Vorlesung Informatik 1 (Wintersemester 2020/2021)

Kapitel 9: Adressen und Zeiger

Martin Frieb
Johannes Metzger

Universität Augsburg
Fakultät für Angewandte Informatik

17. Dezember 2020

Was ist ein Zeiger?

Definition 9.1 (Zeiger (Pointer))

Ein **Zeiger** ist eine **adresswertige Variable**, also eine Variable, deren Wert die Speicheradresse einer anderen Variable ist:

- Hat ein Zeiger `px` als Wert die Adresse einer Variable `x`, sagt man:
  - **`px` zeigt auf / referenziert `x`**
  - **`x` ist die Bezugsvariable von `px`**
- Zeiger sind **getypt**, d.h man muss angeben, welchen Datentyp die Bezugsvariable hat.

Speicherbedarf

Ein Zeiger speichert eine Adresse. Der Speicherbedarf dafür

- ist implementierungsabhängig,
- kann wie üblich mit `sizeof` abgefragt werden,
- und ist **unabhängig** vom Typ der Bezugsvariable

Deklaration von Zeigern

Deklaration

Ein Zeiger `px` auf eine Variable vom Typ `T` wird wie folgt vereinbart:

```
T *px;
```

- Der Typ von `px` ist `T*`
- Der Wert von `px` darf nur die Speicheradresse einer Variable vom Typ `T` sein
- `T` kann selbst wieder ein Zeigertyp sein: Dann ist `px` ein **mehrfacher Zeiger**, sonst ein **einfacher Zeiger**
- `T` ist der **Bezugstyp** von `px`. Man sagt: **`px` ist ein Zeiger auf `T`.**
- Vereinbarung mehrerer Zeiger in einer Anweisung:
  ```
  T *px, **ppx;
  ```

Beispiel 9.2

```
int *px, **ppx;
```:

- `px` ist ein Zeiger auf `int` (**Einfachzeiger**)
- `ppx` ist ein Zeiger auf `int*` (**Doppelzeiger**, wird später genauer behandelt)

Adressen

Mit dem Adressoperator `&` kann man auf die Speicheradresse einer Variable zugreifen

Adresswertige Ausdrücke

- Ist `x` eine Variable vom Datentyp `T` , so ist
  `&x`
  ein **adresswertiger Ausdruck** vom Typ `T*`. Sein Wert ist die Adresse der ersten Speicherzelle des Speicherbereichs von `x`
- Den Operator `&` nennt man **Adressoperator**.

Der Wert `NULL`

`NULL` ist eine adresswertige symbolische Konstante:

- Wert für **zeigt nirgendwohin** / **keine Adresse gespeichert**
- Bitmuster ist implementierungsabhängig
- Die Typumwandlung eines ganzzahligen Ausdrucks mit Wert 0 in einen Zeigertyp ergibt den Wert `NULL`

Adressen

Adressen können mit der Umwandlungsangabe `%p` als positive ganze Zahlen ausgegeben werden (Ausgabeformat ist implementierungsabhängig):

```
int main() {
    double x;
    int v[5];
    int *p = v;
    printf("Adresse von x: %p\n", &x);
    printf("Adresse von v: %p\n", v);
    printf("Adresse von v: %p\n", p);
    printf("Adresse von p: %p\n", &p);
    printf("Adresse von v[4]: %p\n", &v[4]);
    return 0;
}
```

- **Beachte**: Feldnamen und Zeiger sind adresswertig
- **Beachte**: Wert eines Zeigers $\neq$ Adresse eines Zeigers

Lokale und globale Zeiger

Lokale Zeiger

Ein **lokaler Zeiger** ist in einem Gültigkeitsbereich deklariert.

- Vor der ersten Wertzuweisung sagt man, der Zeiger **zeigt irgendwohin** (zufälliger Wert).
- Wird im Stack abgelegt (wie alle lokalen Variablen)

Globale Zeiger

Ein **globaler Zeiger** ist außerhalb aller Funktionen deklariert.

- Vor der ersten Wertzuweisung hat er als Wert die symbolische Konstante **`NULL`** - man sagt, der Zeiger **zeigt nirgendwohin**.
- Wird im Datenteil abgelegt (wie alle globalen Variablen)

Wertzuweisung

Definition 9.3 (Wertzuweisung)

Ist `p` ein Zeiger auf `T` und `e` ein adresswertiger Ausdruck vom Typ `T*` oder der Ausdruck `NULL`, so ist

```
p = e;
```

eine Wertzuweisung an `p`
(Man sagt: `p` wird nach `e` bzw. `NULL` **umgebogen**)

Beispiele 9.4 (Strenge Typ-Prüfung)

```
int x, *px, **ppx;
px = &x; /*px zeigt auf x*/
ppx = &px; /*ppx zeigt auf px*/
ppx = &x; /*Compilerfehler, da ppx kein Zeiger auf int
*/
double y;
px = &y; /*Compilerfehler, da px kein Zeiger auf double
*/
```

Dereferenzierung

Mit dem **Dereferenzierungsoperator** kann man auf den an einer Adresse gespeicherten Wert zugreifen.

Dereferenzierung

- Sei `e` ein adresswertiger Ausdruck vom Typ `T*` ungleich `NULL`.
  An der Adresse `e` sei der Wert `x` (von Typ `T`) gespeichert. Dann ist
  `*e`
  ein **Ausdruck vom Typ `T` mit Wert `x`**.
- Sei `p` ein Zeiger auf eine Variable `x` vom Typ `T`. Dann ist
  `*p`
  eine **Variable vom Typ `T`** und ein **Alias** (anderer Name) für `x`.
- Den Operator `*` nennt man **Dereferenzierungsoperator**.

Achtung

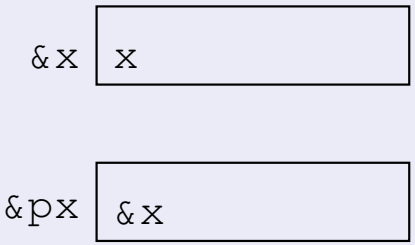
Ein Zeiger auf `NULL` kann **nicht** dereferenziert werden. Der Versuch führt zu einem Programmabbruch.

Notation

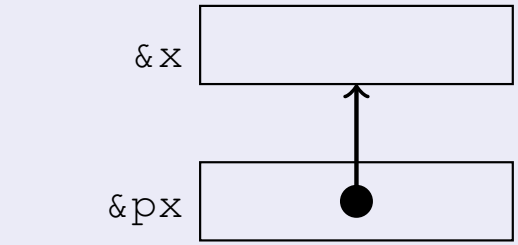
Darstellung von Zeigern im Speicher

```
T *px = &x;
```

Darstellung im Speicher mit Adressen:



Kurz-Darstellung:



Beispiel 9.5

```
double y,z,*p;
p = &y; /*Hier ist *p ein Alias fuer y*/
*p = 5; /*y und *p haben den Wert 5*/
p = &z; /*Jetzt ist *p ein Alias fuer z*/
```

Der Modifikator `const`

Konstante Zeiger

Deklaration eines **konstanten Zeigers** `p` auf `T`:

```
T * const p;
```

- `p` kann **nicht** umgebogen werden
- `*p` kann geändert werden

Deklaration eines Zeigers `p` auf `T` **auf einen konstanten Wert**:

```
const T *p;
```

- `p` kann umgebogen werden
- `*p` kann **nicht** geändert werden

Kombination:

```
const T * const p;
```

Typische Fehler bei der Benutzung von Zeigern

1. Zeiger wird dereferenziert, zeigt aber **irgendwohin** (d.h. er wurde nicht initialisiert) oder **nirgendwohin** (d.h. er hat den Wert `NULL`). Beispiel:
```
int *p;
*p = 5; /*Zugriff in nicht reservierten Bereich */
```
2. Zeiger wird dereferenziert, aber die referenzierte Variable wurde **nicht initialisiert**. Beispiel:
```
int x;
int *p = &x;
++(*p); /*Rechnen mit undefiniertem Wert */
```
3. Wertzuweisung an Zeiger mit **inkompatiblem Datentyp**. Beispiele:
```
int x;
double *p = &x; /*Compilerfehler */
int *q = x; /*Compilerfehler */
```
4. Wertzuweisung **an Feldvariable**. Beispiel:
```
int v[5];
++v; /*Compilerfehler */
```

Was ist das Call-by-Reference-Prinzip?

Definition 9.6 (Call-by-Reference-Prinzip)

Wird bei Funktionsaufruf **die Adresse einer Variablen übergeben**, so kann der Wert dieser Variablen bei Abarbeitung der Funktion über Dereferenzierung geändert werden.

Beispiel 9.7

An die `scanf`-Funktion wird bei Aufruf die Adresse einer Variablen übergeben. Über diese Adresse wird mittels Dereferenzierung der umgewandelte Wert in der Variable gespeichert.

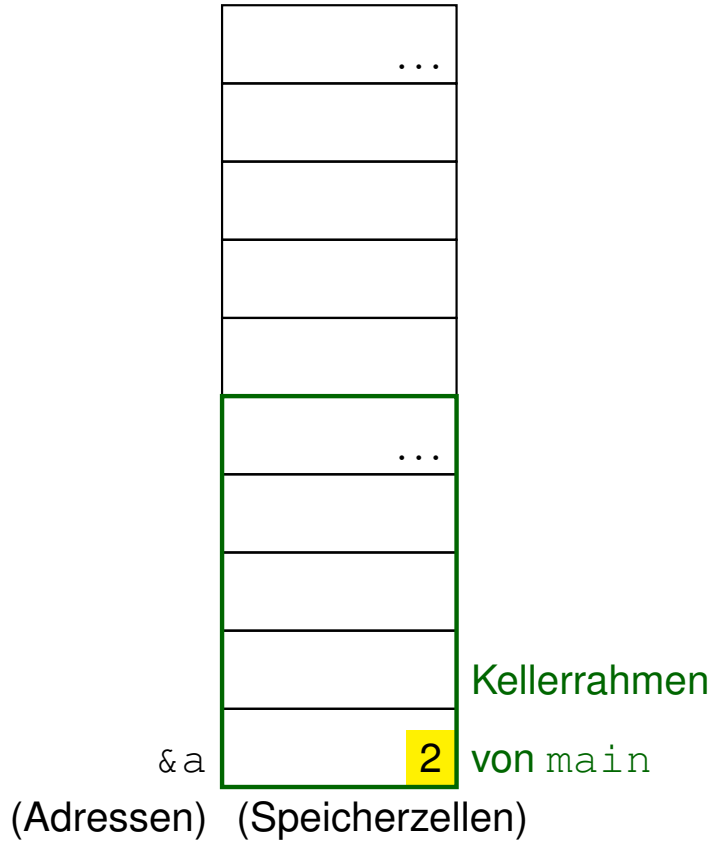
Achtung

Die übergebenen Adressen unterliegen wie besprochen dem Call-by-Value-Prinzip, d.h. Zeiger können in einer Funktion **nicht umgebogen werden**.

Beispiel für das Call-by-Reference-Prinzip

1 Variable anlegen und initialisieren

```
1  #include <stdio.h>
2
3  void increment(int *n);
4
5  int main() {
6    int a = 2;
7    increment(&a);
8    printf("%i", a);
9    return 0;
10 }
11
12 void increment(int *n) {
13   ++(*n);
14 }
```



Beispiel für das Call-by-Reference-Prinzip

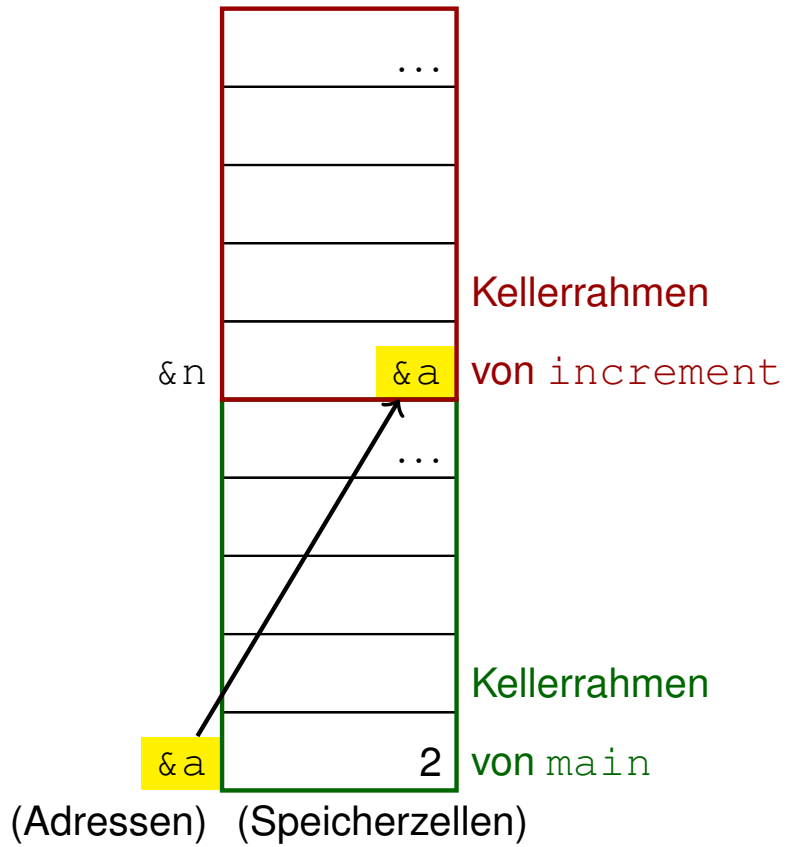
1. Variable anlegen und initialisieren
2. Adresse der Variable übergeben

```
1  #include <stdio.h>

3  void increment(int *n);

5  int main() {
6    int a = 2;
7    increment(&a);
8    printf("%i", a);
9    return 0;
10 }

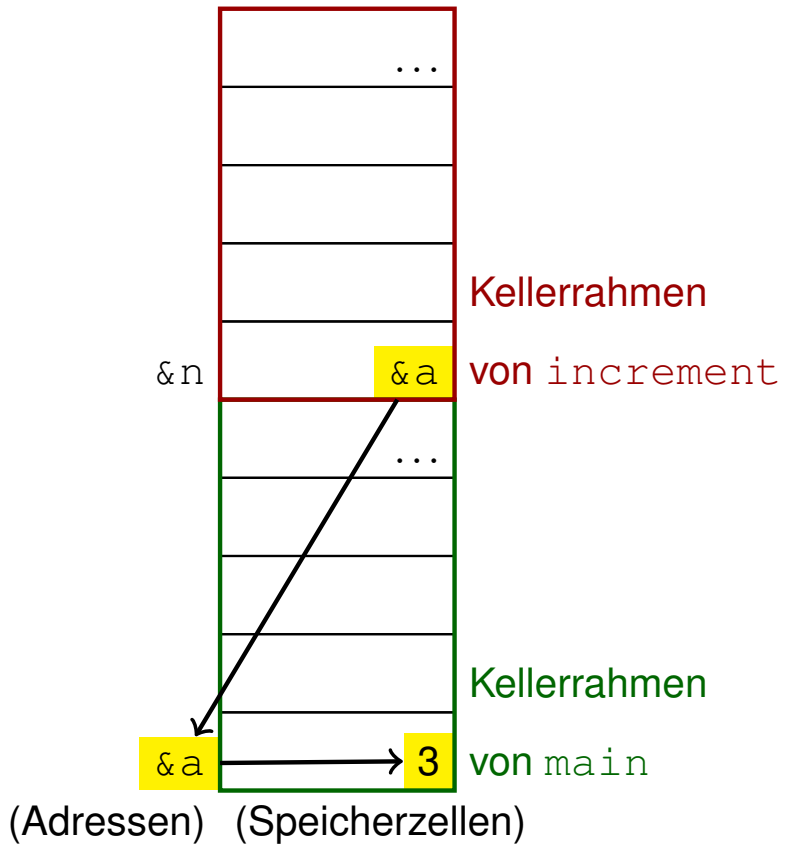
12 void increment(int *n) {
13   ++(*n);
14 }
```



