[Zeiger] Übersicht

- Zeiger
- Zeigerarithmetik
- Zeiger und Felder
- Zeigervektoren
- Zeiger auf Zeiger
- Zeiger als Funktionswerte
- Parameter des Hauptprogramms
- Funktionszeiger

[Zeiger] *Dereferenzierung*

```
Beispiel Wertzuweisung:

v = e;
```

- Name einer Variable: repräsentiert Adresse
- An die Adresse werden Werte geschrieben
- rechte Seite des Ausdrucks: wenn Variablen angegeben werden, interessieren Werte, nicht deren Adresse
  => automatische Dereferenzierung
- linke Seite des Ausdrucks: hier macht nur eine Wertzuweisung an eine Adresse Sinn, daher wird hier keine Dereferenzierung vorgenommen.

```
Noch ein Beispiel:
(automatische Dereferenzierung, Inkrementierung, Speichern des
Ergebnisses an die Adresse von i)
i++;
```

[Zeiger] Spezielle Operatoren

```
&   Adressoperator

*   Dereferenzierungsoperator
```

```
/* Ein bekanntes Beispiel
    für die Verwendung des
    Adressoperators */


void main()
{
    int i;
    int j;
    scanf("%d",&i);
}
```

```
#include <stdio.h>
void meinscanf(int*);
int main()
{
	int a = 0;
	printf("vorher:  %d\n",a);
	meinscanf(&a);
	printf("nachher: %d\n",a);
	return 0;
}

void meinscanf(int *i)
{
	char c = 0;
	int d = 0;
	while ((c = getchar()) != '\n')
	{
		if (c >=48 && c <=57)
		{
			d *= 10;
			d += c - 48;
		}
	}
	*i = d;
}
```

[Zeiger] Spezielle Operatoren

```
int* i;
int j = 17;

i = &j;
```

```
int i;
int j;
int *k;

k = i < j ? &i : &j;
*k = 10;
```

- Warum muss ein Zeiger überhaupt einen Typ haben (wo der Wert doch immer eine Adresse ist)

[Zeiger] Zeiger und Felder

- Feldname (ohne Indizes) ist eine Zeigerkonstante
- Dieser Zeiger repräsentiert die Adresse der ersten Feldkomponente.
- Der Zugriff auf die Feldkomponenten erfolgt durch Berechnung der Speicherabbildungsfunktion

```
int v[10];
int *p;

/* nachfolgende Ausdrücke sind äquivalent */

p = v;
p = &v[0];

/*  nächster Ausdruck ist nach Standard auch möglich
    (wenngleich eine compiler-Warnung ausgegeben wird*/

p = &v;
p[0];
```

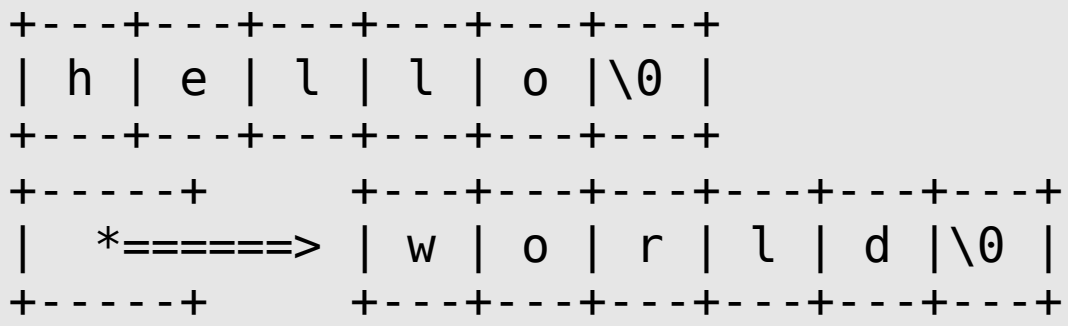
[Zeiger] Zeiger und Felder

Die Funktionalität von Zeigern (char*) und Feldern ähneln sich zwar stark, es gibt aber keien vollständige Entsprechung!

