

Dynamische Speicherverwaltung

- Um Informationen im Speicher abzubilden, wurden bisher
 - Variablen
 - Arraysverwendet.
- Arrays haben eine feste Größe, es ist schwierig, Daten zu modellieren, wenn der Umfang bzw. die Anzahl der zu erwartenden Daten unbekannt ist.

Speicheraufteilung

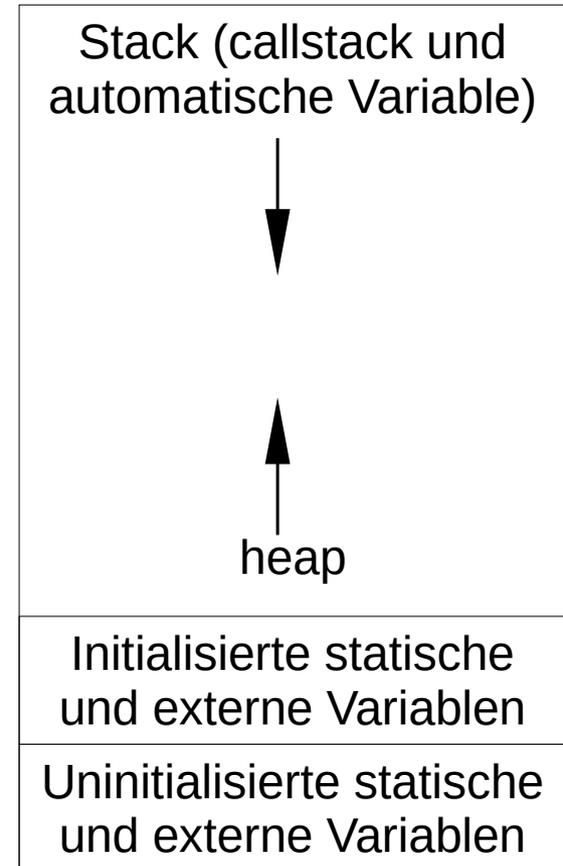
- Der Speicher für ein c-Programm ist in der Regel in verschiedene Segmente aufgeteilt.
 - Codesegment (Programmcode)
 - Stacksegment (lokale und statische Variable)
 - Datensegment (dynamischer Speicher oder heap)
- Je nach Plattform und Speichermodell können Segmente auch zusammengefasst sein, z.Bsp. Stack- und Datensegment.

heap

- Als heap (Halde) bezeichnet man einen Speicherbereich, aus dem man Speicher bedarfsgerecht entnehmen und wieder zurückgeben kann.
- Dazu bedient man sich der Funktionen malloc, free und realloc.

Speicheraufteilung

- In einer Konstellation, in der Stack und Datensegment zusammengefasst sind, wachsen Heap und Stack gegenläufig.
- Das Aufeinandertreffen der Speicherbereiche führt idealer Weise zu einer Exception und zu einem geordneten Programmabbruch.
- Das Verhalten ist plattformabhängig.



malloc

- Die Funktion malloc stellt bedarfsgerecht Speicher aus dem Heap zur Verfügung.
- Als Parameter erhält sie die Anzahl angeforderter Bytes, in der Regel wird diese Länge des angeforderten Bereiches unter Nutzung von sizeof ermittelt.
- malloc gibt einen **generischen Pointer**, vom Typ void* zurück.
- Ist nicht genügend Speicher vorhanden, so gibt malloc NULL (einen Nullpointer) zurück.

void*

- void* ist, wie es das Sternchen andeutet, ein Pointertyp.
- Der generische Pointer void* ist mit keinem Datentyp assoziiert, deshalb kann er auch nicht dereferenziert werden und es funktioniert keine Pointerarithmetik mit void*.
- Der generische Pointer ist aber mit jedem getypten Pointer zuweisungskompatibel.

malloc/realloc

- malloc stellt im Erfolgsfall den angeforderten Speicher zur Verfügung und gibt einen Pointer darauf zurück.
- realloc ermöglicht es, einen mit malloc allokierten Speicherbereich in der Größe zu verändern.

```
void *realloc(void *ptr, size_t size);
```

- Dabei wird sichergestellt, dass die Inhalte des alten Speicherbereiches (über ptr übermittelt) in den neuen Speicherbereich, dessen Adresse via return als Pointer zurückgegeben wird, übernommen werden.
- Als Sonderfall kann der Parameter ptr auch NULL sein, dann entspricht die Funktionalität der der Funktion malloc.

Beispiel 1

```
24 tStud getStudent()
25 {
26     tStud st;
27     printf("%-10s: ", "Name");
28     fgets(buf, 32, stdin);
29     buf[strlen(buf)-1]=0;
30     strcpy(st.name, buf);
31     printf("%-10s: ", "MatrikelNr");
32     fgets(buf, 32, stdin);
33     st.matrNr=atoi(buf);
34     st.noteKl=0;
35     st.noteBel=0;
36     return st;
37 }
```

- Das Beispiel Studentenverwaltung aus dem Kapitel Strukturen/benutzerdefinierte Datentypen wird um eine Funktion getStudent erweitert.
- In der Funktion wird ein Student erfasst und als Wert an den Aufrufer übergeben.

```

39 int main(int argc, char** argv)
40 {
41     char proceed='j';
42     tStud *pdata=NULL;
43     tStud *ptmp;
44     tStud *pstud=NULL;
45     int count=0;
46     int i;
47     do
48     {
49         ptmp=realloc(pdata,sizeof(tStud)*(count+1));
50         if(ptmp)
51         {
52             pdata=ptmp;
53             pdata[count]=getStudent();
54             count++;
55         }
56         printf("Weitere Eingabe von Daten (j/n): ");
57         fgets(buf,128,stdin);
58
59         proceed=buf[0];
60     }
61     while (proceed=='j');
62     for (i=0; i<count;i++) displayStudent(pdata+i);
63
64     free (pdata);
65
66     return 0;
67 }

```

Beim ersten Durchlauf NULL

In der Version, In der ein Pointer Übergeben wird.

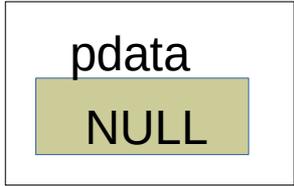
Über realloc wird der Speicher für zunächst einen Studenten bereitgestellt. Er nimmt einen Studenten, erfasst durch get-Student, auf. Der Speicherbereich wird mit jedem Durchlauf um Platz für einen Studenten erweitert.

Am Programmende wird der gesamte Speicher für die Studenten wieder frei gegeben (free (pdata);)

Die Funktion displayStudent wurde im Kapitel Strukturen eingeführt und wird hier unverändert übernommen.