Beispiel für das Call-by-Reference-Prinzip

1. Variable anlegen und initialisieren
2. Adresse der Variable übergeben
3. Wert über Dereferenzierung verändern

```
1  #include <stdio.h>
2
3  void increment(int *n);
4
5  int main() {
6    int a = 2;
7    increment(&a);
8    printf("%i", a);
9    return 0;
10 }
11
12 void increment(int *n) {
13   ++(*n);
14 }
```



Beispiel für das Call-by-Reference-Prinzip

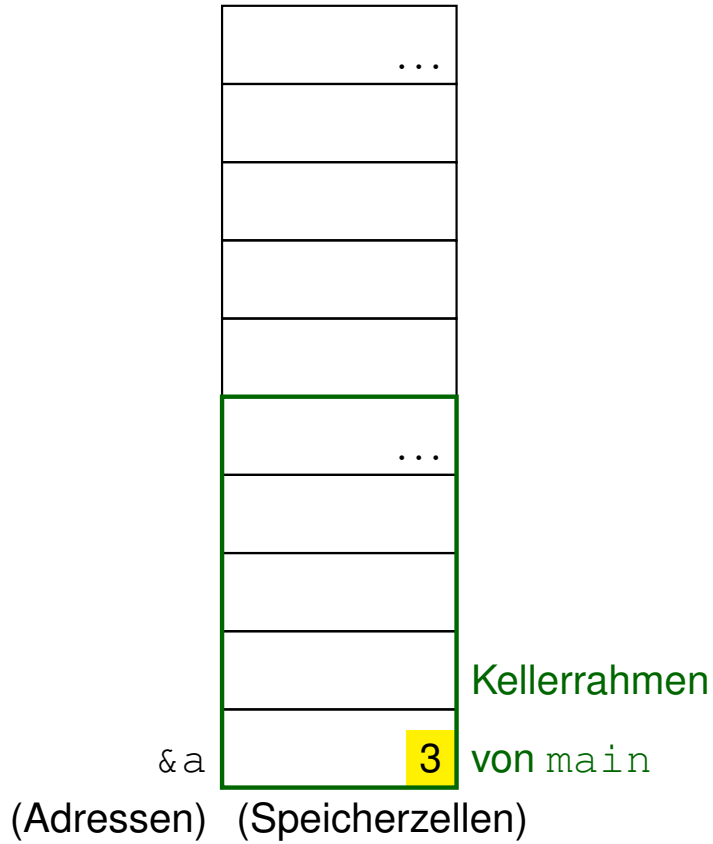
1. Variable anlegen und initialisieren
2. Adresse der Variable übergeben
3. Wert über Dereferenzierung verändern
4. Neuen Wert verwenden

```
1  #include <stdio.h>

3  void increment(int *n);

5  int main() {
6    int a = 2;
7    increment(&a);
8    printf("%i", a);
9    return 0;
10 }

12 void increment(int *n) {
13   ++(*n);
14 }
```



Call by Reference in einer Eingabefunktion

Einlesen einer ganzen Zahl mit einer Funktion (hier ohne Berücksichtigung von Pufferfehlern):

- Verwende einen Zeiger auf `int` als Eingabeparameter (Zeile 1)
- Speichere die Benutzereingabe an der Bezugsadresse des Zeigers (Zeile 3)
- Benutze den Rückgabewert **allein** zur Unterscheidung zwischen Erfolgs- und Fehlerfällen (Zeilen 5 und 7):
  Da der eingelesene Wert hier nicht zurückgegeben wird, hat man keine Probleme mehr mit dem Festlegen von Rückgabewerten für Fehlerfälle!

```
1  int read_pos_p(int * in)
2  {
3    if (scanf("%i", in) != 1 || *in < 0 || getchar() != '\n' )
4    {
5      flush();
6      return INVALID_INPUT;
7    }
8    return VALID_INPUT;
9  }
```

Call by Reference im Hauptprogramm

Einlesen einer ganzen Zahl mit einer Funktion (hier ohne Berücksichtigung von Pufferfehlern):

- Lege eine Variable für die einzulesende Zahl an (Zeile 3)
- Übergebe die Adresse der Variable an die Einlesefunktion (Zeile 6)
- Führe Fehlerbehandlung durch anhand des Rückgabewerts (Zeilen 4 und 6)

```
1  int main(void)
2  {
3      int status = INVALID_INPUT, n;
4      while (status == INVALID_INPUT) {
5          printf("Nicht-negative ganze Zahl eingeben:\n");
6          if ((status = read_pos_p(&n)) == INVALID_INPUT)
7              printf("Eingabe ungueltig\n");
8      }
9      printf("Eingabe: %i\n", n);
10     return EXIT_SUCCESS;
11 }
```

Anmerkung: Jetzt sollte klar sein, wieso an `scanf` Adressen von Variablen übergeben werden

Adressen können verschoben werden

Definition 9.8 (Adressverschiebung)

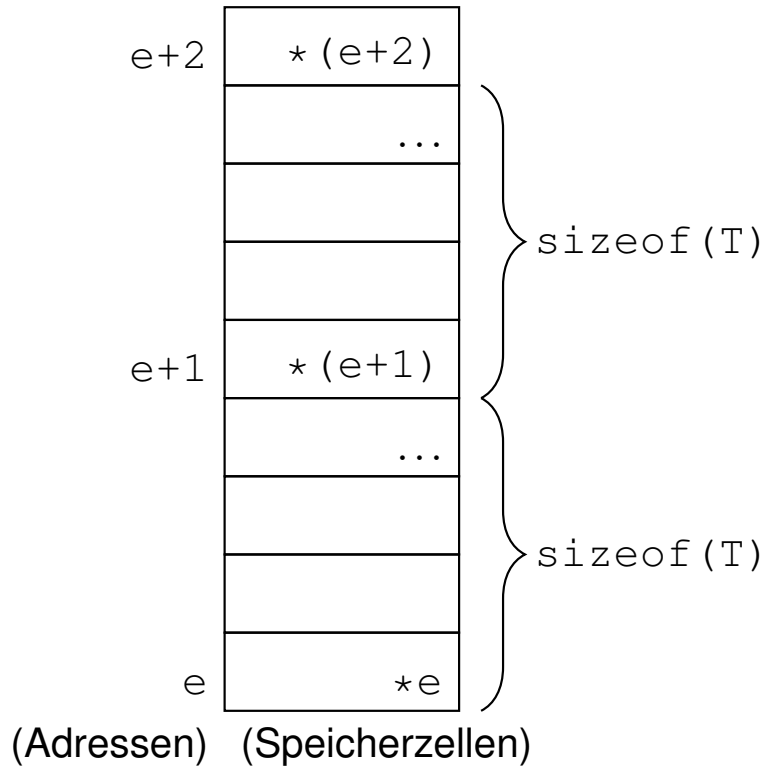
Ist `e` ein adresswertiger Ausdruck vom Typ `T*` mit Wert `a` und `n` ein ganzzahliger Ausdruck, so ist

`e + n`

ein **adresswertiger Ausdruck** vom Typ `T*` mit Wert

`a + (n * sizeof(T))`

(Die Adresse wird um `n`-mal den Speicherbedarf des Bezugstyps `T` verschoben)



Achtung

Vermeide Adressverschiebung in nicht reservierten Bereich

Anwendung: Felder

Definition 9.9

Ist `v` ein Feld vom Typ `T`, und ist `a` die Adresse der ersten Speicherzelle des Speicherbereichs von `v`, so ist

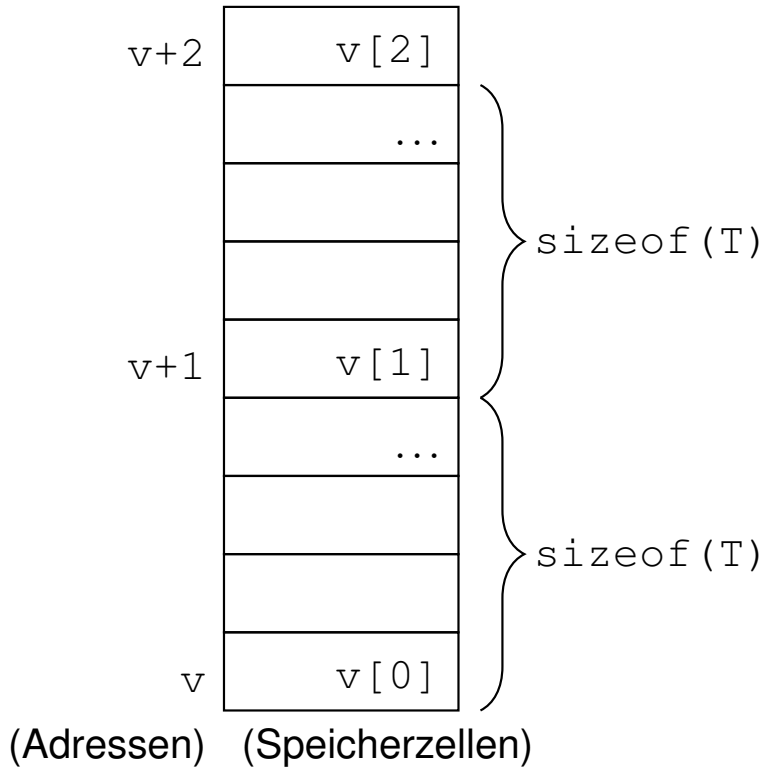
`v`

ein **konstanter adresswertiger Ausdruck** vom Typ `T*` mit Wert `a`.

Folgerungen

Ist `v` ein Feld vom Typ `T` und `n` ein ganzzahliger Ausdruck, so gilt

- `v+n` entspricht `&v[n]`
- `*(v+n)` entspricht `v[n]`



Notationen für Felder und Zeiger

Die Notationen für Zeiger und Felder entsprechen sich:

Feldnotationen

Ist `v` **ein Feld vom Typ `T`** und `n` ein ganzzahliger Ausdruck, so gilt

- `v+n` entspricht `&v[n]`
- `*(v+n)` entspricht `v[n]`

Zeigernotationen

Ist `p` **ein Zeiger auf `T`** und `n` ein ganzzahliger Ausdruck, so gilt

- `p+n` entspricht `&p[n]`
- `*(p+n)` entspricht `p[n]`

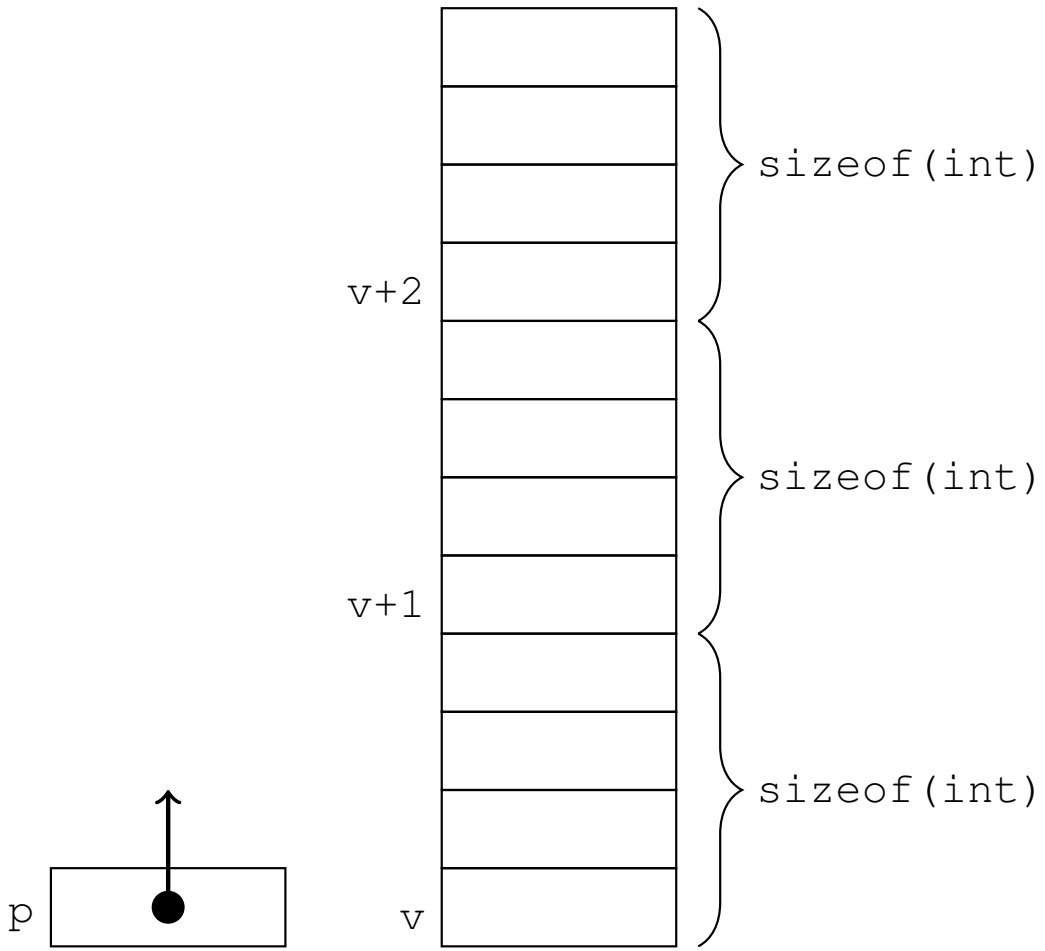
Zeiger und Felder als Eingabeparameter

Die Eingabeparameter `T v[]` und `T *v` sind gleichbedeutend. In beiden Fällen wird eine Adresse übergeben und in der Funktion mit einem Zeiger gerechnet.

