 | Die Funktion scanf() | 617 |
| 31.6 | ASCII-Tabelle | 619 |

Vorwort

Ein paar Worte zu C und seiner Geschichte

Zu Beginn wird hier kurz auf die Historie von C und die verschiedenen Standardisierungen von C eingegangen.

Die Entstehung von C

Die Sprache C wurde im Jahre 1972 von *Dennis M. Ritchie* in den Bell-Laboratorien bei AT&T entwickelt und von *Brian W. Kernighan* in den Jahren 1973/74 weiter verbessert. Vorläufer von C waren die Sprachen *BCPL* (Basic Combined Programming Language) und *B*.

C war schon immer sehr eng mit dem Betriebssystem Unix verbunden, da die Sprache C auf diesem damals noch jungen Betriebssystem entwickelt wurde und Unix wiederum selbst mit allen seinen Dienstprogrammen nahezu vollständig in C geschrieben wurde. In den 80er Jahren hat sich aber C als eine universell einsetzbare Sprache entpuppt, was dazu führte, dass C heute auf nahezu jedem beliebigen Betriebssystem (Linux, Unix-Systeme, MS-DOS, Windows95/98/NT/..., VMS, OS/2, ...) angeboten wird.

In den 1970er und 1980er Jahren gab es keine Standardbeschreibung zur Sprache C. Statt dessen galt die 1. Ausgabe des Buches „*The C Programming Language*“ von Kernighan und Ritchie (Prentice Hall, 1978) als die Bibel für alle C-Fragen. Diese „Bibel“ ließ jedoch einige Fragen offen. So wurde bereits in den frühen 80er Jahren die Notwendigkeit für einen wirklichen C-Standard erkannt:

❑ C89 – der erste Standard für C

Im Jahre 1983 begann das ANSI-Komitee X3J11¹ mit dem Unterfangen, die Sprache C zu standardisieren. Im selben Jahr noch entschied das Komitee X3J11, dass nur *ein* C-Standard geschaffen werden soll, welcher von beiden Organisationen ANSI und ISO verabschiedet wird. C wurde zum ersten Mal Ende des Jahres 1989 mit der Annahme des ANSI-Standards standardisiert. Diese Version von C wird allgemein als *C89* bezeichnet. Dieser Standard wurde auch im Jahre 1990 von ISO übernommen. C89 wurde im Jahre 1995 leicht verbessert.

❑ C99 – der neue Standard für C

Im Jahre 1999 wurde ein neuer Standard für C geschaffen. Diese Version nennt sich *ISO C99*, oder kurz auch nur *C99*. C99 enthält eine Reihe von Verbesserungen und einige neue Konstrukte, welche teilweise von C++ übernommen wurden.

Alle C-Programme, die die von C99 vorgegebenen Vorschriften einhalten, werden *C99 kompatibel* genannt. Solche Programme haben den Vorteil, dass sie leicht portierbar sind, also ohne größere Änderungen von einem anderen C99-Compiler auf einer anderen Maschine in eine ablauffähige Version übersetzt werden können.

C als Vorstufe zu C++

C++ wurde – beginnend im Jahr 1979 – von *Bjarne Stroustrup* entwickelt. C++, das eine objektorientierte Version zu C ist, hat sich inzwischen sehr verbreitet. Da aber C++ nur eine Erweiterung zu C ist, muss auch jeder, der mit C++ programmieren will, die Sprache C richtig beherrschen, bevor er in die höheren Sphären der objektorientierten Programmierung aufsteigen kann.

¹ANSI (American National Standard Institute) ist eine amerikanische Organisation, welche Mitglied der International Standards Organisation (ISO) ist.

Hinweise zu diesem Buch

Dieses Buch beschreibt die Theorie der Programmiersprache C anhand vieler Beispiele. Dabei begnügt es sich jedoch nicht allein mit der Vorstellung der einzelnen C-Elemente, sondern vermittelt dem C-Neuling auch Einblicke in wichtige Grundlagen der Informatik. Zudem gibt es auch zu den einzelnen C-Konstruktionen Programmieretechniken aus der C-Praxis, typische Anwendungsgebiete, Tipps und Fallgruben, die in C leider nicht allzu selten sind. Diese Grundkenntnisse bilden das Fundament, das für eine erfolgreiche Programmierung in C unverzichtbar ist.

Nach dem Durcharbeiten dieses Buches verfügt der Leser über ein gesundes und breites C-Fundament, und ist in der Lage, erfolgreich in C zu programmieren. Die Intention dieses Buches ist es nämlich:

- ❑ den C-Anfänger systematisch vom C-Basiswissen bis hin zu den fortgeschrittenen Techniken der Programmierung zu führen.
- ❑ dem bereits erfahrenen C-Programmierer – aufgrund der Vielzahl von Tipps, fundamentalen Algorithmen und nützlichen Programmieretechniken – eine Vertiefung bzw. Ergänzung seines C-Wissens zu ermöglichen.

Layout der Programme in diesem Buch

Es sei vorweg darauf hingewiesen, dass sich die in diesem Buch vorgestellten Programme nicht immer an die allgemein gültigen Programmier-Richtlinien halten. So befinden sich z. B. manchmal mehrere Anweisungen in einer Zeile. Dies wurde sowohl aus Platzgründen als auch aus Layout-Gesichtspunkten, die in einem solchen Buch einzuhalten sind, leider notwendig, denn ein Programmlisting, bei dem sich eventuell die entscheidenden Anweisungen über zwei Seiten verteilen, ist sicherlich aus didaktischen Gründen hinsichtlich Lesbarkeit ungünstiger als ein Programm, das sich vollständig auf einer Seite befindet, ohne dass der Leser ständig vor- und zurückblättern muss.

Downloads zu diesem Buch

Alle Programme in diesem Buch können von folgender Webseite

<http://www.millin.de>

heruntergeladen werden.

Windows-Kompatibilität

Auch wenn dieses Buch ursprünglich für die C-Programmierung unter Linux konzipiert wurde, so sind doch nahezu alle in diesem Buch vorgestellten Programme auch unter Windows-Compilern lauffähig. Einschränkungen gibt es lediglich bei einigen Windows-Compilern, die noch nicht den neuesten C99-Standard implementiert haben, aber das betrifft nur einige wenige Programme in diesem Buch.

Auch zu der ursprünglich für Linux eigens im Rahmen dieses Buches entwickelten Graphikbibliothek LCGI existiert zwischenzeitlich eine Windows-Version, die von der oben erwähnten Webseite heruntergeladen werden kann.

Kapitel 1

Einführendes Beispiel

1.1 Erste wesentliche C-Regeln

Das erste C-Programm `erst.c`:

```
/*
 *      Einfuehrendes Beispiel
 */

#include <stdio.h>

int main(void)
{
    printf("*****\n");
    printf(" *                               *\n");
    printf(" *      Das ist mein erstes C-Programm *\n");
    printf(" *                               *\n");
    printf("*****\n");

    return 0; /* auch moeglich: return(0); */
}
```

Hier sind bereits einige grundlegende C-Regeln zu erkennen:

1. Vorläufig müssen alle C-Programme die Zeichenfolge `int main(void)` enthalten, was den eigentlichen Programmanfang kennzeichnet.
2. Der zu `main()` gehörige Programmteil ist in geschweiften Klammern einzubetten. Dabei kennzeichnet:
 - { den Anfang und
 - } das Ende des entsprechenden Programmteils.
3. Ein Kommentar wird mit `/* */` geklammert. Kommentare sind Informationen für den Programmierer und werden vom C-Übersetzer (auch C-Compiler genannt) nicht beachtet. Um ein Programm möglichst lesbar zu machen, sollte es möglichst gut kommentiert sein.
4. Bildschirmausgaben sind mit `printf("Text")` möglich. Auszugebender Text muss dabei mit Anführungszeichen geklammert sein.

- Um die einzelnen C-Anweisungen voneinander zu trennen, muss immer ein Semikolon als Trennzeichen angegeben werden. Vor und nach einem Semikolon können beliebig viele Leerzeichen oder auch Leerzeilen angegeben werden; gilt übrigens auch für andere Wörter wie `main`, `printf` oder `return`.
- `#include <stdio.h>` sollte zunächst immer am Anfang (vor `main()`) angegeben werden. Ebenso sollte vor der abschließenden geschweiften Klammer immer `return 0;` oder `return(0);` angegeben werden.
- Die Angabe von `\n` im Text bei `printf("Text")` bewirkt, dass beim Programmablauf an dieser Stelle auf den Anfang der nächsten Zeile gesprungen wird. Das Steuerzeichen `\n`, das immer in Anführungszeichen anzugeben ist, wird also niemals als Text ausgegeben, sondern bewirkt einen Zeilenvorschub.

Wird als letztes Zeichen in einem auszugebenden Text nicht `\n` angegeben, so wird beim nächsten `printf` in der gleichen Zeile (an der alten Bildschirmposition) mit der Ausgabe fortgefahren, wie es das C-Programm `zweit.c` verdeutlicht:

```
#include <stdio.h>

int main(void)
{
    printf("****\n*  *\n*  *\n****\n\n");
    printf("Das ist ein Viereck ");
    printf("aus Sternchen, ");
    printf("oder nicht ?");
    printf("\nE\nN\nD\nE");
    printf("N");
    printf("D");
    printf("E\n");
    return 0;
}
```

Startet man dieses Programm `zweit`, liefert es die folgende Bildschirmausgabe:

```
****
*  *
*  *
****

Das ist ein Viereck aus Sternchen, oder nicht ?
E
N
D
ENDE
```

Eine Angabe wie:

```
printf("Guten Ta
g");
```

ergibt einen Fehler, da ein Zeilenwechsel innerhalb der in `"..."` eingebetteten Zeichenkette nicht erlaubt ist.

1.2 Zeilen-Kommentare mit // (neu in C99)

Neben dem schon immer vorhandenen C-Kommentar, der in einem C-Programm mit der Zeichenfolge `/*` eingeleitet und mit der Zeichenfolge `*/` beendet wird, hat C99 zusätzlich noch wie C++ auch die Zeichenfolge `//` zum Kommentieren eingeführt: Alle Zeichen ab `//` bis zum Zeilenende werden dabei als Kommentar interpretiert:

```
printf("Frau Holle, "); /* Normaler C-Kommentar, der sich über
                        mehrere Zeilen erstrecken kann und
                        auch weiterhin verwendet werden kann.
                        */
printf("wie gehts?\n"); // Ab hier bis zum Zeilenende Kommentar (neu in C99)
                        // Bei mehrzeiligen Kommentare ist erneut // anzugeben
```

1.3 Gute Lesbarkeit von Programmen

Einrücken von untergeordneten Programmteilen

Wie einzurücken ist, wurde bereits in den vorherigen Programmbeispielen für die Anweisungen innerhalb der geschweiften Klammern von `main()` gezeigt. Es gibt keine allgemeingültige Regel, wie viele Leerzeichen einzurücken sind; üblich ist jedoch das Einrücken um drei, vier oder auch acht Leerzeichen.

Sinnvolle Kommentare

Da Kommentare in einem C-Programm der Mensch/Mensch-Kommunikation dienen, sind sie ein wichtiges Hilfsmittel, um die Lesbarkeit eines Programms zu erhöhen und somit das Verständnis zu erleichtern.

- ❑ **Kommentare sollten zusätzliche Information enthalten**
Die Befehlsbedeutung ist nicht zu kommentieren. Kommentare wie `i=i+1; /* i wird um 1 erhoeht */` sind nicht nur überflüssig, sondern sogar schädlich, da sie den Programmtext unnötig aufblähen, was eher zur Unübersichtlichkeit und damit zu einer schlechteren Lesbarkeit führt.
- ❑ **Nicht zuviel Programmtext auf einmal kommentieren**
Drei Seiten zusammenhängender Kommentar, um die nächsten drei Seiten Programmtext zu beschreiben, dient nicht der besseren Verständlichkeit eines Programms, sondern lässt sich auch nicht leicht ändern, wenn sich der betreffende Programmteil grundlegend ändert.
- ❑ **Ein falscher Kommentar ist schlimmer als gar kein Kommentar**
Kommentare müssen bei jeder Änderung des Programms aktualisiert werden.
- ❑ **Einige gute treffende Sätze sind mehr wert als Romane**
Bei Kommentaren geht es meist darum, den Nagel auf den Kopf zu treffen. Die Professionalität und Güte eines Programmierers lässt sich unter anderem an seinem Stil der Kommentierung erkennen.

1.4 Vermeiden von geschachtelten Kommentaren

Obwohl einige Compiler geschachtelte Kommentare, wie z. B.:

```
.....  
/*  
    .....  
        /*  
            .....  
            .....  
            */  
        .....  
    */  
.....
```

zulassen, ist es nicht empfehlenswert, von solchen Schachtelungen Gebrauch zu machen, da diese nicht durch die C-Standards „abgesegnet“ sind. Verwendet man trotzdem solche Kommentar-Schachtelungen, dann verstößt man gegen die in der SW-Entwicklung wichtige Forderung der Portabilität, da ein solches Programm eventuell auf anderen C89- bzw. C99-Compilern nicht mehr ablauffähig ist.

Es gilt nämlich, dass ab dem ersten Vorkommen von `/*` bis zum ersten Auftreten von `*/` alles als Kommentar interpretiert wird. Für obige Angabe würde dies bedeuten, dass der fett gedruckte Teil vom Compiler als Kommentar interpretiert wird, so dass er dann mit dem Rest nichts mehr anfangen kann.

Kapitel 2

Elementare Datentypen

2.1 Die Grunddatentypen in C

In C existieren die in Abbildung 2.1 gezeigten elementaren Datentypen. Da ein Computer Zeichen – wie z. B. Buchstaben – ganz anders behandelt werden als Gleitpunktzahlen – wie z. B. die Zahl $\pi=3.1415\dots$ –, wurde eine Klassifikation dieser unterschiedlichen Daten notwendig.

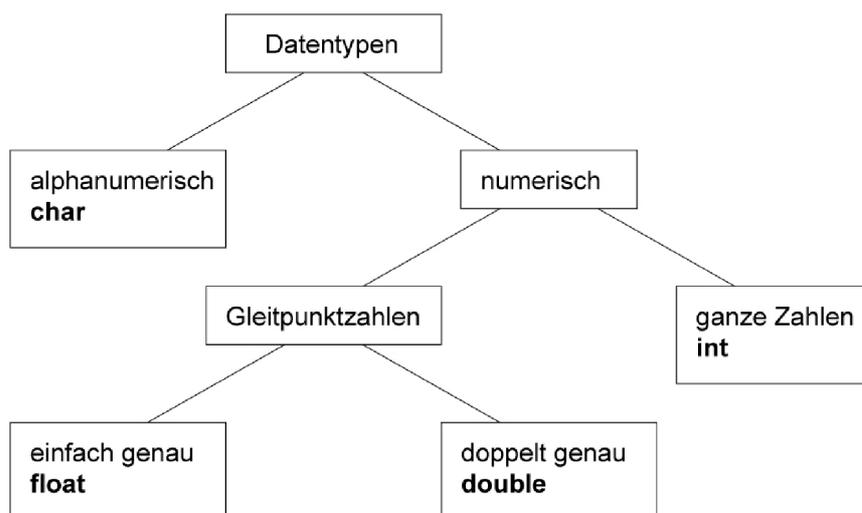


Abbildung 2.1: Die elementaren C-Datentypen

Ordnet man nun in einem Programm Daten bestimmten Klassen wie *Zeichen*, *ganze Zahl*, *einfach/doppelt genaue Gleitpunktzahl* usw. zu, dann teilt man dem Rechner deren *Datentyp* mit. In C existieren die Grunddatentypen: `char`, `int`, `float` und `double`, wobei es für die Typen `int` und `double` Versionen mit unterschiedlichen Genauigkeiten gibt:

char Daten dieses Typs belegen ein Byte Speicherplatz und repräsentieren „Zeichen“. In einem Byte (8 Bit) kann genau ein Zeichen des ASCII-Zeichenvorrats gespeichert werden. Vor `char` darf dabei eines der beiden folgenden Schlüsselwörter angegeben werden:

| | |
|-----------------|--------------------------------|
| signed | 1. Bit ist Vorzeichenbit. |
| unsigned | 1. Bit ist kein Vorzeichenbit. |

int Dieser Datentyp repräsentiert „ganze Zahlen“; dafür sind im allgemeinen 4 Bytes vorgesehen, also viermal soviel Speicherplatz wie für `char`. Mit diesen 4 Bytes (32 Bit) können Zahlen im Bereich $-2\,147\,483\,648 \dots +2\,147\,483\,647$ dargestellt werden.

Vor `int` darf dabei eines der folgenden Schlüsselwörter angegeben werden:

short Durch Voranstellen dieses Schlüsselworts (`short int`) werden typischerweise 2 Bytes (16 Bits) reserviert, was dem Zahlenbereich $-32\,768 \dots +32\,767$ entspricht.

long Durch Voranstellen dieses Schlüsselworts (`long int`) wird auf 32-Bit Maschinen üblicherweise ein Speicherplatz von 4 Bytes (32 Bit) reserviert. Auf 64-Bit Computern hat `long` aber normalerweise eine Genauigkeit von 64 Bit, also 8 Byte, was einem Wertebereich von etwa $\pm 9.22 \dots \cdot 10^{18}$ entspricht.

long long (nur in C99) Ein `long long int` entspricht sowohl auf 32-, als auch auf 64-Bit Architekturen einer 8-Byte Genauigkeit. Dieser Datentyp bietet also für 32-Bit Computer die Möglichkeit, mit einem größeren Wertebereich zu arbeiten, als ihn die Prozessorregister bieten.

unsigned Das Voranstellen dieses Schlüsselworts (`unsigned int`) bewirkt, dass das erste Bit nicht als Vorzeichenbit, sondern als echte Stelle interpretiert wird. Mit `unsigned` kann also festgelegt werden, dass in einem Speicherplatz nur ganze positive, also keine negativen Zahlen stehen können.

Die obigen Schlüsselwörter können auch alleine, d.h. ohne `int`, angegeben werden, was vom C-Compiler genauso wie mit Angabe von `int` interpretiert wird: `short` entspricht `short int`, `long` entspricht `long int`, `long long` entspricht `long long int` und `unsigned` entspricht `unsigned int`.

float Dieser Datentyp ist für Gleitpunktzahlen mit einfacher Genauigkeit vorgesehen; dazu werden im allgemeinen 4 Bytes (32 Bit) reserviert.

double Daten dieses Typs belegen 8 Bytes (64 Bit) Speicherplatz und sind Gleitpunktzahlen mit doppelter Genauigkeit. Wird `long double` angegeben, so bedeutet dies meist, dass ein Speicherplatz von 96 Bits (12 Bytes) reserviert wird.

Wie viele Bytes ein bestimmter Datentyp auf einer gegebenen Architektur tatsächlich belegt, kann mit dem folgenden Programm ermittelt werden (Beispiel für `long double`):

```
#include <stdio.h>
int main(void)
{
    printf( "Der Datentyp belegt %d Bytes \n", sizeof(long double) );
    return 0;
}
```

2.2 Wertebereiche für die einzelnen Datentypen

Tabelle 2.1 zeigt für die einzelnen Datentypen die auf 32-Bit Maschinen üblicherweise verwendeten Bitzahlen und die daraus resultierenden Wertebereiche. Tabelle 2.2, welche die äquivalente Information für 64-Bit Architekturen enthält, verdeutlicht, dass sich auf 64-Bit Computern lediglich der Wertebereich des Datentyps `long` ändert.

