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

Inhalt

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.

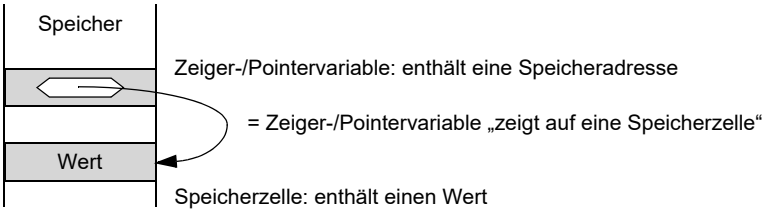


Abbildung 5.1 Speicheradresse in einer Zeiger-/Pointervariablen

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:

Java ...

```
class AngestelltenInfo {
  String name;
  int    personalnummer;
  float  gehalt;
};
...
AngestelltenInfo a = new AngestelltenInfo();
```

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.

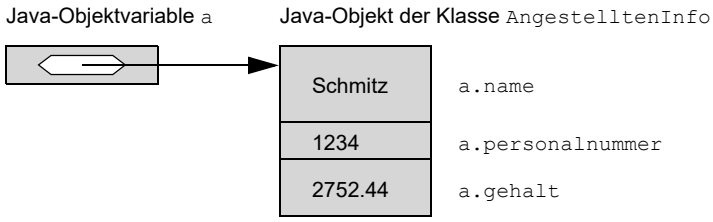


Abbildung 5.2 Objektvariable und Objekt in Java

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

... *C*

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

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.

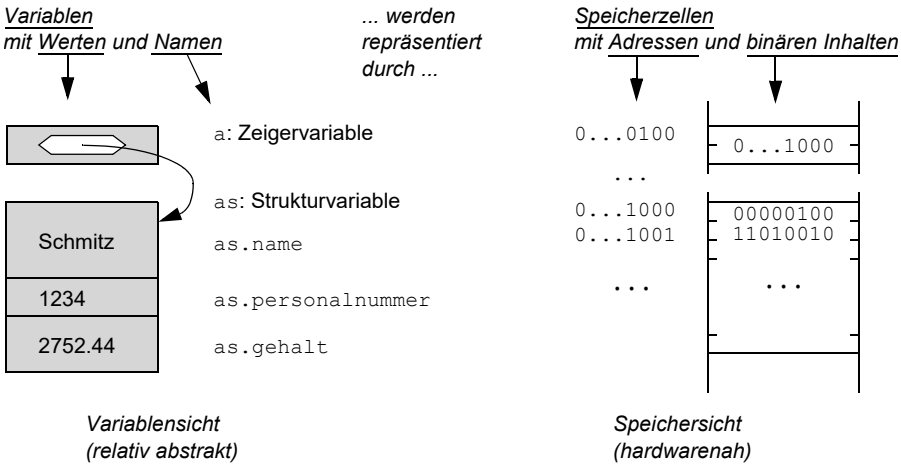


Abbildung 5.3 Zeiger und Zeigervariablen in C – Variablensicht vs. Speichersicht

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).

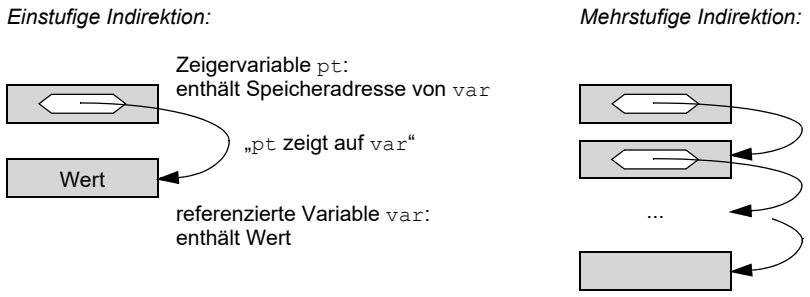


Abbildung 5.4 Indirektion mit Zeigervariablen

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 Variablendeklaration gehört stets zu *einem* Variablennamen. 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 Zeigervariablen 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 Zeigervariablen 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.