Anwendung: Mit Adressverschiebung ein Feld durchlaufen

Mit `++p` wird der Zeiger `p` auf die Adresse der nächsten Feldkomponente umgebogen

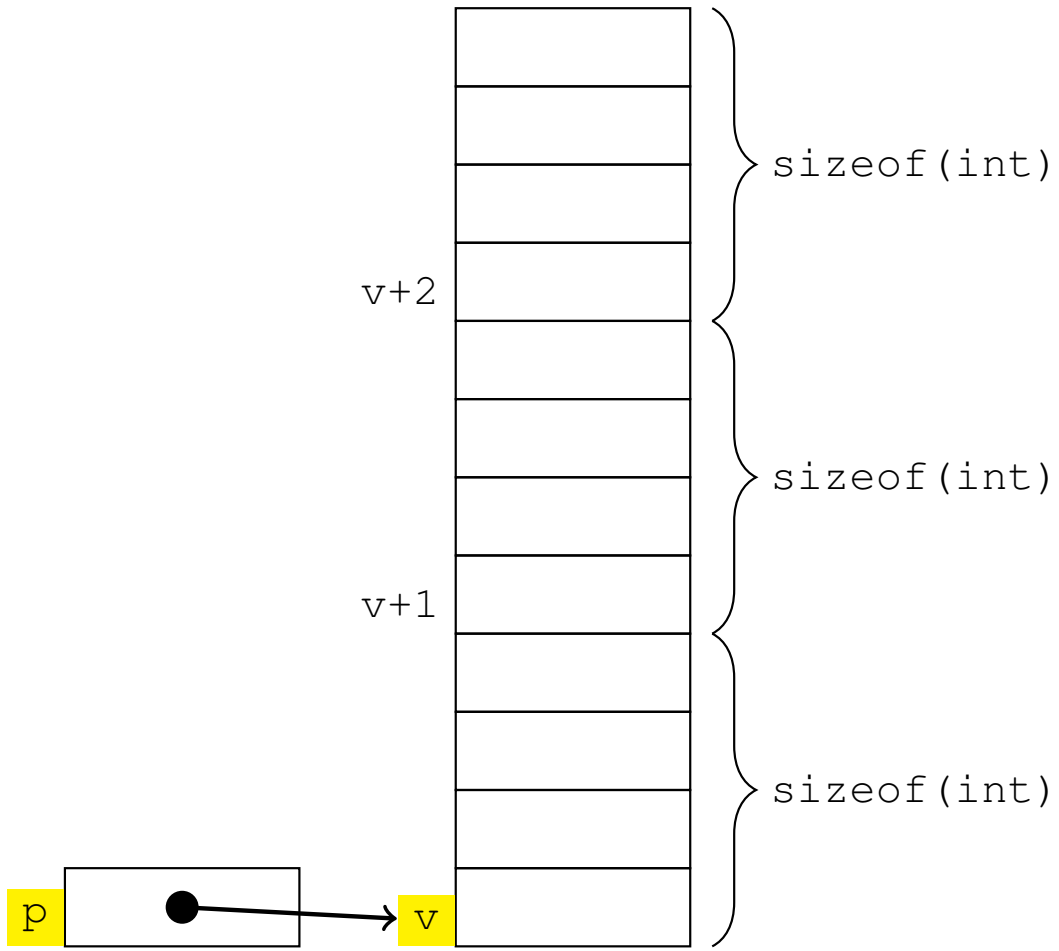
```
int main() {
  int v[3], *p, i;
  p = v;
  for (i = 0; i < 3;
       ++i) {
    *p = i;
    ++p;
  }
  return 0;
}
```



Anwendung: Mit Adressverschiebung ein Feld durchlaufen

Mit `++p` wird der Zeiger `p` auf die Adresse der nächsten Feldkomponente umgebogen

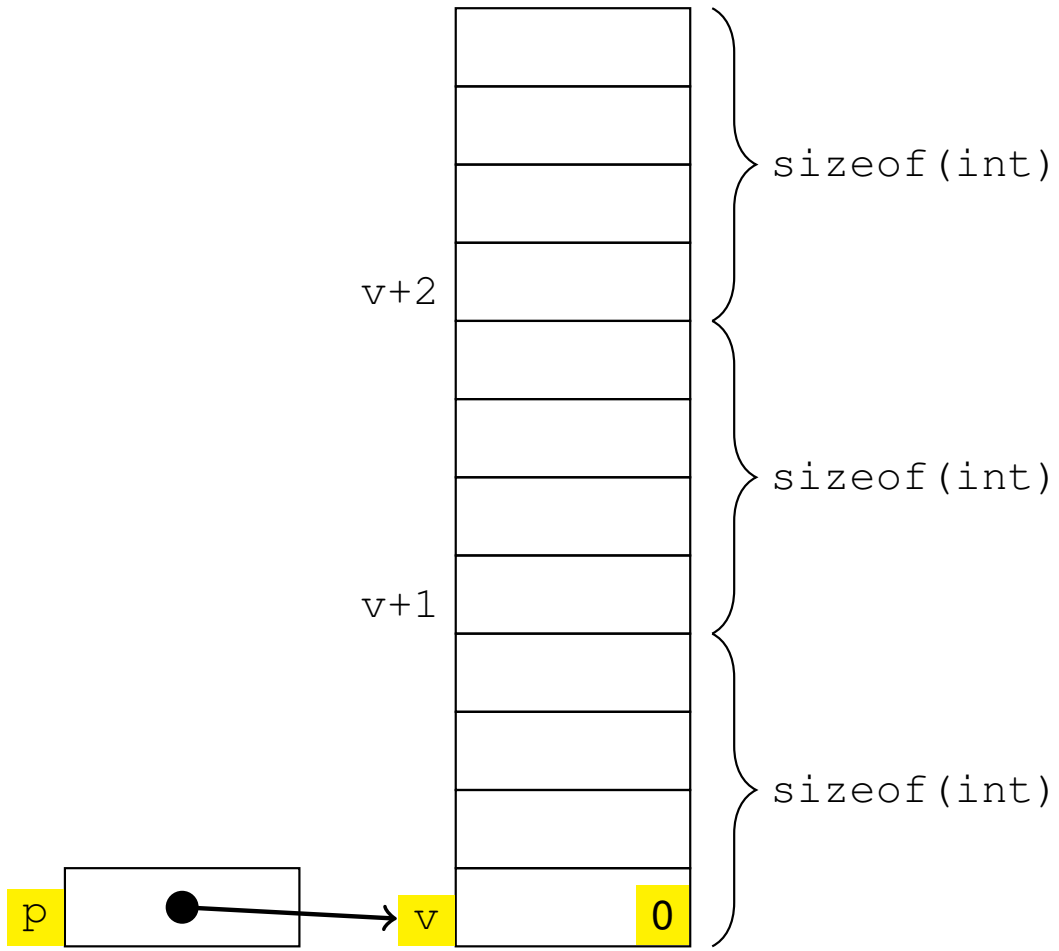
```
int main() {
  int v[3], *p, i;
  p = v;
  for (i = 0; i < 3;
       ++i) {
    *p = i;
    ++p;
  }
  return 0;
}
```



Anwendung: Mit Adressverschiebung ein Feld durchlaufen

Mit `++p` wird der Zeiger `p` auf die Adresse der nächsten Feldkomponente umgebogen

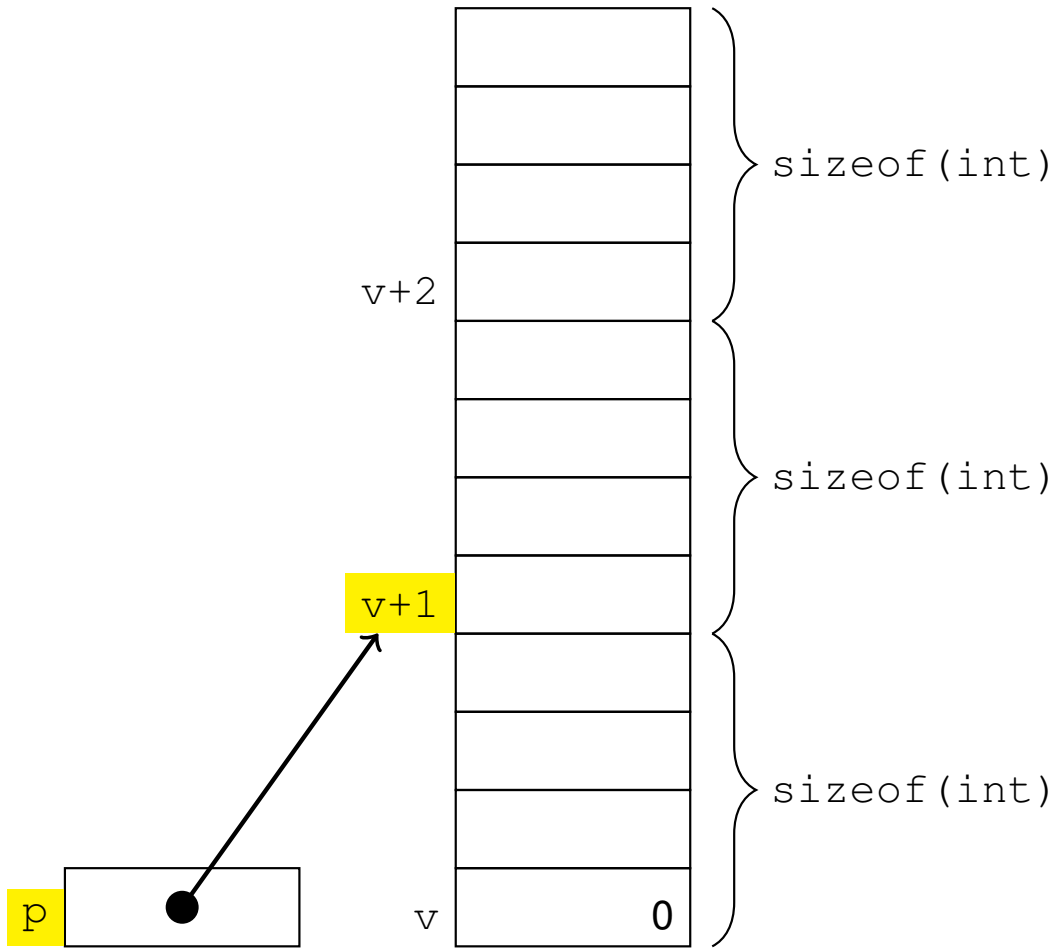
```
int main() {
  int v[3], *p, i;
  p = v;
  for (i = 0; i < 3;
       ++i) {
    *p = i;
    ++p;
  }
  return 0;
}
```



Anwendung: Mit Adressverschiebung ein Feld durchlaufen

Mit `++p` wird der Zeiger `p` auf die Adresse der nächsten Feldkomponente umgebogen

