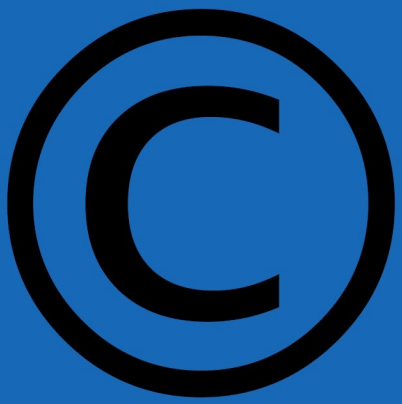
Sven Eric Panitz

Lehrbriefe
C für Java-Programmierer



Zeiger

Zeiger in C

Sven Eric Panitz

12. Juni 2023

Inhaltsverzeichnis

Dynamische Daten

0.1 Adresstypen

Alle Daten stehen irgendwo im Hauptspeicher. Nicht nur Daten sondern, wie wir sehen werden, auch Funktionen, weshalb wir sie auch als Daten benutzen können. Da der Speicher unseres Komputers in einzelne Speicherzellen eingeteilt ist, und diese in Adressen durchnummeriert sind, liegt jedes Datenobjekt im Speicher in einer Speicherzelle mit einer bestimmten Adresse.

0.1.1 Adressoperator

C ermöglicht es über den &-Operator auf eben diese Adresse zuzugreifen. Die Adressen sind dabei eigentlich nur Zahlen, die wir ausgeben können.

GetAddress.c

```c
#include <stdio.h>
int f(int x){return 2*x;}

typedef struct {int x;int y;} Point;

int main(){
  int x = 42;
  int y = 17;
  Point p ={x,y};
  double d=0.0;
  printf("&x: %p\n",&x);
  printf("&y: %p\n",&y);
  printf("&p: %p\n",&p);
  printf("&d: %p\n",&d);
  printf("&f: %p\n",&f);
  return 0;
}
```

Starten wir das Programm, so gibt uns C ein wenig von seinem Innenleben bekannt. Für die einzelnen Variablen wird die Adresse ausgegeben, an deren Stelle im Speicher die einzelnen Variablen stehen.

Shell

```
panitz@px1:~/files/misc$ gcc GetAddress.c
panitz@px1:~/files/misc$ ./a.out
&x: 0x7ffca1ed0750
&y: 0x7ffca1ed0754
&p: 0x7ffca1ed0760
&d: 0x7ffca1ed0758
&f: 0x55597fcf1169
```

Wie man sieht, gibt es für fast alle Daten Adressen: für einfache Typen wie `int` und `double`, für Strukturen aber auch für Funktionen. Wie man der Ausgabe entnehmen kann liegen die lokalen Variablen recht nah beieinander im Hauptspeicher. Die Funktion hingegen an einer komplett anderen Stelle. Wo im Speicher die einzelnen Daten gespeichert werden, darauf haben wir allerdings keinen Einfluss, das bestimmt der Compiler automatisch.

Adressen gibt es allerdings nur für Daten, die in Variablen gespeichert sind. Konstanten und beliebige Ausdrücke sind davon ausgeschlossen. Versucht man es trotzdem, die Adresse zu erfragen, so kommt es zu einem Übersetzungsfehler:

GetAddressError.c

```c
#include <stdio.h>
int main(){
  printf("&42: %u\n",&42);
  printf("&(17+4): %u\n",&(17+4));
}
```

Der Compiler gibt folgende Fehlermeldung:

Shell

```
sep@pc305-3:~/fh/c/student> gcc src/GetAddressError.c
src/GetAddressError.c: In function `main':
src/GetAddressError.c:3: error: invalid lvalue in unary `&'
src/GetAddressError.c:4: error: invalid lvalue in unary `&'
sep@pc305-3:~/fh/c/student>
```

0.1.2 Adresstype

Mit dem `&`-Operator lässt sich für jede Variable die Speicheradresse geben. Doch was für einen Typ hat dieser Operator als Rückgabe? Im oberen Testprogramm haben wir die Rückgabe einfach als Zahl interpretiert ausgegeben. Tatsächlich gibt es zu jedem Typ in C einen passenden Adresstyp, auch als Zeigertyp oder Pointertyp bezeichnet. Er wird mit einem Sternsymbol nach dem Typnamen notiert. So gibt es also zum Typ `int` den Zeigertyp `int*`. Der `&`-Operator liefert zu einer Variablen vom Typ `x` die Adresse des Typs `x*`, also für Variablen des Typs `int` eine Adresse vom Typ `int*`.

