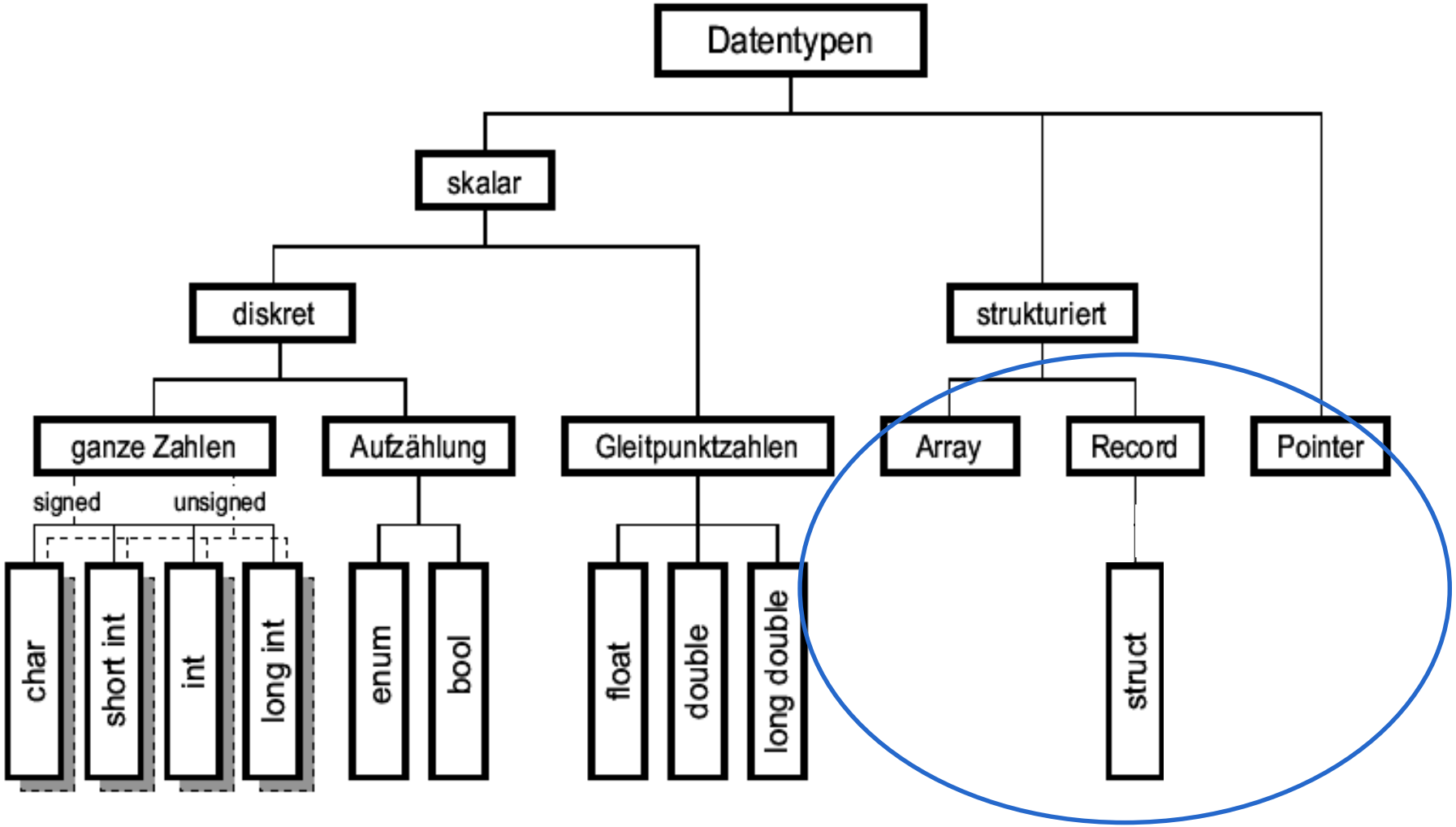
INF1
Arrays und Zeiger

- Arrays – Felder von Elementen gleichen Typs
- Verwenden von Adressen: Zeiger
- Spezielle Zeiger und komplexe Deklarationen

