11 Zeiger

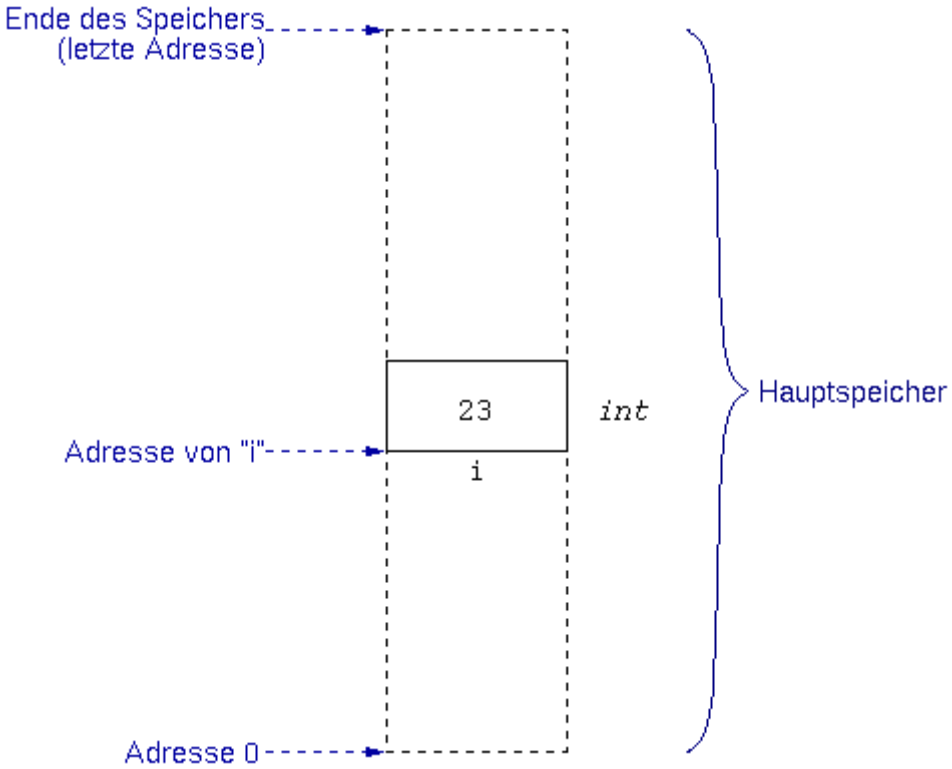
11.1. Grundlagen

Zeiger sind Variablen und benötigen wie alle anderen Variablen Speicherplatz (4Byte). Ein Zeiger ist eine Variable, die die Adresse eines (beliebigen) anderen Objektes enthält. Man kann auf dieses Objekt indirekt über einen Zeiger zugreifen. Diese Speicheradressen sind entweder Adressen von **anderen Variablen** oder **Adressen von Funktionen**.

Der Name "Zeiger" (engl. "Pointer") kommt von der Vorstellung, dass Zeiger auf eine andere Variable zeigen.

Verwendung von Zeigern in C:

- bei der Parameterübergabe
- bei der Bearbeitung von Arrays
- für Textverarbeitung
- Zugriff auf spezielle Speicherzellen



Grundoperationen mit Zeigern:

Deklaration einer Zeigervariablen, die auf eine Variable des angegebenen Typs zeigt:

int *pc;

Eigentlich sollte man ja `int* pc` schreiben, denn gemeint ist ein *Pointer auf* `int`, der den Name *pc* hat. Aber seit Anbeginn der Sprache C wird das Sternchen direkt vor den Variablennamen gesetzt (dem Compiler ist es übrigens egal).

Zeiger zeigen immer auf Objekte eines bestimmten Typs. Sie müssen daher deklariert werden. Auch in der Zeigerdefinition wird der Operator ***** verwendet. Zeigerdefinitionen kann man als Muster verstehen:

```
int *px;           /* px ist Zeiger auf int */
char *zs;          /* zs ist Zeiger auf char */
int x, y;
px = &x;      /* px = (Speicher-)Adresse der Variablen x */
y = *px;           /* y = Wert der Variablen x */
```

Zeiger sind also an bestimmte Objekttypen gebunden. Ein Zeiger auf `int` kann also beispielsweise nur Adressen von `int`-Variablen aufnehmen. Eine Ausnahme bildet der "Generic Pointer", der auf ein beliebiges Objekt zeigen kann.

```
void *pc;
```

In Verbindung mit Zeigern werden hauptsächlich zwei zueinander inverse Operatoren benutzt:

1. Der Adreßoperator `&`, der angewendet auf ein Objekt, die Adresse dieses Objekts liefert.

```
pc=&c;
```

2. & kann auf Variablen und Arrayelemente angewendet werden, nicht aber auf Arraynamen selbst (Warum? Ein Arrayname hat keine Adresse, er ist eine Adresse!). Ebenso haben natürlich Variablen in der Speicherklasse `register` keine Adressen.
3. Der Inhaltsoperator `*`, der angewendet auf einen Zeiger das Objekt liefert, das unter dieser Adresse abgelegt ist.

```
c=*pc;
*pc=5;
```

```
y = *px;          /* y erhält den Wert des Objektes, dessen Adresse in px
steht */
px = &x;          /* px "zeigt" nun auf x */
y = *px;          /* y = x; */
```

Einige Grundregeln:

Die Kombination ***Zeiger** kann in Ausdrücken überall dort auftreten, wo auch das Objekt, auf das der Zeiger zeigt, selbst stehen könnte:

```
y = *px + 10;
y = *px + *px;
printf("%d\n", *px);
*px = 0;
py = px;    /* falls py auch Zeiger auf int */
```

Bei der Verwendung des Operators ***** muss man die Operatorrangfolge und -assoziativität genau beachten. Dies erscheint zunächst etwas schwierig, da dieser Operator ungewohnt ist. Hier einige Beispiele mit dem ***** Operator und anderen Operatoren:

```
y = *px + 1;    /* Inhalt von px plus 1                      */
y = *(px+1);    /* Inhalt der Adresse px+1                   */
*px += 1;       /* Inhalt von px = Inhalt von px plus 1      */
(*px)++;        /* Inhalt von px inkrementieren              */
*px++;          /* wie *(px++); (Assoziativität)
                   Inhalt der Adresse px; px = px plus 1     */
*++px;          /* Inhalt der Adresse px+1; px = px plus 1 */
```

Besonders wichtig:

1. * und & haben höhere Priorität als arithmetische Operatoren.
2. Werden * und ++ direkt hintereinander verwendet, wird der Ausdruck von rechts nach links abgearbeitet.

Zeiger haben nur dann sinnvolle Werte, wenn sie die Adresse eines Objektes oder NULL enthalten. NULL ist eine globale symbolische Konstante, die in der Standardbibliothek definiert ist und überall als NULL-Zeiger benutzt werden kann. Für den Zeigerwert NULL ist garantiert, dass er nirgends hinzeigt. NULL ist definiert als eine Adresse mit dem Wert 0; Sie sollten auch immer NULL verwenden, **niemals** den Zahlenwert 0, denn es ist nicht sicher, ob die Länge einer Integer-Variablen auch der Länge einer Adresse entspricht.

11.2. Zeigerarithmetik

Es können mit Zeigern bestimmte arithmetische Operationen und Vergleiche durchgeführt werden. Es sind natürlich nur solche Operationen erlaubt, die zu sinnvollen Ergebnissen führen. Zu Zeigern dürfen ganzzahlige Werte addiert und es dürfen ganzzahlige Werte subtrahiert werden. Zeiger dürfen in- und dekrementiert werden und sie dürfen voneinander subtrahiert werden (Dies ist i. a. nur sinnvoll, wenn beide Zeiger auf Elemente des gleichen Objektes (z. B. ein Array) zeigen.).

Man kann Zeiger mittels der Operatoren >, >=, <, <=, **!=** und == miteinander vergleichen. Wie bei der Zeigersubtraktion ist das aber i. a. nur dann sinnvoll, wenn beide Zeiger auf Elemente des gleichen Arrays zeigen. Eine Ausnahme bildet hier der Zeigerwert NULL. denn viele Bibliotheksfunktionen liefern im Fehlerfall einen NULL-Zeiger zurück.

Alle anderen denkbaren arithmetischen und logischen Operationen (Addition von Zeigern, Multiplikation, Division, Shifts oder Verwendung von logischen Operatoren, sowie Addition und Subtraktion von **float** oder **double** Werten) sind mit Zeigern **nicht** erlaubt.

Es ist sinnvoll, Zeigervariablen mit NULL zu initialisieren, d. h. Zeigern, die keinen definierten Wert enthalten, wird der Wert NULL zugewiesen.

Wie funktioniert nun aber die Zeigerarithmetik? Sei `ptr` ein Zeiger und `N` eine ganze Zahl, dann bezeichnet `ptr + N` das n-te Objekt im Anschluss an das Objekt, auf das `ptr` gerade zeigt. Es wird also nicht der Wert `N` zu `ptr` direkt addiert, sondern `N` wird vorher mit der Typlänge des Typs, auf den `ptr` zeigt, multipliziert. Dieser Typ wird aus der Deklaration von `ptr` bestimmt.