char a[10] stellt Platz zur Speicherung von 10 char bereit, char* a erzeugt einen Zeiger, der auf einen char zeigen kann.

a[3] bzw p[3] führt in beiden Fällen womöglich zum gleichen Ergebnis, es wird aber unterschiedlicher Maschinencode erzeugt!

```
char a[] = "hello";
char *p = "world";
```



[Zeiger] Zeigerbeispiele

```
#include <stdio.h>
#define WERTE 1
#define FEHLER 0
#define ADRESSEN 0
#define ZEIGER 0
int main()
{
	int a = 13;
	int b;
	int c;
	int* d;
	char meinstring[]="123TestText";
	int meinintfeld[]={1,2,3,4,5};
	b = a;
	c = &a;
	d = &a;
#if WERTE
	printf("Werte:\n");
	printf("a: %d\n",a);
	printf("b: %d\n",b);
	printf("c: %x\n",c);
	printf("d: %x\n",d);
#endif

#if FEHLER
	printf("c: %x\n",*c);
#endif

#if ADRESSEN
	printf("Adressen:\n);
	printf("a: %x\n",&a);
	printf("b: %x\n",&b);
	printf("c: %x\n",&c);
	printf("d: %x\n",&d);
#endif

#if ZEIGER
	printf("Zeigerwerte:\n");
	printf("(das ist die Adresse der Variable a)\t\t\t\t %x\n", &a);
	printf("(das ist der Wert der Variablen a)\t\t\t\t %d\n", a);
	printf("(auf diese Adresse zeigt der Zeiger)\t\t\t\t %x\n", d);
	printf("(diese Adresse hat der Zeiger)\t\t\t\t\t %x\n", &d);
	printf("(das ist der Wert an der Adresse auf die der Zeiger zeigt):\t %d\n",*d);
#endif
	return 0;
}
```

[Zeiger] Zeigerbeispiele

```
#include <stdio.h>
#define STRING 0
#define FFUNKTION 0

void meinefunktion(int*);

nt main()
{
#if STRING
        d=&meinstring;
        printf("Der String: %s\n",meinstring);
        printf("Oder ein einzelnes Zeichen (das erste): %c\n",meinstring[0]);
        printf("Erstes Zeichen auf das der Pointer zeigt: %c\n",*d);
        printf("Zweites Zeichen auf das der Pointer zeigt: %c\n",*(d+1));
        /* ist gar nicht zweites Zeichen, oder? */
        printf("\njetzt ganz in echt:\nZweites Zeichen auf das der Pointer zeigt: %c\n",*((char*)d+1));
        /*      oder etwas ausfuehrlicher: es wird kein int* sondern ein char* benoetigt
                daher der typecast (explizite Typumwandlung) */
#endif

#if FUNKTION
        meinefunktion(meinintfeld);
        meinefunktion(&a);
        d=meinintfeld;
        meinefunktion(d);
        d=&a;
        meinefunktion(d);
#endif

        return 0;
}

void meinefunktion(int*i)
{
        printf("(dieser Wert steht an der Stelle auf die der Zeiger zeigt):\t%d\n", *i) ;
        printf("(auf diese Adresse zeigt der Zeiger):\t\t\t%x\n", i) ;
        printf("(diese Adresse hat der Zeiger):\t\t\t\t%x\n", &i) ;

}
```

[Zeiger] Zeiger und const

```
const int *p = &i;   /* gleichbedeutend: int const *p */

*p = 2 ;     /* Fehler: referenzierter Wert kann nicht geändert werden */
p = &j;      /* erlaubt: neue Zuweisung an Zeigervariable */
```

```
int * const p = &i;

*p = 2 ;     /* erlaubt: referenzierter Wert wird gändert */
p = &j;      /* Fehler: Zeigervariable konstant */
```

```
int const * const p = &i;

*p = 2 ;     /* Fehler: referenzierter Wert kann nicht geändert werden */
p = &j;      /* Fehler: Zeigervariable konstant */
```