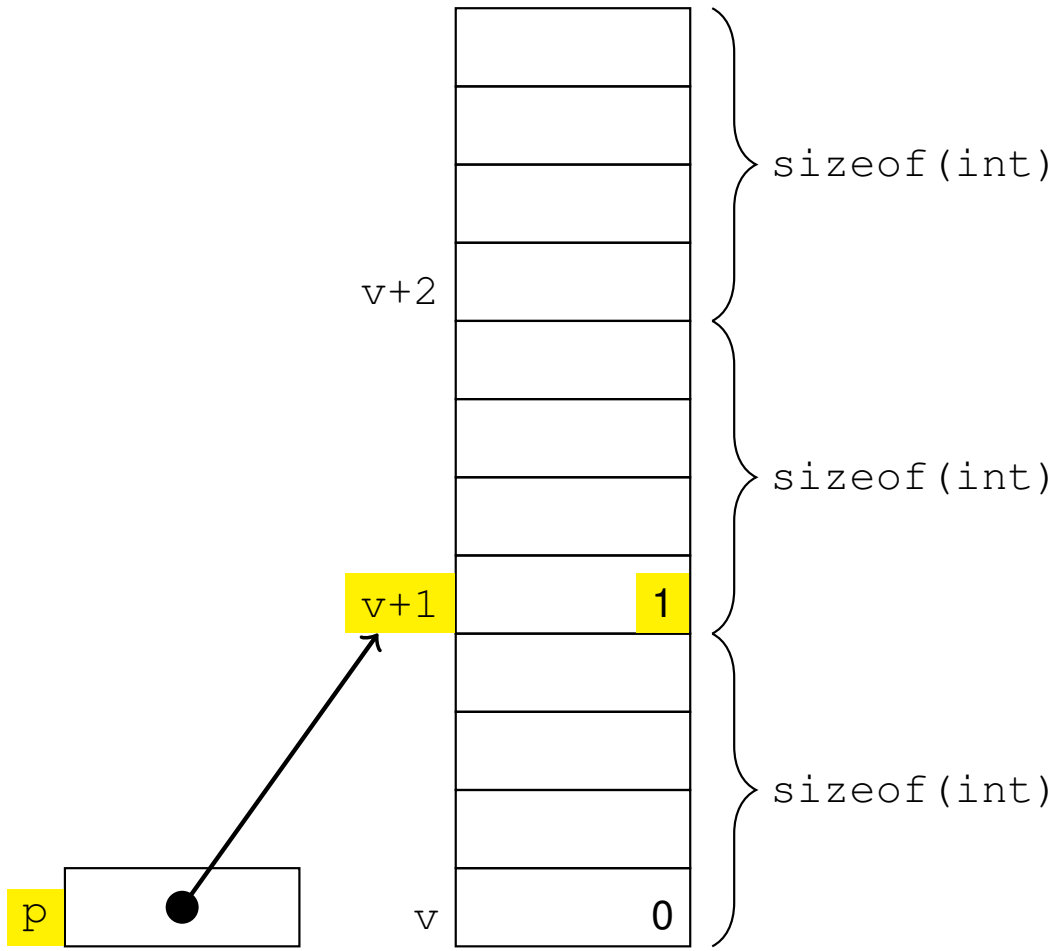
```
int main() {
  int v[3], *p, i;
  p = v;
  for (i = 0; i < 3;
       ++i) {
    *p = i;
    ++p;
  }
  return 0;
}
```



Anwendung: Mit Adressverschiebung ein Feld durchlaufen

Mit `++p` wird der Zeiger `p` auf die Adresse der nächsten Feldkomponente umgebogen

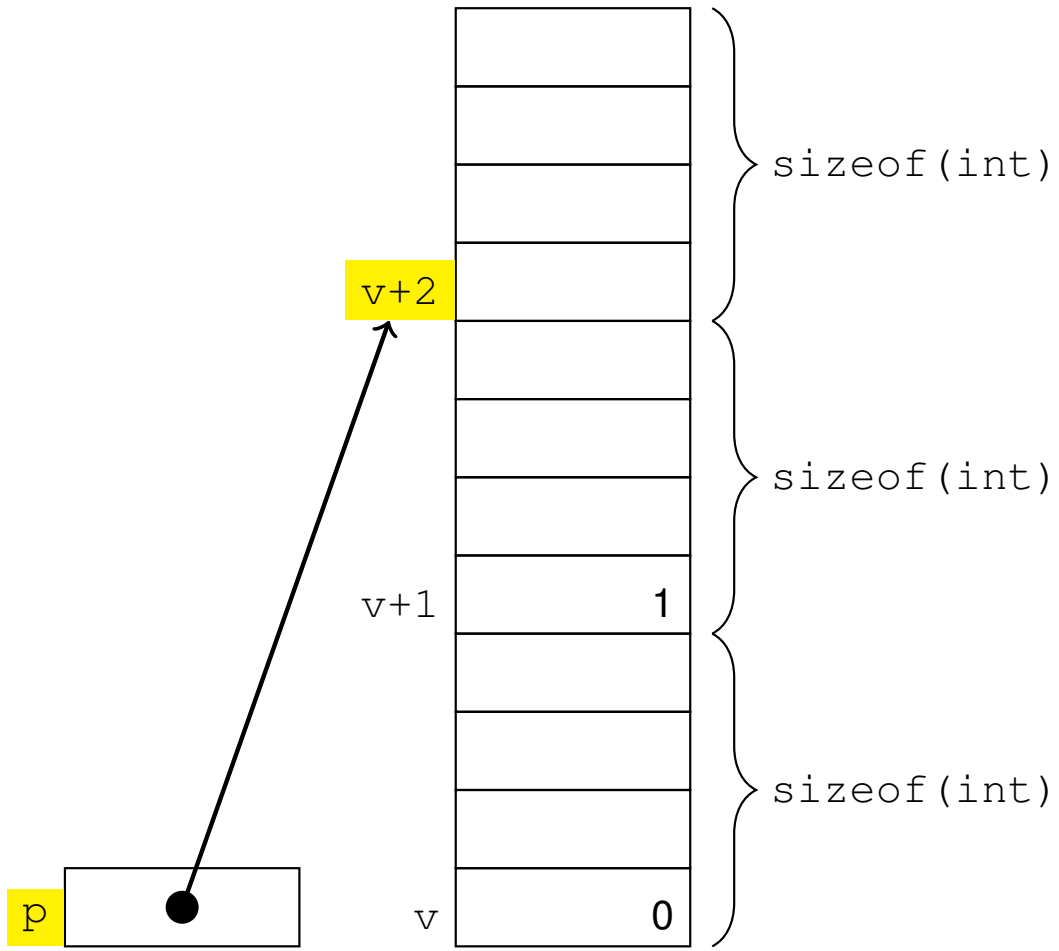
```
int main() {
  int v[3], *p, i;
  p = v;
  for (i = 0; i < 3;
        ++i) {
    *p = i;
    ++p;
  }
  return 0;
}
```



Anwendung: Mit Adressverschiebung ein Feld durchlaufen

Mit `++p` wird der Zeiger `p` auf die Adresse der nächsten Feldkomponente umgebogen

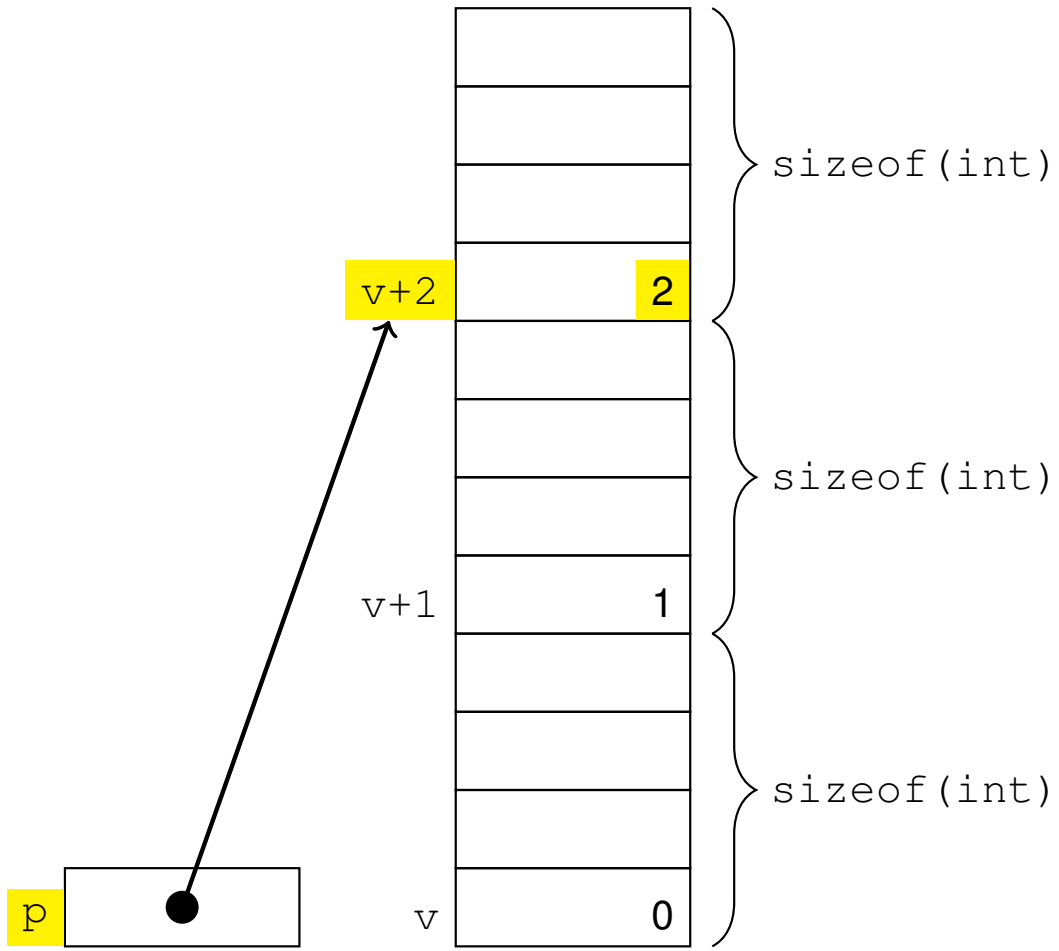
```
int main() {
  int v[3], *p, i;
  p = v;
  for (i = 0; i < 3;
        ++i) {
    *p = i;
    ++p;
  }
  return 0;
}
```



Anwendung: Mit Adressverschiebung ein Feld durchlaufen

Mit `++p` wird der Zeiger `p` auf die Adresse der nächsten Feldkomponente umgebogen

```
int main() {
  int v[3], *p, i;
  p = v;
  for (i = 0; i < 3;
       ++i) {
    *p = i;
    ++p;
  }
  return 0;
}
```



Zeiger auf `char` sind Zeichenketten

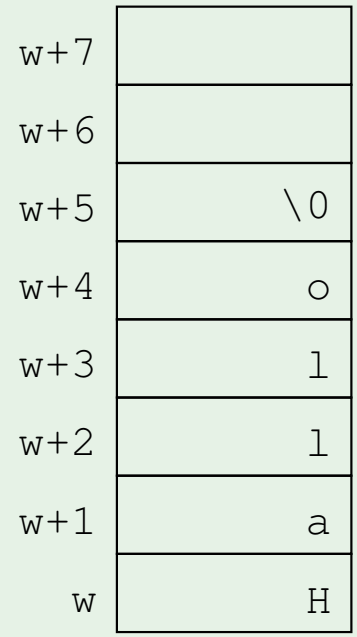
Definition 9.10 (Zeichenkette)

Ein Zeiger `p` auf `char` repräsentiert genau die Folge von Zeichen (**Zeichenkette**), die im Speicher von der Adresse `p` bis zur Adresse des nächsten `'\0'`-Zeichens abgelegt sind.

Beispiel 9.11

```
char w[8];
strcpy(w,"Hallo");
```

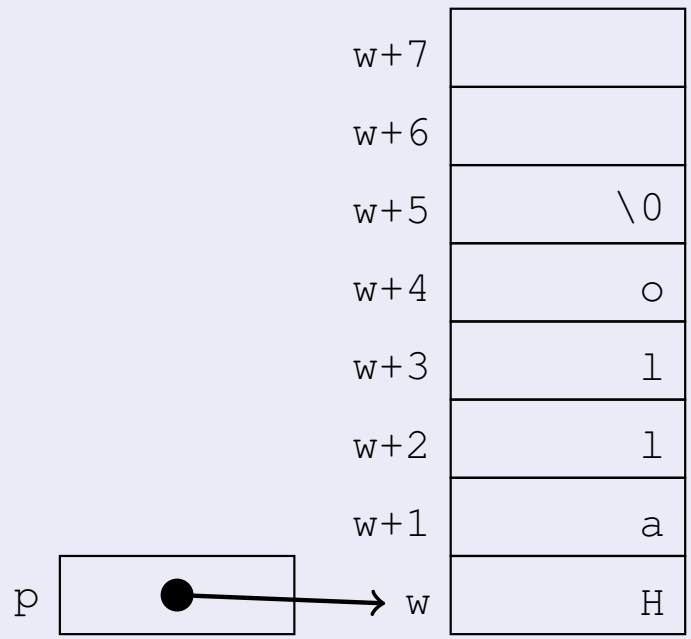
- `char *p = w;`:
  `p` repräsentiert die Zeichenkette `"Hallo"`
- `char *p = w + 2;`:
  `p` repräsentiert die Zeichenkette `"llo"`



Zeichenkette als Feld vs. Zeichenkette als Zeiger

Besonderheiten bei der Wertzuweisung in Deklarationen

- `char w[] = "Hallo";`
  `w` ist eine **konstante Adresse** einer Zeichenkette
  (`++w;` erzeugt Compilerfehler)
- `char *p = "Hallo";`
  `p` ist ein variabler Zeiger auf eine **konstante Zeichenkette**
  (`p[0] = 'e';` erzeugt Compilerfehler)



Zeiger und Zeichenketten - Zeichenkette kopieren

Benutze Adressverschiebung statt Feldindex zum Durchlaufen von Zeichenketten.

- Durchlaufe `eingabe` und `ausgabe` vorwärts mit Adressverschiebung und kopiere die Zeichen von `eingabe` nach `ausgabe` (Zeile 2)

```
1 void my_strcpy(char *ausgabe, const char *eingabe) {
2   while ((*(ausgabe++) = *(eingabe++)) != '\0') {}
3 }
```

(unsichere Variante!)

Zeiger als Eingabeparameter

Ein Eingabeparameter `p` der Form

`T p[]` oder `T * p`

ist adresswertig (vom Typ `T*`) und wird in der Funktion als Zeiger (auf `T`) verwendet.

- Beide Formen sind gleichwertig
- **Folgerung**: In einer Funktion gibt `sizeof(p)` nicht die Anzahl der Komponenten von `p` zurück, sondern den Speicherbedarf einer Adresse
- In der Regel wird in einer Funktion **nicht** überprüft, ob für `p` der Wert `NULL` übergeben wurde: Der Programmierer ist selbst verantwortlich für die korrekte Benutzung der Funktion.

Beispiele aus `string.h`

- `size_t strlen(const char * cs)`
- `char * strcpy(char * s, const char * ct)`

Zeiger als Rückgabetyp

Rückgabeadressen

Ist `T` eine Datentyp und
`T*`
der Rückgabetyp einer Funktion, so gibt die Funktion eine Adresse vom Typ `T*` oder `NULL` zurück.