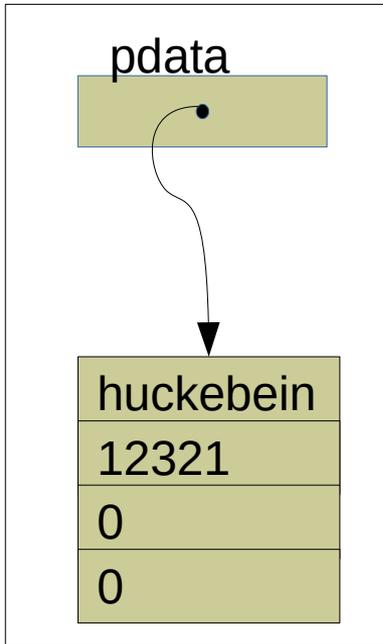
- In Zeile 49 wird der Pointer auf den neu bereitgestellten Speicher zunächst auf ptmp übernommen, da realloc, wenn es nicht erfolgreich ist, NULL zurückgibt.
- Somit wäre der Pointer auf die Daten verdorben.
- Erst in Zeile 52 wird der Pointer nach pdata übernommen.
- In Zeile 53 wird ein neuer Datensatz erfasst und in den bereitgestellten Speicher übernommen (hineinkopiert).
- In Zeile 64 wird der Speicher wieder frei gegeben.

count=0

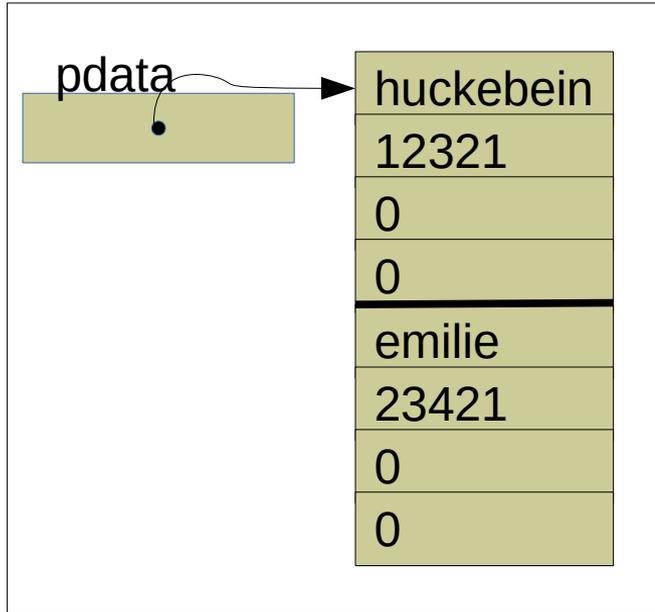


Darstellung der Situation mit 0, 1, 2 oder 3 erfassten Datensätzen

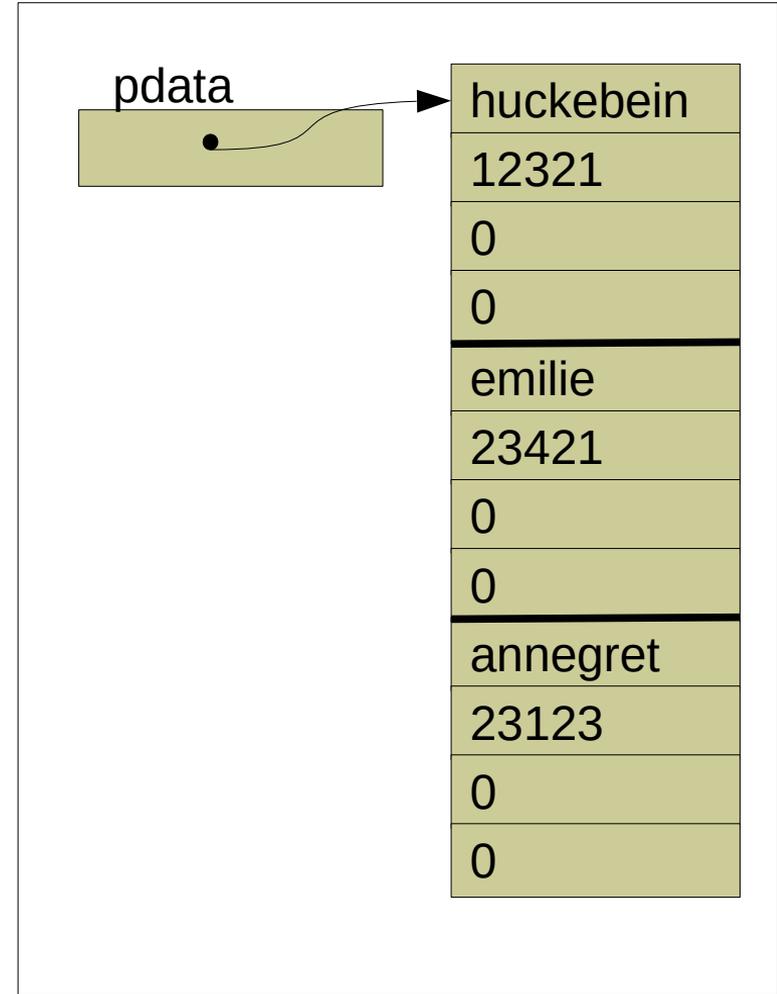
count=1



count=2



count=3



Kompletter Code Beispiel 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  char buf[128];
6
7  typedef struct
8  {
9      char name[30+1];
10     int matrNr;
11     float noteKl;
12     int noteBel;
13 }tStud;
14
15 void displayStudent(tStud* s)
16 {
17     printf("%-30s, %6d, %3.1f, %d\n",
18           s->name,
19           s->matrNr,
20           s->noteKl,
21           s->noteBel);
22 }
23
24 tStud getStudent()
25 {
26     tStud st;
27     printf("%-10s: ", "Name");
28     fgets(buf, 32, stdin);
29     buf[strlen(buf)-1]=0;
30     strcpy(st.name, buf);
31     printf("%-10s: ", "MatrikelNr");
32     fgets(buf, 32, stdin);
33     st.matrNr=atoi(buf);
34     st.noteKl=0;
35     st.noteBel=0;
36     return st;
37 }
```

```
38
39 int main(int argc, char** argv)
40 {
41     char proceed='j';
42     tStud *pdata=NULL;
43     tStud *ptmp;
44     tStud *pstud=NULL;
45     int count=0;
46     int i;
47     do
48     {
49         ptmp=realloc(pdata, sizeof(tStud)*(count+1));
50         if(ptmp)
51         {
52             pdata=ptmp;
53             pdata[count]=getStudent();
54             count++;
55         }
56         printf("Weitere Eingabe von Daten (j/n): ");
57         fgets(buf, 128, stdin);
58
59         proceed=buf[0];
60     }
61     while (proceed=='j');
62     for (i=0; i<count; i++) displayStudent(pdata+i);
63
64     free (pdata);
65
66     return 0;
67 }
68
```

