

12 Zeiger (Pointer)

Im Grunde sind Zeiger aber gar nicht so kompliziert, wie sie oft dargestellt werden. Zeiger sind im Prinzip nichts anderes als ganz normale Variablen, die statt Datenobjekten wie Zahlen, Zeichen oder Strukturen eben Adressen eines bestimmten Speicherbereichs beinhalten.

Was können Sie mit Zeigern auf Adressen so alles machen? Hierzu ein kleiner Überblick über die Anwendungsgebiete von Zeigern:

- Speicherbereiche können dynamisch reserviert, verwaltet und wieder gelöscht werden.
- Mit Zeigern können Sie Datenobjekte direkt (*call-by-reference*) an Funktionen übergeben.
- Mit Zeigern lassen sich Funktionen als Argumente an andere Funktionen übergeben.
- Rekursive Datenstrukturen wie Listen und Bäume lassen sich fast nur mit Zeigern erstellen.
- Es lässt sich ein typenloser Zeiger (`void *`) definieren, womit Datenobjekte beliebigen Typs verarbeitet werden können.

Auf den nächsten Seiten erläutere ich erst einmal die Grundlagen der Zeiger (die häufig auch *Pointer* genannt werden). Im Laufe des Buchs werden dann die zuvor genannten Punkte besprochen. Ich empfehle Ihnen, sich für dieses Kapitel viel Zeit zu nehmen. Es stellt auf jeden Fall die Grundlage für den Fortgang des Buchs und Ihre Karriere als C-Programmierer dar.

12.1 Zeiger deklarieren

Die Deklaration eines Zeigers hat die folgende Syntax:

```
Datentyp *zeigervariable;
```

Der `Datentyp` des Zeigers muss vom selben Datentyp wie der sein, auf den er zeigt (referenziert).

Hinweis

Wenn ich im Weiteren von »auf etwas zeigen« spreche, ist damit natürlich gemeint, dass auf einen bestimmten Speicherbereich (eine Adresse im Arbeitsspeicher) referenziert wird.

Das Sternchen vor `zeigervariable` kennzeichnet den Datentyp als Zeiger. Im Fachjargon heißt dieser Operator *Indirektionsoperator*. Die Position für das Sternchen befindet sich zwischen dem Datentyp und dem Zeigernamen. Beispiel:

```
int *zeiger1;
int* zeiger2;
char *zeiger3;
char* zeiger4;
float *zeiger5;
```

Hier sehen Sie zwei verschiedene Schreibweisen, wobei beide richtig sind; es hat sich aber folgende eingebürgert:

```
int *zeiger1;
```

```
int *zeiger2;  
int *zeiger3;
```

Mit dieser Schreibweise wird der gemachte Fehler deutlicher:

```
int *zeiger1 , zeiger2;
```

Hier wurde nur ein Zeiger deklariert, was auch recht schnell zu sehen ist. Bei der folgenden Schreibweise ist dieser Fehler nicht mehr so eindeutig zu erkennen:

```
int* zeiger1 , zeiger2;
```

Hier könnte man fälschlicherweise annehmen, es seien zwei Zeiger deklariert worden. Am besten verwenden Sie also die übliche Schreibweise. Damit können Sie sich einige Probleme ersparen.

12.2 Zeiger initialisieren

Hier beginnt eine gefährliche Operation. Wird im Programm ein Zeiger verwendet, der zuvor nicht initialisiert wurde, kann dies zu schwerwiegenden Fehlern führen – sogar bis zum Absturz eines Betriebssystems (bei 16-Bit-Systemen). Die Gefahr ist, dass bei einem Zeiger, der nicht mit einer gültigen Adresse initialisiert wurde und auf den jetzt zurückgegriffen werden soll, stattdessen einfach auf irgendeine Adresse im Arbeitsspeicher zurückgegriffen wird. Wenn sich in diesem Speicherbereich wichtige Daten oder Programme bei der Ausführung befinden, kommt es logischerweise zu Problemen.

Um das Prinzip der Zeiger zu verstehen, müssen Sie nochmals zurück zu den normalen Datentypen springen, beispielsweise zu folgender Initialisierung:

```
int x = 5;
```

Durch diese Initialisierung ergibt sich im Arbeitsspeicher folgendes Bild:

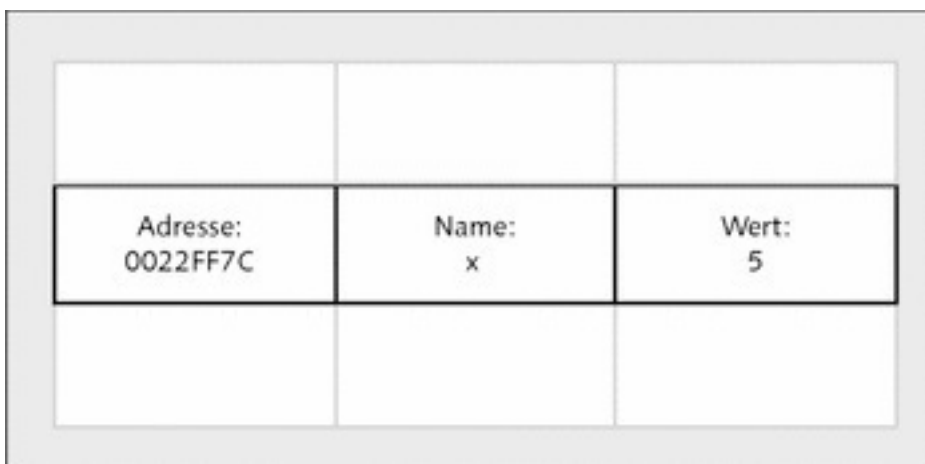


Abbildung 12.1 Darstellung einer Variablen im Arbeitsspeicher

Die Adresse ist eine erfundene Adresse im Arbeitsspeicher, auf die Sie keinen Einfluss haben. Diese wird vom System beim Start des Programms vergeben. Damit der Rechner weiß, von wo er den Wert einer Variablen auslesen soll, wird eine Adresse benötigt. Ebenso sieht es mit der Initialisierung einer Variablen aus, falls dieser ein Wert zugewiesen wird. Der Name einer Variablen ist der Name, den Sie bei der Deklaration selbst festgelegt haben. Der Wert 5 wurde zu Beginn des Programms definiert. Dieser Block oben hat eine Speichergröße von vier Bytes (`int` = vier Bytes oder, auf 16-Bit-Systemen, zwei Bytes).

Hundertprozentig stimmt diese Analyse eines Datentyps nicht. Es gibt noch einige weitere Attribute, die ein Datentyp besitzt, und zwar folgende:

- Wann bekommt die Variable ihren Speicherplatz zugeordnet? (Das ist abhängig vom Schlüsselwort `static` oder `auto`.)
- Wie lange bleibt der Speicherort dieser Variablen gültig?
- Wer kann diesen Wert ändern bzw. abrufen? (Das ist abhängig vom Gültigkeitsbereich und von der Sichtbarkeit der Variablen: global, lokal, Schlüsselwort `const`.)
- Wann wird die Variable gespeichert? (Das ist abhängig vom Schlüsselwort `volatile`.)

Dies dient allerdings hier nur zur Information, denn die Dinge sollten jetzt nicht komplizierter gemacht werden, als sie sind.

Benötigen Sie die Adresse einer Variablen im Arbeitsspeicher, dann kann diese mit dem Formatzeichen `%p` und dem Adressoperator `&` abgefragt und ausgegeben werden:

```
/* ptr1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x = 5;

    printf("Die Adresse von x ist %p \n", &x);
    return EXIT_SUCCESS;
}
```

In diesem Beispiel wurde mithilfe des Adressoperators und des Formatzeichens `%p` die aktuelle Speicheradresse der Variablen `x` ausgegeben.

Jetzt ist auch klar, warum `scanf()` eine Fehlermeldung ausgibt, wenn kein Adressoperator mit angegeben wird:

```
scanf("%d", x); /* Wohin damit ...??? */
```

Das wäre dasselbe, als wenn der Postbote einen Brief zustellen soll, auf dem sich keine Anschrift befindet. Der Brief wird niemals sein Ziel erreichen. Genauso läuft es in Ihrem PC ab, egal ob Sie jetzt ein Computerspiel spielen oder ein Textverarbeitungsprogramm verwenden. Jedes Speicherobjekt, das Sie definieren, hat eine Adresse, einen Namen und eine bestimmte Speichergröße (je nach Datentyp). Der Wert ist der einzige dieser vier Angaben, der zur Laufzeit festgelegt oder verändert werden kann.

Wie kann jetzt einem Zeiger die Adresse einer Variablen übergeben werden? Dies soll das folgende Beispiel demonstrieren:

```
/* ptr2.c */
#include <stdio.h>

#include <stdlib.h>

int main(void) {
    int abfrage;
    int Kapitel1 = 5;
    int Kapitel2 = 60;
    int Kapitel3 = 166;
    int Nachtrag = 233;
    int *Verzeichnis; /* Zeiger */

    do {
```

```

printf("\tINDEXREGISTER VOM BUCH\n");
printf("\t*****\n\n");
printf("\t-1- Kapitel 1\n");
printf("\t-2- Kapitel 2\n");
printf("\t-3- Kapitel 3\n");
printf("\t-4- Nachtrag\n");
printf("\t-5- Ende\n");
printf("\n");
printf("\tAuswahl : ");
scanf("%d",&abfrage);
printf("\tKapitel %d finden Sie auf ",abfrage);

switch(abfrage) {
    case 1 : Verzeichnis =& Kapitel1;
             printf("Seite %d\n", *Verzeichnis);
             break;
    case 2 : Verzeichnis =& Kapitel2;
             printf("Seite %d\n", *Verzeichnis);
             break;
    case 3 : Verzeichnis =& Kapitel3;
             printf("Seite %d\n", *Verzeichnis);
             break;
    case 4 : Verzeichnis =& Nachtrag;
             printf("Seite %d\n", *Verzeichnis);
             break;
    default : printf("Seite ???\n");
             break;
}
} while(abfrage < 5);
return EXIT_SUCCESS;
}

```

Der Zeiger des Programms ist:

```
int *Verzeichnis;
```

Hiermit wurde ein Zeiger mit dem Namen `Verzeichnis` deklariert. Bis zur `switch`-Verzweigung geschieht so weit nichts Neues. Aber dann finden Sie in der ersten `case`-Anweisung:

```
Verzeichnis =& Kapitel1;
```



Abbildung 12.2 Programm zur Verwendung der Zeiger in Aktion

Damit wird dem Zeiger `Verzeichnis` die Adresse der Variablen `Kapitel1` übergeben. Dies können Sie am Adressoperator `&` erkennen, der sich vor der Variablen `Kapitel1` befindet. Falls Sie den Adressoperator vor der Variablen `Kapitel1` vergessen, wird der Compiler das Programm nicht übersetzen, da ein Zeiger eine Adresse und nicht den Wert einer Variablen haben will.

Zu diesem Beispiel folgt ein kleiner Ausschnitt, der verdeutlicht, was im Speicher alles geschieht:

```
int Kapitel1 = 5;
int Kapitel2 = 60;
int Kapitel3 = 166;
int Nachtrag = 233;
int *Verzeichnis;
```


Adresse: 00000005	Name: Kapitel1	Wert: 5
Adresse: 0000003C	Name: Kapitel2	Wert: 60
Adresse: 000000A6	Name: Kapitel3	Wert: 166
Adresse: 000000E9	Name: Nachtrag	Wert: 233
Adresse: 00402000	Name: Verzeichnis	Adresse: -----

Abbildung 12.3 Darstellung im Arbeitsspeicher

Zunächst erfolgt beispielsweise die Adressübergabe der Variablen `Kapitel1` an den Zeiger `Verzeichnis` mit dem Adressoperator:

```
Verzeichnis = & Kapitel1;
```

Adresse: 00000005	Name: Kapitel1	Wert: 5
Adresse: 0000003C	Name: Kapitel2	Wert: 60
Adresse: 000000A6	Name: Kapitel3	Wert: 166
Adresse: 000000E9	Name: Nachtrag	Wert: 233
Adresse: 00402000	Name: Verzeichnis	Adresse: 00000005



The diagram shows a pointer arrow originating from the 'Verzeichnis' row (Name: Verzeichnis, Adresse: 00402000) and pointing to the 'Kapitel1' row (Name: Kapitel1, Adresse: 00000005). This indicates that the pointer variable 'Verzeichnis' now holds the memory address of 'Kapitel1'.

Abbildung 12.4 Der Zeiger verweist hier auf die Adresse der Variablen »Kapitel1«.

Daran lässt sich erkennen, wie der Zeiger `Verzeichnis` die Adresse der Variablen `Kapitel1` enthält. Ein wenig anders sieht es dann hiermit aus:

```
printf("Seite %d\n", *Verzeichnis);
```

Hier kommt zum ersten Mal der Indirektionsoperator (*) ins Spiel. Dieser dereferenziert den Wert der Adresse, mit der der Zeiger zuvor mit

```
Verzeichnis =& Kapitel1;
```

initialisiert wurde. Lassen Sie bei der Ausgabe einfach einmal den Indirektionsoperator weg:

```
printf("Seite %d\n", Verzeichnis); /* ohne '*' */
```

Übersetzen Sie dieses Programm erneut, und lassen Sie sich das Verzeichnis von Kapitel1 ausgeben. Es wird irgendeine Zahl ausgegeben, nur nicht die Zahl 5. Warum? Eine Umänderung der Zeile

```
printf("Seite %d\n", *Verzeichnis);
```

in

```
printf("Adressen %p %p\n", *Verzeichnis, Kapitel1);
```

zeigt mehr. Jetzt soll wieder das erste Kapitel bei der Abfrage verwendet werden. Danach müssten beide Male dieselben Adressen ausgegeben werden. Mit Verzeichnis =& Kapitel1 wurde doch nur die Adresse übergeben. Und im Zeiger selbst befindet sich auch nur die Adresse von Kapitel1. Ohne den Indirektionsoperator ist der Zeiger hier nutzlos. Nur mit diesem Operator können Sie auf den Inhalt einer Variablen mithilfe eines Zeigers zugreifen.