FirstPointer.c

```c
#include <stdio.h>
typedef struct {int x;int y;} Point;

int main(){
  int x = 42;
  int* xp = &x;
  int** xpp = &xp;
  Point p = {17,4};
  Point* pp = &p;
  return 0;
```

Aus didaktischen Gründen schreiben wir in diesem Lehrbrief den Zeigertypen so, dass das Sternsymbol direkt nach dem Typnamen folgt, dann ein Leerzeichen und dann die Variable, die diesen Typ hat, also: `int* xp`.
Der in C gängigere Stil ist, das Leerzeichen vor dem Sternsymbol zu schreibenm also: `int *xp`. Dabei sieht das Sternsymbol aber nicht aus, wie ein Bestandteil des Typs, sondern wie ein Operator, der auf eine Variable angewendet wird.
Problematisch wird unsere Schreibweise allerdings, wenn man merhere Variablen mit einer Typdeklaration notieren will:
`int* x,y,z;`
Hier ist nach C nur die erste Variable vom Typ `int*`. `y` und `z` haben nur den Typ `int`.

```
}
```

Wie man an der Variablen `xpp` sieht kann man das Spielchen tatsächlich mehrfach spielen. Man kann auch von einer Adressvariablen sich wiederum die Adresse geben lassen.

Betrachten wir das Programm noch einmal im Detail. In Zeile 5 wird eine Variable deklariert. Hierzu bedarf es einem Speicherplatz im Hauptspeicher um eine Zahl zu speichern. Dieser Speicherplatz hat eine Adresse. In Zeile 6 wird mit `&x` die Adresse des Speicherplatz für die Variable `x` erfragt. Diese Adresse hat den Typ `int*`. Auch sie soll in einer Variablen gespeichert werden. Deshalb wird die Variable `xp` vom Typ `int*` deklariert. Auch die Variable `xp` wird im Speicher in einer Speicherzelle mit einer eigenen Adresse gespeichert. Diese Adresse lässt sich mit `&xp` erfragen. Sie ist vom Typ `int**` und wird in Zeile 7 in der Variablen `xpp` gespeichert.

0.1.3 Stern-Operator

Mit dem `&`-Operator lässt sich die Speicheradresse einer Variablen in Form eines Zeigertyps erfragen. Als inversen Operator gibt es in C den Sternoperator `*`. Wird er auf einem Zeigertyp angewendet, dann werden die Daten angesprochen, die an der entsprechenden Adresse gespeichert sind. Er kehrt quasi die Operation des `&`-Operators wieder um. Während der `&`-Operator für ein `int` eine Adresse auf ein `int` berechnet, also ein `int*`, berechnet der *-Operator für ein `int*` wieder ein `int`.

Jetzt können wir das obige Programm erweitern, und mit Hilfe des Sternoperators für die Zeigervariablen wieder die Werte erfragen, die in den Speicherzellen gespeichert sind, auf die die Zeigervariablen verweisen.

StarOperator.c

```c
#include <stdio.h>
typedef struct {int x;int y;} Point;

int main(){
  int x = 42;
  int* xp = &x;
  int y = *xp;
```

```
  printf("y = %i\n",y);

  int** xpp = &xp;
  int z = **xpp;
  printf("z = %i\n",z);
  printf("**xpp = %i\n",**xpp);

  *xp = 17;
  printf("x = %i\n",x);

  Point p = {17,4};
  Point* pp = &p;
  printf("(*pp).x = %i\n",(*pp).x);
  return 0;
}
```

Shell

```
sep@pc305-3:~/fh/c/student> bin/StarOperator
z = 42
**xpp = 42
x = 17
(*pp).x = 17
sep@pc305-3:~/fh/c/student>
```

0.1.4 Pfeiloperator

Beim Zugriff auf die Felder einer Struktur, die über einen Zeiger zugegriffen wird, ist zunächst mit dem Sternoperator die Referenz aufzulösen, um dann mit dem Punktoperator auf das Feld der Struktur zuzugreifen. Das gibt dann Ausdrücke der Form `(*pp).x`, wie bereits im letzten Beispiel gesehen. C bietet hierfür eine speziellen Operator an, der dieses etwas kürzer notieren lässt, der Pfeiloperator `->`. `(*pp).x` kann mit ihm als `pp->x` ausgedrückt werden.