```
$. /a.out
Name      : huckebein
MatrikelNr: 12321
Weitere Eingabe von Daten (j/n): j
Name      : emilie
MatrikelNr: 23421
Weitere Eingabe von Daten (j/n): j
Name      : annegret
MatrikelNr: 23123
Weitere Eingabe von Daten (j/n): n
huckebein      , 12321, 0.0, 0
emilie         , 23421, 0.0, 0
Annegret       , 23123, 0.0, 0
$
```

Beispiel 2

```
24 tStud* getStudent()
25 {
26     tStud* ps=malloc(sizeof(tStud));
27     if (ps)
28     {
29         printf("%-10s: ", "Name");
30         fgets(buf,32,stdin);
31         buf[strlen(buf)-1]=0;
32         strcpy(ps->name,buf);
33         printf("%-10s: ", "MatrikelNr");
34         fgets(buf,32,stdin);
35         ps->matrNr=atoi(buf);
36         ps->noteKl=0;
37         ps->noteBel=0;
38     }
39     return ps;
40 }
```

Bereitstellen
von Speicher
für einen Student

Speicher wurde
erfolgreich
bereitgestellt
(ps!=NULL)

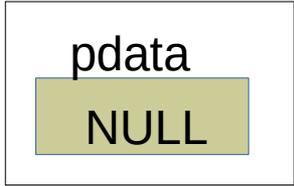
Pointer auf den
neuen Studenten oder NULL
wird zurückgegeben

- Es wird am Anfang Speicher für ein Datenobjekt des Typs tStud bereitgestellt.
- In Zeile 31 wird das abschließende '\n' entfernt, bevor der Name in die Struktur übernommen wird.

Beispiel 2

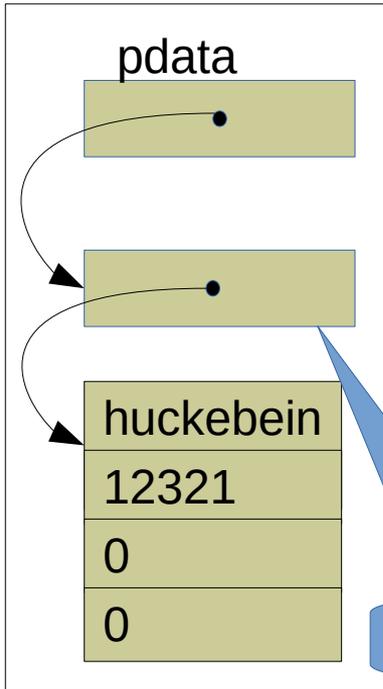
- Der Student wird in Beispiel 2 sofort in einem eigenen Speicherbereich erfasst. Die Variable `st` in `getStudent` entfällt.
- Es wird nur noch ein Pointer zurückgegeben und in `main` zugewiesen (viel effizienter).
- In Beispiel 2 wird ein Pointerarray eingeführt, das ebenfalls dynamisch erzeugt wird und mittels `realloc` „wächst“.
- Die Studenten können nun irgendwo im Speicher gestreut liegen, Sie werden über das Pointerarray verwaltet.
- Will man die Daten sortieren, so müssen nur die Pointer sortiert werden, was viel effizienter ist, als die Daten selbst zu sortieren.

count=0

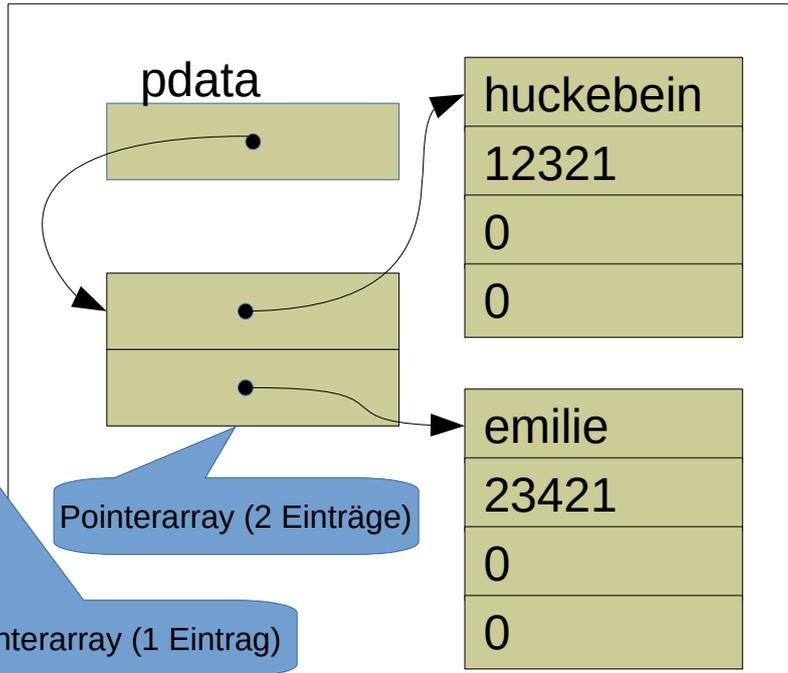


Darstellung der Situation Beispiel 2 mit 0, 1, 2 oder 3 erfassten Datensätzen. Die Datensätze selbst stehen gestreut, irgendwo im heap.

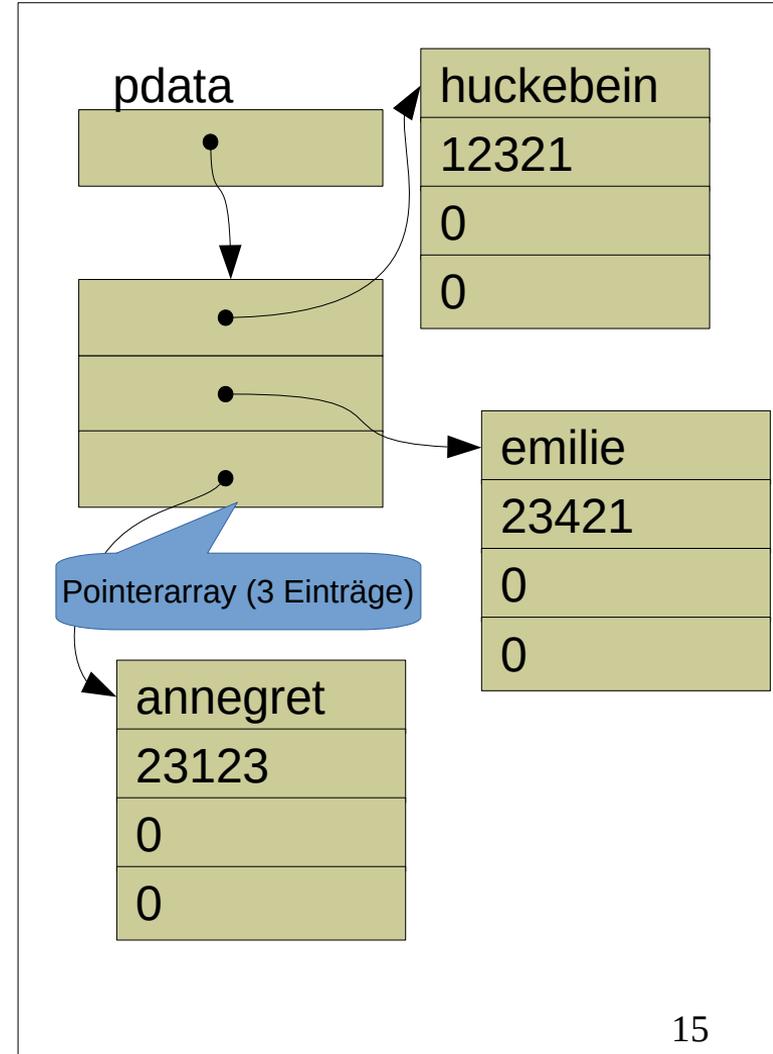
count=1



count=2



count=3



Beispiel

- Zeile 45: hier haben wir einen Pointer, der auf Pointer vom Typ tStud zeigt.
- Zeile 52: Ein Student wird erfasst. Falls malloc nicht erfolgreich war, wird NULL zurück gegeben (Zeile 53)
- Zeile 55: Hier wird das Pointerarray um einen weiteren Pointer vergrößert.

