Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

# Informatik 1

## Kapitel 9 – Adressen und Zeiger

---

## Inhaltsverzeichnis

In diesem Kapitel beschäftigen wir uns damit, wie man mit Adressen arbeitet – man spricht von sogenannten *Zeigern*, da eine Adresse auf eine andere Speicherzelle „zeigt “.

Wir haben schon hier und da das *Call-by-Reference*-Prinzip angesprochen. Dabei wird einer Funktion nicht die Kopie einer Variable als Parameter übergeben, sondern die Adresse dieser Variable. Damit kann die Original-Variable verändert werden. Beispielhaft haben wir das bei Feldern oder der Funktion `scanf` gesehen. Hier haben wir eine Variable mit vorangestelltem & übergeben. Somit wurde die Adresse der Variable übergeben und `scanf` konnt einen eingelesenen Wert in der Variable speichern, obwohl diese außerhalb der Funktion scanf gespeichert ist.

Ein anderes Beispiel wäre ein Zahlen-Tausch Beispiel. Wir wollen eine Funktion schreiben, die die übergebenen Werte tauscht. Hierfür haben wir zwei Funktionen beispielhaft vorgegeben:

Beispiel:

```
/* Call-by-Value Implementierung */
void tausch1(int a, int b)
{
    int temp = a;

    a = b;
    b = temp;
}

/* Call-by-Reference Implementierung */
void tausch2(int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}

int main(void)
{
    int x = 21;
    int y = 9;

    /* Nichts passiert. Die Werte werden als
       Kopien übergeben und die Kopien werden
       getauscht. Die Originalwerte bleiben
       gleich. */
    tausch1(x, y);
    /* Es werden Adressen übergeben. Damit
       werden die originalen Werte getauscht. */
```

```
    tausch2(&x, &y);

    return 0;
}
```

Die obere Funktion arbeitet nicht mit Zeigern und ist somit „nutzlos“. Der Grund dafür ist, dass Kopien zweier Werte übergeben werden und diese nur lokal innerhalb der Funktion getauscht werden. Die originalen Werte bleiben unverändert. Auch können wir nicht mit Rückgabewerten arbeiten, da man in C nicht zwei Variablen gleichzeitig zurück geben kann.

Deswegen muss in diesem Beispiel mit Zeigern gearbeitet werden. Man übergibt die Adressen der Variablen und kann sie somit direkt verändern (in unserem Fall: tauschen).

Als Speicherzellen betrachtet würde es folgendermaßen aussehen:

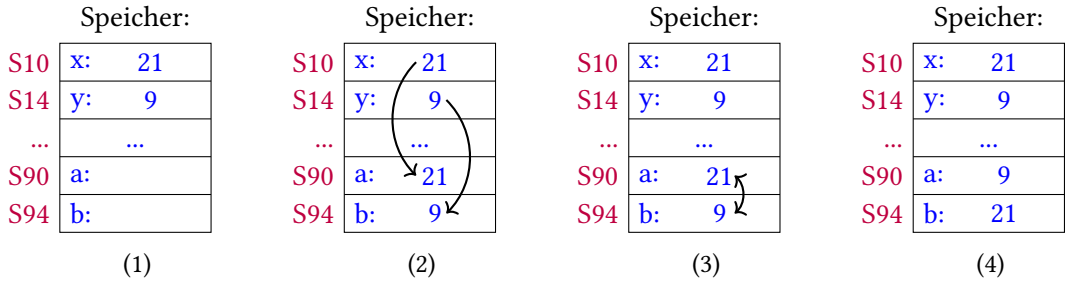



Abbildung 1: Speicherzellen Ansicht der Tauschfunktiom im Fall von Call-by-Value. Es werden keine Zeiger verwendet, der Tausch wird nur auf den lokalen Variablen `a` und `b` vollzogen.

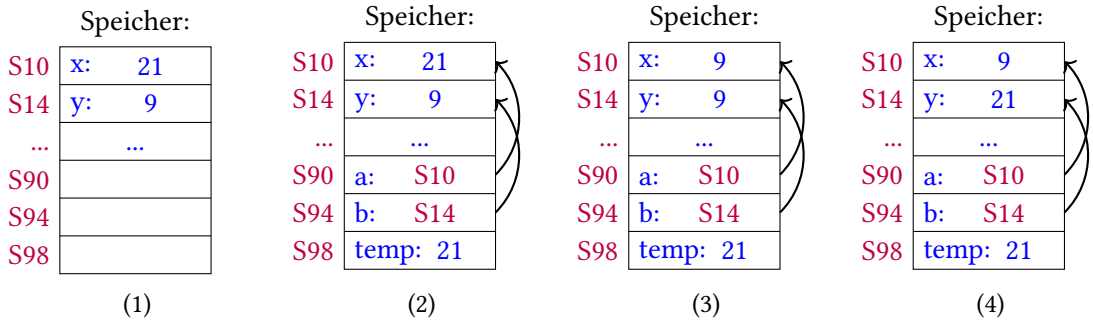



Abbildung 2: Speicherzellen Ansicht der Tauschfunktion im Fall von Call-by-Reference - es werden also Zeiger verwendet. Da wir direkt die Adressen der Variablen `a` und `b` ansprechen, können wir direkt den Inhalt in den jeweiligen Speicherzellen verändern.

Im Folgenden wollen wir nun Zeiger und Adressen formaler und genauer einführen und uns anschauen, wie man mit ihnen arbeitet und welche weiteren Möglichkeiten sie bieten.

## 9.1. Zeiger

### 9.1.1. Zeiger und ihre Deklaration

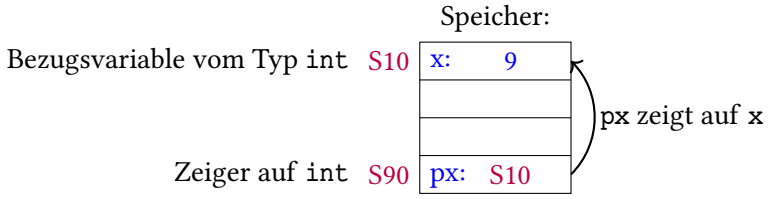
#### Was ist ein Zeiger?

Definition: 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.

Beispiel:




Ein Zeiger speichert eine Adresse. Der Speicherbedarf dafür ist implementierungsabhängig und kann von System zu System verschieden sein. Sie kann aber wie üblich mit `sizeof` abgefragt werden und ist unabhängig vom Typ der Bezugsvariable.

#### Deklaration von Zeigern

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

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

Beispiel:

`int *px, **ppx;`:

- px ist ein Zeiger auf `int` (**Einfachzeiger**)

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

**Coding Convention**
Es gibt mehrere Möglichkeiten, einen Zeiger zu deklarieren:

```
int *zeiger1;
int* zeiger1;
```

In der C Programmierung muss das Sternchen nur zwischen Datentyp und Variablenname sein. Wo genau ist jedoch dem Programmierer bzw. der Programmiererin überlassen.
Es hat sich aber folgende Konvention durchgesetzt[1]:

```
int *zeiger1;
int *zeiger2;
```

Das liegt daran, dass dadurch weniger Verwirrungen entstehen können, wie im folgenden Codestück zu sehen:

```
int* a, b;
```

Man könnte fälschlicherweise annehmen, dass zwei Zeiger deklariert wurden: `a` und `b`. Es ist jedoch nur `a` ein Zeiger und `b` ist eine `int`-Variable. Damit solche Unklarheiten nicht entstehen können, wird das Sternchen immer am Variablennamen angegeben:

```
int *a, b;
```

### 9.1.2. Adressen

Mit dem Adressoperator `&` kann man auf die Speicheradresse einer Variable zugreifen. Dabei gilt: 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**.

Beispiel:

Adressoperator `&`:
Direkter Zugriff auf eine Variable

- `&wert1` liefert die Adresse `S10`
- `&wert2` liefert die Adresse `S14`

Speicher:

| | |
|---|---|
| S10 | wert1: 11 |
| S14 | wert2: 7.2 |
| ... | ... |

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

[1] vgl. Linux Kernel Coding Style:
`https://www.kernel.org/doc/html/v4.10/process/coding-style.html#spaces`

Beispiel:

```c
int main(void)
{
    int a = 5;
    int *x = &a;

    /* Gibt die Adresse von a zurück. */
    printf("%p\n", &a);

    /* Gibt die Adresse von a zurück. */
    printf("%p\n", x);

    /* Gibt die Adresse von x zurück. */
    printf("%p\n", &x);

    return 0;
}
```

Beispiel:

```c
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

**Der Wert `NULL`:**
Wir wissen, dass wir auf lokale Variablen nicht lesend zugreifen dürfen, bevor sie initialisiert wurden. Wenn wir einen Zeiger deklarieren, ihm aber noch keine Adresse zuweisen wollen, verwenden wir für die Initialisierung den 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`

Es gibt auch die Möglichkeit, `NULL` in Bedingungen zu verwenden. Die symbolische Konstante wird dabei als 0 (falsch) gewertet.

Beispiel:

```
if (a == NULL) {
    /* Zeiger a zeigt nirgendwohin. Nur möglich,
        wenn vorher NULL an a zugewiesen wurde
       */
}
```

Da NULL in einer Bedingung wie 0 (falsch) gewertet wird, können wir auch schreiben:

```
if (p) {
    /* Wenn in p eine Adresse hinterlegt ist,
       wird dieser Fall ausgeführt. */
} else {
    /* Wenn p == NULL, wird dieser Fall ausgefü
       hrt. */
}
```

Es muss aber beachtet werden: Der Ausdruck `NULL` $\neq 0$

Beispiel:

Ein (anschauliches, gutes) Beispiel wäre anhand von Klopapier.