ArrowOperator.c

```
#include <stdio.h>
typedef struct {int x;int y;} Point;

int main(){
  Point p = {17,4};
  Point* pp = &p;
  printf("(*pp).x = %i\n",(*pp).x);
  printf("pp->x = %i\n",pp->x);
  return 0;
}
```

0.1.5 Referenzübergabe

Was haben wir mit den Zeigern gewonnen (außer dass es etwas komplizierter wird Programme zu verstehen)? In C werden die Parameter an eine Funktion immer per Wert übergeben. Das bedeutet, dass die Parameterwerte kopiert werden. Mit Zeigertypen gibt es nun eine Möglichkeit, nicht den Wert, sondern die Speicheradresse einer Variablen an eine Funktion zu übergeben. Man kann so ein Programm schreiben, in dem die Funktion tatsächlich die übergebene Variable verändern kann.

RefParam.c

```
#include <stdio.h>
int f1(int* x){
  *x = 2* (*x);
  return *x;
}
int main(){
  int y = 42;
  int z = f1(&y);
  printf("y = %i\n",y);
  printf("z = %i\n",z);
  return 0;
}
```

Wie man an der Ausgabe sehen kann, ändert der Aufruf von `f1` jetzt tatsächlich den Wert der Variablen y.

Shell

```
sep@pc305-3:~/fh/c> ./student/bin/RefParam
y = 84
z = 84
sep@pc305-3:~/fh/c>
```

Parameter per Zeiger zu übergeben statt per Wert, kann insbesondere sinnvoll sein, wenn es sich bei den Parametern um Strukturen handelt. Strukturobjekte können mitunter sehr groß sein und viel Speicherplatz beanspruchen. Auch will man gerade recht oft Strukturobjekte durch Prozeduraufrufe verändern.

Betrachten wir hierzu ein weiteres Mal die Struktur `Punkt` aus dem letzten Lehrbrief. Wir haben bereits festgestellt, dass bei einer Wertübergabe die Funktion `verschiebe` das ursprünglichen Punktobjekt gar nicht verändert. Jetzt per Übergabe des Punktobjets als Zeiger tritt der gewünschte Effekt ein:

Punkt4.c

```
#include <stdio.h>
typedef struct {
  int x;
  int y;
} Punkt;
```

```c
void printPunkt(Punkt p){
  printf("(%i,%i)\n",p.x,p.y);
}

void verschiebe(Punkt* p){
  p->x=p->x+1;
  p->y=p->y+2;
}

int main(){
  Punkt p={17,4};
  printPunkt(p);
  verschiebe(&p);
  printPunkt(p);
  return 0;
}
```

0.1.6 Zeiger auf lokale Variablen

Bisher vertrauen wir auf die automatische Speicherverwaltung lokaler Variablen in C. Für eine lokale Variable einer Funktion ebenso wie für einen Funktionsparameter, wird beim Aufruf einer Funktion Platz im Speicher angelegt. An dieser Stelle werden die Daten der Variablen gespeichert. Sobald die Funktion fertig ausgewertet ist und ihr Ergebnis berechnet hat, wird dieser Speicherplatz nicht mehr benötigt. Die Speicherstelle wird wieder freigegeben, um sie später für eventuelle andere Variablen zu benutzen.

Da wir uns explizit mit dem &-Operator die Adresse einer Speicherzelle geben lassen können, so können wir eine solche Adresse auch als Funktionsergebnis zurückgeben. Es lässt sich folgende Funktion definieren:

DeadVariableError.c

```c
#include <stdio.h>

int* f(int y){
  int x= y;
  return &x;
}
```

Bei einem Aufruf dieser Funktion erhalten wir einen Zeiger auf eine Speicherzelle, in der während des Aufrufs eine lokale Variable gespeichert war. Auf die Variable selbst lässt sich nicht mehr zugreifen. Sie existiert quasi nicht mehr. Wir haben also einen Zeiger in einen Speicherbereich, der einmal für eine lokale Variable genutzt wurde. Diese Variable gibt es aber gar nicht mehr.

Der Compiler macht uns auch tatsächlich darauf aufmerksam, dass hier etwas merkwürdig ist:

Shell

```
sep@pc305-3:~/fh/c> gcc DeadVariableError.c
DeadVariableError.c: In function `f':
DeadVariableError.c:5: warning: function returns address of local variable
sep@pc305-3:~/fh/c>
```

Da es sich aber nur um eine Warnung handelt, vielleicht wissen wir ja was wir tun, können wir das Programm übersetzen. Wir vervollständigen es um eine zweite Funktion und um eine Hauptfunktion.