```
42 int main(int argc, char** argv)
43 {
44     char proceed='j';
45     tStud **pdata=NULL;
46     tStud **ptmp;
47     tStud * pstud=NULL;
48     int count=0;
49     int i;
50     do
51     {
52         pstud=getStudent();
53         if (pstud)
54         {
55             ptmp=realloc(pdata,sizeof(tStud)*(count+1));
56             if(ptmp)
57             {
58                 pdata=ptmp;
59                 pdata[count]=pstud;
60                 count++;
61             }
62             else {perror(NULL);}
63         }else {perror(NULL);}
64         printf("Weitere Eingabe von Daten (j/n): ");
65         fgets(buf,128,stdin);
66
67         proceed=buf[0];
68     }
69     while (proceed=='j');
70     for (i=0; i<count;i++) displayStudent(pdata[i]);
71
72     return 0;
73 }
```

Seite davor!

Student erfolgreich Erfasst

Erläuterungen zum Beispiel

- Im Beispiel wird für jeden Student Speicher allokiert.
- Bei jedem Schleifendurchlauf wird `pdata` mit `realloc` um einen Pointer vergrößert, da man nicht weiß, ob das erfolgreich ist, wird erst mal auf `ptmp` der neue Pointer zwischengespeichert (Zeile 55), in Zeile 58 wird der neue Pointer dann wieder nach `pdata` übernommen.
- `realloc` weist neuen Speicher zu und übernimmt die Inhalte von der Quelle in den neuen Speicher.
- Nach jedem `malloc` oder `realloc` muss der Erfolg der Operation verifiziert werden (Zeilen 27, 53, 56).

Erläuterungen zum Beispiel

- Der Pointer auf den jeweils neuen Studenten wird als letzter Eintrag in das Pointerarray übernommen.
- Auf diese Weise können ‚beliebig‘ viele Einträge erfasst werden.

Freigabe von Speicher

- Wie alles, was man sich borgt (Bücher, Autos, ...) muss man auch den Speicher am Ende zurück geben.
- Dazu benutzt man `free`. Diese Funktion bekommt den Pointer, den man via `malloc` oder `realloc` erhalten hat, als Parameter.

```
70     for (i=0; i<count;i++) displayStudent(pdata[i]);
71
72     for (i=0; i<count;i++) free(pdata[i]);
73     free (pdata);
74
75     return 0;
76 }
77
```

Freigabe eines jeden Studenten

Freigabe des Pointerarrays

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  char buf[128];
6
7  typedef struct
8  {
9      char name[30+1];
10     int matrNr;
11     float noteKl;
12     int noteBel;
13 }tStud;
14
15 void displayStudent(tStud* s)
16 {
17     printf("%-30s, %6d, %3.1f, %d\n",
18           s->name,
19           s->matrNr,
20           s->noteKl,
21           s->noteBel);
22 }
23
24 tStud* getStudent()
25 {
26     tStud* ps=malloc(sizeof(tStud));
27     if (ps)
28     {
29         printf("%-10s: ", "Name");
30         fgets(buf,32,stdin);
31         buf[strlen(buf)-1]=0;
32         strcpy(ps->name,buf);
33         printf("%-10s: ", "MatrikelNr");
34         fgets(buf,32,stdin);
35         ps->matrNr=atoi(buf);
36         ps->noteKl=0;
37         ps->noteBel=0;
38     }
39     return ps;
40 }

```

```

42 int main(int argc, char** argv)
43 {
44     char proceed='j';
45     tStud **pdata=NULL;
46     tStud **ptmp;
47     tStud * pstud=NULL;
48     int count=0;
49     int i;
50     do
51     {
52         pstud=getStudent();
53         if (pstud)
54         {
55             ptmp=realloc(pdata,sizeof(tStud)*(count+1));
56             if(ptmp)
57             {
58                 pdata=ptmp;
59                 pdata[count]=pstud;
60                 count++;
61             }
62             else {perror(NULL);}
63         }else {perror(NULL);}
64         printf("Weitere Eingabe von Daten (j/n): ");
65         fgets(buf,128,stdin);
66
67         proceed=buf[0];
68     }
69     while (proceed=='j');
70     for (i=0; i<count;i++) displayStudent(pdata[i]);
71
72     for (i=0; i<count;i++) free(pdata[i]);
73     free(pdata);
74
75     return 0;
76 }

```

Verkettete Liste

- Im Folgenden versehen wir die Struktur mit einer zusätzlichen Komponente next.
- Sie ist ein Zeiger, der auf das nächste Strukturobjekt, also immer auf den nächsten Datensatz zeigt.
- Jeder Datensatz kann nun irgendwo im Speicher abgelegt sein.
- Man hangelt sich durch den Datenbestand von Datensatz zu Datensatz.

```

int main(int argc, char** argv)
{
    char proceed='j';
    tStud * pdata=NULL;
    tStud * ptmp;
    int count=0;
    int i;
    do
    {
        ptmp=getStudent();
        if (ptmp)
        {
            ptmp->next= pdata;
            pdata=ptmp;
        }else {perror(NULL);}
        printf("Weitere Eingabe von Daten (j/n): ");
        fgets(buf,128,stdin);

        proceed=buf[0];
    }
    while (proceed=='j');
    for (ptmp=pdata; ptmp ;ptmp=ptmp->next) displayStudent(ptmp);
    while(pdata) {ptmp=pdata->next; free(pdata); pdata=ptmp;}
    return 0;
}