```
Klopapier = 0;
```

besagt, dass die Klopapier Rolle leer ist (also keine Blätter mehr vorhanden sind - nur noch die innere Papprolle)

```
Klopapier = NULL;
```

besagt, dass gar keine Rolle vorhanden ist. Weder Blätter noch die innere Papprolle existieren [a].

[a] vgl. `https://img.devrant.com/devrant/rant/r_2347874_pNrMx.jpg`

### 9.1.3. Lokale und globale Zeiger

**Lokale Zeiger:**
Ein **lokaler Zeiger** ist in einem Gültigkeitsbereich deklariert. Vor der Wertzuweisung sagt man, der Zeiger **zeigt irgendwohin** (zufälliger Wert). Er wird wie alle lokalen Variablen im Stack ab-

gelegt.

**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**. Er wird wie alle globalen Variablen im Datenteil abgelegt.

Beispiel:

Lokale und globale Zeiger lassen sich analog zu lokalen und globalen Variablen deklarieren und nutzen:

```
/* Ein globaler Zeiger. Es kann überall in
   dieser C-Datei verwendet werden. */
int *global_p;

int main(void)
{
    /* Ein lokaler Zeiger. Es kann nur innerhalb
        der main-Funktion verwendet werden. */
    int *lokal_p;

    return 0;
}
```

### 9.1.4. 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**)

Die Wertzuweisung funktioniert also bei Zeigern genauso wie bei anderen Variablen auch. C achtet jedoch sehr genau darauf, worauf ein Zeiger zeigt und akzeptiert die Zuweisungen nur, wenn der Typ exakt übereinstimmt. Demnach kann ein Einfach-Zeiger auf einen `int` z.B. keine Adresse einer `double`-Variable speichern. Ebenso kann ein Doppelzeiger nur die Adresse eines Zeigers des entsprechenden Typs speichern.

Beispiel: 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:

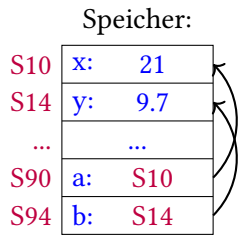
- Sei `e` ein adresswertiger Ausdruck vom Typ `T*` ungleich `NULL`.
  An der Adresse `e` sei der Wert `x` (von Typ `T`) gespeichert. Dann ist
  `*e`
  ein **Ausdruck vom Typ `T` mit Wert `x`**.
- Sei `p` ein Zeiger auf eine Variable `x` vom Typ `T`. Dann ist
  `*p`
  eine **Variable vom Typ `T`** und 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.

Beispiel:

Dereferenzierungsoperator `*`:
Zugriff auf den Wert an einer Adresse

- `*a` liefert 21
- `*b` liefert 9.7




Beispiel:

- `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 */`

Man beachte: Im Zusammenhang mit Zeigern hat der Stern `*` zwei Bedeutungen:

1. Bei der **Deklaration** zeigt er an, dass ein Zeiger deklariert wird.
2. Innerhalb von Funktionen steht er für die Dereferenzierung.

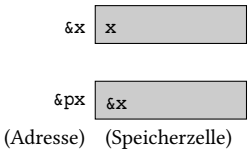
Machen Sie sich beide Bedeutungen bewusst und achten Sie darauf, wo was gemeint ist.

### Notation

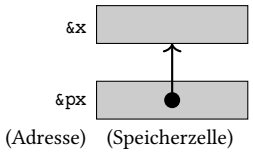
Darstellung von Zeigern im Speicher:

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

Darstellung im Speicher mit Adressen:




Kurz-Darstellung:




### Der Modifikator `const`

Wie bei den Datentypen, die wir bisher kennengelernt haben, können wir mit dem Modifikator `const` auch Zeiger zu Konstanten machen. Man muss jedoch aufpassen, an welche Stelle man `const` schreibt, da man je nach Position den Zeiger oder den Wert, auf den er zeigt, konstant macht.

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;
```

Beispiel:

Im Folgenden wird ein kleines Programm zum Ausprobieren vorgestellt. Damit es kompiliert, müssen die nicht funktionierenden Codestellen auskommentiert werden.

```
#include <stdio.h>

int main()
{
    int x1 = 1;
    int x2 = 2;
    int x3 = 3;
    int bieger = 50;

    /* Konstanter Zeiger, zeigt immer auf x1 */
    int * const p1 = &x1;
```

```
/* Zeiger auf konstanten Wert */
const int *p2 = &x2;

/* Zeiger und Wert sind unveränderbar */
const int * const p3 = &x3;

/* Gibt 1, 2 und 3 aus. Noch sollte alles
   bekannt sein */
printf("p1: %i, p2: %i, p3: %i\n", *p1, *p2,
    *p3);

/* Funktioniert NICHT, da p1 ein konstanter
   Zeiger ist. Es kann deswegen nicht
   umgebogen werden. */
p1 = &bieger;
/* Funktioniert, da der Zeiger p2 umgebogen
   werden kann. Nur der Wert der
   Speicherzelle kann nicht verändert werden
   . */
p2 = &bieger;
/* Kann als Kombination natürlich auch nicht
    umgebogen werden. */
p3 = &bieger;
printf("p1: %i, p2: %i, p3: %i\n", *p1, *p2,
    *p3);

/* Funktioniert, da p1 ein konstanter Zeiger
    ist. Es kannn deswegen nicht umgebogen
   werden, der Wert kann jedoch verändert
   werden. */
*p1 = 99;
/* Funktioniert NICHT, da nur der Zeiger p2
   umgebogen werden kann. Der Wert der
   Speicherzelle kann nicht verändert werden
   . */
*p2 = 99;
/* Kann als Kombination natürlich auch nicht
    verändert werden. */
*p3 = 99;
printf("p1: %i, p2: %i, p3: %i\n", *p1, *p2,
    *p3);

return 0;
```

```
}
```

### 9.1.5. 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 */
   ```
   Feldvariablen sind immer konstante Zeiger!

Beispiel:

Im Folgenden werden viele kleinere Beispiele zum Ausprobieren und Herumspielen vorgegeben und erklärt. Es wird auch auf die richtige Benutzung hingewiesen.

```
#include <stdio.h>

void bsp1(void)
{
    /* Dieses Beispiel wird mit allen 4
       Compilerschaltungen nicht kompilieren.
       Der Zeiger wurde zwar richtig erstellt,
       jedoch nicht initialisiert. Der Zeiger
       existiert, jedoch existiert noch keine
       Speicherzelle, um Werte zu speichern */

    int *p;
    *p = 5;
```

```
    printf("%i", *p);
}

void bsp2(void)
{
    /* Das Problem kann einfach behoben werden,
       indem eine Variable des gleichen
       Datentyps erstellt wird. Man weist nun
       die Adresse der Variable dem int-Zeiger
       zu und kann dann den Wert setzen. */

    int var;
    int *p;
    p = &var;

    /*
    Alternativ:
    int *p = &var
    */

    *p = 5;

    printf("%i", *p);
}

void bsp3(void)
{
    int *p;
    int x;

    p = &x;

    *p = 5;

    /* Dadurch, dass p auf die Speicherstelle
       von x zeigt, besitzen beide nun auch die
       selben Werte. Es wird "5, 5" ausgegeben.
       */
    printf("%i, %i", *p, x);
}

void bsp4(void)
{
    /* Das Beispiel 3 wird hier erweitert. Zwei
```

```
        Zeiger zeigen auf die selbe
        Speicherstelle. */
    int x = 5
    int *p1 = &x;
    int *p2 = &x;

    /* Beide geben 5 aus. */
    printf("p1: %i, p2: %i", *p1, *p2);

    /* Der Wert, auf den Zeiger p1 zeigt, wird
        verändert */
    *p1 = 10;

    /* Beide Zeiger zeigen auf die selbe
        Speicherzelle. Da die Speicherzelle, auf
        die p1 zeigt, auf den Wert 10 gesetzt
        wurde, wirkt sich das natürlich auch auf
        p2 aus. */
    printf("p1: %i, p2: %i", *p1, *p2);
}

void bsp5(void)
{
    /* Zeiger müssen nicht nur einzelne Werte
        speichern können. Sie können auch auf
        Felder selbigen Datentypes zeigen. */

    int v[5] = {5, 6, 7, 8, 9};
    /* BEACHTE: Um v zu übergeben darf kein
        Adressoperator & angegeben werden. Bei
        Feldern bezeichnet bereits der Feldname (
        hier: v) die Adresse. */
    int *p = v;

    /* Gibt 5 aus. */
    printf("%i", v[0]);

    /* Das ist nun ein interessanter Fall. Der
        Zeiger p zeigt momentan auf das erste
        Element vom Feld v. Somit wird hier 5
        ausgegeben.
    Im Kapitel 9.3 wird auf die
        Adressverschiebung eingegangen. Damit
        kann man durch p auch die anderen Stellen
```