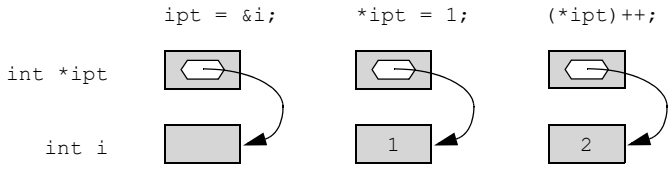


Abbildung 5.5 Basisoperationen auf Zeigervariablen

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):

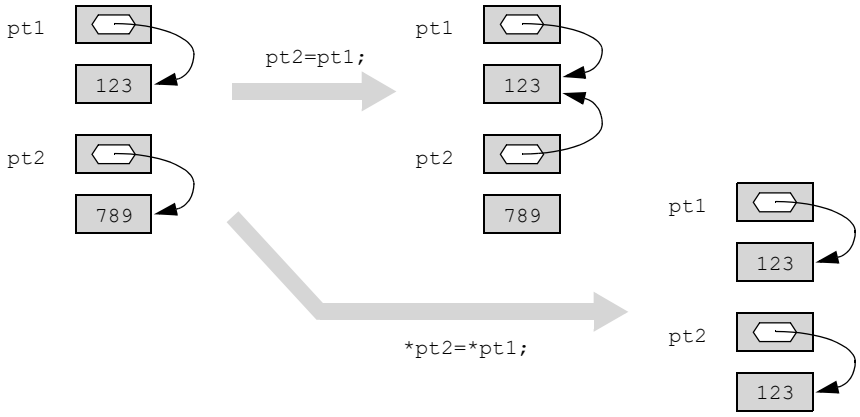


Abbildung 5.6 Zuweisung an Zeigervariable versus Zuweisung an referenzierte Variable

- 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. Anderenfalls 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 waehlen: 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 Variablennamen 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.

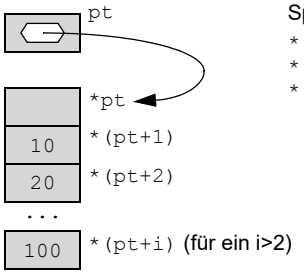
5.3 Adressarithmetik

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).



Speicherzustand nach:

```
*(pt+1) = 10;
*(pt+2) = *(pt+1) + 10;
*(pt+i) = 100;
```

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).

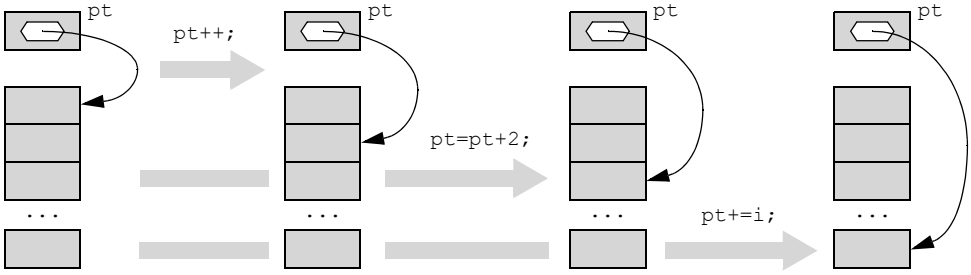


Abbildung 5.8 Zeigerarithmetik – Rechnen mit Adressen und Zuweisung an Zeigervariablen

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*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 tiefere Systemkenntnisse 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.

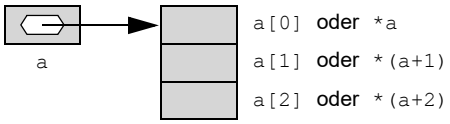


Abbildung 5.9 Adressierung von Arrays mit Index- oder mit Zeigerschreibweise

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