11.3. Zeiger und Array

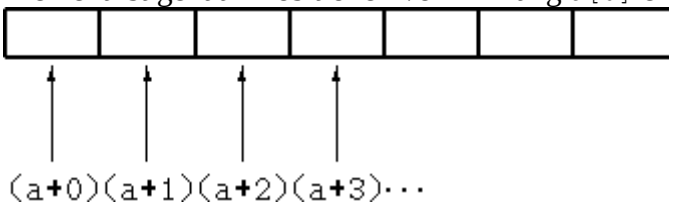
Zeiger und eindimensionale Arrays

In C besteht zwischen Zeigern und Feldern eine ausgeprägte Korrespondenz. Jede Operation, die durch die Indizierung von Feldelementen formuliert werden kann, kann auch mit Zeigern ausgedrückt werden. Die Zeigerversion wird im allgemeinen schneller sein; sie ist aber am Anfang schwerer zu verstehen.

`a` ist ein Integer-Array und `pa` ein Zeiger auf den Datentyp `int`, deklariert durch

```
int a[10];          /* Array mit 10 Elementen */
int *pa;            /* Zeiger auf int */
```

Zur Erinnerung:
`a` hat 10 Elemente, wobei `a[0]` das erste und `a[9]` das letzte Element ist. `a[i]`, das i-te Element ist genau i Positionen vom Anfang `a[0]` entfernt.



Die Zuweisung

```
pa = &a[0];
```

bewirkt, dass `pa` auf das `0`-te Element von `a` zeigt, d. h. dass `pa` die Adresse von `a[0]` enthält. Statt der obigen Zuweisung kann auch die folgende

```
pa = a;
```

verwendet werden, wo auf der rechten Seite der Feldname `a` steht. Die einzelnen Elemente des Feldes `a` können auf drei verschiedene Weisen angesprochen werden:

```
a[i]        i-tes Feldelement
*(pa+i)     Pointer pa + i * (Laenge eines Elementes von a)
*(a+i)      Arrayanfang + i * (Laenge eines Elementes von a)
```

`a` ist also in C äquivalent zu `&a[0]`. So kann man statt `pa = &a[0]` auch schreiben `pa = a`. Die Verwandtschaft zwischen Arrays und Pointern kann beim Programmieren recht praktisch und effizient sein, führt aber auch zu unsauberem Stil und unverständlichen Programmen.

```
}
```

Die Korrespondenz von Indizieren und Zeigerarithmetik ist daher sehr eng: Die Schreibweise `pa + i` bedeutet die Adresse des `i`-ten Objekts hinter demjenigen, auf welches `pa` zeigt. Es gilt:

Jeder Array-Index-Ausdruck kann auch als Pointer-Offset-Ausdruck formuliert werden und umgekehrt.
Der Ausdruck `pa + i` bedeutet nicht, daß zu `pa` der Wert `i` addiert wird, sondern `i` legt fest, wie oft die Länge des Objekt-Typs von `pa` addiert werden muss.

Es besteht jedoch ein gravierender Unterschied zwischen Array-Namen und Zeigern:

- Ein Zeiger ist eine Variable, deren Wert jederzeit geändert werden darf, z.B.:

```
int x,*px;
px = &x;     /* Veränderung ist jederzeit zulässig */
```

- Ein Array-Name ist eine Zeiger-Konstante, ihr Wert ist nicht veränderbar, z. B.:

```
int z, y[5];
y = &z;      /*  U n z u l ä s s i g */
```

Zeiger und zweidimensionale Arrays

Zur Erinnerung: bei statischer Speicherbelegung sind mehrdimensionale Arrays intern eindimensional angelegt, die Arrayzeilen liegen alle hintereinander.

Bsp:

```
double x[10][5] ;

double **ptr  = x ;
```

Folgende Ausdrücke sind dann äquivalent:

*x[0] = **x = x[0][0]

*ptr[0] = **ptr = ptr[0][0]

Der Unterschied hier ist wieder oben steht eine Zeiger Konstante und unten ein veränderbarer Zeiger.