```
            von v ansteuern. */
        printf("%i", *p);
    }
```

## 9.2. Call by Reference

### Was ist das Call-by-Reference-Prinzip?

Definition: 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:

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. die original Zeiger können in einer Funktion **nicht umgebogen werden** (nur die Kopien).

### Beispiel für das Call-by-Reference-Prinzip

Im Foliensatz ab Folie 16 befindet sich ein anschauliches Beispiel zum Call-by-Reference Prinzip. Es wurde nicht in das Skript übernommen, da das Beispiel nur im Foliensatz funktioniert und die Veranschaulichung kaputt gehen würde.

### Call by Reference in einer Eingabefunktion

In Kapitel 7 hatten wir eine Funktion `read_pos` entwickelt, die eine positive Zahl einliest und im Fehlerfall eine negative Zahl zurück gibt. Da die negativen Zahlen dort für Fehlerfälle reserviert waren, ist eine Erweiterung zum Einlesen sowohl positiver als auch negativer Zahlen problematisch (ist bspw. die Zahl -1 nun eine Eingabe oder ein Fehlerwert?). Mit Hilfe von Zeigern können wir die Funktion jedoch so umschreiben, dass eingegebene Zahl und Fehlerstatus getrennt zurück gegeben werden: Die Zahl in einer Variablen, deren Zeiger übergeben wurde und der Fehlerstatus als Rückgabe der Eingabefunktion:

```
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   }
```

- 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 6 und 8):
  *Da der eingelesene Wert hier nicht zurückgegeben wird, hat man keine Probleme mehr mit dem Festlegen von Rückgabewerten für Fehlerfälle!*

#### Call by Reference im Hauptprogramm

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

```
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 }
```

- Lege eine Variable `n` 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)

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

## 9.3. Adressverschiebung

Bei der *Adressverschiebung* hat man eine Adresse gegeben, aber greift auf eine der darauf folgenden Speicherzellen zu. Dieses Prinzip kennen wir bereits von Feldern. Dort haben wir z.B. ein Feld `v` gegeben. In `v` ist die Adresse der ersten Speicherzelle des Feldes hinterlegt. Mit `v[2]` greifen wir auf den Wert zu, der an zweiter Stelle hinter dem Wert `v[0]` steht.

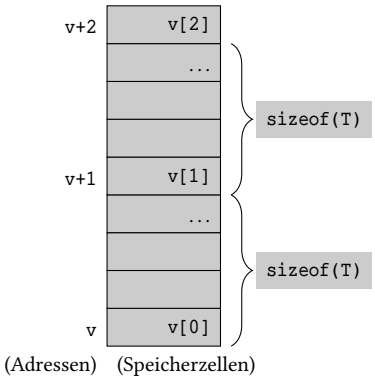
#### Adressverschiebung anhand von Feldern

Rechts sehen wir, wie ein Feld im Speicher abgelegt ist: Es beginnt ganz unten bei der Adresse `v`, wo der Eintrag `v[0]` gespeichert ist. Jeder Eintrag braucht eine bestimmte Anzahl an Speicherzellen – die genaue Anzahl lässt sich mit `sizeof` bestimmen. Beispielsweise ergibt `sizeof(int)` auf bestimmten Systemen 4. Zur besseren Übersicht beschriften wir nur jene Speicherzelle, wo ein

Datenwert beginnt. Bei `int` ist also nach 4 Speicherzellen die nächste Zahl `v[1]` hinterlegt, nach weiteren 4 `v[2]` usw.

**Definition:**

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`.




Wenn wir also ein Feld definieren (z.B. `int v[5]`), dann ist an der Adresse `v` der erste Wert des Feldes hinterlegt. Wir wissen nun, dass `v` ein konstanter Zeiger ist (z.B. ein `int*`). Demnach können wir via `*v` auf den Eintrag `v[0]` zugreifen.

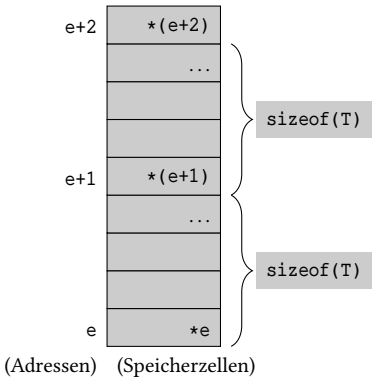
Damit man über die Adresse nicht nur auf den ersten Eintrag `v[0]` zugreifen kann, sondern auch auf alle weiteren, können wir zur Adresse die Nummer des gewünschten Eintrags addieren: Mit `v+1` bezeichen wir die Adresse des Eintrags `v[1]`, mit `v+2` die des Eintrags `v[2]` usw. Allgemein ist somit `v+n` identisch zu `&v[n]`. Wenn wir auf einen Wert zugreifen wollen, können wir diese Adress-Notation mit dem Dereferenzierungsoperator `*` verbinden: Via `*(v+1)` erhalten wir den Wert, der in `v[1]` gespeichert ist, mit `*(v+2)` jenen an `v[2]` usw. Allgemein entspricht also `*(v+n)` der Schreibweise `v[n]`.

Statt der Feldvariable `v` können wir auch einen Zeiger verwenden, der auf das Feld zeigt. Dieser hat den Vorteil, dass er nicht konstant ist (wie die Feldvariable `v`), sondern verändert werden kann. Damit lassen sich Adressen weiterzählen, wie wir im Beispiel unten sehen werden.

**Definition: 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)




So wie wir oben mit der Feldvariable `v` die Notation `v+n` verwendet haben, können wir auch einen adresswertigen Ausdruck `e` mit Adresse `a` verwenden und genauso `e+n` bzw. `*(e+n)` schreiben, um auf die Speicherzelle `e[n]` zuzugreifen.

Hierbei ist zu beachten, dass man `n` stets so wählt, dass man nur reservierten Speicher zugreift!

Beispiel:

```c
#include <stdio.h>

int main(void)
{
    int v[5] = {5, 6, 7, 8, 9};
    int *p = v;

    /* Für die Adressverschiebung gibt es nun
       mehrere Möglichkeiten, wie man herangeht.
        */
    /* Zum einen kann man den Zeiger p wie ein
       Feld verwenden. Ausgabe: 9 */
    printf("%i", p[4]);

    /* Alternativ dazu kann man auch folgendes
       schreiben: */
    printf("%i", *(p + 4));

    /* ABER: Man muss aufpassen, dass man nicht
       in nicht-reservierten Bereich landet (wie
        in diesem Fall). Sonst wird irgendwas
       zurück gegeben oder das Programm stürzt
       ab.*/
    printf("%i", *(p + 10));

    /* Das funktioniert auch mit Adressen. Im
       folgenden Beispiel wird die Adresse des
       Feldes v zurück gegeben: */
    printf("%p", p);

    /* Und nun die Adresse sizeof(int)-
       Speicherzellen weiter. */
    printf("%p", &p[1])

    /* Alternativ kann man auch schreiben: */
    printf("%p", p+1);

    /* Auch hier kann man in nicht-reservierten
       Bereich landen: */
    printf("%p", p+10);

    return 0;
}
```

### 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.

> Beispiel:
>
> Wir hatten im Kapitel 6 die Funktion `strcmp` kennengelernt. Diese ist im C89-Standard wie folgt definiert[a]:
> `int strcmp(const char *s1, const char *s2);`
>
> Wir hatten im Kapitel 6 immer Felder übergeben, z.B. `strcmp(v, w)`, woei `v` und `w` als `char v[5]` u.ä. deklariert waren. Nun wissen wir, dass die Funktion alternativ wie folgt definiert sein könnte:
> `int strcmp(const char s1[], const char s2[]);`
>
> Da `char *s1` und `char s1[]` als Eingabeparameter völlig gleichbedeutend sind, konnten wir im Kapitel 6 darauf verzichten, Zeiger ausführlich zu erklären, sondern einfach direkt Felder übergeben.
>
> [a] siehe `http://port70.net/~nsz/c/c89/c89-draft.html#4.11.4.2`

### Anwendung: Mit Adressverschiebung ein Feld durchlaufen

Im Foliensatz ab Folie 26 befindet sich ein anschauliches Beispiel zur Adressverschiebung beim Durchlaufen eines Feldes. Es wurde nicht in das Skript übernommen, da das Beispiel nur im Foliensatz funktioniert und die Veranschaulichung kaputt gehen würde.

## Kleines Verständnisbeispiel

> Beispiel:
>
> Kleine Aufgabe für das große Verständnis:
> Was steht in welcher Speicherzelle?
> Das Ergebnis gibt es am Ende des Skriptes, im Anhang auf Seite 55.