[Zeiger] Zeiger und const, Beispiel

```
void testfunktion(int* fp)
{
    int feld[] = {1,2,3,4};

    /* mit der folgenden Anweisung ist die Adresse,
        welche zuvor noch vom Zeiger beherbergt wurde,
        unwiderruflich verloren (zumidest im Kontext dieser Funktion */
    fp = feld;
}

/* mit einer Funktion dem folgendenen Prototypen genügend wäre das nicht
    möglich (compiler bemerkt und bemängelt den Fehler) */

void testfunktion(int* const fp);
```

[Zeiger] Zeigerarithmetik

```
int v[10];
int *p = v;

/* nachfolgende Ausdrücke sind äquivalent */

v[7]   = 9;
*(p+7) = 9;
*(v+7) = 9;

/* oder auch: */
p = v + 7;
*p = 9;
```

```
/* damit sind auch folgende Ausdrücke gültig und machen
   (vor allem bei Zeigern auf Felder) Sinn */
p++;
p+3;
```

[Zeiger] Zeigerarithmetik

```
float vektorsumme(const float v[], int laenge)
{
    float summe = 0;
    while (--laenge >= 0)
        summe += v[laenge];
    return summe;
}
```

```
float vektorsumme(const float *v, int laenge)
{
    float summe = 0;
    while (laenge--)
        summe += *v++;
    return summe;
}
```

- Im zweiten Beispiel zeigt sich erst durch die Funktionslogik, daß der Parameter ein Vektor ist!

[Zeiger] Zeigerarithmetik

```
/* Differenz zweier Zeiger (gleichen Typs) */
/* es wird die Differenz gebildet der Indizes der
   Komponenten */
#include <stddef.h>

...

int v[10];
int *vp1 = &v[0];
int *vp2 = &v[4];
ptrdiff_t d1 = vp1 – vp2;     /* -4 */
ptrdiff_t d2 = vp2 – vp1;     /*  4 */
```

```
/* Addition von Zeigern ist nicht möglich */
/* (und macht im Gegensatz zur Subtraktion auch weniger Sinn) */

int v[20], *start = v, *ende = v + 20, *mitte;

mitte = start + (ende-start)/2;

mitte = (start + ende) / 2;   /* nicht zulässig !*/
```

[Zeiger] Zeiger als Parameter

```
/* lexikographischer Vergleich zweier Strings */
/* Funktionswert <0  wenn s < t
                 ==0 wenn s == t
                 >0  wenn s > t */

int strcmp1(const char s[], const char t[])
{
    int i = 0;
    while ( s[i] == t[i] && s[i] )  /* False am Ende zweier gleicher Strings */
        i++;
    return s[i] - t[i];
}
```

```
int strcmp2(const char *s, const char *t)
{
    while ( *s == *t && *s )
        s++,t++;
    return *s - *t;
}
```

[Zeiger] Zeigerprobleme

- Es ist möglich, Differenzen zweier Zeiger zu berechnen, aber nicht immer sinnvoll

```
int v1[10], v2[20];

if (v1 < v2)
    ....
```

```
/* etwas sinnvoller */
char v1[10];
char* p = v1;
while (*p !='\0') p++;
if (p - &v1[5] > 0)
    ....
```

Hier werden die Adressen verglichen, also welches Feld vorher im Speicher steht. Das ist nicht deterministisch und fast immer auch nicht beabsichtigt (ausser Operationen auf dem gleichen Feld).

- Typumwandlungen

```
int i, *ip = &i;
float *fp;
i=13;
fp = (float *) ip;
printf("%d\n",i);
*fp = 6.5f;
printf("%d\n",i);
```

[Zeiger] Zeigerprobleme

```
char str1[] = "hello";

char *str2 = "hallo";    /* str2 ist ein Zeiger auf den
                            Anfangswert des Vektors */

str1[0]='b';             /* möglich */
str2[0]='b';             /* sehr wahrscheinlich ein Programmabsturz */

str1 = str2;             /* ungültig, da str1 eine Zeigerkonstante ist

str2 = str1;             /* möglich; ABER: der Vektor, auf den str2 zuvor
                            gezeigt hat, ist verloren (obwohl er noch
                            im Speicher steht) */

char const str3[] = "holla";
                         /* so kann der Vektor nicht einfach
                            'verloren' gehen */

const char * const str4 = "hulla";
                         /* jetzt kann weder der Vektor 'umgebogen'
                            noch der Inhalt des Vektors verändert
                            werden */
```