Datentypen Übersicht



Arrays – Felder von Elementen gleichen Typs

Beispiel

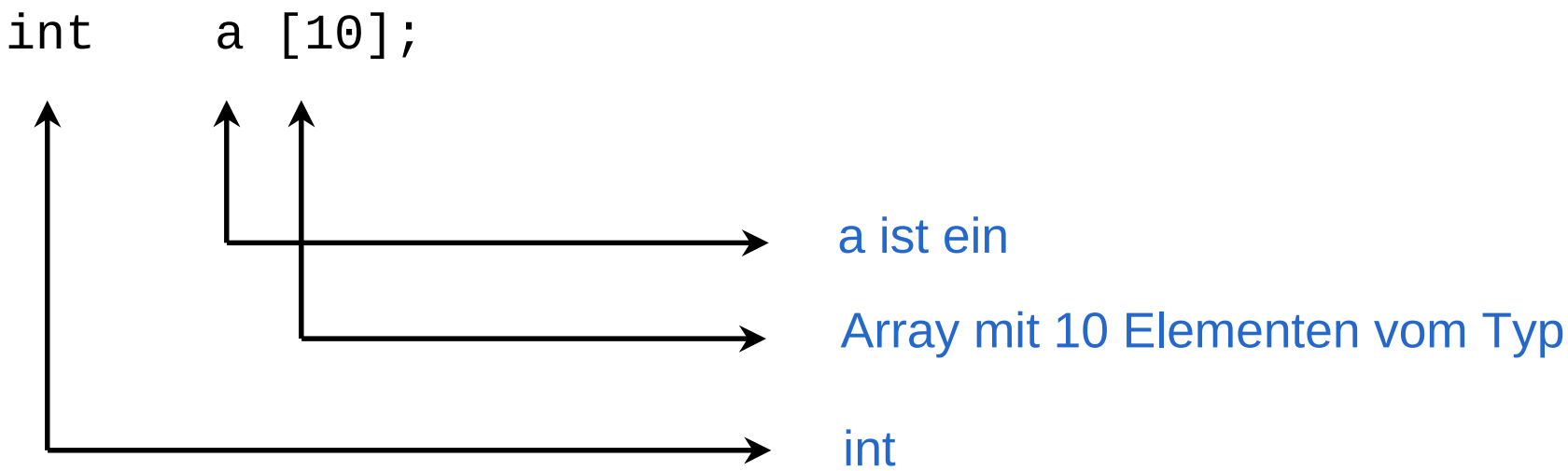
- Aufgabe:
  Es soll eine Zahlenreihe eingelesen werden

- Für solche Zwecke gibt es eine spezielle Datenstruktur:
  Das *Array*

- Ein Array (auch Datenfeld genannt) ist eine homogene Datenstruktur, die aus mehreren Elementen des gleichen Datentyps besteht

- Die Elemente werden durch den Variablennamen und einen Index bestimmt (der Index ist eine ganze Zahl)

Definition



- Es wird Platz für 10 int-Elemente reserviert
- Aber: Die Elemente erhalten keine Default-Werte!

Beispiele

```
double zahlen[300];    /* 300 double-Objekte
                          zahlen[0] .. zahlen[299] */

char text[20];         /* 20 Zeichen
                          text[0]   .. text[19] */
```

- Was fällt auf?

Hinweise

- Array-Grenzen beginnen immer bei 0

```
int werte[5];
```

- Das letzte Element dieses Arrays ist werte[4]

```
werte[3] = 5;
werte[2] = werte[4] + werte[0];

index = 1;
werte[index] = 233;
```

Hinweise

- Es findet keine Bereichsüberprüfung statt (weder zur Übersetzungs- noch zur Laufzeit)
- Mit anderen Worten: Es wird nicht geprüft, ob der Zugriff auf ein Arrayelement wirklich „innerhalb“ des Arrays liegt!


```
int a[5];

a[3] = 4;
a[8] = 222;     /* Keine Fehlermeldung zur Laufzeit!!! */
a[-3] = 36;     /* Keine Fehlermeldung zur Laufzeit!!! */
                /* Aber Folgefehler wahrscheinlich */
```

Initialisierung

- Arrays können bei der Definition auch gleich initialisiert werden

```
int a[5] = {4,7,12,77,2};   /* Alle 5 Elemente werden initialisiert */
int b[]  = {4,7,12,77,2};   /* Länge wird implizit auf 5 gesetzt */
int c[5] = {4,3,88,5};      /* OK, letztes Element erhält Wert 0 */
int d[9] = {0};             /* Alle Elemente 0 */
int d[5] = {4,3,88,5,3,6};  /* Kompilierfehler */
```

Zuweisung?

