Zeiger (engl. Pointer)

Zeiger

Ein Zeiger (engl. Pointer) speichert eine Adresse, unter der ein Wert im Speicher des Computers gespeichert werden kann.

Eine Variable im Gegensatz speichert einen Wert.

Der Name eines Zeigers ist mit einer Adresse verbunden, ein Variablenname dagegen mit einem bestimmten Wert.

Zeiger werden z.B. für folgende Zwecke benutzt:

- Dynamische Speicherverwaltung
- Parameterübergabe in Funktionen
- Zeichenkettenverarbeitung

Definition von Zeigervariablen

- Bei der Definition eines Zeigers wird ein Stern ("*") vor den Bezeichner geschrieben.
- Werden die Bezeichner mit Kommas getrennt in einer Definitionsliste angegeben, so muss der Stern vor jeden einzelnen Bezeichner geschrieben werden, der als Zeiger definiert werden soll.

```
long *LZ, LZ2;  // LZ ist ein Zeiger, LZ2 eine Variable
float summe, * psumme;    // summe ist eine Variable,
                          // psumme ein Zeiger
```

- Für eine gute Übersichtlichkeit, besser Variablen und Zeiger in jeweils einer eigenen Zeile definieren!

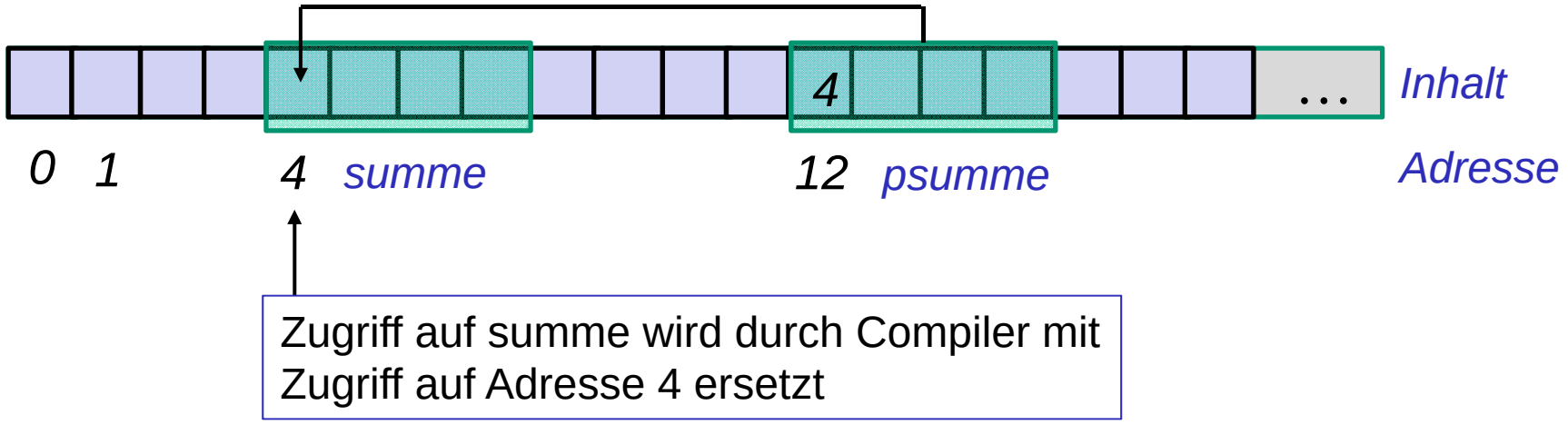
```
float summe;
float* psumme;
```

Veranschaulichung einer Zeigervariablen

Der Speicher eines Rechners wird über Adressen verwaltet, die die Speicherstellen (Bytes) linear durchnummerieren. Die Nummer des angesprochenen Bytes ist die Adresse.

Hier ein Beispiel (stark vereinfacht) :

```
float summe;
…
float* psumme = &summe; // Zeiger mit Adresse von summe
```



Der &-Operator

Zeiger - Adressoperator

- Um die Adresse einer Variablen zu erhalten, kann ein spezieller Operator verwendet werden.
- Dieser Operator wird als Adressoperator bezeichnet. Er wird mit dem Zeichen "&" (engl. Ampersand) symbolisiert.