*x[i] = **(x+i) = x[i][0]

*ptr[i] = **(ptr+i) = ptr[i][0]

*(x[0] + j) = *(*x + j) = x[0][j]

*(ptr[0] + j) = *(*ptr + j) = ptr[0][j]

*(x[i] + j) = *(*(x+i) + j) = x[i][j]

*(ptr[i] + j) = *(*(ptr+i) + j) = ptr[i][j]

11.4. Zeiger-Array

Zeiger-Arrays sind Arrays deren Komponenten Zeiger sind. Genauso wie man Vektoren aus den Grunddatentypen (char, int, float und double) bilden kann, kann man dies auch mit Zeigern tun. Ein Vektor von Zeigern wird so definiert:

```
Grunddatentyp *Vektorname []
```

gelesen von rechts nach links (wegen des Vorranges von []): Vektor von Zeigern. Dagegen ist

```
(*Vektorname) []
```

ein Zeiger auf einen Vektor!

Die Komponenten des Vektors können natürlich auch Adressen von Arrays sein. Damit ergibt sich eine Ähnlichkeit zu mehrdimensionalen Arrays.

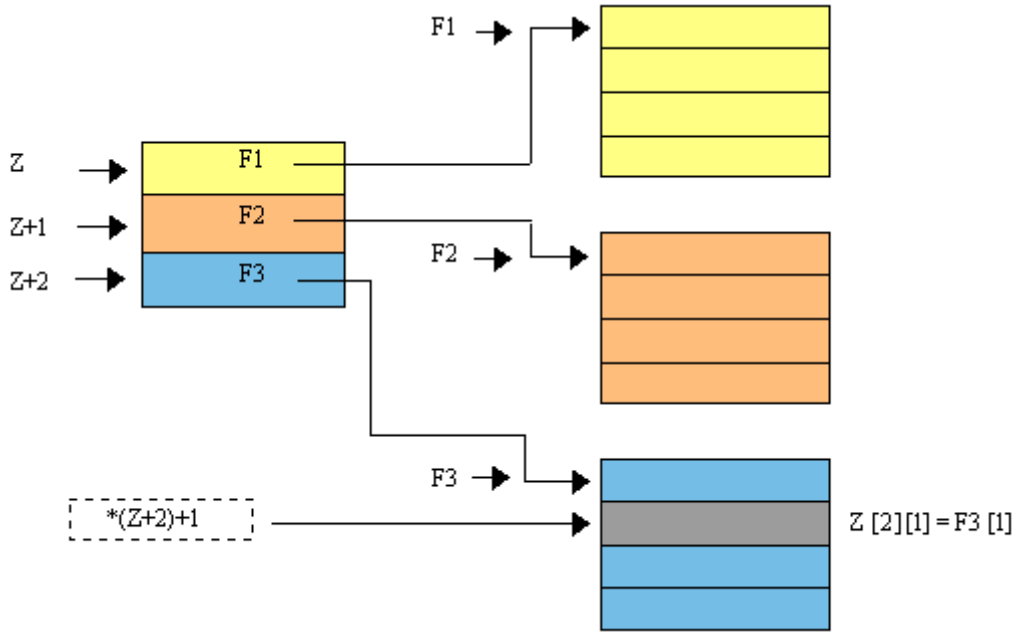
Beispiel:

```
char *Z[3];    /* Definition für ein Array mit 3 Elementen,
                  die jeweils Zeiger auf char-Typen sind.
                  Z ist also ein Zeiger auf ein Array mit
                  3 char-Zeigern */

char F1 [4],   /* Definition für 3   */
     F2 [4],   /* char-Arrays mit jeweils */
     F3 [4];   /* der Länge 4        */

Z[0] = F1;   /* Wertzuweisung an die char-Zeiger */
Z[1] = F2;
Z[2] = F3;
```

Die Situation stellt sich bildlich folgendermaßen dar:



Man sieht folgende Analogie zwischen Zeiger-Arrays und mehrdimensionalen Arrays:

Z+i	ist ein Zeiger auf das (i+1)-te Element des Zeiger-Arrays Z
*(Z+i) = Z[i]	ist das Array, auf das das (i+1)-te Element des Zeiger-Arrays zeigt, also ein Zeiger auf das erste Element dieses Arrays.
*(Z+i) + j = Z[i] + j	ist ein Zeiger auf das (j+1)-te Element des Arrays, auf das das (i+1)-te Element des Zeiger-Arrays Z zeigt.
((Z+i) + j) = *(Z[i] + j) = = Z [i][j]	ist das (j+1)-te Element des Arrays, auf das das (i+1)-te Element des Zeiger-Arrays Z zeigt.

Das obige Beispiel geht davon aus, dass die Elemente des Zeiger-Arrays alle auf Arrays gleicher Größe verweisen. Dies bedingt die große Ähnlichkeit zu mehrdimensionalen Arrays. Bei häufigen Anwendungen von Zeiger-Arrays besitzen die referenzierten Arrays unterschiedliche Längen, z.B. verschieden lange Strings.