- Das Füllen des Arrays mit {...} funktioniert nur bei der Deklaration der Variablen
- Folgendes gibt einen Kompilierfehler:

```
int a[5];
a = {1, 3, 66, 34, 7};  /* Fehler */
```

- Die Grösse eines Arrays kann nach der Deklaration nicht mehr verändert werden

Grösse von Arrays

- Normalerweise: Konstante Grösse erforderlich
- Bei älteren Compiler Versionen muss die Konstanten mit #define festgelegt werden.

```
const int KONSTANTE= 5;

#define ALT_KONSTANTE 4


int a[3];                    /* kein Problem */
int b[ALT_KONSTANTE ];       /* kein Problem: nur Textersetzung */
int c[KONSTANTE];            /* geht ab C99 */
int foo (int nichtKonstant) {
    int c[KONSTANTE];     /* geht ab C99 */
    int d[KONSTANTE+ 4];  /* Compiler kann dies auswerten */
    int e[nichtKonstant]; /* geht ab C99 Array auf Stack */
}
```

Achtung kein ";"

Beispiele

```
const int MAXZ = 300;
double zahlen[MAXZ];   /* 300 double-Werte
                          zahlen[0] .. zahlen[299] */

char text[20];         /* 20 Buchstaben
                          text[0]   .. text[19] */


for (i = 0; i < MAXZ; i++) {
    zahlen[i] = i;     /* Zuweisung in einer Schleife */
}

text[4] = 'c';
```

Verwenden von Adressen: Zeiger

Zeiger

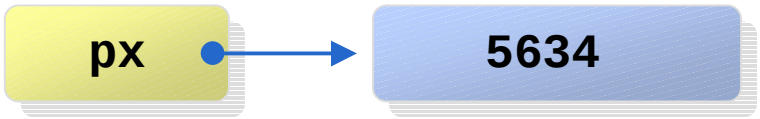
- Bisher: Zugriff erfolgte immer direkt auf die Variablen; ohne Berücksichtigung wo (unter welcher Adresse) diese Variable im Speicher abgelegt ist
- In C/C++ arbeitet man häufig mit den Adressen von Variablen: Dazu existieren in C/C++ die Zeiger (engl. Pointer)

> Ein Zeiger ist eine Variable, welche die Adresse einer andern Variablen enthält

- Wenn der Zeiger die Adresse einer anderen Variable enthält, sagt man, dass der Pointer auf diese Variable zeigt

Zeiger

- Ein Zeiger enthält somit die Adresse auf den eigentlichen Wert
- Greift man über den Zeiger auf die Variable zu, wird der Zeiger *dereferenziert*



Zeiger definieren

School of Engineering

- Beispiel

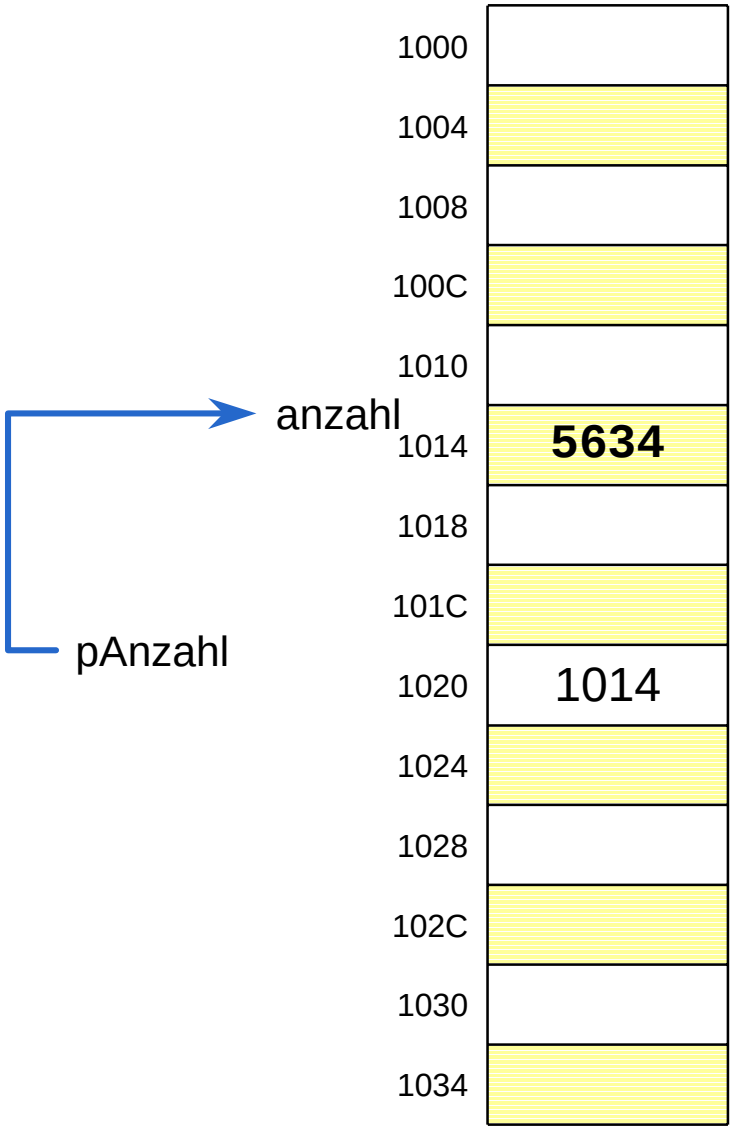
Konvention mit einem "p" beginnen

```
int      *px;      /* Zeiger auf int  */
char     *pc;      /* Zeiger auf char */
```

