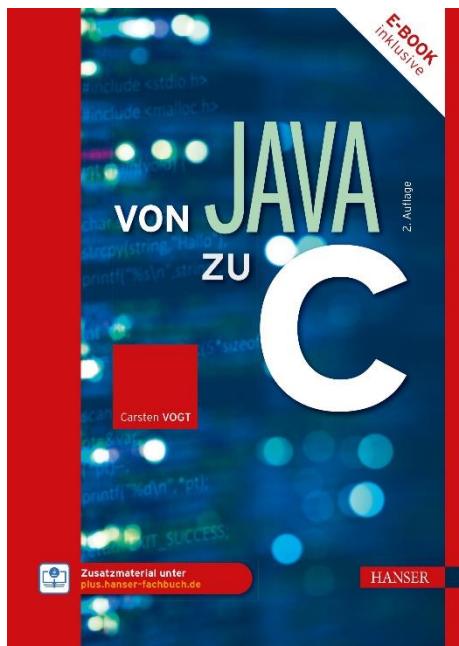


HANSER



Leseprobe

zu

Von Java zu C

von Carsten Vogt

Print-ISBN: 978-3-446-48103-9

E-Book-ISBN: 978-3-446-48128-2

Weitere Informationen und Bestellungen unter
<https://www.hanser-kundencenter.de/fachbuch/artikel/9783446481039>
sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort	XI
Zusatzmaterial zum Buch.....	XIII
1 Einführung.....	1
1.1 C und Java von den Anfängen bis heute	1
1.1.1 Die Entwicklung von C.....	1
1.1.1.1 Der Ursprung.....	1
1.1.1.2 Grundlegende Eigenschaften	1
1.1.1.3 Standards	2
1.1.2 Objektorientierte Nachfolgesprachen	3
1.1.2.1 C++.....	3
1.1.2.2 Java.....	3
1.1.3 Einsatzgebiete von C und Java	4
1.2 C und Java im Sprachvergleich.....	4
1.2.1 Drei Beispieldateien.....	4
1.2.1.1 Einfaches Programm mit Ausgabe.....	4
1.2.1.2 Programm mit Eingabe und C-spezifischen Datentypen	5
1.2.1.3 Programm mit einer Funktion	7
1.2.2 Eigenschaften von Java vs. Eigenschaften von C	8
1.2.2.1 Tabellarischer Vergleich	8
1.2.2.2 Objektorientierung vs. Prozedurorientierung.....	9
1.2.2.3 Interpretation vs. Übersetzung	10
1.3 Zu diesem Buch.....	12
1.3.1 Aufbau	12
1.3.2 Benutzung	13
1.3.3 Weitere Quellen	14
2 Struktur und Übersetzung von C-Programmen	17
2.1 Struktur von C-Programmen	17
2.1.1 C-Quellcode in einer einzelnen Datei	17
2.1.2 C-Quellcode in mehreren Dateien	18
2.2 Übersetzung von C-Programmen.....	19
2.2.1 Phasen der Übersetzung.....	19
2.2.2 Modularisierung.....	21
2.2.3 GCC und weitere Programmierwerkzeuge	22
2.3 Anweisungen des Präprozessors	24
2.3.1 #include: Einfügen von Header-Dateien.....	25
2.3.2 #define: einfache Ersetzung von Zeichenketten	26
2.3.3 #define: Makros mit Parametern.....	27
2.3.4 #ifdef, #if: bedingte Übersetzung	29
2.4 Übungsaufgaben.....	30

3	Kontrollstrukturen.....	33
3.1	Blöcke	33
3.2	Bedingte Anweisungen	34
3.3	Schleifen.....	34
3.4	Ausnahmebehandlung und goto	35
3.5	Übungsaufgaben.....	36
4	Datenorganisation	39
4.1	Skalare Datentypen	39
4.1.1	Zahlen- und Zeichentypen	39
4.1.2	Wahrheitswerte	41
4.1.3	Operationen.....	42
4.2	Konstanten und Variablen.....	44
4.2.1	Konstanten	44
4.2.2	Definition und Initialisierung von skalaren Variablen	45
4.2.3	Wertzuweisungen.....	45
4.3	Arrays.....	46
4.3.1	Eindimensionale Arrays.....	46
4.3.2	Mehrdimensionale Arrays.....	49
4.3.3	Zeichenketten.....	50
4.4	Strukturen.....	52
4.4.1	Grundlegende Eigenschaften von Strukturen	52
4.4.2	Strukturtypen	54
4.4.3	Schachtelung von Strukturen	55
4.5	Unions und Bitfelder	55
4.5.1	Unions	55
4.5.2	Bitfelder	56
4.6	Selbstdefinierte Wert- und Typnamen	58
4.6.1	Aufzählungstypen	58
4.6.2	Der typedef-Operator	59
4.7	Übungsaufgaben.....	60
5	Zeiger.....	63
5.1	Java-Objektvariablen vs. C-Zeigervariablen.....	63
5.2	Grundlegende Begriffe und Operatoren.....	65
5.2.1	Speicheradressen und Zeigervariablen	65
5.2.2	Adress- und Dereferenzierungsoperator	67
5.2.3	Zwei Programmbeispiele	68
5.2.4	Ungetypte Zeiger	70
5.3	Adressarithmetik	70
5.3.1	Operationen.....	70
5.3.2	Adressarithmetik bei Arrays	72
5.3.3	Exkurs: Zeichenkettenvariablen und -konstanten.....	74
5.4	Dynamische Speicherverwaltung	75
5.4.1	malloc().....	75
	5.4.1.1 Objekterzeugung in Java vs. Speicherbelegung in C	75

5.4.1.2	Definition von malloc().....	76
5.4.2	free().....	77
5.4.3	Zwei Programmbeispiele	78
5.5	Zeiger auf Strukturen	79
5.5.1	Arrays mit Zeigern auf Strukturen.....	80
5.5.2	Strukturen mit Zeigern auf Strukturen.....	81
5.6	Zeiger auf Zeiger.....	82
5.7	Übungsaufgaben.....	83
6	Funktionen	87
6.1	Java-Methoden vs. C-Funktionen	87
6.2	Schnittstellen	89
6.2.1	Prototypen.....	89
6.2.2	Weitere Besonderheiten von C	91
6.3	Ausführung.....	93
6.3.1	Ablauf	93
6.3.2	Parameterübergabe.....	94
6.3.2.1	Wertaufruf	94
6.3.2.2	Referenzaufruf.....	95
6.3.2.3	Übergabe von Arrays	97
6.3.3	Ergebnisrückgabe.....	98
6.4	Das Hauptprogramm main()	100
6.5	Sichtbarkeiten und Lebensdauern	101
6.5.1	Lokale Variablen.....	102
6.5.1.1	Automatische Variablen.....	102
6.5.1.2	Statische Variablen.....	102
6.5.1.3	Registervariablen.....	103
6.5.2	Globale Variablen	104
6.5.2.1	Programme in einer einzelnen Datei	104
6.5.2.2	Programme in mehreren Dateien	105
6.5.3	Tabellarische Zusammenfassung	107
6.6	Funktionsbibliotheken.....	107
6.6.1	Definition und Benutzung	107
6.6.2	Die Standardbibliothek	108
6.6.2.1	Funktionen für Zeichen und Zeichenketten	109
6.6.2.2	Mathematische Funktionen	111
6.6.2.3	Betriebssystemnahe Dienste.....	112
6.7	Techniken für Fortgeschrittene	114
6.7.1	Zeiger auf Funktionen.....	114
6.7.2	Funktionen als Parameter.....	116
6.7.3	Funktionen mit variabler Anzahl von Parametern	116
6.8	Übungsaufgaben.....	118
7	Ein-/Ausgabe und Dateizugriffe.....	123
7.1	Grundlegende Konzepte.....	123
7.1.1	Datenströme in Java und in C	123

