§6 Zeigerarithmetik - Allgemeines

Leitideen: *Der Zeigertyp ermöglicht eine Zeigerarithmetik, bei der die Addition von 1 die Adresse der nächsten Komponente eines C-Vektors angibt.*

Bei C-Vektoren wird nur die Startadresse mit Typinformation übergeben, das motiviert die in vielen Fällen automatische Typumwandlung C-Vektor in Zeiger.

Die Ausdehnung der Komponentenzugriffsschreibweise auf Zeiger mittels $p[i] := *(p + i)$ *ermöglicht es, zusammenhängende Speicherbereiche wie C-Vektoren zu behandeln. Das ist nützlich bei der Übergabe von C-Vektoren und der dynamischen Speicherallokation.*

Die Übergabe mehrdimensionaler C-Felder erfordert weitere Betrachtungen, in der Regel ist das Anlegen zusätzlicher C-Vektoren erforderlich.

§6 Zeigerarithmetik - Themenübersicht

- Motivation
- Inkrement/Dekrementoperatoren
- Unterschiede C-Vektor - Zeiger
- [$*$] Beispiele zur Umwandlung C-Vektor$\rightarrow$Zeiger
- [$*$] Zeigerarithmetik bei C-Matrizen
- Übergabe von C-Vektoren in Funktionen
- [$*$] Übergabe von C-Matrizen in Funktionen I,II,III
- Kommandoargumente
- Dynamische Speicherreservierung I,II,III

Zeigerarithmetik - Motivation

- Zeiger *p* ist Adresse mit Typ $\Rightarrow$ Adresse eines direkt folgenden Objekts bestimmbar (Bez.: $p+1$)
  Bsp.: `int *ip; // ip: 100` $\Rightarrow$ `ip+1: 104`
  (falls int 4 Byte)
- Adressarithmetik besonders wichtig bei C-Vektoren:
  &a[i] := &a[0]+i $\equiv$&(a[0])+i (aufeinanderfolgende Speicherung der Komponenten)
  a[i] $\equiv *$(&a[0]+i) (nur Adresse von Komp. 0 notwendig!)
- Automatische Umwandlung: C-Vektor $\rightarrow$ Zeiger *(oft!)*
  Zeigerwert: Adresse der Komponente 0, d.h. a $\rightarrow$ &a[0]
  [*Wichtigste Ausnahme:* C-Vektor steht *links* in Initialisierung oder Zuweisung]
- Also: a[i] $\equiv *$(&a[0]+i) $\equiv *$(a+i)
- Allgemein: p[i] := $*$(p+i) . (Sogar: i[p]:=$*$(i+p)$\equiv *$(p+i)$\equiv$p[i])

Zeigerarithmetik - Inkrement/Dekrementoperatoren

Inkrement- und Dekrementoperatoren sind auch auf Zeiger anwendbar.

Beispiel: Vorbesetzen eines Felds mit $a_i = i$

```
double a[N],*p;
p=a;
for (i=0; i<N; i++,p++) *p = i;
```

Auch zulässig:

```
for (i=0; i<N; i++) *p++ = i;
```

Funktioniert, weil `*p++=i` $\equiv$ `*(p++)=i` $\equiv$ `*p=i; p=p+1`

Unzulässig, weil &a[0] Konstante:

```
for (i=0; i<N; i++,a++) *a = i;
for (i=0; i<N; i++) *a++ = i;
```

Unterschiede C-Vektor - Zeiger

Bsp.: `double a[10],*p;`

- Definition von a reserviert Speicherplatz für 10 double-Zahlen (hier: 640 Bit)

  Definition von p reserviert *keinen(!)* Speicherplatz für eine double-Zahl, sondern lediglich Speicherplatz für eine Adresse (hier: 32 Bit bzw. 64 Bit)

- Umwandl. a$\rightarrow$&a[0] liefert defacto eine Zeiger*konstante*, keine Zeiger*variable*. Zuweisungen an a bzw. &a[0] sind daher unzulässig. (Sonst Informationsverlust über die Speicherlokalisation von a!).

  p sei eine Zeiger*variable*.

