

Meine Bürotür, bald wieder offen  
für Sie.

# Informatik II

## Speicher und Listen

**Irene Rothe**

Zi. B 241

[irene.rothe@h-brs.de](mailto:irene.rothe@h-brs.de)

Instagram: [irenerothedesign](#)



Hochschule  
Bonn-Rhein-Sieg

Vorlesung\_L\_SpeicherListen

# Ungeplante Software ist ein Zufallsprodukt.



# Das Marie Kondo Prinzip

→ If it doesn't spark joy, get rid of it.

Wenn etwas keinen Spaß macht, ändere es!

If a function or line doesn't spark joy, get rid of it.

Quelle: <https://www.karim-geiger.de/blog/das-marie-kondo-software-design-principle>

# Informatik: 2 Semester für Ingenieure

## Informatik = Lösen von Problemen mit dem Rechner

✓ Zum Lösen von Problemen mit dem Rechner braucht man **Programmierfähigkeiten** (nur mit Übung möglich): Was ist Programmieren?

✓ Was ist ein Flussdiagramm?

### → Programmiersprache C:

- ✓ Elementare Datentypen
- ✓ Deklaration/Initialisierung
- ✓ Kontrollstrukturen: if/else, while, for
- ✓ Funktionen
- ✓ Felder (Strings)
- ✓ Zeiger
- ✓ struct

→ Speichieranforderung: malloc

→ Listen

→ Bitmanipulation

✓ Wie löst der Rechner unsere Probleme? → mit **Dualdarstellung** von Zeichen und Zahlen und mit Hilfe von **Algorithmen**





✓ Ein Beispiel für ein Problem: **Kryptografie**

→ Sind Rechner auch Menschen? → **Künstliche Intelligenz**

✓ Für alle Probleme gibt es viele Algorithmen. Welcher ist der Beste? → **Aufwand** von Algorithmen



# Design der Folien

-  hinterlegt sind alle Übungsaufgaben. Sie sind teilweise sehr schwer, bitte absolut nicht entmutigen lassen! Wir können diese in Präsenz besprechen oder über Fragen im Forum.
-  hinterlegte Informationen und grüne Smileys sind wichtig und klausurrelevant.
- Alles hinter „**Achtung**“ unbedingt beachten!
-  verwende ich, wenn überraschende Probleme auftreten können. Wenn Sie schon programmiererfahrend sind, können das eventuell besonders große Überraschungen für Sie sein, wenn Sie eine andere Sprache als C kennen.
- „Tipp“ benutze ich, um Ihnen einen Weg zu zeigen, wie ich damit umgehen würde.
- „Bemerkung“ in Folien beziehen sich meist auf Sonderfälle, die nicht unbedingt klausurrelevant sind, aber für Sie beim Programmieren eine Bedeutung haben könnten
-  hinter diesem Symbol ist ein Link fürs Anhören bzw. Gucken weiterer Infos

### Aufbau der Folien:

- Am Anfang motiviere ich gerne mit einem Beispiel, das eventuell schwer verständlich ist. Wem das nicht zusagt, dem empfehle ich, diese Folien zu überspringen.
- Weiter arbeite ich mit vielen Beispielen, die oftmals immer wieder das Gleiche erklären nur auf unterschiedliche Arten. Hat man einen Sachverhalt einmal verstanden, braucht man eventuell diese Beispiele nicht.
- Folien, die mit **Einschub** beginnen, beinhalten Zusatzinformationen, die nicht nötig für das Verständnis des Themas sind.
- Grün hinterlegte Informationen sind das, was Sie aus der Vorlesung rausnehmen sollen, alles andere sind vertiefende Informationen und Motivation.



## → Dynamischer Speicher

- Speicher anfordern
- Speicher verändern
- Speicher freigeben
- dynamische Felder
- flexible Eingabe
- Speichermüllbeseitigung

## → Listen

- Definition einer Liste
- Anlegen einer Liste
- Ausgabe einer Liste
- Einfügen eines Listenelements
- Löschen eines Listenelements
- Vor- und Nachteile: Listen und Felder



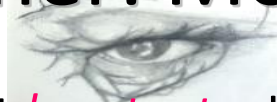
# Speicher und Listen: Motivation

1. Okay, ich will viele gleiche Dinge abspeichern. Ich weiß aber nicht wie viele. Ich könnte ein Feld definieren, aber da muss ich ja wissen, wie groß das Feld sein soll
2. Da muss sich doch jemand etwas Cleveres ausgedacht haben!!!
3. Ebenso ist doch ein Feld doof, wenn ich da in die Mitte etwas einfügen will oder löschen will, da ordne und kopiere ich mich ja tot.





# Dynamischer Speicher: Motivation



Bis jetzt können wir Felder nur mit *konstanter* Länge *vor* der Compilierung vereinbaren.

Daraus ergeben sich folgende Probleme:

- Länge zu groß: Speicherplatz verschwendet!
- Länge zu klein: Falle!

Oft ergibt sich erst während des Programmlaufs in Abhängigkeit von eingegebenen Daten der Wunsch, neuen Speicher zu reservieren und/oder nicht mehr benötigten Speicher frei zu geben (damit ein anderes Programm den nutzen kann!).

Dies gilt besonders für Programme, die sehr lange laufen wie z.B. Betriebssysteme.

# Dynamischen Speicher: Hauptproblem



## → Dynamischer Speicher

- Speicher anfordern
- Speicher verändern
- Speicher freigeben
- dynamische Felder
- flexible Eingabe
- Speichermüllbeseitigung

## → Listen

- Definition einer Liste
- Anlegen einer Liste
- Ausgabe einer Liste
- Einfügen eines Listenelements
- Löschen eines Listenelements
- Vor- und Nachteile: Listen und Felder



# Dynamischer Speicher: Anfordern


- Zusammenhängenden Speicher anfordern auf dem Heap (engl. memory allocate):

Einfügen ...

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

Dort wird Folgendes definiert: `void* malloc(int numberOfBytes);`

Adresse zeigt auf jeden beliebigen  
Datentyp



Zurückgegeben wird die Adresse, wo der angeforderte Speicher beginnt.

- Freigeben von Speicher:

In

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

wird Folgendes definiert: `void free(void* zeiger);`

**Also:** man muss nur die stdlib einbinden und kann malloc und free nutzen.

**Bemerkung 1:** normalerweise leben alle Variablen auf dem Stack (Stapel), der viel **kleiner** ist, aber **schnell** ist und automatisch verwaltet wird vom BS. Der Heap (Halde)-Speicher ist viel **größer**, aber **langsamer** und wird selbst verwaltet!