Zum Beispiel:

```
int Io = 1024;
int* IZ1 = &Io; // IZ erhält die Adresse von Io
                // und nicht den Wert 1024
```

Zeiger

Ein Zeiger kann auch mit einem anderen Zeiger desselben Typs initialisiert werden.

```
// jetzt enthält auch IZ2 die Adresse von Io
int* IZ2 = IZ1;
```

Mehrfachzeiger: Von einer Zeigervariable kann ebenfalls die Adresse gebildet werden.

```
int ** IZ3 = &IZ1; // Zeiger auf Zeiger
```

Beispiele zur Benutzung von Zeigern

```
void swap(float* x1, float* x2) { // zwei Werte vertauschen
        float temp = *x1; *x1 = *x2; *x2 = temp;
}

void swap2(float** x1, float** x2) { // zwei Zeiger vertauschen
        float* temp = *x1; *x1 = *x2; *x2 = temp;
}
int main(void)
{       float a, b, *pa, *pb;
        a = 1.0, b = 2.0, pa = &a, pb = &b;
        swap(&a, &b); /* Tauscht die Werte a und b wirklich* /
        …
        swap2(&pa, &pb); /*  tauscht nur die Zeiger */
        …
}
```

Der * - Operator

„*****“ ist der Inhaltsoperator. Damit kann ein Zeiger dereferenziert werden,
d.h. auf den Wert der Variable (auf die gezeigt wird) zugegriffen werden.

```
int i = 1234;
int* pi; // pi ist also vom Typ „Zeiger auf int“
pi = &i; // pi zeigt jetzt auf i
*pi = 5678; // i hat jetzt den Wert 5678,
// der Zugriff auf i erfolgt hier durch Dereferenzieren
// des Zeigers pi.
```

Dereferenzierung:

```
Typ* ptr; // hier wird eine Zeigervariable definiert
*ptr = ...; ... = *ptr; // hier wird ptr dereferenziert
                        // und damit auf den Inhalt zugegriffen
```

Zeiger und Typen

Jeder Zeiger ist mit einer Variablen eines bestimmten Typs verbunden

Der Datentyp gibt den Typ des Datenobjekts an, das mit Hilfe dieses Zeigers adressiert wird.

Ein Zeiger des Typs *int* zeigt zum Beispiel auf ein Objekt des Typs *int*.

```
int intvar;
int* intZeiger = &intvar;
```

Um auf ein Objekt des Typs *double* zu zeigen, muss der Zeiger mit dem Datentyp *double* definiert werden.

```
double dvar;
double* dZeiger = &dvar;
```

Zeiger - Speicherbedarf von Zeigern

Der Speicherplatz, der für einen Zeiger reserviert wird, muss eine Speicheradresse aufnehmen können.

Das bedeutet, dass ein Zeiger des Typs *int* und ein Zeiger auf einen Datentyp *double* normalerweise gleich groß sind.

Der Typ, der einem Zeiger zugeordnet ist, gibt den Inhalt und damit auch die Größe des adressierten Speicherbereichs an.

int pint ; // pint belegt 4 Byte*

double pdouble; // pdouble belegt auch 4 Byte*

Der Speicherplatz der für Zeiger verwendte wird ist implementierungsabhängig und typischerweise 32 oder 64 Bit (d.h. 4 oder 8 Byte lang).

Zugriff auf Felder durch Zeiger (1)

In C sind Variablen für Felder und Zeiger zuweisungskompatibel.

Beispiel:

```
int zahlenfeld[20];
int einzelzahl;
int *iptr;

iptr = zahlenfeld; // dem Zeiger kann der Name eines Feldes
                   // zugewiesen werden
einzelzahl = iptr[5]; // Ein Zeiger kann wie ein Feld benutzt werden
```

Ein Feld mit Elementen eines bestimmten Typs wird als Variable wie ein Zeiger auf ein Element des Typs behandelt.

Gleichzeitig kann ein Zeiger auch wie ein Feldbezeichner mit Indexklammern versehen werden.

Zugriff auf Felder durch Zeiger (2)

Hintergrund:

```
int feld[10];
int* ptr = feld; // entspricht: int* ptr = &feld[0]
```

Der Feldname entspricht damit der Adresse des ersten Feldelementes, d.h. dem mit dem Index 0.