Tabelle 2.1: Typische Wertebereiche für die einzelnen Datentypen auf 32-Bit-Architekturen

| Datentyp-Bezeichnung | Bitanzahl | Wertebereich |
|--|-----------|---|
| <code>char</code> , <code>signed char</code> | 8 | -128 .. 127 |
| <code>unsigned char</code> | 8 | 0 .. 255 |
| <code>short</code> , <code>signed short</code> , <code>short int</code> , <code>signed short int</code> | 16 | -32 768 .. 32 767 |
| <code>unsigned short</code> , <code>unsigned short int</code> | 16 | 0 .. 65 535 |
| <code>int</code> , <code>signed</code> , <code>signed int</code> | 32 | -2 147 483 648 .. 2 147 483 647 |
| <code>unsigned</code> , <code>unsigned int</code> | 32 | 0 .. 4 294 967 295 |
| <code>long</code> , <code>signed long</code> <code>long int</code> , <code>signed long int</code> | 32 | -2 147 483 648 .. 2 147 483 647 |
| <code>unsigned long</code> , <code>unsigned long int</code> | 32 | 0 .. 4 294 967 295 |
| <code>long long</code> , <code>signed long long (C99)</code> <code>long long int</code> , <code>signed long long int (C99)</code> | 64 | -9 223 372 036 854 775 808 9 223 372 036 854 775 807 |
| <code>unsigned long long (C99)</code> <code>unsigned long long int (C99)</code> | 64 | 0 .. 18 446 744 073 709 551 615 |
| <code>float</code> | 32 | $1.2 \cdot 10^{-38}$.. $3.4 \cdot 10^{38}$ |
| <code>double</code> | 64 | $2.2 \cdot 10^{-308}$.. $1.8 \cdot 10^{308}$ |
| <code>long double</code> | 96 | $3.4 \cdot 10^{-4932}$.. $1.1 \cdot 10^{4932}$ |

Tabelle 2.2: Typische Wertebereiche für die einzelnen Datentypen auf 64-Bit-Architekturen

| Datentyp-Bezeichnung | Bitanzahl | Wertebereich |
|--|-----------|---|
| <code>int</code> , <code>signed</code> , <code>signed int</code> | 32 | -2 147 483 648 .. 2 147 483 647 |
| <code>long</code>, <code>signed long</code> <code>long int</code>, <code>signed long int</code> | 64 | -9 223 372 036 854 775 808 9 223 372 036 854 775 807 |
| <code>unsigned long</code>, <code>unsigned long int</code> | 64 | 0 .. 18 446 744 073 709 551 615 |
| <code>unsigned long long (C99)</code> <code>unsigned long long int (C99)</code> | 64 | 0 .. 18 446 744 073 709 551 615 |
| <code>long long</code> , <code>signed long long (C99)</code> <code>long long int</code> , <code>signed long long int (C99)</code> | 64 | -9 223 372 036 854 775 808 9 223 372 036 854 775 807 |

Kapitel 3

Konstanten

Die Eigenschaft von Konstanten ist, dass sie – im Gegensatz zu Variablen – einen festen Wert besitzen, der einem der gerade besprochenen Datentypen zugeordnet ist. Jede Konstante hat dabei einen Datentyp, welcher durch ihre Form und Wert bestimmt wird (engl. *self-typing*).

3.1 char-Konstanten

Der Datentyp `char` wird meist durch 1 Byte realisiert. `char`-Konstanten müssen mit einfachem Hochkomma (z. B.: `'a'`) geklammert werden. Im Byte des `char`-Datentyps wird dann der zugehörige ASCII-Wert gespeichert, der dieses Zeichen repräsentiert, wie z. B.:

| char-Konstante | Dezimaler ASCII-Wert | Dualdarstellung im char -Byte |
|-----------------------|----------------------|--------------------------------------|
| <code>'a'</code> | 97 | 01100001 |
| <code>'W'</code> | 87 | 01010111 |
| <code>'*'</code> | 42 | 00101010 |
| <code>'8'</code> | 56 | 00111000 |

Bei `char` handelt es sich anders als bei `float` und `double` um einen Ganzzahltypen. C ist nun bei weitem nicht so typstrenge wie z. B. PASCAL. So ist es in C ohne weiteres möglich, im `char`-Datentyp auch einen numerischen Wert zu speichern. Es ist dabei lediglich zu beachten, dass bei einem numerischen Wert dann nicht der ASCII-Wert wie bei einer `char`-Konstante, sondern die Dualdarstellung dieser Zahl abgespeichert wird. Während also z. B. für `'8'` der ASCII-Wert 56, der dieses Zeichen repräsentiert als Dualzahl (00111000) abgelegt wird, wird dagegen für die Dezimalzahl 8 die dieser Zahl entsprechende Dualzahl (00001000) im `char`-Datentyp abgelegt.

3.2 Ganzzahlige Konstanten

Es gibt drei verschiedene Arten von ganzzahligen Konstanten:

- ❑ *Dezimal-Konstanten*, die immer mit einer von 0 verschiedenen Ziffer beginnen
- ❑ *Oktal-Konstanten*, die immer mit einer 0 beginnen müssen
- ❑ *Hexadezimal-Konstanten*, die immer mit `0x` oder `0X` beginnen müssen.

Kapitel 4

Variablen

4.1 Variablen und die C-Regeln für Variablennamen

Die Eigenschaft von Variablen ist, dass sich ihre Werte ständig ändern können. Es handelt sich bei Variablen um eine Art Beschriftung eines Speicherbereichs. Während aber die Beschriftung immer gleich bleibt, kann sich der Inhalt eines Speicherbereichs ändern.

Die *Beschriftung eines Speicherplatzes*, d. h. die Vergabe eines Namens an eine Variable, wird vom Programmierer vorgenommen. Natürlich muss festgelegt werden, um welche Art von Variable (siehe auch Datentypen in Kapitel 2 ab Seite 7) es sich handelt. Diese Festlegung wird ebenfalls vom Programmierer vorgenommen; wie das geschieht, wird im nächsten Kapitel gezeigt. Um nun einen Speicherplatz zu „beschriften“, muss diesem ein Name, der *Variablenname*, gegeben werden. Die Vergabe von Variablennamen ist in C an bestimmte Regeln gebunden:

1. Es dürfen nur Buchstaben (keine Umlaute ä, ü, ö, Ä, Ü, Ö oder ß), Ziffern und der Unterstrich (`_`) verwendet werden.
2. Das erste Zeichen eines Variablennamens muss immer ein Buchstabe oder ein Unterstrich sein.
3. Ein Variablenname darf beliebig lang sein.
4. Wie jede höhere Programmiersprache verfügt auch C über einen festen Wortschatz von Befehlen (siehe unten). Solche Befehle dürfen natürlich nicht als Variablennamen verwendet werden, weil sonst der C-Compiler nicht entscheiden könnte, ob ein Befehl oder ein Variablenname gemeint ist.
5. Obwohl nach der ersten Regel erlaubt, sollten Variablennamen niemals mit einem Unterstrich beginnen, da es eine Reihe intern vordefinierter Namen dieser Form gibt.

Wie oben erwähnt, dürfen C-Schlüsselwörter nicht als Variablennamen verwendet werden. In folgender Tabelle sind alle C-Schlüsselwörter angegeben, wobei die in C99 neu hinzugekommenen Schlüsselwörter fett gedruckt sind. Während die Verwendung von Schlüsselwörtern als alleinstehende Variablennamen verboten ist, dürfen sie jedoch in Variablennamen eingebettet sein; z. B. wäre `automobil` ein erlaubter Variablenname.

| | | | | | | | |
|----------------------------|------------------------------|---------------------------|------------------------------|--------------------------------|-----------------------|-----------------------|---------------------|
| <code>auto</code> | <code>break</code> | <code>case</code> | <code>char</code> | <code>const</code> | <code>continue</code> | <code>default</code> | <code>do</code> |
| <code>double</code> | <code>else</code> | <code>enum</code> | <code>extern</code> | <code>float</code> | <code>for</code> | <code>goto</code> | <code>if</code> |
| <code>int</code> | <code>long</code> | <code>register</code> | <code>return</code> | <code>short</code> | <code>signed</code> | <code>sizeof</code> | <code>static</code> |
| <code>struct</code> | <code>switch</code> | <code>typedef</code> | <code>union</code> | <code>unsigned</code> | <code>void</code> | <code>volatile</code> | <code>while</code> |
| <code>inline</code> | <code>restrict</code> | <code>_Bool</code> | <code>_Complex</code> | <code>_Imaginary</code> | | | |

Kapitel 5

Ausdrücke und Operatoren

5.1 Der einfache Zuweisungsoperator

5.1.1 Allgemeines zum einfachen Zuweisungsoperator

Der Zuweisungsoperator hinterlegt einen Wert in einer Variablen. Durch die Anweisung

```
zahl_var = 5;
```

wird z. B. in der Variablen `zahl_var` der Wert 5 abgelegt. Soll nun der Wert -17 in `zahl_var` gespeichert werden, so ist lediglich

```
zahl_var = -17;
```

zu schreiben, wodurch der alte Wert von `zahl_var` überschrieben wird. Der Operator `=` ist *nicht* mathematisch als Gleichheitszeichen zu verstehen, sondern mehr im Sinne von „ergibt sich aus“:

```
zaehler = 19;
```

Der Wert der Variablen `zaehler` ergibt sich aus 19, d. h.: im Speicherplatz mit dem Namen `zaehler` wird der Wert 19 gespeichert.

Beispiel:

In der Variablen `zahl_1` sei der Wert 4 und in der Variablen `zahl_2` der Wert -24 gespeichert. Durch die Zuweisung:

```
zahl_1 = zahl_2; /* Speichere in zahl_1 den Wert von zahl_2) */
```

gilt dann Folgendes:

Vor dieser Zuweisung:

| | |
|--------|-----|
| zahl_1 | 4 |
| zahl_2 | -24 |
| | ... |

Nach dieser Zuweisung:

| | |
|--------|------------------|
| zahl_1 | 4 -24 |
| zahl_2 | -24 |
| | ... |

Der ursprüngliche Inhalt von `zahl_1` wurde mit dem Inhalt der Variablen `zahl_2` überschrieben.

Beispiel:

Das C-Programm `tausch.c` vertauscht die Werte von zwei `int`-Variablen:

```
#include <stdio.h>

int main(void)
{
    int variab_1, variab_2, hilf; /* Deklaration von drei ganzzahligen Variablen mit den Namen
                                   'variab_1', 'variab_2', 'hilfe' */
    variab_1 = 105;                /*
                                   variab_1 I   105   I
                                   I_____I          */
    variab_2 = -14;               /*
                                   variab_1 I   105   I
                                   I_____I          */
                                   variab_2 I   -14   I
                                   I_____I          */

    /****** Vertauschen *****/
    /* Zwischenspeichern der Werte von 'variab_1' in der "Hilfsvariable" 'hilfe' */
    hilf = variab_1;              /*
                                   variab_1 I   105   I
                                   I_____I          */
                                   variab_2 I   -14   I
                                   I_____I          */
                                   hilf    I   105   I
                                   I_____I          */

    /* Speichern des Wertes von 'variab_2' in 'variab_1', (d. h. der ursprüngliche
       Wert von 'variab_1' wird ueberschrieben).
       Der urpruengliche Wert von 'variab_1' wurde zuvor in 'hilfe' gesichert */
    variab_1 = variab_2;          /*
                                   variab_1 I   -14   I
                                   I_____I          */
                                   variab_2 I   -14   I
                                   I_____I          */
                                   hilf    I   105   I
                                   I_____I          */

    /* Speichern des in 'hilfe' gespeicherten Wertes in 'variab_2'. Nach dieser Zuweisung sind
       die urspruenglichen Werte von 'variab_1' und 'variab_2' vertauscht. */
    variab_2 = hilf;              /*
                                   variab_1 I   -14   I
                                   I_____I          */
                                   variab_2 I   105   I
                                   I_____I          */
                                   hilf    I   105   I
                                   I_____I          */

    return 0;
}
```

5.5.5 Bit-Operatoren nur für ganzzahlige Datentypen erlaubt

Die Bit-Operatoren `~`, `&`, `|` und `^` dürfen nicht auf Operanden angewendet werden, die vom Datentyp `float` oder `double` sind. Z. B. würde für das folgende C-Programm `bitgleit.c` ein Fehler gemeldet:

```
#include <stdio.h>

int main(void) {
    int x;
    float zahl = 10*23;
    x = zahl & 200; /* zahl vom Typ float --> keine Bit-Operatoren dafuer erlaubt */
    printf("%d\n", x);
    return 0;
}
```

5.5.6 Fallgruben

Logische Operatoren und Bit-Operatoren unterscheiden sich

Das Ergebnis einer logischen Operation ist `FALSE` (in C: 0) oder `TRUE` (in C: 1). Bei den Bit-Operatoren dagegen werden die einzelnen Bits verglichen, und das Ergebnis ist dann das Bitmuster, und nicht unbedingt 0 oder 1. Die nachfolgenden Beispiele sollen dies verdeutlichen. Hat z. B. die `short`-Variable `z1` den Wert 2 und die `short`-Variable `z2` den Wert 5, so haben die beiden Operatoren `&` und `&&` unterschiedliche Auswirkungen:

z = z1 & z2;

| | Vorzeichen | $2^{14} = 16384$ | $2^{13} = 8192$ | $2^{12} = 4096$ | $2^{11} = 2048$ | $2^{10} = 1024$ | $2^9 = 512$ | $2^8 = 256$ | $2^7 = 128$ | $2^6 = 64$ | $2^5 = 32$ | $2^4 = 16$ | $2^3 = 8$ | $2^2 = 4$ | $2^1 = 2$ | $2^0 = 1$ | |
|---------|------------|------------------|-----------------|-----------------|-----------------|-----------------|-------------|-------------|-------------|------------|------------|------------|-----------|-----------|-----------|-----------|-----|
| z1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | = 2 |
| z2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | = 5 |
| z1 & z2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | = 0 |

z = z1 && z2;

| | Vorzeichen | $2^{14} = 16384$ | $2^{13} = 8192$ | $2^{12} = 4096$ | $2^{11} = 2048$ | $2^{10} = 1024$ | $2^9 = 512$ | $2^8 = 256$ | $2^7 = 128$ | $2^6 = 64$ | $2^5 = 32$ | $2^4 = 16$ | $2^3 = 8$ | $2^2 = 4$ | $2^1 = 2$ | $2^0 = 1$ | |
|----------|------------|------------------|-----------------|-----------------|-----------------|-----------------|-------------|-------------|-------------|------------|------------|------------|-----------|-----------|-----------|-----------|------------|
| z1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | = 2 (TRUE) |
| z2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | = 5 (TRUE) |
| z1 && z2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 1 (TRUE) |

TRUE && TRUE = TRUE (in C durch 1 dargestellt)

Hat z. B. die `short`-Variable `z1` den Wert 2 und die `short`-Variable `z2` den Wert 5, so haben die beiden Operatoren `|` und `||` unterschiedliche Auswirkungen:

| | | Vorzeichen | 2^{14} | 2^{13} | 2^{12} | 2^{11} | 2^{10} | 2^9 | 2^8 | 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | |
|----------------|----------------------|------------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----|
| $z = z1 z2;$ | <code>z1</code> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | = 2 |
| | <code>z2</code> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | = 5 |
| | <code>z1 z2</code> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | = 7 |

| | | Vorzeichen | 2^{14} | 2^{13} | 2^{12} | 2^{11} | 2^{10} | 2^9 | 2^8 | 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | |
|-----------------|-----------------------|------------|----------|----------|----------|----------|----------|-------|-------|-------|-------|-------|-------|-------|-------|-------|------------|------------|
| $z = z1 z2;$ | <code>z1</code> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | = 2 (TRUE) |
| | <code>z2</code> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | = 5 (TRUE) |
| | <code>z1 z2</code> | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | = 1 (TRUE) | |

TRUE || TRUE = TRUE (in C durch 1 dargestellt)

Niedrige Priorität der dyadischen Bit-Operatoren

Die Bitoperatoren lassen sich – wie in Tabelle 5.2 gezeigt – in die Prioritätstabelle der bisher kennengelernten Operatoren einordnen.

In Tabelle 5.2 ist erkennbar, dass die Priorität der Bitoperatoren sehr niedrig ist. Dies führt häufig zu äußerst ärgerlichen und schwer auffindbaren Fehlern in der praktischen Programmierung.

| | | |
|---------------------------------|---|------------|
| () | ↓ | |
| ! ~ (Tilde) + (Vorz.) - (Vorz.) | ↓ | höhere |
| * / % | ↓ | Priorität |
| + - | ↓ | |
| < <= > >= | ↓ | |
| == != | ↓ | |
| & | ↓ | |
| ^ | ↓ | |
| | ↓ | |
| && | ↓ | niedrigere |
| | ↓ | Priorität |
| = | ↓ | |

Tabelle 5.2: Priorität der Bit-Operatoren

Kapitel 6

Symbolische Konstanten

Die Programmiersprache C sieht die Möglichkeit vor, an Konstanten Namen zu vergeben. Wie wir im nachfolgenden sehen werden, bringt die Verwendung von Konstanten viele Vorteile mit sich, wie z. B. bessere Lesbarkeit von C-Programmen, Arbeitersparnis beim Programmieren und leichtere Änderbarkeit von Programmen. In C können auf zwei verschiedene Arten symbolische Namen an Konstanten vergeben werden:

- ❑ Definition mit `#define` oder
- ❑ Definition unter Verwendung des Schlüsselworts `const`.

6.1 Konstanten-Definition mit `#define`

6.1.1 Die Direktive `#define`

Nehmen wir einmal an, dass Sie ein Programm erstellen sollen, in dem die Zahl π immer mit 9 Stellen Genauigkeit nach dem Komma verwendet werden muss. Sie müssten dann bei jedem Vorkommen von π die Zahlenschlange `3.141592654` eintippen. Mit der Verwendung von `#define` ist es nun jedoch möglich, am Anfang des Programms einem Zahlenwert einen Namen zuzuordnen, wie z. B.

```
#define PI 3.141592654
```

Anstelle des Zahlenwerts `3.141592654` können Sie dann bei den entsprechenden Anweisungen im gesamten Programm den symbolischen Namen (`PI`) verwenden. Dies kann eine erhebliche Arbeitersparnis für das restliche Programm bedeuten, da nicht ein ständiges Nachschlagen und konzentriertes Abtippen der 10 Ziffern notwendig ist, sondern anstelle dessen nur der Name `PI` anzugeben ist.

Hinweis:

`#define`-Angaben werden nicht wie die anderen C-Anweisungen mit Semikolon abgeschlossen.

Kapitel 7

Ein- und Ausgabe

Bevor wir die beiden Bibliotheks-Funktionen `printf()` und `scanf()` in diesem Kapitel ausführlich behandeln, werden wir uns zunächst mit den sogenannten Headerdateien beschäftigen.