7.1.2	Standardströme/-dateien	125
7.1.3	Klassen von E/A-Funktionen.....	125
7.2	Funktionen für die Standardein-/ausgabe	127
7.2.1	printf(): formatierte Ausgabe	127
7.2.1.1	Grundidee.....	127
7.2.1.2	Allgemeine Form	128
7.2.1.3	Weitere Beispiele	128
7.2.2	scanf(): formatierte Eingabe	129
7.2.2.1	Grundidee.....	129
7.2.2.2	Allgemeine Form	130
7.2.2.3	Pufferung der Eingabedaten.....	131
7.2.2.4	Weitere Beispiele	131
7.2.3	Weitere Funktionen für Zeichen und Zeichenketten	134
7.3	Funktionen für beliebige Datenströme.....	135
7.3.1	Öffnen und Schließen	135
7.3.2	Ein-/Ausgabe einzelner Zeichen.....	138
7.3.3	Ein-/Ausgabe von Zeichenketten.....	138
7.3.4	Formatierte Ein-/Ausgabe	139
7.3.5	Ein-/Ausgabe beliebiger Bytefolgen.....	140
7.3.6	Wahlfreier Zugriff.....	141
7.3.7	Spezielle Funktionen.....	143
7.4	Operationen auf dem Dateisystem	145
7.5	Übungsaufgaben.....	145
8	Dynamische Datenstrukturen	149
8.1	Dynamische Datenhaltung in Java und in C	149
8.2	Listen.....	150
8.2.1	Eigenschaften.....	150
8.2.2	Einfach verkettete Listen	151
8.2.2.1	Typ der Knoten	151
8.2.2.2	Durchlaufen einer Liste.....	152
8.2.2.3	Suchen von Einträgen	153
8.2.2.4	Einfügen von Knoten	153
8.2.2.5	Entfernen von Knoten	156
8.2.3	Doppelt verkettete Listen.....	159
8.2.3.1	Typ der Knoten	159
8.2.3.2	Durchlaufen einer Liste.....	160
8.2.3.3	Suchen von Einträgen	160
8.2.3.4	Einfügen von Knoten	161
8.2.3.5	Entfernen von Knoten	163
8.2.4	Queues und Stacks	165
8.2.4.1	Queues.....	165
8.2.4.2	Stacks	166
8.3	Hashtabellen	166
8.3.1	Eigenschaften.....	167
8.3.2	Realisierung in Java und in C	167

8.4	Bäume	169
8.4.1	Eigenschaften.....	169
8.4.2	Binärbäume.....	170
8.4.2.1	Eigenschaften und Beispiele	170
8.4.2.2	Realisierung in C.....	172
8.4.2.3	Durchlaufen eines Binärbaums	173
8.4.2.4	Löschen eines Binärbaums.....	175
8.4.2.5	Suchen eines Werts in einem Suchbaum	176
8.4.2.6	Einfügen eines Werts in einen Suchbaum.....	176
8.4.2.7	Löschen eines Werts aus einem Suchbaum	177
8.5	Mengen.....	180
8.5.1	Realisierung durch Listen und Bäume	180
8.5.1.1	Grundlegende Mengenoperationen auf C-Listen	180
8.5.1.2	Bilden der Vereinigungsmenge.....	181
8.5.1.3	Bilden der Differenzmenge	182
8.5.1.4	Bilden der Schnittmenge	182
8.5.2	Realisierung durch Bitmaps.....	183
8.6	Übungsaufgaben.....	185
A	Auswertung von Ausdrücken.....	187
A.1	Implizite Typkonversionen	187
A.1.1	Konversionen in Rechenausdrücken.....	187
A.1.2	Konversionen bei Zuweisungen.....	188
A.2	Sequenzpunkte	189
A.3	Bindungsstärken und Auswertungsreihenfolgen.....	190
B	Vordefinierte Konstanten.....	191
B.1	Wertebereiche der skalaren Typen.....	191
B.2	Mathematische Konstanten	192
C	Standardbibliothek	193
C.1	Dateizugriffe und Ein-/Auszgabe	193
C.1.1	Thematische Übersicht über die Funktionen	193
C.1.2	Funktionen in alphabetischer Reihenfolge	195
C.2	Zeichen, Zeichenketten und Bytefolgen	207
C.2.1	Test einzelner Zeichen	207
C.2.2	Umwandlung von Zeichen	207
C.2.3	Zeichenketten.....	208
C.2.4	Bytefolgen/Arrays.....	209
C.2.5	Konversionen	210
C.3	Mathematische Funktionen	210
C.4	Betriebssystemnahe Dienste.....	212
C.4.1	Dynamische Speicherverwaltung	212
C.4.2	Zeitfunktionen.....	212
C.4.3	Weitere Funktionen.....	214

D Häufig benötigte Tabellen	215
D.1 ASCII	215
D.2 Variablengrößen und Wertebereiche.....	216
D.3 Bindungsstärke von Operatoren.....	217
D.4 Optionen für fopen()	218
D.5 Konversionsangaben für die Ein-/Ausgabe.....	219
D.5.1 printf()	219
D.5.2 scanf()	221
Literatur und Internet	223
Bücher	223
Standardisierungsdokumente.....	223
Internet-Quellen.....	224
Index	225

Vorwort

Dieses Buch gibt eine Einführung in die Programmiersprache C und setzt dabei Kenntnisse in der Sprache Java voraus. Auf den ersten Blick mag das ungewöhnlich erscheinen, ist doch C ein Vorläufer von Java und nicht umgekehrt. Der Ansatz ist dennoch sinnvoll, da in vielen Studiengängen Java als erste Programmiersprache gelehrt wird. In weiterführenden Fächern und der darauf aufbauenden Berufspraxis werden jedoch auch C-Kenntnisse benötigt, beispielsweise zur hardwarenahen Programmierung oder zur Programmierung an der Schnittstelle eines Betriebssystems. C muss also „nachgelernt“ werden.

Das Buch wendet sich daher an Studentinnen, Studenten und andere Interessierte, die bereits Erfahrung mit Java haben und C als weitere Programmiersprache lernen wollen oder müssen. Es ist keine grundständige Darstellung von C, sondern konzentriert sich auf die Besonderheiten der Sprache im Vergleich zu Java. Damit bietet es eine zwar vergleichsweise kurze, aber doch recht detaillierte und tiefgängige Einführung in C. Profitieren wird man auch, wenn man schon einmal mit C in Berührung gekommen ist und nun seine Kenntnisse vertiefen möchte.