Der Zusammenhang zwischen Feldern und Zeigern wird oft bei der Übergabe von Feldern in Funktionsparametern ausgenutzt:

```
void auswahlfunktion( int *feld, int index, int *wert )
{
  *wert = feld [index];
}
```

Addition und Subtraktion bei Zeigern

Bei der Addition (oder Subtraktion) eines ganzzahligen Wertes (z.B. 2), wird der Wert der entsprechenden Adresse um die Größe von z.B. zwei Objekten dieses Datentyps (hier int) erhöht (oder erniedrigt). Wenn z.B. ein char 1 Byte, ein int 4 Byte und ein double 8 Byte belegt, dann erhöht sich bei einer Addition um 2 zu einem Zeiger, der Wert der im Zeiger gespeicherten Adresse um 2, 8 bzw. 16 Byte.

Beispiele:

```
int i; // z.B. Adresse 0x0100
int j; // z.B. Adresse 0x0104
int k; // z.B. Adresse 0x0108
int feld[10]; // z.B. Adresse 0x010C, 0x0110, 0x0114 ...
int* ptr = & i; // ptr enthält Adresse von i, d.h 0x0100
ptr = ptr + 2; // ptr enthält nun 0x0108, d.h. Adresse von k
ptr++; // ptr enthält nun 0x010C, d.h. Adresse feld[0]
```

Operationen mit Zeigern (1)

Einem Zeiger kann man einen Zeiger des gleichen Typs zuweisen.

```
int* ptr1 = &i;
int* ptr2 = ptr1;
```

Ein Zeiger kann auch inkrementiert oder dekrementiert werden.
So bedeutet z.B. „+5“ hier eine Inkrementierung um 5 Schritte mit der Schrittweite „sizeof(int)“:

```
int* ptr3 = ptr2 + 5;
```

Eine Zuweisung eines anderen Typs ist nur über Typecast möglich:

```
char* pchar = (char*) ptr2; // typecast
```

Operationen mit Zeigern (2)

Es kann die Differenz zweier Zeiger gebildet werden:

```
int feld[10];
int* ptr1 = &feld[1];
int* ptr8 = &feld[8];
int anzahl = ptr8 – ptr1; // ergibt den Wert 7
// Besser:
ptrdiff_t anzahl = ptr8 – ptr1;
```

(ptrdiff_t ist in *cstddef* bzw. *stddef.h* vereinbart)

Einem Zeiger darf der Wert *0 (NULL)* zugewiesen werden. Dadurch wird der Zeiger als nicht initialisierter Zeiger gekennzeichnet:

```
int* ptr = 0; // oder: int* ptr = NULL;
```

Die Vergleichsoperationen == und != sind zulässig, um zwei Zeiger auf Gleichheit zu testen, d.h. ob sie auf die gleiche Adresse verweisen. Dieses hat nichts mit den Werten zu tun, die die Adressen enthalten.

NULL - Zeiger

Es gibt einen besonderen Zeigerwert, der als Konstante NULL definiert ist. Die interne Darstellung der NULL ist implementierungsabhängig, aber kompatibel zur ganzzahligen Null (0). NULL kann jeder Zeiger-variablen zugewiesen werden, und jede Zeigervariable kann auf NULL getestet werden;

```
p = NULL; /* Beide Anweisungen sind */
p = 0; /* korrekt und bedeutungsgleich! */
. . .
if (p==NULL) . . .  /* Alle drei */
if (!p) . . .          /* Abfragen sind korrekt */
if (p==0) . . .       /* und bedeutungsgleich! */
```

Diese Operationen können auch dazu verwendet werden, um zu ermitteln, ob ein Zeiger bereits initialisiert wurde:

```
int *ptr = 0, *ptr2;
. . .
if (ptr == 0) { … // initialisiere jetzt ptr }
```

Weitere Bedeutung von Zeigern

Zeiger zur Übergabe von Parametern an Funktionen

"Call by Reference" – Parameter sind Zeiger

```
void function(int x, int *y)
{
  *y = 2*x*x – 3*x +5;
   // der Wert auf den y zeigt wird verändert
}
```

Zeiger für dynamische Speicherstrukturen

```
int *p;
int anzahl_werte; // konkreter Wert ergibt sich zur Laufzeit
…
p = malloc(anzahl_werte*sizeof(int));
…
```