- Zeiger zeigt auf Daten von einem bestimmten Typ

Der Zeiger muss gesetzt werden

```
int anzahl;
int *pAnzahl;

/* Adresse von anzahl zuweisen */
pAnzahl = &anzahl;

/* Über Zeiger zugreifen: */
*pAnzahl = 77;
```



Zeiger und Adressoperationen

- *Adressoperator &*
  - liefert Adresse eines Objekts

&

- *Dereferenzierungsoperator **
  - greift auf das Objekt zu, auf das der Zeiger zeigt

*

- Bei scanf schon verwendet

```
scanf("%lf",&value);
```

Damit scanf weiss, wohin der Wert gespeichert werden soll, muss der Speicherort (=Adresse der Variablen) übergeben werden

Zeiger und Funktionen - pass by reference (später)

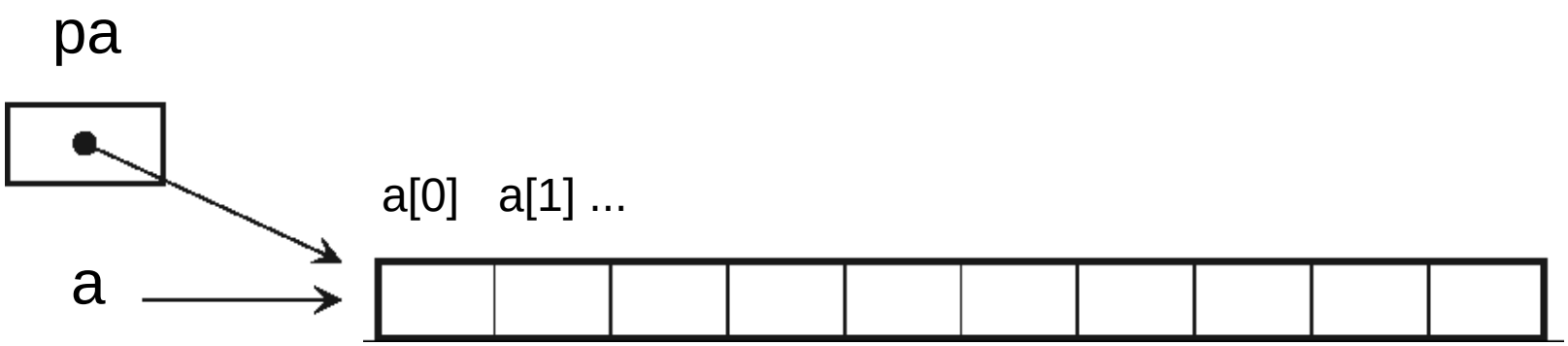
```
void foo(int* pi) {
... ...
}
int i;
foo (&i)

scanf(char* format, *p1,...)...
...
scanf("%d",&i)
```

- Wenn in Signatur von Methoden Pointer Variablen stehen
  - es wird nicht der Wert sondern die Adresse übergeben (Pass by Reference später)