```
int a = 5;      // Im Speicher an Adresse S20
int *b = &a;    // Im Speicher an Adresse S24
int *c = b;     // Im Speicher an Adresse S28
```

```
*b = 6;
c++;
int *d;          // Im Speicher an Adresse S2C
```

Speicher:

| | |
|---|---|
| S20 | a: |
| S24 | b: |
| S28 | c: |
| S2C | d: |

## 9.4. Zeichenketten als Zeiger

### Zeiger auf `char` sind Zeichenketten

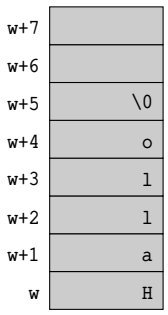
**Definition: Zeichenkette**

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:**

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

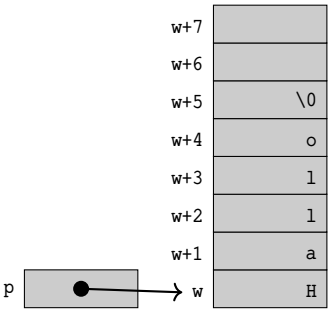
- `char *p = w;`:
  p repräsentiert die Zeichenkette `"Hallo"`
- `char *p = w + 2;`:
  p repräsentiert die Zeichenkette `llo"`




### 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, da keine Längenüberprüfung stattfindet und man dadurch auf einen nicht reservierten Speicherbereich zugreifen könnte!)

## 9.5. Zeiger und Funktionen

Es sollte schon aus oberen Beispielen bekannt sein, dass man Zeiger auch Funktionen übergeben kann. Alternativ ist es auch möglich, Zeiger als Rückgabeparameter zu nutzen.

**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.

Beispiel: Beispiele aus `string.h`

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

Beispiel:

```
void print_me(int *p1, int len)
{
    int i;
    for (i = 0; i < len; i++) {
        printf("%i\n", *(p1 + i));
    }
}
```

Alternativ könnte der Funktionskopf auch folgendermaßen aussehen:

```
void print_me(int p1[], int len)
```

### Zeiger als Rückgabetyp einer Funktion

Ist `T` ein 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:

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

- `char *strncpy(char *s1, const char *s2, size_t n);`
- `char *strncat(char *s1, const char *s2, size_t n);`
- `char *strchr(const char *s, int c);`
- `char *strstr(const char *s1, const char *s2);`
- `char *strtok(char *s1, const char *s2);`

Beispiel:

Die beiden Funktionen `getRandom1` und `getRandom2` zeigen zwei verschiedene Möglichkeiten, einen Zeiger zurück zu geben:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* Beachte: Das Sternchen neben dem
   Funktionsnamen. Es zeigt an, dass ein Zeiger
   zurück gegeben wird. */
int *getRandom1(void)
{
    /* Es wird, wie schon von oben bekannt sein
       sollte, ein Zeiger erzeugt und deklariert
       . */
    static int n;
    static int *p = &n;

    n = rand();

    /* Da p schon ein Zeiger ist, reicht es, nur
        p zurück zu geben. Es muss NICHT mit dem
```

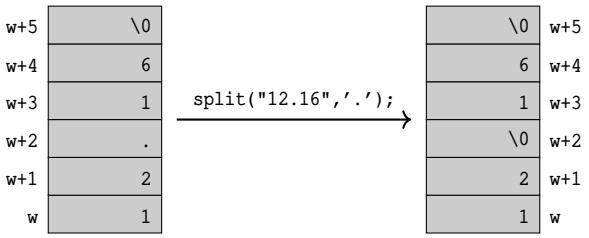
```
        &-Operator gearbeitet werden. */
    return p;
}

int *getRandom2(void)
{
    /*
    Eine andere Möglichkeit wäre die Rückgabe
       einer Adresse. Es wird für diesen Fall
       kein Zeiger erzeugt, sondern eine
       statische Variable erzeugt. Statisch,
       damit die Variable nicht beim Verlassen
       der Funktion wieder gelöscht wird.
    */
    static int n;

    n = rand();

    /* Die Variable n ist kein Zeiger. Es kann
       dennoch dessen Adresse mithilfe des &-
       Operators zurück gegeben werden. */
    return &n;
}


int main(void)
{
    int *p1;
    int *p2;

    srand(time(NULL));

    p1 = getRandom1();
    p2 = getRandom2();

    printf("%i, %i", *p1, *p2);

    return 0;
}
```

**Achtung:** Zeiger auf lokale Variablen dürfen **nur** zurück gegeben werden, wenn diese statisch sind. „Normale“ lokale Variablen werden beim Verlassen der Funktion freigegeben und Zeiger darauf verlieren ihre Gültigkeit (bzw. zeigen auf Speicher, dessen Inhalt nicht mehr definiert ist).

### Beispiel: 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;
}
```




**Achtung:**
Niemals eine Adresse eines freigegebenen Speicherbereichs zurückgeben!

### Beispiel: Murks mit Rückgabeadressen

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

Durch die Freigabe ist der Speicher nicht mehr reserviert und man würde auf einen undefinierten Bereich zugreifen. Es kann sein, dass der Inhalt des Speichers noch existiert. Greift man darauf zu, kann es einem so vorkommen, als ob „alles okay sei“. Es kann aber auch passieren, dass die freigegebenen Speicherzellen überschrieben wurden oder noch überschrieben werden.

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

```
}

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

## 9.6. Dynamische Speicherverwaltung

### 9.6.1. 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. Dadurch ist man jedoch auf die gewählte maximale Anzahl der Komponenten beschränkt. Auch tritt das Problem auf, dass Speicherplatz verschwendet wird, sollte man weniger Komponenten benötigen.

C bietet uns auch die Möglichkeit, Speicher dynamisch (d.h. zur Laufzeit des Programms) zu reservieren. Das schauen wir uns im Folgenden genauer an.

Beispiel:

**Beispiel: Dynamisches Einlesen einer Zahlenfolge**
Das folgende Beispiel greift den kompletten Inhalt des Kapitels vor. Es werden die `malloc` und `free` Funktionen verwendet. `malloc` ist für die dynamische Reservierung von Speicher zuständig, `free` ist für das Freigeben von Speicher zuständig. Sie werden im Laufe des Kapitels noch genauer erklärt.

- Zeile 20: Reservierung des erforderlichen Speicherbereichs über einen Zeiger
- Zeilen 23 - 26: Fehlerbehandlung
- Zeilen 29 - 37: i-te Zahl einlesen mit Fehlerüberprüfung
- Zeile 34: Speicherfreigabe im Fehlerfall
- Zeile 47: Speicherfreigabe, wenn Speicher nicht mehr benötigt wird

```
1 /*Zahlenfolgen einlesen mit dynamischer
     Speicherverwaltung*/
2 #include <stdio.h>
3 #include <stdlib.h>
4
```

```c
#include "inputp.h"

int main(void)
{
    int size, i;
    int * sequence;

    /*Eingabe der Länge der Zahlenfolge*/
    printf("Wie viele Zahlen willst du eingeben?\n");
    if (read_pos_p(&size) == INVALID_INPUT) {
        printf("Eingabe ungueltig\n");
        return EXIT_FAILURE;
    }

    /*Speicherplatz dynamisch reservieren*/
    sequence = malloc(size * sizeof(int));

    /*Fehlerbehandlung*/
    if (sequence == NULL) {
        printf("Speicherfehler\n");
        return EXIT_FAILURE;
    }

    /*Zahlen einlesen*/
    for (i = 0; i < size; ++i) {
        printf("Eingabe %i-te Zahl: ", i + 1);
        /*Fehlerbehandlung mit Speicherfreigabe*/
        if (read_pos_p(&sequence[i]) == INVALID_INPUT) {
            printf("%i-te Eingabe ungueltig\n", i + 1);
            free(sequence);
            return EXIT_FAILURE;
        }
    }

    /*Ausgabe*/
    printf("Eingabe:\n");
    for (i = 0; i < size; ++i) {
        printf("%i ", sequence[i]);
    }
    printf("\n");
```

```
45
46      /*Speicherfreigabe*/
47      free(sequence);
48
49      return EXIT_SUCCESS;
50  }
```

### 9.6.2. Dynamische Reservierung von Speicher – 1. Möglichkeit

**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 (z.B. weil nicht genügend Speicher zur Verfügung steht), und gibt die Adresse (nicht getypt) der ersten Speicherzelle des reservierten Bereichs im Erfolgsfall zurück.

Am Ende wird der Speicherplatz **nicht automatisch** wieder freigegeben, dies muss durch einen separaten Funktionsaufruf zu `free` erfolgen. Der Typ `size_t` wird verwendet, da es sich um eine Größenangabe handelt[2]. Er ist außerdem Rückgabetyp des `sizeof`-Operators.

**Systemunabhängige Festlegung der Speichergröße mit `sizeof`:**
Es ist recht unpraktisch, jedes mal die Anzahl der Bytes anzugeben. Dazu müsste man sich merken (oder nachschauen), wie viele Bytes jeder Datentyp auf welchem System besitzt. Mithilfe des `sizeof`-Operators lässt sich dieses Problem umgehen.