**Bemerkung 2:** `malloc` gibt typenlosen Zeiger zurück, damit der Speicher beliebig genutzt werden kann.



# Dynamischer Speicher: Beispiel - Feld

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

int main(){
//Deklaration des Zeigers speicheradresse
double *speicheradresse;
int groesse;
printf("Wie viel Speicher wird benötigt?");
scanf("%i",&groesse); //z.B. 2

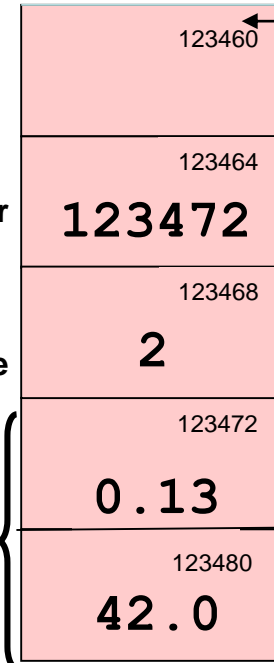
speicheradresse=malloc(groesse*sizeof(double));
if(groesse>1){
    *speicheradresse=0.13;
    *(speicheradresse+1)=42.0;
}
else{
    printf("Schade, niemand wollte Speicher!");
}
return 0;
}
```

Funktion, die Größe von ... (hier z.B. double)  
zurück gibt

speicher  
adresse

groesse

Adresse  
des  
Speicherp  
latzes



<https://youtu.be/cfukc0Qbn60>



# Speicher anfordern, noch besser

```
double z;  
double *zeiger;  
  
zeiger = malloc(sizeof(double));  
//Test, ob genügend Speicher  
//hintereinander vorhanden  
if (zeiger!= NULL){  
    *zeiger = 1.113;  
}  
else{  
    // Fehlerbehandlung, z.B. return 1;  
    //z.B. Kein Speicher mehr da!  
}
```





# Speichergröße verändern

```
double *speicheradresse;  
  
speicheradresse=malloc(2*sizeof(double));  
if (speicheradresse!= NULL){  
    *speicheradresse=1;  
    *(speicheradresse+1)=2;  
}  
else{  
    //Fehlerbehandlung  
}  
//neue Groesse ist 3*double  
speicheradresse = realloc(speicheradresse,3*sizeof(double));  
*(speicheradresse+2) = 42;
```

Ist auch in stdlib.h definiert

Der Inhalt vom alten Speicher (realloc nur nach einem malloc) wird umkopiert, wenn es die neue Größe zulässt. Der Speicherbereich im Rechner kann dabei ein völlig anderer sein.



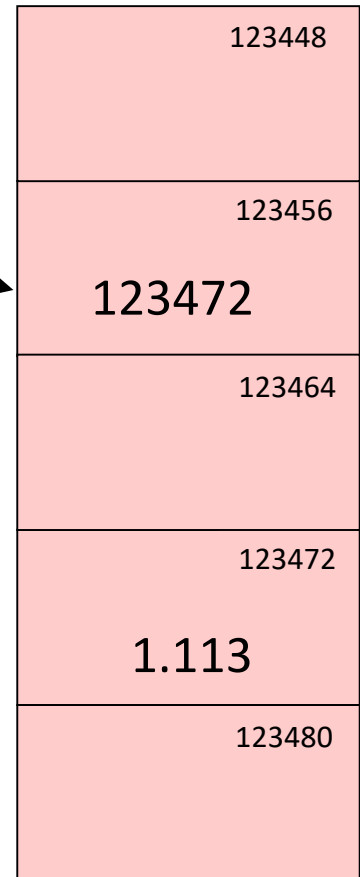
# Dynamischen Speicher wieder freigeben:

```
double *zeiger;
```

```
zeiger = malloc(sizeof(double));
```

```
*zeiger = 1.113;
```

```
free(zeiger);
```



# Dynamischer Speicher: Feld (eindimensional)

```
int feld[100]; //nicht dynamisch
int *dyn_feld; //Zeiger für dynamisches Feld
int feldgroesse;

printf("Wie groß soll das Feld sein? ");
scanf("%i", &feldgroesse);
//Anlegen des dynamischen Feldes dyn_feld
dyn_feld = malloc(sizeof(int) * feldgroesse);
if(dyn_feld == NULL){
    //Fehlerbehandlung
}
else{
    dyn_feld[88]=13; //schlimme Fehlerquelle!
    feld[88]=13;
}
```

Frage: Warum ist dies eine schlimme Fehlerquelle?



# Dynamischer Speicher: Feld (2-dim.)

```
int matrix[2][3]; // nicht dynamisch
int **dyn_matrix;
int zeilen, spalten;
int i;
```

```
zeilen = 2;
spalten = 3;
```

```
dyn_matrix = malloc(sizeof(int*) * (zeilen));
```

```
for(i=0; i<zeilen; i++){
    dyn_matrix[i] = malloc(sizeof(int) * (spalten));
    /* (dyn_matrix+i) = malloc(sizeof(int) * (spalten));
}
```

**Bemerkung:** Die Zeilen einer dynamischen Matrix müssen nicht unbedingt hintereinander liegen!



<https://youtu.be/Mp0H8fRstuo>



# Dynamischer Speicher: Flexible Eingabe

```
#include <stdio.h>
#include <string.h> //strlen: Länge von String, strcpy: Kopieren
//String in String
#include <stdlib.h>
int main(){
    1: char *hilfsfeld;
    2: char *eingabe;

    3: hilfsfeld = malloc(sizeof(char) * 100);
    4: printf("Eingabe eines Wort kleiner 100 Buchstaben:");
    5: scanf("%s",hilfsfeld);
        //Nicht wichtig: löscht das Enter in der Eingabe
    6: fflush(stdin);
    7: eingabe = malloc(sizeof(char)*(strlen(hilfsfeld)+1));
        //hilfsfeld wird nach eingabe kopiert
    8: strcpy(eingabe,hilfsfeld);
    9: free(hilfsfeld);
    return 0;
}
```

Bemerkung: Das +1 im malloc wird für das Endezeichen '\0' von Strings benötigt.  
Beispiel: Beim Umzug Dinge in der Garage zwischengelagern...



# Achtung: Dynamischer Speicher: Freigeben

## Fehlerquelle



Verlust des Speichers: man weiß nicht mehr, wo die 1.113 gespeichert ist (oder: wenn man die Nummer eines Schließfaches vergisst, ist die Tasche im Schließfach zwar noch da, aber man weiß nicht wo sie ist)

```
double z;  
double *zeiger;  
  
z = 99.9;  
  
zeiger = malloc(sizeof(double));  
*zeiger = 3.333;  
zeiger = &z;
```



# Dynamischer Speicher: Übung

Schreibe eine Funktion, die 2 Strings nacheinander in einen schreibt!

Achtung: Auf der nächsten Folien steht die Lösung!



# Dynamischer Speicher: Lösung

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main(){
    1: char string1[]="Hasen";
    char string2[]="futter";
    2: char *string3;//Zeiger auf ein Zeichen
    3: string3=zusammenfuegenZweierStrings(string1,string2);
    4: printf("%s",string3);
    return 0;
} //aller Speicher wird frei gegeben
```