- `NULL` wird üblicherweise zur Anzeige eines aufgetretenen Fehlers verwendet und kann beim Aufrufer zur Fehlerbehandlung verwendet werden
- Ist `void*` der Rückgabetyp einer Funktion, so gibt die Funktion eine beliebige Adresse (ohne Bezugstyp) zurück: Vor einer Dereferenzierung muss die Adresse einem getypten Zeiger zugewiesen werden (mit automatischer Typumwandlung).

Beispiel 9.12

Einige Zeichenkettenfunktionen aus `string.h` haben den Rückgabetyp `char*`.

Zeiger als Rückgabetyp

Achtung

Niemals eine Adresse eines freigegebenen Speicherbereichs zurückgeben!

Beispiel 9.13 (Murks mit Rückgabeadressen)

Die zurückgebene Adresse befindet sich in einem nach Funktionsabarbeitung wieder freigegebenen *function stack frame*

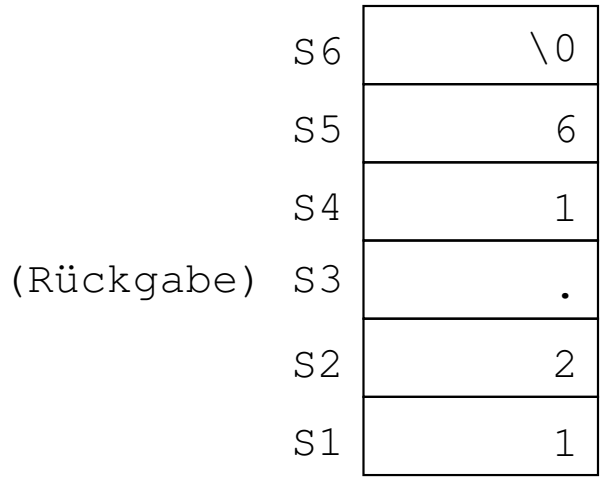
```
int * murks() {
  int n = 5;
  return(&n);
}

int main() {
  int *p;
  p = murks();
  printf("%d\n", *p); /*Zugriff auf undefinierten Bereich*/
}
```

Zeiger als Rückgabetyp

Beispiele aus `string.h` (unvollständig):

- `char * strcpy(char * s, const char * ct)`: liefert Adresse von `s`
- `char *strchr(const char * cs, int c)`: liefert die Adresse des ersten Vorkommens von `c` in `cs`, oder `NULL`, falls `c` in `cs` nicht vorkommt:



`strchr("12.16",'.');` liefert Adresse `S3` - also die Zeichenkette `".16"`!

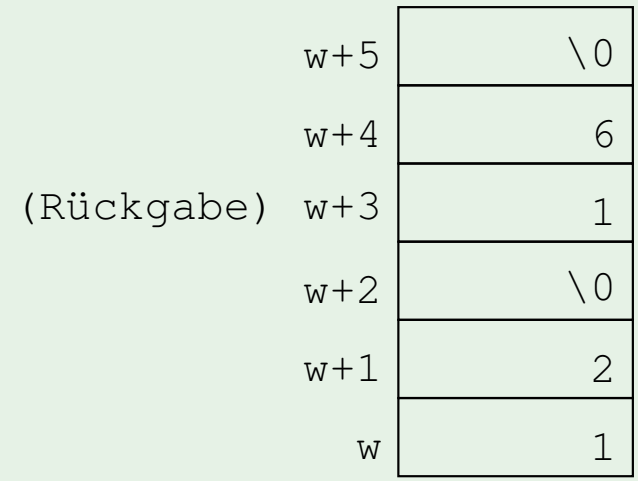
Zeiger als Rückgabetyp

Beispiel 9.14 (Zerlegen von Zeichenketten - eigene Funktion)

- Ersetzt das erste Vorkommen von `c` in `w` durch `\0` und gibt die Adresse des nachfolgenden Zeichens zurück, falls `c` vorkommt. Ansonsten wird `NULL` zurückgegeben (**siehe auch Bibliotheksfunktion `strtok`**).
- Die Zeichenkette `w` wird so in zwei Teile zerlegt. Auf den ersten Teil greift man mit `w` zu, auf den zweiten Teil mit der zurückgegebenen Adresse.

```
char * split(char * w, char c) {
  int i = 0;
  while (w[i] != c && w[i] != '\0')
    ++i;
  if (w[i] == c) {
    w[i] = '\0';
    return &w[i + 1];
  }
  else
    return NULL;
}
```

```
split("12.16",'.');
```



Motivation

Verwendet man Felder zur Verwaltung von Zahlenfolgen und Zeichenketten, so muss man **statisch** (d.h. zum Zeitpunkt der Übersetzung des Quellcodes in Maschinencode) die Anzahl der Feldkomponenten festlegen. Üblicherweise wählt man hierfür die maximale Anzahl der voraussichtlich benötigten Komponenten.

- Damit können zur Laufzeit höchstens so viele Komponenten verwendet werden.
- Wenn man weniger Komponenten braucht, verschwendet man Speicherplatz.

Verbesserung

Möglichkeit, die Anzahl der Feldkomponenten **dynamisch** (d.h. zur Laufzeit des Programms) festzulegen.

Beispiel: Dynamisches Einlesen einer Zahlenfolge

- Zeile 6: Reservierung des erforderlichen Speicherbereichs über einen Zeiger
- Zeilen 7 - 10: Fehlerbehandlung
- Zeile 13: Speicherfreigabe

```
1  int main(void)
2  {
3    int size, i;
4    int * sequence;
5    /*Eingabe der Länge der Zahlenfolge*/
6    sequence = malloc(size * sizeof(int));
7    if (sequence == NULL) {
8      printf("Speicherfehler\n");
9      return EXIT_FAILURE;
10   }
11   /*Zahlen einlesen*/
12   /*Ausgabe*/
13   free(sequence);
14   return EXIT_SUCCESS;
15 }
```

Beispiel: Dynamisches Einlesen einer Zahlenfolge

- Zeile 10: $(i+1)$-te Zahl einlesen mit Fehlerüberprüfung
- Zeile 12: Speicherfreigabe im Fehlerfall

```
1  int main(void)
2  {
3    int size, i;
4    int * sequence;
5    /*Eingabe der Länge der Zahlenfolge*/
6    /*Speicherplatz dynamisch reservieren*/
7    /*Fehlerbehandlung*/
8    for (i = 0; i < size; ++i) {
9      printf("Eingabe %i-te Zahl: ", i + 1);
10     if (read_pos_p(&sequence[i]) == INVALID_INPUT) {
11       printf("%i-te Eingabe ungueltig\n", i + 1);
12       free(sequence);
13       return EXIT_FAILURE;
14     }
15   }
16   /*Ausgabe*/
17   /*Speicherfreigabe*/
18 }
```

Dynamische Reservierung von Speicher

Speicherreservierung ohne Initialisierung

Die `stdlib.h`-Bibliotheksfunktion

```
void * malloc (size_t size)
```

versucht einen **zusammenhängenden Speicherbereich im Heap** in der Größe von `size` Byte zu reservieren, hat den Rückgabewert `NULL` im Fehlerfall, und gibt die Adresse (nicht getypt) der ersten Speicherzelle des reservierten Bereichs im Erfolgsfall zurück.

- Der Speicherplatz wird **nicht automatisch** wieder freigegeben, dies muss durch einen separaten Funktionsaufruf erfolgen
- Der Typ `size_t` ist der Rückgabetyp des `sizeof`-Operators. Beim `gcc` entspricht er dem Typ `unsigned long`.

Dynamische Reservierung von Speicher

Systemunabhängige Festlegung der Speichergröße mit `sizeof`

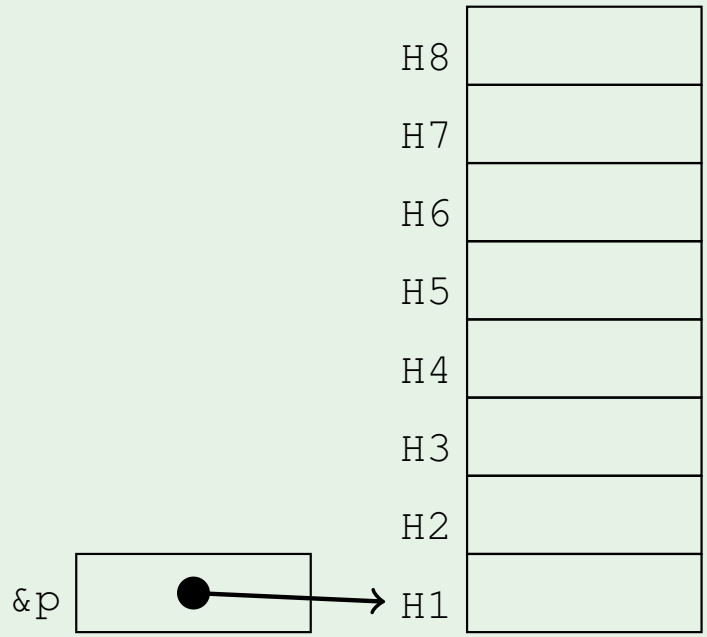
Größe des benötigten Speicherplatzes festlegen:

- für Grund-Datentyp `T`: `malloc(sizeof(T))`
- für Feld der Länge `n` vom Typ `T`: `malloc(n * sizeof(T))`

Beispiel 9.15 (Dynamische Speicherreservierung im Speicher)

```
int * p = malloc(2 *
sizeof(int))
```

- Es werden 8 Byte Speicherplatz im Heap reserviert
- `p` zeigt auf diesen Speicherplatz
- Der Wert von `p` ist die Adresse `H1`
- `&p` und `H1` sind unterschiedliche Adressen
- `p[0]` ist eine `int`-Variable an Adresse `H1`
- `p[1]` ist eine `int`-Variable an Adresse `H5`



Dynamische Reservierung von Speicher

Beispiel 9.16 (Dynamisches Kopieren von Zeichenketten)

- Zeile 3: Benötigten Speicherplatz dynamisch reservieren (Platz für `'\0'` nicht vergessen!)
- Zeilen 4 - 5: Fehlerbehandlung
- Zeile 6: Kopieren und Adresse zurückgeben

```
1  char * string_d_copy(const char * org)
2  {
3    char * copy = malloc((strlen(org) + 1) * sizeof(char));
4    if (copy == NULL)
5      return NULL;
6    return strcpy(copy, org);
7  }
```

Dynamische Reservierung von Speicher

Beispiel 9.17 (Dynamisches Kopieren von Zeichenketten)

- Zeile 3: Zeichenkette kopieren - der in der Funktion `string_d_copy` dynamisch reservierte Speicherplatz kann in `main` weiter benutzt werden
- Zeilen 4 - 7: Fehlerbehandlung
- Zeile 9: Freigabe des in `string_d_copy` dynamisch reservierten Speicherbereichs

```
1  int main(void)
2  {
3    char * error = string_d_copy("Error");
4    if (error == NULL) {
5      printf("Speicherfehler\n");
6      return EXIT_FAILURE;
7    }
8    printf("Zeichenkette: %s\n", error);
9    free(error);
10   return EXIT_SUCCESS;
11 }
```

Dynamische Reservierung von Speicher

Schema zur Anwendung dynamischer Speicherreservierung

1. Aufruf einer Bibliotheksfunktion zur Speicherreservierung und Zuweisung des Rückgabewerts an spezifischen Zeiger (automatische Typumwandlung nach `T*`):
   ```
   T *p = malloc (n * sizeof(T))
   ```
2. Fehlerbehandlung durchführen (ein Zeiger auf `NULL` kann nicht dereferenziert werden!):
   ```
   if (p == NULL)/*Fehlerbehandlung */
   ```
3. Speicherung und Verwaltung von Werten im reservierten Bereich über Dereferenzierung des Zeigers:
   ```
   p[i] = /*Wertzuweisung */
   ```
4. Nach der Benutzung den reservierten Speicherplatz über den Zeiger wieder freigeben:
   ```
   free(p)
   ```

Speicherplatz wieder freigeben

- Dynamisch im Heap reservierter Speicherplatz muss explitzit wieder freigegeben werden. Dies geschieht nicht automatisch.
- Dynamisch reservierter Speicherplatz kann so auch in einer Funktion reserviert werden, und außerhalb der Funktion weiterbenutzt werden

Speicherplatz freigeben

Die `stdlib.h`-Bibliotheksfunktion

```
void free (void *p)
```

gibt dynamisch reservierten Speicherplatz, **auf den `p` zeigt**, wieder frei.

- Falls `p` **nicht** auf einen dynamisch reservierten Speicherbereich zeigt, ist das Verhalten undefiniert und es kann sogar zum Programmabsturz kommen
- Zeigt `p` auf `NULL`, so ist die Anweisung wirkungslos

Speicherplatz wieder freigeben

Speicherleck

Ein **Speicherleck** ist ein dynamisch reservierter und nicht wieder freigegebener Speicherbereich, der

- nicht mehr benutzt wird
- und auf den u.U. sogar nicht mehr zugegriffen werden kann

Falls zuvor dynamisch reservierter Speicherbereich nicht mehr benutzt werden soll (z.B. bei Auftreten eines Fehlers oder Beendigung des Programms), muss dieser wieder freigegeben werden.

Dynamisch reservierten Speicherplatz immer freigeben!

Wenn man Speicher immer nur reserviert und nicht wieder freigibt, wenn er nicht mehr benötigt wird, kann er schließlich voll werden, was zu schweren Fehlern führt

Dynamische Reservierung von Speicher (Fortsetzung)

Speicherreservierung mit Initialisierung

Die `stdlib.h`-Bibliotheksfunktion

```
void * calloc (size_t n, size_t size)
```

versucht einen **zusammenhängenden Speicherbereich im Heap** in der Größe von `n * size` Byte zu reservieren und hat folgenden Rückgabewert:

- `NULL` im Fehlerfall
- Adresse (nicht getypt) des reservierten Bereichs im Erfolgsfall

Im Erfolgsfall wird der Speicherbereich mit `0`-Werten initialisiert.