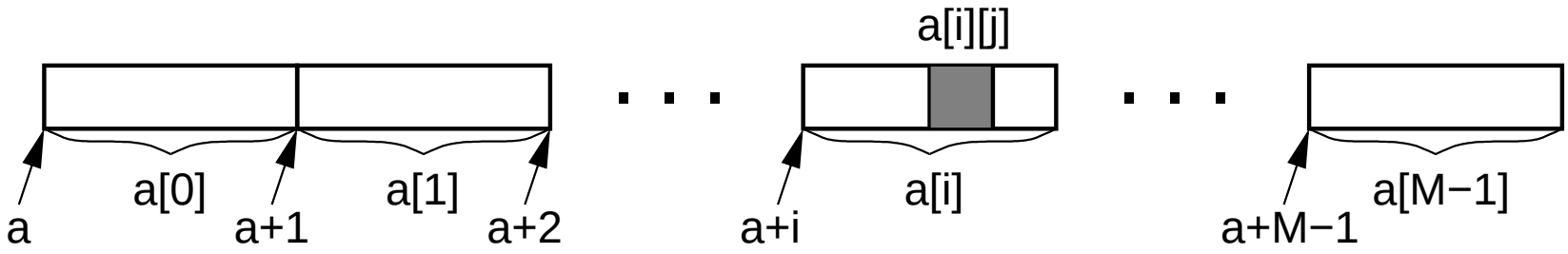
```
a = p; // Unzulaessig: C-Vektor steht links
       //               vom Gleichheitszeichen
p = a; // Zulaessig:   a -> &a[0], p: &a[0]
       // Bem.:  Es gilt  p[i]==a[i] (i=0..9)
```

[∗] Beispiele zur Umwandlung C-Vektor→Zeiger

1. ```
   double a[N],b[N],*c;
   a = b;    // Unzulaessig: a C-Vektor links von =
   a[i] = b[i]; // Zul. und aeq. zu *(a+i)=*(b+i)
   c = b;    // Zulaessig:  b -> &b[0]
   ```
2. ```
   double b[N],*c;
   c = &b    // Unzulaessig:
             // Li: Zeiger auf double
             // Re: Zeiger auf C-Vektor von double
   ```
3. ```
   double b[N],(*c)[N];
   c = &b;   // Korrekt:
             // Li: Zeiger auf C-Vektor von double
             // Re: Zeiger auf C-Vektor von double
   ```
4. ```
   double a[M][N],b[M][N];
   a = b;    // Unzulaessig: a C-Vektor links von =
   a[i] = b[i]; // Unzul.: a[i] C-Vek. links von =
   a[i][j] = b[i][j] // Zulaessig und aequiv. zu
                     // *(*(a+i)+j) = *(*(b+i)+j)
   ```

[∗] Zeigerarithmetik bei C-Matrizen

```
double a[M][N]
```



1. a+i $\equiv$ &a[0]+i $\equiv$ &a[i]
   wegen a+i $\equiv$&(∗(a+i)) $\equiv$ &a[i]
   Datentyp von a[0] bzw. a[i]: N-Vektor von double
2. a[i]+j $\equiv$ &a[i][0]+j $\equiv$ &a[i][j]
   wegen a[i]+j $\equiv$ &(∗(a[i]+j)) $\equiv$ &a[i][j]
   Datentyp von a[0][0] bzw. a[i][0] bzw. a[i][j]: double

Caveat

- Zwar sind die Adresswerte &a, &a[0] und &a[0][0] gleich , nicht aber die Datentypen!
  Daher i.allg.: &a+1 $\neq$ &a[0]+1 $\neq$ &a[0][0]+1

Übergabe von C-Vektoren in Funktionen

Prinzip: C-Vektoren werden nicht kopiert, statt dessen wird Adresse des C-Vektors (d.h. die Adresse der Komponente 0) übergeben.

Problem: Die Länge des C-Vektors geht verloren.
Auch mit `sizeof` *nicht* abfragbar!