DeadVariableError.c

```
void f2(int y){
  int z= y;
}

int main(){
  int* xp = f(42);
  f2(17);
  printf("%i\n",*xp);
  return 0;
}
```

Das Programm kann zu folgender Ausgabe führen:

Shell

```
sep@pc305-3:~/fh/c> ./DeadVariable
17
sep@pc305-3:~/fh/c>
```

Hier lässt sich etwas über die interne Speicherverwaltung unseres Compilers lernen. Wir lassen uns die Speicherzelle der lokalen Variablen `x` aus der Funktion `f` zurückgeben. Anschließend rufen wir die Funktion `f2` auf. Diese hat ihrerseits eine lokale Variable. Nach Aufruf der Funktion `f2` befindet sich ein Wert in der Speicherzelle die einstmals die lokale Variable `x` speicherte, der zuletzt in der lokalen Variablen `z` der Funktion `f2` stand.

Tatsächlich ist es ziemlicher Zufall, was sich am Speicherplatz einer nicht mehr existierenden lokalen Variable befindet.

0.1.7 der NULL-Zeiger

Wir haben uns bemüht Variablen immer gleich bei der Deklaration auch einen initialen Wert zuzuweisen. Wird einer Variablen kein Wert zugewiesen, so hängt es vom Zufall ab, was für einen Wert man bekommt, wenn auf eine solche noch nicht initialisierte Variable zugegriffen wird. Dasselbe gilt auch für Zeigervariablen. Es gibt aber Situationen, in denen man eine Zeigervariable deklariert, ohne dass schon der zuzuweisende Adresswert vorhanden ist. Oder umgekehrt: für eine Zeigervariable ist im Laufe des Programms keine gültige Adresse mehr vorhanden. Um in

einer Zeigervariablen nicht einen zufälligen oder ungültigen Adresswert speichern zu müssen, gibt es in C die Möglichkeit eines ausgezeichneten null-Zeigers. Er wird durch das Wort `NULL` bezeichnet. Man kann mit der Gleichheit testen, ob ein Zeiger mit `NULL` belegt ist.

NullVar.c

```c
#include <stdlib.h>
#include <stdio.h>
int* f(int* y){
  static int* x=NULL;
  if (NULL!=x) x=y;
  else *y=*x;
  return x;
}

int main(){
  int x= 42;
  printf("*f(&x)%i\n",*f(&x));
  x=17;
  printf("*f(&x)%i\n",*f(&x));
  printf("x%i\n",x);
  return 0;
}
```

Um den `NULL`-Zeiger verwenden zu können, ist es notwendig die Bibliothek `stdlib.h` zu inkludieren.

0.2 der void-Zeiger

Wir haben gesehen, dass es für jeden Typ, einen entsprechenden Zeigertyp gibt. So gibt es zu `int` den Zeigertyp `int*` und für eine Struktur `struct Punkt` den Zeigertyp `struct Punkt*`. Technisch gesehen sind alle diese Zeigertypen eine Zahl, die eine Adresse in unserem Speicherbereich bezeichnet. Also sind eigentlich alle Zeigertypen Daten von derselben Art. Die Programmiersprache C bietet einen besonderen allgemeinen Zeigertyp an, der ausdrückt, dass es sich um irgendeinen Zeiger handelt, der aber nicht angibt, was in der Speicheradresse gespeichert ist, auf die der Zeiger verweist. Dieser allgemeine Zeigertyp wird mit `void*` bezeichnet. Das Wort `void` sollte hier so gelesen werden, dass nicht bekannt ist, was für Daten an der Speicherstelle stehen, auf die verwiesen wird.

Zunächst definieren wir als Beispiel eine kleine Struktur:

VoidPointer.c

```c
#include <stdio.h>

typedef struct {
  int x;
  int y;
} Punkt;
```

Wir wollen einmal einen `void`-Zeiger benutzen:

VoidPointer.c

```
int main(){
  void* object;
```

In diesem `void`-Zeiger lässt sich jetzt z.B. die Adresse einer `int`-Vatiablen speichern.

VoidPointer.c

```
  int x = 17;
  object = &x;
```

Soll die Adresse über den `void`-Zeiger angesprochen werden und mit der darin gespeicherte Zahl gerechnet werden, so müssen wir explizit den Zeiger zu einem Zeiger auf eine `int`-Zahl umwandeln. Hierzu ist die Notation in runden Klammern zu benutzen, wie wir sie schon vom Rechnen mit primitiven Zahlen gewohnt sind.