Wenn Sie den Zeiger jetzt auf Kapitel3 (Verzeichnis =& Kapitel3) verweisen lassen, ergibt sich folgender Stand im Arbeitsspeicher:

Adresse: 00000005	Name: Kapitel1	Wert: 5
Adresse: 0000003C	Name: Kapitel2	Wert: 60
Adresse: 000000A6	Name: Kapitel3	Wert: 166
Adresse: 000000E9	Name: Nachtrag	Wert: 233
Adresse: 00402000	Name: Verzeichnis	Adresse: 000000A6

Abbildung 12.5 Der Zeiger verweist jetzt auf die Adresse von »Kapitel3«.

Merke

Wird der Indirektionsoperator (*) vorangestellt, erkennen Sie, dass nicht auf den Zeiger zurückgegriffen werden soll, sondern auf das Datenobjekt, dessen Anfangsadresse sich im Zeiger befindet.

Zum besseren Verständnis folgt dazu ein weiteres Programm, das die Verwendung von Zeigern detaillierter darstellen soll. Es ist ein lehrreiches Beispiel, und es lohnt sich, es zu studieren:

```
/* ptr3.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int x=5;
    int *y;

    printf("Adresse x=%p, Wert x=%d\n", &x, x);

    /* Führt bei manchen Systemen zum Programmabsturz,
     * ggf. auskommentieren. */
    printf("Adresse *y=%p, Wert *y=%d(unsinn)\n", &y, *y);
    printf("\ny=&x;\n\n");

    /* y hat jetzt die Adresse von x. */
    y = &x;
    printf("Adresse x=%p, Wert x=%d\n", &x, x);
    printf("Adresse *y=%p, Wert *y=%d\n", &y, *y);
    printf("\nAdresse, auf die y zeigt, ist %p\n", y);
    printf("und das ist die Adresse von x = %p\n", &x);

    printf("\nACHTUNG!!!\n\n");
    *y=10;
    printf("*y=10\n\n");
    printf("Adresse x=%p, Wert x=%d\n", &x, x);
    printf("Adresse *y=%p, Wert *y=%d\n", &y, *y);
    printf("\nAdresse, auf die y zeigt, ist %p\n", y);
    printf("weiterhin die Adresse von x (%p)\n", &x);
    return EXIT_SUCCESS;
}
```



```
user@desktop: ~
Datei Bearbeiten Ansicht Terminal Beiter Hilfe
user@desktop:~$ ./ptr3
Adresse x=0xbff44ce0, Wert x=5
Adresse *y=0xbff44cdc, Wert *y=-15156339(unsinn)

y=&x;

Adresse x=0xbff44ce0, Wert x=5
Adresse *y=0xbff44cdc, Wert *y=5

Adresse, auf die y zeigt, ist 0xbff44ce0
und das ist die Adresse von x = 0xbff44ce0

ACHTUNG!!!

*y=10

Adresse x=0xbff44ce0, Wert x=10
Adresse *y=0xbff44cdc, Wert *y=10

Adresse, auf die y zeigt, ist 0xbff44ce0
weiterhin die Adresse von x (0xbff44ce0)
user@desktop:~$
```

Abbildung 12.6 Die Ausgabe des Programms unter Linux

Folgende Zeile dürfte Ihnen bei diesem Programm aufgefallen sein:

```
*y = 10;
```

Hiermit wird der Wert der Variablen x dereferenziert. Mit dieser Dereferenzierung kann jederzeit auf den Wert der Variablen x zugegriffen werden. Dadurch kann mithilfe eines Zeigers der Inhalt der Variablen verändert werden, und zwar so, als würden Sie direkt darauf zugreifen. Die folgenden Abbildungen verdeutlichen den Verlauf.

```
int x = 5;
int *y;
```

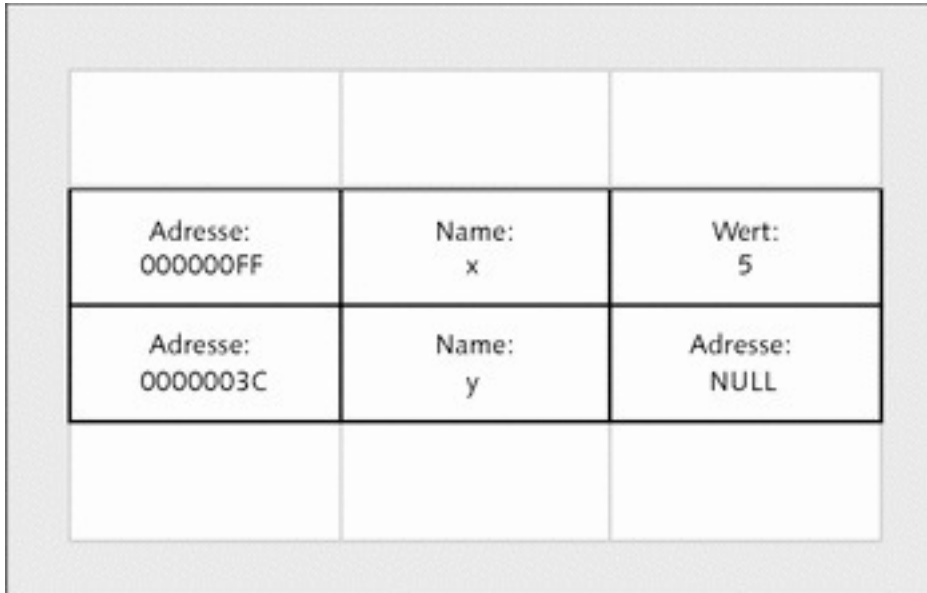


Abbildung 12.7 Speicheradressierung der Variablen »x« und des Zeigers »y«

```
y = &x;
```

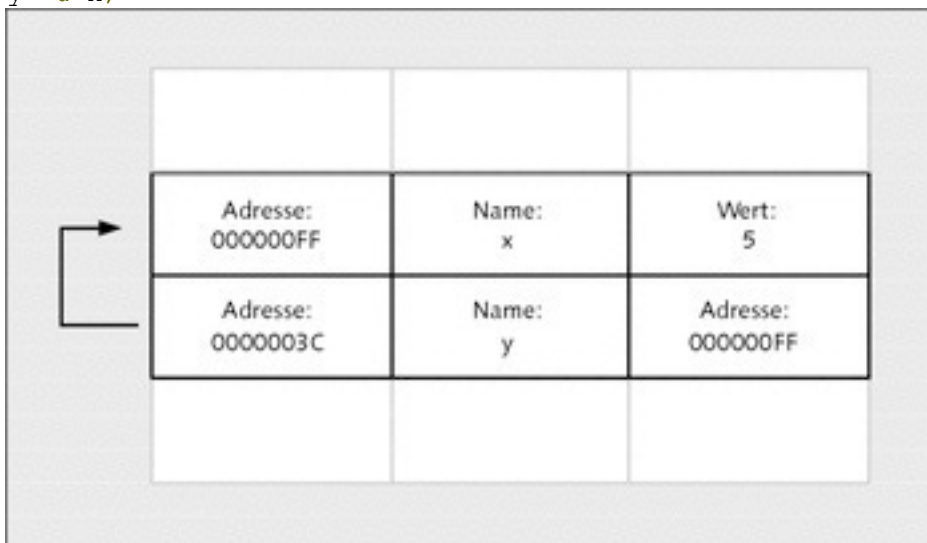


Abbildung 12.8 Der Zeiger »y« verweist jetzt auf die Adresse von Variable »x«.

```
*y = 10;
```

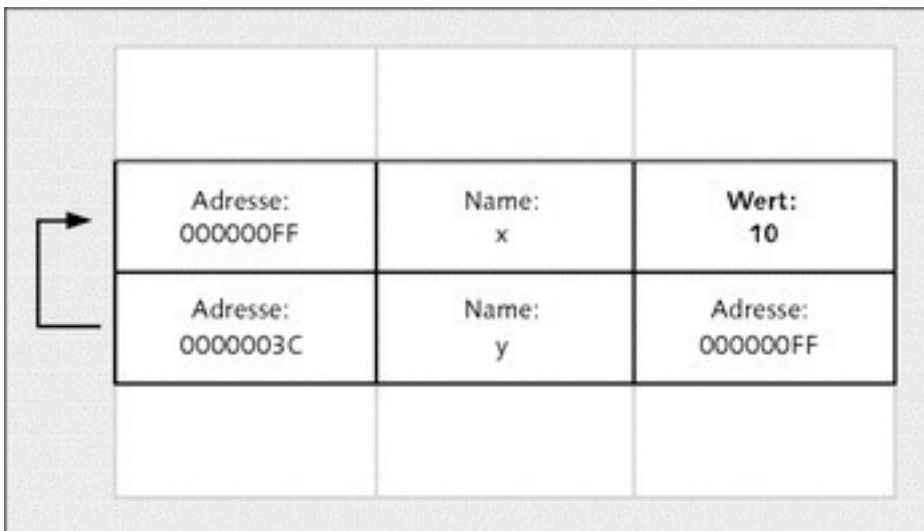



Abbildung 12.9 Dereferenzierung der Variablen »x«

Somit gilt: Wenn Sie mit dem Indirektionsoperator den Wert einer Variablen auslesen können, dann kann damit auch die Variable verändert werden. Das Wichtigste ist, dass Sie verstehen, dass einem Zeiger kein Wert übergeben wird, sondern eine Adresse (ich wiederhole mich), um anschließend mit dem Wert dieser Adresse zu arbeiten. Aber Achtung, das folgende Programm könnte böse Folgen haben:

```
/* ptr4.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *y;
    *y=10;

    printf("Der Wert von *y ist %d\n", *y);
    return EXIT_SUCCESS;
}
```

Dem Zeiger `y` wurde hier zuvor keine gültige Adresse zugewiesen. Dies bedeutet, dass dem Zeiger `y` beim Start des Programms eine Adresse zur Verfügung steht, die durch ein zufälliges Bitmuster vom Linker erzeugt wurde. Das Programm kann theoretisch sogar korrekt ablaufen. Irgendwann kann (wird) es jedoch ein vollkommen falsches Ergebnis zurückliefern. Auch könnte es sein, dass auf einen Speicherbereich zugegriffen wird, der bereits Daten beinhaltet. Dies könnte zu erheblichen Problemen bei der Programmausführung bis hin zum Absturz führen.

Solche Fehler können Sie vermeiden, indem Sie einen nicht verwendeten Zeiger mit `NULL` initialisieren und vor der Verwendung des Zeigers eine Überprüfung auf `NULL` durchführen. Der Wert oder genauer der Zeiger `NULL` ist meistens eine Konstante, die mit dem Wert 0 definiert ist:

```
#define NULL (void *)0
```

Einfach ausgedrückt handelt es sich bei diesem `NULL`-Zeiger um einen *Ich-zeige-auf-keine-gültige-Adresse*-Wert.

Hinweis

Auf `NULL` gehe ich in einem anderen Abschnitt (Abschnitt 14.3, »Das `NULL`-Mysterium«) nochmals etwas genauer ein. Allerdings empfehle ich Ihnen, hierzu auch die deutsche FAQ der `de.comp.lang.c` (<http://www.dclc-faq.de/inhalt.htm>) zu lesen. Hier wurde dem Thema `NULL`-Zeiger ein ganzes Kapitel gewidmet, da besonders Anfänger `NULL` zu sehr vergleichen – was aber nicht immer so sein muss. Theoretisch muss ein `NULL`-Zeiger nämlich kein Zeiger auf `null` sein, sondern kann auch eben nur als `0` definiert werden.

Hier sehen Sie das Erwähnte in der Praxis:

```
/* ptr5.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *y=NULL;      /* Zeiger mit NULL initialisieren */

    if(y == NULL) {
        printf("Der Zeiger besitzt keine gültige Adresse\n");
        return EXIT_FAILURE;
    }
    else
        *y = 10;
    return EXIT_SUCCESS;
}
```

Ein Tipp zu einer sichereren Überprüfung von:

```
if(y == NULL)
```

Es kann dabei schnell passieren, dass Sie statt einer Überprüfung auf `NULL` den Zeiger mit `NULL` initialisieren:

```
if(y = NULL) /* Fehler */
```

Mit folgender Überprüfung kann Ihnen dieser Fehler nicht mehr unterlaufen:

```
if(NULL == y)
```

Denn sollten Sie `NULL` den Zeiger `y` zuweisen wollen, wird der Compiler das Programm nicht übersetzen, da dies rein syntaktisch falsch ist.

Natürlich geht dies auch umgekehrt. Sie können einer normalen Variablen auch den Wert eines Zeigers übergeben, auf den dieser zeigt. Beispiel:

```
/* ptr6.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *ptr;
    int var=10, tmp;

    /* ptr zeigt auf Adresse von var. */
    ptr = & var;
    /* Variable tmp bekommt den Wert, den ptr dereferenziert. */
    tmp = *ptr;      /* tmp=10 */
    *ptr = 100;      /* Inhalt von var wird verändert var=100. */
    if(var > 50)      /* Ist var größer als 50 ... ? */
        var = tmp;    /* ... wieder den alten Wert */
    printf("var=%d\t*ptr=%d\n", var, *ptr); /* var=10    *ptr=10 */
    return EXIT_SUCCESS;
}
```

Wichtig ist allerdings dabei, dass Sie den Indirektionsoperator verwenden. Denn dieser dereferenziert den Wert, auf den der Zeiger zeigt:

```
tmp = *ptr; /* tmp=10 */
```

Sollten Sie den Indirektionsoperator vergessen, lässt sich das Programm ohnehin nicht übersetzen, denn es würde ja versucht werden, der Variablen `tmp` eine Adresse zu übergeben.