Dynamische Reservierung von Speicher (Fortsetzung)

Speicheranpassung

Die `stdlib.h`-Bibliotheksfunktion

```
void * realloc (void *p, size_t size)
```

versucht einen dynamisch reservierten Speicherbereich im Heap, auf den `p` zeigt, auf die Größe von `size` Byte **anzupassen** (zu vergrößern oder zu verkleinern) und hat folgenden Rückgabewert:

- `NULL` im Fehlerfall (der von `p` referenzierte Bereich bleibt unverändert)
- Adresse (nicht getypt) des neu reservierten Bereichs im Erfolgsfall

Für die Eingabeparameter gilt:

- Falls `p` **nicht** auf einen dynamisch reservierten Speicherbereich zeigt, ist das Verhalten undefiniert und es kann zum Programmabsturz kommen.
- Hat `p` den Wert `NULL`, so verhält sich `realloc` wie `malloc`
- Hat `size` den Wert `0`, so verhält sich `realloc` wie `free`.

Dynamische Reservierung von Speicher (Fortsetzung)

Speichervergrößerung

- Im Falle einer Vergrößerung des Speicherbereichs wird zuerst versucht, den bisherigen Bereich zu vergrößern.
- Falls dies nicht möglich ist, wird an einer anderen Stelle ein neuer Bereich ausreichender Größe gesucht
- Im Erfolgsfall wird der bisherige Inhalt dorthin kopiert (der alte Speicherbereich wird dann freigegeben). Der Inhalt des zusätzlichen Bereichs ist undefiniert.

Speicherverkleinerung

Der Inhalt im verkleinerten Speicherbereich bleibt erhalten.

Dynamische Speicheranpassung

Beispiel 9.18 (Dynamisches Verlängern von Zeichenketten)

- Zeile 3: Speicherplatz anpassen (es wird Platz für `first`, `second` und `'\0'` benötigt)
- Zeilen 4 - 5: Fehlerbehandlung
- Zeile 6: Verlängern und Adresse zurückgeben

```
1  char * string_d_cat(char * first, const char * second)
2  {
3    first = realloc(first, (strlen(first) + strlen(second) +
         1) * sizeof(char));
4    if (first == NULL)
5      return NULL;
6    return strcat(first, second);
7  }
```

Dynamische Speicheranpassung

Beispiel 9.19 (Dynamisches Verlängern von Zeichenketten)

- Zeile 5: Zeichenkette verlängern - der in der Funktion `string_d_cat` dynamisch reservierte Speicherplatz kann in `main` weiter benutzt werden
- Zeile 13: Freigabe des in `string_d_cat` dynamisch reservierten Speicherbereichs

```
1  int main(void)
2  {
3    char * w;
4    char * error = string_d_copy("Error");
5    w = string_d_cat(error, ": Invalid Input");
6    if (w == NULL) {
7      printf("Speicherfehler\n");
8      free(error);
9      return EXIT_FAILURE;
10   }
11   error = w;
12   printf("Zeichenkette: %s\n", error);
13   free(error);
14   return EXIT_SUCCESS;
15 }
```

Dynamische Speicheranpassung

Beispiel 9.20 (Dynamisches Verlängern von Zeichenketten)

Zeilen 5 - 11: Fehlerbehandlung, inklusive Freigabe von nicht mehr benötigtem Speicherplatz
Zeile 5: die Adresse kann nicht an `error` zugewiesen werden, da `error` dann im Fehlerfall mit `NULL` überschrieben würde In diesem Fall entstünde ein Speicherleck, da keine Speicherfreigabe über `error` mehr möglich (Zeile 8)

```
1  int main(void)
2  {
3    char * w;
4    char * error = string_d_copy("Error");
5    w = string_d_cat(error, ": Invalid Input");
6    if (w == NULL) {
7      printf("Speicherfehler\n");
8      free(error);
9      return EXIT_FAILURE;
10   }
11   error = w;
12   printf("Zeichenkette: %s\n", error);
13   free(error);
14   return EXIT_SUCCESS;
15 }
```

Überblick

Eine **zweidimensionale** Datenstruktur wird repräsentiert durch eine Variable, deren Wert eine Adresse ist, an der wieder eine Adresse gespeichert ist.

Deklaration zweidimensionaler Datenstrukturen

- `T **p;`
  Zeiger auf einen Zeiger auf `T` / **Doppelzeiger**
- `T *v[N];`
  Feld von `N` Zeigern auf `T` (`[]` bindet stärker als `*`)
- `T (*p)[N];`
  Zeiger auf Feld mit `N` Komponenten vom Typ `T`
- `T v[N][M];`
  Feld von `N` Feldern mit je `M` Komponenten vom Typ `T`

Wertzuweisungen an Doppelzeiger

Eine Adresse kann einem Zeiger zugewiesen werden (in einer Wertzuweisung oder Parameterübergabe), wenn ihr Typ dem Bezugstyp des Zeigers entspricht.

Einem Zeiger `T** p` kann zugewiesen werden (Bezugstyp: `T*`):

- der Wert eines anderen Zeigers auf `T*`:
  ```
  T** q;
  p = q;
  ```
- die Adresse eines Zeigers auf *T*:
  ```
  T* q;
  p = &q;
  ```
- eine Feldadresse vom Typ `T*` (d.h. ein Feld von Zeigern):
  ```
  T* v[N];
  p = v;
  ```

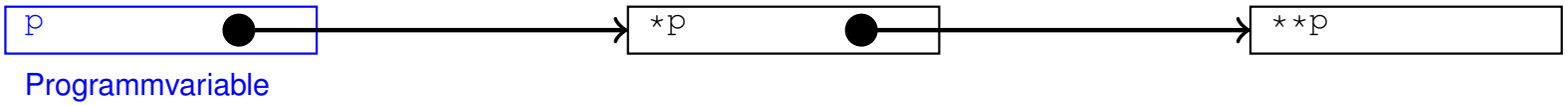
Folgerung

Die Eingabeparameter `T** p` und `T* v[]` sind gleichwertig. In beiden Fällen wird in der Funktion mit einem Doppelzeiger gerechnet.

Anwendung von Doppelzeigern

```
T **p;
```

- `p` ist ein Zeiger auf `T*` (`T*` ist also der Bezugstyp)
- `*p` ist ein Zeiger auf `T`.
- `**p` ist eine Variable vom Typ `T`.



Vor der Benutzung der Programmvariablen `p` muss man folgendes machen:

- Speicher für `*p` reservieren (statisch oder dynamisch) und initialisieren
- Speicher für `**p` reservieren (statisch oder dynamisch) und initialisieren

Anwendung Doppelzeiger: Einfachzeiger in Funktion umbiegen

Verwendung der `stdlib.h`-Funktion

```
double strtod(const char *s, char **endp)
```

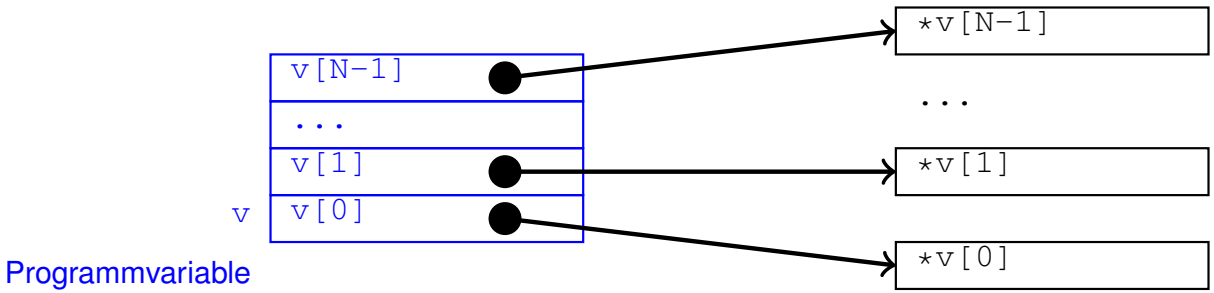
- Wandelt den Anfang der Zeichenkette `s` in `double` um, dabei werden Zwischenraumzeichen (White Space) am Anfang ignoriert.
- Speichert einen Zeiger **auf einen nicht umgewandelten Rest** der Zeichenkette `s` bei `*endp`, falls `endp` nicht `NULL` ist
- Bei Aufruf wird für `endp` die Adresse eines Einfach-Zeigers übergeben:

```
char *rest;
double x = strtod("123.5ab", &rest);
```

Felder von Zeigern

```
T *v[N];
```

- `v` ist ein Feld mit `N` Komponenten vom Typ `T*`. Der Compiler reserviert hierfür `N * sizeof(T*)` Byte Speicherplatz.
- `v[0],...,v[N-1]` sind Zeiger auf `T`.
- `*v[0],...,*v[N-1]` sind Variablen vom Typ `T`.



Vor der Benutzung der Programmvariablen `v` muss man folgendes machen:

- Speicher für `*v[0],...,*v[N-1]` reservieren (statisch oder dynamisch) und initialisieren

Anwendung Feld von Zeigern: Gemeinsamer Zugriff auf verschiedene Variablen

Gleiche Operationen auf verschiedenen Variablen, indem Variablen über gemeinsames Feld von Zeigern erreichbar gemacht werden

```
1  int main(void) {
2      int *zeigerfeld[LAENGE_ZEIGERFELD];
3      int a = 1;
4      int b = 2;
5      int c = 3;
6      int i;
7      zeigerfeld[0] = &a;
8      zeigerfeld[1] = &b;
9      zeigerfeld[2] = &c;
10     for (i = 0; i < LAENGE_ZEIGERFELD; i++) {
11         *zeigerfeld[i] *= 10;
12     }
13     for (i = 0; i < LAENGE_ZEIGERFELD; i++) {
14         printf("Variable␣%c:␣", i+97);
15         printf("%i\n", *zeigerfeld[i]);
16     }
17     return 0;
18 }
```

Motivation

Es gibt viele Problemstellungen und Anwendungen, deren Daten nicht geeignet mit eindiemsionalen Feldern bzw. einfachen Zeigern verwaltet werden können.

Beispiel 9.21 (Temperatur für jeden Tag speichern)

Speichere für jeden Tag eines Jahres die Mittagstemperatur:

- Bisher können wir die Temperaturen als Folge $(t_i)_{1 \leq i \leq 365}$ von Werten für jeden Tag darstellen, die wir in einem Feld speichern.
- Dies ist jedoch für den Zugriff über eine Datumsangabe ungeeignet (Was war die Temperatur am 1. Mai?)
- Bessere Darstellung: Speichere die Temperaturen **als Tabelle** mit 12 Zeilen und 31 Spalten. Dann lässt sich die Temperatur über die jeweilige Monatszeile und Tagesspalte ablesen.
- Gesucht: Geeignete Datenstruktur in C für T.

Motivation

Es gibt viele Problemstellungen und Anwendungen, deren Daten nicht geeignet mit den bisher verwendeten Datenstrukturen gespeichert werden können.

Beispiel 9.22 (Lösung linearer Gleichungssysteme)

Löse ein lineares Gleichungssystem $A \cdot x = b$ mit Vektoren $x = (x_1, \ldots, x_n)$ und $b = (b_1, \ldots, b_m)$ und einer **Matrix** $A = (a_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}$

- Gesucht: Geeignete Datenstruktur in C für A.
- **Matrixoperationen** werden dann **als Funktionen** in einer eigenen Übersetzungseinheit implementiert

Zweidimensionale Datenstrukturen in C

In C gibt es verschiedene zweidimensionale Datenstrukturen **mit statischer oder dynamischer Speicherreservierung**, mit denen sich Tabellen oder Matrizen speichern lassen. Deren Auswahl hängt von den Erfordernissen der Anwendung ab.

Wertzuweisungen an Zeiger auf ein Feld

Eine Adresse kann einem Zeiger zugewiesen werden (in einer Wertzuweisung oder Parameterübergabe), wenn ihr Typ dem Bezugstyp des Zeigers entspricht.

Einem Zeiger `T (*p)[N]` kann zugewiesen werden (Bezugstyp: `T[N]`):

- der Wert eines anderen Zeigers auf ein Feld gleicher Länge:
```
T (*q)[N];
p = q;
```
- eine Feldadresse eines 2-dimensionalen Feldes vom Typ `T[N]`:
```
T v[M][N];
p = v;
```

Folgerung

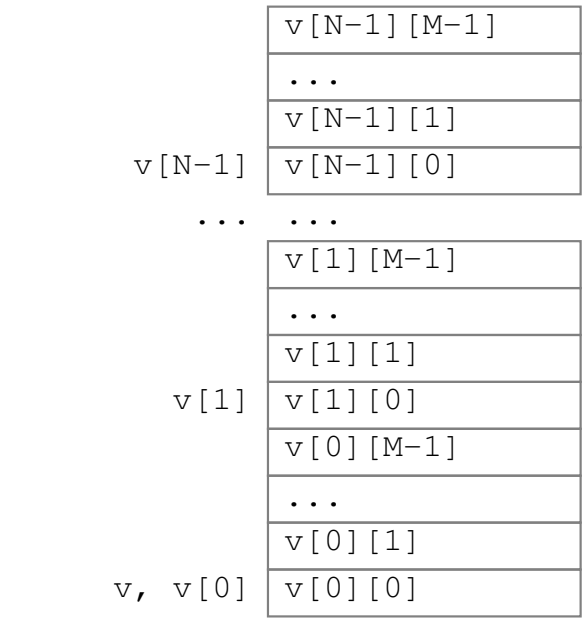
Die Eingabeparameter `T (*p)[N]` und `T v[][N]` sind gleichwertig. In beiden Fällen wird in der Funktion mit einem Zeiger auf ein Feld mit `N` Komponenten gerechnet.

