Dynamische Speicherverwaltung

- Um Informationen im Speicher abzubilden, wurden bisher
  - Variablen
  - Arrays

  verwendet.
- Arrays haben eine feste Größe, es ist schwierig, Daten zu modellieren, wenn der Umfang bzw. die Anzahl der zu erwartenden Daten unbekannt ist.

Speicheraufteilung

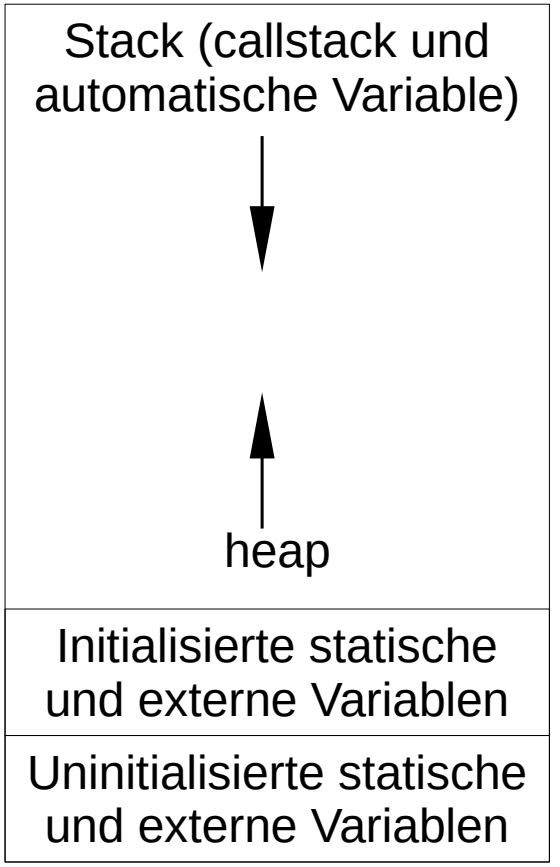
- Der Speicher für ein c-Programm ist in der Regel in verschiedene Segmente aufgeteilt.
  - Codesegment (Programmcode)
  - Stacksegment (lokale und statische Variable)
  - Datensegment (dynamischer Speicher oder heap)
- Je nach Plattform und Speichermodell können Segmente auch zusammgefasst sein, z.Bsp. 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 Datensegement zusammengefasst sind, wachsen Heap und Stack gegenläufig.
- Das Aufeinandertreffen der Speicherbereiche führt idealer Weise zu einer Exception und zu einem geregelten Programmabbruch.
- Das Verhalten ist plattformabhängig.



malloc

- Die Funktion malloc stellt bedarfsgerecht Speicher aus dem Datensegment 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 Funktionaltät 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.

```c
int main(int argc, char** argv)
{
  char proceed='j';
  tStud  *pdata=NULL;
  tStud  *ptmp;
  tStud  *pstud=NULL;
  int count=0;
  int i;
  do
  {
    ptmp=realloc(pdata,sizeof(tStud)*(count+1));
    if(ptmp)
    {
        pdata=ptmp;
        pdata[count]=getStudent();
        count++;
    }
    printf("Weitere Eingabe von Daten (j/n): ");
    fgets(buf,128,stdin);

    proceed=buf[0];
  }
  while (proceed=='j');
  for (i=0; i<count;i++) displayStudent(pdata+i);

  free (pdata);

  return 0;
}
```

Beim ersten Durchlauf NULL

In der Version, In der ein Pointer Übergeben wird.

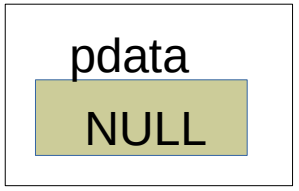
Über realloc wird der Speicher für zunächst einen Studenten bereitgestellt. Er nimmt einen Studenten, erfasst durch get-Student, auf. Der Speicherbereich wird mit jedem Durchlauf um Platz für einen Studenten erweitert.

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.

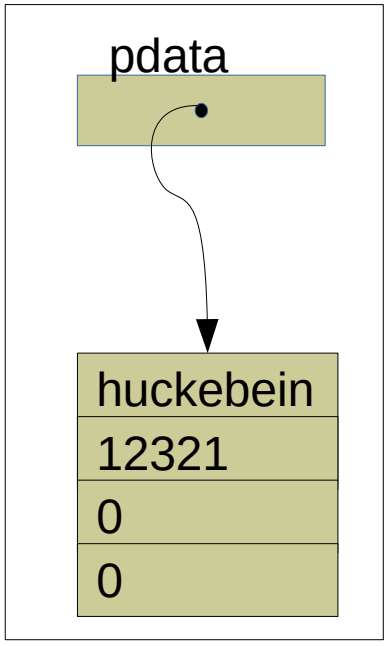
- In Zeile 49 wird der Pointer auf den neu bereitgestellten Speicher zunächst auf ptmp übernommen, da realloc, wenn es nicht erfolgreich ist, NULL zurückgibt.
- Somit wäre der Pointer auf die Daten verdorben.
- Erst in Zeile 52 wird der Pointer nach pdata übernommen.
- In Zeile 53 wird ein neuer Datensatz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