Leserinnen und Leser lernen zunächst die grundlegenden Unterschiede in den Sprachansätzen von C und Java, aber auch die vielfältigen Gemeinsamkeiten beider Sprachen kennen. Sie werden dann mit den Besonderheiten von C vertraut gemacht und lernen, die C-spezifischen Konzepte praktisch anzuwenden. Insbesondere werden sie dazu befähigt, sicher mit Zeigern/Pointern (einem fundamentalen Sprachkonstrukt, das es in Java so nicht gibt) umzugehen und dynamische Datenstrukturen, die in Java durch vordefinierte Klassen bereitgestellt werden, in C selbst auszuprogrammieren.

Das Buch kann man auf drei Arten nutzen:

- Wenn man sich rasch einen Überblick über C verschaffen möchte, so sollte man die acht „Schnelleinstiege“ zu Beginn der einzelnen Kapitel lesen. Sie ermöglichen den unmittelbaren Einstieg in die praktische C-Programmierung.
- Wenn man C im Detail kennenlernen möchte, so sollte man die Kapitel des Buchs sukzessive durcharbeiten und die Beispielprogramme praktisch ausprobieren. Man lernt dabei nicht nur die sprachlichen Möglichkeiten von C, sondern auch typische Programmiertricks und -fallen kennen.
- Wenn man bei der späteren praktischen Arbeit bestimmte Details nachschlagen möchte, so sollte man dazu die Anhänge benutzen. Insbesondere findet man ganz am Ende des Buchs eine tabellarische Darstellung von Informationen, die man bei der C-Programmierung häufig benötigt.

Viele Beispiele und Grafiken verdeutlichen den Stoff und Verweise innerhalb des Buchs zeigen Zusammenhänge zwischen den Teilbereichen auf. Tricks, Fallen und Informationen für Fortgeschrittene sind typografisch hervorgehoben. Übungsaufgaben dienen zur Überprüfung des Lernerfolgs.

Die erste Auflage des Buchs erschien unter dem Titel „C für Java-Programmierer“. Diese zweite Auflage mit dem Titel „Von Java zu C“ wurde bezüglich einiger weniger technischer Details aktualisiert. Die Änderungen halten sich aber in engen Grenzen, da C eine sehr stabile Programmiersprache ist. Zudem wurden die Quellenhinweise und die Empfehlungen zu Programmierwerkzeugen aufgefrischt sowie Fehler korrigiert. Schließlich wurde der Text im Hinblick auf eine geschlechtergerechte Sprache überarbeitet, was auch der Grund für die Änderung des Buchtitels war. Sterne * treten aber nach wie vor nur als Operatoren der Programmiersprache C auf.

Köln/Bergisch Gladbach, im Sommer 2024

Carsten Vogt

Zusatzmaterial zum Buch

Zu diesem Buch stehen Ihnen weitere Inhalte digital zur Verfügung:

- die Beispielprogramme,
- die Lösungen der Übungsaufgaben,
- die nach Drucklegung entdeckten Fehler

Gehen Sie dazu einfach auf

<https://plus.hanser-fachbuch.de>

und geben Sie dort diesen Code ein:

plus-12abc-8xyz9

Hinweise auf Dokumentationen und Werkzeuge, die im Internet frei verfügbar sind, gibt der Literaturteil auf Seite 223.

5 Zeiger

Das **Zeiger-/Pointerkonzept** ist eine charakteristische Eigenschaft der Programmiersprache C: Zeigervariablen enthalten Adressen von Speicherzellen. Sie „zeigen“ somit auf diese Speicherzellen und ermöglichen dadurch den Zugriff auf die dort gespeicherten Werte. Im Zeigerkonzept wird also die grundlegende Eigenschaft von C deutlich, nicht nur eine anwendungsorientierte, sondern auch eine hardwarenahe Sprache zu sein.

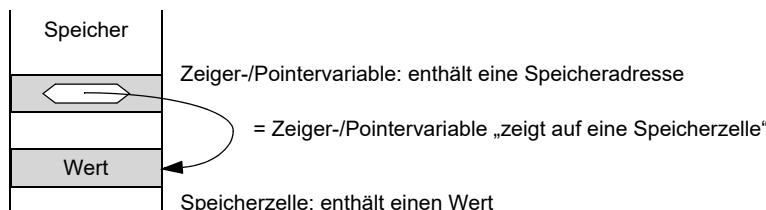


Abbildung 5.1 Speicheradresse in einer Zeiger-/Pointervariablen

Zeiger erlauben eine sehr flexible Programmierung: Mit ihnen kann ein Programm während seiner Ausführung, also „dynamisch“, bestimmen, auf welchen Speicherzellen es arbeitet, und dabei auf beliebige Bereiche seines Speichers zugreifen. Zeiger sind aber auch gefährlich: Bitmuster in Zellen sind ohne eine zwingende Typprüfung oder andere Schutzmechanismen zugänglich, so dass die Fehlergefahr hoch ist. In Java hat man daher auf ein allgemeines Zeigerkonzept verzichtet und sich auf typsichere Objektreferenzen beschränkt.

5.1 Java-Objektvariablen vs. C-Zeigervariablen

Die von Java her bekannten **Objektreferenzen** sind Verweise auf Objekte. Objektreferenzen werden in Objektvariablen gespeichert, über die man auf die Objekte zugreifen kann. Objekte und Objektvariablen sind typisiert, gehören also Klassen an, und bei jeder Operation auf einer Objektvariablen findet eine strenge Typprüfung statt.

Ein einfaches Java-Programm mit einem Objekt, das Informationen über eine Person in einer Firma enthält, könnte beispielsweise wie folgt aussehen:

```
class AngestelltenInfo {
    String name;
    int personalnummer;
    float gehalt;
};

...
AngestelltenInfo a = new AngestelltenInfo();
```

Java ...

Das Programm definiert die Klasse `AngestelltenInfo` (wobei, um einen unmittelbaren Vergleich mit einer C-Struktur ziehen zu können, keine Methoden vereinbart werden, ins-

besondere auch keine `get-` und `set-`Methoden und kein Konstruktor). Es erzeugt dann ein Objekt dieser Klasse und legt in der Variablen `a` eine Referenz darauf ab. Abbildung 5.2 illustriert die zugrunde liegende Sichtweise: Eine typisierte Objektvariable verweist auf ein typisiertes Objekt. Davon, dass das Objekt und auch die Variable durch Bitmuster in Speicherzellen realisiert werden, wird vollständig abstrahiert.