```

```

$ ./a.out
Name      : Hans
MatrikelNr: 12
Weitere Eingabe von Daten (j/n): j
Name      : Anna
MatrikelNr: 23
Weitere Eingabe von Daten (j/n): j
Name      : Otto
MatrikelNr: 34
Weitere Eingabe von Daten (j/n): j
Name      : Henriette
MatrikelNr: 45
Weitere Eingabe von Daten (j/n): n
Henriette      ,      45, 0.0, 0
Otto            ,      34, 0.0, 0
Anna           ,      23, 0.0, 0
Hans           ,      12, 0.0, 0

```

Verkettete Liste

- Hier ist der einfachste Fall einer verketteten Liste implementiert.
- Eine Zeigervariable zeigt auf den ersten Datensatz.
- Jeder neue Datensatz wird unmittelbar hinter diesem Zeiger in die Liste eingefügt:

```
ptmp->next= pdata;  
pdata=ptmp;
```
- Diese Liste zeigt ein Stackverhalten, die Reihenfolge der Daten wird hier invertiert.

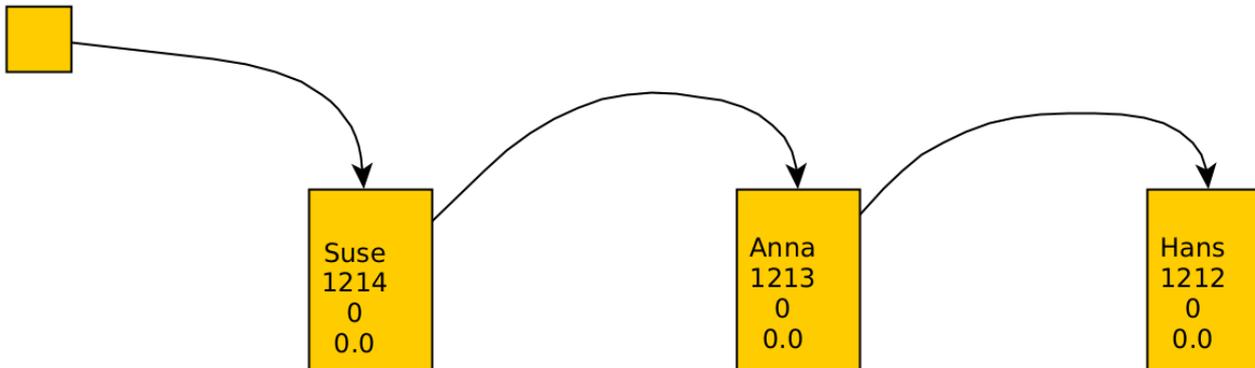
Daten mit einfach verketteter Liste

pData



Liste mit einem Datenelement

pData



Liste mit zwei weiteren Datenelementen

Verkettete Liste

- Die Daten werden genau verkehrt herum ausgegeben.
- Der untere Kasten zeigt die Ausgabe der Daten (erste Zeile)
- Und das Wiederfreigeben des ausgefassten Speichers

```
for (ptmp=pdata; ptmp ;ptmp=ptmp->next) displayStudent(ptmp);  
while(pdata) {ptmp=pdata->next; free(pdata); pdata=ptmp;}
```

Verkettete Liste

- Erweitert man das Ganze um einen weiteren Zeiger pLast, kann man eine Liste bauen, bei der jeder neue Datensatz hinten angefügt wird.
- Eine solche Liste zeigt das Verhalten einer Warteschlange.
- Hier werden die Daten in der Reihenfolge der Eingabe auch ausgegeben.

```

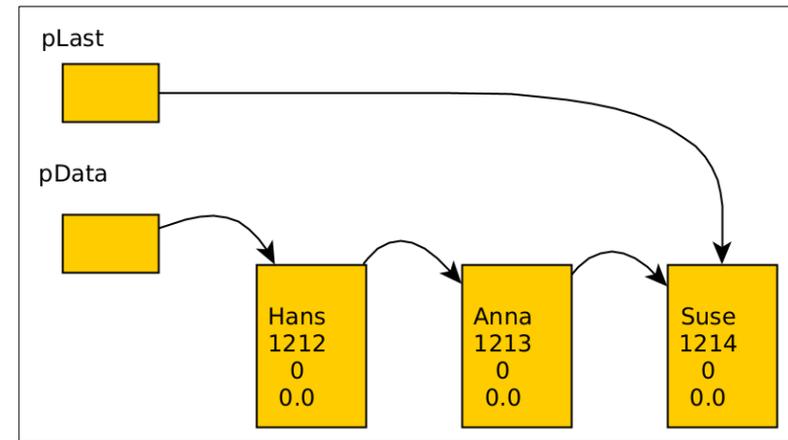
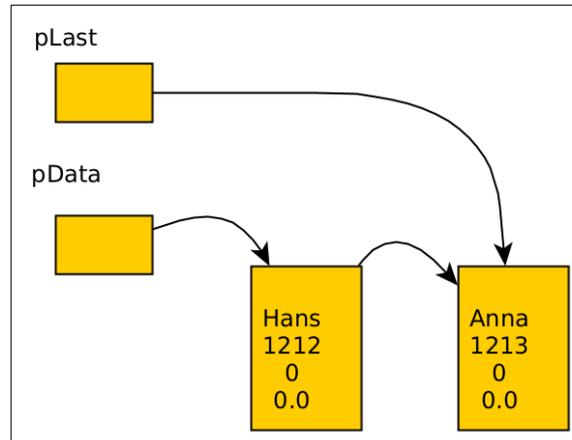
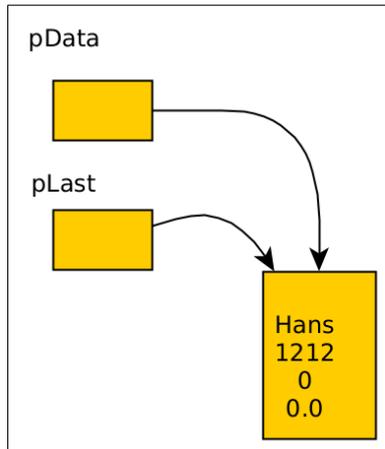
int main(int argc, char** argv)
{
    char proceed='j';
    tStud * pdata=NULL;
tStud * plast=NULL;
    tStud * ptmp;
    int count=0;
    int i;
    do
    {
        ptmp=getStudent();
        if (ptmp)
        {
            if (pdata==NULL) pdata=ptmp;
            if (plast)plast->next=ptmp;
            plast=ptmp;
        }else {perror(NULL);}
        printf("Weitere Eingabe von Daten (j/n): ");
        fgets(buf,128,stdin)
        proceed=buf[0];
    }
    while (proceed=='j');
    // . . . Ausgabe, wie vorher.
    return 0;
}

```

Daten mit Liste mit Last-Zeiger

Es wird immer am Listenende ein neuer Datensatz angefügt, in dem der Inhalt von pLast in den neuen Datensatz übernommen und pLast auf den neuen Datensatz gestellt wird.