count=0

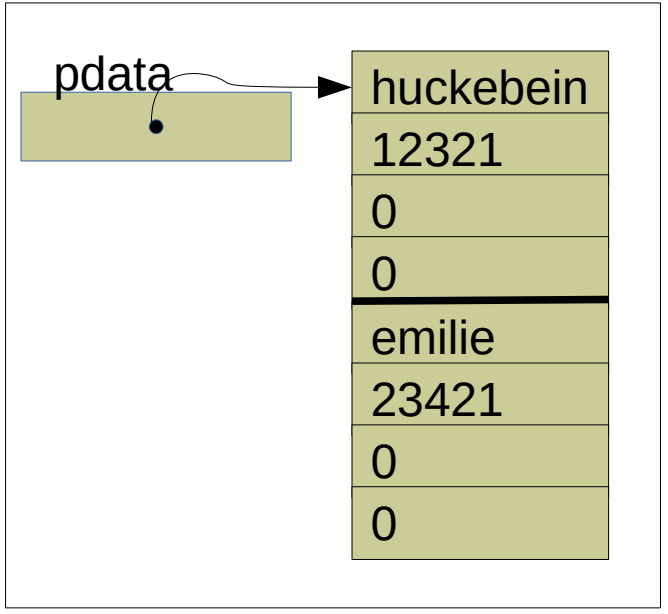


Darstellung der Situation mit 0, 1, 2 oder 3 erfassten Datensätzen

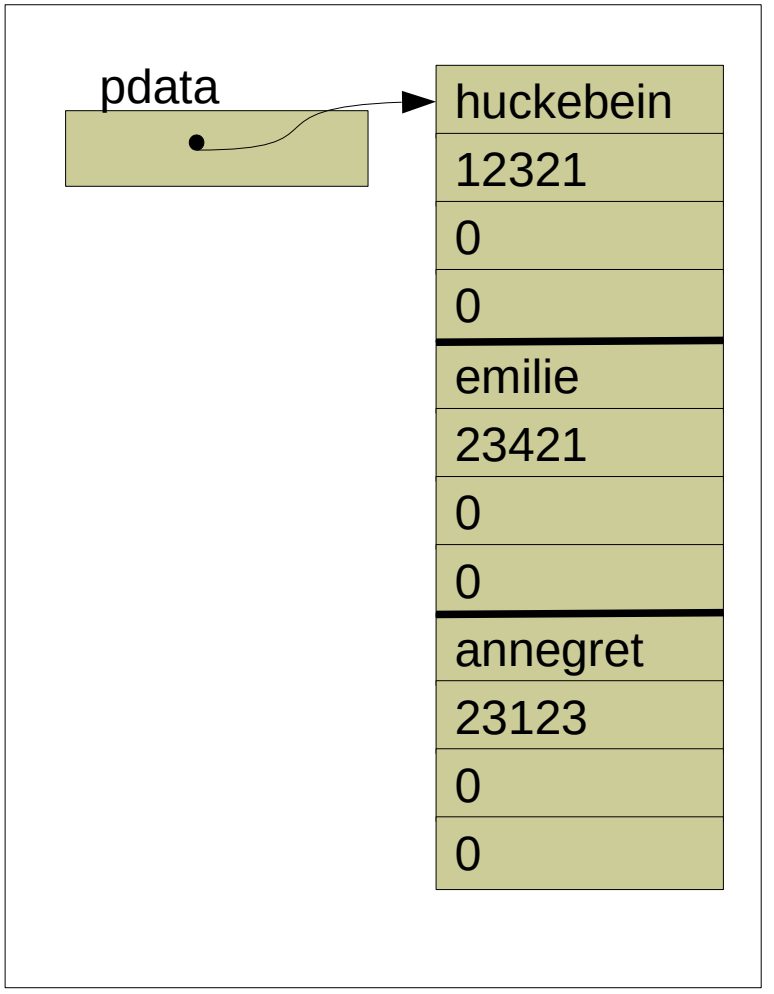
count=1



count=2



count=3



Kompletter Code Beipiel 1

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

char buf[128];

typedef struct
{
  char  name[30+1];
  int   matrNr;
  float noteKl;
  int   noteBel;
}tStud;

void displayStudent(tStud* s)
{
    printf("%-30s, %6d, %3.1f, %d\n",
           s->name,
           s->matrNr,
           s->noteKl,
           s->noteBel);
}

tStud getStudent()
{
    tStud st;
    printf("%-10s: ","Name");
    fgets(buf,32,stdin);
    buf[strlen(buf)-1]=0;
    strcpy(st.name,buf);
    printf("%-10s: ","MatrikelNr");
    fgets(buf,32,stdin);
    st.matrNr=atoi(buf);
    st.noteKl=0;
    st.noteBel=0;
    return st;
}
```

```c

int main(int argc, char** argv)
{
  char proceed='j';
  tStud  *pdata=NULL;
  tStud  *ptmp;
  tStud  *pstud=NULL;
  int count=0;
  int i;
  do
  {
    ptmp=realloc(pdata,sizeof(tStud)*(count+1));
    if(ptmp)
    {
        pdata=ptmp;
        pdata[count]=getStudent();
        count++;
    }
    printf("Weitere Eingabe von Daten (j/n): ");
    fgets(buf,128,stdin);

    proceed=buf[0];
  }
  while (proceed=='j');
  for (i=0; i<count;i++) displayStudent(pdata+i);

  free (pdata);

  return 0;
}

```

```
$ ./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)

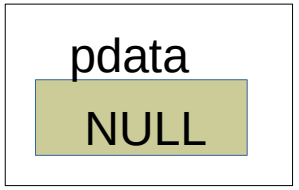
Pointer auf den neuen Studenten oder NULL wird zurückgegeben