VoidPointer.c

```
  *(int*)object += 4;
  *(int*)object *= 2;
  printf("x = %i\n",x);
```

Wir können aber auf die gleiche Weise versuchen, in der `void`-Zeigervariablen ein Objekt unserer Punktstruktur zu speichern. Um wieder auf das Objekt zugreifen zu können, ist dann der Zeiger wieder entsprechend umzuwandeln.

VoidPointer.c

```
  Punkt p = {2,6};
  object = &p;

  printf("p.x: %i\n",p.x);
  printf("((Punkt*)object)->x: %i\n",((Punkt*)object)->x);

  return 0;
}
```

Im obigen Beispiel ist mit dem Namen `object` schon angedeutet, wofür man den `void`-Zeiger verwenden kann: für die Möglichkeit, Zeiger auf Daten belieber Art zu speiechern. In der objektorientierten Programmierung wird dieses in dem Konzept eines allgemeinen Objektes münden. Am Ende dieses Kapitels werden wir ein Beispiel sehen, in dem wir `void`-Pointer sinnvoll nutzen.

0.3 Rechnen mit Zeigern

Zeiger sind technisch nur die Adressen bestimmter Speicherzellen, also nichts weiteres als Zahlen. Und mit Zahlen können wir rechnen. So können tatsächlich arithmetische Operationen auf Zeigern durchgeführt werden.

Beispiel 0.2 Wir erzeugen mehrere Zeiger auf ganze Zahlen und betrachten die Nachbarschaft eines dieser Zeiger.

ZeigerArith.c

```c
#include <stdio.h>

int main(){
  int i = 5;
  int j = 42;
  int *pI = &i;
  int *pJ = &j;

  printf
   ("i: %i, pI: %p, pI+1: %p, *(pI+1): %i,\npI-1: %p, *(pI-1): %i, pJ: %p, *pJ: %i\n"
    ,i   ,pI   ,pI+1   ,*(pI+1)   ,pI-1   ,*(pI-1)   ,pJ   ,*pJ );
  return 0;
}
```

Dieses Programm erzeugt z.B. folgende interessante Ausgabe:

Shell

```
panitz@px1:~$ gcc ZeigerArith.c
panitz@px1:~$ ./a.out
i: 5, pI: 0x7ffdff1ca4c0, pI+1: 0x7ffdff1ca4c4, *(pI+1): 42,
pI-1: 0x7ffdff1ca4bc, *(pI-1): 0, pJ: 0x7ffdff1ca4c4, *pJ: 42
```

Das Ergebnis einer Zeigerarithmetik hängt sehr davon ab, wie die einzelnen Variablen im Speicher abgelegt werden. In der Regel wird man die Finger von ihr lassen und nur ganz bewußt auf sie zurückgreifen, wenn man genau weiß, was man tut, um eventuell Optimierungen durchführen zu können.

Erstaunlicher Weise werden wir aber ein C-Konstrukt kennenlernen, das eines der gebräuchlichsten Konzepte in C überhaupt ist, und tatsächlich auf der Zeigerarithmetik basiert.

0.4 Speicherverwaltung

Dieses Kapitel trägt den Titel dynamische Daten. Hiervon haben wir vorerst aber noch nicht wirklich etwas gesehen. Bisher haben wir nur geschaut, in welchen Speicherzellen die Daten unserer Variablen gespeichert sind. Damit ist in einem Programm relativ statisch durch die Anzahl der Variablen festgelegt, wieviel Daten das Programm speichern kann. Normaler Weise wünscht

man sich aber, dass ein Programm während des Programmdurchlaufs dynamisch neuen Speicher nach Bedarf für weitere Daten anlegen kann. Hierzu bietet C die Möglichkeit an über, Speicherallokationsfunktionen neuen Speicher zu reservieren.

0.4.1 Allocation

In der C-Standardbibliothek gibt es zwei ein Funktionen, mit der Speicher beliebiger Größe vom Compiler angefordert werden kann.

malloc Die Funktion `malloc` hat einen Parameter. In diesem wird angegeben, wieviel Speicherplatz bereitgestellt werden soll. Diese Angabe ist die Anzahl der Byte. Das Ergebnis ist ein Zeiger auf diesen neu bereitgestellten Speicher. Da die Funktion `malloc` nicht wissen kann, was für Daten einmal in diesem Speicherbereich gespeichert werden sollen, ist ihr Rückgabetyp der allgemeine Zeiger `void*`.