Bei Aufnahme des ersten Datensatzes muss pData zusätzlich den Pointer des neuen Datensatzes bekommen.



Nicht intrusive Verkettete Listen

- Man unterscheidet zwischen intrusiven und nicht intrusiven Listen.
- Bei einer intrusiven Liste enthalten die durch die Liste verwalteten Datensätze selbst den/die Listenzeiger (vorhergehende Beispiele).
- Bei einer nicht intrusiven Liste gibt es Listenelemente, die auf das jeweils nächste Element und auf die Daten zeigen.

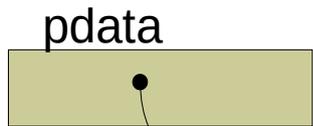
Nicht intrusive Verkettete Listen

- Bei den bisherigen Listen muss man die Listenfunktion (ein-/ausketten) für die jeweiligen Daten neu implementieren.
- Nicht intrusive Liste ermöglichen die Realisierung einer Listenlösung, die einmal programmiert, für beliebige Daten angewandt werden kann.
- Sie kann in eine speziellen Library oder einem Modul implementiert sein.

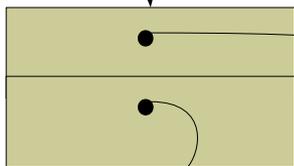
Nicht intrusive Verkettete Listen

- Ausgangspunkt ist hier das Pointerarray, das für jeden neuen Datensatz mit realloc vergrößert wurde.
- Wir wollen nun auf realloc verzichten, da es u.Ustd. recht aufwändig in der Ausführung ist.
- Dazu brechen wir das Pointerarray auf und versehen jeden der Pointer auf die Daten mit einem zweiten Pointer auf den nächsten.
- Wir schaffen so Verbindungselemente, die aus zwei Pointern bestehen.

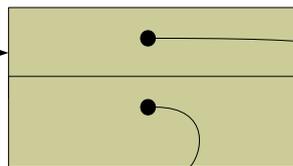
pdata



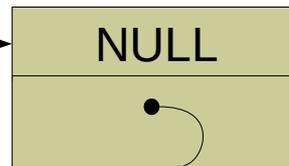
Einfach verkettete offene Liste



huckebein
12321
0
0



annegret
23123
0
0



emilie
23421
0
0

Verbindungselement
mit zwei Pointern

Einfach verkettete offene Liste

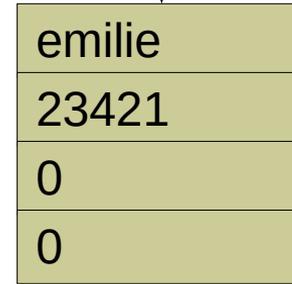
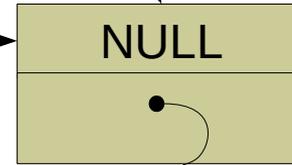
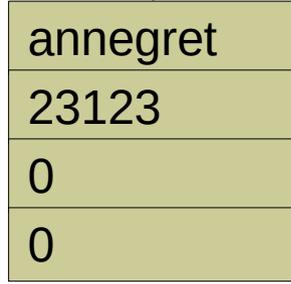
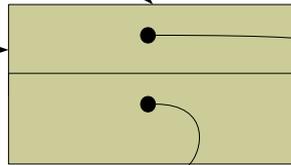
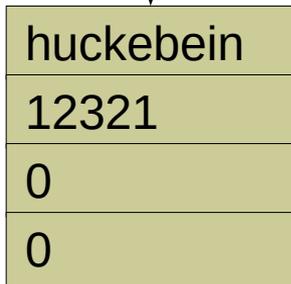
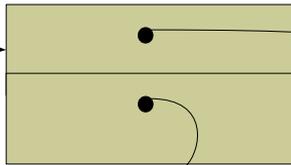
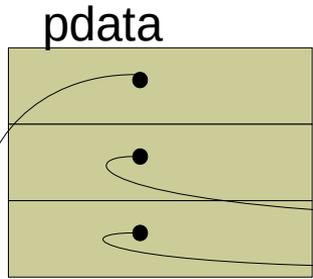
- Vielen Ansprüchen genügt diese Form von Listen.
- Das Verbindungselement besteht aus zwei Pointern
 - Pointer auf die Daten (void* , damit man die Liste, einmal programmiert immer wieder verwenden kann)
 - Pointer auf das nächste Verbindungselement
- Es kann leicht am Listenanfang ein Element eingefügt oder entfernt werden.
- Hilfreich kann eine Listenkopfstruktur sein:
 - Zeiger auf das erste Listenelement
 - Zeiger auf das letzte Listenelement, damit man auch leicht am Listenende anfügen kann.
 - Zeiger auf ein aktuelles Listenelement, damit man durch die Liste iterieren kann.

Einfach verkettete offene Liste

Zeiger aus das letzte Element,
Damit man auch am Listenende
Daten anfügen kann

pCurr ist NULL
oder zeigt auf ein
gerade aktuelles
Listenelement

Zeiger auf das
erste Listen-
element



Arbeit mit Listen

- Es ist sinnvoll, eine solche Listenimplementation als gesonderten c-Modul zu entwerfen.
- Er sollte in der Lage sein, beliebige Daten zu verwalten, dazu wird ein void*-Pointer als Zeiger auf die Daten verwendet.
- Nebenstehend ist das Headerfile eines Listmoduls für eine einfach verkettete, offene Liste zu sehen.
- Sind die Funktionen dazu einmal implementiert, kann man sie immer wieder zur Verwaltung beliebiger Daten einsetzen.

```
1 typedef struct tconnect
2 {
3     struct tconnect *pnxt;
4     void* pdata;
5 }tCnct;
6
7 typedef struct
8 {
9     tCnct* pfirst;
10    tCnct* plast;
11    tCnct* pcurr;
12 }tList;
13
14 tList* createList();
15 void* getFirst(tList* pList);
16 void* getNext(tList* pList);
17 int insFirst(tList* pList,void* pdata);
18 int insLast (tList* pList,void* pdata);
19 void removeFirst(tList* pList);
20 void deleteList(tList* pList);
```