Hier ein schneller Überblick zum Zugriff und zur Dereferenzierung von Zeigern:

```
// Deklaration
int *ptr;
int var, var2;

// Initialisieren: ptr bekommt die Adresse von var.
ptr =& var;

// Dereferenzierung : var bekommt den Wert 100 zugewiesen.
*ptr=100;

// var2 mit demselben Wert wie var initialisieren
var2 = *ptr;

*ptr+=100;      // Dereferenzierung: var wird um 100 erhöht.
(*ptr)++;      // Dereferenzierung: var hat jetzt den Wert 201.
(*ptr)--;      // var hat wieder den Wert 200.
ptr=&var2;      // ptr zeigt auf var2.

printf("%d", *ptr); // Gibt Wert von var2 aus.
printf("%p", &ptr); // Gibt Adresse von ptr aus.
printf("%p", ptr);  // Gibt Adresse von var2 aus.
```

Sicherlich sind Ihnen im Beispiel auch die Klammern bei den Zeilen

```
(*ptr)++;      // Dereferenzierung: var hat jetzt den Wert 201.
(*ptr)--;      // var hat wieder den Wert 200.
```

aufgefallen. Diese waren nötig, da die Operatoren `++` und `--` einen höheren Rang haben als der Indirektionsoperator `(*)` (siehe Anhang A.1, »Rangfolge der Operatoren«). Wenn Sie diese Klammerung vergessen, kann das fatale Folgen haben.

12.2.1 Speichergröße von Zeigern

Ich komme bei der Speicherverwaltung zwar nochmals genauer auf dieses Thema zurück, aber es soll hier schon einmal kurz erwähnt werden. Die Größe eines Zeigers hängt nicht von dem Datentyp ab, auf den dieser verweist. Das ist schließlich nicht notwendig, denn Zeiger sollen ja keine Werte, sondern Adressen speichern. Und zur Speicherung von Adressen werden in der Regel zwei oder vier Bytes benötigt. Der Beweis:

```
/* ptr7.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char    *v;
    int     *w;
    float   *x;
    double  *y;
    void    *z;
```

```

printf("%d\t %d\t %d\t %d\t %d \n",
    sizeof(v), sizeof(w), sizeof(x), sizeof(y), sizeof(z));
return EXIT_SUCCESS;
}

```

Zeiger auf 64-Bit-Architekturen

Auf 64-Bit-Architekturen mit einem 64-Bit-Betriebssystem und dem LP64-Typenmodell ist ein Zeiger (wie auch der Datentyp `long`) üblicherweise auch 64 Bit breit und somit 8 Bytes.

Sicherlich stellt sich die Frage, warum man Zeiger dann überhaupt typisieren sollte, wenn der Speicherverbrauch immer gleich ist. Dies ist ziemlich wichtig in C. Nur dadurch lässt sich in C die Zeigerarithmetik realisieren. Denn nur durch das Wissen um die Speichergröße des assoziierten Typs kann die Adresse des Vorgänger- oder Nachfolgeelementes berechnet werden. Darüber hinaus ermöglicht die Typisierung von Zeigern dem Compiler, Verletzungen der Typkompatibilität zu erkennen.

Hinweis

Es gibt in der Tat Zeiger, die keinem Typ zugeordnet sind (`void`-Zeiger; siehe Abschnitt 12.11, »void-Zeiger«). Diese Zeiger können allerdings nicht dereferenziert, inkrementiert oder dekrementiert werden.

12.3 Zeigerarithmetik

Folgende Rechenoperationen können mit einem Zeiger und auf dessen Adresse verwendet werden:

- Ganzzahlwerte erhöhen
- Ganzzahlwerte verringern
- inkrementieren
- dekrementieren

Wenn Sie Folgendes eingeben würden:

```

int *ptr;
int wert;

ptr =& wert;
ptr += 10;

```

Auf welche Adresse zeigt dann der Zeiger `ptr`? 10 Bytes von der Variablen `wert` entfernt? Nein, ein Zeiger wird immer um den Wert der Größe des Datentyps erhöht bzw. heruntergezählt. Auf einem 32-Bit-System würde der Zeiger auf eine Stelle verweisen, die 40 Bytes von der Anfangsadresse der Variablen `wert` entfernt ist.

Solch eine Erhöhung der Adresse ist ohnehin sehr gefährlich, da der Zeiger danach höchstwahrscheinlich nicht mehr auf einen reservierten Speicherbereich zeigt.

Des Weiteren sind bei Verwendung eines Zeigers natürlich auch die Vergleichsoperatoren `<`, `>`, `!=`, `==`, `<=` und `=>` erlaubt. Die Verwendung ist aber hierbei nur sinnvoll, wenn die Zeiger auf Elemente eines Arrays zeigen.

12.4 Zeiger, die auf andere Zeiger verweisen

Zeiger können letztendlich auch auf andere Zeiger verweisen:

```
int *ptr1;  
int *ptr2;
```

```
ptr1 = ptr2;
```

In diesem Fall zeigen beide Zeiger auf dieselbe Adresse. Dies ist ein wenig verwirrend. Daher folgt ein kleines Beispiel dazu:

```
/* ptr8.c */  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void) {  
    int wert=10;  
    int *ptr1;  
    int *ptr2;  
  
    ptr1 = &wert;  
    ptr2 = ptr1;  
  
    printf("ptr1 verweist auf %p\n", ptr1);  
    printf("Wert in %p ist %d\n\n", ptr1, *ptr1);  
  
    printf("ptr2 verweist auf %p\n", ptr2);  
    printf("Wert in %p ist %d\n\n", ptr2, *ptr2);  
  
    printf("Adresse von ptr1 : %p\n", &ptr1);  
    printf("Adresse von ptr2 : %p\n", &ptr2);  
    printf("Adresse von wert : %p\n\n", &wert);  
  
    printf("ptr1 [%p] -> [%p] = [%d]\n", &ptr1, ptr1, *ptr1);  
  
    printf("ptr2 [%p] -> [%p] = [%d]\n", &ptr2, ptr2, *ptr2);  
    return EXIT_SUCCESS;  
}
```

Bei diesem Programm verweisen beide Zeiger auf denselben Wert (genauer gesagt auf dieselbe Adresse), nämlich auf die Adresse der Variablen `wert`. Mit `ptr2 = ptr1` bekommt `ptr2` dieselbe Adresse zugewiesen, auf die schon `ptr1` zeigt. Auffällig ist hierbei auch, dass kein Adressoperator verwendet wird. Dieser wird nicht benötigt, da der Wert eines Zeigers schon eine Adresse ist und ein Zeiger auch einen Wert als Adresse erwartet.

Würde jetzt

```
*ptr1 = 11;
```

im Programm eingefügt, würde der Wert der Variablen `wert` auf 11 geändert, und somit wäre die Ausgabe des Programms immer 11. Falls Sie

```
wert = 20;
```

schreiben, werden auch die beiden Zeiger (mit Verwendung des Dereferenzierungsoperators) 20 ausgeben, da die Zeiger ja weiterhin auf die Speicheradresse der Variablen `wert` zeigen.

Adresse: 00000005	Name: wert	Wert: 10
Adresse: 000000FF	Name: ptr1	Adresse: 00000005
Adresse: 0000003C	Name: ptr2	Adresse: 00000005

Abbildung 12.10 Ein Zeiger bekommt die Adresse eines anderen Zeigers.

12.4.1 Subtraktion zweier Zeiger

In der Standard-Headerdatei `<stddef.h>` befindet sich ein primitiver Datentyp `ptrdiff_t`, der meist mit `int` deklariert ist. Dieser wird verwendet, um das Ergebnis aus der Subtraktion zweier Zeiger zurückzugeben. Diese Subtraktion wird verwendet, um zu berechnen, wie weit zwei Zeiger zum Beispiel in einem Vektorelement voneinander entfernt sind. Näheres über den Zusammenhang von Arrays bzw. Strings und Zeigern können Sie in Abschnitt 12.7 lesen. Hier sehen Sie ein einfaches Listing:

```
/* ptr9.c */
#include <stdio.h>
#include <stddef.h> /* für ptrdiff_t */
#include <stdlib.h>

int main(void) {
    char *ptr1, *ptr2;
    ptrdiff_t diff; /* Primitiver Datentyp */
    char string[] = { "Hallo Welt\n" };

    /* ptr2 auf Anfangsadresse von string */
    ptr2 = string;
    /* ptr1 6 Bytes weiter von der Adresse ptr2 platzieren */
    ptr1 = ptr2 + 6;
    /* Wie weit liegen beide Zeiger voneinander entfernt? */
    diff = ptr1 - ptr2;
    // Nach dem neuen C99-Standard können Sie die formatierte
    // Ausgabe auch mit dem Argumenttyp-Modifikator t für
    // ptrdiff_t verwenden, also %td anstatt %d.
    printf("Differenz der beiden Zeiger : %d Bytes\n", diff);
    printf("%s", ptr1); /* Welt */
    printf("%s", ptr2); /* Hallo Welt */
    return EXIT_SUCCESS;
}
```

Mit der Zeile

```
ptr1 = ptr2 + 6;
```

lassen Sie den Zeiger `ptr1` auf die Adresse `string[6]` zeigen. `ptr2` zeigt hingegen weiterhin auf die Anfangsadresse des Strings (`string[0]`). Die Differenz dieser beiden Zeiger beträgt 6 Bytes, da diese so weit voneinander entfernt sind.

12.5 Typensicherung bei der Dereferenzierung

Zeiger sind in C streng typisiert. Sie können einen Zeiger vom Datentyp `int` nicht auf die Adresse eines `double`-Werts zeigen lassen, wie im folgenden Beispiel zu sehen ist:

```
/* ptr10.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *int_ptr;
    double double_wert=999.999;

    int_ptr=&double_wert;
    printf("*int_ptr=%d double=%f\n", *int_ptr, double_wert);
    return EXIT_SUCCESS;
}
```

Die Ausgabe des Zeigers wird irgendwelchen Unsinn ergeben. Es ist aber auch möglich, die Typensicherung durch explizite Typumwandlung oder über einen `void`-Zeiger zu umgehen. Aber dazu später mehr.

12.6 Zeiger als Funktionsparameter (call-by-reference)

Funktionen, die mit einem oder mehreren Parametern definiert werden und mit `return` einen Rückgabewert zurückliefern, haben wir bereits verwendet (*call-by-value*). Der Nachteil dieser Methode ist, dass bei jedem Aufruf erst einmal alle Parameter kopiert werden müssen, sodass diese Variablen der Funktion anschließend als lokale Variablen zur Verfügung stehen. Betrachten Sie beispielsweise folgendes Programm:

```
/* ptr11.c */
#include <stdio.h>
#include <stdlib.h>
#define PI 3.141592f

float kreisflaeche(float wert) {
    return (wert = wert * wert * PI);
}

int main(void) {
    float radius, flaeche;

    printf("Berechnung einer Kreisfläche!!\n\n");
    printf("Bitte den Radius eingeben : ");
    scanf("%f", &radius);
    flaeche = kreisflaeche(radius);
    printf("\nDie Kreisfläche beträgt : %f\n", flaeche);
    return EXIT_SUCCESS;
}
```

In solch einem Fall bietet es sich an, statt der Variablen `radius` einfach nur die Adresse der Variablen als Argument zu übergeben. Die Übergabe von Adressen als Argument einer Funktion wird *call-by-reference* genannt. Das Prinzip sehen Sie im abgeänderten Programmbeispiel:

```

/* ptr12.c */
#include <stdio.h>
#include <stdlib.h>
#define PI 3.141592f

void kreisflaeche(float *wert) {
    *wert = ( (*wert) * (*wert) * PI );
}

int main(void) {
    float radius;

    printf("Berechnung einer Kreisfläche!!\n\n");
    printf("Bitte den Radius eingeben : ");
    scanf("%f", &radius);
    /* Adresse von radius als Argument an kreisflaeche() */
    kreisflaeche(&radius);
    printf("\nDie Kreisfläche beträgt : %f\n", radius);
    return EXIT_SUCCESS;
}

```

Statt einer Variablen als Argument wurde in diesem Beispiel einfach die Adresse der Variablen `radius` übergeben. Bildlich können Sie sich das Prinzip so vorstellen:

```
float radius; /* Wir geben einfach mal 5.5 ein */
```

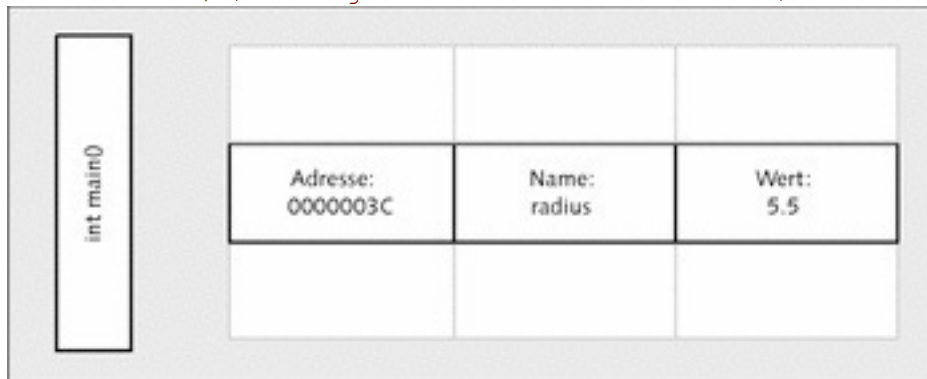


Abbildung 12.11 Die Variable »radius« bekommt den Wert 5.5.

```
kreisflaeche(&radius);
```