- 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

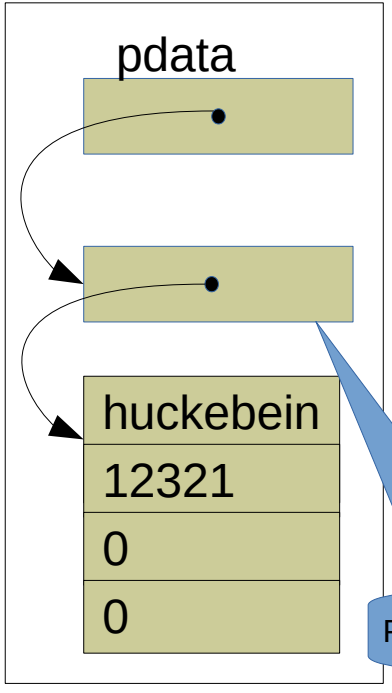
- Der Student wird in Beipiel 2 sofort in einem eigenen Speicherbereich erfasst. Die Variable st in getStudent entfällt.
- Es wird nur noch ein Pointer zurückgegeben und in main zugewiesen (viel effizienter).
- In Beipiel 2 wird ein Pointerarray eingeführt, das ebenfalls dynamisch erzeugt wird und mittels 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

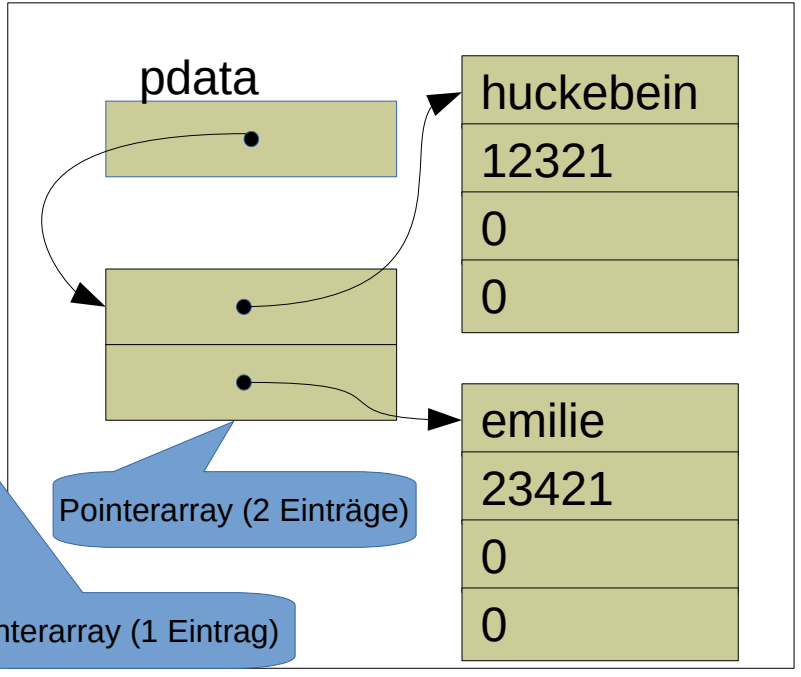


Darstellung der Situation Beispiel 2 mit 0, 1, 2 oder 3 erfassten Datensätzen. Die Datensätze selbst stehen gestreut, irgendwo im heap.

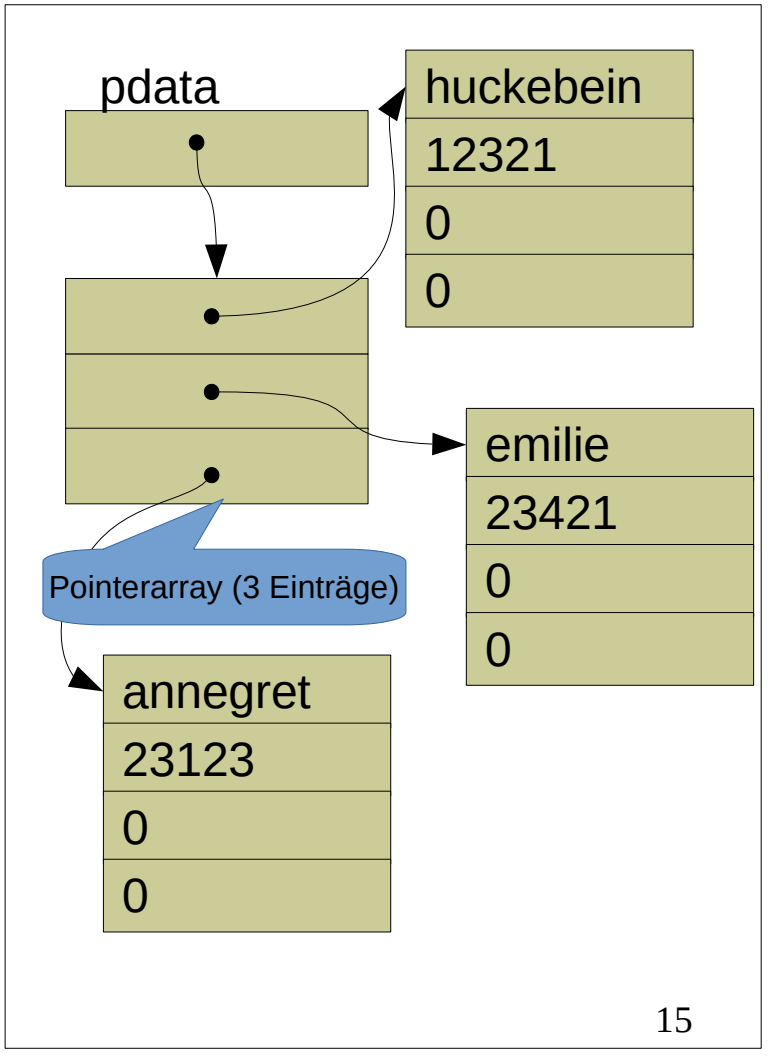
count=1



count=2



count=3



```
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

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.

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  }
```