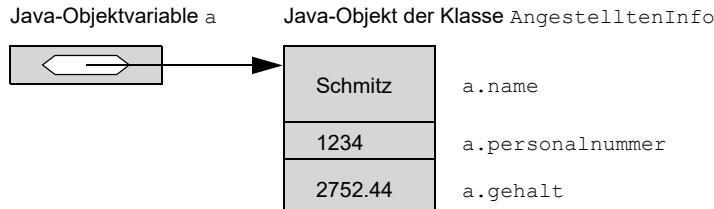


Abbildung 5.2 Objektvariable und Objekt in Java

Ein C-Programm, das diesem Java-Beispiel entspricht, könnte die folgende Form haben:

```

typedef struct {
    char name[41];
    int personalnummer;
    float gehalt;
} angestellten_info;
angestellten_info as;
angestellten_info *a;
a = &as;
  
```

... C

Wie aus → 4.4 her bekannt, wird zunächst ein Strukturtyp `angestellten_info` definiert und eine Variable `as` dieses Typs vereinbart. Neu sind die letzten beiden Zeilen des Programms: Hier wird eine **Zeiger-/Pointervariable** `a` definiert, die Speicheradressen von Variablen des Typs `angestellten_info` aufnehmen kann. Dies wird durch die Typangabe `angestellten_info *` (sprich „Zeiger/Pointer auf `angestellten_info`“) festgelegt. Anschließend wird durch den **Adressoperator** `&` die Speicheradresse von `as` ermittelt und in `a` gespeichert. Die Zeigervariable `a` zeigt jetzt also auf die Strukturvariable `as`.

Abbildung 5.3 verdeutlicht die Sichtweise von C: Variablen und deren Werte werden durch Speicherzellen mit den darin enthaltenen Bitmustern realisiert. Auf die Variablen kann man wahlweise über Namen oder über Speicheradressen zugreifen.

Beim Vergleich der beiden Beispiele fällt übrigens auf, dass im Java-Programm nur die Objektvariable einen Namen hat, nicht jedoch das Objekt selbst, während im C-Programm sowohl die Strukturvariable selbst als auch die Zeigervariable benannt sind. Es ist jedoch auch in C möglich, unbenannte Variablen zu erzeugen, auf die dann nur über (benannte) Zeigervariablen zugegriffen wird. Details dazu findet man in → 5.4.

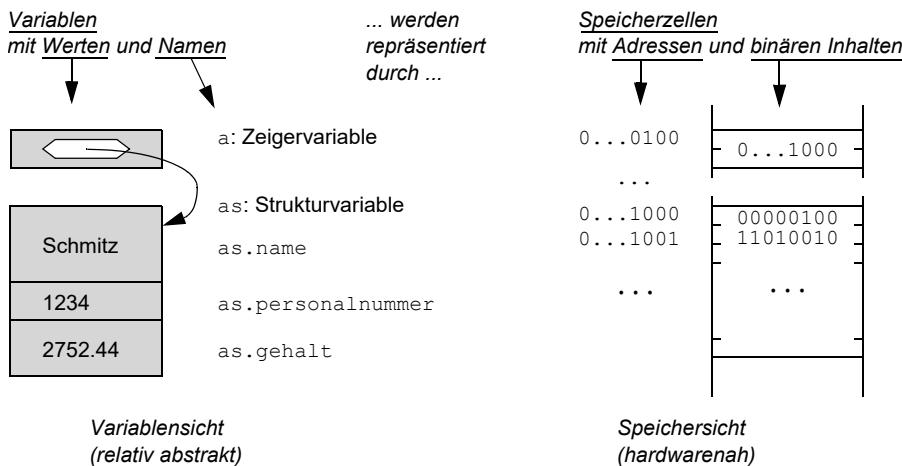


Abbildung 5.3 Zeiger und Zeigervariablen in C – Variablenansicht vs. Speichersicht

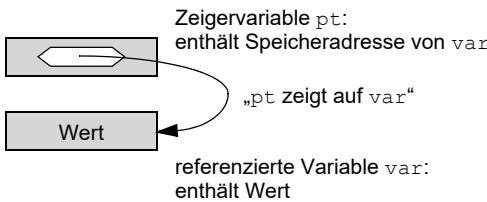
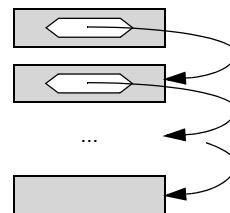
5.2 Grundlegende Begriffe und Operatoren

5.2.1 Speicheradressen und Zeigervariablen

Variablen in C haben **Adressen**: Die Adresse einer Variablen ist die Nummer der Speicherzelle, in der ihr Wert steht (oder, wenn die Variable mehrere Zellen belegt, die Nummer ihrer ersten Zelle, → Abbildung 5.3). Adressen können in benannten **Zeigervariablen** abgelegt werden. Enthält eine Zeigervariable `pt` die Adresse einer Variablen `var`, so sagt man, dass `pt var` **referenziert** oder dass `pt` auf `var` **zeigt** (→ Abbildung 5.4 links). Zeigervariablen werden auch kurz **Zeiger** oder **Pointer** genannt.

Zeigervariablen sind der Ausgangspunkt **indirekter Variablenzugriffe**: Der Zugriff auf die Zeigervariable liefert eine Adresse, über die dann im zweiten Schritt auf die referenzierte (also die „eigentliche“) Variable zugegriffen wird. Man kann so über die Zeigervariable den Wert der referenzierten Variablen auslesen oder man kann ihn überschreiben. Dabei sind auch mehrstufig indirekte Zugriffe möglich: Eine Zeigervariable kann auf eine zweite Zeigervariable zeigen, diese möglicherweise auf eine dritte und so weiter (→ Abbildung 5.4 rechts).

Eine Zeigervariable ist meist **typisiert** und kann dann nur Variablen eines bestimmten Typs referenzieren (siehe aber → 5.2.4). Der Typ wird bei der Deklaration der Zeigervariablen angegeben. Der Zugriff auf eine referenzierte Variable benötigt diese Typinformation, da dann der Wertebereich dieser Variablen und die auf ihr zulässigen Operationen bekannt sein müssen. Zudem ergibt sich aus der Typangabe, wie viele Speicherzellen (ab der durch die Zeigervariable angegebenen Zelle) zur referenzierten Variablen gehören. So verweist beispielsweise ein `char`-Zeiger auf eine einzelne Speicherzelle, ein `double`-Zeiger auf eine Gruppe von (meist) acht Speicherzellen (→ 4.1.1).

Einstufige Indirektion:**Mehrstufige Indirektion:****Abbildung 5.4** Indirektion mit Zeigervariablen

Um eine Zeigervariable von einer „normalen“ Variablen zu unterscheiden, wird ihrem Namen bei der Deklaration ein * vorangestellt. Beispiele für Deklarationen von Zeigervariablen sind die folgenden:

- `char *cpt;`
deklariert eine Variable `cpt`, die Adressen von Variablen des Typs `char` aufnehmen kann.
- `angestellten_info *apt;`
deklariert eine Variable `apt`, die Adressen von Strukturvariablen des Typs `angestellten_info` aufnehmen kann.
- `float **fppt;`
deklariert eine Variable `fppt`, die Adressen von Variablen aufnehmen kann, in denen wiederum Adressen von Variablen des Typs `float` stehen können. Hier wird also eine zweistufige Indirektion realisiert (→ Abbildung 5.4, → 5.6).