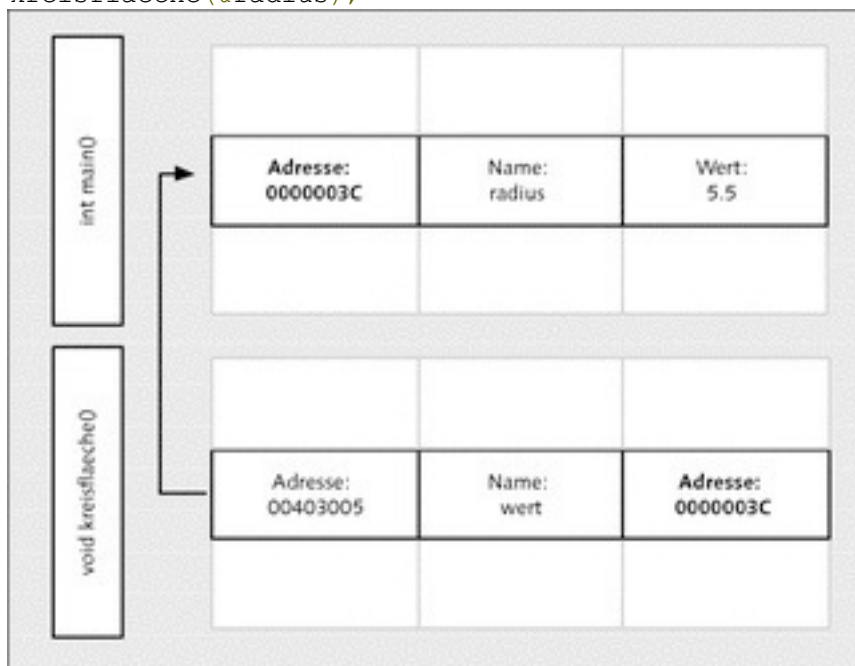


Abbildung 12.12 Adresse als Funktionsparameter (call-by-reference)


```
*wert = (*wert) * (*wert) * PI;
```

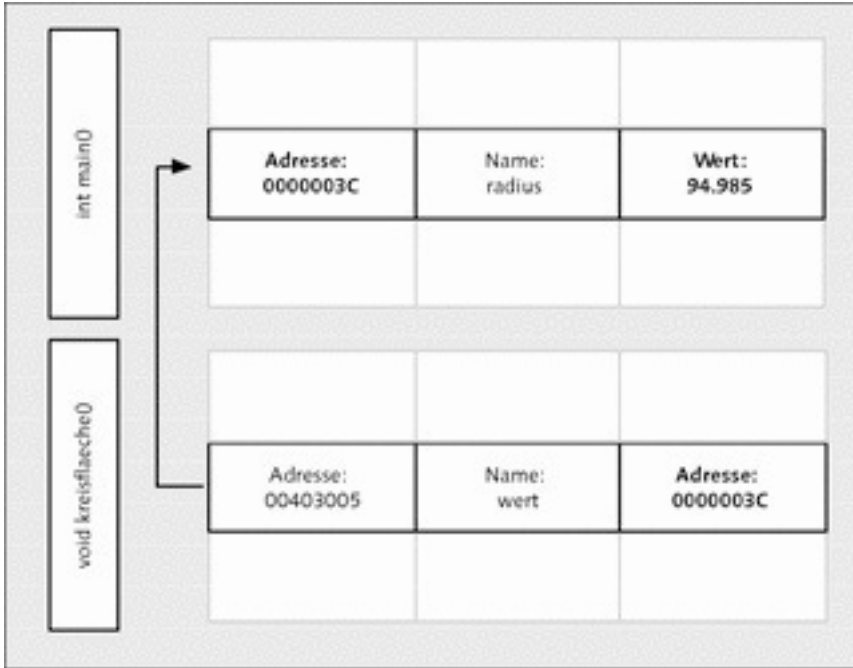


Abbildung 12.13 In der Funktion wird mit der Referenz gerechnet.

In diesem Beispiel übergeben Sie mit dem Funktionsaufruf

```
kreisflaeche(&radius);
```

die Adresse der Variablen `radius` als Referenz an die Funktion:

```
void kreisflaeche(float *wert)
```

In der Funktion `kreisflaeche()` befindet sich als Parameter ein Zeiger namens `wert` vom Typ `float`. Der Zeiger `wert` in der Funktion `kreisflaeche()` bekommt durch den Funktionsaufruf `kreisflaeche(&radius)` die Adresse der Variablen `radius` zugewiesen. Jetzt, da der Zeiger in der Funktion die Adresse kennt, kann mit dem Dereferenzierungsoperator, der ja auf den Wert von `radius` zeigt, gerechnet werden:

```
*wert= (*wert) * (*wert) * PI;
```

Die Klammerung bei der Berechnung kann auch weggelassen werden. Sie dient der besseren Übersicht. Ohne den Dereferenzierungsoperator würde lediglich mit einer Adresse gerechnet werden. Die meisten Compiler geben ohnehin eine Fehlermeldung aus.

Es wird Ihnen sicherlich aufgefallen sein, dass bei der Funktion keine Rückgabe mehr mit `return` erfolgt und der Rückgabewert `void` ist. Das liegt daran, dass bei jeder Neuübersetzung des Programms jeder Variablen eine Adresse zugewiesen wird, die sich während der Laufzeit des Programms nicht mehr ändern lässt. Da die Funktion die Adresse der Variablen `radius` bekommt, wird auch in der Funktion der Wert dieser Variablen verändert. Weil hierbei mit der Variablen `radius` und dem Zeiger `wert` mit derselben Adresse gearbeitet wird, entfällt eine Rückgabe an den Aufrufer.

12.6.1 Zeiger als Rückgabewert

Natürlich ist es auch möglich, einen Zeiger als Rückgabewert einer Funktion zu deklarieren, so wie das bei vielen Funktionen der Standard-Bibliothek gemacht wird. Funktionen, die mit einem Zeiger als Rückgabewert deklariert sind, geben

logischerweise auch nur die Anfangsadresse des Rückgabetyps zurück. Die Syntax dazu sieht folgendermaßen aus:

```
Zeiger_Rückgabetypp *Funktionsname(Parameter)
```

Das Verfahren mit Zeigern als Rückgabewert von Funktionen wird häufig bei Strings oder Strukturen verwendet und ist eine effiziente Methode, Datenobjekte aus einer Funktion zurückzugeben. Speziell bei Strings ist dies die einzige Möglichkeit, eine ganze Zeichenkette aus einer Funktion zurückzugeben. Natürlich ist sie es nicht wirklich. Tatsächlich wird ja nur die Anfangsadresse, also das erste Zeichen an den Aufrufer zurückgegeben. Hierzu ein recht einfaches Beispiel:

```
/* ptr13.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 255

char *eingabe(char *str) {
    char input[MAX];

    printf("Bitte \"%s\" eingeben: ", str);
    fgets(input, MAX, stdin);
    return strtok(input, "\n");
}

int main(void) {
    char *ptr;

    ptr = eingabe("Vorname");
    printf("Hallo %s\n", ptr);
    ptr = eingabe("Nachname");
    printf("%s, interessanter Nachname\n", ptr);
    return EXIT_SUCCESS;
}
```

Der Funktion `eingabe()` wird hierbei als Argument die Adresse eines Strings übergeben. In der Funktion werden Sie aufgefordert, einen Namen einzugeben. Die Anfangsadresse des Strings geben Sie mit folgender Zeile zurück:

```
return strtok(input, "\n");
```

Die Funktion `strtok()` liefert ja selbst als Rückgabewert einen `char`-Zeiger zurück. Da die Funktion `fgets()` beim Einlesen von der Standardeingabe das Newline-Zeichen mit einliest, haben Sie hierbei gleich zwei Fliegen mit einer Klappe geschlagen. Das Newline-Zeichen wird mit `strtok()` entfernt, und die Funktion liefert auch gleich die Anfangsadresse des Strings `input` als Rückgabewert zurück, den Sie direkt mit `return` weiterverwenden. Doch Vorsicht, Folgendes funktioniert nicht:

```
char *eingabe(char *str) {
    char input[MAX];

    printf("Bitte \"%s\" eingeben: ", str);
    fgets(input, MAX, stdin);
    return input;
}
```

Normalerweise sollte hier der Compiler schon melden, dass etwas nicht stimmt. Spätestens aber dann, wenn Sie das Beispiel ausführen, werden Sie feststellen, dass anstatt des Strings, den Sie in der Funktion `eingabe()` eingegeben haben, nur Datenmüll ausgegeben wird.

Wenn Sie die Geschichte mit den Funktionen und dem Stack (in Abschnitt 9.20.1, »Exkurs: Stack«) gelesen haben, wissen Sie, dass beim Aufruf einer Funktion ein

Stack verwendet wird, auf dem alle benötigten Daten einer Funktion (die Parameter, die lokalen Variablen und die Rücksprungadresse) angelegt werden (die Rede ist vom *Stack-Frame*). Dieser Stack-Frame bleibt nun so lange bestehen, bis sich die Funktion wieder beendet.

Die Funktion `eingabe()` gibt eben einen solchen Speicherbereich (lokales Feld) zurück, der sich ebenfalls auf diesem Stack-Frame befindet bzw. befand – und somit bei Beendigung der Funktion nicht mehr vorhanden ist. Wollen Sie also einen Zeiger auf einen Speicherbereich zurückgeben, haben Sie folgende Möglichkeiten. Sie verwenden

- einen statischen Puffer (`static`),
- einen beim Aufruf der Funktion als Argument übergebenen Puffer oder
- einen mittels `malloc()` reservierten Speicher (siehe Kapitel 16, »Ein-/Ausgabe-Funktionen«).

Das folgende Beispiel soll alle drei Möglichkeiten demonstrieren:

```
/* ptr14.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Fehler: Funktion gibt die Adresse
 * einer lokalen Variablen zurück. */
char *test1(void) {
    char buffer[10];
    strcpy(buffer, "testwert");
    return buffer;
}

/* Möglichkeit1: Statische Variable */
char *test2(void) {
    static char buffer[10];
    strcpy(buffer, "testwert");
    return buffer;
}

/* Möglichkeit2: Speicher vom Heap verwenden */
char *test3(void) {
    char *buffer = (char *) malloc(10);
    strcpy(buffer, "testwert");
    return buffer;
}

/* Möglichkeit3: Einen Zeiger als Argument übergeben */
char *test4(char *ptr) {
    char buffer[10];
    ptr = buffer;
    strcpy(buffer, "testwert");
    return ptr;
}

int main(void) {
    char *ptr;

    ptr = test1();
    printf("test1: %s\n", ptr); // meistens Datenmüll
    ptr = test2();
    printf("test2: %s\n", ptr);
    ptr = test3();
    printf("test3: %s\n", ptr);
    test4(ptr);
}
```

```

    printf("test4: %s\n", ptr);
    return EXIT_SUCCESS;
}

```

Hinweis

Bitte beachten Sie außerdem, dass die Verwendung eines statischen Puffers (`static`) nicht mehr funktioniert, wenn eine Funktion rekursiv aufgerufen wird!

12.7 Array und Zeiger

Um Irrtümer gleich zu vermeiden: Arrays und Zeiger sind nicht das Gleiche, auch wenn dies im Verlauf dieses Kapitels den Anschein hat. Ein Zeiger ist die Adresse einer Adresse, während ein Array-Name nur eine Adresse darstellt. Dieser Irrtum, dass Array und Zeiger dasselbe sind, beruht häufig darauf, dass Array- und Zeigerdeklarationen als formale Parameter einer Funktion austauschbar sind, weil hierbei (und nur hierbei) ein Array in einen Zeiger zerfällt. Diese automatische »Vereinfachung« erschwert einem Anfänger das Verständnis jedoch. Weiterhin verstärkt wird dieses Missverständnis, wenn ein Zeiger auf einem Speicherblock, der mit `malloc()` dynamisch Speicher reserviert, wie ein Array verwendet wird (da der Speicherblock auch mit `[]` verwendet werden kann).

Internes

Ein Array belegt zum Programmstart automatisch einen Speicherbereich, der nicht mehr verschoben oder in der Größe verändert werden kann. Einem Zeiger hingegen muss man einen Wert zuweisen, damit dieser auch auf einen belegten Speicher zeigt. Außerdem kann der »Wert« eines Zeigers später nach Belieben einem anderen »Wert« (Speicherobjekt) zugewiesen werden. Ein Zeiger muss außerdem nicht nur auf den Anfang eines Speicherblocks zeigen.

Zuerst folgt ein Beispiel, wie mit Zeigern auf ein Array zugegriffen werden kann:

```

/* arr_ptr1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int element[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };
    int *ptr;
    int i;

    ptr = element;
    printf("Der Wert, auf den *ptr zeigt, ist %d\n", *ptr);
    printf("Durch *ptr+1 zeigt er jetzt auf %d\n", *(ptr+1));
    printf("** (ptr+3) = %d\n", *(ptr+3));
    printf("\nJetzt alle zusammen : \n");
    for(i=0; i<8; i++)
        printf("element[%d]=%d \n", i, *(ptr+i));
    return EXIT_SUCCESS;
}

```

Durch die Anweisung

```
ptr = element;
```

wird dem Zeiger `ptr` die Adresse des Arrays `element` übergeben. Dies funktioniert ohne den Adressoperator, da laut ANSI-C-Standard der Array-Name immer als Zeiger auf das erste Array-Element angesehen wird. Hier sind der Beweis und das Beispiel dazu:

```
/* arr_ptr2.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int element[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };
    int i;

    printf("element      = %d\n", *element);
    printf("**element+1) = %d\n", *(element+1));
    printf("**element+3) = %d\n", *(element+3));

    printf("\nJetzt alle zusammen : \n");
    for(i=0; i<8; i++)
        printf("**element+%d) = %d \n", i, *(element+i));
    return EXIT_SUCCESS;
}
```

Leider sind es aber exakt solche Programmbeispiele, durch die der Eindruck entsteht, Arrays und Zeiger seien gleichwertig. Warum dies nicht so ist, habe ich bereits am Anfang erklärt.