[Zeiger] Zeigerprobleme

```
char str1[] = "hallo";
/*
   char-Feld initialisiert mit einer
   StringKonstante
*/

char *str4 = "hulla";
/*
   Zeigervariable, zeigt auf eine String-Konstante
   Aaaaber: diese Stringkonstante ist nicht
            notwendigerweise modifizierbar.

   char interner_string = "hulla";
   char *str4 = &interner_string[0];

   Diese „interne“ Stringkonstante liegt sehr
   wahrscheinlich in einem readonly-Bereich des Speichers
*/
```

[Zeiger] Zeigervektoren

```
const char ziffern_variante1[][7] =
{
    "null" , "eins" , "zwei" ,"drei" , "vier" ,
    "fuenf" , "sechs" , "sieben" , "acht" , "neun"
};
```

```
const char *const ziffern_variante2[] =
{
    "null" , "eins" , "zwei" ,"drei" , "vier" ,
    "fuenf" , "sechs" , "sieben" , "acht" , "neun"
};
```

```
void umsetzen (int wert)
{
    um_rek(wert, FALSE);
}

void um_rek(int wert, int trennen)
{
    if (wert > 10)
        um_rek(wert / 10, TRUE);
    printf("%s",ziffern_variante1[wert%10]);
    if(trennen)
        printf(" = ");
}
```

[Zeiger] Zeiger auf Zeiger

```
m[i][j]

(*(m + i))[j]

*(m[i] + j)

*(*(m + i) + j)



/* und entsprechend sind auch Zeiger auf Zeiger auf Zeiger auf
    Zeiger denkbar */

int i=13;
int* p1 = &i;
int** p2 = &p1;
int*** p3 = &p2;
int**** p4 = &p3;

printf("%d\n",****p4);
```

[Zeiger] Zeiger als Funktionswerte

```
#include <stdio.h>
char* mystringfunction()
{
    return "Hallo";
}

void main()
{
    char* mystring;
    mystring = mystringfunction();
    printf("%s\n",mystringfunction());
    printf("%s\n",mystring);

    printf("%c\n",mystring[2]);
    printf("%c\n",mystringfunction()[2]);
}
```

```
char* strfind(const char* string, char
character)
{
    while (*string != character)
        if(*string ++ == '\0')
            return NULL;
    return string;
}
```

```
void main()
{
    char* cp;
    char string[] =
        "Quitsch";
    cp=strfind(
        string,'i');
    *cp='a';
}
```

[Zeiger] Parameter des Hauptprogramms

```
int main (void){}

int main (int anzahl, char* argument[]){}


argument[0]: Name des Programms oder leerer String, wenn
             der Name von der Umgebung nicht geliefert wird
argumente[1] – argumente[anzahl-1]:
             Kommandozeilenparameter
argumente[anzahl]: Nullzeiger
```

```
/* Gib alle Kommandozeilenparameter aus (mit Programmname) */
#include <stdio.h>

int main(int anzahl, char* argument[])
{
    while(*argument != NULL)
        printf("%s\n", *argument++);
    return 0;
}
```

[Zeiger] Zeiger auf Zeiger und main-Parameter

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define TRUE 1
#define FALSE 0

char sprache = 'e';

const char *const ziffern_variante_d[] =
{
        "null" , "eins" , "zwei" ,"drei" , "vier" ,
        "fuenf" , "sechs" , "sieben" , "acht" , "neun"
};
const char *const ziffern_variante_e[] =
{
        "zero" , "one" , "two" ,"three" , "four" ,
        "five" , "six" , "seven" , "eight" , "nine"
};


int main(int anzahl, char* argument[])
{
        if (strcmp(argument[1],"-Sd")==0)
                sprache='d';
        if (strcmp(argument[1],"-Se")==0)
                sprache='e';
        printf("Gegebene Zahl %d ergibt das: ", atoi(argument[2]));
        umsetzen(atoi(argument[2]));
        return 0;
}
```