Die Sprechweise ist dann beispielsweise: „`cpt` ist ein Zeiger/Pointer auf `char`“ oder „`fppt` ist ein Zeiger auf Zeiger auf `float`“.



Der Stern bei der Variablen-deklaration gehört stets zu *einem* Variablen-namen. Will man also zwei Zeiger deklarieren, so muss man `int *a, *b` schreiben; `int *a, b` würde eine Zeigervariable `a` und eine „normale“ `int`-Variable `b` deklarieren.

Dass hier kein Beispiel für einen Zeiger auf Arrays angegeben wird, hat einen besonderen Grund: In C ist ein Array nichts anderes als ein Zeiger, nämlich ein Zeiger auf den Anfang der Folge von Speicherzellen, in denen der Inhalt des Arrays steht. Näheres zu diesem Thema findet man in → 5.3.2.

Zeigervariablen können, außer Adressen anderer Variablen, den Wert `NULL` enthalten. `NULL` ist der **Nullzeiger**, der angibt, dass die Zeigervariable zur Zeit auf keine andere Variable verweist. Die Konstante `NULL` ist in den Header-Dateien `stdio.h` und `stdlib.h` definiert; man kann daher in Zuweisungen und Vergleichen statt `NULL` auch den numerischen Wert 0 verwenden.



Eine Zeigervariable, die zwar definiert, aber noch nicht initialisiert wurde, verweist auf irgendeine Zelle des Speichers. Ein Zugriff auf diese Speicherzelle ist kritisch, denn dabei könnte der Wert der Variablen, die zufällig an dieser Stelle steht, über-

schrieben werden. Da hier weder vom C-Compiler noch beim Programmablauf eine Fehlermeldung geliefert wird, muss man bei der Programmierung selbst darauf achten, dass Zeigervariablen zuerst initialisiert und erst danach benutzt werden. Einer Zeigervariable kann insbesondere auf die folgenden beiden Arten ein Anfangswert zugewiesen werden:

- Durch Zuweisung der Adresse einer existierenden Variablen (→ 5.2.2) oder des Nullzeigers.
- Durch Belegung eines zuvor freien Speicherbereichs und Zuweisung von dessen Adresse (→ 5.4.1).

Übrigens können Zeigervariablen nicht nur auf andere Variablen, sondern auch auf Funktionen verweisen. Mit Zeigern auf Funktionen beschäftigt sich → 6.7.

5.2.2 Adress- und Dereferenzierungsoperator

Zur Arbeit mit Zeigern gibt es in C zwei grundlegende Operatoren (siehe hierzu auch → Abbildung 5.5 unten):

- Der **Adressoperator** & liefert zu einer Variablen deren Adresse. Man kann diese Adresse in einer Zeigervariable speichern:

```
int i;
int *ipt;
ipt = &i;
```

Auch kann man mit so ermittelten Adressen „rechnen“, also beispielsweise die Adresse der im Speicher vorangehenden oder folgenden Variablen ermitteln (→ 5.3).

- Der **Dereferenzierungsoperator** * liefert zu einem Zeiger die Variable, auf die dieser Zeiger verweist. Beispielsweise wird durch

```
int i;
int *ipt;
ipt = &i;
*ipt = 1;
```

der Variablen *i* der Wert 1 zugewiesen. Durch

```
*ipt = *ipt + 1;
```

oder auch

```
(*ipt)++;
```

wird der Wert von *i* um 1 erhöht.

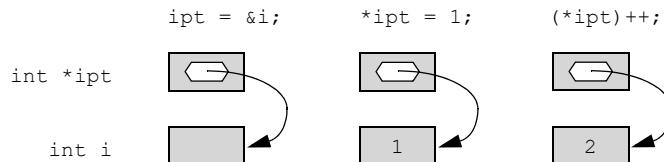


Abbildung 5.5 Basisoperationen auf Zeigervariablen



Bei der Programmierung mit Zeigern muss man stets gut überlegen, mit welcher Variablen das Programm arbeiten soll – mit der Zeigervariablen selbst oder mit der Variablen, auf die die Zeigervariable verweist. Beispielsweise besteht ein erheblicher Unterschied zwischen den Zuweisungen $pt2 = pt1$ und $*pt2 = *pt1$ (wobei $pt1$ und $pt2$ zwei Zeigervariablen sind, → Abbildung 5.6):

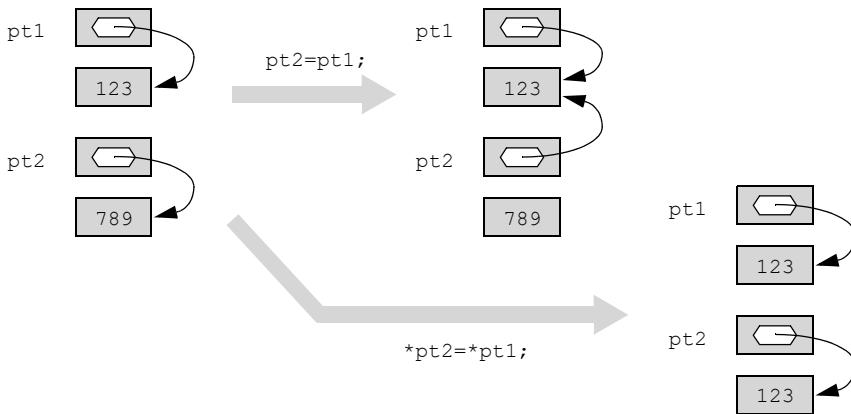


Abbildung 5.6 Zuweisung an Zeigervariable versus Zuweisung an referenzierte Variable

- Durch $pt2 = pt1$ wird der Inhalt der Zeigervariablen $pt1$ (eine Adresse) in die Zeigervariable $pt2$ kopiert. Beide Zeigervariablen referenzieren also anschließend dieselbe Variable; die Inhalte der referenzierten Variablen selbst bleiben dagegen unverändert.
- Durch $*pt2 = *pt1$ wird der Inhalt der Variablen, auf die $pt1$ verweist, in die Variable kopiert, auf die $pt2$ verweist. Der Inhalt einer referenzierten Variablen ändert sich also, die Inhalte der Zeigervariablen bleiben aber unverändert.