7.1 Headerdateien und `#include`

7.1.1 Bibliotheken und Headerdateien

Zu jedem C-Compiler wird eine Bibliothek mitgeliefert, in der Funktionen hinterlegt sind. Wie der Aufbau einer Funktion ist, können wir an `main()` sehen:

```
Funktionskopf, evtl. mit Parametern  int main(void)
                                     {
                                     ...
Funktionsrumpf                       ...
                                     ...
                                     }
```

Der Anfang des Funktionsrumpfs wird durch `{` und das Ende durch `}` gekennzeichnet.

Nun haben wir aber in den vorherigen Programmen bereits Funktionsaufrufe wie `printf()` oder `scanf()` benutzt, zu denen der Funktionsrumpf fehlte. Diese Funktionen sind in einer Bibliothek hinterlegt, in der sich der Linker, nachdem kompiliert wurde, die zum entsprechenden Funktionsaufruf gehörige Funktion sucht und zu Ihrem Programm dazubindet. Sie können also z. B. die Funktion `printf()` benutzen und Bildschirmausgaben veranlassen, obwohl Sie den dazugehörigen Programmteil (Funktion) nicht selbst programmiert haben; dies hat der Compiler-Hersteller bereits für Sie getan.

Seit C89 sind die Funktionsnamen und deren Funktionalität, die jedes C-Kompilierungssystem anbieten muss, genau festgelegt. Diese Funktionen sind üblicherweise in einer Standardbibliothek hinterlegt. Diese Standardbibliothek ist im Prinzip eine Datei, die den für den Linker bindbaren Objekt-Code dieser Funktionen enthält.

Damit aber auch bereits dem C-Compiler Informationen zu solchen Standardfunktionen bekannt sind, existieren so genannte Headerdateien, in denen die Funktionsköpfe der entspre-

Kapitel 8

Datentypumwandlungen

8.1 Implizite Datentypumwandlungen

8.1.1 Der sizeof-Operator

Bevor wir uns genauer mit den impliziten Datentypumwandlungen beschäftigen, werden wir einen neuen Operator kennenlernen: den `sizeof`-Operator. Mit diesem Operator kann die Größe eines Datentyps, einer Variable oder eines Ausdrucks in Bytes ermittelt werden. So sind z. B. die folgenden Angaben möglich:

- `sizeof(double)` liefert Anzahl von Bytes, die vom Datentyp `double` belegt werden.
- `sizeof(var)` liefert Anzahl von Bytes, die von der Variable `var` belegt werden.
- `sizeof(a+b)` liefert Anzahl von Bytes, die zur Berechnung von `a+b` benötigt werden.

Um festzustellen, wie viele Bytes Ihr Compiler für die Grunddatentypen verwendet, kann z. B. das folgende Programm `typgroes.c` dienen.

```
#include <stdio.h>

int main(void) {
    printf("%20s: %d Bytes\n", "char", sizeof(char));
    printf("%20s: %d Bytes\n", "signed char", sizeof(signed char));
    printf("%20s: %d Bytes\n", "unsigned char", sizeof(unsigned char));
    printf("-----\n");
    printf("%20s: %d Bytes\n", "short", sizeof(short));
    printf("%20s: %d Bytes\n", "signed short", sizeof(signed short));
    printf("%20s: %d Bytes\n", "unsigned short", sizeof(unsigned short));
    printf("-----\n");
    printf("%20s: %d Bytes\n", "int", sizeof(int));
    printf("%20s: %d Bytes\n", "signed int", sizeof(signed int));
    printf("%20s: %d Bytes\n", "unsigned int", sizeof(unsigned int));
    printf("-----\n");
    printf("%20s: %d Bytes\n", "long", sizeof(long));
    printf("%20s: %d Bytes\n", "signed long", sizeof(signed long));
    printf("%20s: %d Bytes\n", "unsigned long", sizeof(unsigned long));
    printf("-----\n");
}
```

Kapitel 9

Die Headerdateien `<limits.h>` und `<float.h>`

9.1 `<limits.h>` - Grenzwerte von Ganzzahltypen

Diese Headerdatei definiert die Grenzwerte für die verschiedenen Ganzzahltypen. C legt für jeden dieser Werte nur einen Mindestwert fest, wie dies in Tabelle 9.1 gezeigt ist. Der absolute Betrag dieses Mindestwerts (mit gleichem Vorzeichen) darf von keinem Compiler unterschritten werden, der sich C89- bzw. C99-Compiler nennt.

Tabelle 9.1: Konstanten mit Mindestwert aus `<limits.h>`

| Konstante | Mindestwert | Beschreibung |
|------------|--|---|
| CHAR_BIT | 8 | maximale Bitanzahl für ein Byte |
| SCHAR_MIN | -127 | Minimalwert für <code>signed char</code> |
| SCHAR_MAX | +127 | Maximalwert für <code>signed char</code> |
| UCHAR_MAX | 255 | Maximalwert für <code>unsigned char</code> |
| CHAR_MIN | SCHAR_MIN oder 0 ¹ | Minimalwert für <code>char</code> |
| CHAR_MAX | SCHAR_MAX oder UCHAR_MAX ² | Maximalwert für <code>char</code> |
| MB_LEN_MAX | 1 | maximale Bytes für Vielbyte-Zeichen (länderspezifische Zeichen wie z. B. japanische oder chinesische Zeichen) |
| SHRT_MIN | -32767 | Minimalwert für <code>short int</code> |
| SHRT_MAX | +32767 | Maximalwert für <code>short int</code> |
| USHRT_MAX | 65535 | Maximalwert für <code>unsigned short</code> |
| INT_MIN | -32767 | Minimalwert für <code>int</code> |
| INT_MAX | +32767 | Maximalwert für <code>int</code> |
| UINT_MAX | 65535 | Maximalwert für <code>unsigned int</code> |
| LONG_MIN | -2147483647 | Minimalwert für <code>long int</code> |
| LONG_MAX | +2147483647 | Maximalwert für <code>long int</code> |
| ULONG_MAX | 4294967295 | Maximalwert für <code>unsigned long</code> |
| LLONG_MIN | -9223372036854775807 | (neu in C99) Minimalwert für <code>long long int</code> |
| LLONG_MAX | +9223372036854775807 | (neu in C99) Maximalwert für <code>long long int</code> |
| ULLONG_MAX | 18446744073709551615 | (neu in C99) Maximalwert für <code>unsigned long long</code> |

¹Abhängig davon, ob `char` ein vorzeichenbehafteter oder vorzeichenloser Datentyp ist

²Abhängig davon, ob `char` ein vorzeichenbehafteter oder vorzeichenloser Datentyp ist

Kapitel 10

Anweisungen und Blöcke

Fügt man zu einem Ausdruck wie `schieb >>= 4` oder `scanf("%d", &zahl)` oder `zaehl = 7` oder `nieder--` ein Semikolon hinzu, so erhält man eine *C-Anweisung*, wie z. B. folgende Anweisungen:

```
schieb >>= 4;
scanf("%d", &zahl);
zaehl = 7;
nieder--;
```

Um mehrere Anweisungen zu einem Block zusammenfassen zu können, müssen die geschweiften Klammern `{` und `}` verwendet werden. Ein solcher Block wird dann wie eine einzelne Anweisung interpretiert, wie z. B.

```
{
    zahl1 = -7;
    zahl2 = 15;
    sum = zahl1+zahl2;
    printf("%d + %d = %d\n", zahl1, zahl2, sum);
}
```

Hinweise:

1. In C beendet ein Semikolon eine Anweisung, während z. B. in Pascal ein Semikolon ein Trennzeichen zwischen den einzelnen Anweisungen ist.
2. Wie wir später sehen werden, können in einem Block auch Variablen vereinbart werden.

Kapitel 11

Die if-Anweisung

11.1 Die zweiseitige if-Anweisung

Die `if`-Anweisung wird für Programmverzweigungen benötigt: `if` (deutsch *Wenn*) und `else` (deutsch *Sonst*). Die Syntax für die vollständige (zweiseitige) `if`-Anweisung ist:

```
if (ausdruck)  
    anweisung1  
else  
    anweisung2  
anweisung1 und anweisung2 kann ein mit {} geklammerter Block von Anweisungen sein.
```

Der nach `if` in Klammern () angegebene *ausdruck* ist meist eine Bedingung. Ist die Bedingung erfüllt bzw. der Wert dieses Ausdrucks von 0 verschieden, dann wird *anweisung1*, sonst *anweisung2* ausgeführt, denn es gilt ja in C immer:

- ❑ Der Wert 0 entspricht `FALSE` (falsch).
- ❑ Ein Wert verschieden von 0 entspricht `TRUE` (wahr).

Die Funktionsweise der `if`-Anweisung kann am besten durch einen so genannten Programmablaufplan veranschaulicht werden, wie er in Abbildung 11.1 gezeigt ist.

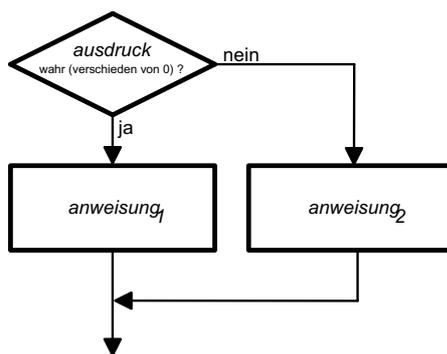


Abbildung 11.1: Programmablaufplan zur `if`-Anweisung

Kapitel 12

Die bedingte Bewertung ?:

Wenn wir das Maximum der beiden Variablen `zahl1` und `zahl2` an die Variable `max` zuweisen wollen, so würden wir mit unseren bisherigen Kenntnissen folgenden Programmteil erstellen:

```
if (zahl1 > zahl2)
    max=zahl1;
else
    max=zahl2;
```

Ist der Wert von `zahl1` größer als der Wert von `zahl2`, so wird der Wert von `zahl1` der Variablen `max`, andernfalls der Wert von `zahl2` der Variablen `max` zugewiesen.

Mit der bedingten Bewertung hätten wir diesen Programmteil auch wie folgt angeben können:

```
max = (zahl1 > zahl2) ? zahl1 : zahl2;
```

Zunächst wird hier der Ausdruck `(zahl1 > zahl2)` bewertet. Trifft diese Bedingung zu, dann wird der Variablen `max` der Wert von `zahl1`, andernfalls der Wert von `zahl2` zugewiesen.

ausdr1 ? ausdr2 : ausdr3

Zunächst wird dabei der *ausdr1* bewertet.

Ist sein Wert verschieden von 0, so wird *ausdr2* bewertet, andernfalls *ausdr3*.

In jedem Fall wird nur einer der beiden Ausdrücke *ausdr2* oder *ausdr3* bewertet; dieser Wert ist dann das Ergebnis der gesamten bedingten Bewertung.

Da der Operator `? :` eine sehr niedrige Priorität besitzt, sind runde Klammern bei *ausdr1*, *ausdr2* und *ausdr3* meist nicht notwendig, erhöhen aber die Lesbarkeit.

Das folgende C-Programm `mannweib.c` zeigt eine Anwendung des bedingten Operators:

```
#include <stdio.h>
int main(void) {
    char zeichen;
    printf("Geben Sie den Buchstaben m oder w ein: ");
    printf("----> Sie haben   ***%s***   gewaehlt\n",
        ((zeichen=getchar()) == 'm') ? ("maennlich") : ("weiblich"));
    return 0;
}
```

Kapitel 13

Die switch-Anweisung

13.1 Die switch-Anweisung

Mit der `switch`-Anweisung kann unter mehreren Alternativen, nicht nur unter zwei wie bei der `if`-Anweisung, ausgewählt werden. Die Syntax für die `switch`-Anweisung ist:

```
switch (ausdruck)
{
    case ausdr1:    anweisungen1
    case ausdr2:    anweisungen2
    ...
    case ausdrN:    anweisungenN
    default:         anweisungenD
}
```

Es wird der bei `switch` angegebene (*ausdruck*) ausgewertet und das Ergebnis mit den einzelnen `case`-Ausdrücken, die `int`- oder `char`-Werte (Konstanten) liefern müssen, verglichen.

Wird keine Übereinstimmung gefunden, so verzweigt das Programm zur `default`-Marke, falls diese angegeben wurde.

Wird keine Übereinstimmung gefunden und es ist keine `default`-Marke angegeben, so wird keine Anweisung des `switch`-Blocks ausgeführt. Fehlt also die `default`-Marke, so wird der gesamte `switch`-Block übersprungen, falls keine Übereinstimmung in den `case`-Marken vorliegt.

Die `switch`-Anweisung ist mit einem Programm-Schalter zu vergleichen und wird im Struktogramm - wie in Abbildung 13.1 gezeigt - dargestellt.

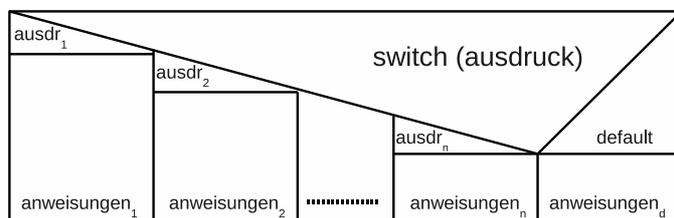


Abbildung 13.1: Struktogramm zur `switch`-Anweisung

Kapitel 14

Der Komma-Operator

14.1 Der Komma-Operator

Der Komma-Operator fasst mehrere Ausdrücke syntaktisch zu einem einzigen Ausdruck zusammen.

Sind mehrere Ausdrücke durch ein Komma (,) getrennt, so werden sie von links nach rechts bewertet; Datentyp und Wert des Gesamtausdrucks ist dann gleich dem Datentyp und Ergebnis der letzten Operation.

Beispiel:

```
zeich = 'J', zaehl = 9, pi = 3.1415;
```

Alle drei Zuweisungen werden nacheinander berechnet, das Ergebnis des Gesamtausdrucks ist gleich 3.1415.

Im Zusammenhang mit Schleifenvariablen, die wir im nächsten Kapitel behandeln werden, wird dieser Komma-Operator häufig verwendet. Der Komma-Operator erlaubt nämlich die Unterbringung von mehreren Zuweisungen an einer Stelle, wo sonst nur eine erlaubt wäre.

Mann kann den Komma-Operator auch innerhalb einer `if`-Anweisung verwenden, um sich die geschweiften Klammern einer Blockanweisung zu sparen, wie z. B.:

```
if (...)
    zaehl++, printf ("\n Zaehler erhoeht\n");
```

Sonst hätte man schreiben müssen:

```
if (...) {
    zaehl++;
    printf ("\n Zaehler erhoeht\n");
}
```

Aus Gründen der besseren Lesbarkeit ist aber immer die zweite Vorgehensweise vorzuziehen.

Kapitel 15

Die for-Anweisung

15.1 Die for-Anweisung

Syntaktisch sieht die `for`-Schleife wie folgt aus:

```
for (ausdruck1; ausdruck2; ausdruck3)  
  anweisung
```

Die *anweisung* kann ein Block von Anweisungen sein, der mit { . . . } zu klammern ist.

- *ausdruck1* initialisiert die Schleifenvariable(n).
- *ausdruck2* legt das Abbruchkriterium für die Schleife fest, wenn *ausdruck2* nicht erfüllt ist.
- *ausdruck3* reinitialisiert die Schleifenvariable(n).

Die Komponenten (*ausdrücke*) können einzeln oder auch insgesamt fehlen, jedoch müssen die Semikolons in der Klammer an den richtigen Stellen verbleiben.

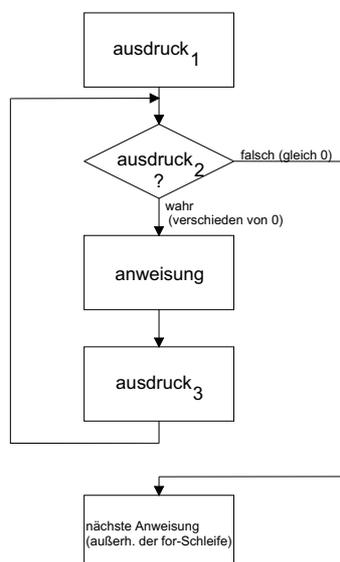


Abbildung 15.1: Programmablaufplan zur `for`-Schleife

Jetzt wollen wir nochmals die geometrische Reihe berechnen, wobei wir alle erforderlichen Schleifenanweisungen im `for`-Schleifenkopf unterbringen, so dass überhaupt keine Schleifenanweisung mehr übrigbleibt.

```
#include <stdio.h>

int main(void)
{
    int    zaehl, n;
    double reih_teil, summe;

    printf("Geben Sie n ein !\n");
    scanf("%d",&n);

    for (reih_teil = 1, summe = 1, zaehl = 1;
        zaehl <= n;
        reih_teil /= 2, summe += reih_teil, ++zaehl)
        ; /* leere Anweisung */

    printf("\n\nDie Summe der "
           "geometrischen Reihe bis %d ist:\n", n);
    printf("%lf\n", summe);
    return 0;
}
```

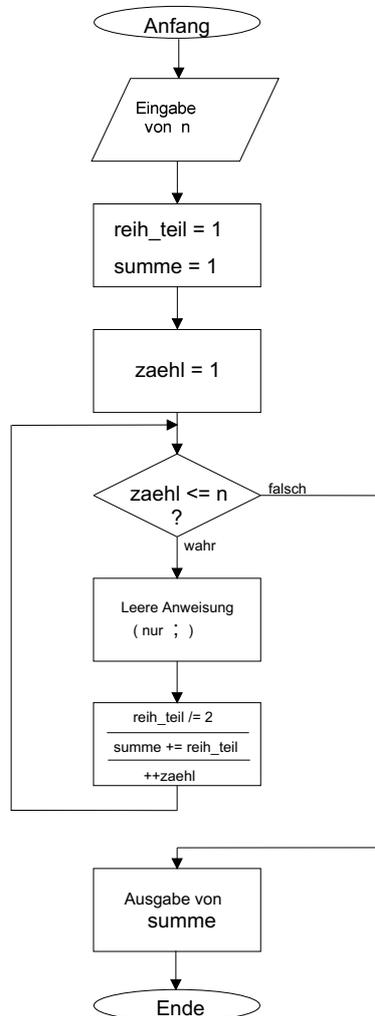


Abbildung 15.6: C-Programm `georeih3.c`

Hier haben wir als einzige Schleifenanweisung eine leere Anweisung. Für dieses Programm ist `reih_teil` mit dem Wert 1 vorzubereiten.

Der Komma-Operator erlaubt also, mehrere Ausdrücke zu einem einzigen Ausdruck zu verschmelzen. Diese Technik lässt sich vor allen Dingen bei der `for`-Schleife anwenden, wenn mehrere Schleifenvariablen zu initialisieren und/oder zu reinitialisieren sind.

Generell sollte man jedoch darauf achten, den `for`-Schleifenkopf nicht zu überladen. Um die Übersichtlichkeit und Lesbarkeit eines Programms zu wahren, ist es öfteren besser, Variablen in eigenen Anweisungen zu initialisieren oder zu verändern, wenn sie nicht unmittelbar zur Schleifensteuerung gehören.