Malloc1.c

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

int main(){
  void* vp = malloc(7);
  int* ip=(int*)vp;
  *ip=42;
  printf("%i\n",*ip);
  return 0;
}
```

Im obigen Beispiel haben wir willkürlich sieben Byte Speicher reservieren lassen, um sie dann zur Speicherung von einer `int`-Zahl zu benutzen. Man sollte natürlich am Besten immer genau soviel Speicher anfordern, wie für die geplanten Zwecke gebraucht wird. Wir kennen bereits die Funktion `sizeof`, die Angaben darüber macht, wieviel Speicher ein bestimmter Typ benötigt. Mit dieser lässt sich genau der erforderlicher Speicherplatz anfordern:

Malloc2.c

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

int main(){
  void* vp = malloc(sizeof(int));
  int* ip=(int*)vp;
  *ip=42;
  printf("%i\n",*ip);
  return 0;
}
```

calloc Es gibt in C eine Variante der Funktion `malloc`: die Funktion `calloc`. Nur, dass es bei der Funktion calloc() nicht einen, sondern zwei Parameter gibt. Im Gegensatz zu `malloc` können wir mit `calloc` noch die Anzahl von Speicherobjekten angeben, die reserviert werden soll. Wird z.B. für 10 Zahlen vom Typ `int` Speicherplatz benötigt, so kann dies mit `calloc` erledigt werden. Auf die mehreren Elemente dieses Speicherbereichs kann dann mit der Zeigerarithmetik zugegriffen werden:

Calloc.c

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

int main(){
  int* xs = (int*)calloc(10,sizeof(int));
  int i=0;
  for(i=0;i<10;i++)  *(xs+i) = 2*i;
  for(i=0;i<10;i++)  printf("%i, ",*(xs+i));
  printf("\n");
  return 0;
}
```

Wir haben somit ein kleines Programm geschrieben, in dem wir Speicher für 10 Zahlen angefordert haben und mit diesen 10 Speicherplätzen auch gearbeitet haben:

Shell

```
sep@pc305-3:~/fh/c/student> bin/Calloc
0, 2, 4, 6, 8, 10, 12, 14, 16, 18,
sep@pc305-3:~/fh/c/student>
```

Zusätzlich werden mit der Funktion `calloc` alle Werte des angeforderten Speichers automatisch mit binären 0-Werten initialisiert. Bei `malloc` hat der reservierte Speicherplatz zunächst einen undefinierten Wert. Braucht man die zusätzliche Initialisierung des Speichers nicht, so kann natürlich auch mit `malloc` Speicher für mehrere Elemente angefordert werden. Der Aufruf
`calloc(10,sizeof(int));`
ist hierzu zu Ersetzen durch:
`malloc(10*sizeof(int));`

realloc Die Funktion `realloc` nimmt einen Zeiger auf einen Speicherbereich und eine neue Größe, des zu reservierenden Speichers. Ändert die Größe des dort verwendeten Speichers, so bleiben die alten Daten in dem Speicherbereich unberührt. Wird der Speicher größer, ist der neue Bereich undefiniert. Rückgabe ist ein neuer Zeiger. Dieser kann identisch mit den ursprünglichen Zeiger sein, dann war imSpeicher noch genügend Pltz, um die neue Größe zu reservieren, es kann aber auch ein neuer Zeiger sein. Dann mussten intern die Daten in einen neueren Speicherbereich kopiert werden.

Wird als erster Parameter `NULL` übergeben, dann entspricht der Aufruf von `realloc(NULL, n)` dem Aufruf `malloc(n)`.

0.4.2 Speicherfreigabe

Speicher anzufordern ist ein feine Sache, aber wenn wir auch viel davon in heutigen Rechner haben, so ist der Speicher trotzdem endlich. Wenn wir sorglos immer wieder neuen Speicher anfordern bekommen wir ein Müllproblem. Wie sich ein solches auswirkt, lässt sich leicht ausprobieren. Man starte einmal das folgende Progamm.

NoFree.c

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int i=0;
  int* ip;
  for (;i<10000000;i++){
    ip=(int*)malloc(sizeof(int));
    *ip=i;
    printf("%i ",*ip);
  }
  return 0;
}
```

Mit dem Betriebssystembefehl `top` lässt sich schön beobachten, wie dieses Programm immer neuen Speicher benötigt und schließlich über 100 Megabyte Speicher reserviert hat (und damit den Rechner schon arg belastet). Man spricht in einem solchen Fall von einem Speicherleck. Eine der großen Schwierigkeiten der Programmiersprache C ist es, dass dynamisch allokierter Speicher explizit wieder frei gegeben werden muss. Hierzu gibt es die Funktion `free`. Sie erwartet einen Zeiger und gibt alles mit diesem Zeiger allokierten Speicherbereich wieder frei zur neuen Benutzung. Rufen wir nun im obigen Programm am Ende des Schleifenrumpfs die Funktion `free` auf, so können wir beobachten, dass das Programm mit konstanten Speicherbelegung läuft.