Übrigens sind direkte Zuweisungen zwischen Zeigervariablen nur dann möglich, wenn beide Variablen vom selben Typ sind. Andernfalls muss eine explizite Typumwandlung vorgenommen werden:

$pt2 = (t2 *) pt1;$ (wobei $pt2$ vom Typ $t2 *$ ist)

5.2.3 Zwei Programmbeispiele

Das erste Programmbeispiel demonstriert die Effekte verschiedener Adress- und Dereferenzierungsoperationen:

```
int *pt1, *pt2;
int var1 = 100, var2 = 200;

pt1 = &var1;           /* pt1 zeigt nun auf var1 */
*pt1 = *pt1 + 1;      /* entspricht var1 = var1 + 1 */
pt2 = pt1;            /* pt2 zeigt nun auch auf var1 */
pt1 = &var2;           /* pt1 zeigt nun auf var2 */
```

```
(*pt1)++;           /* entspricht var2 = var2 + 1; */
*pt2 = 150;          /* entspricht var1 = 150 */
pt1 = &var1;          /* pt1 zeigt nun wieder auf var1 */
pt2 = &var2;          /* pt2 zeigt nun auf var2 */
*pt2 = *pt1;         /* entspricht var2 = var1; */
```

Das zweite Programmbeispiel zeigt die Verwendung des Adress- und des Dereferenzierungsoperators in einem konkreten Anwendungsproblem, nämlich bei der Verwaltung von Bankkonten. Hier kann man durch eine Eingabe eines von zwei Konten auswählen und dann auf das gewählte Konto einen bestimmten Betrag einzahlen:

```
float kontostand_1 = 0.0,
      kontostand_2 = 0.0,
      *kontozeiger,
      einzahlung;

int wahl;
printf("Bitte wählen: 1 = Konto 1, 2 = Konto 2   ");
scanf("%d",&wahl);

if (wahl==1)
    kontozeiger = &kontostand_1;
else
    kontozeiger = &kontostand_2;
printf("Bitte Einzahlungsbetrag eingeben:   ");
scanf("%f",&einzahlung);
*kontozeiger = *kontozeiger + einzahlung;
```

Nach der if-else-Anweisung verweist die Zeigervariable `kontozeiger` auf die Variable, die den Stand des ausgewählten Kontos angibt – also entweder auf `kontostand_1` oder auf `kontostand_2`. Diese Variable wird dann in der letzten Anweisung um den Einzahlungsbetrag erhöht. Hier ergibt sich also erst während des Programmablaufs (also „dynamisch“ bei der Programmausführung), mit welcher Variablen gearbeitet wird; zur Zeit der Programmübersetzung liegt das noch nicht fest.

Man könnte einwenden, dass der gewünschte Effekt genauso gut durch die Anweisung

```
if (wahl==1)
    kontostand_1 = kontostand_1 + einzahlung;
else
    kontostand_2 = kontostand_2 + einzahlung;
```

erzielt würde – also ganz ohne Zeigervariable. Für das einfache Beispiel hier ist das sicher richtig. Sollen aber auf der gewählten Variablen mehrere Operationen ausgeführt werden, würde das Programm ohne Zeigervariable deutlich komplexer, da dann jede Operation eine neue if-else-Fallunterscheidung erfordert.

In diesem Beispiel wird übrigens auch die Bedeutung des `&` vor dem Variablenamen im `scanf()`-Aufruf klar: Es liefert die Adresse der Variablen – also die Information, in welche Speicherzelle(n) der eingelesene Wert gebracht werden soll.

5.2.4 Ungetypte Zeiger



Zeigervariablen werden meist bezüglich eines bestimmten Typs deklariert und können damit nur Variablen dieses Typs referenzieren. Dies ist aber nicht zwingend notwendig:

```
void *pt;
```

deklariert eine Variable `pt`, die Adressen von Variablen eines beliebigen Typs speichern kann. Man kann `pt` also im Laufe ihres „Lebens“ Adressen von Variablen unterschiedlicher Typen zuweisen:

```
int i = 1234;
float f = 1.2345;
pt = &i;
printf("Wert von *pt: %d\n",*((int *)pt));
pt = &f;
printf("Wert von *pt: %f\n",*((float *)pt));
```

Wie das Beispiel zeigt, muss hier jeweils eine explizite Typumwandlung des Werts der Zeigervariablen stattfinden, wenn auf die referenzierte Variable zugegriffen werden soll.

5.3 Adressarithmetik

5.3.1 Operationen

Zeigervariablen enthalten Speicheradressen, also ganzzahlige Nummern von Speicherzellen. Mit Zeigervariablen lässt es sich daher rechnen oder, wie man auch sagt, **Adressarithmetik** betreiben. Beispielsweise kann man einen Speicherbereich durchlaufen, indem man eine Adresse schrittweise erhöht: Ist `pt` eine Zeigervariable, die eine Variable im Speicher referenziert, so ist `pt+1` die Adresse der nächsten Variablen, `pt+2` der übernächsten und so weiter. So kann man mit Anweisungen wie

```
* (pt+1) = 10;
* (pt+2) = * (pt+1) + 10;
* (pt+i) = 100; // mit einer ganzzahligen Variablen i
```

auf verschiedenen referenzierten Variablen arbeiten (→ Abbildung 5.7).

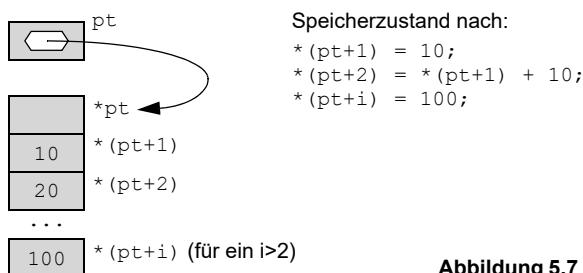


Abbildung 5.7 Zeigerarithmetik – Rechnen mit Adressen

Durch Zuweisungen der Form

```
pt++;
pt = pt + 2;
pt += i;
```

lässt sich der Wert der Zeigervariablen selbst ändern (→ Abbildung 5.8).

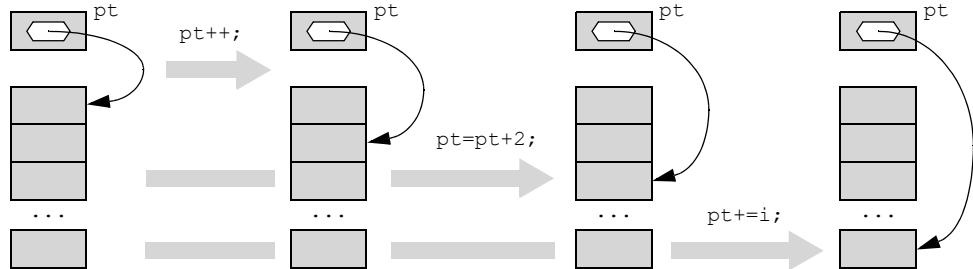


Abbildung 5.8 Zeigerarithmetik – Rechnen mit Adressen und Zuweisung an Zeigervariablen

Zahlenwerte, die in Ausdrücken der Adressarithmetik auftreten, stehen nicht für eine Anzahl von Bytes, sondern für eine Anzahl von Variablen. So wird beispielsweise durch $pt=pt+1$ (oder $pt++$) der Adresswert in pt um so viele Bytenummern erhöht, dass pt nun auf die nächste Variable im Speicher verweist. Wie viele Bytes das sind, hängt vom Typ ab, für den pt deklariert ist (→ 5.2.1): Beispielsweise beträgt die Schrittweite bei Zeigern auf `char` ein Byte, bei Zeigern auf `double` aber z.B. acht Byte (abhängig von der konkreten Plattform). Allgemein gilt: Referenziert pt Variablen des Typs T , so entspricht ein Zahlenwert n , der in einem Ausdruck mit pt auftritt, $n * \text{sizeof}(T)$ Speicherbytes.