- Beim Aufruf muss die Adresse einer Variablen übergeben werden

Zeigerarithmetik und Arrays

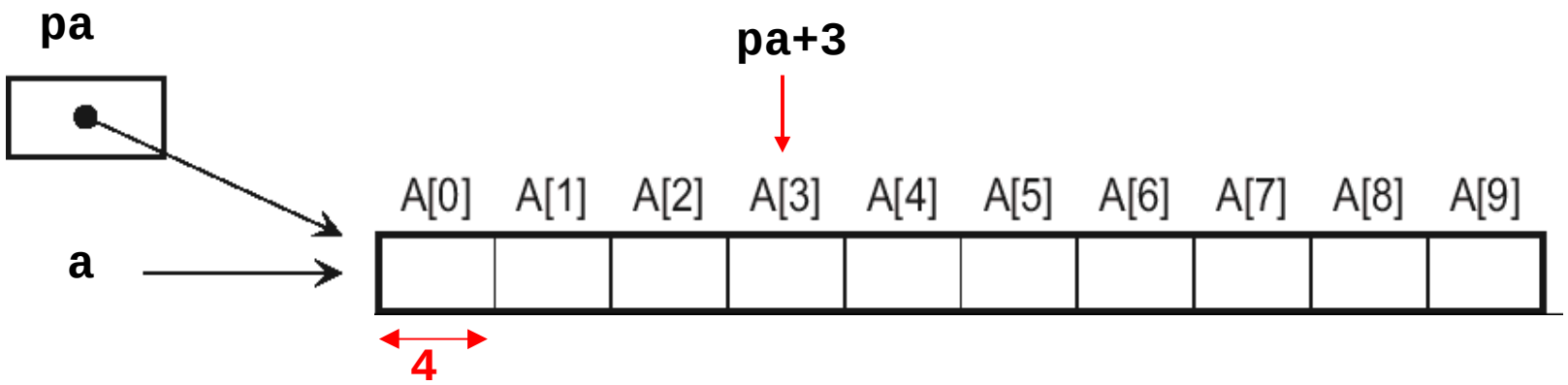
```
int    a[10];
int    *pa;
pa = &a[0];
```



Zeigerarithmetik und Arrays

- Zeigerarithmetik statt Arrayzugriff

```
int a[10];
int *pa = &a[0];
a[3] = 42;                  // a[3] den Wert 42 zuweisen

*(pa+3) = 42;               // a[3] den Wert 42 zuweisen
```



Zeigerarithmetik und Arrays

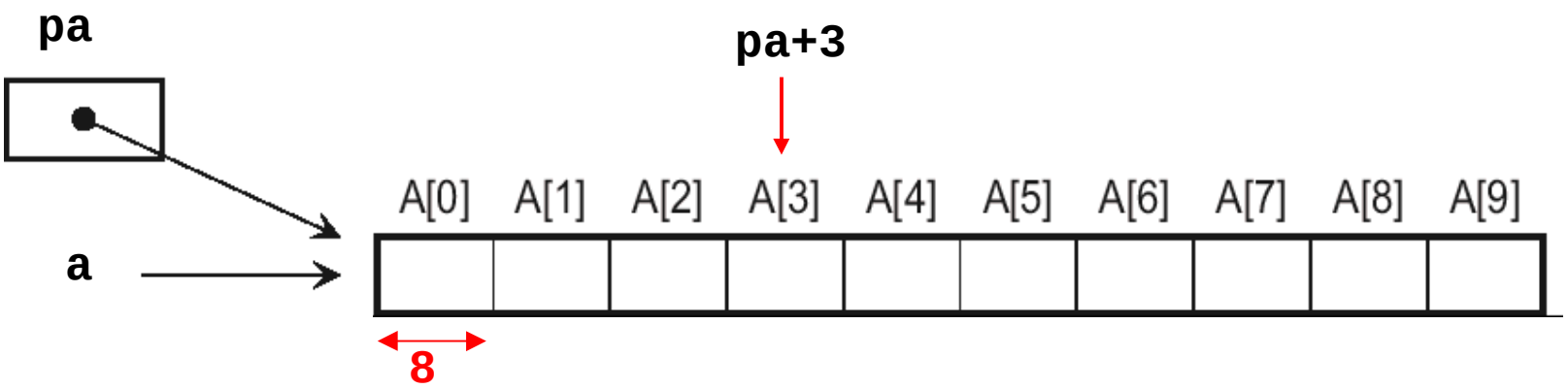
- Mit Zeigern kann gerechnet werden
- Bsp: pa + 4, pa++, pa--

```
int a[10];
int *pa = &a[0];
a[3]       = 42 ;
*(pa + 3)  = 42 ;        /* gleich wie a[3]=42; */

pa         = pa + 3;     /* pa ist variabel     */
*pa        = 42 ;        /* gleich wie a[3]=42; */
```

Zeigerarithmetik und Arrays

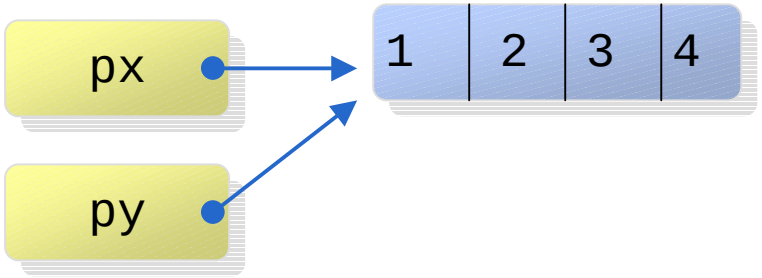
- Da zu jedem Zeiger ein Datentyp gehört, wird bei Operationen auf Zeiger die Grösse des Datentyps berücksichtigt

```
double a[10];
double *pa = &a[0];
a[3]       = 42.0 ;
*(pa + 3)  = 42.0 ;        /* gleich wie a[3]=42; */
pa         = pa + 3;       /* pa ist variabel     */
*pa        = 42.0 ;        /* gleich wie a[3]=42; */
```