Listenverbindungs-
element

Listenkopf

```

42 int main(int argc, char** argv)
43 {
44     char proceed='j';
45     tList *pList=createList();
46     tStud *pstud=NULL;
47     if(pList)
48     {
49         do
50         {
51             pstud=getStudent();
52             if (pstud)
53             {
54                 insLast(pList,pstud);
55             }else {perror(NULL);}
56
57             printf("Weitere Eingabe von Daten (j/n): ");
58             fgets(buf,128,stdin);
59             proceed=buf[0];
60         }
61         while (proceed=='j');
62         for (pstud=getFirst(pList); pstud; pstud=getNext(pList))
63             displayStudent(pstud);
64         deleteList(pList);
65     }
66     return 0;
67 }

```

Erzeugung der leeren Liste

- Die main-Funktion der Beiepielanwendung vereinfacht sich wesentlich.
- In Zeile 54 wird der neue Datensatz in die Liste aufgenommen.
- Zeilen 62/63 dienen der Ausgabe. Die for-Schleife durchläuft mit getNext die gesamte Liste. Es ist eine for-Schleife, die hier nicht als Zählschleife agiert.

```

1  #include <stdlib.h>
2  #include <malloc.h>
3  #include "list1.h"
4
5  tList* createList()
6  {
7      tList* ptmp;
8      ptmp=malloc(sizeof(tList));
9      if (ptmp)
10         ptmp->pfirst=ptmp->plast=ptmp->pcurr=NULL;
11     return ptmp;
12 }
13 void* getFirst(tList* pList)
14 {
15     void* ptmp=NULL;
16     if (pList->pfirst)
17     {
18         ptmp=pList->pfirst->pdata;
19         pList->pcurr=pList->pfirst;
20     }
21     return ptmp;
22 }
23 void* getNext(tList* pList)
24 {
25     void* ptmp=NULL;
26     if (pList->pcurr)
27     {
28         pList->pcurr=pList->pcurr->pnxt;
29         if (pList->pcurr)
30             ptmp=pList->pcurr->pdata;
31     }
32     return ptmp;
33 }
34 int insFirst(tList* pList, void* pdata)
35 {
36     tCnct *ptmp= malloc(sizeof(tCnct));
37     if (ptmp)
38     {
39         ptmp->pdata=pdata;
40         ptmp->pnxt =pList->pfirst;
41         pList->pfirst=ptmp;
42         if (pList->plast==NULL) pList->plast=ptmp;
43     }
44     return (int)(long)ptmp;
45 }

```

Implementation der Listenfunktionen einer einfachverketteten, offenen Liste

```

46 int insLast (tList* pList, void* pdata)
47 {
48     tCnct *ptmp= malloc(sizeof(tCnct));
49     if (ptmp)
50     {
51         ptmp->pdata=pdata;
52         ptmp->pnxt =NULL;
53         if (pList->pfirst==NULL) pList->pfirst=ptmp;
54         else pList->plast->pnxt=ptmp;
55         pList->plast=ptmp;
56     }
57     return (int)(long)ptmp;
58 }
59 void removeFirst(tList* pList)
60 {
61     if (pList->pfirst)
62     {
63         tCnct* tmp=pList->pfirst;
64         pList->pfirst=tmp->pnxt;
65         free(tmp);
66     }
67 }
68 void deleteList(tList* pList)
69 {
70     while (pList->pfirst!=NULL) removeFirst(pList);
71     free(pList);
72 }
73

```

Doppelt verkettete Liste

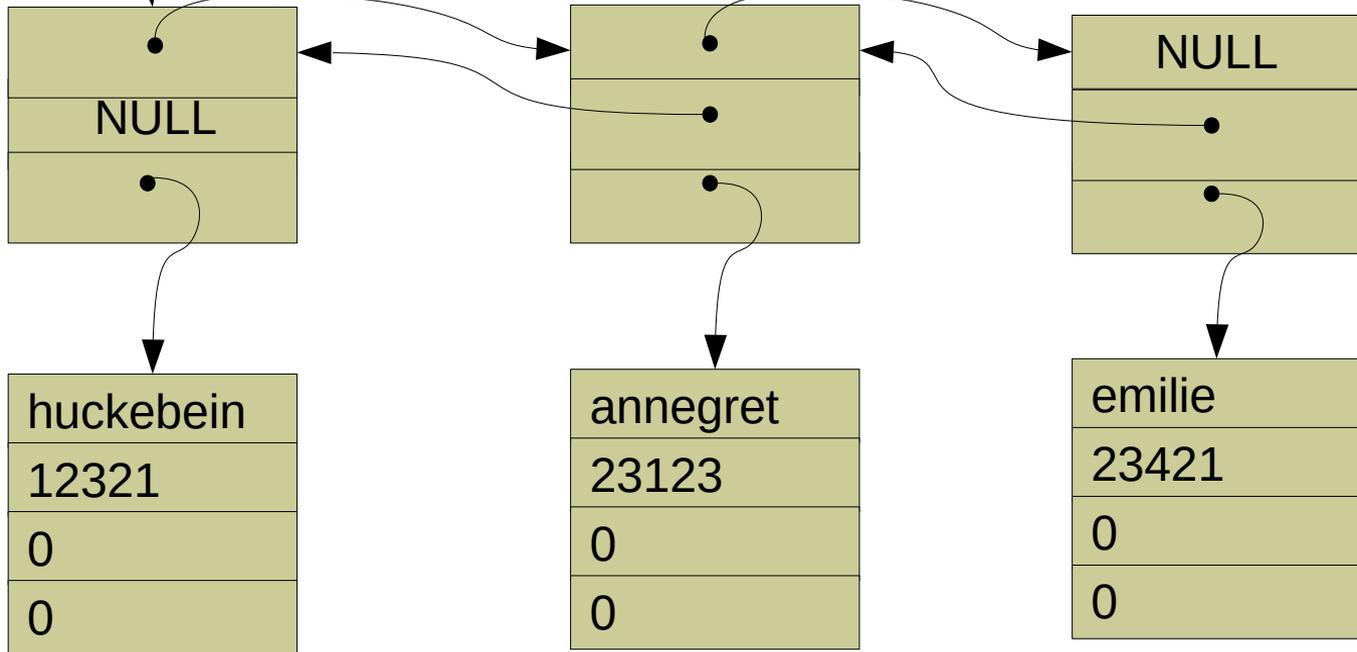
- Ein paar Wünsche lässt die bis hier betrachtete einfach verkettete Liste aber noch offen, man kann hinten oder irgendwo in der Mitte schlecht ausketten (Daten aus der Liste entfernen).
- Man kann ein Element an beliebiger Stelle schlecht einfügen.
- Beides wird mit doppelt verketteten Listen besser möglich.
- Dabei gibt es immer einen Zeiger auf das nachfolgende Listenelement und einen auf das vorhergehende.
- Das Ende der Vorwärts- und Rückwärtsverkettung ist durch einen Nullpointer gekennzeichnet.
- Diese Art der Listen, wie sie im Bild auf der folgenden Seite dargestellt sind, werden erst einmal nicht weiter verfolgt.