Kombiniert man die Adressarithmetik mit dem Dereferenzierungsoperator, so muss man die Regeln zur Auswertungsreihenfolge der Operatoren beachten (→ Anhang D.3): So wird bei $*pt++$ zuerst die Adresse in pt inkrementiert und dann die resultierende Adresse dereferenziert, denn Postfixoperationen werden vor Präfixoperationen ausgeführt. Möchte man dagegen den Inhalt der Speicherzelle, auf die pt zeigt, inkrementieren, so muss man Klammern setzen: $(*pt)++$.

Neben der Addition ganzer Zahlen auf Zeigervariablen ist auch die Subtraktion ganzer Zahlen wie $pt--$ oder $pt=pt-2$ zulässig. Auch kann man zwei Zeigervariablen per $==$ und $!=$ auf Gleichheit prüfen, sofern sie vom selben Typ sind; ein Vergleich mit `NULL` ist immer möglich. Zeigen zwei Zeigervariablen auf Komponenten desselben Arrays, so kann man sie durch $<$, $>$, $<=$ und $>=$ miteinander vergleichen und ihre Werte voneinander subtrahieren. Andere Operationen, wie beispielsweise die Multiplikation zweier Zeigervariablen, sind dagegen nicht sinnvoll und daher unzulässig.

Mit Hilfe der Adressarithmetik kann man also sehr flexibel programmieren. Die Adressarithmetik ist aber auch gefährlich, da weder vom C-Compiler noch beim Programmablauf hinreichend geprüft wird, ob sie sinnvolle Resultate liefert. So kann eine Zeigervariable nach der Adressrechnung durchaus auf einen Speicherbereich mit Variablen verweisen, de-

ren Typ nicht zum Typ der Zeigervariablen passt. Die Bitmuster in diesem Bereich werden dann fehlinterpretiert.

Man sollte daher nur dann mit Adressarithmetik arbeiten, wenn einem die Organisation des Speichers für das Programm genau bekannt ist. Das ist nicht so ohne Weiteres der Fall: Selbst bei skalaren Variablen, die unmittelbar hintereinander definiert wurden, kann man nicht unbesehen davon ausgehen, dass sie im Speicher in derselben Reihenfolge zusammenhängend abgelegt sind. In zwei Fällen lässt sich jedoch auch ohne tieferes Systemkenntnis die Adressarithmetik sicher benutzen:

- Bei Arrays, also zusammengesetzten Variablen, die eine Folge von Werten desselben Typs enthalten (→ 5.3.2).
- Bei Speicherblöcken, die das Programm vom Betriebssystem angefordert hat und deren Verwaltung es dann selbst übernimmt (→ 5.4).

5.3.2 Adressarithmetik bei Arrays

Arrays sind das ideale Anwendungsgebiet der Adressarithmetik: Sie bestehen aus mehreren Komponenten desselben Typs, auf die man über einen ganzzahligen Index zugreift (→ 4.3). Da zudem die Komponenten eines Arrays im Speicher aufeinanderfolgend abgelegt sind, lässt sich die **Arrayindizierung** unmittelbar durch Adressarithmetik realisieren.

Für ein C-Programm ist ein Arrayname `a` nichts anderes als eine Zeigerkonstante, die auf die Speicherzelle verweist, ab der die Arrayeinträge abgelegt sind. Eine Zuweisung an die i -te Komponente von `a` lässt sich dann wahlweise schreiben als `a[i]=0` oder als `*(a+i)=0` (→ Abbildung 5.9). Die Indexschreibweise, die für die Programmierung meist bequemer ist, wird dabei intern stets auf die Adressarithmetik zurückgeführt.

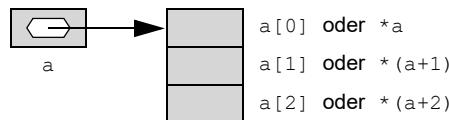


Abbildung 5.9 Adressierung von Arrays mit Index- oder mit Zeigerschreibweise

Das Programmstück, das in 4.3.1 folgendermaßen lautete:

```
unsigned int fibonacci[20];
fibonacci[0] = 1;
fibonacci[1] = 1;
for (int i=2; i<20; i++)
    fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
```

sieht in Zeigerschreibweise beispielsweise wie folgt aus:

```
unsigned int fibonacci[20];
*fibonacci      = 1;
*(fibonacci+1) = 1;
for (int i=2; i<20; i++)
    *(fibonacci+i) = *(fibonacci+i-1) + *(fibonacci+i-2);
```

Index

#define	26, 27	Datentyp in C	172
#endif.....	29	Standardoperationen	173
#if.....	29	Betriebssystem	
#ifdef	29	Dienste	112, 212
#include	25	Kommando	113, 214
&	43, 67	Bibliothek.....	19, 20, 107
&&.....	43	Standardbibliothek	108, 193
*	67	binär vs. formatiert.....	127, 129
^	43	Binärbaum.....	170
.....	43	Ausgeben	174
.....	43	Durchlaufen	173
Adressarithmetik	70	Einfügen.....	176
Adressoperator.....	67	Entfernen.....	177
Aggregat	49, 53	Löschen.....	175
Alignment.....	53	Suchen.....	176
ANSI-C.....	2	Binärkode	21
Anweisung, bedingte	34	Binärmodus	136, 140
Argument.....	94	Binder.....	20
Array.....	46	Bindungsstärke.....	42, 217
Adressarithmetik	72	Bitfeld	56
eindimensionaler	46	Bitoperatoren.....	43
Initialisierung	49, 50	Block	33
Linearisierung	50	boolean	41
mehrdimensionaler	49	C++	3
als Parameter	92, 97	C-Standards	2
mit Strukturen.....	53	Call	
mit Zeigern	80	by reference	95
asctime()	212	by value.....	94
Assembler.....	20	calloc()	112, 212
assert()	35	Cast	46, 187
atexit()	114, 214	char	39, 40, 44
atox()	110, 210	char16_t.....	40
Aufzählungstyp	58	char32_t.....	40
Ausgabe, formatierte	127	clearerr().....	144, 195
Ausnahmebehandlung	35	CLion	24
Auswertung		Clock Tick.....	113, 212
von Ausdrücken	42, 189, 217	clock()	113, 212
partielle.....	43	Codesegment	107
Baum	169	Compiler	20
binärer	170	const	44
		ctime()	113, 213