Abhilfe: Länge ggf. zusätzlich als Argument übergeben.

- Eingesetzte C-Vektoren werden in Zeiger umgewandelt.
  (Spezialfall der Umwandlung C-Vektor $\rightarrow$ Zeiger)
  Bsp.: `double a[N],b[N]; int n;`
  `skalar(a,b,n)` $\rightarrow$ `skalar(&a[0],&b[0],n)`
- Formale C-Vektorparam. werden in Zeiger umgewandelt.
  C-Vektordimensionen können weggelassen werden!
  Bsp.: `double skalar(double a[N],double b[N],int n)`
  $\equiv$ `double skalar(double a[],double b[],int n)`
  $\equiv$ `double skalar(double *a,double *b,int n)`

[∗] Übergabe von C-Matrizen in Funktionen

Prinzip: Umwandlung C-Vektor $\rightarrow$ Zeiger findet nur einmal statt!

1. Methode

```
    int lingl(double a[M][N],...,int m,int n)
≡   int lingl(double a[][N] ,...,int m,int n)
≡   int lingl(double (*a)[N],...,int m,int n)
```

Denn: `double a[M][N]` M-Vektor von N-Vektor von double

$\rightarrow$ `double (*a)[N]` Zeiger auf N-Vektor von double

- Grundsätzlich möglich, aber unpraktisch, weil bei getrenntem Übersetzen von Haupt- und Unterprogramm die *Konstante N* übereinstimmen müsste.
- Wieder interessant in C 99, weil dort in beschränktem Umfang variable Längen für Vektoren zulässig sind. Bisher allerdings *nicht* in C++ übernommen!

[∗] Übergabe von C-Matrizen in Funktionen II

2. Methode

```
    int lingl(double *(ap[M]),...,int m,int n)
≡   int lingl(double *(ap[]) ,...,int m,int n)
≡   int lingl(double **ap    ,...,int m,int n)
```

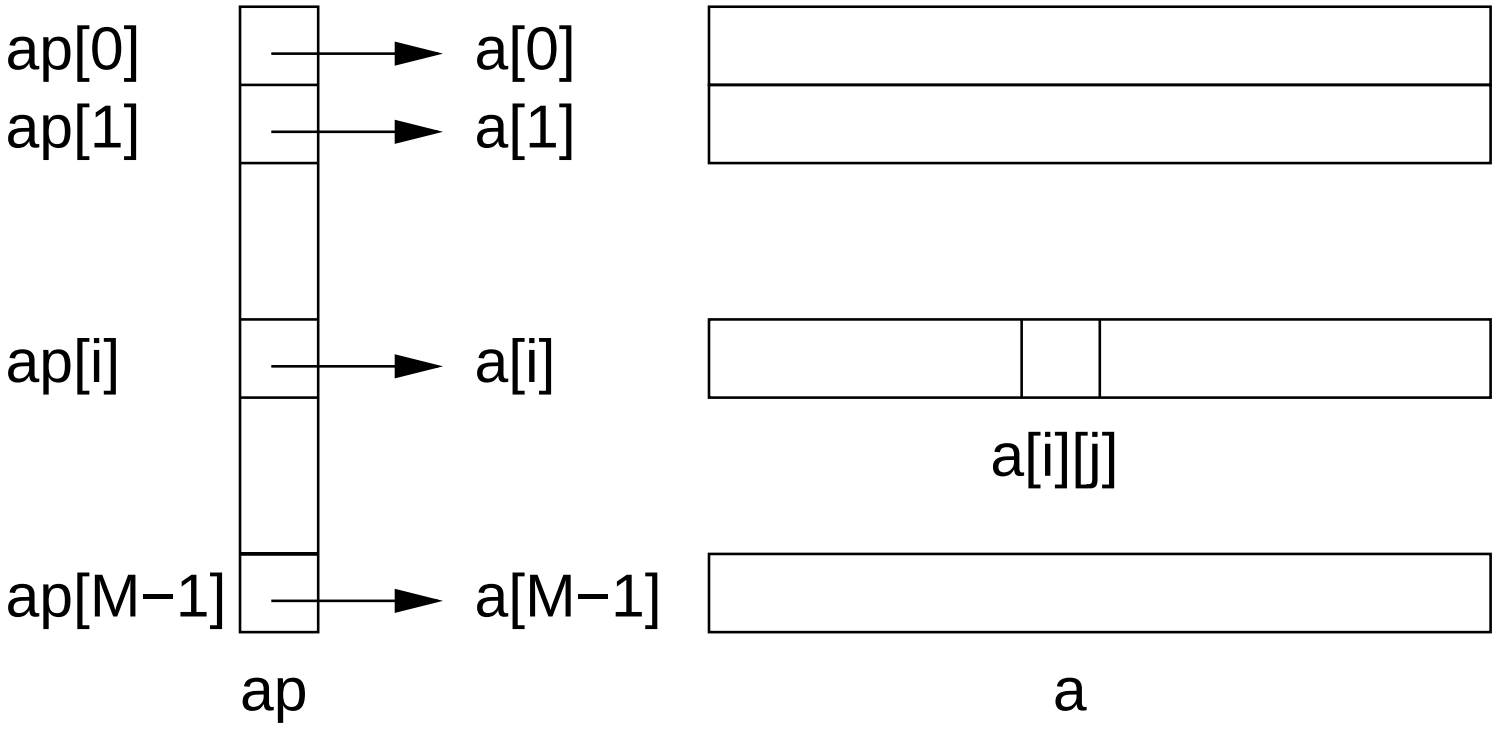
Denn: `double *ap[M]` M-Vektor von Zeigern auf double