# Dynamischer Speicher: Lösung

```
1: char * zusammenfuegenZweierStrings(char *s1, char *s2){
    int i=0;
2:     char *neuer_string;//Zeiger für das Endwort
3:     neuer_string=malloc(sizeof(char)*(strlen(s1)+strlen(s2)+1));
4:     if(neuer_string!=NULL){
5:         while(*(s1+i)!='\0'){
6:             *(neuer_string+i)=*(s1+i);
                i++;
            }//i ist jetzt am Ende des ersten Wortes
            while(*(s2+i-strlen(s1))!='\0'){
                *(neuer_string+i)=*(s2+i-strlen(s1));
                i++;
            }
7:         *(neuer_string+i)='\0';
            }else{//...}
8://     int *a,b;
        return neuer_string;
    }
```

# Dynamischer Speicher: Lösung mit call by reference

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main () {
    char string1[]="Hasen";
    char string2[]="futter";
    char *string3;
    kombiniereStrings(string1,string2,&string3);
    printf("%s", string3);
    return 0;
}
```

# Dynamischer Speicher: Lösung mit call by reference

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void kombiniereStrings(char str1[],char str2[],char **str3){
    int laengestr1,laengestr2,i;
    laengestr1=strlen(str1);
    laengestr2=strlen(str2);
    //char *str3;
    *str3=malloc(sizeof(char)*(laengestr1+laengestr2+1));
    for(i=0;i<laengestr1;i++){
        *(*str3+i)=str1[i];
    }
    for(i=0;i<laengestr2;i++){
        *(*str3+i+laengestr1)=str2[i];
    }
    /*(*str3+laengestr1+laengestr2) = '\0';
    *(str3[0]+laengestr1+laengestr2) = '\0';
}
```



# Bemerkungen: Speichermüllbeseitigung

Wird häufig Speicher angefordert und teilweise wieder freigegeben, so kann es passieren, dass der Heap-Arbeitsspeicher in viele kleine Bereiche fragmentiert wird, die abwechselnd frei und belegt sind → Löchersalve

Irgendwann kann man keine längeren Speicherbereiche mehr reservieren, obwohl insgesamt noch genügend Platz ist.

Dann muss man *Garbage Collection* durchführen, wobei belegte Speicherbereiche umkopiert werden, so dass sie lückenlos hintereinander liegen.

In C muss man dies selbst programmieren, wofür es die Funktionen `memcpy` und `memmove` gibt.

In Sprachen, wie z.B. Java, ist so eine Garbage Collection eingebaut, was aber auch nicht immer super ist (Zeitverzögerung, man weiß nie, wann das passiert).



# Dynamische Speicheranforderung

- Funktion **malloc**: Anforderung von Speicher auf dem Heap  
→ kann schief gehen!
- Funktion **free**: Freigabe von Speicher
- Achtung: wer viel Macht hat, kann viel falsch machen!!!
- Nullpointer **NULL** bedeutet „obdachlos“ (mehr als obdachlos geht nicht)
- Anschauliches Bild für **malloc**: Dash-Kamera im Auto

Coco benutzt malloc:

<https://www.youtube.com/watch?v=GoCROdDejeE>



Coco startet mit der C Programmierung Teil 7 Dynamische Speicherplatzreservierung



Hochschule  
Bonn-Rhein-Sieg

Vorlesung\_L\_SpeicherListen

27

# In Präsenz: Speicheranforderung

- LangsamerTod.ppt
- Stack overflow bei mir bei Deklaration eines Feldes der Größe 1000.000
- Speicherlecks.c (siehe Taskmanager für Arbeitsspeicherbelegungsbeobachtung, so kann Computer sogar stehen bleiben) → das führe ich ganz am Ende vor!

## ✓ Dynamischer Speicher


- Speicher anfordern
- Speicher verändern
- Speicher freigeben
- dynamische Felder
- flexible Eingabe
- Speichermüllbeseitigung

## → Listen (einfach verkettet)

- Definition einer Liste
- Anlegen einer Liste
- Ausgabe einer Liste
- Einfügen eines Listenelements
- Löschen eines Listenelements
- Vor- und Nachteile: Listen und Felder



# Listen: Motivationsbeispiele

1. Telefonliste (Beispiel jetzt veraltet, da es SMS gibt)
2. Folien einer Vorlesung
3.  <https://www.youtube.com/watch?v=p8ZLiDG1WnY>

# Listen: Motivation – Felder können zu wenig

Nachteile von Feldern:

1. Felder sind statisch

```
int array[20];
```



2. Einfügen/Löschen in der Mitte aufwendig

```
int array[20]= {11,100,110,500};
```

```
//105 einfügen  
array[4]=array[3];  
array[3]=array[2];  
array[2]=105;
```

→ Nicht nur der Algorithmus sollte effektiv sein, auch die Datenstruktur!

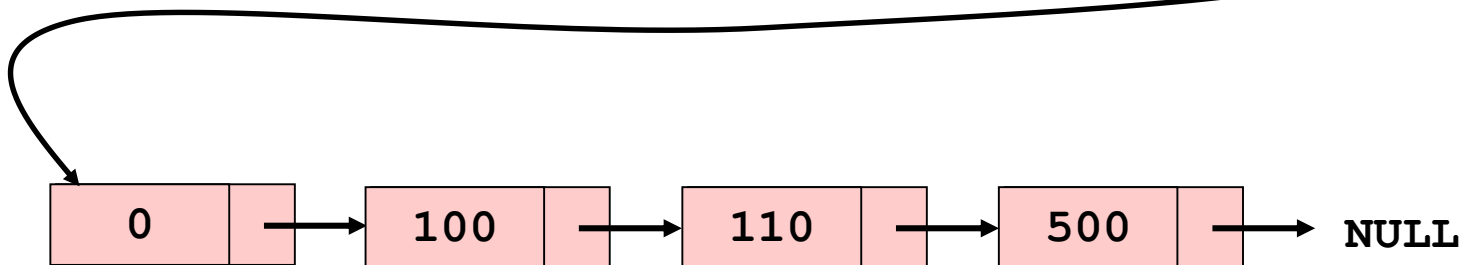
# Listen: Warum und wofür

- Listen sind **dynamische** Datenstrukturen, wie z.B. Dateien.
- Eine (einfach) verkettete Liste ist eine durch Zeiger verkettete Folge von Daten gleichen Aufbaus (dargestellt durch **struct**), genannt Listenelemente. Jedes Listenelement, bis auf das letzte, hat genau einen Nachfolger.
- Listen liegen im Speicher **nicht unbedingt hintereinander** (im Gegensatz zu Feldern).
- **Vorteile:** Einfügen und Löschen von Daten geht sehr einfach. Speicher wird nur angefordert, wenn er gebraucht wird.
- Verkettete Listen werden durch eine **rekursive** Struktur (Zeiger auf eine Struktur vom selben Typ) definiert.
- Folgende (schon bekannte) Dinge spielen bei der Arbeit mit Listen in C eine Rolle:
  - Strukturen: **struct**,
  - Zeiger,
  - Speichieranforderung: `malloc`,
  - Rekursion



