

2.5 Zeiger



Motivation:

Zeiger enthalten die Adresse eines Objektes eines bestimmten Datentyps. Somit ist es möglich, auf diese Objekte "indirekt" mittels Zeiger zuzugreifen. Zeiger spielen zentrale Rolle in C.

2.5 Zeiger (Pointer)



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.

2.5.1 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).

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;
```

2.5.1 Zeiger deklarieren



- Es hat sich folgende Schreibweisen eingebürgert
- Mit dieser Schreibweise wird ein Fehler deutlich
- Hier könnte man fälschlicherweise annehmen, es seien zwei Zeiger deklariert worden.

```
int *zeiger1;  
int *zeiger2;  
int *zeiger3;
```

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

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

2.5.1 Zeiger deklarieren



- Der **Zeigeroperator &** liefert die Adresse seines Operanden.

```
int a = 5;  
int *pi;  
pi = &a; /* pi zeigt auf a */
```

- Der **Zeigeroperator *** löst eine Adressbezugnahme auf, d.h. er ermöglicht, auf das Datenobjekt "indirekt" zuzugreifen, auf welches der Operand verweist.

```
int x, y, *pint;  
x = 10;  
pint = &x;  
y = *pint - 1;      /* y = x - 1; */  
*pint = 0;          /* x = 0; */  
*pint += 2;         /* x += 2; */
```

2.5.2 Zeiger initialisieren



Wird 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.

```
int x = 5;
```

Adresse: 0022FF7C	Name: x	Wert: 5


2.5.2 Zeiger initialisieren



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

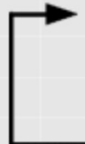
Adresse: 000000FF	Name: x	Wert: 5
Adresse: 0000003C	Name: y	Adresse: NULL

```
y = &x;
```



Adresse: 000000FF	Name: x	Wert: 5
Adresse: 0000003C	Name: y	Adresse: 000000FF

```
*y = 10;
```



Adresse: 000000FF	Name: x	Wert: 10
Adresse: 0000003C	Name: y	Adresse: 000000FF

2.5.3 NULL Zeiger



Der NULL-Zeiger ist ein vordefinierter Zeiger, dessen Wert sich von einem regulären Zeiger unterscheidet. Er wird vorwiegend bei Funktionen zur Anzeige und Überprüfung von Fehlern genutzt, die einen Zeiger als Rückgabewert zurückgeben.

- NULL ist ein Zeiger auf 0.
- NULL ist ein typenloser Zeiger auf 0.
- Es gibt keinen NULL-Zeiger, sondern nur NULL, und das ist eben 0.

Möglichen defines für NULL:

```
#define NULL 0  
#define NULL 0L  
#define NULL (void *) 0 ← häufigste Definition
```


2.5.4 Grösse vom Zeigern



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, vier** oder **acht** Bytes benötigt. Hängt von der **Bitbreite des BS** ab.

```
#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;
}
```

2.5.5 Zeigerarithmetik



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

- inkrementieren
- Dekrementieren

```
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.

2.5.6 Zeiger als Funktionsparameter



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.

```
#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;
}
```

2.5.6 Zeiger als Funktionsparameter



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.

```
#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;
}
```

2.5.7 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 Rückgabetyt deklariert sind, geben logischerweise auch nur die Anfangsadresse des Rückgabetyts zurück.

```
Zeiger_Rückgabetyt *Funktionsname (Parameter)
```

2.5.7 Zeiger als Rückgabewert



Achtung:

```
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.

2.5.8 Array's und Zeiger



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.

```
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;
}
```


2.5.8 Array's und Zeiger



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

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.

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


2.5.8 Array's und Zeiger



`ptr = element;`

`*(ptr+1);`

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: 022FF60

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: 022FF64

2.5.9 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'.

2.6 Explizite Typkonvertierung



C bietet die Möglichkeit zur expliziten Typkonvertierung mit der sogenannten **cast-Konstruktion**

```
(typ_spezifikation) ausdrück
```

kann der Typ von `ausdruck` erzwungen werden. Dies entspricht praktisch der impliziten Typumwandlung bei einer Wertzuweisung.

```
int i1 = 100;
```

```
char* = (char*) &i1;
```