Wenn Sie in dem eben gezeigten Beispiel unbedingt einen Adressoperator verwenden wollen, können Sie dies auch so schreiben:

```
ptr = &element[0]; /* identisch zu ptr=element */
```

Auf beide Arten wird dem Zeiger die Anfangsadresse des ersten Elements vom Array mit dem Index `[0]` übergeben. Der Verlauf des Programms soll jetzt genauer analysiert werden.

```
*(ptr+1);
```

Mit dieser Anweisung wird aus dem Array `element` der Wert 2 ausgegeben, also `element[1]`. Wieso dies so ist, möchte ich Ihnen wieder anhand einiger Grafiken veranschaulichen.

```
int *ptr;
...
int element[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };
```

Adresse: 0022FF60	elemente	Wert: 1
Adresse: 0022FF64		Wert: 2
Adresse: 0022FF68		Wert: 4
Adresse: 0022FF6C		Wert: 8
Adresse: 0022FF70		Wert: 16
Adresse: 0022FF74		Wert: 32
Adresse: 0022FF78		Wert: 64
Adresse: 0022FF7C		Wert: 128
Adresse: 004030FF	Name: ptr	Adresse: NULL

Abbildung 12.14 Visuelle Darstellung des Zeigers und des Arrays im Speicher

Das Array hat die Speicheradresse 0022FF60 bis 0022FF7C und eine Gesamtgröße von 32 Bytes (auf 16-Bit-Systemen: 16 Bytes). Ein Element hat die Größe von vier Bytes, da `int` vier Bytes groß ist (auf 32-Bit-Rechnern). Daher erfolgt auch die Adressierung immer in Vierer-Schritten. Durch die Anweisung

```
ptr = element /* oder */ ptr =& element[0]
```

sieht es im Speicher folgendermaßen aus:


	elemente	Adresse: 0022FF60
		Wert: 1
		Adresse: 0022FF64
		Wert: 2
		Adresse: 0022FF68
		Wert: 4
		Adresse: 0022FF6C
		Wert: 8
		Adresse: 0022FF70
		Wert: 16
		Adresse: 0022FF74
		Wert: 32
		Adresse: 0022FF78
		Wert: 64
		Adresse: 0022FF7C
		Wert: 128
Adresse: 004030FF	Name: ptr	Adresse: 0022FF60

Abbildung 12.15 Der Zeiger »ptr« verweist auf das erste Array-Element.

Damit verweist der Zeiger auf das erste Element im Array (oder genauer: auf die Speicheradresse des ersten Elements). Danach wird mit

```
*(ptr+1);
```

die Adresse 0022FF60 um vier Bytes erhöht. Genauso läuft dies auch mit den Arrays intern ab, wenn der Indexzähler erhöht wird.

Damit der Zeiger tatsächlich auf die nächste Adresse zeigt, muss `ptr+1` zwischen Klammern stehen, weil Klammern eine höhere Bindungskraft als der Dereferenzierungsoperator haben und somit zuerst ausgewertet werden. Sollten Sie die Klammern vergessen, würde nicht auf die nächste Adresse verwiesen, sondern auf den Wert, auf den der Zeiger `ptr` zeigt, und dieser wird um eins erhöht.

Jetzt zeigt der Zeiger `ptr` durch `*(ptr+1)` auf:

Das Diagramm zeigt eine Tabelle, die den Speicherlayout darstellt. Die Tabelle ist in drei Spalten unterteilt: 'Adresse', 'Name' und 'Wert'. Die ersten neun Zeilen zeigen ein Array von Elementen, die von Adresse 0022FF60 bis 0022FF7C reichen. Die Werte sind 1, 2, 4, 8, 16, 32, 64, 128 und 256. Die zehnte Zeile zeigt die Variable 'ptr' an der Adresse 004030FF, die den Wert 0022FF64 enthält. Ein Pfeil zeigt von der Adresse 0022FF64 in der ersten Spalte auf die Variable 'ptr' in der zehnten Zeile.

Adresse:		Wert:
0022FF60		1
0022FF64		2
0022FF68		4
0022FF6C		8
0022FF70		16
0022FF74		32
0022FF78		64
0022FF7C		128
004030FF	Name: ptr	Adresse: 0022FF64

Abbildung 12.16 Die Adresse des Zeigers wurde erhöht.

Somit wäre die Ausgabe 2. Kommen wir jetzt zur nächsten Anweisung:

```
*(ptr+3);
```

Hiermit wird der Wert der Adresse auf 0022FF6C erhöht. Deshalb wird auch der Wert 8 ausgegeben:

Adresse: 0022FF60	elemente	Wert: 1
Adresse: 0022FF64		Wert: 2
Adresse: 0022FF68		Wert: 4
Adresse: 0022FF6C		Wert: 8
Adresse: 0022FF70		Wert: 16
Adresse: 0022FF74		Wert: 32
Adresse: 0022FF78		Wert: 64
Adresse: 0022FF7C		Wert: 128
Adresse: 004030FF	Name: ptr	Adresse: 0022FF6C

Abbildung 12.17 Nach einer weiteren Erhöhung der Adresse des Zeigers

Um also auf das n-te Element eines Arrays zuzugreifen, haben Sie die folgenden Möglichkeiten:

```
int array[10];           // Deklaration
int *pointer1, *pointer2;
pointer1 = array;        // pointer1 auf Anfangsadresse von array
pointer2 = array + 3;    // pointer2 auf 4.Element von array

array[0]      = 99;      // array[0]
pointer1[1]   = 88;      // array[1]
*(pointer1+2) = 77;      // array[2]
*pointer2     = 66;      // array[3]
```

Dasselbe gilt auch für Funktionsaufrufe von Array-Namen. Einen Array-Parameter in Funktionen können Sie auf zwei Arten deklarieren:

```
int funktion(int elemente[])

// Gleichwertig mit ...
```

```
int funktion(int *elemente)
```

Also kann eine Funktion mit folgenden Argumenten aufgerufen werden:

```
int werte[] = { 1, 2, 3, 5, 8 };
int *pointer;
pointer = werte;

funktion(werte);      // 1. Möglichkeit
funktion(&werte[0]);  // 2. Möglichkeit
funktion(pointer);    // 3. Möglichkeit
```

Natürlich ist es auch möglich, die Adresse des n-ten Elements an eine Funktion zu übergeben:

```
funktion(&werte[2]);  // Adresse vom 3.Element an funktion
```

Hierzu ein kleines Beispiel:

```
/* arr_ptr3.c */
#include <stdio.h>
#include <stdlib.h>
```



```

void funktion(int *array, int n_array) {
    int i;

    for(i=0; i < n_array; i++)
        printf("%d ", array[i]);
    printf("\n");
}

int main(void) {
    int werte[] = { 1, 2, 3, 5, 8, 13, 21 };

    funktion(werte, sizeof(werte) / sizeof(int));
    return EXIT_SUCCESS;
}

```

Wie sieht es aber mit dem Laufzeitverhalten aus? Was passiert, wenn die Funktion mit Feldindex verwendet wird?

```

void funktion(int array[], int n_array)

```

Compiler optimieren den Code bei der Übersetzung in der Regel selbst. Die Umwandlung eines Feldindex in einen Zeiger macht dem Compiler heutzutage keine Probleme mehr. Somit dürfte es keine bemerkbaren Laufzeitverluste bei der Verwendung des Indizierungsoperators geben.

12.8 Zeiger auf Strings

Alles, was bisher zu den Zeigern mit Arrays gesagt wurde, gilt auch für Zeiger auf Strings. Häufig wird dabei irrtümlicherweise von einem Zeiger gesprochen, der auf einen String verweist. Dieses Missverständnis entsteht durch folgende Deklaration:

```

char *string = "Hallo Welt";

```

Dies ist eine Stringkonstante, auf die ein Zeiger zeigt. Genauer: Der Zeiger zeigt auf die Anfangsadresse dieser Konstanten, den Buchstaben 'H' – genauer, auf die Adresse des Buchstabens 'H'. Hierzu ein Beispiel:

```

/* str_ptr1.c */
#include <stdio.h>
#include <stdlib.h>

void funktion(char *str) {
    printf("%s\n", str);
}

int main(void) {
    char *string = "Hallo Welt";

    funktion(string);
    printf("Anfangsadresse auf die *string zeigt = %p\n", *string);
    printf("Der Inhalt dieser Anfangsadresse = %c\n", *string);
    return EXIT_SUCCESS;
}

```

Wie diese Funktion hier werden übrigens alle ANSI-C-Funktionen deklariert, die Zeichenketten verarbeiten. All diese Funktionen (printf() zum Beispiel) bekommen als Argument nur die Anfangsadresse einer Zeichenkette übergeben. Anschließend lesen diese Funktionen Zeichen für Zeichen aus, bis sie auf das Stringende-Zeichen '\0' treffen. Dabei dürfte Ihnen bei einem genaueren Blick auffallen, dass bei vielen Funktionen die Variablen-Parameter mit dem Schlüsselwort `const` deklariert sind.

12.8.1 Zeiger auf konstante Objekte (Read-only-Zeiger)

Zeiger, die als Zusatz das Schlüsselwort `const` enthalten, sind sogenannte *Read-only-Zeiger*. Das bedeutet, dass auf diese Zeiger nur lesend zugegriffen werden kann. Angenommen, die Syntax von `printf()` ist in der Headerdatei `<stdio.h>` folgendermaßen deklariert:

```
int printf (const char*, ...);
```

Mit dieser Angabe kann aus dem Parameter der Funktion `printf()` nur gelesen werden. Es ist also nicht möglich, diesen Inhalt irgendwie mit einem anderen Zeiger zu manipulieren. Hierzu ein Beispiel:

```
/* const_ptr.c */
#include <stdio.h>
#include <stdlib.h>

void funktion1(char *str) {
    char *ptr;

    ptr = str+5;
    *ptr = '-';
}

int main(void) {
    char string1[] = "Hallo Welt\n";

    funktion1(string1);
    printf("%s\n", string1);
    return EXIT_SUCCESS;
}
```

Hier wird die Zeichenfolge `Hallo Welt` in der Funktion manipuliert. Zwischen `Hallo` und `Welt` wird in der Funktion ein Bindestrich eingefügt. Wollen Sie dies vermeiden, müssen Sie nur die Funktionsdeklaration ändern:

```
void funktion1(const char *str)
```

Sie können jetzt zwar die einzelnen Inhalte der Zeichenkette lesen, aber nicht mehr in der Funktion ändern. Außerhalb der Funktion ist der Schreibschutz natürlich wieder aufgehoben.

12.9 Zeiger auf Zeiger und Stringtabellen

»Zeiger auf Zeiger« sind ein recht schwieriges Thema, aber es zu verstehen, lohnt sich. Die Syntax von Zeigern auf Zeiger sieht so aus:

```
datentyp **bezeichner;
```

Was heißt jetzt »Zeiger auf Zeiger« genau? Sie haben einen Zeiger, der auf einen Zeiger zeigt, der auf eine Variable zeigt, und auf diese Variable zurückgreifen kann. Im Fachjargon wird dabei von einer *mehrfachen Indirektion* gesprochen. Theoretisch ist es auch möglich, Zeiger auf Zeiger auf Zeiger usw. zu verwenden. In der Praxis machen allerdings solche mehrfachen Indirektionen kaum noch Sinn. Meistens verwenden Sie Zeiger auf Zeiger, also zwei Dimensionen.

Das Haupteinsatzgebiet von Zeigern auf Zeiger ist die dynamische Erzeugung von mehrdimensionalen Arrays wie beispielsweise Matrizenberechnungen. Aber darauf gehe ich in Kapitel 14, »Dynamische Speicherverwaltung«, ein.

Sehen wir uns zuerst ein Beispiel zu diesem komplexen Thema an:

```

/* ptrptr1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int wert = 10;
    /* ptr ist ein Zeiger auf int wert. */
    int *ptr=&wert;
    /* ptr_ptr ist ein Zeiger auf den Zeiger int *ptr. */
    int **ptr_ptr =& ptr;

    printf("*ptr      : %d\n", *ptr);
    printf("***ptr_ptr : %d\n", **ptr_ptr);

    /* Verändert den Wert, auf den int *ptr zeigt. */
    **ptr_ptr = 100;
    printf("*ptr      : %d\n", *ptr);
    printf("***ptr_ptr : %d\n", **ptr_ptr);

    /* Verändert nochmals den Wert. */
    *ptr = 200;
    printf("*ptr      : %d\n", *ptr);
    printf("***ptr_ptr : %d\n", **ptr_ptr);
    return EXIT_SUCCESS;
}

```

Wichtig in diesem Beispiel ist, dass Sie bei der Veränderung der Variablen den doppelten Indirektionsoperator (**) einsetzen, genauso wie bei der Deklaration des Zeigers auf einen Zeiger. Hätten Sie nämlich anstatt

```
**ptr_ptr = 100;
```

Folgendes geschrieben:

```
*ptr_ptr = 100;
```

würde der Zeiger `ptr_ptr` auf die Speicheradresse 100 verweisen. Und dies ist zumeist irgendwo im Nirwana des Speichers. Wie gesagt, in Kapitel 14 wird dieses Thema nochmals aufgegriffen, und es wird Ihnen dort einiges sinnvoller erscheinen.

12.9.1 Stringtabellen

Um es jetzt noch komplizierter zu machen, will ich gleich noch die Stringtabellen hinzunehmen, die den Zeigern auf Zeiger nicht unähnlich sind (aber nicht dasselbe sind!).