Free.c

```
#include <stdio.h>
#include <stdlib.h>
int main(){
  int i=0;
  int* ip;
  for (;i<10000000;i++){
    ip=(int*)malloc(sizeof(int));
    *ip=i;
    printf("%i ",*ip);
    free(ip);
  }
  return 0;
}
```

Schon seit Anfang der 60er Jahre gibt es mit *Lisp* eine Programmiersprache, in der die dynamische Speicherverwaltung automatisch vom Compiler durchgeführt wird. Dieses macht die sogenannte *garbage collection*. Mitte der 90er Jahre setzte sich mit *Java* eine Mainstream Program-

miersprache durch, die auch eine automatische Speicherverwaltung per *garbage collection* hat.

Ganz allein lässt uns C allerdings auch nicht mit der Speicherverwaltung. Zum einen wird der Speicher lokaler Variablen nach Ablauf einer Funktionsberechnung wieder frei gegeben, zum anderen müssen wir dem Compiler in der Funktion `free` nicht mitteilen, wieviel Speicher denn wieder frei gegeben werden soll. Die C-Laufzeit merkt sich für jede dynamisch angeforderte Speicherzelle, wieviel Speicher angefordert wurde und gibt genau diese Speichergröße dann auch wieder frei.

0.4.3 objektorientiertes Arbeiten mit Strukturzeigern

Es ist Vorteilhaft auch in C schon einen weitgehendst objektorientierten Programmierstil zu kultivieren. Objektarten werden durch Strukturen definiert. Objekte dieser Strukturen sollen nur über einen Zeiger benutzt werden. Wir können eine weitgehendst objektorientiert angelegte Struktur gerne auch schon als *Klasse* bezeichnen.

Hierzu nocheinmal das Beispiel mit den Punkten im zweidimensionaloen Raum:

Punkt5.c

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

typedef struct {
  int x;
  int y;
} PunktStrukt;

typedef PunktStrukt* Punkt;
```

Es ist sinnvoll eine Funktion vorzusehen, die aus gegebenen Werte ein Objekt einer Struktur erzeugt. Hierzu ist der entsprechende Speicher zu allokieren und die einzelnen Felder des Strukturobjekts mit den Werten der Parameter zu übergeben. Eine solche Funktion wird als Konstruktor bezeichnet. Eine sinnvolle Namensgebung ist den Konstruktornamen mit `new` beginnen zu lassen, gefolgt von dem Namen der Struktur:

Punkt5.c

```c
Punkt newPunkt(int x,int y){
  Punkt this = (Punkt)malloc(sizeof(PunktStrukt));
  this->x = x;
  this->y = y;
  return this;
}
```

Ebenso sollte man eine Funktion vorsehen, die es erlaubt ein Objekt wieder aus dem Speicher zu entfernen. Man spricht dabei von einen Destruktor. Meistens reicht es aus, hier einfach die

Funktion `free` aufzurufen, wir werden aber bald auch ein Beispiel sehen, in dem dies nicht ausreicht. Für die Punkt-Klasse erhalten wir den einfachen Destruktor:

Punkt5.c

```
void deletePunkt(Punkt this){
  free(this);
}
```

Nun können wir Punktobjekte erzeugen und löschen. Jetzt wollen wir mit diesen Objekten auch gerne etwas vornehmen. Dazu schreiben wir Funktionen, die als ersten Parameter ein entsprechendes Objekt erhalten. In Hinblick auf objektorientierte Sprachen ist es sinnvoll diesen ersten Parameter dann mit den Namen `this` zu bezeichnen. Funktionen, die als ersten Parameter Objekte einer bestimmten Klasse erhalten, sollen als *Objektmethode* kurz auch nur *Mehtode* dieser Klasse bezeichnet werden.

So können wir eine einfache Methode schreiben, die ein Punktobjekt auf der Kommandozeile ausgibt:

Punkt5.c

```
void printPunkt(Punkt this){
  printf("(%i,%i)\n",this->x,this->y);
}
```

Mehtoden können auch Rückgaben haben. Die folgende Mehtode errechnet den Betrag eines Punktes, also seine Entfernung zum Ursprung. Hierzu wird die Formel von Pythagoras verwendet:

Punkt5.c