# Listen: Begriffe und Beispiel

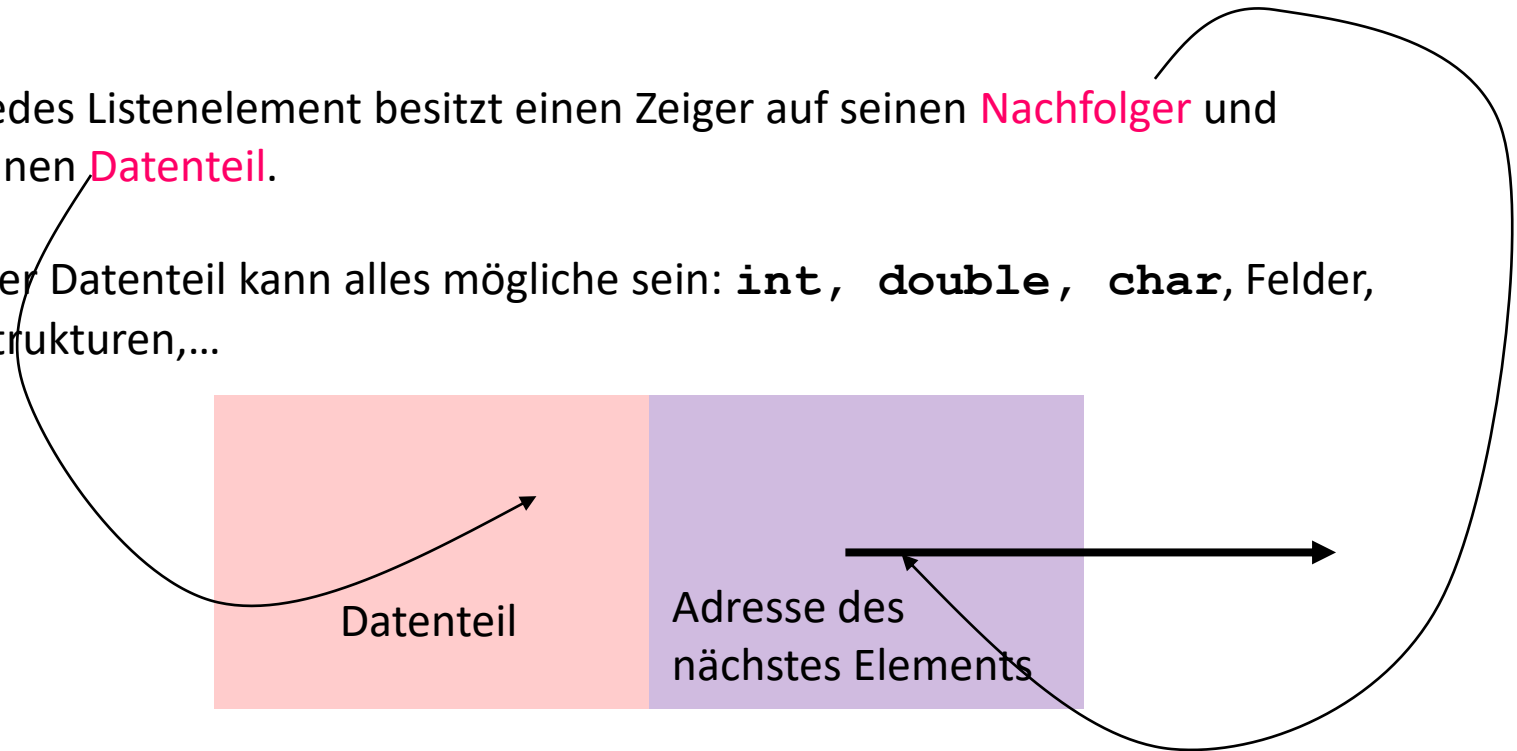
- Eine (einfach) verkettete Liste ist eine durch Zeiger verkettete Folge von Datenstrukturen, genannt **Listenelemente**.
- Jedes Listenelement sieht **gleich** aus.
- Jedes Element, bis auf das letzte, hat genau einen Nachfolger.
- Auf eine Liste greift man zu über die **Adresse des ersten Elements**.
- Vom ersten Listenelement ausgehend, „hängelt“ man sich von Listenelement zu Listenelement.
- Kommt man bei NULL an, ist das Ende der Liste erreicht.
- Listenelemente liegen im Speicher nicht unbedingt hintereinander. Nur der Vorgänger weiß, wo das nächste Listenelement im Speicher liegt.



# Listen: Bestandteile – Datenteil und Nachfolger

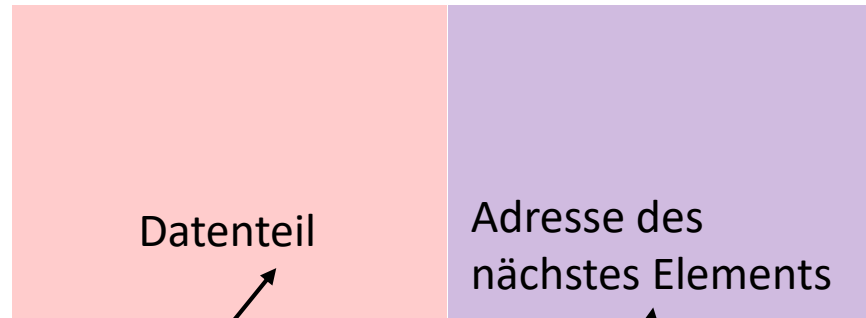
Jedes Listenelement besitzt einen Zeiger auf seinen **Nachfolger** und einen **Datenteil**.

Der Datenteil kann alles mögliche sein: **int**, **double**, **char**, Felder, Strukturen,...



Ein Listenelement selbst **definiert** man in einer **Struktur**, weil es *immer* aus einem Datenteil und einer Adresse besteht, was in der Regel mindestens zwei verschiedene Datentypen sind, die ein Listenelement bilden.