Zuweisen von Zeigervariablen

```
int x[4] = { 1, 2, 3, 4 };
int *px = &x[0];
int *py;
py = px;
printf("%d",*px); // 1
*py = 5;
printf("%d",*px); // 5
```



- Wenn Zeigervariablen Werte zugewiesen (kopiert) werden, dann werden nur die Zeiger (aber nicht die Werte auf diese) kopiert.
- Man erhält lediglich zwei "Zugriffspfade" auf dieselben Daten
- Wenn Werte über py verändert werden sind sie auch über px verändert

Typischer Fehler

```
int *px;
*px = 25;
```

- In der zweiten Zeile wird irgendeine Stelle im Speicher mit 25 überschrieben!
- Zeigervariablen mit unkorrekten Werten führen - im besseren Fall- zu Laufzeitfehlern wie Bustrap-Error, Segment-Violation, Memory-Violation oder gar zum System-Crash
- Im schlechtesten Fall zeigt der Zeiger in einen gültigen Speicherbereich, das Programm läuft aber es wird irgendwo (!) Speicher überschrieben
- Unbestimmtes Verhalten (bei jedem Programmstart wieder anders), plötzlich veränderte Werte in andern Variablen, die NICHTS mit dem eigenen Programm zu tun haben.
  Sog. DANGLING POINTER - DER ABSOLUT SCHLIMMSTE FEHLER IN DER PROGRAMMIERUNG ÜBERHAUPT

NULL

- Wenn ein Zeiger auf nichts zeigen soll, wird ihm NULL zugewiesen

- Sich angewöhnen, jeden Zeiger bei der Deklaration mit NULL zu initialisieren.

```
int *pi = NULL;
*pi = 25;
```

- Das Programm stürzt kontrolliert ab

Array vs Pointerschreibweise

Beispiel: Kopieren einer Zeichenkette

```
char i;
for (i=0; i<10; i++) {
     b[i] = a[i];
}
```

In Zeiger-Notation:

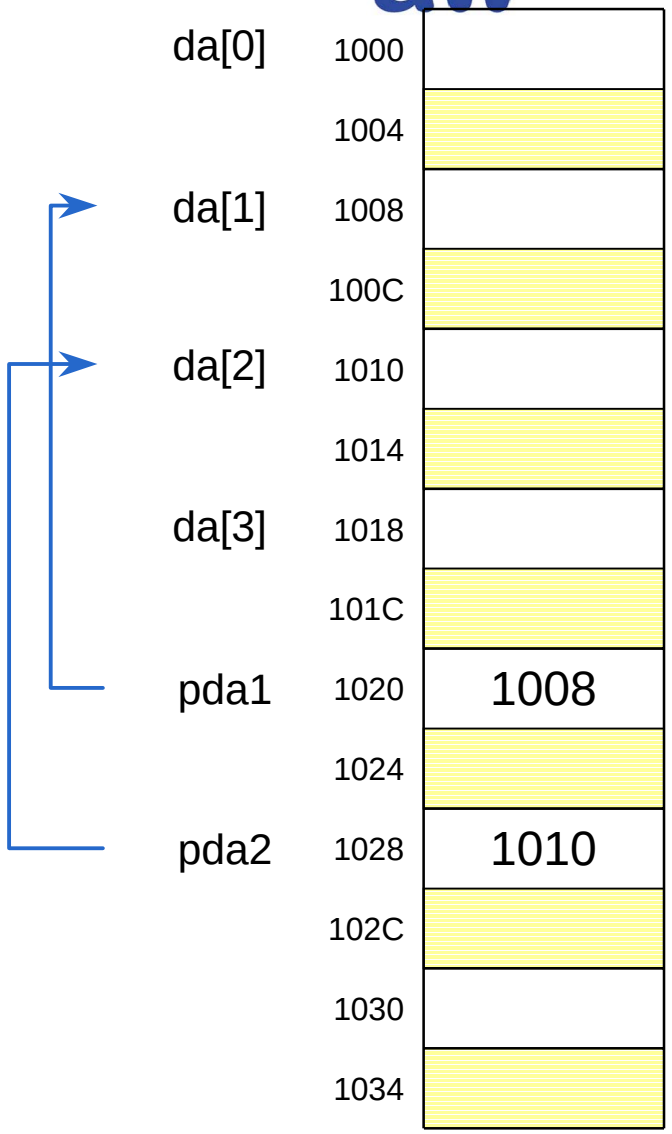
```
char *pa = &a[0];
char *pb = &b[0];
int i;
for (i = 0;i< 10;i++) {
     *pb = *pa;
      pa++; pb++;
}
```

```
for (;*pb++=*pa++;);
```

- Die rechten Versionen sind schlechter lesbar!