Möchte man genug Speicherplatz für *einen* Wert des Datentypes `T` reservieren, kann man Folgendes schreiben:

```
malloc(sizeof(T));
```

Möchte man stattdessen Speicherplatz für *mehrere* Werte reservieren (bspw. um eine Liste zu realisieren), kann man den obigen Code ganz einfach erweitern:

```
malloc(n * sizeof(T));
```

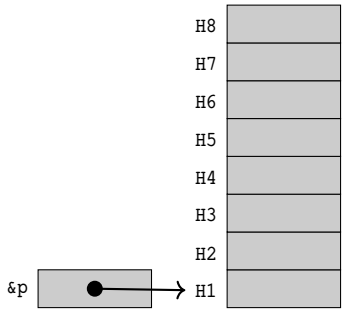
Beispiel: Dynamische Speicherreservierung im Speicher

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

- Es werden 8 Byte Speicherplatz im Heap reserviert (da ein int 4 Byte groß ist)
- p zeigt auf die erste Speicherzelle – also das 0. Element der Liste
- Der Wert von p ist die Adresse `H1`

[2] siehe Kapitel 6.2

- `&p` und `H1` sind unterschiedliche Adressen (`p` ist ein Zeiger und wird im Stack gespeichert, H1 wurde dynamisch im Heap reserviert. `p` zeigt auf H1)
- `p[0]` ist eine `int`-Variable an Adresse `H1`
- `p[1]` ist eine `int`-Variable an Adresse `H5`




Beispiel:

```c
int main(void)
{
    /* Diese Art ein Feld zu erstellen sollte
       schon bekannt sein: */
    int a[10];

    /* Stattdessen können wir nun folgendes tun:
        */
    int *a = malloc(10 * sizeof(int));

    /* Mit dem Zeiger a können wir genauso
       arbeiten wie mit dem Feld a[...],
       insbesondere auf a[0], a[1], ...
       zugreifen. */

    /*
    Cooler restlicher Code
    */

    return 0;
}
```

Beispiel: 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  }
```

Die dazugehörige main-Funktion kann folgendermaßen aussehen:

- 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 }
```

**Schema zur Anwendung dynamischer Speicherreservierung:**
Die Arbeit mit dynamischem Speicher folgt eigentlich immer dem selben Schema:

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)`

Beispiel:

```
int main(void)
{
    /* 1. Speicherplatz reservieren */
    int *p = malloc(10 * sizeof(int));

    /* 2. Fehlerbehandlung. Kann vorkommen, wenn
        z.B. nicht genug Speicher zur Verfügung
       steht */
    if (p == NULL) {
        printf("Fehler");
        return 1;
    }

    /* 3. Mache etwas mit p */
    ...


    /* 4. Speicherplatz freigeben, nachdem man
       fertig ist. */
    free(p);

    /* Restlicher Code - unabhängig von p */
    ...

    return 0;

}
```

### 9.6.3. Dynamische Reservierung von Speicher – 2. Möglichkeit

**Speicherreservierung mit Initialisierung:**
Eine Alternative zu `malloc` ist die Funktion `calloc`. Die beiden Funktionen verhalten sich sehr ähnlich. Beide initialisieren, wenn möglich, Speicherplatz. Der große Unterschied ist die Art der Initialisierung. Bei `malloc` bleibt der initialisierte Speicherbereich unverändert. Verwendet man stattdessen `calloc`, wird der reservierte Speicherbereich mit 0-Werten initialisiert.

Beispiel:

```
#include <stdio.h>
```

```c
#include <stdlib.h>

int main(void)
{
    int i;
    int *m = malloc(5 * sizeof(int));
    int *c = calloc(5, sizeof(int));

    /* Gibt irgendwelche Werte vom reservierten
       Bereich im Heap aus. Grundsätzlich zu
       vermeiden, da diese nicht initialisiert
       wurden. Kann zum Programmabsturz führen!
       */
    for (i = 0; i < 5; i++) {
        printf("%i. Stelle von m: %i\n", i, *(m+
            i));
    }
    printf("\n");

    /* Gibt nur 0 aus */
    for (i = 0; i < 5; i++) {
        printf("%i. Stelle von c: %i\n", i, *(c+
            i));
    }

    return 0;
}
```

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.

### 9.6.4. Speicherplatz wieder freigeben

Dynamisch im Heap reservierter Speicherplatz muss explitzit wieder freigegeben werden. Dies geschieht nicht automatisch, da der reservierte Speicherplatz in einer Funktion reserviert und außer-

halb der Funktion weiterbenutzt werden kann. C hat somit keinen Überblick, wann der reservierte Speicherplatz nicht mehr benötigt wird.

**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.

Die `free`-Funktion kann überall und in jeder Funktion aufgerufen werden. Es gilt zu beachten, dass sie immer da, wo es sinnvoll ist, genutzt werden sollte. Sinnvoll bedeutet hier: Wenn der Zeiger nicht mehr benötigt wird. Spätestens beim Beenden des Programms (am Ende der `main`-Funktion) muss jeder reservierte Speicherplatz freigegeben werden.

**Speicherleck:**
Ein **Speicherleck** ist ein dynamisch reservierter und nicht wieder freigegebener Speicherbereich, der nicht mehr benutzt wird und auf den u.U. nicht mehr zugegriffen werden kann (z.B. weil der zugehörige Zeiger überschrieben wurde).

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.

Es gilt aber zu beachten: 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.

**Tipp:** Sobald Sie irgendwo ein `malloc` hinschreiben, schreiben Sie sofort auch das `free` an die passende Stelle – dann vergessen Sie es später nicht mehr.

### 9.6.5. Dynamische Speicheranpassung

Bisher haben wir nur Speicherplatz reserviert und wieder freigegeben. Was ist aber, wenn wir schon Speicherplatz besitzen und merken, dass wir nicht genug (oder sogar zuviel) Speicherplatz reserviert haben? Eine Möglichkeit wäre die Anforderung von weiterem Speicher. Dies ist jedoch wenig effizient, da man dann kurze Zeit fast doppelt so viel Speicherplatz reserviert wie man eigentlich braucht (nämlich den „alten“, der zu klein geworden ist und den „neuen“, in den man die Daten auch noch umziehen muss). C nimmt uns diese Arbeit zum Glück mit der Funktion `realloc` ab, die dynamisch zur Laufzeit die Größe eines reservierten Speicherbereichs ändert.

**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`.

Man beachte, dass der Rückgabezeiger nicht mit `p` übereinstimmen muss. Insbesondere wenn Speicher vergrößert wird, kann es sein, dass sich der neue Speicherbereich woanders befindet als der alte, da hinter dem alten bereits andere Daten liegen könnten.

**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 automatisch freigegeben). Der Inhalt des zusätzlichen Bereichs ist nicht initialisiert.

**Speicherverkleinerung:**
Der Inhalt im verkleinerten Speicherbereich bleibt erhalten.

**Beispiele**

Beispiel:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *c = calloc(2, sizeof(int));
    int *tester;

    /* Irgendwas passiert hier.. */

    /* Man merkt, dass c nicht groß genug ist.
       Vergrößere deshalb den verfügbaren
       Speicherplatz mit realloc. Der neue
```

```
        Speicherbereich wird zuerst einem zweiten
         Zeiger zugewiesen. Wir überprüfen diesen
         zweiten Zeiger auf einen Fehlerfall. */
    tester = realloc(c, 10 * sizeof(int));

    /* Beachtung des Fehlerfalles */
    if (tester == NULL) {
        printf("Fehlerfall");
        /* Im Fehlerfall müssen wir den
           Speicherbereich freigeben, der vor
           dem realloc-Aufruf reserviert war.
           Danach können wir das Programm
           beenden. */
        free(c);
        return 1;
    }

    /* War der realloc erfolgreich, können wir
       den alten Zeiger überschreiben */
    c = tester;

    /* andere Codestücke */

    return 0;
}
```

Der zweite Zeiger ist für einen eventuellen Fehlerfall nötig. Würde der `realloc` folgendermaßen aussehen

```
c = realloc(c, 10 * sizeof(int));
```

dann verliert man den Zeiger auf die ursprünglichen Speicherzellen. Im Fehlerfall können wir nicht mehr auf diese zugreifen – was ein Speicherleck ist.

### Beispiel: 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   }
```

Für die Funktion `string_d_cat` ist kein zweiter Zeiger notwendig, da die Funktion auf einer Kopie des Zeigers arbeitet. Im Fehlerfall wird `NULL` zurück gegeben. Somit kann man den Rückgabewert auf einen Fehlerfall überprüfen und falls nötig den ursprünglich reservierten Speicher freigeben.

Die `main`-Funktion dafür sieht folgendermaßen aus:

- Zeile 5: Zeichenkette verlängern – der in der Funktion `string_d_cat` dynamisch reservierte Speicherplatz kann in `main` weiter benutzt werden
- Zeilen 6 - 10: Fehlerbehandlung, inklusive Freigabe von nicht mehr benötigtem Speicherplatz
- 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  }
```

## 9.7. Doppelzeiger: Zeiger auf Zeiger

Von einem *Doppelzeiger* sprechen wir, wenn ein Zeiger auf einen Zeiger oder ein Feld von Zeigern zeigt. Von dort aus wird auf Werte gezeigt.

Man hat also einen Zeiger, der auf einen Zeiger oder ein Feld von Zeigern zeigt. Von dort aus zeigt

man auf den/die eigentlichen Wert(e).

Dies lässt sich sogar auf Felder von Feldern erweitern, was wir uns im nächsten Kapitel 9.8 genauer anschauen.

### 9.7.1. Überblick

Je nachdem, was wir mit Doppelzeigern machen wollen, gibt es verschiedene Möglichkeiten, diese zu deklarieren. 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`, Details im Kapitel 9.8

Deklariert man jedoch einen Doppelzeiger über die letzte Möglichkeit (`T v[N][M]`), muss beachtet werden, dass für die hinteren Dimensionen bei Eingabeparametern immer eine Größe angegeben werden muss, damit C weiß, wie das Feld strukturiert ist. Immer wenn eine Funktion ein `T[N][M]` als Parameter übergeben bekommt, muss also eine Zahl für M angegeben werden.

### 9.7.2. Wertzuweisungen an Doppelzeiger

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

Hier ist unser Bezugstyp `T*`. Damit kann einem Zeiger `T** p` zugewiesen werden:

- 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;
  ```

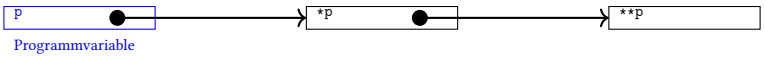
Folgerung:
Die Eingabeparameter `T** p` und `T* v[]` sind gleichwertig. In beiden Fällen wird in der Funktion mit einem Doppelzeiger gearbeitet. Daher können wir in der `main` auch entweder `char *argv[]` oder `char **argv` schreiben.