# Listen: Definition eines Listenelementes

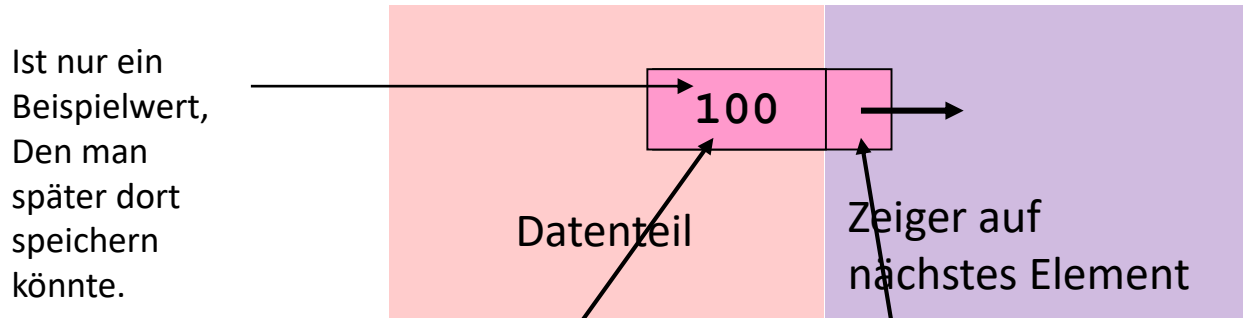


```
struct Listenelement {  
    //Element der Struktur  
    <Datenteil>;  
    //Verbindung zum nächsten Element  
    struct Listenelement *nachfolger;  
};
```

Muss nicht  
„Listenelement“ heißen,  
kann auch anders heißen!

```
//mit folgendem Befehl kann ab jetzt statt  
//struct Listenelement einfach nur Liste geschrieben werden  
typedef struct Listenelement Liste;
```

# Listen: Beispiel - Definition der Datenstruktur



```
struct Listenelement {  
    //Element der Struktur  
    int wert;  
    //Verbindung zum nächsten Element  
    struct Listenelement *nachfolger;  
};  
typedef struct Listenelement Liste;
```

# Liste: Funktionsbeispiele für die Arbeit mit Listen – Teil 1

In den restlichen Folien dieses Foliensatzes werden verschiedene Funktionen definiert, die nützlich bei der Arbeit mit Listen sein können. Dabei wird sich beispielhaft auf eine Liste bezogen, deren Listenelemente alle simpel wie folgt aussehen:

```
struct Listenelement {  
    int wert;  
    struct Listenelement *nachfolger;  
};  
typedef struct Listenelement Liste;
```

Die Funktionen sollen verdeutlichen, wozu Listen nützlich sind, was man mit ihnen so machen kann usw.. Die Funktionen selbst gehören also nicht zum Lernstoff, sondern sollen nur die Handhabung von Listen erklären.

# Liste: Funktionsbeispiele für Listen – Teil 2

Folgende Funktionen werden nun definiert:

- Funktion zum Anlegen der Liste, definiert wie in vorhergehender Folie:  
**Liste \*anlegenListe()**  
→ Zurückgegeben wird die Adresse, wo die Liste beginnt.
- Funktion zum Ausgeben der Liste:  
**void ausgebenListe(Liste \*liste)**  
→ der Funktion wird die Adresse übergeben, wo die Liste beginnt.
- Funktion zum Einfügen eines Listenelementes in unsere Liste:  
**void einfuegendavorinListe(Liste \*liste, int position, int neuerWert)**  
→ der Funktion wird die Adresse übergeben, wo die Liste beginnt, die Position, wovor das neue Listenelement eingefügt werden soll und der neue Wert, den das neue Listenelement haben soll.
- Funktion zum Löschen eines Listenelementes in unsere Liste:  
**void loescheninListe(Liste \*liste, int position)**  
→ der Funktion wird die Adresse übergeben, wo die Liste beginnt und die Position des Listenelementes, das gelöscht werden soll.

# Liste: Funktionsbeispiele für Listen – Teil 3

In allen diesen Funktionen wird die Adresse des ersten Listenelements übergeben.

**Achtung:** Der Einfachheit halber wird das erste Listenelement immer übersprungen und als Dummy-Kopf behandelt, so wie „Max Mustermann“ bei Formularen ein Dummy-Eintrag ist. Die Liste beginnt dann also immer beim **zweiten Listenelement**.

# Liste: Beispiel – Funktion fürs Anlegen der Liste

```
Liste *anlegenListe() {...}
```

→ Die Adresse (vom Typ **Liste**, also der Zeiger zeigt auf einen Platz, wo etwas vom Typ **Liste** gespeichert ist), wo die Liste beginnt, wird zurückgegeben.

Bemerkung zu nächster Folie: Normalerweise sollte hier nur ein erstes Listenelement zum „Aufhängen“ weiterer Elemente angelegt werden, aber wegen der besseren Verständlichkeit lege ich hier mal ausnahmsweise 3 Elemente an.





# Liste: Anlegen – Speicher für Elemente anfordern

```
Liste *anlegenListe(){  
    struct Listenelement *e0 = malloc(sizeof(Liste));  
    Liste *e1 = malloc(sizeof(Liste));  
    Liste *e2 = malloc(sizeof(Liste));  
    e0->wert = 0;//interpretiert als Kopf  
    e0->nachfolger = e1;  
    e1->wert = 1;  
    e1->nachfolger = e2;  
    e2->wert = 2;  
    e2->nachfolger = NULL;  
    return e0;  
}
```

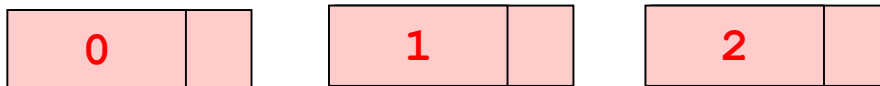
} Speicherplatz bereit stellen



Bemerkung: `Liste *liste  $\triangleq$  struct Listenelement *liste`

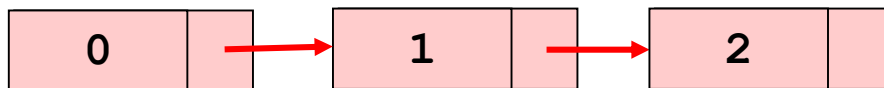
# Liste: Anlegen – Datenteile der Elemente füllen

```
Liste *anlegenListe(){
    Liste *e0 = malloc(sizeof(Liste));
    Liste *e1 = malloc(sizeof(Liste));
    Liste *e2 = malloc(sizeof(Liste));
    e0->wert = 0; //interpretiert als Kopf
    e0->nachfolger = e1;
    e1->wert = 1;
    e1->nachfolger = e2;
    e2->wert = 2;
    e2->nachfolger = NULL;
    return e0;
}
```



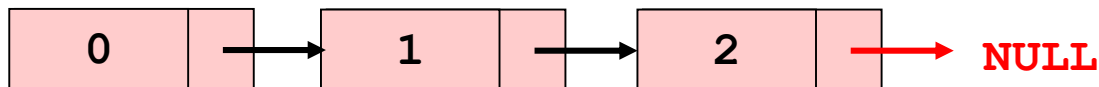
# Liste: Anlegen – Aneinanderketten der Elemente

```
Liste *anlegenListe() {  
    Liste *e0 = malloc(sizeof(Liste));  
    Liste *e1 = malloc(sizeof(Liste));  
    Liste *e2 = malloc(sizeof(Liste));  
    e0->wert = 0; //interpretiert als Kopf  
    e0->nachfolger = e1;  
    e1->wert = 1;  
    e1->nachfolger = e2;  
    e2->wert = 2;  
    e2->nachfolger = NULL;  
    return e0;  
}
```