Zeigerarithmetik und Arrays

```
double    da[4];
double    *pda1;
double    *pda2;
int       diff;

pda1 = &da[1];
pda2 = &da[2];

diff = pda2 - pda1;
```

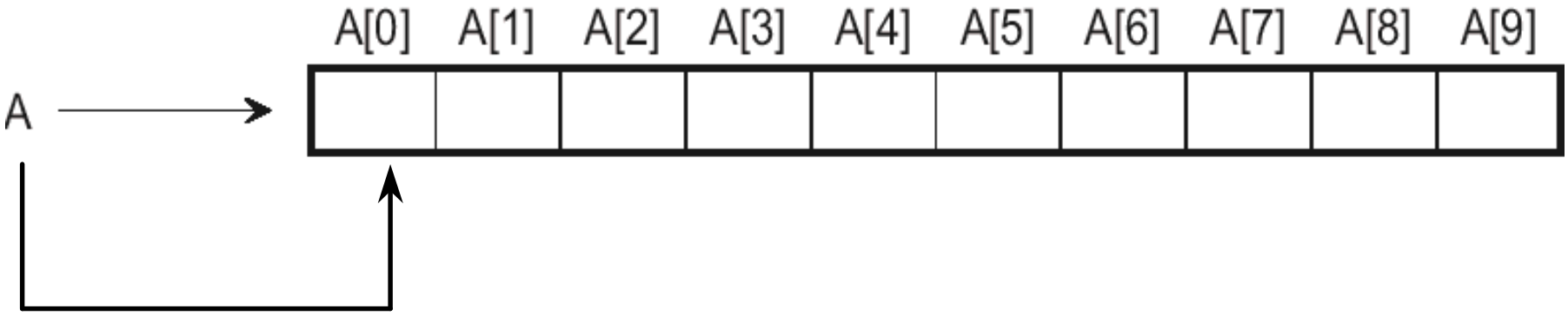


Wenn zwei Zeiger auf ein Array voneinander subtrahiert werden, erhält man die Anzahl der *Elemente* dazwischen

Zeigerarithmetik und Arrays

- Zeiger und Arrays sind in C eng verbunden
- Der Arrayname kann als **konstanter Zeiger** auf das erste Array-Element aufgefasst werden

```
int a[10];
pa = a;           /* wie: pa = &a[0]; */
*(a + 3) = 42 ;   /* gleich wie a[3]=42; vorsicht */
a++ ;             /* nicht erlaubt !! */
```



Fallstricke

- Mehrere Zeiger Variablen definieren

```
int    *pi, pa;   /* nur pi ist ein Zeiger, pa NICHT *
int  *  pi, pa;   /* dito */
int*    pi, pa;   /* dito */
```

Spezielle Zeiger und komplexe Deklarationen

void-Pointer

- Der void-Zeiger *void** ist zu jedem Datentyp kompatibel
- Daher auch *untypisierter* oder *generischer Zeiger* genannt
- Ein void-Zeiger lässt sich in jeden anderen Zeiger und zurück wandeln
- Nur in begründeten Spezialfällen diesen Pointer verwenden.

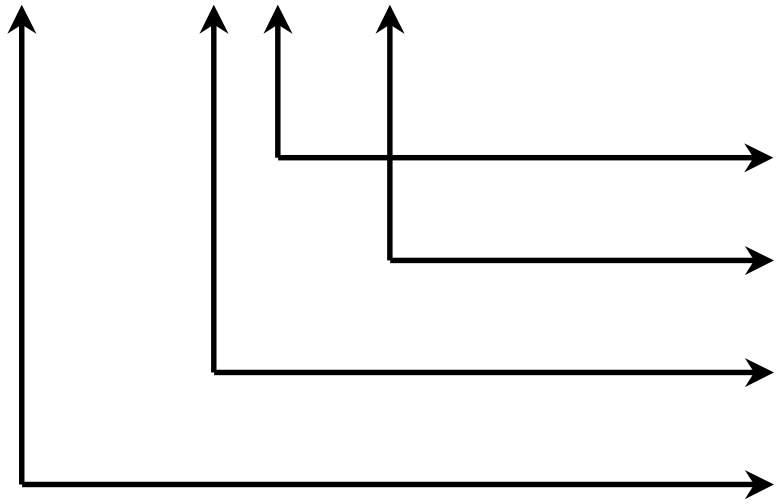
```
int a;
int* pa = &a, *pb;
void* pv;
pv = pa;
pb = (int*)pv;
```

void-Pointer Arithmetik Beispiel

```
int *pint;
void *pvoid;
pvoid=pint;             /* beide zeigen jetzt auf das
                           gleiche Element */
pint+=1;                /* zeigt nun auf das naechste Element */
pvoid+=sizeof(int);     /* zeigt jetzt auch auf das naechste
                           int Element */
```