Statische 2-dimensionale Felder

```
T v[N][M];
```

Hier wird ein Feld deklariert, dessen Komponenten wieder Felder sind (lies die Deklaration ausgehend von `v` **von innen nach außen**):

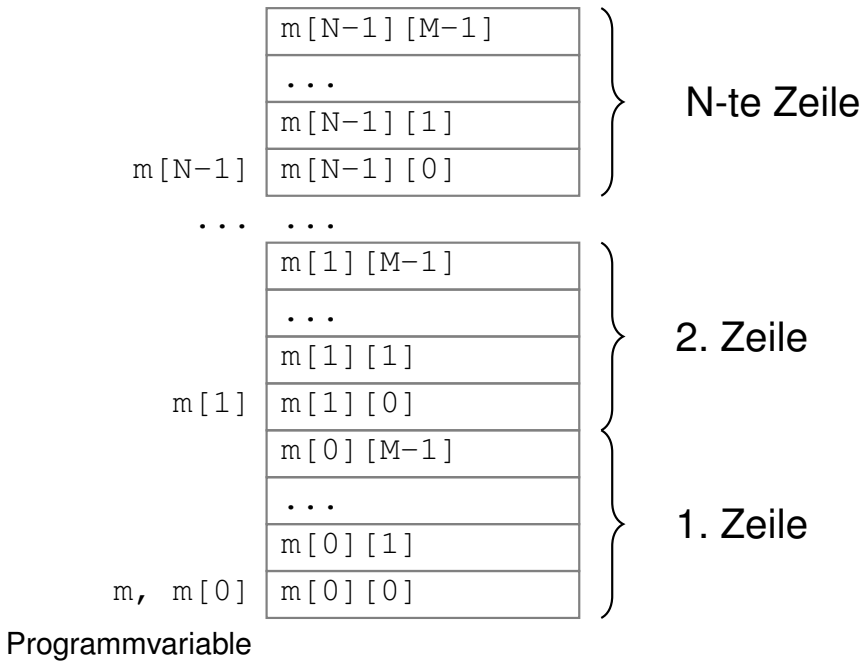
- `v` ist ein Feld mit `N` Komponenten. Der Compiler reserviert hierfür `N * M * sizeof(T)` Byte Speicherplatz.
- `v[i]` sind jeweils Felder mit `M` Komponenten ($0 \leq i \leq N-1$)
- `v[i][j]` sind Variablen vom Typ `T` ($0 \leq i \leq N-1, 0 \leq j \leq M-1$)



Anwendung 2-dimensionale Felder: Statische Matrizenrechnung

Repräsentiere eine Matrix durch ein zweidimensionales Feld

```
T m[N][M];
```



- `m[i]` ist die Adresse der `i+1`-ten Zeile
- `m[i][j]` ist der `j+1`-te Eintrag der `i+1`-ten Zeile
- Der Speicherbereich wird statisch reserviert und kann zur Laufzeit auch nur teilweise benutzt werden

Anwendung 2-dimensionale Felder: Statische Matrizenrechnung

Bei Übergabe **eines zweidimensionalen Feldes als Eingabeparameter** wird

- nur die zweite Dimension angegeben (wird für die Adressverschiebung benötigt: Wo beginnt die nächste Zeile?)
- Aber nicht die erste Dimension (denn nicht alle reservierten Zeilen müssen benutzt werden)

```
void matrix_init(int m[][MAX_COLUMNS], int ze, int sp) {
  int i,j;
  for(i = 0; i < ze; ++i) {
    for(j = 0; j < sp; ++j)
      m[i][j] = rand() % 1000;
  }
}
```

- Der **Zugriff auf die Matrixeinträge** einer Matrix `m` erfolgt über `m[i][j]` (in verschachtelten Schleifen mit zwei Zählvariablen)
- Beachte die analoge Implementierung im Funktionsrumpf wie bei dynamischen Matrizen

Anwendung 2-dimensionale Felder: Statische Matrizenrechnung

Ein zweidimensionales Feld muss vor der Benutzung **statisch angelegt** werden.

- Für die verwendeten Dimensionen verwendet man symbolische Konstanten (hier: `MAX_ROWS`, `MAX_COLUMNS`)
- Diese Dimensionen stellen Obergrenzen für die Benutzung zur Laufzeit dar

Beispielprogramm: Matrix statisch anlegen, mit Zufallszahlen initialisieren und ausgeben (die Matrix-Dimensionen können innerhalb der Obergrenzen `MAX_ROWS`, `MAX_COLUMNS` auch zur Laufzeit eingegeben werden)

```
int main() {
  srand(time(NULL));
  int matrix[MAX_ROWS][MAX_COLUMNS];
  matrix_init(matrix,5,7);
  matrix_print(matrix,5,7);
  return 0;
}
```

Dynamische 2-dimensionale Felder

2-dimensionale Felder auch mit dynamischer Speicherreservierung realisierbar:

1. Doppelzeiger für Zugriff
2. Speicher für erste Dimension (Zeilen) reservieren
   Als `T*`-Feld: Speichert jeweils Zeiger auf Feld für Zeile
3. Speicher für zweite Dimension (Spalten) reservieren
   - Als `T`-Feld: Speichert die Werte der Spalten in dieser Zeile
   - Bei Fehler: bisher reservierten Speicher freigeben
     $\rightarrow$ Alle bereits reservierten Zeilen + `T*`-Feld!
4. Im 2-dimensionalen Feld arbeiten
5. Speicher der zweiten Dimension freigeben
6. Speicher der ersten Dimension freigeben

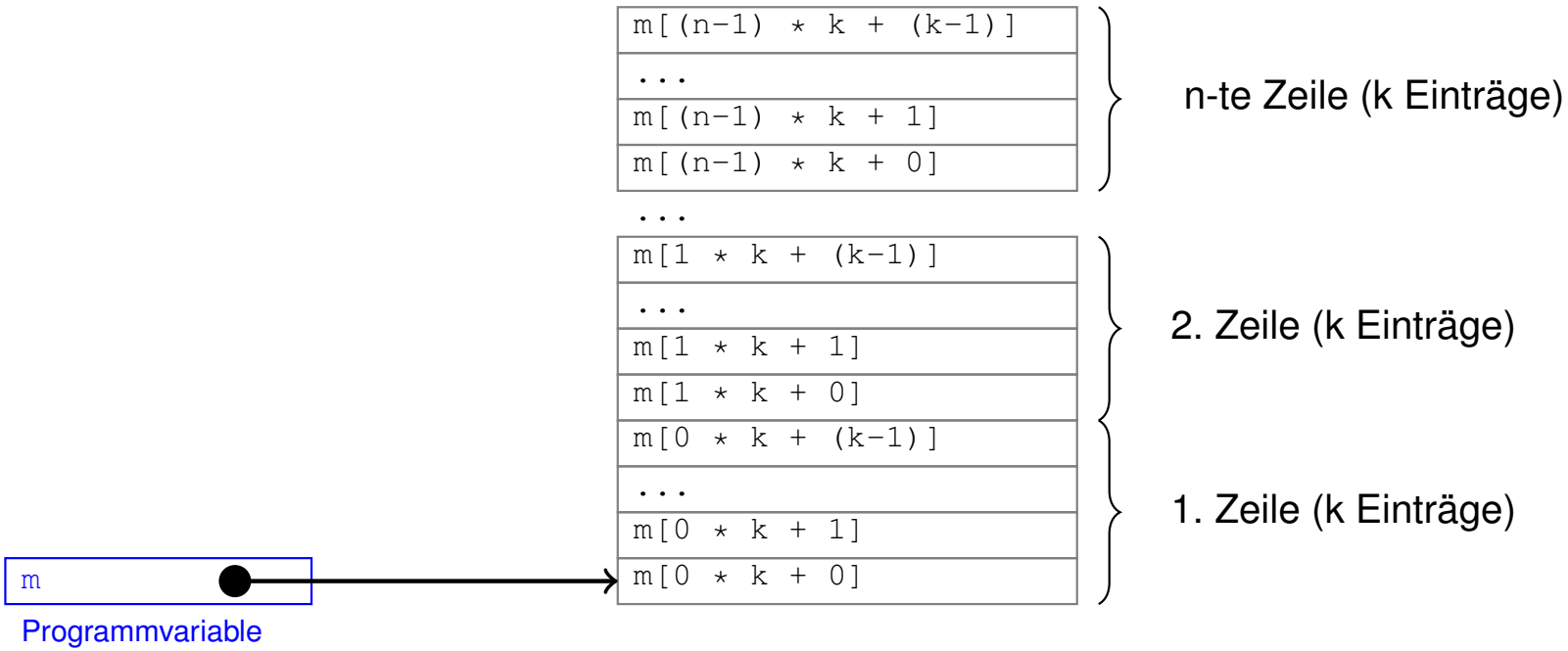
Dynamische 2-dimensionale Felder

```
1  int main(void) {
2      int i;
3      int **m;
4      m = malloc(ROWS * sizeof(int*));
5      if (m == NULL) {
6          printf("1. Fehlerfall");
7          return 1;
8      }
9      for (i = 0; i < ROWS; i++) {
10         m[i] = malloc(COLUMNS * sizeof(int));
11         if (m[i] == NULL) {
12             int j;
13             for (j = 0; j < i; j++) {
14                 free(m[j]);
15             }
16             free(m);
17             printf("2. Fehlerfall");
18             return 1;
19         }
20     }
21     /* Freigabe analog zum 2. Fehlerfall */
22 }
```

Dynamische Matrizenrechnung mit Einfach-Zeigern

Repräsentiere eine Matrix mit `n` Zeilen und `k` Spalten durch einen Einfach-Zeiger auf Speicherbereich mit `n * k * sizeof(T)` Byte

```
T *m = malloc(n * k * sizeof(T));
```

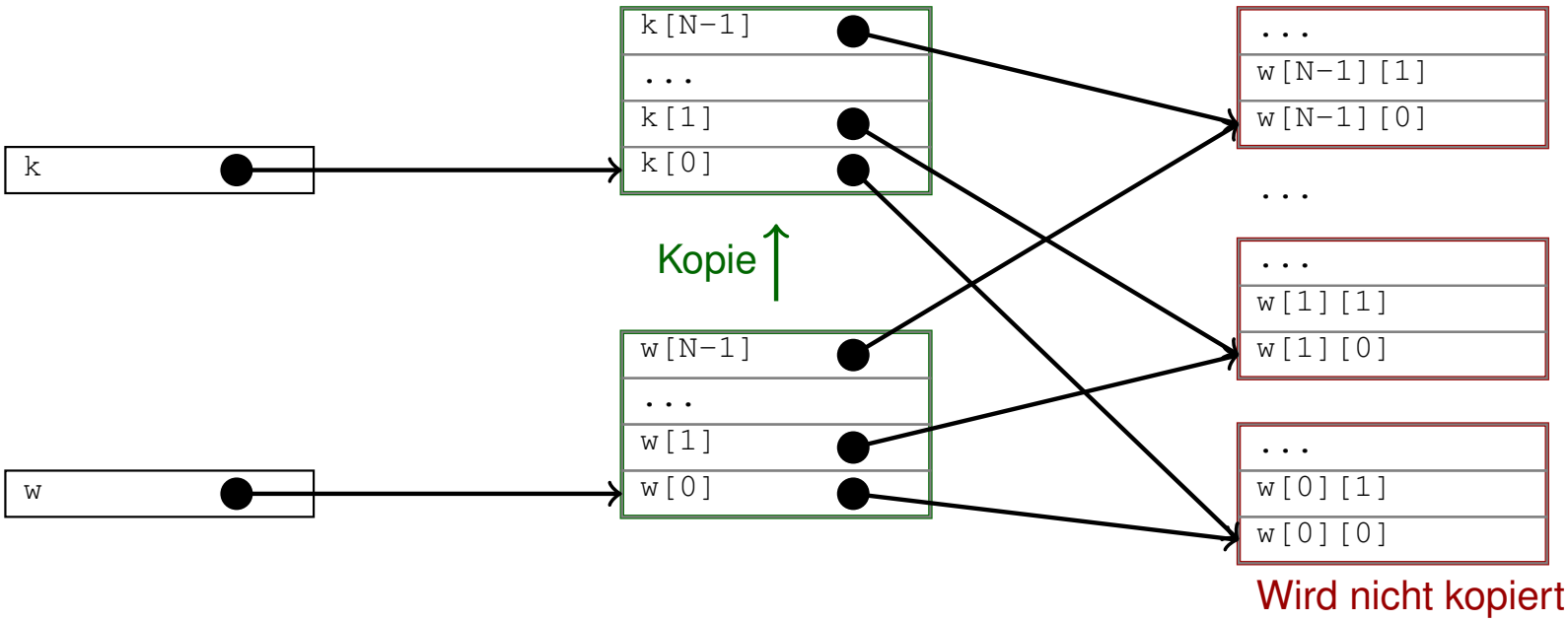


- `m[i * k + j]` ist der `j+1`-te Eintrag der `i+1`-ten Zeile
- Der Speicherbereich wird dynamisch reserviert

Flache Kopien

Bei einer **flachen Kopie** einer adresswertigen Variablen werden nur die Adressen kopiert, **aber nicht die Dateninhalte.**