Freigabe eines jeden Studenten

Freigabe des Pointerarrays

Der komplette Code Beipiel 2

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

char buf[128];

typedef struct
{
  char  name[30+1];
  int   matrNr;
  float noteKl;
  int   noteBel;
}tStud;

void displayStudent(tStud* s)
{
    printf("%-30s, %6d, %3.1f, %d\n",
           s->name,
           s->matrNr,
           s->noteKl,
           s->noteBel);
}

tStud* getStudent()
{
    tStud* ps=malloc(sizeof(tStud));
    if (ps)
    {
        printf("%-10s: ","Name");
        fgets(buf,32,stdin);
        buf[strlen(buf)-1]=0;
        strcpy(ps->name,buf);
        printf("%-10s: ","MatrikelNr");
        fgets(buf,32,stdin);
        ps->matrNr=atoi(buf);
        ps->noteKl=0;
        ps->noteBel=0;
    }
    return ps;
}
```

```c
int main(int argc, char** argv)
{
  char proceed='j';
  tStud **pdata=NULL;
  tStud **ptmp;
  tStud * pstud=NULL;
  int count=0;
  int i;
  do
  {
    pstud=getStudent();
    if (pstud)
    {
        ptmp=realloc(pdata,sizeof(tStud*)*(count+1));
        if(ptmp)
        {
            pdata=ptmp;
            pdata[count]=pstud;
            count++;
        }
        else {perror(NULL);}
    }else {perror(NULL);}
    printf("Weitere Eingabe von Daten (j/n): ");
    fgets(buf,128,stdin);

    proceed=buf[0];
  }
  while (proceed=='j');
  for (i=0; i<count;i++) displayStudent(pdata[i]);

  for (i=0; i<count;i++) free(pdata[i]);
  free(pdata);

  return 0;
}
```

Verkette Liste

- Im Folgenden versehen wir die Struktur mit einer zusätzlichen Komponente next.
- Sie ist ein Zeiger, der auf das nächste Struktu-robjekt, 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
```

Verkette 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;
```

- Dies Liste zeigt ein Stackverhalten, die Reihenfolge der Daten ist hier invertiert.

Verkette 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;}
```