```
void um_rek(int wert, int trennen)
{
        char *const *ziffern;
        if(sprache == 'd')
                ziffern = ziffern_variante_d;
        else if (sprache == 'e')
                ziffern = ziffern_variante_e;
        else
                exit(1);
        if (wert > 10)
                um_rek(wert / 10, TRUE);
        printf("%s",ziffern[wert%10]);
        if(trennen)
                printf(" = ");
}
void umsetzen (int wert) {    um_rek(wert, FALSE); }
```

[Zeiger] Zeiger auf Funktionen

```
funktionstyp (*name)(parametertyp , ... , parametertyp);
```

```
/*  Zeiger auf eine Funktion, welcher als Parameter ein
    char-Wert übergeben wird und selbst int als Funktionswert
    besitzt: */

int (*fptr)(char);

/*  Die Klammern um *fptr sind notwendig und unterscheiden
    diesen Ausdruck von einer Funktionsdeklaration */
```

```
/* Implementierung von Funktionszeigern (eine Möglichkeit) */

typedef rückgabewert (*meinfunktionszeigername)(parameter1,...,parameterN);
```

[Zeiger] Zeiger auf Funktionen: Beispiel

```
#include <stdio.h>

typedef double (*zweiDimFun)(double, double);

double summe (double a, double b)
{ return a + b; }

double produkt(double a, double b)
{ return a * b;  }

void main(void)
{
        zweiDimFun funktionsZeiger;
        unsigned int op = 0;
        double a = 0.0 , b = 0.0;

        printf("Eingabe: Operator Argument1 Argument2\n"
                        "Operatoren: 0 = Summe    1 = Produkt\n");
        scanf("%u %lf %lf",&op, &a, &b);

        if (op == 0)
                funktionsZeiger = &summe;
        else
                funktionsZeiger = &produkt;

        printf("Ergebnis: %f\n", (*funktionsZeiger)(a, b));
}
```

[Zeiger] Zeiger auf Funktionen: Beispiel

```
#include <stdio.h>

typedef double (*zweiDimFun)(double, double);

double summe (double a, double b)
{ return a + b; }

double produkt(double a, double b)
{ return a * b;  }

void main(void)
{
        zweiDimFun funktionsZeiger;
        unsigned int op = 0;
        double a = 0.0 , b = 0.0;

        printf("Eingabe: Operator Argument1 Argument2\n"
                        "Operatoren: 0 = Summe    1 = Produkt\n");
        scanf("%u %lf %lf",&op, &a, &b);

        if (op == 0)
                funktionsZeiger = &summe;
        else
                funktionsZeiger = &produkt;

        printf("Ergebnis: %f\n", (*funktionsZeiger)(a, b));
}
```

```
zweiDimFun vfptr[2] = { summe, produkt };
```

```
printf(“Ergebnis: %f\n“, vfptr[op](a, b));
```

[Zeiger] Zeiger angewandt: memcpy

```
void* memcpy(void* dest, const void* src, size_t n);
void* memmove(void* dest, const void* src, size_t n);
```

- Den Funktionen werden Zeiger übergeben, deren Typ *void* ist, um die Funktion für beliebige Typen (oder Arten von Speicherbelegung) zu ermöglichen (im Gegensatz zu strcpy)

```
#include <stdio.h>
#include <string.h>
#define LAENGE 10

int main()
{
        int v1[LAENGE];
        int v2[LAENGE] = { 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9};
        int i;
        for (i=0;i<LAENGE;i++)
                printf("%d ",v1[i]);
        memcpy(v1 , v2 ,  LAENGE*sizeof(int) );
        for (i=0;i<LAENGE;i++)
                printf("%d ",v1[i]);
        return 0;
}
```

[Zeiger] Beispiele

Aufgaben:

- Vergleichen Sie obige Beispiele

- Untersuchen Sie das Quicksort-Beispiel im Skript und machen Sie sich klar, wie es funktioniert