$\rightarrow$ `double **ap` Zeiger auf Zeiger von double

- Zusätzlich im Hauptprogramm Hilfsvektor erforderlich:
  `double a[M][N], *ap[M]`
  `for (i=0; i<m; ++i) ap[i]=a[i];`
- Funktioniert, weil
  `ap[i][j]≡*(ap[i]+j)≡*(a[i]+j)≡a[i][j].`

[*] Übergabe von Matrizen in Funktionen III

```
double a[M][N], *ap[M];
for(i=0; i<M; ++i) ap[i]=a[i];
```



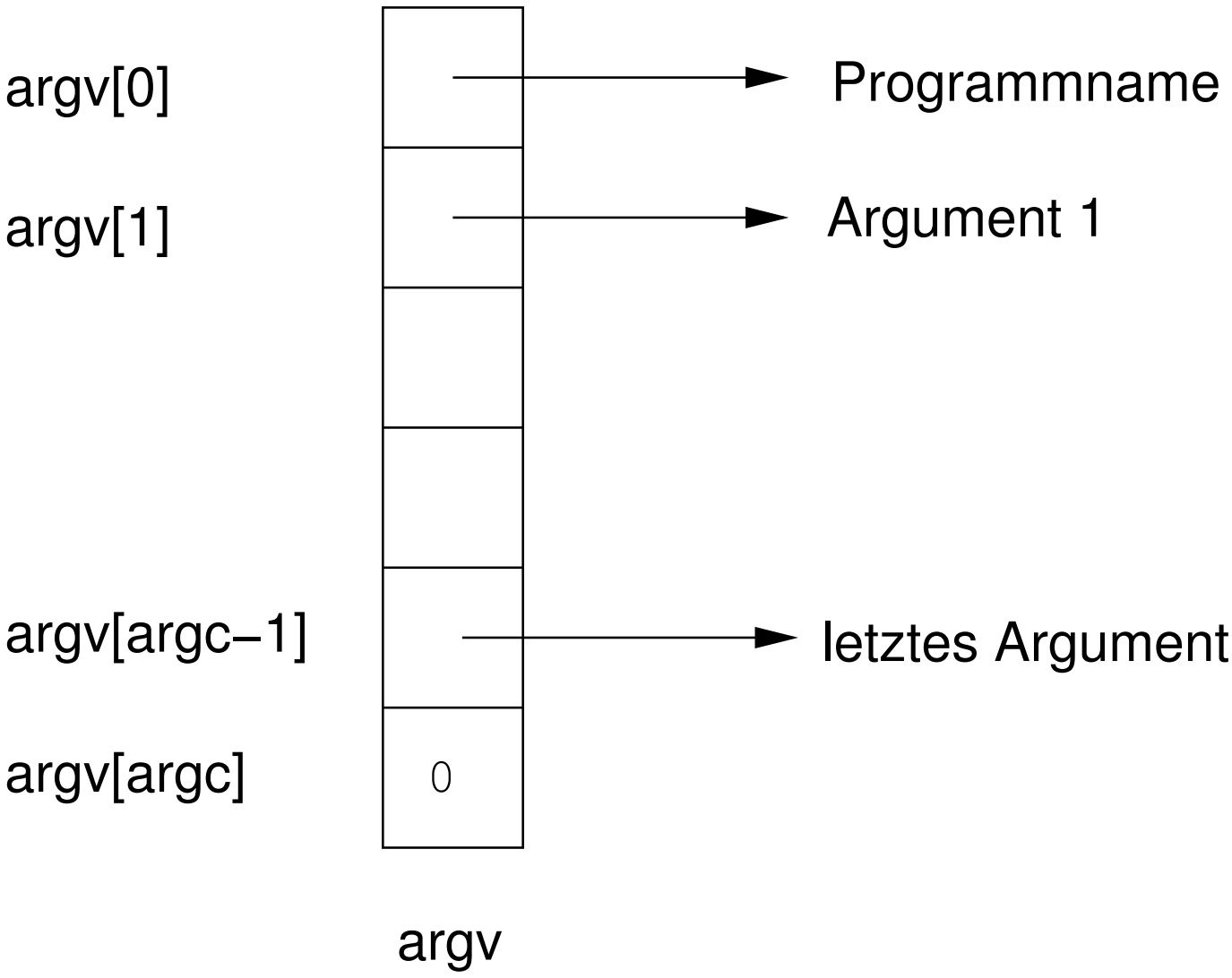
Bem.:

`a`	M-Vektor von N-Vektor von double
`a[i]`	N-Vektor von double
`ap`	M-Vektor von Zeigern auf double
`ap[i]`	Zeiger auf double

Gilt `ap[i][j]`$=$`a[i][j]` ?

Ja, denn: `ap[i][j]`$\equiv$`*(ap[i]+j)`$\equiv$`*(a[i]+j)`$\equiv$`a[i][j]`.

Kommandoargumente



Dynamische Speicherreservierung I

Operatoren new und delete

- `new` liefert Zeiger vom passenden Typ
- Syntax wie Variablenvereinbarung, aber:
  `new` davorschreiben, Variablenname weglassen
- Lebensdauer bis zum korrespondierenden `delete`,
  unabhängig von Blockgrenzen
- Zwei Ausprägungen bei delete:
  `delete` Einsatz für skalare Objekte
  `delete[]` Einsatz für C-Vektoren
- Operand von `new`: Typ (Variablenvereinb. ohne Name)
  Operand von `delete` bzw. `delete[]`: Zeiger
- Behandlung von Allokationsproblemen mit Hilfe von
  Exceptions *(später!)*
- STL-Behälterklassen vorrangig vor `new` und `delete`
  verwenden

Dynamische Speicherreservierung II

Bsp.: Dynamisch allozierte Integervariable

```
int *ip;
ip  = new int;
*ip = 17;  // Bsp. für Wertzuweisung

delete ip; // Freigabe des Speicherplatzes
```

Bsp.: Dynamisch allozierter double-Vektor

```
double *x;
x = new double[n];
x[n-1] = 3.14; // Bsp. für Wertzuweisung

delete[] x;    // Freigabe des Speicherplatzes
               // delete[] statt delete !!
```

[∗] Dynamische Speicherreservierung III

Bsp.: Dynamisch allozierte $m \times n$-Matrix

```
a = new double*[m];

for (int i=0; i<m; ++i)
  a[i] = new double[n];

// Freigabe der Matrix siehe Inf.bl.13,S.2
```