pdata



Doppelt verkettete offene Liste

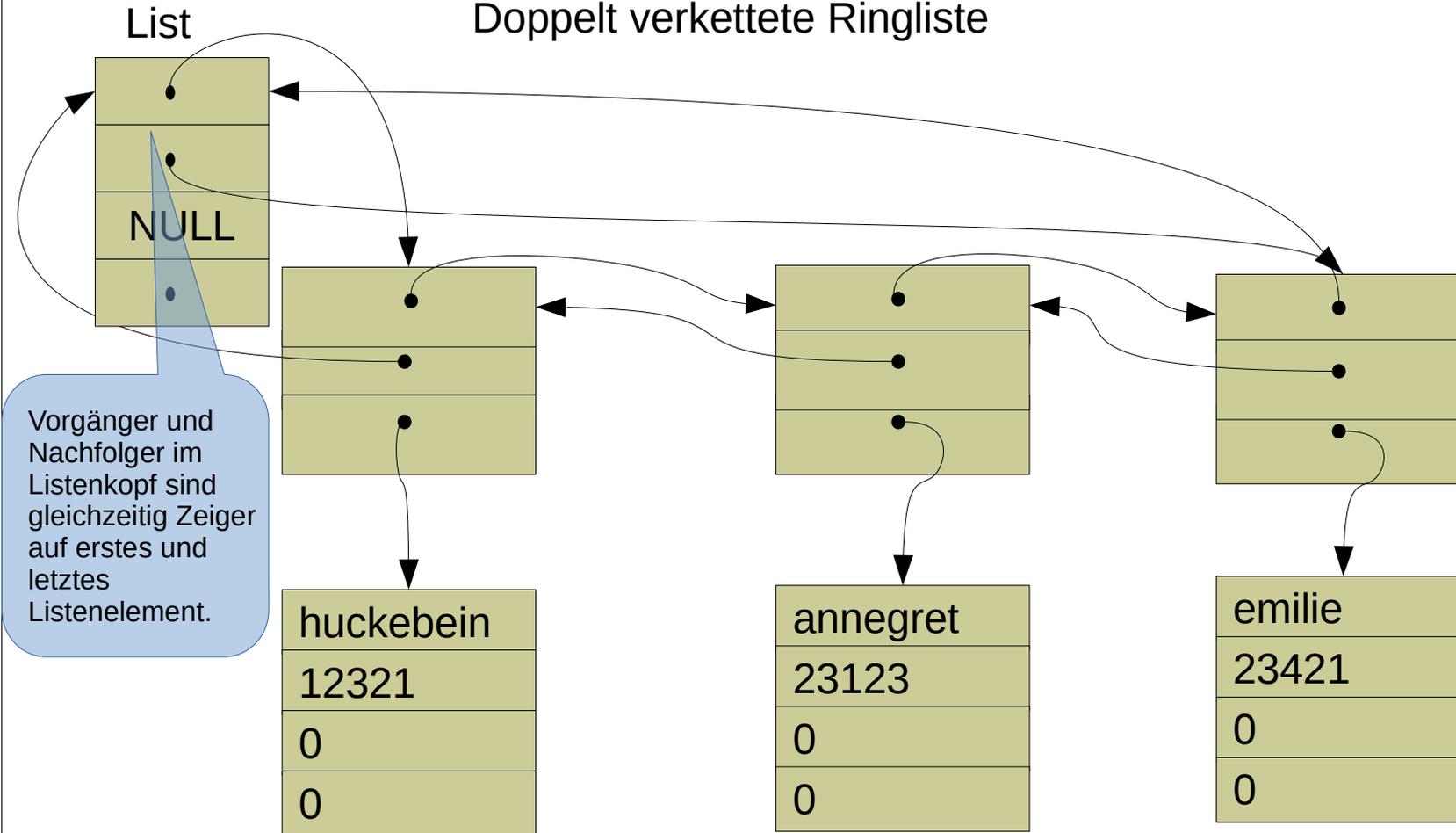
Verbindungselement
mit drei Pointern



Doppelt verkettete Ringliste

- Bei der Ringliste verweist das jeweils letzte Listenelement wieder auf das erste. Es entstehen zwei Ringe.
- Das wirkt zunächst kompliziert, macht aber die Programmierung der Liste viel einfacher, weil jedes Element in jeder Richtung einen Nachfolger und einen Vorgänger hat.
- Somit entfallen die Sonderfälle für das erste und letzte Element.
- Der Listenkopf selbst wird als Listenelement ohne Daten implementiert.

Doppelt verkettete Ringliste



- Das sieht sehr kompliziert aus.
- Der Listenkopf ist selbst ein Element, ergänzt um einen pCurrent-Zeiger.
- Der Zeiger auf die Daten bleibt NULL.
- Vorteil: Man braucht nur einmal das Ein- und Ausketten zu programmieren und deckt damit alle Fälle (einketten vorn, hinten, an beliebiger Stelle) ab.
- Diese Variante ist viel einfacher zu programmieren, als die offene Liste.

Der tatsächliche Aufbau
der Datenstrukturen
bleibt hier dem Anwender
Verborgен.

Headerfile für einen
Modul Doppelt
verkettete Ringliste

```
1  /* list.h */
2
3  #define OK 1
4  #define FAIL 0
5  /*-----*/
6  /* Prototypen fuer die Funktionen */
7
8  struct tlst;»      »      »      // Forewarddeclaration
9  typedef struct tlst tList;
10
11  tList * CreateList(void);          /* erzeuge leere Liste */
12  int   DeleteList(tList* pList);    /* loesche leere Liste */
13
14  int   InsertBehind (tList* pList, void *pItemIns);/* fuege ein hinter % */
15  int   InsertBefore (tList* pList, void *pItemIns);/* fuege ein vor  % */
16  int   InsertHead   (tList* pList, void *pItemIns);/* fuege vorn ein  */
17  int   InsertTail   (tList* pList, void *pItemIns);/* fuege hinten ein */
18  int   RemoveItem   (tList* pList);          /* loesche % */
19
20  void* GetSelected   (tList* pList);          /* gib aktuellen DS */
21  void* GetFirst      (tList* pList);          /* gib ersten DS */
22  void* GetLast       (tList* pList);          /* gib letzten DS */
23  void* GetNext       (tList* pList);          /* gib naechsten DS */
24  void* GetPrev       (tList* pList);          /* gib vorigen DS */
25  void* GetIndexed    (tList* pList,int Idx);  /* gib DS lt. Index */
26
27  void* addItemToList (tList* pList,
28                      void * pItem,
29                      int(*fcmp)(void*pItList,void*pItNew));
30
31  /* % steht fuer aktuellen Satz */
```