Cygwin	22	FIFO	165
Datei	123, 135	File Pointer.....	124, 137
Öffnen	135	Flag	57
Positionszeiger	136, 141	float	41, 44
Puffer.....	131, 136, 144	fopen()	135, 197
Schließen	137	for	34
Zugriffsfunktionen	125, 193	Formatangabe/	
Dateizeiger.....	124, 137	-element/-string	128, 129, 201, 204
Datensegment	75, 107	fprintf().....	139, 198
Datenstrom	123, 135	fputc().....	138, 198
Datenstruktur		fputs().....	139, 198
dynamische	149	fread().....	140, 198
rekursive	172	free().....	77, 212
Datentypen (siehe auch Typen).....	39	freopen().....	137
Definition.....	33	fscanf()	139, 199
Deklaration	33	fseek()	142, 199
Dereferenzierungsoperator	67	fsetpos().....	142
Dev-C++.....	24	ftell()	141, 199
difftime()	213	Funktion	87
double	41, 44	Argument	94
do-while.....	34	Aufruf	93
Downcast	46, 187	Definition	89
Eclipse	24	Deklaration	89
Ein-/Ausgabe	123	Kopf nach Kernighan/Ritchie	93
Standardfunktionen	125, 193	mathematische	111, 210
Eingabe, formatierte	129	als Parameter.....	116
Ellipse	117	Prototyp	18, 89
else.....	34	Rückgabewert	98
Endlosschleife	35	Zeiger darauf.....	114
enum	58	Funktionsbibliothek	107
Enumerationstyp.....	58	Standardbibliothek	108, 193
Error.....	20	Funktionsparameter.....	94
Escape-Sequenzen	44	Array	92, 97
exit()	114, 214	Funktion	116
fclose().....	137, 195	Struktur	91
Fehlerausgabe	125, 144	Übergabe	94
feof().....	143, 196	variable Anzahl	116
ferror()	144, 196	Zeiger	92, 95
fflush()	143, 196	fwrite()	141, 200
fgetc()	138, 196	ganze Zahlen	39
fgetpos().....	142	Ganzzahlkonstante	44
fgets().....	138, 197	Garbage Collection	77
		GCC (Compiler).....	22
		gcc (Kommando)	23

getc()	138, 196
getchar()	134, 200
Gleitkommakonstante	44
Gleitkommazahlen	41
goto	35
Hashfunktion	167
Hashtabelle	166
Hauptprogramm	100
Header-Datei	17, 25, 90
Heapsegment	75, 107
if	34
Indirektion	65
Initialisierung	45, 102, 103, 104
von Arrays	49, 50
von Stringvariablen	51
von Strukturen	53
von Unions	56
Inorder	173
int	39, 44
Integrated Development Environment (IDE)	24
Interpretation	11
isxxx()	109, 207
Java	3
im Vergleich zu C	8
Kalenderzeit	113, 213
Keller	151, 166
Kernighan-Ritchie-C	2
komplexe Zahlen	41
Konstante	44
benannte	26, 44
mathematische	192
symbolische	26, 58
vordefinierte	191
Zeichenkette	51, 74
Kontrollstruktur	33
Konversionsangabe	128, 129, 201, 204
Label	35
Lademodul	21
Laufzeitsystem	21
libc	108
Library	19
LIFO	166
Linker	20
Linux	108
Liste	150
Datentyp	151, 159
doppelt verkettete	150
Durchlaufen	152, 160
einfach verkettete	150
Einfügen	153, 161
Entfernen	156, 163
lineare	150
Suchen	153, 160
zyklisch verkettete	151
localtime()	113, 213
long	39, 44
long double	41, 44
long long	44
lvalue	46
main	100
Makro	27
malloc()	75, 76, 212
Maschinensprache	10
Memory Leak	77
memxxx()	112, 209
Menge	180
Realisierung durch Bitmap	183
Realisierung durch Liste	180
sortierte	151
MinGW	22
Modifikator	40
Modul	18, 21
Modularisierung	21
Nullzeiger	66
Objektmodul	20
objektorientiert	9
Öffnen einer Datei	135
Operatoren	42
aussagenlogische	42, 43
Bindungsstärke	42, 217
relationale	42

Padbyte	53	short.....	39, 40
Parameter.....	94	signed.....	41
perror().....	144, 200	size_t.....	76
Pointer (siehe auch Zeiger).....	65	sizeof.....	40
Pointervariable.....	64	Speicherbelegung, dynamische	75, 112, 212
Positionszeiger.....	136, 141	Speicherklasse.....	101
Postorder.....	174	auto	102
Präprozessor	20, 24	extern	104
Anweisungen.....	17	register	103
Preorder	173	static	102
printf()	127, 200	sprintf()	140, 206
Programmentwicklungsumgebung	24	Sprunganweisung.....	35
Programmiersprache		sscanf()	140, 206
objektorientierte	9	Stack.....	151, 166
prozedurale	10	Stacksegment	75, 94, 107
Prototyp	18, 89	Standardbibliothek	108, 193
prozedural	10	Standardströme/-dateien	125
Prozessorzeit.....	113	stdin/stdout/stderr.....	125
Pufferung der Eingabe	131	strcat()	51, 208
Punktnotation.....	52, 55	strcmp()	51, 208
putc()	138	strcpy()	51, 208
putchar()	135, 203	String (siehe auch Zeichenkette).....	50
puts()	135, 203	strlen()	51, 208
Queue.....	151, 165	Struktur (struct)	52
realloc()	112	als Funktionswert	99
Referenzaufruf.....	95	Initialisierung	53
Rekursion.....	172	als Parameter	91
wechselseitige	90	Typdefinition	54
remove()	145, 203	Zeiger darauf	79
rename().....	145, 203	mit Zeigern auf Strukturen	81
return	98	Strukturbau	171
rewind()	142, 204	strxxx()	109, 208
scanf()	129, 204	Suchbaum	171
Schachtelung		Einfügen	176
von Blöcken	33	Entfernen	177
von Strukturen	55	Suchen	176
Schleife	34	switch	34
endlose	35	system()	113, 214
Schließen einer Datei.....	137	Textmodus	136, 140
Sequenzpunkt	42, 189	time()	113, 213
setvbuf()	144, 206	tmpfile()	145, 206
Shift	43	tolower()	109, 207
		toupper()	109, 207
		Type Cast	46, 187

typedef	59	Wahrheitswerte	41
Typen.....	39	Warning.....	20
für Aufzählungen	58	Warteschlange.....	151, 165
global definierte	105	wchar_t.....	40
skalare	39	Wertaufruf.....	94
für Strukturen	54	Wertzuweisung	45, 188
Umwandlung	46	while.....	34
Wertebereiche	39, 41, 191, 216	Whitespace	109
Typumwandlung.....	187	WSL	22
Überlauf.....	133	Zeichen.....	39
Übersetzer.....	20	Konstanten	44
Übersetzung.....	11, 20	Standardfunktionen.....	109, 207
bedingte	29	Zeichenkette (String)	50
Umwandlungsfunktionen	110, 210	Initialisierung	51
ungetc().....	138	konstante	51, 74
Union	55	Standardfunktionen.....	109, 208
UNIX	108	Zeiger	65
unsigned.....	40, 44	als Funktionswert.....	99
Variable	39	als Parameter.....	92, 95
automatische.....	102	auf Funktionen	114
globale	45, 104	auf Zeiger	82
Initialisierung	45, 102, 103, 104	ungetypt	70
lokale	45, 102	Zeigervariable	64
statische	102	Zeitfunktionen.....	113, 212
Variable Length Arrays (VLAs).....	47	Zuweisung.....	45, 188
Variablenzugriff, indirekter.....	65		
Vektoroperationen	97		
Verdeckung	33		
Visual Studio	24		
void *	70		