Ein Beispiel: Folgende Stringkonstanten sollen nach Alphabet sortiert werden (ohne Verwendung der Headerdatei `<string.h>`):

```
"Zeppelin", "Auto", "Amerika", "Programmieren"
```

Sie wissen ja noch, dass `*ptr` dieselbe Anfangsadresse wie `ptr[0]` repräsentiert. Und Gleiches gilt jetzt auch für:

```
**ptrptr und *ptrptr[0]
```

Damit haben Sie ein Array von Zeigern. Und so würde dies im Beispiel aussehen:

```

char *sort[] = {
    "Zeppelin", "Auto", "Amerika", "Programmieren"
};

```

Hier haben Sie eine sogenannte Stringtabelle. Wie kann jetzt auf die einzelnen Strings einer Stringtabelle zugegriffen werden? Dazu ein kleines Beispiel:

```

/* ptrptr2.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char *sort[] = {
        "Zeppelin", "Auto", "Amerika", "Programmieren"
    };

    printf("%s\n", sort[1]);           /* Auto          */
    printf("%s ", (sort[2]+2));       /* erika         */
    printf("%s %s\n", (sort[0]+6), sort[2]); /* in Amerika   */
    printf("%.5s\n", (sort[3]+5-2));  /* gramm         */
    return EXIT_SUCCESS;
}

```

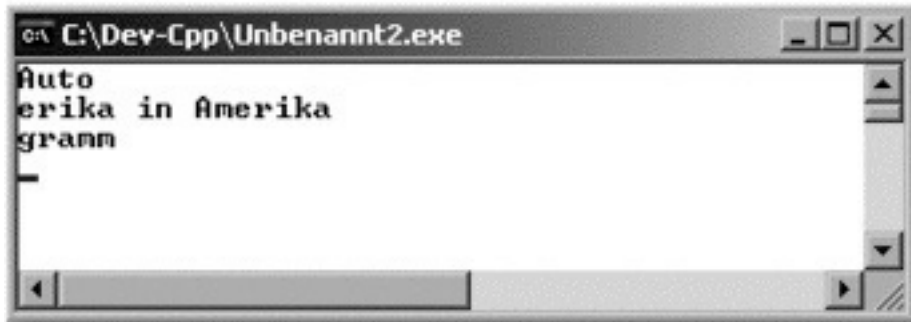


Abbildung 12.18 Verwendung einer Stringtabelle

Der Zugriff auf die Stringtabelle erfolgt ähnlich wie bei den mehrdimensionalen Arrays. Die erste Ausgabe

```
printf("%s\n", sort[1]);
```

gibt das Wort »Auto« auf dem Bildschirm aus. Daher kann davon ausgegangen werden, dass bei

```

sort[0] = "Zeppelin"
sort[1] = "Auto"
sort[2] = "Amerika"
sort[3] = "Programmieren"

```

mithilfe des Indizierungsoperators auf die Anfangsadressen der einzelnen Zeichenketten verwiesen wird. Mit der zweiten Anweisung

```
printf("%s ", (sort[2] + 2) );
```

wird der Name »erika« ausgegeben. Das lässt sich so erklären: `sort[2]` repräsentiert die Anfangsadresse von »Amerika« also »A«. Danach kommt +2 hinter dem Feldindex hinzu. Der Zeiger, der ohne +2 weiterhin auch auf den Anfang von »Amerika« gezeigt hätte, zeigt jetzt auf den dritten Buchstaben des Wortes, also auf »e«. Oder genauer: auf die Adresse von »e«. Mit dem Formatzeichen `%s` wird anschließend veranlasst, dass der String von dieser Adresse an auf dem Bildschirm ausgegeben wird. Genauso verläuft dies bei der nächsten Ausgabe. Die Schreibweise

```
printf("%s\n", (sort[3] + 5 - 2) );
```

dient nur der Demonstration, dass es so auch geht. Natürlich lässt sich das leichter lesen mit:

```
printf("%s\n", (sort[3] + 3) );
```

Das Programm soll nochmals anhand von Adressen demonstriert werden:

```

/* ptrptr3.c */
#include <stdio.h>

```

```
#include <stdlib.h>

int main(void) {
    char *sort[] = {
        "Zeppelin", "Auto", "Amerika", "Programmieren"
    };
    printf("%p = %c\n", **sort, **sort);

    printf("%p = %c\n", *sort[0], *sort[0]);
    printf("%p = %c\n", *(sort[0]+0), *(sort[0]+0));
    printf("%p = %s\n", sort[0], sort[0]);
    printf("%p = %s\n", *sort, *sort);

    printf("%p = %s\n", (sort[0]+1), (sort[0]+1));
    printf("%p = %s\n", (sort[0]+2), (sort[0]+2));

    printf("*sort = %p, **sort = %p\n", *sort, **sort);
    return EXIT_SUCCESS;
}
```

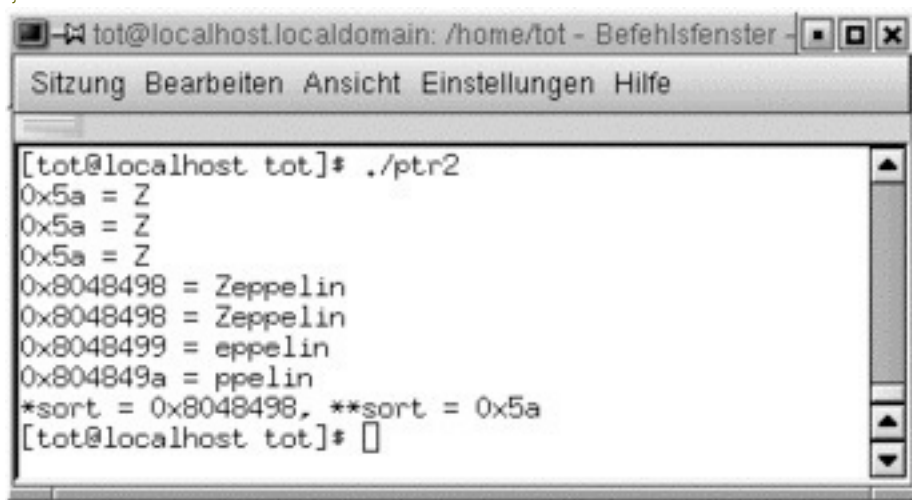


Abbildung 12.19 Ausgabe der Adressen einer Stringtabelle

Bei den ersten drei Ausgaben

```
printf("%p = %c\n", **sort, **sort);
printf("%p = %c\n", *sort[0], *sort[0]);
printf("%p = %c\n", *(sort[0]+0), *(sort[0]+0));
```

wurden immer die (Anfangs-)Adressen und Inhalte verwendet, auf die der zweite Zeiger zeigt – was die Ausgabe auch bestätigt. Anschließend wird nur die Adresse des ersten Zeigers benutzt:

```
printf("%p = %s\n", sort[0], sort[0]);
printf("%p = %s\n", *sort, *sort);
```

Der Inhalt ist bei Benutzung von einem Zeiger natürlich derselbe wie bei der Benutzung von zwei Zeigern. Aber bei der Übersetzung des Programms haben beide Zeiger eine andere Adresse. Die Ausgabe von

```
printf("*sort = %p, **sort = %p\n", *sort, **sort);
```

bestätigt alles dies erneut. Jeder einzelne Zeiger benötigt also seinen Speicherplatz und somit auch eine eigene Adresse. Ich versuche, es noch einmal anders zu erklären:

```
*( * (Variable + x) + y)
```

Hiermit wird auf das *y*-te Zeichen im *x*-ten String gezeigt. Bei dem Programm sieht dies so aus:

```
*( * (sort + 1) + 2)
```

oder auch – wie schon bekannt – so:

```
*( (sort[1]) + 2)
```

Hiermit würde auf das 3-te Zeichen im 2-ten String verwiesen, was hierbei dem Zeichen »t« vom String »Auto« entspricht.

Jetzt soll diese Stringtabelle nach Alphabet sortiert werden. Dabei wird nicht die ganze Textzeile verlagert und unnötig hin- und herkopiert, sondern es müssen lediglich die Zeiger in die richtige Reihenfolge gebracht werden:

```
/* ptrptr4.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *sort[] = {
        "Zeppelin", "Auto", "Amerika", "Programmieren"
    };
    int i, j;
    char *temp;

    for(i = 0; i < 4; i++) {
        for(j = i + 1; j < 4; j++) {
            if( (strcmp(sort[i], sort[j]) > 0) ) {
                temp = sort[i];
                sort[i] = sort[j];
                sort[j] = temp;
            }
        }
    }
    for(i = 0; i < 4; i++)
        printf("%s\n", sort[i]);
    return EXIT_SUCCESS;
}
```

Bei diesem Sortieralgorithmus handelt es sich um »Selektion Sort«. Die folgenden Zeilen sortieren die Felder mit Zeigern:

```
for(i = 0; i < 4; i++) {
    for(j = i + 1; j < 4; j++) {
        if( (strcmp(sort[i], sort[j]) > 0) ) {
            temp = sort[i];
            sort[i] = sort[j];
            sort[j] = temp;
        }
    }
}
```

Zuerst wird das erste Element in der Stringtabelle mit allen anderen verglichen. So wird das kleinste Element gefunden, das an den Anfang gestellt wird. Danach wird das zweite Element mit allen vor ihm liegenden verglichen. Dies geht so weiter bis zum letzten Element in der Stringtabelle. Mehr zu den Algorithmen finden Sie in Kapitel 22.

Das Wichtigste – wie schon mehrmals erwähnt wurde – ist, dass Zeiger für Adressen da sind und sonst nichts. Beispielsweise bedeutet

```
char *text[500];
```

nichts anderes als ein char-Array mit 500 char-Zeigern. Genauer gesagt, kann jeder dieser 500 Zeiger z. B. auf einen String (char-Array) zeigen. Beweis gefällig? Bitte sehr:

```

/* ptrptr5.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char *text[500];
    char str1[] = "Text1";
    char str2[] = "Text2";
    char str3[] = "Text3";

    text[0] = str1;
    text[1] = str2;
    text[2] = str3;

    printf("%s %s %s\n", text[0], text[1], text[2]);
    return EXIT_SUCCESS;
}

```

In diesem Beispiel wurde den ersten drei Zeigern jeweils die Anfangsadresse einer Stringkonstante übergeben. Mit einfachen Arrays war dies nicht ausführbar. Natürlich ist es jetzt noch nicht möglich, die Anfangsadresse eines zur Laufzeit erstellten Textes so zuzuweisen. Dazu brauchen Sie Kenntnisse in der dynamischen Speicherverwaltung.

Als es darum ging, Strings zu sortieren, konnte mithilfe der Zeiger auf die Anfangsadresse der Strings wesentlich effektiver (schneller) sortiert werden, als wenn dies mit dem ganzen String gemacht würde. Dies rührt daher, dass ja nur Adressen auf einem String benutzt werden.

Und anstatt

```

char *sort1 = "Zeppelin";
char *sort2 = "Auto" ;
char *sort3 = "Amerika";
char *sort4 = "Programmieren";

```

zu schreiben, ist doch diese Schreibweise

```

char *sort[] = {
    "Zeppelin", "Auto", "Amerika", "Programmieren"
};

```

viel effektiver und kürzer. Hier sind es vier Zeiger auf ein `char`-Array, die auf die Anfangsadresse eines jeden einzelnen Wortes zeigen.

Folgende Vorteile ergeben sich für den Programmierer, wenn er Stringtabellen verwendet:

- Mit Stringtabellen wird das Programm übersichtlicher.
- Es wird Speicherplatz gespart.
- Die Verwaltung von Stringtabellen ist einfacher und effizienter.
- Sollte ein Programm in mehreren Sprachen geschrieben werden, kann dies leichter lokalisiert werden.

Hier ein Beispiel, wie Sie Stringtabellen effektiv einsetzen können:

```

/* ptrptr6.c */
#include <stdio.h>
#include <stdlib.h>
#define ASK 0
#define WORDS 1
#define START 2

#define ENGLISH 1

```

```

#ifdef GERMAN
const char *language[] = {
    "Du sprichst Deutsch?", "Einige Worte: ",
    "Feuer", "Erde", "Wasser", "Luft", "Leben", NULL
};
#elif ENGLISH
const char *language[] = {
    "Do you speak english?", "Some words: ",
    "Fire", "earth", "water", "air", "life", NULL
};
#else /* FRENCH */
const char *language[] = {
    "Tu parle francais?", "quelques mots: ",
    "Le feu", "La terre", "de l'eau", "de l'air", "La vie", NULL
};
#endif

int main(void) {
    int i;

    printf("%s\n", language[ASK]);
    printf("%s\n", language[WORDS]);

    for(i = START; language[i] != NULL; i++)
        printf("\t%s,\n", language[i]);
    return EXIT_SUCCESS;
}

```

Hierbei handelt es sich um ein einfaches Listing, das mit bedingter Kompilierung ein Programm in entsprechender Sprache übersetzt. In diesem Beispiel wurde mit

```
#define ENGLISH 1
```

die Sprache auf Englisch eingestellt. Bei Ausgabe des Programms wird dies auch bestätigt. Müssen Sie jetzt eine Version für Ihren spanischen Kollegen schreiben, müssen Sie nur nach der Stringtabelle suchen und entsprechende Einträge übersetzen und hinzufügen. So entsteht ohne allzu großen Aufwand ein internationales Programm:

```

#ifdef GERMAN
const char *language[] = {
    "Du sprichst Deutsch?", "Einige Worte: ",
    "Feuer", "Erde", "Wasser", "Luft", "Leben", NULL
};
#elif ENGLISH
const char *language[] = {
    "Do you speak english?", "Some words: ",
    "Fire", "earth", "water", "air", "life", NULL
};
#elif FRENCH
const char *language[] = {
    "Tu parle francais?", "quelques mots: ",
    "Le feu", "La terre", "de l'eau", "de l'air", "La vie", NULL
};
#else /* ESPANOL */
const char *language[] = {
    "Habla Usted espanol", "algunas palabras: ",
    "Fuego", "tierra", "agua", "aire", "vida", NULL
};
#endif

```

Mit Stringtabellen lassen sich auch komfortabel Fehlermeldungen auf dem Bildschirm ausgeben:

```

char *fehlermeldung[] = {
    "Mangel an Speicherplatz",
    "Speicherbereichsüberschreitung",
}

```



```

    "Wertbereichsüberschreitung",
    "Die Syntax scheint falsch",

    "Zugriff verweigert - keine Rechte",
    "Zugriff verweigert - falsches Passwort",
    "Unbekannter Fehler trat auf"
};

```

Zugegriffen wird auf die einzelnen Fehlermeldungen mit dem Feldindex von Nr.[0]–[6]. »Zugriff verweigert – keine Rechte« beispielsweise ist somit `fehlermeldung[4]`.

Nach diesem Abschnitt über Zeiger auf Zeiger und den Stringtabellen kommen sicherlich jetzt die einen oder anderen Fragen auf. Vor allem wurden die Beispiele immer nur mit konstanten Werten gegeben. Um sich also wirklich effektiv und sinnvoll mit dem Thema auseinanderzusetzen, müssen Sie sich noch ein wenig gedulden, bis Sie zur (ich wiederhole mich) dynamischen Speicherverwaltung gelangen (Kapitel 14).