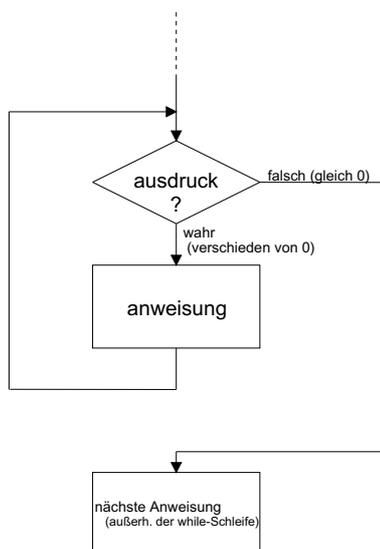
Kapitel 16

Die while-Anweisung

16.1 Die while-Anweisung

while (*ausdruck*)
anweisung
anweisung kann wieder ein mit { . . } zu klammernder Block von Anweisungen sein.

Die Funktionsweise der `while`-Schleife lässt sich am besten anhand der Abbildung 16.1 zeigen.



Vor jedem Schleifendurchlauf wird *ausdruck* berechnet. Solange das Ergebnis verschieden von 0 (TRUE) ist, wird die Schleifen-*anweisung* ausgeführt. Erst, wenn die Auswertung von *ausdruck* 0 (FALSE) liefert, wird die Schleife beendet und mit der nächsten nicht zur Schleife gehörigen Anweisung fortgefahren.

Ist das Ergebnis von *ausdruck* bereits bei der ersten Berechnung gleich 0 (FALSE), so wird die Schleife gar nicht durchlaufen, sondern sofort mit der nächsten nicht zur Schleife gehörigen Anweisung fortgefahren.

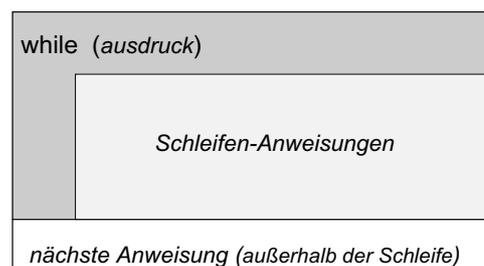


Abbildung 16.1: Programmablaufplan und Struktogramm zur `while`-Schleife

Da die Abfrage der Schleifen-Bedingung wie bei der `for`-Schleife am Anfang durchgeführt wird, wird die `while`-Schleife genauso wie die `for`-Schleife im Struktogramm dargestellt.

Kapitel 17

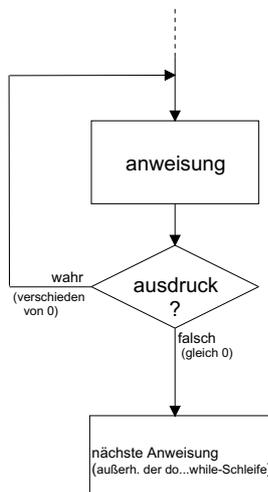
Die do...while-Anweisung

17.1 Die do...while-Anweisung

```
do  
    anweisung  
while (ausdruck);
```

Die *anweisung* kann selbstverständlich wieder ein Block von Anweisungen sein, der dann mit { . . } zu klammern ist.

Die Funktionsweise der do...while-Schleife lässt sich am besten anhand der Abbildung 17.1 zeigen.



Nach jedem Schleifendurchlauf wird *ausdruck* berechnet. Ist das Ergebnis verschieden von 0 (TRUE), so wird die Schleifen-*anweisung* nochmals ausgeführt. Erst, wenn die Auwertung von *ausdruck* 0 (FALSE) liefert, wird die Schleife beendet und mit der nächsten nicht zur Schleife gehörigen Anweisung fortgefahren.

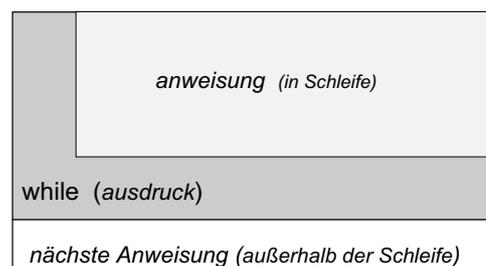


Abbildung 17.1: Programmablaufplan und Struktogramm zur do...while-Schleife

Da die Abfrage der Bedingung bei der do...while-Schleife am Ende stattfindet, wird diese Schleife im Struktogramm – wie rechts in Abbildung 17.1 gezeigt – dargestellt.

Kapitel 18

Die break-Anweisung

18.1 Die break-Anweisung

Nicht immer ist es sinnvoll, eine Schleife nur über das Abbruchkriterium zu verlassen. Die `break`-Anweisung dient dazu, um eine `for`-, `while`-, `do...while`-Schleife oder eine `switch`-Anweisung vorzeitig zu verlassen. Dazu muss lediglich `break` aufgerufen werden.

for-Schleife:

```
for (ausdr1 ; ausdr2 ; ausdr3) {
    .....
    .....
+----break;
| .....
| .....
| }
+-->anweisung; /* ausserhalb for-Schleife */
```

while-Schleife:

```
while (ausdruck) {
    .....
    .....
+----break;
| .....
| .....
| }
+-->anweisung; /* ausserhalb while-Schleife */
```

do...while-Schleife:

```
do {
    .....
    .....
+----break;
| .....
| .....
| } while (ausdruck);
+-->anweisung /* ausserhalb do...while */
```

switch-Anweisung:

```
switch (ausdruck) {
    .....
    case x:
        .....
+-----break;
| .....
| .....
| }
+-->anweisung; /* ausserhalb switch */
```

Kapitel 19

Die continue-Anweisung

19.1 Die continue-Anweisung

Die `continue`-Anweisung ist der `break`-Anweisung ähnlich. Sie bewirkt allerdings im Unterschied zur `break`-Anweisung nicht den Abbruch der gesamten Schleife, sondern nur den Abbruch des aktuellen Schleifendurchlaufs, also einen Sprung zum Schleifenende. `continue` leitet somit unverzüglich den nächsten Schleifendurchlauf ein und hat auch im Gegensatz zu `break` keine Auswirkung auf `switch`-Anweisungen.

Die `continue`-Anweisung bewirkt also eine sofortige Wiederholung der betreffenden Schleife. Bei `while` und `do...while` bedeutet dies, dass sofort wieder die Bedingung (*ausdruck*) ausgewertet wird. Bei `for` wird als nächstes die Reinitialisierung (*ausdr3*) durchgeführt und dann zur Schleifenbedingung (*ausdr2*) verzweigt.

Das folgende C-Programm `vollkomm.c` gibt alle vollkommenen Zahlen zwischen `x` und `y` aus. Eine Zahl `z` ist vollkommen, wenn die Summe aller ihrer Teiler (ohne `z` selbst) gleich `z` ist.

```
#include <stdio.h>
int main(void) {
    long int    sum, innen, aussen, von, bis;
    printf("Vollkommene Zahlen von? "); scanf("%ld", &von);
    printf("Vollkommene Zahlen bis? "); scanf("%ld", &bis);
    for (ausсен = von; aussen <= bis; aussen++) {
        sum = 0;
        for (innen = 1; innen <= aussen / 2; innen++) {
            if ((ausсен % innen) != 0)
                continue;
            sum += innen;
            if (sum > aussen)
                break;
        }
        if (sum == aussen)
            printf("%ld\n", sum);
    }
    return 0;
}
```

Kapitel 20

Marken und die goto-Anweisung

20.1 Marken und die goto-Anweisung

Die Syntax der `goto`-Anweisung ist:

```
goto marke ;  
Diese Anweisung bewirkt einen Sprung zu der Programmstelle, an der  
marke :  
angegeben ist.
```

Eine `marke` hat dabei die gleiche Form wie ein Variablenname¹; anschließend folgt ein Doppelpunkt:

```
marke: anweisung;
```

`marken` können vor jeder Anweisung stehen, unabhängig davon, ob sie innerhalb des Programms mit `goto` angesprungen werden oder nicht. Nach einer `marke` muss immer mindestens eine Anweisung angegeben sein, eventuell auch nur die leere Anweisung:

```
marke: ;
```

In einem Programm darf dieselbe Marke nur einmal angegeben sein.

20.2 Programmiertechniken

20.2.1 goto nur im äußersten Notfall

Mit Einführung der Strukturierten Programmierung in den 60er Jahren wurde die Springerei mit `goto` sehr verpönt. Damals waren Programme vor allen Dingen in Assembler geschrieben. Dort war und ist zum Teil noch der Sprung zu einer Code-Adresse ein fundamentales Prinzip. Später wurde dieser Stil vor allen Dingen in **BASIC** übernommen. Man stellte damals sehr schnell fest, dass Programme, die mit Vorwärts- und Rückwärtssprüngen gespickt sind, nicht nur schwer verständlich, sondern auch kaum wartbar waren. Dies führte zu einer erheblichen Verteuerung der Software. Ein Ausweg aus diesem Dilemma war die *Strukturierte Programmie-*

¹muss mit Buchstabe oder Unterstrich beginnen und des weiteren dürfen nur Buchstaben, Ziffern oder Unterstriche folgen

Kapitel 21

Grafikprogrammierung unter Linux

Im Rahmen dieses Buches wurde eine einfache Grafik-Bibliothek namens LCGI entwickelt, welche von der Homepage heruntergeladen werden kann.

21.1 Benutzung von LCGI

1. Inkludieren von `<graphics.h>`

Da die hier vorgestellten Funktionen in der Headerdatei `graphics.h` deklariert sind, sollte immer Folgendes angegeben werden, wenn man von diesen Funktionen in seinem Programm Gebrauch macht.

```
#include <graphics.h>
```

2. Angabe von `main()` mit Parametern

Wenn Sie in Ihrem Programm Routinen aus diesem Kapitel aufrufen, müssen Sie statt

```
int main(void)
```

immer Folgendes angeben:

```
int main( int argc, char *argv[] )
```

3. Kompilieren und Linken von Grafikprogrammen

Wenn Sie ein Programm erstellt haben, das Routinen aus diesem Kapitel aufruft, müssen Sie dieses immer mit dem mitgelieferten Kommando `lcc` statt `gcc` bzw. `cc` kompilieren und linken.

Wenn Sie z. B. ein Programm `polygon.c` entworfen haben und dieses nun kompilieren wollen, müssen Sie z. B. Folgendes aufrufen:

```
user@linux:~ > lcc -o polygon polygon.c ↵
```

`lcc` bietet die gleichen Optionen an wie das Kommando `gcc` bzw. `cc`.

Geben Sie danach zum Testen die folgende Kommandozeile ein:

```
user@linux:~ > ./polygon ↵
```

21.2 Grafikmodus ein- und ausschalten

`initgraph(int breite, int hoehe)`

schaltet den Grafikmodus ein, indem ein Fenster eingeblendet wird, das `breite` Pixel breit und `hoehe` Pixel hoch ist.

`closegraph()`

beendet den Grafikmodus. Da in diesem Fall auch das Grafikfenster gelöscht wird, wird meist vor `closegraph()` noch ein `getch()` angegeben, so dass erst auf einen Tastendruck des Benutzers hin der Grafikmodus verlassen wird.

Typisch für die Grafikprogrammierung ist deshalb z. B. folgender Programmausschnitt:

```
#include <graphics.h>
.....
int main( int argc, char *argv[] )
{
    initgraph( 640, 480 );
    .....
    Grafik-Programmteil
    .....
    getch(); /* auf einen Tastendruck warten */
    closegraph();
}
```

21.3 Eingaben im Grafikmodus

Während man im Grafikmodus arbeitet, kann man nicht mehr die Standardroutinen für die Ein- und Ausgabe (`printf()`, `scanf()`, `getchar()` und `putchar()`) verwenden, sondern muss die eigens dafür angebotenen Routinen verwenden, welche nachfolgend vorgestellt sind:

`getcharacter(char *text, ...)`

`getstring(char *text, ...)`

`getint(char *text, ...)`

`getdouble(char *text, ...)`

fordern den Benutzer durch Ausgabe des Textes `text` zur Eingabe eines Zeichens, eines Textes, einer ganzen Zahl oder einer Gleitpunktzahl auf und liefern diese Eingabe zurück.

`kbhit()`

prüft, ob eine Taste gedrückt wurde. Falls eine Taste gedrückt wurde, liefert `kbhit()` einen `int`-Wert ungleich 0 (TRUE), falls nicht, liefert diese Funktion 0 (FALSE) zurück. Das dabei eingegebene Zeichen kann mit der nachfolgend vorgestellten Routine `getch()` nachträglich erfragt werden.

`getch()`

liest ein Zeichen ein. Anders als bei `getchar()` findet hier jedoch keine Zwischenpufferung statt, sondern das Zeichen wird direkt von der Tastatur gelesen. Das eingegebene Zeichen wird dabei nicht am Bildschirm angezeigt. In jedem Fall liefert diese Routine

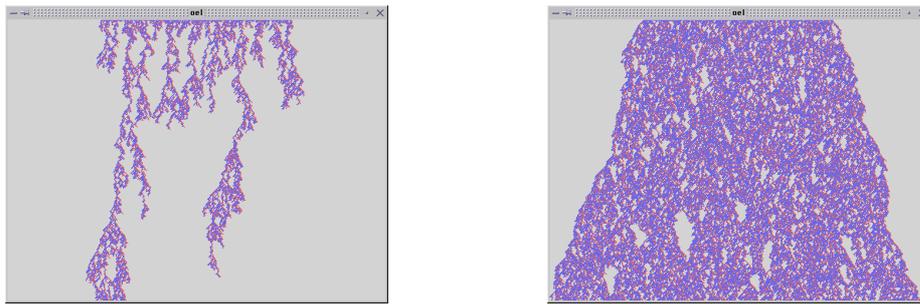


Abbildung 21.3: Wahrscheinlichkeit von 0.62 (links) und 0.7 (rechts) für das Eindringen von Öl

Beispiel:

Das folgende Programm `pixel.c` wählt zufällig eine Hintergrundfarbe und malt dann 1000 kleine Vierecke an zufällige Bildschirmpositionen mit zufällig gewählten Farben. Ein Viereck sind dabei 9 Pixel:

```
xxx
xox
xxx
```

wobei nur der Mittelpunkt (o) zufällig gewählt ist. Danach wird der Inhalt dieses Fensters gelöscht, mit einer zufälligen Hintergrundfarbe gefüllt und wieder mit neuen zufälligen kleinen Rechtecken bemalt. Mit einem Tastendruck kann dieses Programm beendet werden.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <graphics.h>
#define PIXEL_ZAHL 1000
int main( int argc, char *argv[] ) {
    int i, maxx, maxy, x, y, farbe, max_farbe;
    srand(time(NULL));
    initgraph( 640, 480 );
    maxx=getmaxx(); maxy=getmaxy(); max_farbe = getmaxcolor();
    while ( !kbhit() ) {
        cleardevice( rand()%(max_farbe+1) );
        for ( i=1 ; i<=PIXEL_ZAHL ; i++ ) {
            x = rand()%maxx; y = rand()%maxy; farbe = rand()%(max_farbe+1);
            putpixel(x, y, farbe); putpixel(x-1,y-1,farbe); putpixel(x,y-1,farbe);
            putpixel(x+1,y-1,farbe); putpixel(x+1,y,farbe); putpixel(x+1,y+1,farbe);
            putpixel(x,y+1,farbe); putpixel(x-1,y+1,farbe); putpixel(x-1,y,farbe);
        }
    }
    closegraph();
    return 0;
}
```

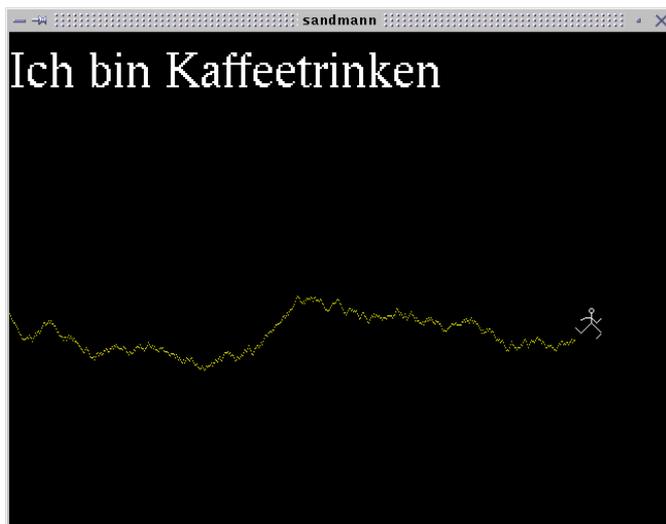
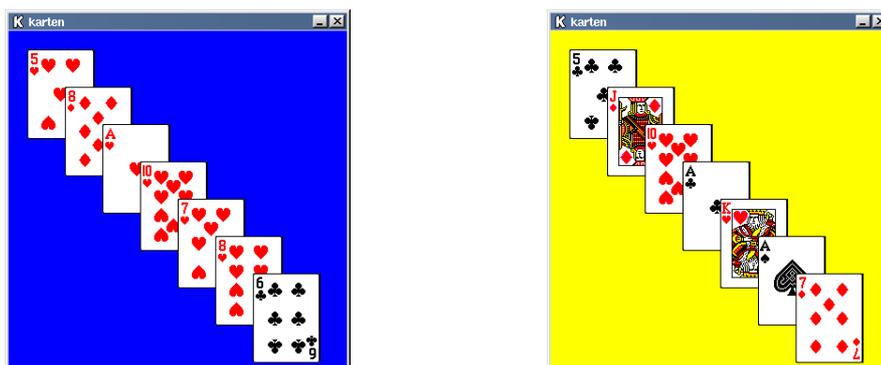
Abbildung 21.10: Anzeige des Programms `sandmann.c`

Abbildung 21.10 zeigt das von Programm `sandmann.c` angezeigte Fenster zu einem bestimmten Zeitpunkt. Dieses Programm könnte z. B. gestartet werden, wenn man seinen Bildschirmplatz längere Zeit verlässt, und man Information am Bildschirm darüber ausgeben lassen möchte, wo man sich gerade befindet und wann man wieder zurückkommt. Hierzu müsste man jedoch dann die auszugebende Meldung einlesen lassen. Eine entsprechende Anpassung dieses Programms an die speziellen eigenen Bedürfnisse sollte aber nicht allzu schwierig sein.

Beispiel:

Das folgende Programm `karten.c` blendet bei jedem Tastendruck immer sieben zufällig ausgewählte Karten ein, die es übereinander legt (siehe auch Abbildung 21.11). Mit der `ESC`-Taste kann dieses Programm beendet werden.

Abbildung 21.11: Unterschiedliche Anzeigen des Programms `karten.c` nach einem Tastendruck

Kapitel 22

Funktionen

In der Programmierpraxis sind häufig Probleme anzutreffen, die einander von der Aufgabenstellung her ähnlich sind. Dieser Tatsache wird in der Programmiersprache C mit so genannten Funktionen (*functions*) Rechnung getragen.

22.1 Allgemeines zu Funktionen

Wir haben bisher schon – ohne es ausführlich zu erwähnen – mit Funktionen gearbeitet. So haben wir z. B. zur Ausgabe auf dem Bildschirm die Funktion `printf()` aufgerufen. Hinter diesem Funktionsnamen verbirgt sich ein Programmteil, der in der Standardbibliothek definiert ist und bei jedem Aufruf von `printf()` ausgeführt wird.

22.1.1 Allgemeines Beispiel zu Funktionen

Am besten können wir das Prinzip von Funktionen an einem allgemeinen Beispiel nachvollziehen. In diesem Beispiel wird nicht die Programmiersprache C, sondern – zum besseren Verständnis – die deutsche Sprache verwendet. Es soll bei der Personalabteilung einer Firma immer ein einheitliches Papier für die Spesenabrechnung abgegeben werden. Wir nehmen dazu an, dass in der Standardbibliothek eine Funktion (Schablone) mit Namen `spesen()` definiert sei, was natürlich nicht für die wirkliche Programmbibliothek gilt.

| Funktionsname | Liste der formalen Parameter |
|---------------|------------------------------|
| | |
| | |
| v | v |