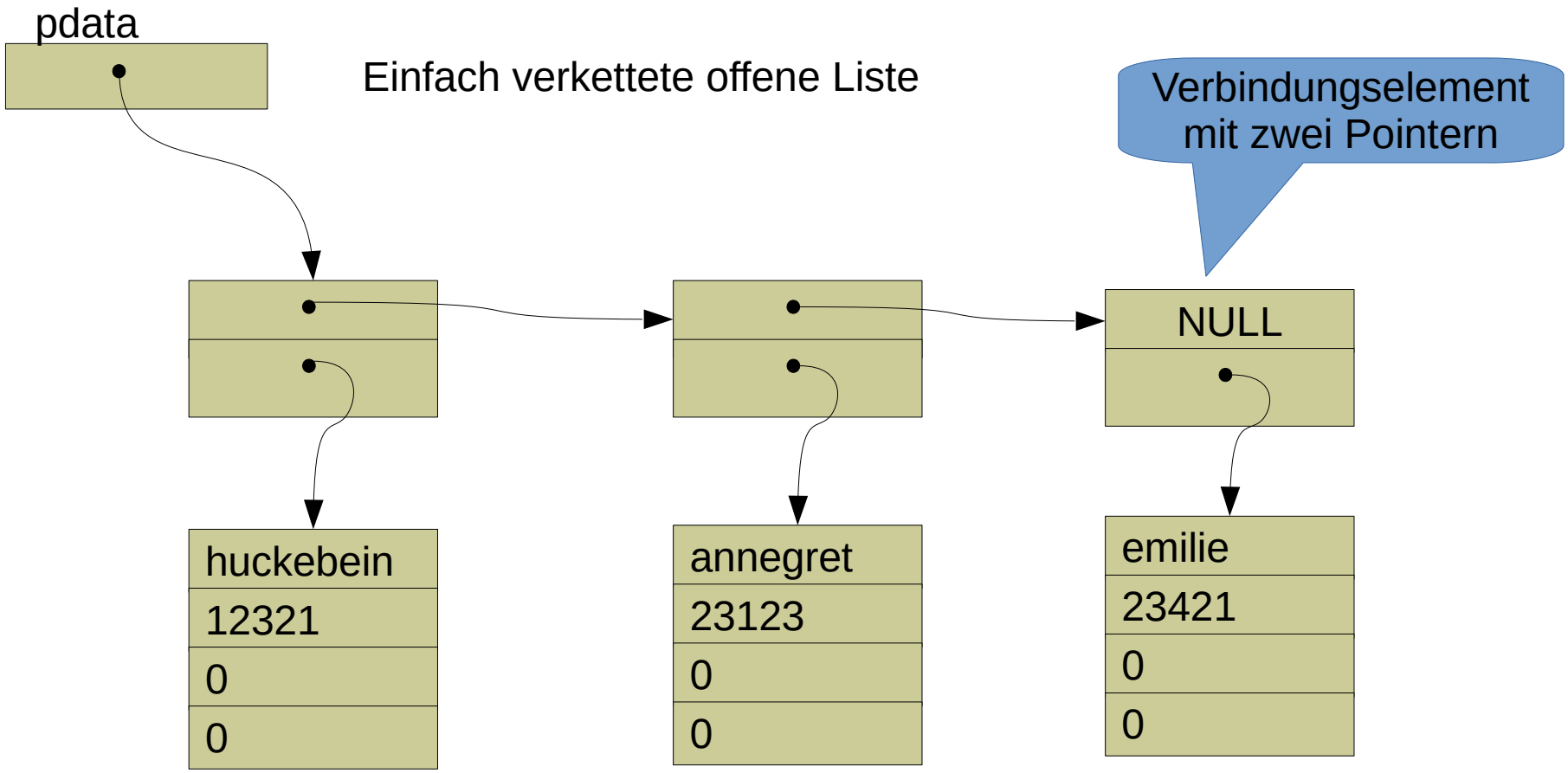
Verkette Liste

- 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;
}
```

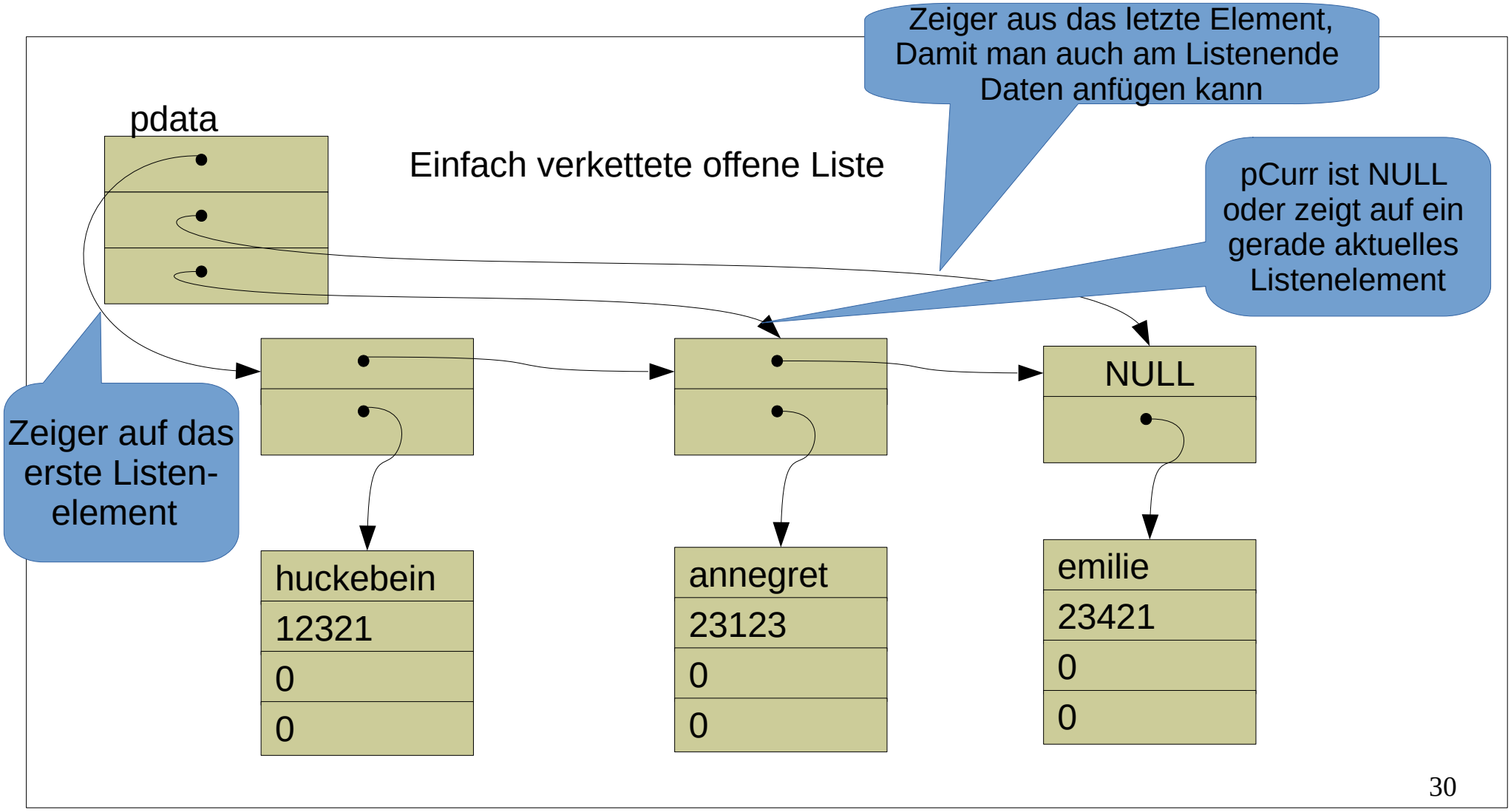
Nicht intrusive Verkette Listen

- 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 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
Verbindungselement
mit zwei Pointern
NULL
huckebein
12321
0
0
annegret
23123
0
0
emilie
23421
0
0

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.


Zeiger aus das letzte Element, Damit man auch am Listenende Daten anfügen kann
pdata
Einfach verkettete offene Liste
pCurr ist NULL oder zeigt auf ein gerade aktuelles Listenelement
Zeiger auf das erste Listen-element
NULL
huckebein
12321
0
0
annegret
23123
0
0
emilie
23421
0
0

Arbeit mit Listen

- **Es ist sinnvoll, eine 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, ofene List zu sehen.
- Sind die Funktionen dazu einmal implementiert, kann man sie immer wieder zur Verwaltung beliebiger Daten einsetzen.

Listenverbindungs-element