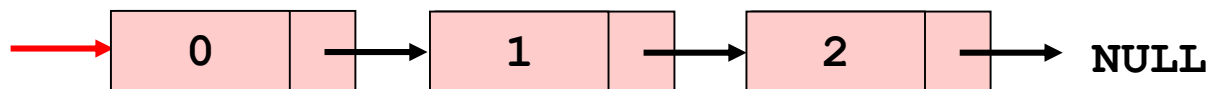
# Liste: Anlegen - Endes festlegen

```
Liste *anlegenListe(){  
    Liste *e0 = malloc(sizeof(Liste));  
    Liste *e1 = malloc(sizeof(Liste));  
    Liste *e2 = malloc(sizeof(Liste));  
    e0->wert = 0;//interpretiert als Kopf  
    e0->nachfolger = e1;  
    e1->wert = 1;  
    e1->nachfolger = e2;  
    e2->wert = 2;  
    e2->nachfolger = NULL;  
    return e0;  
}
```



# Liste: Anlegen – Funktionsrückgabe: Adresse, wo die Liste beginnt

```
Liste *anlegenListe() {  
    Liste *e0 = malloc(sizeof(Liste));  
    Liste *e1 = malloc(sizeof(Liste));  
    Liste *e2 = malloc(sizeof(Liste));  
    e0->wert = 0; //interpretiert als Kopf  
    e0->nachfolger = e1;  
    e1->wert = 1;  
    e1->nachfolger = e2;  
    e2->wert = 2;  
    e2->nachfolger = NULL;  
    return e0;  
}
```



# Liste: Ausgabe einer Liste

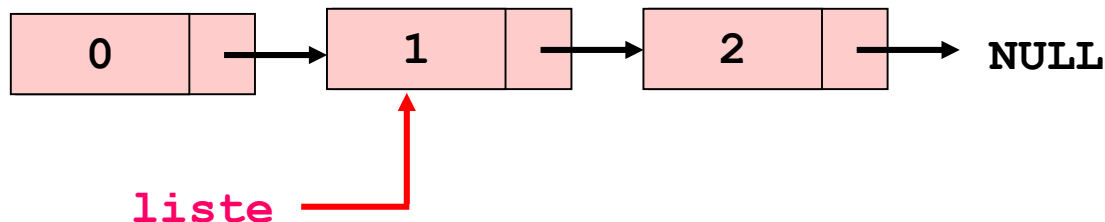
Funktion zum Ausgeben der Liste:

```
void ausgebenListe(Liste *liste)
```

Der Funktion wird die Adresse übergeben, wo die Liste beginnt.

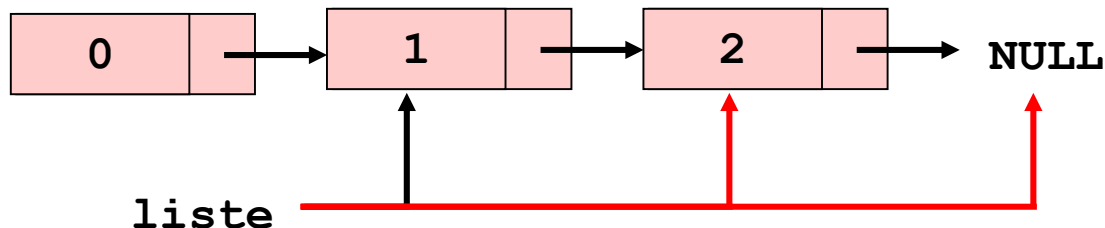
## Liste: Ausgabe als Funktion (erstes (Dummy-) Element wird überspringen)

```
void ausgebenListe(Liste *liste){//≙struct Listenelement *liste
    liste = liste->nachfolger;//Ueberspringen 1.Element
    while(liste != NULL)    {//solange man nicht am Ende ist
        printf("Element: %i \n",liste->wert);
        //huepfen von Element zu Element
        liste = liste->nachfolger;
    }
}
```



# Liste: Ausgabe - in der Liste voranschreiten

```
void ausgebenListe(Liste *liste){  
    liste = liste->nachfolger;//Kopf, wird uebersprungen  
    while(liste != NULL)    {  
        //Datenteil des Listenelementes ausgeben  
        printf("Element: %i\n",liste->wert);  
        liste = liste->nachfolger;  
    }  
}
```





# Liste: Einfügen eines Elements in eine Liste als Funktion

Ein neues Listenelement wird **vor** das Listenelement mit der Position **position** in der Liste eingefügt. **neuerWert** ist der neue Datenteil des neuen Listenelements. Wie immer wird die Anfangsadresse der Liste übergeben, damit die Funktion weiß, **in welcher** Liste das neue Listenelement eingefügt werden soll:

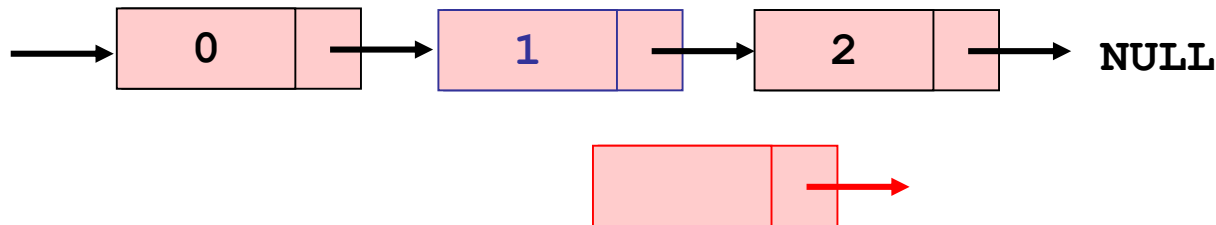
```
void einfuegendavorinListe(Liste *liste, int position, int neuerWert)
```

Siehe Video: <https://www.youtube.com/watch?v=p8ZLiDG1WnY>



# Liste: Einfügen - Speicher anfordern

```
void einfuegendavorinListe(Liste *liste, int position, int neuerWert){
    Liste *neuesElement;
    int i=1;
    while (liste != NULL){
        if(i == position){
            neuesElement = malloc(sizeof(Liste));
            neuesElement->wert = neuerWert;
            neuesElement->nachfolger = liste->nachfolger;
            liste->nachfolger = neuesElement;
        }
        liste=liste->nachfolger;
        i=i+1;
    }
}
```