12.10 Zeiger auf Funktionen

Mit den Zeigern können Sie auch auf Maschinencode von anderen Funktionen zeigen, die schließlich ebenfalls eine Anfangsadresse im Speicher besitzen. Ein einfaches Beispiel dazu:

```

/* ptr_func1.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int (*ptr) (const char*, ...);
    ptr=printf;
    (*ptr) ("Hallo Welt\n");
    return EXIT_SUCCESS;
}

```

Jetzt eine Erklärung zur folgenden Schreibweise:

```
int (*ptr) (const char*, ...);
```

Dies ist ein Zeiger auf eine Funktion, die einen variablen, langen String erhält und einen `int`-Wert zurückgibt. Die Funktion `printf()` zum Beispiel ist eine solche Funktion:

```
int printf (const char*, ...);
```

Daher bekommt der Zeiger die Adresse dieser Funktion mit folgender Anweisung:

```
ptr = printf;
```

Jetzt können Sie die Funktion `printf()` mit dem Zeiger aufrufen:

```
(*ptr) ("Hallo Welt\n");
```

Die erste Klammerung des Zeigers `ptr` ist wichtig. Würden Sie diese weglassen, dann würden Sie `ptr` als eine Funktion deklarieren, die einen `int`-Zeiger zurückgibt. Sie können den (Funktions-)Zeiger auch auf eine andere Funktion zeigen lassen, mit demselben Rückgabewert und dem- bzw. denselben Argument(en). Hier sehen Sie das Listing von eben mit weiteren Adressierungen:

```

/* ptr_func2.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {

```

```

int (*ptr)(const char*, ...);
int zahl;

ptr=printf;
(*ptr)("Bitte eine Zahl eingeben: ");
ptr=scanf;
(*ptr)("%d",&zahl);
ptr=printf;
(*ptr)("Die Zahl lautet %d\n",zahl);
return EXIT_SUCCESS;
}

```

Dieses Beispiel sollte natürlich keine Schule machen und Ihnen nur zeigen, wie Sie mit Zeigern auf Funktionen zeigen können.

Zeiger auf Funktionen können ganz nützlich sein. Sie können die Zeiger in einem Feld speichern und eine Sprungtabelle daraus machen. Damit ist gemeint, dass die einzelnen Funktionen mit einem Feldindex angesprochen werden können, ähnlich wie bei den Stringtabellen – also ein Array von Funktionen, wenn Sie so wollen. Dazu soll das Listing von eben wieder herhalten:

```

/* ptr_func3.c */
#include <stdio.h>
#include <stdlib.h>
#define INPUT 0
#define OUTPUT 1

int main(void) {
    int (*ptr[])(const char *, ...) = { scanf, printf };
    int zahl;

    (*ptr[OUTPUT])("Bitte eine Zahl eingeben: ");
    (*ptr[INPUT])("%d",&zahl);

    (*ptr[OUTPUT])("Die Zahl lautet %d\n",zahl);
    return EXIT_SUCCESS;
}

```

Viel verändert hat sich hierbei nicht. Statt

```
int (*ptr)(const char*, ...);
```

wurde hier einfach noch ein Indizierungsoperator hinzugefügt. Am Ende befinden sich zwischen den geschweiften Klammern noch die einzelnen Funktionen, auf die Sie im Programm mithilfe des Indizierungsoperators und dem entsprechenden Index zurückgreifen können:

```
int (*ptr[])(const char *, ...) = { scanf, printf };
```

In diesem Beispiel zeigt (*ptr[0]) auf die Adresse der Funktion scanf(), und (*ptr[1]) zeigt auf die Funktion printf(). Im Programm wurden hierbei symbolische Konstanten verwendet, um diese beiden Funktionen besser auseinanderzuhalten.

Voraussetzung dafür, dass diese Zeiger auf Funktionen auch funktionieren, ist immer, dass der Rückgabewert (hier vom Typ int) und der/die Parameter der Funktion (hier (const char *, ...)) übereinstimmen. Sie können hierbei nicht einfach zusätzlich z. B. die Funktion fgets() zum Einlesen von Strings anhängen:

```

/* falsch */
int (*ptr[])(const char *, ...) = { scanf, printf, fgets };
Die Funktion fgets() erwartet andere Argumente, hier (const char *, ...).

```

Wenn es möglich ist, mit Zeigern auf Funktionen der Standard-Bibliothek zu zeigen, dann ist es selbstverständlich auch möglich, mit Zeigern auf selbst geschriebene Funktionen zu zeigen.

```
/* ptr_func4.c */
#include <stdio.h>
#include <stdlib.h>
/* Bei Linux für math.h das Compiler-Flag -lm mit angeben:
 * gcc -o programm programm.c -lm
 */
#include <math.h> /* sqrt() */

int addition(int zahl) {
    int y;

    printf("%d+>", zahl);
    scanf("%d", &y);
    fflush(stdin);
    return zahl += y;
}

int subtraktion(int zahl) {
    int y;

    printf("%d->", zahl);
    scanf("%d", &y);
    fflush(stdin);
    return zahl -= y;
}

int division(int zahl) {
    int y;

    printf("%d/>", zahl);
    scanf("%d", &y);
    fflush(stdin);
    return zahl /= y;
}

int multiplikation(int zahl) {
    int y;

    printf("%d*>", zahl);
    scanf("%d", &y);
    fflush(stdin);
    return zahl *= y;
}

int sqrtw(int zahl) {
    double x=sqrt((double) zahl);

    printf("(sqrt)%f>", x);
    return (int)x;
}

int (*rechenfunk[]) (int) = {
    addition, subtraktion, division, multiplikation, sqrtw
};

int main(void) {
    char op;
    static int zahl;

    printf("no.>");
    scanf("%d",&zahl);
```

```

do {
    printf(" op>");
    scanf("%c",&op);
    fflush(stdin);
    switch(op) {
        case '+': printf("%d", zahl = (*rechenfunk[0])(zahl));
                  break;
        case '-': printf("%d", zahl = (*rechenfunk[1])(zahl));
                  break;
        case '/': printf("%d", zahl = (*rechenfunk[2])(zahl));
                  break;
        case '*': printf("%d", zahl = (*rechenfunk[3])(zahl));
                  break;
        case 'q': printf("%d", zahl = (*rechenfunk[4])(zahl));
                  break;
        default : printf("op '=','+', '-', '/', '*', 'q'\n");
    }
} while(op != '=');
printf("Gesamtergebnis=%d\n", zahl);
return EXIT_SUCCESS;
}

```

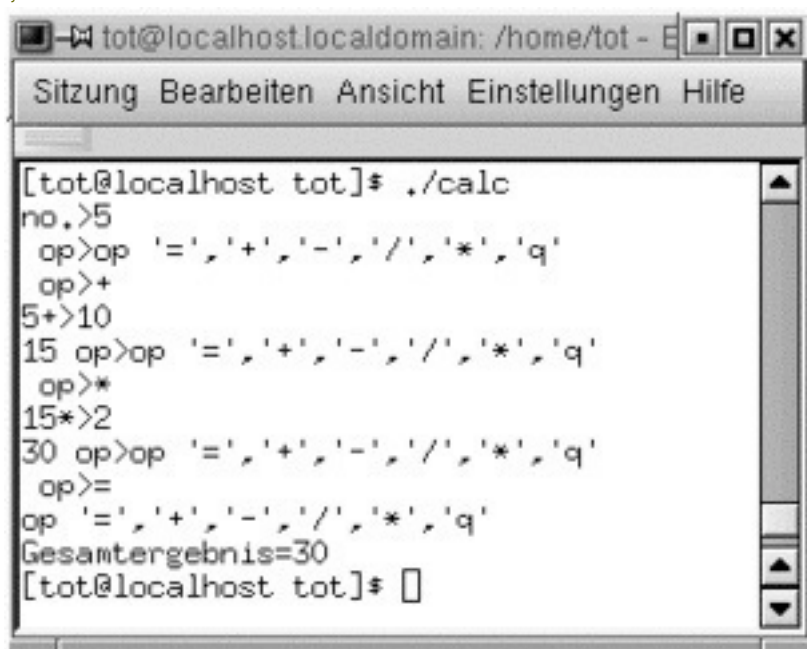


Abbildung 12.20 Zeiger auf selbst geschriebene Funktionen

Dieses Programm stellt einen kleinen Taschenrechner dar, der allerdings stark verbesserungswürdig ist. Es können fünf Rechenoperationen wie Addition, Subtraktion, Multiplikation, Division und die Quadratwurzel verwendet werden. Es wird so lange gerechnet, bis das =-Zeichen eingegeben wurde. Hierzu folgt ein kurzer Trocken-Durchlauf. Nachdem das Programm gestartet wurde, sollte auf dem Bildschirm Folgendes erscheinen:

no.>

Hier geben Sie die erste Zahl ein und drücken , beispielsweise die Zahl 8. Anschließend geht es in der do while-Schleife weiter. Jetzt erscheint auf dem Bildschirm:

op>

Hier muss der Rechenoperator eingegeben werden: entweder +, -, *, / oder q. In diesem Beispiel soll es eine Addition sein, also geben Sie das +-Zeichen ein und

drücken **Enter**. Danach wird das Zeichen im switch-Schalter darauf geprüft, ob es sich dabei um ein gültiges Zeichen handelt. In diesem Fall wäre das:

```
case '+': printf("%d", zahl = (*rechenfunk[0])(zahl));
```

Hiermit wird die Rechenfunktion mit dem Index [0] aufgerufen. Dieser Funktion wird als Parameter der Wert der Variablen `zahl` übergeben. Sehen Sie sich dazu die Funktionstabelle an:

```
int (*rechenfunk[]) (int) = {  
    addition, subtraktion, division, multiplikation, sqrtw  
};
```

Die Rechenfunktion mit dem Index [0] ist die Funktion `addition`. `subtraktion` hat den Index [1], `division` den Index [2] usw.

In der Funktion `addition()` geben Sie dann eine Zahl ein, mit der der übergebene Wert addiert werden soll, beispielsweise die Zahl 2, und anschließend drücken Sie wieder **Enter**:

```
10 op>
```

Jetzt kann ein weiterer Operator verwendet oder mit dem `--`-Zeichen das Programm beendet werden.

12.11 void-Zeiger

Ein Zeiger auf `void` ist ein typenloser und vielseitiger Zeiger. Wenn der Datentyp des Zeigers noch nicht feststeht, wird der `void`-Zeiger verwendet. `void`-Zeiger haben den Vorteil, dass Sie diesen eine beliebige Adresse zuweisen können. Außerdem kann ein `void`-Zeiger durch eine explizite Typumwandlung in jeden anderen beliebigen Datentyp umgewandelt werden. Beispielsweise:

```
/* void_ptr1.c */  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void) {  
    int a = 10;  
    char *string = "void-Zeiger";  
    void *ptr;  
  
    /* void-Zeiger auf Variable int a */  
    ptr = (int *)&a;  
    printf("ptr = %p a=%p\n", ptr, &a);  
    /* void-Zeiger auf string */  
  
    ptr = (char *)string;  
    printf("ptr = %p string = %p\n", ptr, string);  
    return EXIT_SUCCESS;  
}
```

Natürlich sollten Sie darauf achten, dass Sie für das Casting einen Zeiger angeben und nicht etwa einen Datentyp:

```
/* Richtig */  
ptr=(typ *)&ptr2;  
  
/* Falsch: typ ist kein Zeiger, sondern eine Variable */  
ptr=(typ)&ptr2;
```

Zwar wurde hier im Beispiel ein Cast von `void *` nach `datentyp *` gemacht, aber dies ist in C nicht unbedingt nötig – C++ allerdings macht sehr wohl einen Unterschied und braucht einen Cast von `void *` nach `datentyp *`.

Würden Sie im Beispiel oben die Casts entfernen und das Beispiel als C++-Projekt übersetzen (was bei vielen Compilern unter MS-Windows häufig voreingestellt ist), würde der Compiler eine Warnung ausgeben, wie beispielsweise:

```
[Warning]:  
In function int main : invalid conversion from `void*' to `int*`
```

Und genau diese Warnmeldung lässt viele Programmierer vermuten, dass etwas am Listing falsch sei und man den `void`-Pointer immer casten müsste. Sofern Sie Ihren Compiler nicht davon überzeugen können, dass Sie gern ein C-Projekt schreiben würden, sollten Sie meiner Meinung nach dem sanften Druck des Compilers nachgeben (auch wenn das Programm tut, was es tun soll und ohne Problem ausgeführt werden kann) und ihm sein Cast geben – da dies ja auch nicht unbedingt »falsch« ist. Schließlich zählt zu einem der oberen Gebote eines Programmierers, dass man niemals Warnmeldungen eines Compilers ignorieren soll.

Vorwiegend findet ein `void`-Zeiger Anwendung in Funktionen, die mit unterschiedlichen Zeigern aufgerufen werden können. Beispielsweise ist die Funktion `memcmp()` in der Headerdatei `<string.h>` folgendermaßen angegeben:

```
int memcmp (const void*, const void*, size_t);
```

Somit kann diese Funktion mit unterschiedlichen Zeigertypen verwendet werden, wie das folgende Beispiel zeigt:

```
/* void_ptr2.c */  
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
int main(void) {  
    char str1[]="Hallo";  
    char str2[]="Hallo";  
    int num1[] = { 1,2,3,4,5,6 };  
    int num2[] = { 1,3,5,7,9,1 };  
    int cmp;  
    /* Casts sind nicht unbedingt nötig. */  
    cmp=memcmp( (char *)str1, (char *)str2, sizeof(str1));  
    if(cmp ==0)  
        printf("Beide Strings sind gleich\n");  
    else  
        printf("Die Strings sind nicht gleich\n");  
    /* Casts sind nicht unbedingt nötig. */  
    cmp=memcmp( (int *)num1, (int *)num2, sizeof(num1)/sizeof(int));  
    if(cmp == 0)  
        printf("Der Inhalt der beiden Zahlenarrays ist gleich\n");  
    else  
        printf("Die Zahlenarrays sind unterschiedlich\n");  
    return EXIT_SUCCESS;  
}
```

Die Umwandlung in einen entsprechenden Zeigertyp findet mit einem einfachen Type-Casting statt – was auch hier nicht unbedingt nötig gewesen wäre (siehe den Abschnitt vor dem Listing).