```
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);
```

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 Beipielanwendung 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.

Implementation der Listenfunktionen einer einfachverketteten, offenen Liste

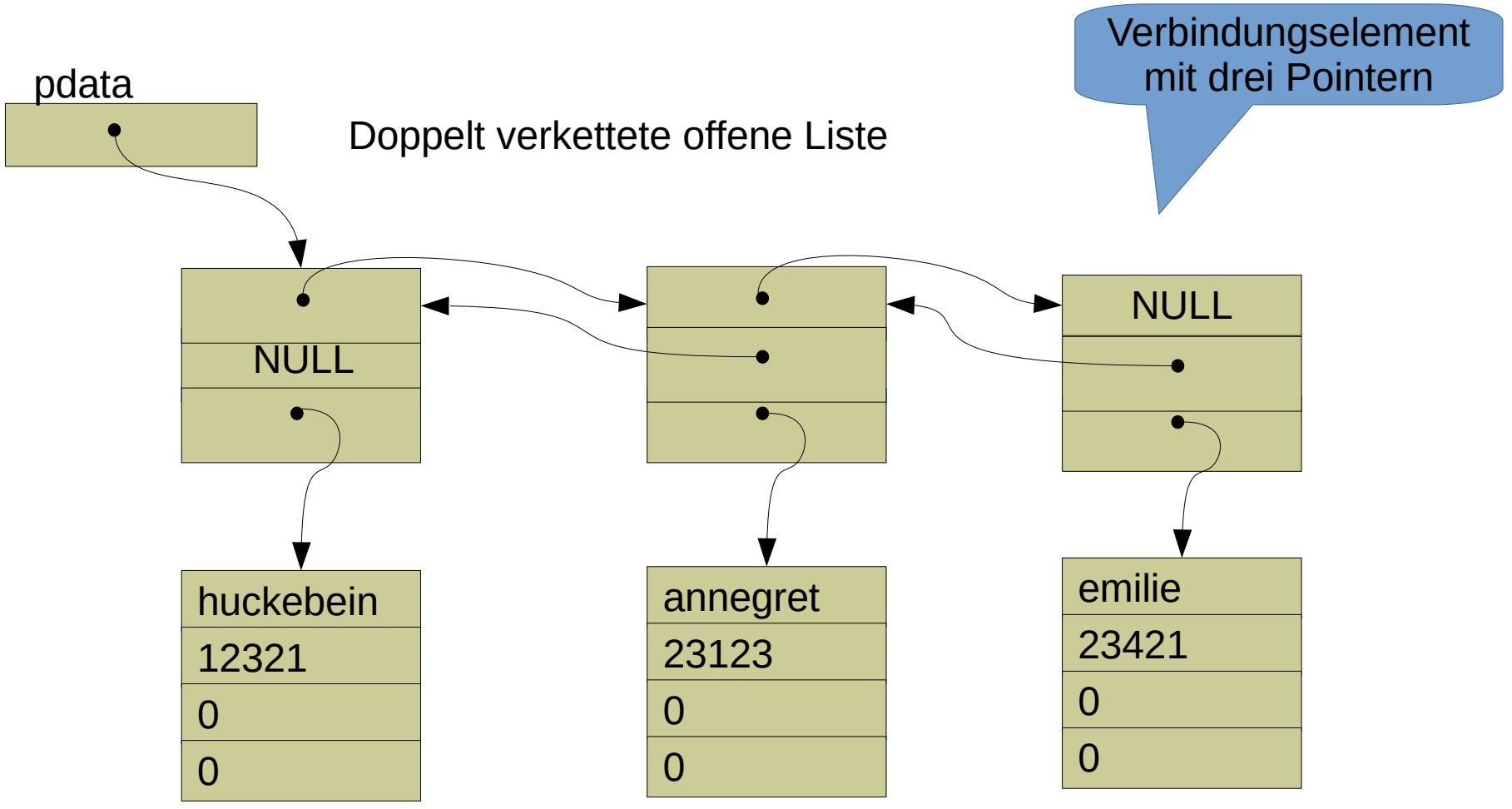
```c
#include <stdlib.h>
#include <malloc.h>
#include "list1.h"

tList* createList()
{
  tList* ptmp;
  ptmp=malloc(sizeof(tList));
  if (ptmp)
      ptmp->pfirst=ptmp->plast=ptmp->pcurr=NULL;
  return ptmp;
}
void*  getFirst(tList* pList)
{
    void* ptmp=NULL;
    if (pList->pfirst)
    {
      ptmp=pList->pfirst->pdata;
      pList->pcurr=pList->pfirst;
    }
    return ptmp;
}
void*  getNext(tList* pList)
{
    void* ptmp=NULL;
    if (pList->pcurr)
    {
      pList->pcurr=pList->pcurr->pnxt;
      if (pList->pcurr)
        ptmp=pList->pcurr->pdata;
    }
    return ptmp;
}
int    insFirst(tList* pList, void* pdata)
{
    tCnct *ptmp= malloc(sizeof(tCnct));
    if (ptmp)
    {
        ptmp->pdata=pdata;
        ptmp->pnxt =pList->pfirst;
        pList->pfirst=ptmp;
        if (pList->plast==NULL) pList->plast=ptmp;
    }
    return (int)(long)ptmp;
}
```