```
spesen( name, stadt, preis, vor, haupt, nach, tagzeit, waehrung, datum )
{
    Drucke folgendes aus:
    "Herr/Frau name hat am datum in stadt
    zu(m) tagzeit:
        vor als Vorspeise
        haupt als Hauptspeise und
        nach als Nachspeise
    gegessen.
    Der Preis fuer dieses Essen betrug preis waehrung."
}
```

In der Definition von `tausch()` werden mit der Deklaration

```
void tausch(int *a, int *b)
```

zwei Speicherplätze für `int`-Zeiger und ein Speicherplatz für die lokale Variable `hilf` reserviert.

Mit dem Aufruf

```
tausch(&zahl_1, &zahl_2);
```

werden dabei dann die Adressen von `zahl_1` und `zahl_2` als aktuelle Argumente an die Funktion `tausch()` übergeben; als Erinnerung: `&` = Adreßoperator. Bei dieser Übergabe werden also in die beiden reservierten Speicherplätze `a` und `b` der Funktion `tausch()` die Adressen von `zahl_1` und `zahl_2` abgelegt; siehe Abbildung 22.11.

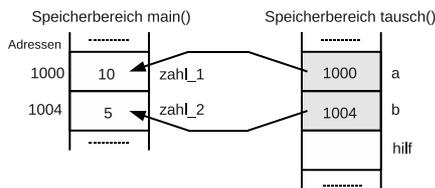
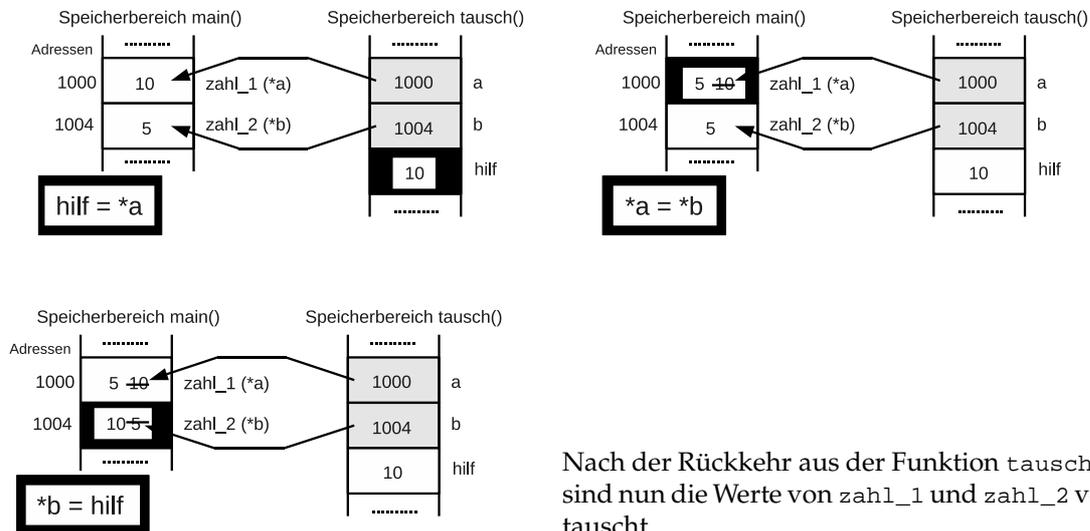


Abbildung 22.11: Übergabe der Adressen von `zahl_1` und `zahl_2` an `tausch()`

Der Ablauf innerhalb der Funktion `tausch()` ist bildlich in der Abbildung 22.12 dargestellt.

```
hilf = *a;
*a = *b;
*b = hilf;
```



Nach der Rückkehr aus der Funktion `tausch()` sind nun die Werte von `zahl_1` und `zahl_2` vertauscht.

Abbildung 22.12: Call by Reference durch Übergabe von Adressen

und

```
float *welch_funktion(...);
/* welch_funktion = Funktion, die Zeiger auf float-Wert liefert */
```

sollten Sie sich klarmachen.

Will man nun über den Zeiger `welch_funktion` eine Funktion aufrufen, so muss man `*welch_funktion` angeben, da `welch_funktion` ein Zeiger und folglich `*welch_funktion` die eigentliche Funktion ist.

Wenn z. B. `welch_funktion` auf die Funktion `zyl_vol()` zeigt, dann entspricht die Anweisung

```
return rad*(*welch_funktion)(rad);
```

der Anweisung

```
return rad * zyl_vol(rad);
```

wie dies auch in Abbildung 22.20 gezeigt ist.

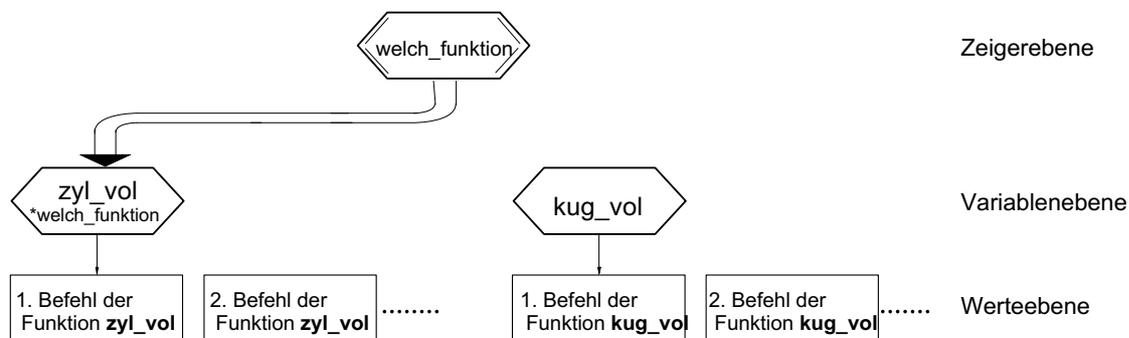


Abbildung 22.20: `*welch_funktion` entspricht `zyl_vol()`

Um das Zeigerprinzip bei Funktionen besser verstehen zu können, wollen wir ein weiteres Deklarations-Beispiel angeben:

```
float *(*welch_funktion)(...);
```

Hier wäre `welch_funktion` ein Zeiger auf eine Funktion, die einen Zeiger auf einen `float`-Wert liefert.

22.7.2 Typische Anwendungen

Auswahl von Funktionen zur Programmaufzeit

Ein typischer Anwendungsfall für Zeiger auf Funktionen liegt vor, wenn erst zum Zeitpunkt der Programmausführung bekannt ist, welche Funktion auszuführen ist. Nehmen wir z. B. ein Sortierprogramm, bei dem erst während der Programmausführung bekannt ist, welche Daten zu sortieren sind: Ganze Zahlen, Zeichenketten oder sonstige Daten. In diesem Fall schreibt man nur eine Sortierfunktion, die neben anderen Parametern auch einen Zeiger auf eine Funktion erwartet, die für den Vergleich der einzelnen Daten zuständig ist. Ist dann während der

Kapitel 23

Speicherklassen und Modultechnik

Es wurde schon einmal erwähnt, dass sich C-Programme aus mehreren Dateien zusammensetzen können. Jede Programmdatei – auch *Modul* genannt – kann einzeln kompiliert werden, aber ist für sich allein nicht lauffähig.

Diese getrennt kompilierten Module können anschließend mit Hilfe des Linkers zu einem einzigen Programm zusammengebunden werden.

23.1 Gültigkeitsbereich, Lebensdauer, Speicherort

23.1.1 Gültigkeitsbereich

Unter dem Gültigkeitsbereich eines Objekts (*Funktionsname*, *Variablenname*) versteht man die Bereiche in den einzelnen Programmeinheiten, in denen es angesprochen werden kann.

Beim Gültigkeitsbereich wird zwischen folgenden Arten unterschieden:

- modulglobal
- programmglobale
- lokal (in einem Block von { ... })

Modulglobale Gültigkeit

Ein *modulglobales* Objekt ist innerhalb der gesamten Programmdatei (Modul) ansprechbar, in der es deklariert ist, aber nicht in einer anderen Programmdatei (Modul). Jede Variable, die außerhalb einer Funktion deklariert ist, werden wir als *modulglobal* bezeichnen.

Da modulglobale Variablen außerhalb von Funktionen vereinbart sind und somit vielen Funktionen zur Verfügung stehen, kann innerhalb von Funktionen auf modulglobale Variablen zugegriffen werden, ohne dass diese als Parameter zu übergeben sind.

Beispiel:

In diesem Beispiel werden wir ein C-Programm `tausch3.c` erstellen, das mit Hilfe einer Funktion `tausch()` die Werte zweier Variablen vertauscht. In diesem Programm werden die entsprechenden Variablen jedoch nicht als Zeigervariablen übergeben.

Kapitel 24

Präprozessor-Direktiven

Der Präprozessor verarbeitet den Quelltext einer Programmdatei, wobei alle Präprozessor-Kommandos (Präprozessor-Direktiven) mit dem Zeichen # beginnen. Zwischenraum-Zeichen (engl. *whitespace*: Leerzeichen, \f, \n, \r, \t oder \v) und Kommentare sind vor # zugelassen; zwischen # und Anfang der restlichen Präprozessor-Direktive sind nur Leerzeichen, \t und Kommentare zugelassen.

Beispiel:

So sind z. B. die folgenden drei Angaben erlaubte Präprozessor-Direktiven in C89 und C99:

```
/* Kommentar */ #if TEST
# /* Kommentar */ if TEST
# if /* Kommen-
        tar */ TEST
```

Üblicherweise ruft der Compiler automatisch den Präprozessor auf, bevor er mit der Übersetzung beginnt.

C89 schreibt vor, dass der Präprozessor wie ein eigener Schritt vor dem eigentlichen Compilerlauf zu verstehen ist¹.

Der Präprozessor bietet nun die folgenden Dienstleistungen an:

- Bedingte Kompilierung (#ifdef, ...)
- Einkopieren von anderen Dateien (#include)
- Makro-Ersetzung (#define)
- Makrodefinition aufheben (#undef)
- Vordefinierte Makronamen (__LINE__, __FILE__, ...)
- Zeilenumerierung (#line)
- Fehlermeldungen (#error)
- Pragmas (#pragma)
- Null-Direktive (#)

¹Das heißt nicht, dass der Präprozessor-Lauf als eigener Durchgang (wie es in heutigen Compilern oft der Fall ist) realisiert sein muss, sondern sich nur so verhalten muss

Beispiele:

```
#if bed1
    progteil1
#elif bed2
    progteil2
#elif bed3
    progteil3
#else
    progteil4
#endif
```

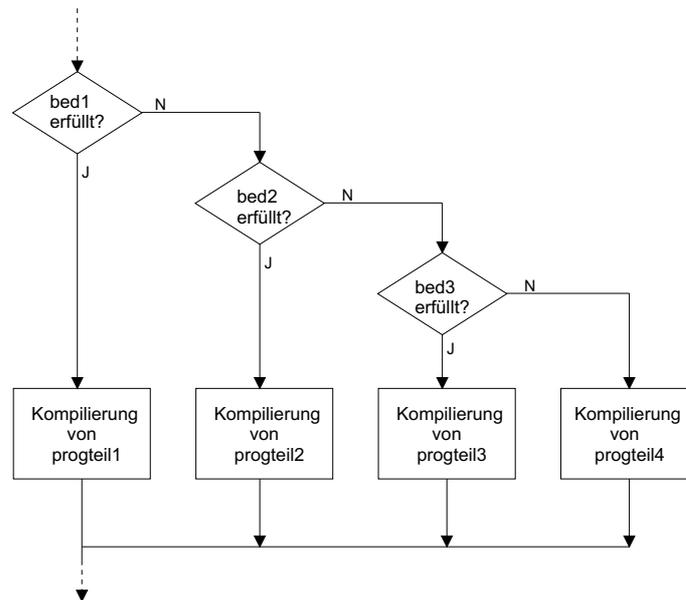


Abbildung 24.1: Programmablaufplan zu einer #if-Kaskade

Schachteln von #if-Konstruktionen ist erlaubt, wie z. B.:

```
#if beda
#   if beda1
        progteila1
#   else
        progteila2
#   endif
#else
#   if bedb1
        progteila1
#   else
        progteila2
#   endif
#endif
```

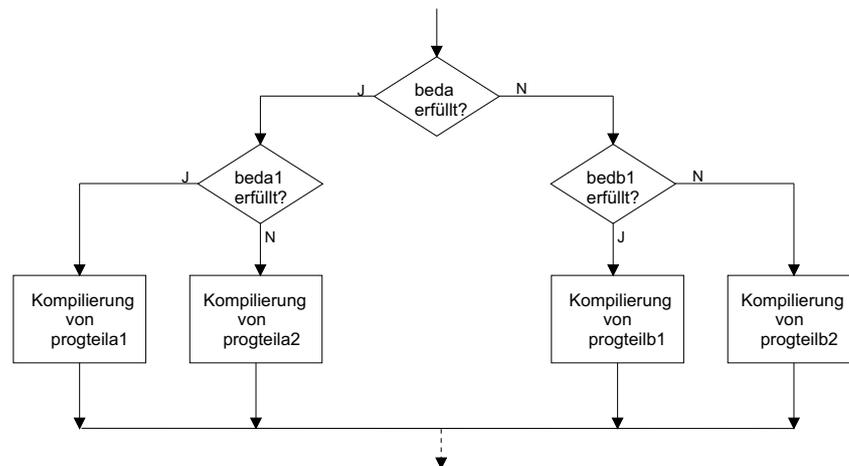


Abbildung 24.2: Programmablaufplan zu einer geschachtelten #if-Konstruktion

Kapitel 25

Zeiger und Arrays

In der Programmierpraxis sind häufig Probleme anzutreffen, bei denen man nicht mit einfachen Variablen auskommt, sondern einen ganzen Block von Variablen des gleichen Typs benötigt. Ein Beispiel dafür wären die Monatumsätze eines Unternehmens für ein Jahr. In diesem Fall ist es sehr nützlich, 12 `double`-Variablen in einem Block zu definieren, an dem nur ein Name vergeben wird. Zur Unterscheidung der Monate wird dann z. B. ein Index (ganze Zahl) verwendet.

25.1 Eindimensionale Arrays

Ein *Array* ist die namentliche Zusammenfassung einer Anzahl von gleichartigen Objekten eines Datentyps, wie z. B. `int`- oder `char`-Variablen. Anstelle des Begriffes *Array* wird manchmal auch der Begriff *Vektor* verwendet. Hier werden wir immer den Begriff *Array* benutzen.

25.1.1 Eindimensionale Arrays

Ein eindimensionales Array mit 26 Elementen vom Typ `char` könnte wie folgt vereinbart werden:

```
char buchst[26];
```

Mit dieser Angabe wird ein Array `buchst` mit 26 Elementen vom Typ `char` definiert. Die Anzahl der Elemente wird bei der Definition in eckigen Klammern angegeben. Unter

```
char buchst[26];
```

können wir uns die Reservierung eines Speicherblocks mit Namen `buchst` und mit 26 aufeinanderfolgenden Elementen (hier also 26 `char`-Variablen) vorstellen, wie dies auch links in Abbildung 25.1 gezeigt ist. Mit der Definition

```
char buchst[26];
```

werden also, wenn man so will, auf einmal 26 `char`-Variablen (`buchst[0]`, `buchst[1]`, `buchst[2]`, ..., `buchst[25]`) festgelegt.

Werden n Elemente (hier: $n = 26$) für ein Array definiert, so erfolgt die Adressierung über so genannte Indizes von Element 0 bis Element $n - 1$.

Beachten Sie, dass jedes Array mit der Elementnummer 0 und nicht mit 1 beginnt!

Mit den Anweisungen

```
buchst[3] = 'a';
buchst[0] = '!';
buchst[24] = 'H';
```

ergäbe sich dann ein Speicherbild, wie es rechts in Abbildung 25.1 gezeigt ist.

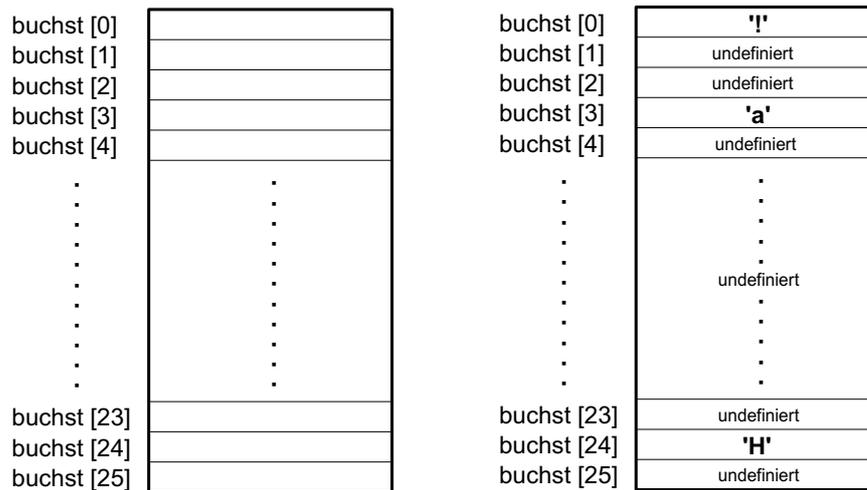


Abbildung 25.1: char-Array buchst mit 26 Speicherplätzen

Beispiele:

```
int tage[32];
    definiert ein Array tage mit 32 int-Variablen tage[0], ..., tage[31].

int *zeig[20];
    definiert ein Array zeig mit 20 Zeigern zeig[0], ..., zeig[19] auf int-Variablen.
```

Beispiel:

Wir wollen nun ein C-Programm `buchstnr.c` erstellen, das zunächst in einem Array `buchst` alle Kleinbuchstaben speichert. Danach kann der Benutzer eine Nummer eingeben, zu der ihm der entsprechende Kleinbuchstabe am Bildschirm ausgegeben wird. Unser Programm soll zur Ermittlung des Buchstabens auf das Array `buchst` zugreifen.