### 9.7.3. Anwendung von Doppelzeigern

Betrachten wir zunächst den Fall eines (Doppel-)Zeigers, der auf einen (Einfach-)Zeiger zeigt, der wiederum auf einen Wert zeigt:

```
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

> **Beispiel: Tausch von Zeigern**
>
> Wir hatten im Kapitel 9.2 beschrieben, dass als Parameter übergebene Zeiger nicht innerhalb einer Funktion umgebogen werden können. Mit Hilfe von Doppelzeigern geht dies aber doch.
>
> Greifen wir unser Einstiegsbeispiel zu Zeigern von Seite 3 nochmal auf. Dieses ändern wir nun so ab, dass nicht zwei `int`-Zahlen getauscht werden, sondern zwei Zeiger `int*`. Dazu müssen wir die Adresse der beiden Zeiger übergeben. Bei den Eingabeparamtern der Funktion heißt das, dass wir ein `*` vor jeden Parameter schreiben müssen. Das Innere der Tausch-Funktion bleibt abgesehen vom neuen Datentyp `int*` für `temp` gleich: Wir schauen, welche Adressen in den übergebenen Variablen gespeichert sind und überschreiben die Original-Variablen. Ebenso hat sich die `main`-Funktion nicht wesentlich geändert. Statt den Adressen zweier `int`-Variablen übergeben wir nun die Adressen zweier Zeiger `int*`.
>
> ```
> /* Implementierung genauso wie tausch2, nur dass
>     * in den Eingabeparametern ergänzt wurde */
> void tausch3(int **a, int **b)
> {
>     int* temp;
> ```

```
    temp = *a;
    *a = *b;
    *b = temp;
}

int main(void)
{
    int i = 21;
    int j = 9;
    int* x = &i;
    int* y = &j;

    /* Es werden Adressen der Zeiger übergeben.
       Damit werden die Zeiger vertauscht.
       Beachte: gleicher Aufruf wie bei tausch2
       */
    tausch3(&x, &y);

    printf("Wert hinter x: %i\n", *x);
    printf("Wert hinter y: %i\n", *y);

    return 0;
}
```

### Beispiel: Einfachzeiger in Funktion umbiegen

In diesem Beispiel ändern wir unsere Funktion `split` von Seite 25 derart ab, dass der Beginn der zweiten Zeichenkette nicht via `return` zurückgegeben wird, sondern in einem Zeiger v gespeichert wird, der als weiterer Parameter übergeben wird:

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

Statt `return &w[i + 1]` wie auf Seite 25 schreiben wir unsere Rückgabe nun in den Zeiger `v`: `*v = &w[i + 1]`. Dazu dereferenzieren wir den Doppelzeiger und erhalten die Original-Adresse von `v`. Diese überschreiben wir mit der Adresse des Zeichens, das nach dem ersetzten Zeichen kommt.

`main`:

```
int main (void)
{
    char in[] = "Informatik 1";
    char *rest;

    split_dzeiger(in, &rest, 'r');
    if (rest == NULL) {
        printf("Zeichen nicht gefunden\n");
        return 1;
    }

    printf("Erster Teil: %s\n", in);
    printf("Zweiter Teil: %s\n", rest);

    return 0;
}
```

Man beachte, dass wir hier in `split_dzeiger(in, &rest, 'r');` die Adresse eines „normalen“ Zeigers übergeben. Da es sich um die Adresse eines Zeigers handelt, arbeiten wir mit einem Doppelzeiger.

### Beispiel: Bibliotheksfunktion mit Doppelzeiger

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);
printf("Rest: %s\n", rest);
```

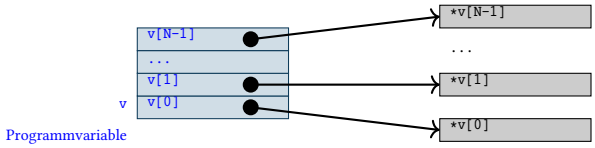
Da hier ein Doppelzeiger übergeben wurde, kann der Zeiger innerhalb der Funktion umgebogen werden.

### 9.7.4. Felder von Zeigern

Wir können auch direkt ein Feld von Zeigern anlegen, mit der bekannten Deklaration von Feldern:

```
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

> **Beispiel: Verschiedene Variablen über Feld von Zeigern erreichbar machen**
>
> In diesem Beispiel gibt es einzelne `int`-Variablen `a`, `b`, `c` und `d`, in denen Werte gespeichert sind. Auf diesen sollen nun jeweils die gleichen Operationen ausgeführt werden (hier: Multiplikation mit 10 und Ausgabe).
>
> Da die Variablen nicht als Feld vorliegen, ist das normalerweise umständlich, da man für jede davon eigenen Code schreiben müsste. Stattdessen nutzen wir ein Feld von Zeigern, in dem wir die Adressen dieser Variablen hinterlegen. Dadurch können wir anschließend Schleifen nutzen, um die gemeinsamen Operationen auf all diesen Variablen auszuführen.

```
1  #include <stdio.h>
2
3  #define LAENGE_ZEIGERFELD 3
4
5  int main(void) {
6      int *zeigerfeld[LAENGE_ZEIGERFELD];
7      int a = 1;
8      int b = 2;
9      int c = 3;
10     int i;
11
12     /* Speichere Adressen der Variablen im int*-
          Feld */
13     zeigerfeld[0] = &a;
14     zeigerfeld[1] = &b;
```

```
15      zeigerfeld[2] = &c;
16
17      /* Nun können die Variablen einheitlich
           erreicht werden */
18      for (i = 0; i < LAENGE_ZEIGERFELD; i++) {
19          *zeigerfeld[i] *= 10;
20      }
21
22      /* Ausgabe der Variablen */
23      for (i = 0; i < LAENGE_ZEIGERFELD; i++) {
24          printf("Variable %c: ", i+97);
25          printf("%i\n", *zeigerfeld[i]);
26      }
27
28      return 0;
29  }
```

## 9.8. Zweidimensionale Felder

Ein *zweidimensionales Feld* kann man sich wie eine Tabelle vorstellen: Das Feld ist in Zeilen und Spalten eingeteilt, in jeder Zeile stehen mehrere Werte. Dies lässt sich realisieren, indem man ein Feld von Zeigern anlegt – dies ist die erste Dimension. Es speichert für jede Zeile einen Zeiger, der auf ein Feld von Werten zeigt. Der erste Eintrag in diesen Feldern entspricht dem jeweiligen Wert in der ersten Spalte dieser Zeile, der zweite dem in der zweiten Spalte usw. Darüber ist also die zweite Dimension erreichbar. Einen Eintrag kann man somit über `v[Zeile][Spalte]` abrufen.

Solche zweidimensionalen Felder werden mit Hilfe von Doppelzeigern, wie wir sie gerade in Kapitel 9.7 kennengelernt haben, verwirklicht.

### 9.8.1. Motivation

Ein Beispiel für zweidimensionale Felder ist die `main`-Funktion:

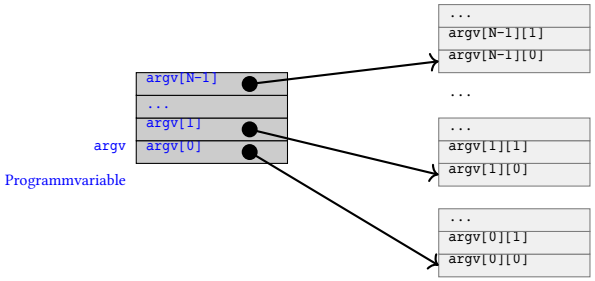
```
int main(int argc, char *argv[])
```

bzw. der Zeiger-Schreibweise:

```
int main(int argc, char **argv)
```

Der Parameter `argv` ist ein zweidimensionaler Zeiger. Wie aus früheren Kapiteln bekannt sein sollte, werden darin dem Programm übergebene Zeichenketten gespeichert.
Man besitzt mit `argv` ein Feld von Zeigern, das auf Zeichenketten zeigt (erste Dimension). Die

Zeichenketten wiederum sind Felder von `chars` (zweite Dimension) – auf sie wird mit den Zeigern gezeigt. Somit besitzt man mit `argv` einen Doppelzeiger.




- `argv[i]` ist die `i+1`-te Zeichenkette in der Liste
- `argv[i][j]` ist das `j+1`-te Zeichen der `i+1`-ten Zeichenkette
- Die Speicherbereiche, auf die die Zeiger `argv[i]` zeigen, werden dynamisch reserviert

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

### Beispiel: Temperatur für jeden Tag speichern

Speichere für jeden Tag eines Jahres die Mittagstemperatur:

- Bisher können wir die Temperaturen als Folge $(t_i)_{1 \leqslant i \leqslant 365}$ 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.

### Beispiel: 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** $A = (a_{i,j})_{1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n}$.[a]

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

[a] Falls Sie Vektoren und Matrizen nicht schon aus der Schule oder der Mathematik-Vorlesung kennen, belesen Sie sich selbst dazu. Im Anhang A.1 finden Sie eine kurze Zusammenfassung.

**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.

### 9.8.2. 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.

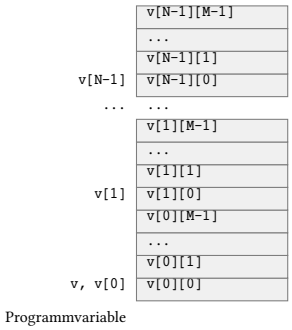
### 9.8.3. Statische 2-dimensionale Felder

Die einfachste Möglichkeit, um einen zweidimensionales Feld zu deklarieren, ist statisch mit der bekannten Feldschreibweise:

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

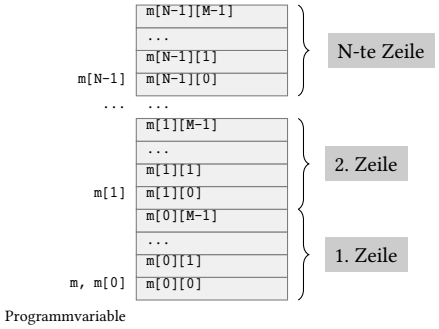
Hier wird ein Feld deklariert, dessen Komponenten wieder Felder sind (lies die Deklaration ausgehend von `v` **von innen nach außen**):

- `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 ($0 \leqslant i \leqslant N-1$)
- `v[i][j]` sind Variablen vom Typ `T` ($0 \leqslant i \leqslant N-1, 0 \leqslant j \leqslant M-1$)




**Beispiel:**

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, für die verwendeten Dimensionen verwendet man symbolische Konstanten `N`, `M`. Wenn er nicht vollständig gebraucht wird (kleinere Matrizen), ist es auch möglich, nur Teile davon zu verwenden.

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 (siehe Anhang A.2).

**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;
}
```

Erstellen Sie zur Übung selbst eine Funktion `matrix_print`!

### 9.8.4. Dynamische 2-dimensionale Felder

Auch dynamisch können wir ein zweidimensionales Feld anlegen. Wir benötigen hierfür lediglich einen Doppelzeiger und die aus Kapitel 9.6 bekannten Operationen `malloc` und `free`.

Beispiel:

In diesem Beispiel legen wir einen Doppelzeiger an, der auf ein `int*`-Feld mit 10 Einträgen (erste Dimension) zeigt. Von dort aus kann auf `int`-Felder zugegriffen werden, die jeweils 5 Einträge besitzen (zweite Dimension). Man kann es sich also auch als Tabelle mit 10 Zeilen und 5 Spalten vorstellen.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define ROWS 10
5  #define COLUMNS 5
6
7  int main(void) {
8      int i;
9      int **m;
10
```

```
11      /* Reserviere zuerst Speicherplatz für die
           erste Dimension. Beachte: Es muss Platz f
           ür ein Feld von int*-Zeigern reserviert
           werden */
12      m = malloc(ROWS * sizeof(int*));
13
14      /* Überprüfe auf einen Fehlerfall */
15      if (m == NULL) {
16          printf("1. Fehlerfall");
17          return 1;
18      }
19
20      /* Nun müssen wir den Speicherplatz für die
           10 int-Felder reservieren (zweite
           Dimension). Dafür kann man eine Schleife
           nutzen. */
21      for (i = 0; i < ROWS; i++) {
22          /* Dafür spricht man, analog wie bei
               einem normalen Feld, die einzelnen
               Einträge der ersten Dimension (int*-
               Feld) an. */
23          m[i] = malloc(COLUMNS * sizeof(int));
24
25          /* Auch hier müssen wir nochmal einen
               Fehlerfall überprüfen */
26          if (m[i] == NULL) {
27              /* Im Falle eines Fehlers müssen wir
                   den vorher reservierten
                  Speicherplatz freigegben. Sonst
                  haben wir Speicherlecks - was wir
                   natürlich vermeiden. Dafür gehen
                   wir alle bisher erzeugten Felder
                   durch und rufen die Funktion
                  free() darauf auf. */
28              int j;
29              for (j = 0; j < i; j++) {
30                  free(m[j]);
31              }
32              /* Das int*-Feld der ersten
                   Dimension dürfen wir auch nicht
                   vergessen */
33              free(m);
34
35              printf("2. Fehlerfall");
```

```
36                  return 1;
37              }
38          }
39
40          /* Nun können wir auf die einzelnen Elemente
                via m[x][y] zugreifen, wobei 0 <= x < 10
                und 0 <= y < 5 */
41
42          /* Freigabe analog zum 2. Fehlerfall */
43          for (i = 0; i < ROWS; i++) {
44              free(m[i]); /* alle int-Felder der
                   zweiten Dimension freigeben */
45          }
46          /* Das int*-Feld der ersten Dimension dürfen
                wir auch nicht vergessen */
47          free(m);
48
49          return 0;
50      }
```

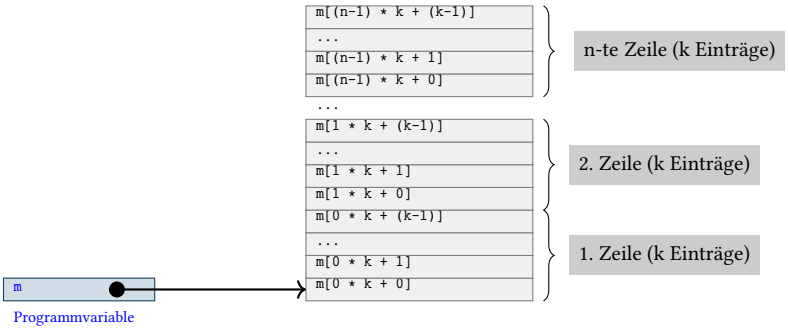
Ein wichtiges Beispiel, welches ebenso beide Dimensionen dynamisch reserviert, ist die dynamische Matrizenrechnung, welche im Anhang A.2 ausführlich erklärt wird.

### 9.8.5. Dynamische Matrizenrechnung mit Einfach-Zeigern

Statt mit Doppelzeigern ein Feld der Art `T v[N][M]` zu erstellen, kann man auch Einfachzeiger nutzen, um ein großes eindimensionales Feld `T v[N*M]` zu erstellen.

Repräsentiere eine Matrix mit `n` Zeilen und `k` Spalten durch einen Einfach-Zeiger auf Speicherbereich mit `n * k * sizeof(T)` Byte und reserviere den Speicherbereich dynamisch:
`T *m = malloc(n * k * sizeof(T));`

Alternativ lässt es sich auch statisch anlegen: `T m[N*M];`
Hierbei müssen `N` und `M` jedoch zum Zeitpunkt des Compilierens bekannt sein, d.h. als symbolische Konstanten realisiert werden.




Beim Zugriff muss man etwas umdenken:
`m[i * k + j]` ist der `j+1`-te Eintrag der `i+1`-ten Zeile. Es emmpfiehlt sich daher, Hilfsfunktionen zu schreiben, um den Zugriff zu vereinfachen (z.B. im Wetter-Beispiel: `int get_temperatur(int tag, int monat)`

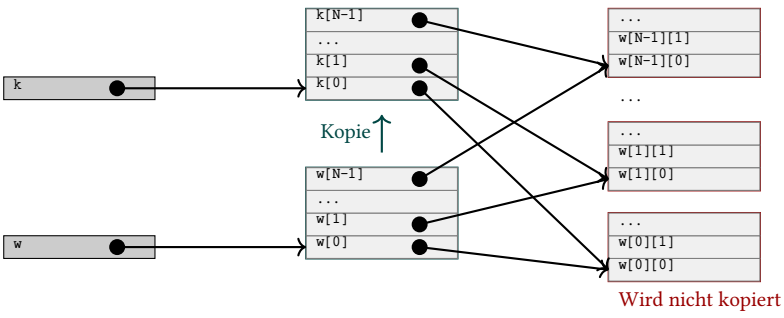
## 9.9. Kopien adresswertiger Variablen

Wenn man ein Feld, eine Tabelle, Matrix o.ä. kopiert (z.B. um sie einer Funktion zu übergeben), hat man immer zwei Möglichkeiten: Entweder man kopiert nur die Adressen oder man kopiert alles. Man spricht von flachen oder tiefen Kopien.

### 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`



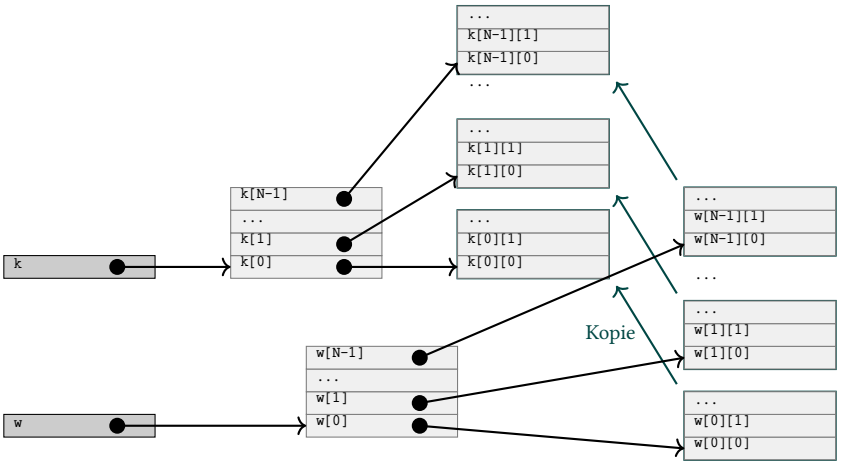

Konsequenz: Spart Speicher, aber die Kopie lässt sich nicht unabhängig vom Original verändern

Beispiel:

Ein anderes Beispiel wäre die Übergabe von Feldern an eine Funktion. Es wird nur die Adresse des Feldes übergeben und die Daten können direkt verändert werden.

### Tiefe Kopien

Bei einer **tiefen Kopie** einer adresswertigen Variablen wird **die unterste Ebene der Dateninhalte** kopiert. Das bedeutet, es wird *alles* kopiert.




Beispiel:

Flache und tiefe Kopien lassen sich am besten anhand der Musik-Industrie erklären.
Eine flache Kopie wäre das Nutzen von Streaming Diensten. Jeder NutzerIn bekommt Adressen, mit denen er/sie auf dieselben Dateien zugreifen kann wie alle anderen NutzerInnen. Mit dieser Adresse kann man die Musik dann abrufen und abspielen. Man besitzt aber nicht die Dateninhalte.
Eine Tiefe Kopie wäre das Kaufen einer CD. Die Musik befindet sich auf der CD - die Inhalte wurden also auf die CD kopiert. Man besitzt damit die Dateninhalte.

# A. Anwendungen von zweidimensionalen Feldern

## A.1. Wiederholung: Vektoren und Matrizen

### Notationen für Vektoren

Definition: 9.21 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 $v = (v_i)_{1 \leqslant i \leqslant n}$

### Operationen auf Vektoren

- Addition von $v, w \in X^n$: $v + w := (v_i + w_i)_{1 \leqslant i \leqslant n} \in X^n$
- Skalarmultiplikation von $v \in X^n$ mit $x \in X$: $x \cdot v := (x \cdot v_i)_{1 \leqslant i \leqslant n} \in X^n$
- 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: 9.22 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 $A = (a_{i,j})_{1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n}$ oder

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

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

### Operationen für Matrizen

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

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

## A.2. Anwendung Doppelzeiger: Dynamische Matrizenrechnung

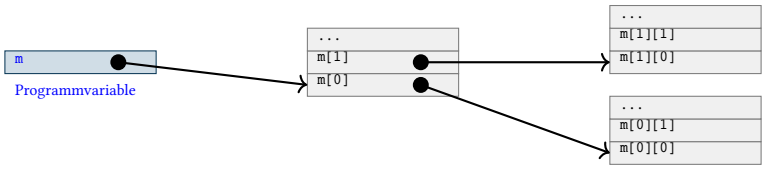
Das große Ziel (wo es viele Beispiele für Doppelzeiger gibt) sind Funktionen, um dynamisch mit Matrizen umgehen zu können. Vorerst beschränken wir uns dabei auf das Erzeugen, Initialisieren, Ausgeben und Löschen einer beliebigen Matrix.

Eine Matrix kann man mithilfe von Doppelzeigern realisieren.