```c
int    insLast (tList* pList, void* pdata)
{
    tCnct *ptmp= malloc(sizeof(tCnct));
    if (ptmp)
    {
        ptmp->pdata=pdata;
        ptmp->pnxt =NULL;
        if (pList->pfirst==NULL) pList->pfirst=ptmp;
        else    pList->plast->pnxt=ptmp;
        pList->plast=ptmp;
    }
    return (int)(long)ptmp;
}
void   removeFirst(tList* pList)
{
    if (pList->pfirst)
    {
        tCnct* tmp=pList->pfirst;
        pList->pfirst=tmp->pnxt;
        free(tmp);
    }
}
void   deleteList(tList* pList)
{
    while (pList->pfirst!=NULL) removeFirst(pList);
        free(pList);
}
```

Doppelt verkettete Liste

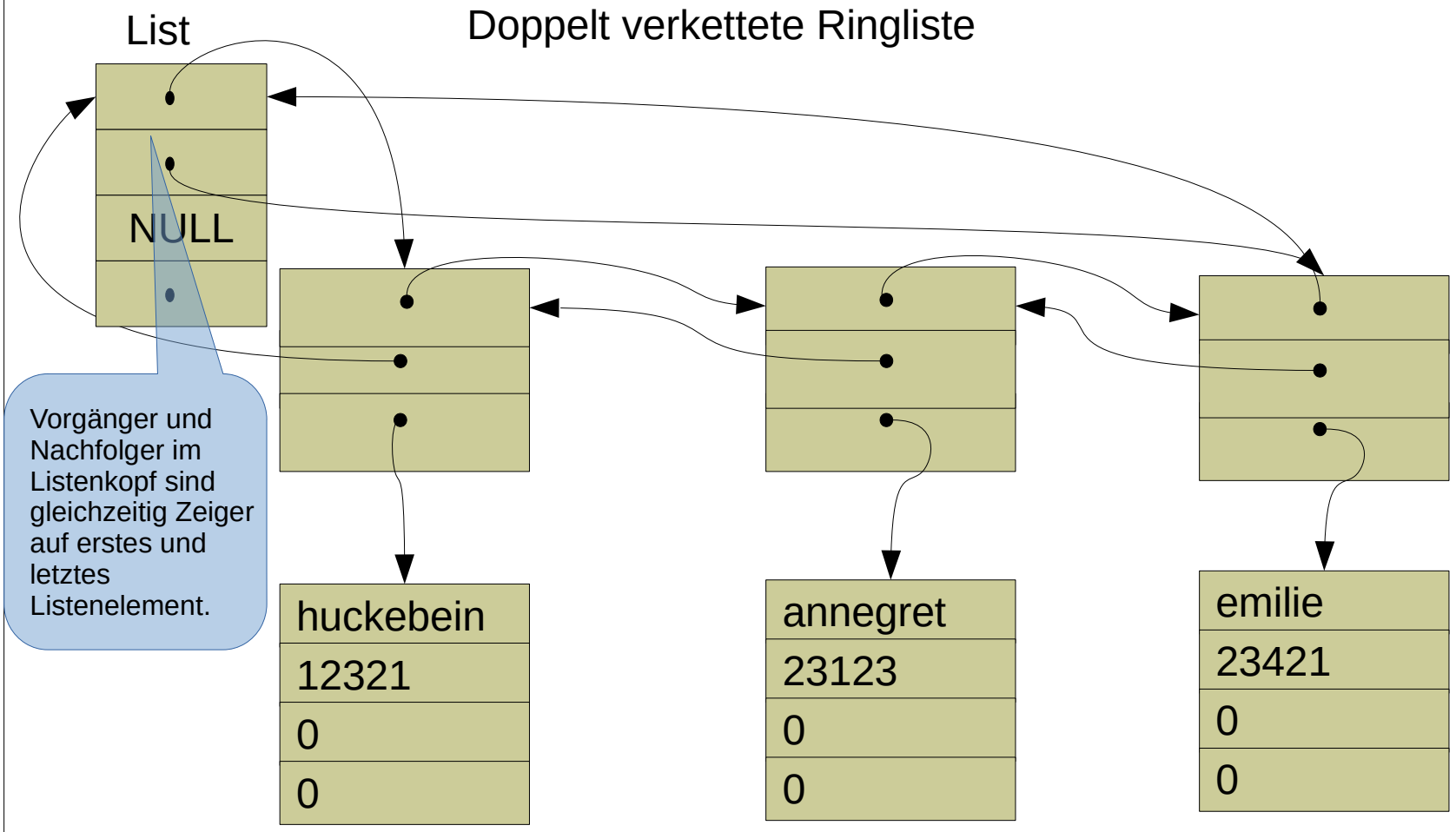
- Ein paar Wünsche lässt die bis hier betrachtete einfach verkettete Liste aber noch offen, man kann hinten oder irgendwo in der Mitte schlecht aus-ketten (Daten aus der Liste entfernen).
- Man kann ein Element an beliebiger Stelle schlecht einfügen.
- Beides wird mit doppelt verkettenten Listen besser 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 Nullpoin-ter gekennzeichnet.
- Diese Art der Listen, wie sie im Bild auf der folgenden Seite dargestellt sind, werden erst einmal nicht weiter verfolgt.


Verbindungselement
mit drei Pointern
pdata
Doppelt verkettete offene Liste
NULL
NULL
huckebein
12321
0
0
annegret
23123
0
0
emilie
23421
0
0

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 Aus-ketten zu programmieren und deckt damit alle Fäl-le (einketten vorn, hinten, an beliebiger Stelle) ab.
- Diese Variante ist viel einfacher zu programmie-ren, als die offene Liste.

Der tatsächlichliche Aufbau der Datenstrukturen bleibt hier dem Anwender Verborgen.

Headerfile für einen Modul Doppelt verkettete Ringliste