```
#include <stdio.h>

int main(void) {
    char buchst[26];
    int i, zahl;
    for (i = 0; i < 26; i++) /* Array mit Kleinbuchstaben belegen */
        buchst[i] = 'a' + i;
```

Kapitel 26

Argumente auf der Kommandozeile

26.1 Die Parameter `argc` und `argv` der Funktion `main()`

Jedes C-Programm muss – wie wir wissen – einen Hauptprogrammteil `main()` besitzen. `main()` ist aber nichts anderes als eine Funktion, die beim Start des C-Programms vom Betriebssystem oder dem C-Laufzeitsystem aufgerufen wird. Wir gingen bisher davon aus, dass `main()` ohne Parameter arbeitet:

```
int main(void)
```

In Wirklichkeit werden aber bei jedem Aufruf der Funktion `main()` zwei Argumente mitgegeben, die wie folgt definiert sind:

```
int main(int argc, char *argv[])
```

Die beiden Parameter `argc` und `argv` ermöglichen die Übergabe von Strings an das aufgerufene Programm, wobei gilt:

Erster Parameter **`argc`**

ist die Anzahl der übergebenen Strings (Worte).

Zweiter Parameter **`argv`**

ist ein Array von Zeigern, wobei jeder dieser Zeiger auf den Anfang eines Strings (Worts) aus der Kommandozeile zeigt.

Das folgende Programm `argtest.c` demonstriert die Verwendung von `argc` und `argv`:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    for (i=0; i<argc; i++)
        printf("Wort %d: %s\n", i, argv[i]);
    return 0;
}
```

Üblicherweise wird ein solches Programm, nachdem es kompiliert und gelinkt wurde, wie z. B.:

```
user@linux:~ > cc -o argtest argtest.c ↵
```

Kapitel 27

Dynamische Speicher-Reservierung und -Freigabe

27.1 Nachteile von statischen Arrays

Das folgende Programm `namsort1.c`, das bis zu 100 Namen einliest und diese dann sortiert ausgibt, zeigt die Nachteile von statischen Arrays in gewissen Anwendungen.

```
#include <stdio.h>
#include <string.h>
#define MAX_NAMEN 100
#define MAX_LAENGE 30
int main(void) {
    int i, j, n;
    char nam[MAX_NAMEN][MAX_LAENGE], hilf[MAX_LAENGE];
    printf("Gib Deine Namen ein (Abschluss mit Leerzeile)\n"); /* Einlesen der Namen */
    for (n = 0; n < MAX_NAMEN; n++) {
        gets(nam[n]);
        if (strlen(nam[n]) == 0)
            break;
    }
    for (i = 0; i < n-1; i++) /* Sortieren der Namen */
        for (j = i+1; j < n; j++)
            if (strcmp(nam[i], nam[j]) > 0) {
                strcpy(hilf, nam[i]);
                strcpy(nam[i], nam[j]);
                strcpy(nam[j], hilf);
            }
    printf("----- Sortierte Namensliste\n");
    for (i = 0; i < n; i++)
        printf("%s\n", nam[i]);
    return 0;
}
```

Kapitel 28

Strukturen

Unter einer Struktur versteht man die Zusammenfassung von einer oder mehreren Variablen, die – anders als bei Arrays – untereinander auch unterschiedliche Datentypen besitzen dürfen, zu einer Einheit.

28.1 Deklaration und Definition von Strukturen

28.1.1 Deklaration von Strukturen

Ein typisches Beispiel für eine einfache Struktur wäre die Zusammenfassung der Daten eines Studenten aus einer Studentenkartei zu einer einzigen Einheit, die wir `stu_daten` nennen. Eine solche Zusammenfassung wird in C z. B. folgendermaßen ausgedrückt:

```
struct stu_daten {
    char name[20];      /* Nachname          */
    char vorname[20];  /* Vorname          */
    long postleit_zahl; /* Postleitzahl     */
    char wohnort[20];  /* Wohnort         */
    char strass_nr[20]; /* Strasse, Hausnummer */
    char geburtdat[11]; /* Geburtsdatum: tt.mm.jjjj */
    long matrikelnr;   /* Matrikelnummer   */
    short noten[10];   /* Prüfungsnoten    */
};
```

Bei der obigen Angabe handelt es sich um die *Deklaration einer Struktur*. Aus diesem Beispiel ist der allgemeine Aufbau einer Strukturdeklaration erkennbar:

- ❑ Eine Strukturdeklaration beginnt immer mit dem Schlüsselwort `struct`. Nach Schlüsselwort `struct` kann der Name der Struktur (oben: `stu_daten`) angegeben werden.
- ❑ Eine Strukturdeklaration besteht aus einer Liste von Komponenten (Einzelvariablen, wie z. B. `name`, `vorname`, `postleit_zahl` usw.).
- ❑ Jede Komponente wird mit ihrem Datentyp und ihrem Namen deklariert.
- ❑ Die ganze Komponentenliste ist mit `{ ... }` zu klammern.
- ❑ Jede Strukturdeklaration ist mit Semikolon abzuschließen.

Für unseren Fall ist die 1. Bedingung und somit auch die Gesamtbedingung erfüllt, so dass folgende Anweisung ausgeführt wird:

```
knot_zeig->links = einordnen(knot_zeig->links);
```

Die Funktion `einordnen()` ruft sich also selbst wieder auf, so dass sich das in Abbildung 28.23 gezeigte Bild ergibt.

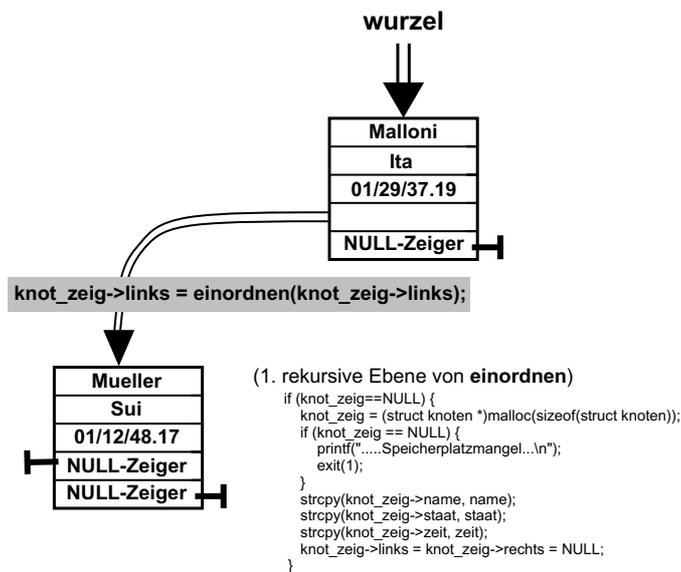


Abbildung 28.23: Einordnen von `Mueller` als linker Unterknoten von `Malloni`

Mit

```
return knot_zeig;
```

wird immer die Adresse des gerade behandelten Knotens an die übergeordnete Ebene zurückgegeben. Von der ersten rekursiven Ebene wird hier z. B. die Adresse des `Mueller`-Knotens zurückgegeben. Diese Adresse wird wegen

```
knot_zeig->links = einordnen(knot_zeig->links);
```

im `Malloni`-Knoten unter der Komponente `knot_zeig->links` (hier `wurzel->links`) gespeichert. Nach der Rückkehr nach `main()` wird dort unter Zuhilfenahme der Funktion `drucke_baum()` folgender Zwischenstand ausgegeben:

```
Zwischenstand nach 2 Läufer
```

```
=====
```

| Platz | Name | Land | Zeit |
|-------|---------|------|-------------|
| 1 | Mueller | Sui | 01/12/48.17 |
| 2 | Malloni | Ita | 01/29/37.19 |

3. eintreffender Läufer In der `while`-Schleife von `main()` werden nun die Daten des nächsten eintreffenden Läufers eingelesen:

```

name:  Hanson
staat: Swe
zeit:  01/20/10.33

```

Danach wird mit `wurzel = einordnen(wurzel);` wieder die Funktion `einordnen()` aufgerufen. In dieser Funktion wird nach Überprüfung der entsprechenden Bedingungen die Anweisung `knot_zeig->links = einordnen(knot_zeig->links);` ausgeführt. Bei diesem rekursiven Aufruf von `einordnen()` wird nach Überprüfung der vorgegebenen Bedingungen die Anweisung `knot_zeig->rechts = einordnen(knot_zeig->rechts);` ausgeführt, so dass sich das in Abbildung 28.24 gezeigte Bild ergibt.

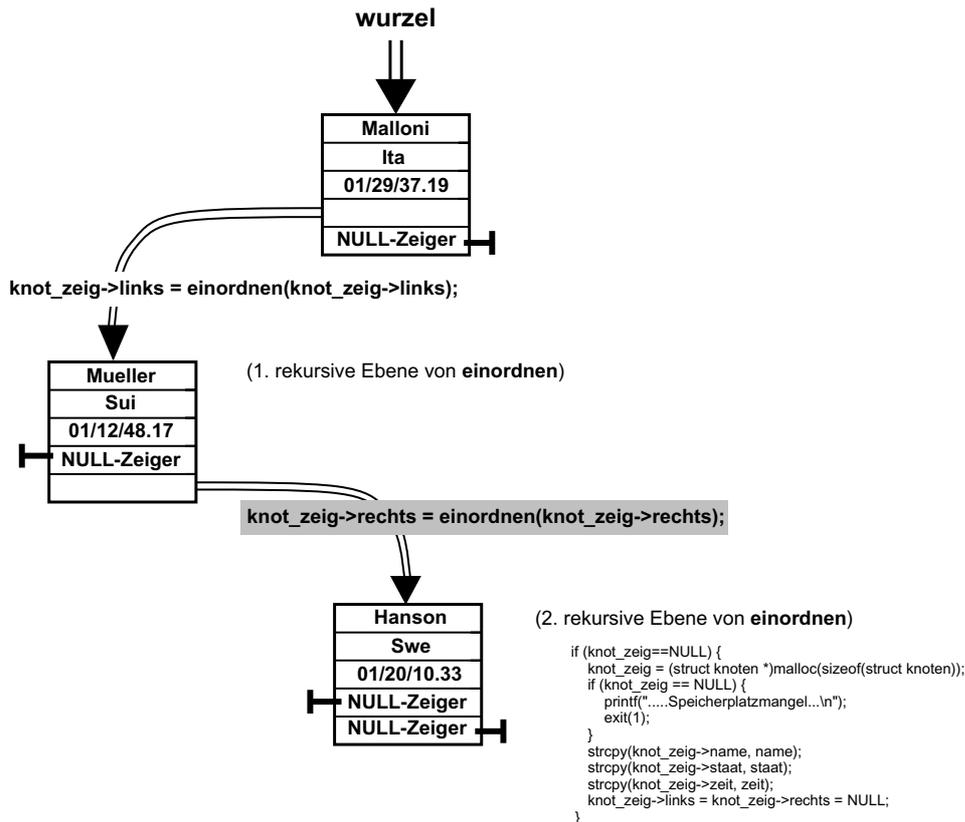


Abbildung 28.24: Einordnen von Hanson als rechter Unterknoten von Mueller

Nach der Rückkehr nach `main()` wird dort unter Zuhilfenahme der Funktion `drucke_baum()` folgender Zwischenstand ausgegeben:

```

Zwischenstand nach 3 Laeufer
=====
    Platz Name      Land  Zeit
    1  Mueller     Sui   01/12/48.17
    2  Hanson      Swe   01/20/10.33
    3  Malloni     Ita   01/29/37.19

```

4. eintreffender Läufer In der `while`-Schleife von `main()` werden nun die Daten des nächsten eintreffenden Läufers eingelesen:

```
name: Bichler
staat: Aut
zeit: 01/11/59.12
```

Danach wird mit `wurzel = einordnen(wurzel);` wieder die Funktion `einordnen()` aufgerufen. Der rekursive Ablauf dieser Funktion soll durch das in Abbildung 28.25 gezeigte Bild verdeutlicht werden.

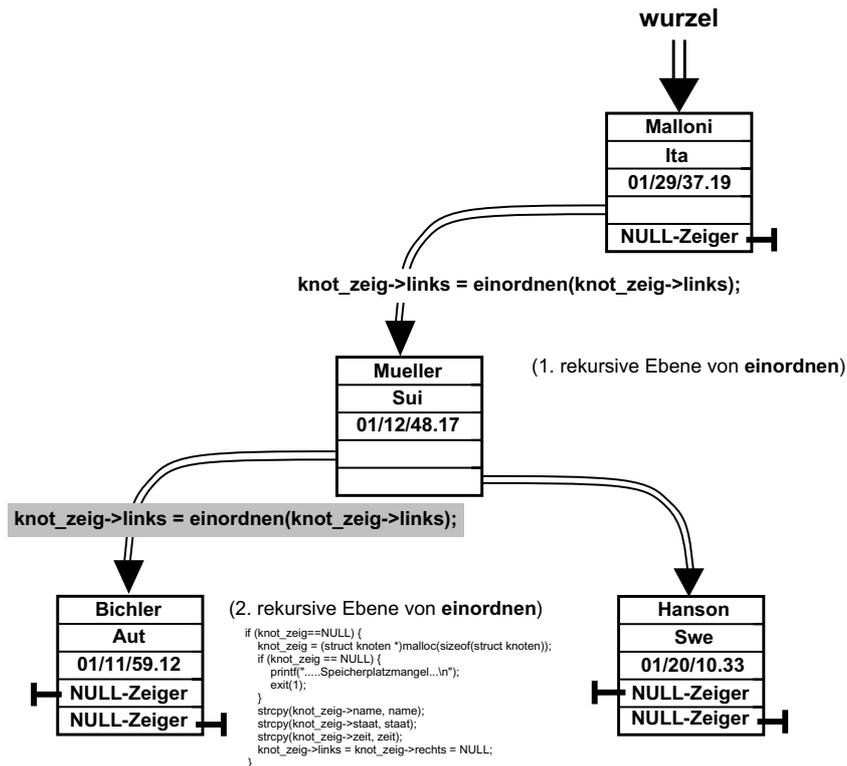


Abbildung 28.25: Einordnen von Bichler als linker Unterknoten von Mueller

Nach der Rückkehr nach `main()` wird dort unter Zuhilfenahme der Funktion `drucke_baum()` wieder der aktuelle Zwischenstand ausgegeben:

```
Zwischenstand nach 4 Laeufer
=====
    Platz Name      Land  Zeit
    ---  -
    1   Bichler     Aut   01/11/59.12
    2   Mueller     Sui   01/12/48.17
    3   Hanson      Swe   01/20/10.33
    4   Malloni     Ita   01/29/37.19
```

Hier können wir die Funktionsweise von `drucke_baum()` sehr schön erkennen: Es wird immer zuerst der vollständige linke Unterbaum ausgegeben, bevor ein Knoten selbst ausgegeben wird.

Kapitel 29

Eigene Datentypen

29.1 Definition eigener Datentypnamen mit typedef

29.1.1 Vergabe neuer Namen an existierende Datentypen mit typedef

Mit dem Schlüsselwort `typedef` können an Datentypen neue, vom Benutzer wählbare Namen vergeben werden, wie z. B.:

```
typedef unsigned int  UINT;
typedef unsigned long ULONG;
```

Mit diesen Angaben wird festgelegt, dass `UINT` ein Synonym für `unsigned int` und `ULONG` eines für `unsigned long` ist. Nach der obigen `typedef`-Angabe könnte man z. B. folgende Definitionen angeben:

```
UINT  zaehler, i;      /* entspricht: unsigned int  zaehler, i;      */
UINT  *zeigarray[10]; /* entspricht: unsigned int  *zeigarray[10]; */
ULONG a, *b;          /* entspricht: unsigned long a, *b;          */
```

Man könnte nun glauben, dass das Schlüsselwort `typedef` völlig überflüssig ist, da man den gleichen Effekt mit `#define` hätte erreichen können, wie z. B.:

```
#define UINT  unsigned int
#define ULONG unsigned long
```

Für diese einfachen Datentypdefinitionen ist dies zwar möglich, doch reicht bei komplexeren Datentypen der einfache Textersatz von `#define` nicht aus, wie z. B. bei:

```
#define STRING char *
```

In diesem Fall würde die folgende Definition:

```
STRING zgr1, zgr2;
```

nicht der erwarteten Definition

```
char *zgr1, *zgr2;
```

entsprechen, sondern der folgenden Definition:

```
char *zgr1, zgr2; /* zgr2 ist nicht - wie erwartet - ein Zeiger */
```

Kapitel 30

Dateien

Eine *Datei* (engl. *file*) ist letztendlich nichts anderes als eine Ansammlung von Daten, die auf einem externen Speicher dauerhaft aufbewahrt werden. Aus Sicht von C kann man sich eine Datei als ein permanentes, großes `char`-Array vorstellen, in das ein Programm Daten schreiben und aus dem ein Programm Daten lesen kann.

Anders als in anderen Programmiersprachen, ist eine C-Datei nicht irgendwie strukturiert, sondern nur eine Folge von Bytes, weswegen eine Datei in C oft auch als Datenstrom (*byte stream*) bezeichnet wird. Jedes einzelne dieser Zeichen (Bytes) kann über eine Positionsnummer in der Datei lokalisiert werden. So besitzt das erste Zeichen einer Datei die Positionsnummer 0, das zweite die Positionsnummer 1 usw.

Man kann in C grundsätzlich auf zwei Ebenen mit Dateien operieren, auf einer „unteren“ (*low level*) oder einer „oberen“ (*high level*).

Die untere Ebene ist die der so genannten *elementaren Dateizugriffe* (*Elementare Ein-/Ausgabe*), bei denen Funktionen verwendet werden, die direkt auf die entsprechenden Routinen des Betriebssystems (*system calls*) für Dateioperationen zurückgreifen. Diese elementaren Funktionen sind natürlich vom Betriebssystem abhängig und gehören nicht zum C-Standard, was bei der Portierung von Programmen von großer Bedeutung ist.

Die obere Ebene ist die der so genannten *höheren Dateizugriffe* (*Höhere Ein-/Ausgabe*), bei denen Funktionen verwendet werden, die zum einen wesentlich komfortabler als die elementaren sind und zum anderen auch dem C-Standard entsprechen und somit vom jeweiligen Betriebssystem unabhängig sind.

Nachfolgend werden nur die *Höheren E/A-Funktionen* vorgestellt.