Für einige ist es verwirrend, wie ein leerer Zeiger (`void`, dt. *leer*) einfach so in irgendeinen Datentyp gecastet werden kann, weil sie gelernt haben, dass `int` vier Bytes Speicher hat, `double` acht Bytes Speicher und `void` eben keinen. Dabei ist `void`

eigentlich auch nicht ganz leer. Wenn Sie mit `void` den `sizeof`-Operator verwenden, erfahren Sie, dass `void` ein Byte an Speicher benötigt.

Aber erinnern Sie sich nochmals an den Anfang des Kapitels, bei dem Sie den `sizeof`-Operator auf alle Typen von Zeigern verwendet haben: Da habe ich gesagt, dass alle Zeiger, egal welchen Typs, einen Speicherbedarf von vier Bytes (32 Bit) haben. Mehr ist auch nicht erforderlich, um eine Speicheradresse zu speichern. Ebenso sieht es mit dem `void`-Zeiger aus. Dieser benötigt wie alle anderen Zeiger vier Byte Speicherplatz.

```
/* void_ptr3.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    void *void_ptr;

    printf("%d Byte\n", sizeof(void_ptr));
    return EXIT_SUCCESS;
}
```

Wollen Sie den Typ, auf den der `void`-Zeiger verweist, dereferenzieren, wird die Sache ein wenig komplizierter. Dafür benötigen Sie einen weiteren Zeiger:

```
/* void_ptr4.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    void *void_ptr;
    int wert = 10;

    void_ptr=(int *)&wert;
    *(int *)void_ptr = 100;
    printf("%d\n", wert); /* 100 */
    return EXIT_SUCCESS;
}
```

Da der gecastete `void`-Zeiger allein noch nicht dereferenziert werden kann, wird hier einfach ein weiterer Zeiger verwendet:

```
*(int *)void_ptr = 100;
```

Jetzt denken Sie sicherlich darüber nach, welchen Vorteil eigentlich ein `void`-Zeiger hat? Bei dem Beispiel, in dem die Funktion `memcmp()` verwendet wurde, ist der Vorteil eigentlich schon klar. Anstatt für jeden Datentyp eine eigene Funktion zu schreiben, wird einfach der `void`-Zeiger verwendet, und der Funktion kann es egal sein, mit welchem Datentyp sie verwendet wird. Wichtig ist dabei nur, dass die Funktion (logischerweise) entsprechend universell geschrieben wurde. Sie können nicht einfach einer Funktion, die mit der Funktion `strcmp()` einzelne Strings vergleicht, als Argument die Anfangsadresse eines `int`-Arrays übergeben.

12.12 Äquivalenz zwischen Zeigern und Arrays

Es wurde ja bereits erwähnt, dass Zeiger und Arrays zwar eine gewisse Ähnlichkeit in ihrer Anwendung aufweisen, aber deswegen noch lange nicht gleich sind – geschweige denn untereinander austauschbar sind. Dennoch wird häufig beides in einen Topf geworfen.

```
/* ptr_versus_array1.c */
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void) {
    int var[] = { 123, 456, 789};

    printf("*var : %d; var[0] : %d\n", *var, var[0]);
    return EXIT_SUCCESS;
}
```

Wenn Sie dieses Beispiel übersetzen, verstehen Sie, worauf ich hinaus will. Die Angaben von `var[0]` und `*var` repräsentieren ja dieselbe Adresse. Somit stehen Ihnen also zwei Möglichkeiten zur Verfügung. Damit Sie bei diesem Wirrwarr noch die Übersicht behalten können, folgt jetzt eine Tabelle, die die Verwandtschaft zwischen den Zeigern und den Arrays verdeutlicht. Falls Sie wieder einmal einen Quellcode finden, bei dem Sie nicht wissen, was das nun wieder sein soll, blättern Sie einfach zu diesen Tabellen. Es sollen folgende Werte für die erste Tabelle verwendet werden:

```
int n=3;
int array[5]={ 0 }; /* eindim. Array mit Platz für 5 Werte*/
int *ptr = array;    /* int-Zeiger verweist jetzt auf array[0] */
```

Folgendes ist jetzt gleichwertig in Bezug auf den Zugriff von Werten. Betrachten wir zuerst die Möglichkeiten des Zugriffs auf das erste Element:

Tabelle 12.1 Äquivalenz beim Zugriff auf das erste Element

Zeiger-Variante	Array-Variante
<code>*ptr</code>	<code>ptr[0]</code>
<code>*array</code>	<code>array[0]</code>

Als Nächstes folgt die Möglichkeit des Zugriffs auf das n -te Element:

Tabelle 12.2 Äquivalenz beim Zugriff auf das n -te Element

Zeiger-Variante	Array-Variante
<code>*(ptr+n)</code>	<code>ptr[n]</code>
<code>*(array+n)</code>	<code>array[n]</code>

Die nächste Tabelle zeigt alle möglichen Zugriffe auf die Anfangsadresse:

Tabelle 12.3 Äquivalenz beim Zugriff auf die Anfangsadresse

Ohne Adressoperator	Mit Adressoperator
<code>ptr</code>	<code>&ptr[0]</code>
<code>array</code>	<code>&array[0]</code>

Jetzt folgt die Tabelle für den Zugriff auf die Speicheradresse des n -ten Elements:

Tabelle 12.4 Äquivalenz beim Zugriff auf die Adresse des n-ten Elements

Ohne Adressoperator	Mit Adressoperator
<code>ptr+n</code>	<code>&ptr[n]</code>
<code>array+n</code>	<code>&array[n]</code>

Nun folgt noch ein Listing, das alle Punkte nochmals demonstriert:

```
/* ptr_versus_array2.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int n=3;
    /* eindim. Array mit Platz für 5 Werte */
    int array[5]={ 1,2,3,4,5 };
    /* int-Zeiger verweist jetzt auf array[0] */
    int *ptr = array;

    /* 4 Möglichkeiten, um auf das erste Element zuzugreifen */
    printf("%d ", *ptr);
    printf("%d ", ptr[0]);
    printf("%d ", *array);
    printf("%d\n", array[0]);
    /* 4 Möglichkeiten, um auf das n-te Element zuzugreifen */
    printf("%d ", *(ptr+n));
    printf("%d ", ptr[n]);
    printf("%d ", *(array+n));
    printf("%d\n", array[n]);
    /* 4 Möglichkeiten, um auf die Anfangsadresse zuzugreifen */
    printf("%p ", ptr);
    printf("%p ", &ptr[0]);
    printf("%p ", array);
    printf("%p\n", &array[0]);
    /* 4 Möglichkeiten, um auf die Adresse des n-ten Elements
       zuzugreifen */
    printf("%p ", ptr+n);
    printf("%p ", &ptr[n]);
    printf("%p ", array+n);
    printf("%p\n", &array[n]);
    return EXIT_SUCCESS;
}
```

12.13 Der »restrict«-Zeiger

Mit dem C99-Standard wurde der Typqualifizierer `restrict` neu eingeführt. Mit diesem Schlüsselwort können Sie Zeiger qualifizieren, sogenannte `restrict`-Zeiger. Der `restrict`-Zeiger hat eine enge Beziehung zu dem Speicherobjekt, auf das er verweist. Mit dem Qualifizierer geben Sie quasi vor, dass während der Lebensdauer des Zeigers das Speicherobjekt, auf das dieser verweist, nicht verändert werden kann und der Zugriff nur über diesen `restrict`-Zeiger erfolgen darf. Einfachstes Beispiel:

```
int * restrict iRptr = malloc (sizeof (int) );
```

Damit geben Sie praktisch vor, dass Sie den von `malloc()` zurückgegebenen reservierten Speicher nur mit dem Zeiger `iRptr` verwenden. Wohlgemerkt: Mit dem Qualifizierer `restrict` geben Sie nur dem Compiler das Versprechen, dass Sie auf das Speicherobjekt ausschließlich mit diesem Zeiger zurückgreifen. Jede Manipulation außerhalb des `restrict`-Zeigers, und sei es nur lesend, ist unzulässig.

Zu überprüfen, ob der `restrict`-Zeiger richtig verwendet wird, Sie also nur über diesen Zeiger auf ein Speicherobjekt zugreifen, ist Ihre Aufgabe. Der Compiler kann nicht überprüfen, ob Sie Ihr *Versprechen* eingehalten haben. Fall Sie die Regeln nicht einhalten, gibt es zwar keine Fehlermeldung des Compilers und häufig auch keine Probleme bei der Ausführung des Programms, aber dennoch ist das Verhalten laut Standard undefiniert. Abgesehen davon: Mit oder ohne den `restrict`-Zeiger bleibt die Ausführung des Programms dieselbe.

Der Vorteil des `restrict`-Zeigers ist es, dass Sie es dem Compiler ermöglichen, Optimierungen des Maschinencodes durchzuführen, die sonst bei Zeigern zu Problemen führen können. Allerdings muss der Compiler auch diesem Hinweis nicht nachkommen und kann den Qualifizierer `restrict` auch ignorieren.

Der `restrict`-Zeiger kann auch sehr gut bei Funktionen verwendet werden, um anzuzeigen, dass sich zwei Zeiger in der Parameterliste nicht überlappen dürfen, sprich, dasselbe Speicherobjekt verwenden. Beispielsweise ist bei

```
int flaeche( int * w, int * b ) {  
    /* ... */  
}
```

nicht klar angegeben, ob sich die beiden Speicherobjekte, auf die die Zeiger `w` und `b` verweisen, überlappen dürfen oder nicht. Mit dem neuen Qualifizierer `restrict` ist dies jetzt sofort erkennbar:

```
int flaeche( int * restrict w, int * restrict b ) {  
    /* ... */  
}
```

Wird trotzdem versucht, die Funktion mit sich überlappenden Speicherobjekte aufzurufen, ist das weitere Verhalten undefiniert. Ein ungültiger Aufruf kann beispielsweise wie folgt aussehen:

```
int val, x=20, y=10;  
// Unzulässig, wegen den restrict-Zeigern,  
// zwei gleiche Speicherobjekte werden verwendet,  
// die sich somit überlappen  
val = flaeche( &x, &x );  
// OK, zwei verschiedene Speicherobjekte  
val = flaeche( &x, &y );
```

Vom `restrict`-Zeiger wird mittlerweile auch rege in der Standard-Bibliothek Gebrauch gemacht. Beispielsweise sieht die Syntax der Funktion `strncpy()` wie folgt aus:

```
#include <string.h>  
  
char *strncpy( char * restrict s1,  
               const char * restrict s2,  
               size_t n );
```

Die Funktion kopiert `n` Bytes vom Quellarray `s2` in das Zielarray `s1`. Dadurch, dass die beiden Zeiger als `restrict` deklariert sind, müssen Sie beim Aufruf der Funktion beachten, dass die Zeiger nicht auf dieselben Speicherobjekte verweisen, sprich, sich nicht überlappen. Betrachten Sie dazu folgendes Beispiel:

```
char arr1[20];  
char arr2[] = { "Hallo Welt" };  
// Ok, 10 Zeichen von arr2 nach arr1 kopieren  
strncpy( arr1, arr2, 10 );
```

```
arr1[10] = '\\0'  
// Unzulässig, Speicherbereiche überlappen sich,  
// das gibt nur Datensalat.  
strncpy( arr1, arr1, 5 );
```

Ein weiteres klassisches Beispiel von Funktionen der Standardbibliothek sind die Funktionen `memcpy()` und `memmove()` in der Headerdatei `<string.h>`. Die Syntax der Funktionen lautet:

```
#include <string.h>  
  
void *memcpy( void * restrict s1,  
              const void * restrict s2,  
              size_t n );  
  
void *memmove( void *s1,  
              const void *s2,  
              size_t n );
```

Bei diesen beiden Funktionen kann man sehr schön sehen, dass sich bei `memcpy()` der Quell- und der Zielbereich nicht überlappen dürfen, weil sonst die `restrict`-Regel verletzt würde. Bei der anderen Standardfunktion `memmove()` hingegen dürfen sich der Quell- und der Zielbereich überlappen.

Hinweis

Da sich der C99-Standard noch nicht auf allen Compilern komplett durchgesetzt hat, wurde bei den Beispielen in diesem Buch auf `restrict`-Zeiger verzichtet. Die Syntaxbezeichnungen der Standardfunktionen hingegen wurden bereits im C99-Standard verfasst.

13 Kommandozeilenargumente

Wenn aber z. B. ältere Programme überholt werden müssen, wird der Umgang mit der Kommandozeile wieder wichtig. Bei Betriebssystemen wie Linux, UNIX oder FreeBSD ist es nach wie vor üblich (teilweise sogar unerlässlich), sehr viel mit einer Kommandozeile zu arbeiten.

Beim Schreiben eines Konsolenprogramms für Linux/UNIX oder MS-DOS (Eingabeaufforderung) sind Kommandozeilenparameter immer noch eines der wichtigsten Konzepte. Da Konsolenprogramme keine grafische Oberfläche besitzen, stellt die Kommandozeile die wichtigste Schnittstelle zwischen dem Anwender und dem Programm dar.

Hinweis

Dem Programm werden Argumente beim Aufruf übergeben. Unter MS-Windows können Sie hierfür die Eingabeaufforderung `cmd.exe` verwenden. Unter Linux genügt eine einfache Konsole bzw. Shell. Sofern Sie Entwicklungsumgebungen (IDEs) verwenden, müssen Sie Kommandozeilenargumente anders übergeben. Viele Entwicklungsumgebungen bieten hierfür beim Menü Ausführen noch ein Untermenü Parameter oder so ähnlich (abhängig von der IDE), um die Argumente noch vor dem Programmstart festzulegen. Mehr dazu finden Sie auf der Buch-CD oder unter der Webseite <http://www.pronix.de/pronix-1168.html>.