```c
/* list.h */

#define OK 1
#define FAIL 0
/*----------------------------------------------------------------------*/
/* Prototypen fuer die Funktionen                                       */

struct tlst;                    // Forewarddeclaration
typedef struct tlst tList;

tList * CreateList(void);                          /* erzeuge leere Liste */
int     DeleteList(tList* pList);                  /* loesche leere Liste */

int    InsertBehind  (tList* pList, void *pItemIns);/* fuege ein hinter %  */
int    InsertBefore  (tList* pList, void *pItemIns);/* fuege ein vor    %  */
int    InsertHead    (tList* pList, void *pItemIns);/* fuege vorn ein      */
int    InsertTail    (tList* pList, void *pItemIns);/* fuege hinten ein    */
int    RemoveItem    (tList* pList);               /* loesche %           */

void* GetSelected    (tList* pList);               /* gib aktuellen DS    */
void* GetFirst       (tList* pList);               /* gib ersten DS       */
void* GetLast        (tList* pList);               /* gib letzten DS      */
void* GetNext        (tList* pList);               /* gib naechsten DS    */
void* GetPrev        (tList* pList);               /* gib vorigen DS      */
void* GetIndexed     (tList* pList,int Idx);       /* gib DS lt. Index    */

void* addItemToList (tList* pList,
                     void * pItem,
                     int(*fcmp)(void*pItList,void*pItNew));

/* % steht fuer aktuellen Satz                                          */
```

```c
/* list.h */
/*------------------------------------------------------------------------*/
/* Datenstruktur eines Listenverbindungselementes                         */

/*------------------------------------------------------------------------*/
/* Prototypen fuer die Funktionen                                         */
#include <stdlib.h>
#include <malloc.h>
/*------------------------------------------------------------------------*/
#include "list.h"
typedef struct Cnctr
{
  struct Cnctr *pNxt;    /* Pointer auf naechstes Verbindungselement      */
  struct Cnctr *pPrv;    /* Pointer auf vorheriges Verbindungselement     */
  void         *pItem;   /* Pointer auf Daten                             */
}tCnct;

typedef struct tlst
{
  tCnct  Head;
  tCnct*pCurr;
}
tList;
int InsCnctBehCnct(tCnct*pBef, void*pI)
{
  tCnct* pCnct=malloc(sizeof(tCnct));
  if(pCnct)
  {
    pCnct->pItem     =pI;
    pCnct->pNxt      =pBef->pNxt;
    pCnct->pPrv      =pBef;
    pCnct->pNxt->pPrv=pCnct;
    pBef->pNxt       =pCnct;
    return OK;
  }
  return FAIL;
}
```

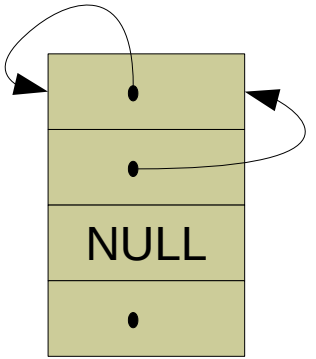
Hier sind die Strukturen jetzt definiert

Das ist die zentrale Funktion zum Einketten in die Liste

```
46  tList* CreateList()                              /* erzeuge leere Liste */
47  {
48    tList* pList;
49    pList=(tList*)malloc(sizeof(tList));
50    if (pList)
51    {
52       pList->pCurr=NULL;
53       pList->Head.pNxt =(tCnct*)pList;
54       pList->Head.pPrv =(tCnct*)pList;
55       pList->Head.pItem=NULL;
56    }
57    return pList;
58  }
59
60  int InsertHead    (tList* pList, void *pItemIns) /* fuege vorn ein      */
61  {
62    return InsCnctBehCnct(&pList->Head, pItemIns);;
63  }
64
65  int InsertTail    (tList* pList, void *pItemIns) /* fuege hinten ein    */
66  {
67    return InsCnctBehCnct(pList->Head.pPrv, pItemIns);
68  }
69
70  int InsertBehind  (tList* pList, void *pItemIns) /* fuege ein hinter %  */
71  {
72    return InsCnctBehCnct(pList->pCurr, pItemIns);
73  }
```

Bei der leeren Liste Zeigen pNxt und pPrv auf den Listenkopf

Die eigentliche Arbeit des Einkettens steckt in InsCnctBehCnct



Listenkopf einer leeren doppelt verketteten Ring-liste

- Die weiteren Funktionen sind trivial.
- Die Funktion addItemToList sortiert ein DatenItem so in die Liste, dass sie sortiert aufgebaut wird, dazu bedarf es eines Funktionspointers, der später behandelt wird.
- Die Funktion getIndexed liefert das Element mit dem Index Idx. Hier kommt man um eine Iteration mittels Schleife nicht umhin.
- Beim Durchlaufen der Liste, gleich in welcher Richtung, kommt man irgendwann zum ListenKopf. Da im Kopf der Zeiger auf die Daten NULL ist, wird NULL auch zrückgegeben. So ergibt sich ganz nebenbei das Endekennzeichen.