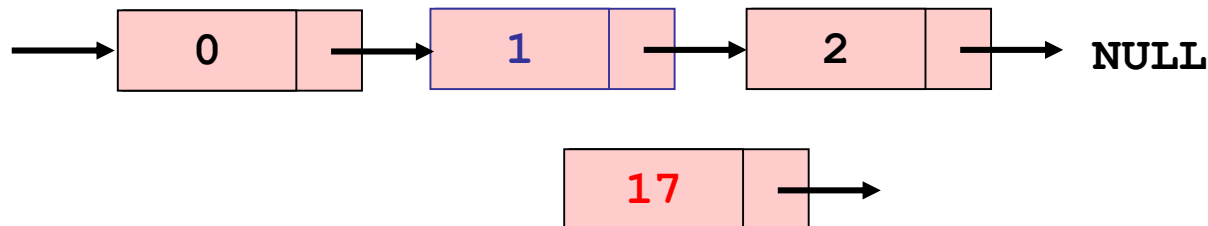


Beispielaufruf: `einfuegendavorinListe(zahlenliste, 2, 17)`



# Liste: Einfügen - Datenteil füllen

```
void einfuegendavorinListe(Liste *liste, int position, int neuerWert){
    Liste *neuesElement;
    int i=1;
    while (liste != NULL){
        if(i == position){
            neuesElement = malloc(sizeof(Liste));
            neuesElement->wert= neuerWert;
            neuesElement->nachfolger = liste->nachfolger;
            liste->nachfolger = neuesElement;
        }
        liste=liste->nachfolger;
        i=i+1;
    }
}
```

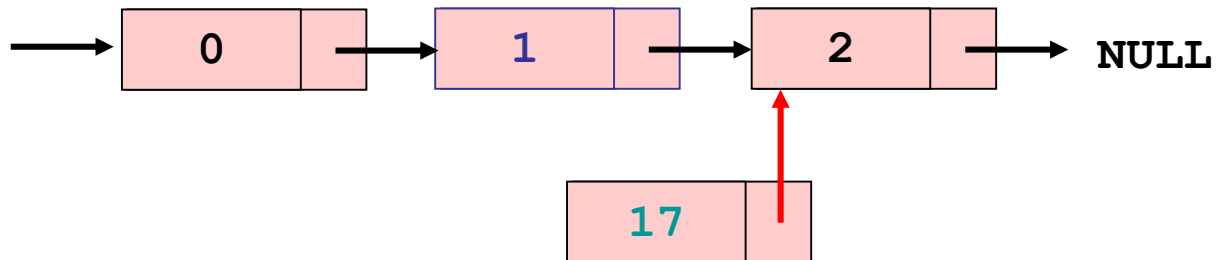


Beispielaufruf: `einfuegendavorinListe(zahlenliste, 2, 17)`



# Liste: Einfügen - Zeiger auf Nachfolger

```
void einfuegendavorinListe(Liste *liste, int position, int neuerWert){
    Liste *neuesElement;
    int i=1;
    while (liste != NULL){
        if(i == position){
            neuesElement = malloc(sizeof(Liste));
            neuesElement->wert= neuerWert;
            neuesElement->nachfolger = liste->nachfolger;
            liste->nachfolger = neuesElement;
        }
        liste=liste->nachfolger;
        i=i+1;
    }
}
```

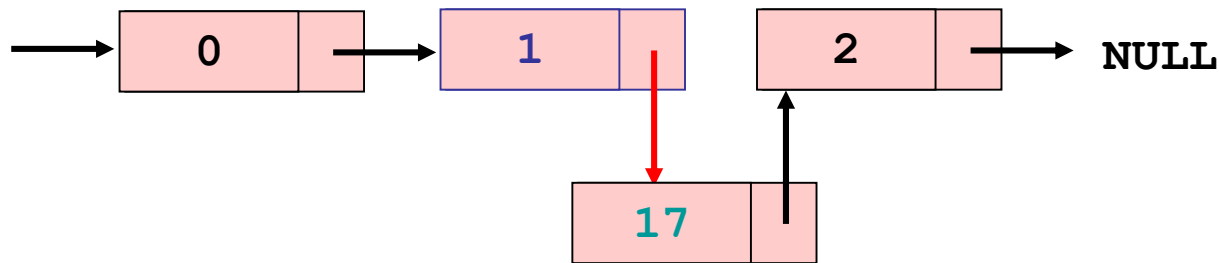


Beispielaufruf: `einfuegendavorinListe(zahlenliste, 2, 17)`



# Liste: Einfügen - Setzen des neuen Nachfolgers

```
void einfuegendavorinListe(Liste *liste, int position, int neuerWert){
    Liste *neuesElement;
    int i=1;
    while (liste != NULL){
        if(i == position){
            neuesElement = malloc(sizeof(Liste));
            neuesElement->wert= neuerWert;
            neuesElement->nachfolger = liste->nachfolger;
            liste->nachfolger = neuesElement;
        }
        liste=liste->nachfolger;
        i=i+1;
    }
}
```



Beispielaufruf: einfuegendavorinListe(zahlenliste, 2, 17)

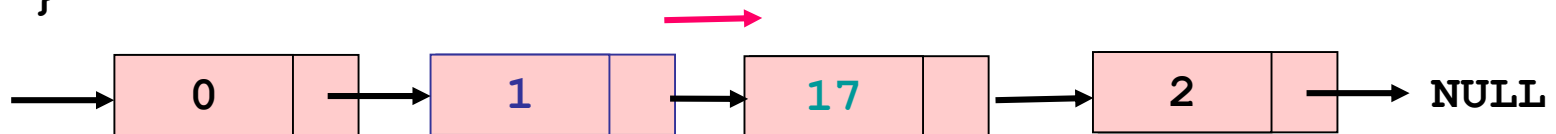
# Liste: Funktion fürs Löschen eines Listenelements

Das Listenelement mit der Position **position** wird gelöscht:

```
void loescheninListe(Liste *liste, int position)
```

# Liste: Löschen - Suche nach dem Listenelement

```
void loescheninListe(Liste *liste, int position){
    int i=1;
    Liste *hilf;
    while(liste != NULL){
        if(i == position){
            //Suche nach dem Listenelementes und merken
            //(Adresse speichern), wo es stand
            hilf=liste->nachfolger;
            liste->nachfolger=liste->nachfolger->nachfolger;
            free(hilf) ;
        }
        liste=liste->nachfolger;
        i=i+1;
    }
}
```