```
T **m;
```

Wir sehen, dass sich an der Schreibweise nichts geändert hat. Es ist „der gute alte Doppelzeiger“, wie wir ihn schon kennengelernt haben.

Visualisiert man diesen Doppelzeiger (die Matrix), kann man ihn wie folgt darstellen:




- `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

**Verwaltungsfunktionen:**
Wie oben schon angesprochen wollen wir Funktionen für Matrizen erstellen:

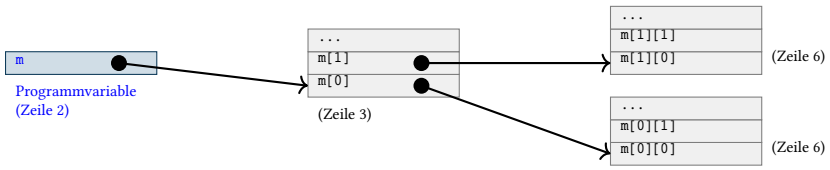
- Speicherplatz reservieren für Matrix mit `ze` Zeilen und `sp` Spalten:
  `int **matrix_create(int ze, int sp);`
  Liefert Zeiger auf den reservierten Speicherplatz im Erfolgsfall und `NULL` sonst
- Für Matrix reservierten Speicherplatz wieder freigeben:
  `void matrix_destroy(int **m, int ze);`
  Gibt 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

`matrix_create`: **Dynamische Speicherreservierung für Matrix**
Die Funktion `matrix_create` soll dynamisch eine leere Matrix erzeugen. Sie soll `ze` viele Zeilen und `sp` viele Spalten besitzen.
Betrachtet man dafür den Code, sollte dem Leser bzw. der Leserin auffallen, dass der Code analog zum Beispiel aus dem Unterkapitel 9.8.4 funktioniert.

```
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 der vorher erfolgreich reservierter Speicher wieder freigegeben werden (Zeilen 7 - 10)

In unserem Bild von gerade eben haben wir ergänzt, welcher Teil in welcher Code-Zeile angelegt wird:




`matrix_init`: **Matrix mit Zufallszahlen initialisieren**
Wir wollen nun die Matrix mit Zufallszahlen initialisieren. Dafür schreiben wir eine Funktion, die einen Doppelzeiger sowie dessen Zeilen- und Spaltenanzahl erwartet. Mittels zwei `for`-Schleifen können wir die Matrix befüllen. Der Zugriff auf die einzelnen Elemente erfolgt dabei einfach über die Feldschreibweise.

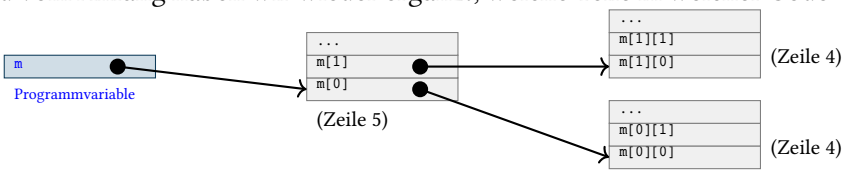
```
1  void matrix_init(int **m, int ze, int sp) {
2      int i,j;
3      for(i = 0; i < ze; ++i) {
4          for(j = 0; j < sp; ++j)
5              m[i][j] = rand() % 1000;
6      }
7  }
```

`matrix_destroy`: **Matrix löschen und Speicher freigeben**
Um die Matrix zu löschen (bei Programmende o.ä.) müssen wir sie wieder in Form eines Doppelzeigers übergeben. Es wird jedoch nur die Zeilenanzahl benötigt, um alle gespeicherten Adressen der Zeilen freigeben zu können.

```
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  }
```

Wir iterieren in einer `for`-Schleife alle Zeilen durch. Damit können wir nacheinander alle Spalten löschen und anschließend den Doppelzeiger selbst.
Es gilt zu beachten: Es reicht NICHT, nur auf dem Doppelzeiger die `free`-Funktion aufzurufen. Es wird damit zwar der Doppelzeiger gelöscht, jedoch nicht der Inhalt des Doppelzeigers. Somit bleiben die Einträge in der Matrix erhalten, auf die wir nicht mehr zugreifen können und es kommt zu einem Speicherleck.

In unserem Bild vom Anfang haben wir wieder ergänzt, welche Teile in welcher Code-Zeile freigeben werden:




`matrix_print`: **Matrix ausgeben**
Sie haben nun gesehen, wie eine Matrix angelegt, initialisiert und gelöscht wird. Damit sollte es ein Leichtes sein, eine Ausgabefunktion zu implementieren. Machen Sie dies zur eigenen Übung selbst!

### Main-Funktion

Nachdem alle Funktionen implementiert wurden, können wir können sie jetzt in einer `main`-Funktion aufrufen und testen:

```
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  }
```

## Kleines Verständnisbeispiel von Seite 20 – Auflösung

Beispiel:

Kleine Aufgabe für das große Verständnis: Was steht in welcher Speicherzelle?
Das Ergebnis gibt es am Ende des Skriptes, im Appendix.

```
int a = 5;        // Im Speicher an Adresse S20
int *b = &a;      // Im Speicher an Adresse S24
int *c = b;       // Im Speicher an Adresse S28
*b = 6;
c++;
int *d;           // Im Speicher an Adresse S2C
```

Speicher:

| | |
|---|---|
| S20 | a: 6 |
| S24 | b: S20 |
| S28 | c: S24 |
| S2C | d: *undefiniert* |

Achtung: Der Zeiger c darf normalerweise **nicht** via c++ umgebogen werden, da er nicht auf ein Feld zeigt und somit unklar ist, worauf er nach dem Weiterzählen zeigt. Es gibt keine Garantie dafür, dass hinter einer Variable die nächste Variable im Speicher hinterlegt ist.