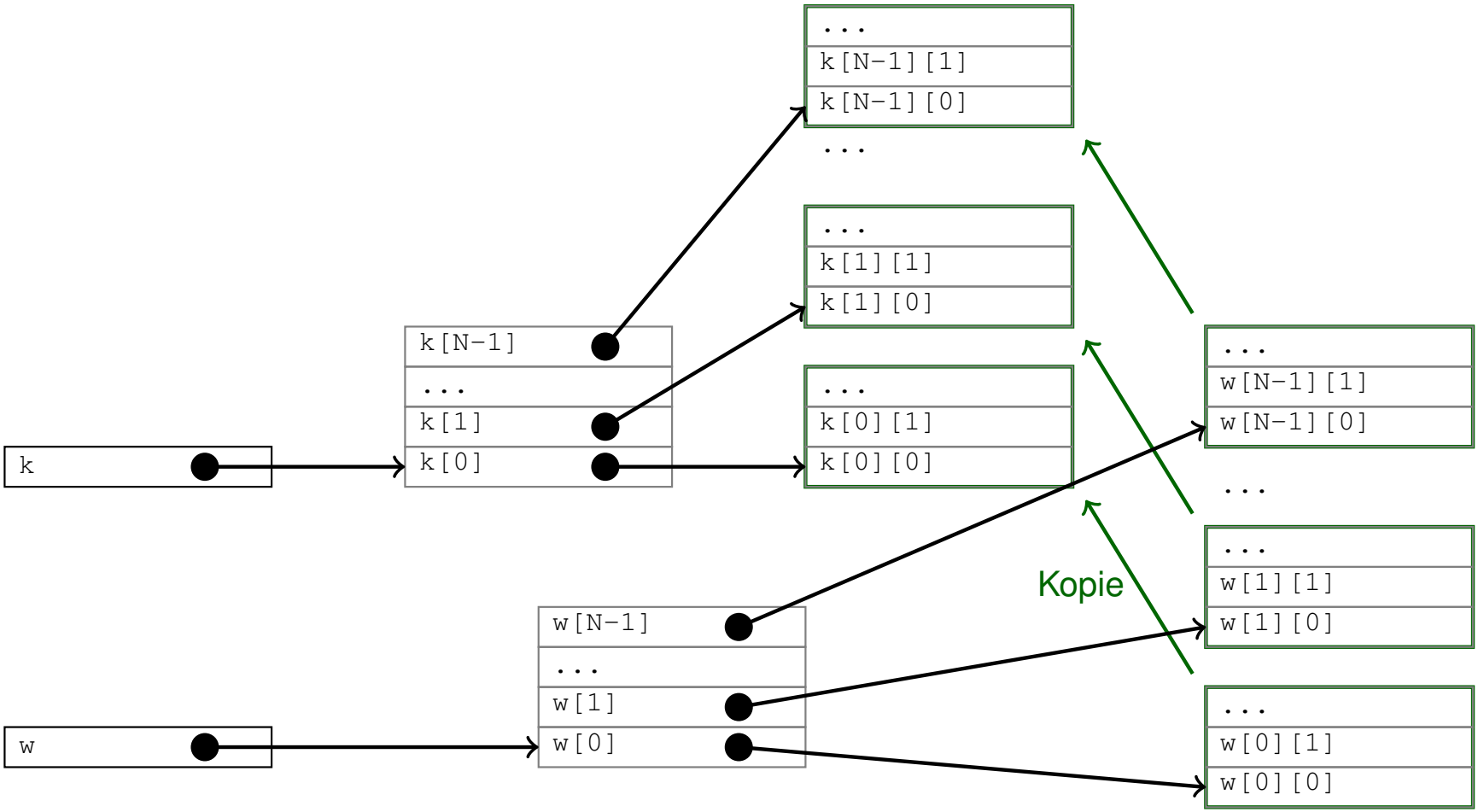
Beispiel: Flache Kopie `k` eines Feldes von Zeigern `w`



Konsequenz: Spart Speicher, aber die Kopie lässt sich nicht unabhängig vom Original verändern

Tiefe Kopien

Bei einer **tiefen Kopie** einer adresswertigen Variablen wird **die unterste Ebene der Dateninhalte** kopiert.



Notationen für Vektoren

Definition 9.23 (Vektor)

Sei X eine (Zahlen-)Menge und $n \in \mathbb{N}$. Ein Element $v \in X^n$ heißt ***n*-dimensionaler Vektor über** X.

v hat die Koeffizienten $v_1, \ldots, v_n$. Wir schreiben $v = (v_1, \ldots, v_n)$ oder $v = (v_i)_{1 \leq i \leq n}$

Operationen auf Vektoren

- Addition von $v, w \in X^n$: $v + w := (v_i + w_i)_{1 \leq i \leq n} \in X^n$
- Skalarmultiplikation von $v \in X^n$ mit $x \in X$: $x \cdot v := (x \cdot v_i)_{1 \leq i \leq n} \in X^n$
- Multiplikation von $v, w \in X^n$: $v \cdot w := \sum_{i=1}^{n} v_i \cdot w_i \in X$

Notationen für Matrizen

Definition 9.24 (Matrix)

Sei X eine (Zahlen-)Menge und $n, m \in \mathbb{N}$. Ein Element $A \in (X^n)^m$ heißt $(m \times n)$**-dimensionale Matrix über** X:

- Wir schreiben $A = (a_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n}$ oder

$$A = \begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \end{pmatrix}$$

  für die Koeffizienten von A.
- A hat m Zeilen $(a_{1,j})_{1 \leq j \leq n}, \dots, (a_{m,j})_{1 \leq j \leq n} \in A^n$
- A hat n Spalten $(a_{i,1})_{1 \leq i \leq m}, \dots, (a_{i,n})_{1 \leq i \leq m} \in A^m$

Notationen für Matrizen

Operationen für Matrizen

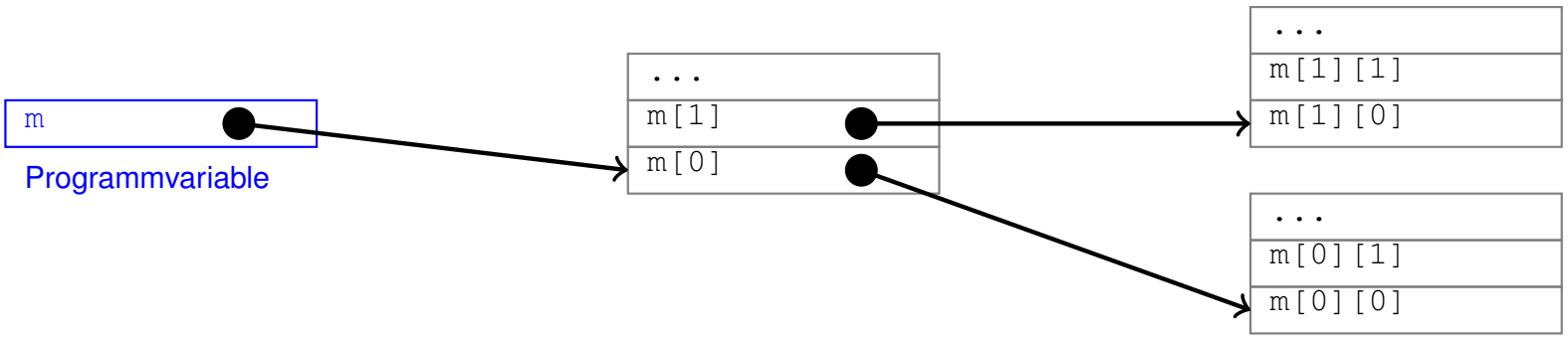
- Addition von $A, B \in (X^n)^m$:
  $A + B := (a_{i,j} + b_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n} \in (X^n)^m$
- Skalarmultiplikation von $A \in (X^n)^m$ mit $x \in X$:
  $x \cdot A := (x \cdot a_{i,j})_{1 \leq i \leq m, 1 \leq j \leq n} \in (X^n)^m$
- Multplikation von $A \in (X^n)^m$ mit $v \in X^n$:
  $A \cdot v := (\sum_{j=1}^n v_j \cdot a_{1,j}, \ldots, \sum_{j=1}^n v_j \cdot a_{m,j}) \in X^m$

$$A \cdot v = \begin{pmatrix} a_{1,1} & \ldots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \ldots & a_{m,n} \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^n v_j \cdot a_{1,j} \\ \vdots \\ \sum_{j=1}^n v_j \cdot a_{m,j} \end{pmatrix}$$

Anwendung Doppelzeiger: Dynamische Matrizenrechnung

Repräsentiere eine Matrix durch einen Doppelzeiger

```
T **m;
```



- `m[i]` ist ein Zeiger auf die `i+1`-te Zeile (in der Feld-Schreibweise)
- `m[i][j]` ist der `j+1`-te Eintrag der `i+1`-ten Zeile (in der Feld-Schreibweise)
- Die Speicherbereiche, auf die die Zeiger `m` und `m[i]` zeigen, werden dynamisch reserviert

Anwendung Doppelzeiger: Dynamische Matrizenrechnung

Repräsentiere eine Matrix durch einen Doppelzeiger

```
T **m;
```

Verwaltungsfunktionen:

- Speicherplatz reservieren für Matrix mit `ze` Zeilen und `sp` Spalten:
  ```
  int **matrix_create(int ze, int sp);
  ```
  Liefert Zeiger auf den reservierten Speicherplatz im Erfolgsfall und `NULL` sonst
- Für Matrix reservierten Speicherplatz wieder freigeben:
  ```
  void matrix_destroy(int **m, int ze);
  ```
  Gibt für die Zeiger `m` und `m[i]` reservierte Speicherbereiche wieder frei
- Matrixeinträge ausgeben:
  ```
  void matrix_print(int **m, int ze, int sp);
  ```
  Übergebe die Matrix als Doppelzeiger und die Dimensionen der Matrix
- Matrixeinträge mit Zufallszahlen initialisieren:
  ```
  void matrix_init(int **m, int ze, int sp);
  ```
  Übergebe die Matrix als Doppelzeiger und die Dimensionen der Matrix

Anwendung Doppelzeiger: Dynamische Matrizenrechnung

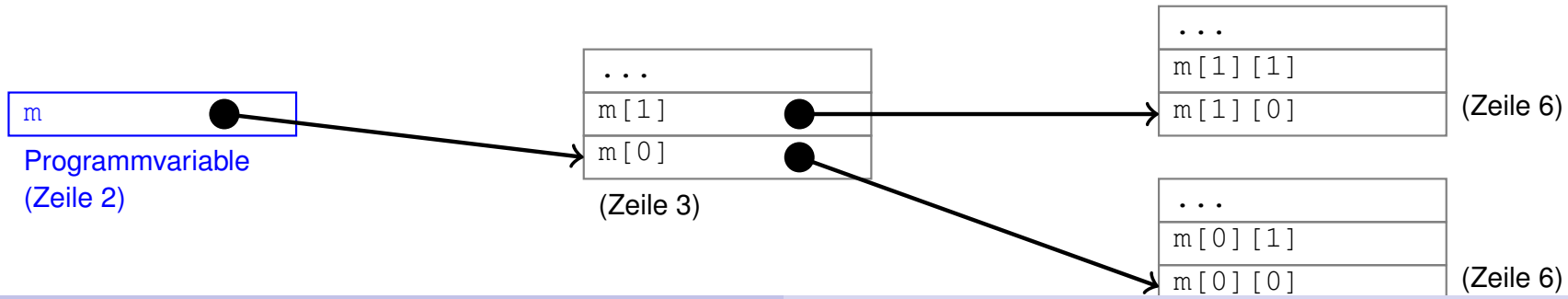
Dynamische Speicherreservierung für Matrix mit `ze` Zeilen und `sp` Spalten

```
1  int **matrix_create(int ze, int sp) { /* Speicher res. */
2    int **m, i, k;
3    m = malloc(ze * sizeof(int*));
4    if (!m) return NULL; /* Fehlerfall */
5    for (i = 0; i < ze; ++i) {
6      m[i] = malloc(sp * sizeof(int));
7      if (!m[i]) { /* Fehlerfall */
8        for (k = 0; k < i; ++k)
9          free(m[k]); /* Speicherlecks vermeiden */
10       free(m); /* Speicherlecks vermeiden */
11       return NULL;
12     }
13   }
14   return m; /* Erfolgsfall */
15 }
```

Schlägt eine der Speicherreservierungen fehl, muss vorher erfolgreich reservierter Speicher wieder freigegeben werden (Zeilen 7 - 10)

Anwendung Doppelzeiger: Dynamische Matrizenrechnung

```
1  int **matrix_create(int ze, int sp) { /* Speicher res. */
2    int **m, i, k;
3    m = malloc(ze * sizeof(int*));
4    if (!m) return NULL; /* Fehlerfall */
5    for (i = 0; i < ze; ++i) {
6      m[i] = malloc(sp * sizeof(int));
7      if (!m[i]) { /* Fehlerfall */
8        for (k = 0; k < i; ++k)
9          free(m[k]); /* Speicherlecks vermeiden */
10       free(m); /* Speicherlecks vermeiden */
11       return NULL;
12     }
13   }
14   return m; /* Erfolgsfall */
15 }
```



Anwendung Doppelzeiger: Dynamische Matrizenrechnung

Beispielprogramm: Matrix dynamisch anlegen, mit Zufallszahlen initialisieren und ausgeben (die Matrix-Dimensionen können auch zur Laufzeit eingegeben werden)

```
1  void matrix_init(int **m, int ze, int sp) {
2    int i,j;
3    for(i = 0; i < ze; ++i) {
4      for(j = 0; j < sp; ++j)
5        m[i][j] = rand() % 1000;
6    }
7  }
```

```
1  int main() {
2    srand(time(NULL));
3    int **matrix = matrix_create(8,10); /* Speicher res. */
4    if (!matrix) return 1; /* Fehlerfall */
5    matrix_init(matrix,8,10); /* Initialisieren */
6    matrix_print(matrix,8,10); /* Ausgeben */
7    matrix_destroy(matrix,8); /* Speicher freigeben */
8    return 0;
9  }
```

Anwendung Doppelzeiger: Dynamische Matrizenrechnung

Freigabe des Speicherbereichs:

```
1  void matrix_destroy(int **m, int ze) { /*Speicher freig.*/
2    int i;
3    for(i = 0; i < ze; ++i)
4      free(m[i]);
5    free(m);
6  }
```

- Vor der Freigabe des Speicherbereichs, auf den `m` zeigt (Zeile 5), müssen die Speicherbereiche, auf die die Zeiger `m[i]` zeigen (Zeile 4), freigegeben werden
- Sonst kann man auf diese nicht mehr zugreifen und es entstehen Speicherlecks