30.1 Höhere E/A-Funktionen

In diesem Kapitel werden E/A-Funktionen beschrieben, die sich in der Standard-Bibliothek befinden und in der Headerdatei `<stdio.h>` definiert sind. Die höheren E/A-Funktionen arbeiten im Gegensatz zu den elementaren E/A-Funktionen mit eigenen optimal eingestellten Puffern, so dass sich der Aufrufer darum nicht selbst kümmern muss. Ein solcher Dateipuffer ist ein Bereich im Arbeitsspeicher, in dem eine bestimmte Menge von Daten, die aus einer Datei gelesen oder in eine Datei geschrieben werden sollen, vor ihrer Übertragung zum Zielort zwischengespeichert werden kann. Dies ist ein Vorteil, denn auf diese Weise muss nicht für jedes

Kapitel 31

Anhang

31.1 Prioritätstabelle für die Operatoren

| Operator (höchste Priorität oben; niedrigste unten) | Assoziativität |
|---|----------------|
| () [] -> . (Punktoperator) | von links her |
| ! ~ (Tilde) ++ -- + (Vorz.) - (Vorz.) sizeof & (Adreßop.) * (Zeigerzugriff) (casting) | von rechts her |
| * / % | von links her |
| + - | von links her |
| << >> | von links her |
| < <= > >= | von links her |
| == != | von links her |
| & (bitweise AND) | von links her |
| ^ | von links her |
| | von links her |
| && | von links her |
| | von links her |
| ?: (Bedingter Operator) | von rechts her |
| = += -= *= /= %= >>= <<= &= = ^= | von rechts her |
| , (Kommaoperator) | von links her |

31.2 C-Schlüsselwörter

In folgender Tabelle sind alle C-Schlüsselwörter angegeben, wobei die in C99 neu hinzugekommenen Schlüsselwörter fett gedruckt sind.

| | | | | | |
|---------------|-----------------|--------------|-----------------|-------------------|----------|
| auto | break | case | char | const | continue |
| default | do | double | else | enum | extern |
| float | for | goto | if | int | long |
| register | return | short | signed | sizeof | static |
| struct | switch | typedef | union | unsigned | void |
| volatile | while | | | | |
| inline | restrict | _Bool | _Complex | _Imaginary | |

Index

- < Operator, 23
- << Operator, 37
- <= Operator, 23
- == Operator, 23
- > Operator, 23
- >= Operator, 23
- >> Operator, 37
- | Operator, 25, 28
- || Operator, 26
- \\, 67, 615
- \', 67, 615
- \", 67, 615
- \a, 67, 615
- \b, 67, 615
- \e, 67, 615
- \f, 67, 615
- \ooo, 67, 615
- \r, 67, 615
- \t, 67, 615
- \v, 67, 615
- \x, 67, 615
- ++ Operator, 40, 43
- , Operator, 127, 133
- Operator, 40, 43
- <assert.h>, 339
- <ctype.h>, 60
- <errno.h>, 608
- <graphics.h>, 199
- <limits.h>, 101
- <math.h>, 86
- <stdarg.h>, 264
- <stdbool.h>, 24
- <stddef.h>, 72, 81
- <stdint.h>, 72, 81
- <stdio.h>, 577
- <stdlib.h>, 413
- <string.h>, 398
- <time.h>, 189
- ?, 119
- Übung
 - block3.c, 293
 - zusop2.c, 46
- # (Null-Direktive), 355
- # Operator, 345
- ## Operator, 346
- #define, 47, 49, 342
- #elif, 332
- #else, 332
- #endif, 332
- #error, 355
- #if, 332
- #ifdef, 332
- #ifndef, 332
- #include, 51, 340
- #line, 354
- #pragma, 355
- #undef, 349
- & Adresoperator, 260
- & Operator, 28
- && Operator, 25, 26
- _Bool Datentyp, 24
- _IOFBF Konstante, 603
- _IOLBF Konstante, 603
- _IONBF Konstante, 604
- __DATE__, 353
- __FILE__, 353
- __LINE__, 353
- __STDC__, 353
- __TIME__, 353
- ! Operator, 23
- ^ Operator, 28
- Bit-Operatoren, 28
- Bitfelder, 564
- Makro
 - __DATE__, 353
 - __FILE__, 353
 - __LINE__, 353
 - __STDC__, 353
 - __TIME__, 353
 - Vordefiniert, 353
- acos(), 86
- add2.c, 84
- add2alt.c, 244
- add2neu.c, 244
- addalt.c, 243
- addiere.c, 58
- addmult.c, 41
- addneu.c, 243
- Adresoperator &, 260
- Aktuelle Koordinate, 204
- Aktueller Parameter, 233, 253
- Aktuelles Argument, 233, 253
- AND-Operator, 26
- ANSI, 1
- Anweisung, 20
- arc(), 210
- argreih.c, 261
- argtest.c, 453
- argtest2.c, 454
- argtest3.c, 455
- Argument, 233, 253
- Argumente
 - Kommandozeile, 453
 - Optionen, 456
- argvor.c, 260
- arithmetische Operatoren, 19
- arr_swap.c, 438
- Array, 357
 - Funktionsadressen, 451
 - Initialisierung, 426
 - mehrdimensional, 365
 - Struktur, 507
 - und Zeiger, 374
 - zweidimensional, 365
- Array-Zeiger, 436
- arryfeh2.c, 364
- arrygros.c, 362
- arrynoname.c, 429
- arryok.c, 364
- arrysize.c, 373
- arrystatic.c, 440
- artizahl.c, 512
- asctime(), 189
- asin(), 86
- assert(), 339
- Assoziativität, 43
- atan(), 86
- atan2(), 86
- atof(), 418
- atoi(), 417
- atol(), 418
- atoll(), 418
- Ausdruck, 20
- Ausgabe
 - ein Zeichen, 53
 - gepuffert, 53
 - printf(), 67, 615
 - putchar(), 53
- Auswertung, 20
- Auswertungszeitpunkt, 43
- auto, 296
- autoadr.c, 303
- autoggt.c, 297
- autokauf.c, 107
- B Sprache, 1
- Balken
 - dreidimensional, 210
 - zweidimensional, 210
- bar(), 210
- bar3d(), 210
- Baum
 - Binär, 546
- BCPL Sprache, 1
- Bedingte Bewertung, 119
- Bedingte Kompilierung, 332
- Bibliothek, 51, 233
- bildschi.c, 323
- bildschi.h, 321
- Binärbaum, 546
- Binärmodus, 596

- binatext.c, 597
 - bitand.c, 30
 - bitfeld.c, 566
 - bitfeld2.c, 568
 - bitgleit.c, 34
 - bitor.c, 32
 - bitumdre.c, 336
 - bitxor.c, 33
 - Blöcke, 290
 - block.c, 291
 - block3.c, 293
 - Blockstruktur, 290
 - bonbon.c, 162
 - bonbon2.c, 163
 - bool Datentyp, 24
 - Bottom-up, 316
 - break, 179
 - Breite eines Textes, 215
 - briefmar.c, 150
 - Bubble-Sort, 389
 - bubble.c, 389
 - buchstat.c, 611
 - buchstnr.c, 358
 - buchzaeh.c, 396
 - BUFSIZ Konstante, 604
 - byte stream, 577
 - bytezahl.c, 583

 - C++, 1
 - C89, 1
 - C99, 1
 - Call
 - by reference, 258
 - by value, 258
 - calloc(), 474
 - Cast, 98
 - cast.c, 100
 - casting, 98
 - ceil(), 86
 - char Datentyp, 8
 - char-Konstanten, 11
 - char-Zeiger, 392
 - Chromsky-Grammatik, 277
 - circle(), 209
 - cleardevice(), 204
 - clearerr(), 581
 - clearviewport(), 223
 - clock(), 189
 - clock1.c, 185
 - closegraph(), 200
 - Compiler, 317
 - const, 49, 310, 434
 - const.c, 312
 - continue, 183
 - copyzeit.c, 593
 - cos(), 86
 - cosh(), 86
 - ctime(), 189

 - dataus.c, 584
 - datbytes.c, 600
 - Datei
 - Öffnen, 578
 - Binärmodus, 596
 - EOF-Flag, 581
 - Fehler-Flag, 581
 - Löschen, 603
 - Lesen (blockweise), 590
 - Lesen (ein Zeichen), 581, 582
 - Lesen (eine Zeile), 584
 - Lesen (formatiert), 585
 - Positionieren, 599, 601
 - Pufferung, 604
 - Schliessen, 578, 580
 - Schreiben (blockweise), 590
 - Schreiben (ein Zeichen), 581, 582
 - Schreiben (eine Zeile), 584
 - Schreiben (formatiert), 585, 594
 - Stream verknüpfen, 601
 - Temporäre, 605
 - Textmodus, 596
 - Umbenennen, 603
 - Zurückschieben (ein Zeichen), 589
- Datentyp, 9, 614
 - _Bool, 24
 - bool, 24
 - char, 8
 - double, 8
 - FILE, 578
 - float, 8
 - fpos_t, 599
 - Grund-, 7
 - int, 8
 - long, 8
 - long long, 8
 - short, 8
 - signed, 8
 - Tabelle, 9, 614
 - unsigned, 8
 - va_list, 264
 - void, 238, 253
 - Wertebereich, 9, 614
 - Datentypen
 - und Operatoren, 43
 - Datentypumwandlung, 93, 98, 246
 - casting, 98
 - explizit, 98
 - implizit, 93, 246
 - implizite, 91
 - Datum des Tages, 193
 - datuminfo.c, 194
 - Datumsangaben, 189
 - defined, 332
 - Definition
 - Funktionen (Alt-C), 237
 - Funktionen (C89/C99), 234
 - Konstanten, 47, 49
 - Variablen, 14
 - Dekrement-Operator --, 40
 - delay(), 201
 - dezimale Konstanten, 11
 - dezumwa2.c, 379
 - dezumwan.c, 359
 - difftime(), 189
 - Directory
 - Löschen, 603
 - do...while, 175
 - Doppelt verkettete Liste, 541
 - double Datentyp, 8
 - drawpoly(), 210
 - dreichi1.c, 446
 - dreichi2.c, 450
 - dreizeic.c, 54
 - dualbcd.c, 431
 - dualzahl.c, 142
 - Dynamische
 - Speicherallozierung, 463
 - Strukturarrays, 522
 - dynarr.c, 435

 - E/A-Funktionen, 577
 - E/A-Umlenkung, 586
 - eifrau1.c, 182
 - eifrau2.c, 182
 - eimalei.c, 137
 - einaus.c, 202
 - einauscp.c, 586
 - Einer-Komplement, 28
 - Eingabe
 - ein Zeichen, 53
 - gepuffert, 53
 - getchar(), 53
 - scanf(), 77, 617
 - einkompl.c, 29
 - einles.c, 293
 - Einseitige
 - if-Anweisung, 110
 - einzeich.c, 53
 - Ellipse
 - ausgefüllt zeichnen, 209, 220
 - zeichnen, 209
 - ellipse(), 209
 - Ellipsen-Prototypen, 263
 - endlos.c, 581
 - Endlose for-Schleife, 143
 - englzah1.c, 442
 - englzah2.c, 443
 - Entwurf
 - Bottom-up, 316
 - Modularer, 316
 - Top-Down, 316
 - enum, 573
 - enumdemo.c, 576
 - EOF-Flag, 581
 - erno Variable, 608
 - errodemo.c, 609
 - erst.c, 3
 - exp(), 86
 - Explizite Datentypumwandlung, 98
 - extern, 293

 - Füllmuster, 209
 - fabs(), 86
 - fakul.c, 273
 - fakul2.c, 273
 - FALSE, 24
 - false Konstante, 24
 - Farbpalette, 222
 - farbstuf.c, 222
 - fclose(), 580
 - fehlausg.c, 269
 - Fehler-Flag, 581
 - fehler.c, 325, 610
 - fehler.h, 322
 - Fehlermeldung, 608
 - fehllhand.c, 609
 - feof(), 581
 - ferror(), 581
 - fflush(), 605
 - fgetc(), 582
 - fgetpos(), 601
 - fgets(), 584
 - fibonaci.c, 276
 - figur.c, 211
 - File
 - siehe Datei, 578
 - FILE Datentyp, 578
 - fillellipse(), 209
 - fillpoly(), 210
 - firmaelt.c, 498, 502
 - firmver2.c, 522
 - firmverw.c, 508
 - float Datentyp, 8
 - floataus.c, 22
 - floatint.c, 97
 - floor(), 86
 - FLT_ROUND, 102
 - fmod(), 86
 - Font, 215

- fopen(), 578
 for, 129
 endlos, 143
 fordrueb.c, 159
 Formaler Parameter, 233, 253
 fprintf(), 585
 fputc(), 582
 fputs(), 584
 Fraktale, 277
 fread(), 590
 free(), 481
 free1.c, 481
 free2.c, 482
 free3.c, 483
 freeimage(), 217
 Freigabe
 Speicher, 463
 fremdspe.c, 398
 freopen(), 601
 freopen.c, 602
 frexp(), 86
 fscanf(), 585
 fseek(), 599
 fsetpos(), 601
 ftell(), 599
 funcbez.c, 271
 functions, 231
 funkproj.c, 227
 Funktion, 231
 acos(), 86
 arc(), 210
 asctime(), 189
 asin(), 86
 atan(), 86
 atan2(), 86
 atof(), 418
 atoi(), 417
 atol(), 418
 atoll(), 418
 bar(), 210
 bar3d(), 210
 calloc(), 474
 ceil(), 86
 circle(), 209
 cleardevice(), 204
 clearerr(), 581
 clearviewport(), 223
 clock(), 189
 closegraph(), 200
 cos(), 86
 cosh(), 86
 ctime(), 189
 Definition (Alt-C), 237
 Definition (C89/C99), 234
 Deklaration, 241
 delay(), 201
 difftime(), 189
 drawpoly(), 210
 ellipse(), 209
 Ellipsen-Prototypen, 263
 exp(), 86
 fabs(), 86
 fclose(), 580
 feof(), 581
 ferror(), 581
 fflush(), 605
 fgetc(), 582
 fgetpos(), 601
 fgets(), 584
 fillemnipse(), 209
 fillpoly(), 210
 floor(), 86
 fmod(), 86
 fopen(), 578
 fprintf(), 585
 fputs(), 584
 fread(), 590
 free(), 481
 freeimage(), 217
 freopen(), 601
 frexp(), 86
 fscanf(), 585
 fseek(), 599
 fsetpos(), 601
 ftell(), 599
 fwrite(), 590
 getc(), 582
 getch(), 200
 getchar(), 581
 getcharacter(), 200
 getcolor(), 207
 getdouble(), 200
 getimage(), 217
 getint(), 200
 getmaxcolor(), 204
 getmaxx(), 204
 getmaxy(), 204
 getpixel(), 204
 gets, 584
 gets(), 422
 getstring(), 200
 getx(), 204
 gety(), 204
 gmtime(), 190
 initgraph(), 200
 isalnum(), 60
 isalpha(), 60
 isblank(), 60
 iscntrl(), 60
 isdigit(), 60
 isgraph(), 60
 islower(), 60
 isprint(), 60
 ispunct(), 60
 isspace(), 60
 isupper(), 60
 isxdigit(), 60
 kbhit(), 200
 ldexp(), 86
 line(), 207
 linere(), 207
 lineto(), 207
 loadimage(), 217
 localtime(), 190
 log(), 86
 log10(), 86
 malloc(), 466
 memchr(), 413
 memcmp(), 412
 memcpy(), 411
 memmove(), 412
 memset(), 412
 mktime(), 190
 modf(), 86
 mouse_button(), 223
 mouse_getpos(), 223
 mouse_hide(), 223
 mouse_inwindow(), 223
 mouse_left(), 223
 mouse_mid(), 223
 mouse_right(), 223
 mouse_setcursor(), 224
 mouse_setpos(), 223
 mouse_setwindow(), 223
 mouse_show(), 223
 mouse_visible(), 223
 moverel(), 207
 moveto(), 207
 outtextxy(), 201
 perror(), 608
 pieslice(), 220
 pow(), 86
 printf(), 585
 Prototypen, 242
 putchar(), 581
 putimage(), 217
 putpixel(), 204
 puts(), 422, 584
 qsort(), 390, 525
 realloc(), 475
 rectangle(), 209
 Rekursiv, 273
 remove(), 603
 rename(), 603
 rewind(), 601
 scanf(), 421, 585
 sector(), 220
 setbuf(), 604
 setcolor(), 207
 setfillstyle(), 209
 setlinestyle(), 207
 setpalette(), 222
 setrgbpalette(), 222
 settextjustify(), 215
 settextstyle(), 215
 setvbuf(), 604
 setviewport(), 223
 setwritemode(), 222
 sin(), 86
 sinh(), 86
 snprintf(), 595
 sprintf(), 419, 595
 sqrt(), 86
 sscanf(), 418, 595
 strcat(), 401
 strchr(), 403
 strcmp(), 402
 strcoll(), 410
 strcpy(), 395, 400
 strcspn(), 408
 strerror(), 410, 608
 strftime(), 190
 strlen(), 403
 strncat(), 401
 strncpy(), 400
 strpbrk(), 406
 strrchr(), 404
 strspn(), 408
 strstr(), 405
 strtod(), 416
 strtof(), 417
 strtol(), 413
 strtold(), 417
 strtoll(), 416
 strtoul(), 416
 strtoull(), 416
 Struktur, 514
 strxfrm(), 411
 tan(), 86
 tanh(), 86
 textheight(), 215
 textwidth(), 215
 time(), 191
 tmpfile(), 606
 tmpnam(), 606
 tolower(), 60
 toupper(), 60
 ungetc(), 589
 vscanf(), 596

- vprintf(), 594
- vscanf(), 596
- vsprintf(), 596
- vsprintf(), 596
- vsscanf(), 596
- Adresse, 451
- assert(), 339
- inline, 270
- va_arg() Makro, 264
- va_end() Makro, 264
- va_start() Makro, 264
- Zeiger auf, 280
- funkzgr1.c, 284
- funkzgr2.c, 285
- fwrite(), 590

- Ganzzahlige Konstanten, 11, 24
- Ganzzahltypen
 - Grenzwerte, 101
- geheim.c, 307
- georeih2.c, 134
- georeih3.c, 135
- georeih4.c, 154
- georeih5.c, 155
- georeihe.c, 132
- gepufferte Ausgabe, 53
- gepufferte Eingabe, 53
- gerade.c, 351
- Geschachtelte Schleifen, 137
- getc(), 582
- getch(), 200
- getchar(), 53, 581
- getcharacter(), 200
- getcolor(), 207
- getdouble(), 200
- getimage(), 217
- getint(), 200
- getmaxcolor(), 204
- getmaxx(), 204
- getmaxy(), 204
- getpixel(), 204
- gets(), 422, 584
- getstring(), 200
- getx(), 204
- gety(), 204
- ggtkgv.c, 177
- Gleichheitsoperatoren, 23
- gleitpkt.c, 562
- Gleitpunkt-Ausdruck, 21
- Gleitpunkt-Variable, 21
- Gleitpunktkonstanten, 12
- gleizae2.c, 158
- gleizae3.c, 158
- gleizae4.c, 158
- gleizaeh.c, 157
- gmtime(), 190
- goldbac1.c, 186
- goldbac2.c, 187
- goldbac3.c, 188
- Goldbach-Vermutung, 186, 188
- goto, 197
- Größe von Text, 215
- Grafikmodus
 - ausschalten, 200
 - einschalten, 200
- Grenzwerte
 - Ganzzahltypen, 101
- groesse.c, 215
- Grunddatentypen, 7
- guelt1.c, 300
- guelt2.c, 301
- guelt3.c, 302

- Höhe eines Textes, 215
- hamming1.c, 156
- hamming2.c, 156
- haus.c, 130
- Headerdatei, 340
 - <assert.h>, 339
 - <ctype.h>, 60
 - <errno.h>, 608
 - <graphics.h>, 199
 - <limits.h>, 101
 - <math.h>, 86
 - <stdarg.h>, 264
 - <stdbool.h>, 24
 - <stddef.h>, 72, 81
 - <stdint.h>, 72, 81
 - <stdio.h>, 577
 - <stdlib.h>, 413
 - <string.h>, 398
 - <time.h>, 189
 - selbst.h, 63, 65
- Headerdateien, 51, 233
- Hebräische Methode, 371
- hebrae.c, 371
- heron.c, 164
- heron2.c, 164
- heron3.c, 164
- Hexadezimal
 - Konstanten, 11
 - Ziffer, 122, 124
- hexd.c, 591
- hexextra.c, 589
- hexokdua.c, 139
- hexziff1.c, 122
- hexziff2.c, 124
- hintquad.c, 154
- hochdrei.c, 350

- if, 105, 110
- Implizite Datentypumwandlung, 93, 246
- implizite Datentypumwandlung, 91
- Information Hidding, 314
- inconst.c, 434
- initgraph(), 200
- Initialisieren Variable, 46
- Initialisierung, 19, 426
 - Struktur, 504
- inkrdekr.c, 41
- Inkrement-Operator ++, 40
- Inline-Funktion, 270
- inline.c, 270
- int Datentyp, 8
- int-Ausdruck, 21
- int-Variable, 21
- intaus.c, 21
- invest.c, 366
- isalnum(), 60
- isalpha(), 60
- isblank(), 60
- isctrl(), 60
- isdigit(), 60
- isgraph(), 60
- islower(), 60
- isnrlow.c, 408
- ISO, 1
- ISO C99, 1
- isprint(), 60
- ispunct(), 60
- isspace(), 60
- isupper(), 60
- isxdigit(), 60