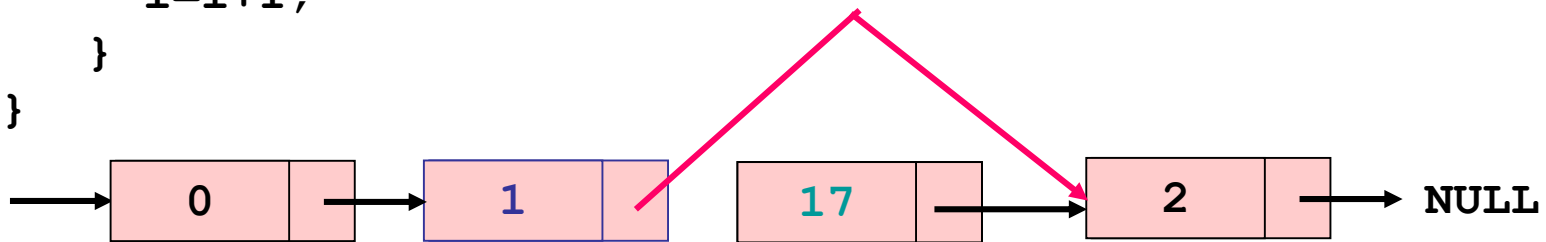


Beispielaufruf: `loescheninListe(zahlenliste, 2)`



# Liste: Löschen - Überspringen des Elements

```
void loescheninListe(Liste *liste, int position){  
    int i=1;  
    Liste *hilf;  
    while(liste != NULL){  
        if(i == position){  
            hilf=liste->nachfolger;  
            //Überspringen des Elementes, dass man löschen will  
            liste->nachfolger=liste->nachfolger->nachfolger;  
            free(hilf) ;  
        }  
        liste=liste->nachfolger;  
        i=i+1;  
    }  
}
```

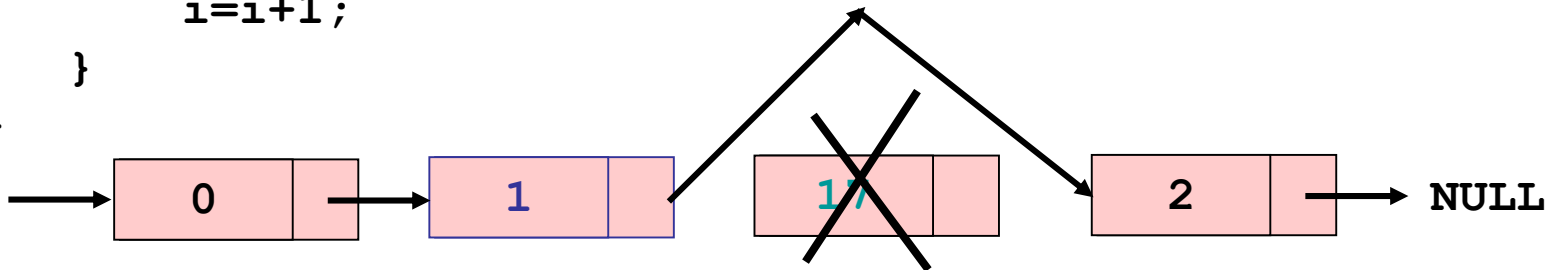


Beispielaufruf: `loescheninListe(zahlenliste, 2)`



# Liste: Löschen - Freigabe des Speichers

```
void loescheninListe(Liste *liste, int position){
    int i=1;
    Liste *hilf;
    while(liste != NULL){
        if(i == position){
            hilf=liste->nachfolger;
            liste->nachfolger=liste->nachfolger->nachfolger;
            //Freigabe des Speichers des Elementes, dass man
            //aus der Liste löschen will
            free(hilf);
        }
        liste=liste->nachfolger;
        i=i+1;
    }
}
```



Beispielaufruf: `loescheninListe(zahlenliste, 2)`

# Liste: Beispiel für ein Hauptprogramm

```
#include<stdlib.h>
#include<stdio.h>
//Definition der Liste
struct Listenelement {
    int wert;
    struct Listenelement *nachfolger;
};
typedef struct Listenelement Liste;
//Definition der Funktionen: anlegenListe, einfuegendavorinListe,
//loescheninListe, ausgebenListe
...
int main(){
    Liste *zahlenliste; //struct Listenelement *zahlenliste
    //Erzeugen der Liste zahlenliste;
    zahlenliste = anlegenListe();//mit 3 Element
    ausgebenListe(zahlenliste);
    //Einfuegen von 33 vor dem 2.Element
    einfuegendavorinListe(zahlenliste, 2, 33);
    //Loeschen des 3.Elements
    loescheninListe(zahlenliste,3);
    return 0;
}
```

Wer sich mehr hierfür interessiert, dem schicke ich gerne ein größeres Beispielprogramm (ListenBeispiel).



# Liste: Übung

Wie könnte eine doppelt verkettete Liste aussehen?

Achtung: auf der nächsten Folie ist die Lösung!



# Liste: Lösung

```
struct Studierender{  
    char name[20];  
    char vorname[20];  
    struct Studierender *vorgaenger;  
    struct Studierender *nachfolger;  
};
```

# Liste: Übung

Wo liegt das Problem?

```
void ausgabenvorletztesElement(Liste *liste){
    while(liste != NULL){
        if(liste -> nachfolger -> nachfolger == NULL){
            printf("vorletztes Element: %i \n",liste->wert);
        }
        liste = liste->nachfolger;
    }
}
```

Achtung: auf der nächsten Folie ist die Lösung!



# Liste: Lösung

```
void ausgabenvorletztesElement(Liste *liste){
    while(liste->nachfolger->nachfolger != NULL){
        liste = liste->nachfolger;
    }
    printf("vorletztes Element: %i\n",liste->wert);
}
```

```
void ausgabenvorletztesElement(Liste *liste){
    while(liste != NULL){
        //hier kracht es dann beim letzten Element
        //weil liste -> nachfolger -> nachfolger nicht
        //definiert ist
        if(liste -> nachfolger -> nachfolger == NULL){
            printf("vorletztes Element: %i \n",liste->wert);
        }
        liste = liste->nachfolger;
    }
}
```

Siehe Liste\_neu.c



# Listen: Wichtig

- Man weiß nie, wie lang eine Liste ist → deshalb immer mit **while**-Schleife und Test auf Ende (NULL) programmieren
- wie immer: wenn Zeiger benutzt werden, dürfen diese nicht auf unerlaubten Speicherplatz zeigen
- Listen liegen nicht unbedingt hintereinander im Speicher, in der Regel nicht. Die einzelnen Listenelemente sind quer über den ganzen Speicher des Rechners verteilt.

Siehe Liste\_neu.c



# Listen und Felder: Unterschied

	Felder	Listen
Flexibilität des Speichers	statisch	dynamisch 
Art der Speicherbelegung	zusammenhängend (auf dem Stack)	nicht notwendig zusammenhängend (finden dadurch eher Platz im Speicher, speziell Heap)
Zugriffszeit auf das i-te Element	konstant 	Erreichbar nur durch Durchlauf der ganzen Liste
Löschen und Einfügen eines Elements	aufwändig	einfach 



# Literatur:

- Jürgen Wolf: „C-Programmierung“, Markt + Technik Verlag