Komplexe Deklarationen

```
char   * args[];
```



args ein

Array mit Elementen vom Typ

Zeiger auf

char

oder auch char**

```
int main (int argc, char* args[])
```

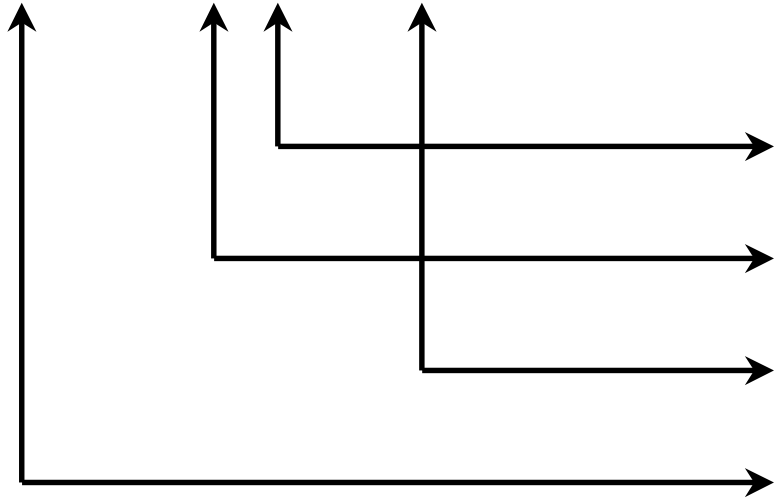
Grösse des Arrays

strings

Also: Zeiger auf char array

… komplexe Deklarationen

```
double  (* c) [10];
```



c ist ein

Zeiger auf ein

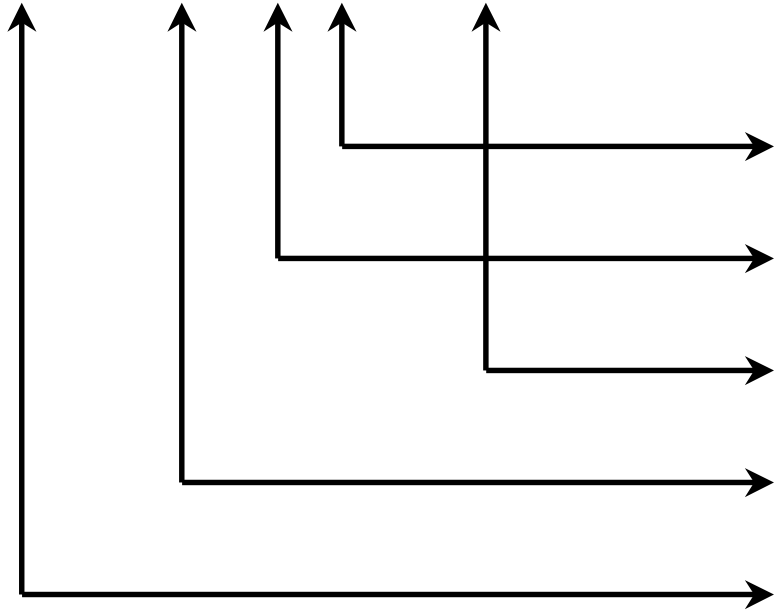
Array mit 10 Elementen vom Typ

double

Also: Zeiger auf 10 double's

… komplexe Deklarationen

```
double  * (* d) [10];
```



d ist ein

Zeiger auf ein

Array mit 10 Elementen vom Typ

Zeiger auf

double

Also: Zeiger auf 10 Zeiger auf double

Fazit Zeiger

- In einer „normalen“ Variablen steht immer der Wert, den man in dieser Variablen speichern will
- Ein Zeiger, *int * pa*, ist auch eine Variable
- Die Grösse eines Zeigers ist architekturabhängig (häufig 4 oder 8 Bytes)
- Dort steht aber nicht „irgendein“ Zahlenwert, sondern die Speicheradresse einer anderen Variablen

Noch Fragen