- jaeger1.c, 172
- jaeger2.c, 173

- just.c, 216

- karten.c, 219
- kbhit(), 200
- keinadr.c, 83
- kleigro1.c, 61
- kleigro2.c, 61
- kleigro3.c, 64
- kleigros.c, 166
- Komma-Operator, 127, 133
- Kommandozeile, 453, 456
 - Argumente, 453
 - Optionen, 456
- Kommentar, 5
- Geschachtelt, 6
- Komplement
 - Einer, 28
 - Zweier, 29
- kompzahl.c, 514
- Konkatenation von Strings, 75
- Konstante
 - _IOFBF, 603
 - _IOLBF, 603
 - _IONBF, 604
 - false, 24
 - L_tmpnam, 606
 - SEEK_CUR, 599
 - SEEK_END, 599
 - SEEK_SET, 599
 - stderr, 580, 586
 - stdin, 580, 586
 - stdout, 580, 586
 - TMP_MAX, 606
 - true, 24
- Konstanten, 11, 12, 47, 49
 - char, 11
 - Dezimal, 11
 - Ganzzahlig, 11, 24
 - Gleitpunkt, 12
 - Hexadezimal, 11
 - Oktal, 11
- konto.c, 56
- konto2.c, 57
- Koordinate
 - aktuell, 204
 - Maximales x, 204
 - Maximales y, 204
 - Transformation, 226
- Kreis
 - ausgefüllt zeichnen, 220
 - zeichnen, 209
- kreis.c, 48
- Kreisbogen
 - zeichnen, 210
- kubikzah.c, 386
- Kuchenstück, 220
- kugzyvol.c, 280

- L_tmpnam Konstante, 606
- ldexp(), 86
- Lebensdauer, 291
- leerpara.c, 253
- Lesbarkeit, 113
- Lindenmayer-Systeme, 277
- line(), 207
- linerel(), 207
- lineto(), 207
- Linie
 - absolut zeichnen, 207
 - Art, 207
 - Dicke, 207
 - relativ zeichnen, 207
 - zeichnen, 207

- linie.c, 207
- Linked List, 529, 534, 541
- Linker, 317
- Liste
 - Ausgeben, 534
 - Doppelt verkettete, 541
 - Einfügen, 534
 - Löschen, 534
 - Linked, 529, 534, 541
 - Operationen, 534
 - Sortierte, 534
 - Verkettete, 529
- loadimage(), 217
- localtime(), 190
- log(), 86
- log10(), 86
- Logische Operatoren, 24
- Logischer Operatoren, 25
- long Datentyp, 8
- long long Datentyp, 8
- lottoza2.c, 152
- lottoza3.c, 153
- lottozah.c, 151
- lshift.c, 37
- lsystem2.c, 278

- Makro, 62, 342
- Makroaufruf, 62
- malloc(), 466
- manhat2.c, 147
- manhatan.c, 144
- mannweib.c, 119
- Marke, 197
- matadd.c, 368
- matadd2.c, 387
- mathverg.c, 88
- Matrix, 365
- Matrizen, 365
- mauscursor.c, 225
- mausdemo.c, 224
- Mausprogrammierung, 223
- max.c, 344
- max2zahl.c, 344
- Mehrdimensionale Arrays, 365
- memchr(), 413
- memcmp(), 412
- memcpy(), 411
- memcpy.c, 411
- memmove(), 412
- memmove.c, 412
- memset(), 412
- menue1.c, 117
- menue2.c, 118
- mfunk1.c, 87
- mineins.c, 116
- mineins2.c, 116
- mittwert.c, 142
- mktime(), 190
- modf(), 86
- Modul, 287, 314
 - bitumdre.c, 336
 - fehler.c, 325
 - paritaet.c, 326
 - pating.c, 324
 - zahlwort.c, 339
- Modularer Entwurf, 316
- Modultechnik, 314
- monataus.c, 432
- mouse_button(), 223
- mouse_getpos(), 223
- mouse_hide(), 223
- mouse_inwindow(), 223
- mouse_left(), 223
- mouse_mid(), 223
- mouse_right(), 223
- mouse_setcursor(), 224
- mouse_setpos(), 223
- mouse_setwindow(), 223
- mouse_show(), 223
- mouse_visible(), 223
- moverel(), 207
- moveto(), 207
- mrz_dez.c, 98
- mrz_dez2.c, 99
- Muster, 209

- nachvor.c, 55
- namlist1.c, 529
- namlist2.c, 535
- namlist3.c, 541
- namsort1.c, 463
- namsort2.c, 467
- namsort3.c, 468
- namsort4.c, 485
- namsort5.c, 486
- namsort6.c, 487
- namsort7.c, 487
- namsort8.c, 488
- namsort9.c, 488
- Nassi-Shneiderman-Diagramm, 106, 110, 121, 129, 161, 175
- NDEBUG, 339
- Negations-Operator, 25
- nreingab.c, 307
- Null-Direktive #, 355
- numsort1.c, 470
- numsort2.c, 475
- numsort3.c, 479
- numsort4.c, 484

- oel.c, 205
- oelbohr.c, 171
- Oktal
 - Konstanten, 11
- Operation, 128
- Operationen mit Zeigern, 381
- Operationen und Datentypen, 43
- Operator
 - <<, 37
 - >>, 37
 - >, 517
 - |, 28, 31
 - ||, 26
 - ++, 40
 - , 40
 - #, 345
 - ##, 346
 - &, 28, 30
 - &&, 26
 - !, 25
 - ^, 28, 33
 - Bit, 28
 - |, 32
 - ++, 43
 - , 43
 - Assoziativität, 43
 - Auswertungszeitupnt, 43
 - cast-, 98
 - Dekrement, 40
 - Inkrement, 40
 - Komma, 127, 133
 - Postfix, 40
 - Präfix, 40
 - Priorität, 42, 128, 613
 - sizeof, 91, 373
 - und Datentypen, 43
- AND, 26
- arithmetisch, 19
- Auswertung, 20
- Logischer, 25
- Negations-, 25
- OR, 26
- Pfeil-, 517
- Priorität, 20, 35, 38
- Punkt, 496
- relationaler, 23
- Shift, 37
- Tilde, 28
- Vergleichs-, 23
- Zuweisung, 17
- Zuweisungs-, 39
- Operatoren, 19, 23, 25, 26, 28, 37, 39, 40, 42, 43
 - Gleichheits-, 23
 - Relationale, 23
- opshift.c, 39
- OR-Operator, 26
- outtextxy(), 201

- Parameter, 233, 253
 - Struktur, 514
- paritaet.c, 326
- pating.c, 324
- perror(), 608
- pfeilfix.c, 518
- Pfeiloperator, 517
- piesarc.c, 221
- pieslice(), 220
- Pixel, 204
- pixel.c, 206
- Polygon
 - ausgefüllt zeichnen, 210
 - zeichnen, 210
- polygon.c, 213
- Positionieren
 - absolut, 207
 - relativ, 207
- Postfix-Operator, 40
- postprae.c, 44
- potenz.c, 235, 237
- potenz3.c, 239
- potenz4.c, 241
- pow(), 86
- Präfix-Operator, 40
- printf(), 67, 585, 615
- printf1.c, 69
- printf2.c, 71
- printf3.c, 72
- printf4.c, 73
- printf5.c, 74
- Priorität, 20, 35, 38, 42, 128, 613
- Privatisierungseffekt, 290
- Programm
 - add2.c, 84
 - add2alt.c, 244
 - add2neu.c, 244
 - addalt.c, 243
 - addiere.c, 58
 - addmult.c, 41
 - addneu.c, 243
 - argreih.c, 261
 - argtest.c, 453
 - argtest2.c, 454
 - argtest3.c, 455
 - argvor.c, 260
 - arr_swap.c, 438
 - arryfeh2.c, 364
 - arryfeh1.c, 363
 - arrygros.c, 362

- arrayname.c, 429
arryok.c, 364
arraysize.c, 373
arraystatic.c, 440
artizahl.c, 512
autoadr.c, 303
autoggt.c, 297
autokauf.c, 107
binatext.c, 597
bitand.c, 30
bitfeld.c, 566
bitfeld2.c, 568
bitgleit.c, 34
bitor.c, 32
bitumdre.c, 336
bitxor.c, 33
block.c, 291
block3.c, 293
bonbon.c, 162
bonbon2.c, 163
briefmar.c, 150
bubble.c, 389
buchstat.c, 611
buchstnr.c, 358
buchzaeh.c, 396
bytezahl.c, 583
cast.c, 100
clock1.c, 185
const.c, 312
copyzeit.c, 593
dataus.c, 584
datbytes.c, 600
datuminfo.c, 194
dezumwa2.c, 379
dezumwan.c, 359
dreichi1.c, 446
dreichi2.c, 450
dreizeic.c, 54
dualbcd.c, 431
dualzahl.c, 142
dynarr.c, 435
eifrau1.c, 182
eifrau2.c, 182
eimalei.c, 137
einaus.c, 202
einauscp.c, 586
einkompl.c, 29
einles.c, 293
einzeich.c, 53
endlos.c, 581
englzh1.c, 442
englzh2.c, 443
enumdemo.c, 576
errodemo.c, 609
erst.c, 3
fakul.c, 273
fakul2.c, 273
farbstuf.c, 222
fehlausg.c, 269
fehler.c, 610
fehlhand.c, 609
fibonaci.c, 276
figur.c, 211
firmaelt.c, 498, 502
firmver2.c, 522
firmverw.c, 508
floataus.c, 22
floatint.c, 97
fordrueb.c, 159
free1.c, 481
free2.c, 482
free3.c, 483
fremdspe.c, 398
freopen.c, 602
funcbez.c, 271
funkproj.c, 227
funkzgr1.c, 284
funkzgr2.c, 285
geheim.c, 307
georeih2.c, 134
georeih3.c, 135
georeih4.c, 154
georeih5.c, 155
georeihe.c, 132
gerade.c, 351
ggtkgv.c, 177
gleitpkt.c, 562
gleizae2.c, 158
gleizae3.c, 158
gleizae4.c, 158
gleizae5.c, 157
goldbac1.c, 186
goldbac2.c, 187
goldbac3.c, 188
groesse.c, 215
guelt1.c, 300
guelt2.c, 301
guelt3.c, 302
hamming1.c, 156
hamming2.c, 156
haus.c, 130
hebrae.c, 371
heron.c, 164
heron2.c, 164
heron3.c, 164
hexd.c, 591
hexextra.c, 589
hexokdua.c, 139
hexziff1.c, 122
hexziff2.c, 124
hintquad.c, 154
hochdrei.c, 350
inline.c, 270
iniconst.c, 434
inkrdekr.c, 41
intaus.c, 21
invest.c, 366
isnrlow.c, 408
jaeger1.c, 172
jaeger2.c, 173
just.c, 216
karten.c, 219
keinadr.c, 83
kleigro1.c, 61
kleigro2.c, 61
kleigro3.c, 64
kleigros.c, 166
kompzahl.c, 514
konto.c, 56
konto2.c, 57
kreis.c, 48
kubikzah.c, 386
kugzyvol.c, 280
leerpara.c, 253
linie.c, 207
lottoza2.c, 152
lottoza3.c, 153
lottozah.c, 151
lshift.c, 37
lssystem2.c, 278
manhat2.c, 147
manhatan.c, 144
mannweib.c, 119
matadd.c, 368
matadd2.c, 387
mathverg.c, 88
mauscursor.c, 225
mausdemo.c, 224
max.c, 344
max2zahl.c, 344
memcpy.c, 411
memmove.c, 412
menue1.c, 117
menue2.c, 118
mfunk1.c, 87
mineins.c, 116
mineins2.c, 116
mittwert.c, 142
monataus.c, 432
mrz_dez.c, 98
mrz_dez2.c, 99
nachvor.c, 55
namlist1.c, 529
namlist2.c, 535
namlist3.c, 541
namsort1.c, 463
namsort2.c, 467
namsort3.c, 468
namsort4.c, 485
namsort5.c, 486
namsort6.c, 487
namsort7.c, 487
namsort8.c, 488
namsort9.c, 488
nreingab.c, 307
numsort1.c, 470
numsort2.c, 475
numsort3.c, 479
numsort4.c, 484
oel.c, 205
oelbohr.c, 171
ophift.c, 39
pfeifix.c, 518
piesarc.c, 221
pixel.c, 206
polygon.c, 213
postprae.c, 44
potenz.c, 235, 237
potenz3.c, 239
potenz4.c, 241
printf1.c, 69
printf2.c, 71
printf3.c, 72
printf4.c, 73
printf5.c, 74
qsort1.c, 391
quader.c, 64
quadrat.c, 149
quadrat2.c, 150
rabatt.c, 253
rabatt2.c, 255
reaktion.c, 203
realloc.c, 477
rechne1.c, 451
rechne2.c, 452
regel1.c, 246
regel1a.c, 93, 246
regel1b.c, 94
regel1c.c, 94
regel5.c, 96
romzahl.c, 249
rueckwae.c, 274
sandmann.c, 218
scanf1.c, 78
scanf2.c, 79
scanf3.c, 80
scanf4.c, 82
scanf5.c, 85
skatkart.c, 574

- skilang.c, 548
- sort1.c, 382
- spiel21.c, 397
- spieltor.c, 304
- sprintf.c, 419
- sscanf.c, 418, 595
- static1.c, 309
- static2.c, 309
- strcat.c, 401
- strchr.c, 404
- strcmp.c, 402
- strcpy.c, 394
- strcpy.c, 395
- strein1.c, 421
- strein2.c, 421
- strein3.c, 422
- strein4.c, 422
- striche.c, 238
- strlen.c, 403
- strncmp(), 402
- strncpy.c, 400
- strrchr.c, 404
- strtok(), 409
- strtok.c, 409
- strtol.c, 414
- structname.c, 506
- struoper.c, 503
- struvararr.c, 558
- struzgr1.c, 516
- struzgr2.c, 518
- struzgr3.c, 521
- suchtext.c, 405
- tabelle.c, 480
- tausch.c, 18, 256
- tausch2.c, 258
- tausch3.c, 287
- textaus1.c, 75
- textaus2.c, 75
- tmpnam.c, 606
- typedef1.c, 570
- typgroes.c, 91
- union.c, 560
- unturing.c, 329
- va_args.c, 348
- vchiffre.c, 250
- vertausc.c, 111
- vfprintf(), 594
- vieladd1.c, 265
- vieladd2.c, 266
- vieladd3.c, 267
- vokzaeh2.c, 425
- vokzaeh1.c, 407
- vollkom2.c, 184
- vollkomm.c, 183
- vormakro.c, 354
- welchdat.c, 193
- welchtag.c, 192
- wurzstruct.c, 506
- wurzzahl.c, 432
- wz.c, 456
- zahltab.c, 145
- zahltab2.c, 146
- zgarrry1.c, 424
- zgarrry2.c, 424
- zins.c, 262
- zufzahl2.c, 169
- zufzahl3.c, 170
- zufzahl4.c, 170
- zusop2.c, 46
- zweit.c, 4
- Programmablaufplan
 - do...while, 175
 - for, 129
 - if (einsetig), 110
 - if (zweiseitig), 106
 - while, 161
- programmglobal, 289
- Projektion von Koordinaten, 226
- Prototypen, 242
- Prozeduren, 238, 239
- Puffer
 - leeren, 605
- Pufferung, 603
 - keine, 604
 - Voll-, 603
 - voreingestellt, 604
- Zeilen-, 603
- Punkt Operator, 496
- putc(), 582
- putchar(), 581
- putimage(), 217
- putpixel(), 204
- puts(), 422, 584
- qsort(), 390, 525
- qsort1.c, 391
- quader.c, 64
- quadrat.c, 149
- quadrat2.c, 150
- Römische Zahlen, 249
- rabatt.c, 253
- rabatt2.c, 255
- reaktion.c, 203
- realloc(), 475
- realloc.c, 477
- rechne1.c, 451
- rechne2.c, 452
- Rechteck
 - ausgefüllt zeichnen, 210
 - zeichnen, 209
- rectangle(), 209
- regel1.c, 246
- regel1a.c, 93, 246
- regel1b.c, 94
- regel1c.c, 94
- regel5.c, 96
- register, 310
- Rekursive Funktion, 273
- Rekursive Strukturen, 529
- Relationale Operatoren, 23
- Relativ Positionieren, 207
- remove(), 603
- rename(), 603
- Reservierung
 - Speicher, 463
- restrict, 394
- return, 238
- rewind(), 601
- romzahl.c, 249
- rueckwae.c, 274
- sandmann.c, 218
- scanf(), 77, 421, 585, 617
- scanf1.c, 78
- scanf2.c, 79
- scanf3.c, 80
- scanf4.c, 82
- scanf5.c, 85
- Schnittstellen, 245
- Schriftart, 215
- Schrittweise Verfeinerung, 316
- sector(), 220
- SEEK_CUR Konstante, 599
- SEEK_END Konstante, 599
- SEEK_SET Konstante, 599
- selbst.h, 63, 65
- self-typing, 11
- setbuf(), 604
- setcolor(), 207
- setfillstyle(), 209
- setlinestyle(), 207
- setpalette(), 222
- setrgbpalette(), 222
- settextjustify(), 215
- settextstyle(), 215
- setvbuf(), 604
- setviewport(), 223
- setwritemode(), 222
- Shift-Operator, 37
- short Datentyp, 8
- signed Datentyp, 8
- sin(), 86
- sinh(), 86
- sizeof, 91, 373
- skatkart.c, 574
- skilang.c, 548
- snprintf(), 595
- sort1.c, 382
- Sortier-Algorithmus, 389
- Sortierte
 - Liste, 534
- Speicher
 - Dynamische Freigabe, 463
 - Dynamische Reservierung, 463
- Speicherort, 292
- spiel21.c, 397
- spieltor.c, 304
- sprintf(), 419, 595
- sprintf.c, 419
- sqrt(), 86
- sscanf(), 418, 595
- sscanf.c, 418, 595
- Stack, 254, 273, 292
- Standard-E/A-Funktionen, 577
- Standard-Headerdateien, 341
- Standardausgabe, 580, 586
- Standardbibliothek, 51, 233
- Standardeingabe, 580, 586
- Standardfehlerausgabe, 580, 586
- static, 304
- static1.c, 309
- static2.c, 309
- stderr Konstante, 580, 586
- stdin Konstante, 580, 586
- stdout Konstante, 580
- stdout Kontante, 586
- Steuerzeichen
 - \\, 67, 615
 - \/, 67, 615
 - \", 67, 615
 - \a, 67, 615
 - \b, 67, 615
 - \e, 67, 615
 - \f, 67, 615
 - \ooo, 67, 615
 - \r, 67, 615
 - \t, 67, 615
 - \v, 67, 615
 - \x, 67, 615
- strcat(), 401
- strcat.c, 401
- strchr(), 403
- strchr.c, 404
- strcmp(), 402
- strcmp.c, 402
- strcoll(), 410
- strcpy(), 395, 400
- strcpy.c, 394