```
double betrag(Punkt this){
  return sqrt(this->x*this->x+this->y*this->y);
}
```

Methoden können nach dem ersten Parameter noch weitere Parameter enthalten. Die folgende Methode kann ein Punktobjekt um die als weitere Werte angegebenen Distanzen verschieben.

Punkt5.c

```
void verschiebe(Punkt this,int deltaX,int deltaY){
  this->x += deltaX;
  this->y += deltaY;
}
```

Nun können wir tatsächlich schon fast wie in einer objektorientierten Sprache mit der Punkt-Klasse arbeiten:

Punkt5.c

```c
int main(){
  Punkt p= newPunkt(17,4);
  printPunkt(p);
  printf("betrag: %f\n",betrag(p));
  verschiebe(p,2,4);
  printPunkt(p);
  deletePunkt(p);
  return 0;
}
```

Das Programm führt zu folgender Ausgabe:

Shell

```
sep@pc305-3:~/fh/c/student> bin/Punkt5
(17,4)
betrag: 17.464249
(19,8)
sep@pc305-3:~/fh/c/student>
```

1 Reihungen (Arrays)

Eines der komplexesten Konstrukte der Programmiersprache C ist gleichzeitig auch das wahrscheinlich am häufigsten benutzte. Es handelt sich dabei um Arrays, auf Deutsch oft als Datenfelder bezeichnet. Innerhalb dieses Skripts werden wir sie mit dem ebenfalls gebräuchlichen Ausdruck *Reihungen* bezeichnen, da wir als Feld auch schon die Attribute von Strukturen bezeichnen. Bei Reihungen geht es darum, mehrere Variablen eines uniformen Datentyps in einer Variablen zu speichern.

Der Typ einer Reihung vor und hinter dem Variablennamen bezeichnet. Vor dem Variablennamen steht der Typ, den die einzelnen Elemente der Reihung haben und nach dem Variablennamen steht ein eckiges Klammernpaar, um zu bezeichenen, dass es sich hier im eine Reihungsvariablen handelt. So bezeichnet `int xs []` eine Reihung von ganzen Zahlen. Eine Reihung kann, ebenso wie wir es von Strukturen kennen, durch die Aufzählung der Werte in geschweiften Klammern intialisiert werden.

Zeit unseren ersten Array zu deklarieren. Statt vier einzelner Variablen (noch einmal als Beispiel drüber) wird eine einzige Variable mit vier Werten deklariert:

FirstArray.c

```c
#include <stdio.h>

int main(){
  int x0 = 1;
  int x1 = 2;
  int x2 = 3;
```

```
  int x3 = 4;
  int x [] = {1,2,3,4};
```

Normale Variablen können wir direkt benutzen, um an ihre Werte zu kommen:

FirstArray.c

```
  printf("x0  =%i, x1  =%i, x2  =%i, x3 =%i\n",x0,x1,x2,x3);
```

In Arrayvariablen verstecken sich mehrere einzelne Werte. Um auf diese einzeln zugreifen zu können wird ein Index benutzt. Der Index ist in eckigen Klammern der Variablen anzuhängen. Das erste Element wird über den Index 0 angesprochen. Die vier Werte unserer ersten Reihung sind also wie folgt anzusprechen:

FirstArray.c

```
  printf("x[0]=%i, x[1]=%i, x[2]=%i, x[3]=%i\n",x[0],x[1],x[2],x[3]);
  return 0;
}
```

Bei der Deklaration einer Reihungsvariablen muss die Anzahl der Elemente bekannt sein. Im ersten Beispiel wurde die Anzahl aus der Elementanzahl in der Initialisierung spezifiziert. Wird eine Reihung nicht direkt initialisiert, so beschwert sich der Compiler darüber, dass er nicht bestimmen kann, wieviel Elemente die Reihung benötigt:

UnknownSizeError.c

```
int main(){
  int xs [];
  return 0;
}
```

Die Übersetzung dieses Programms führt zu einem Fehler:

Shell

```
sep@pc305-3:~/fh/c/student> gcc src/UnknownSize.c
src/UnknownSize.c: In function `main':
src/UnknownSize.c:2: error: array size missing in `xs'
sep@pc305-3:~/fh/c/student>
```

Der C-Übersetzer muss immer in der Lage sein, die Anzahl der Elemente einer Reihung abzuleiten. Der Grund dafür ist, dass sobald eine Variable deklariert wird, der C Compiler den Speicherplatz für die entsprechende Variable anfordern muss. In C bedeutet das bei einer Reihung, dass der Speicher für eine komplette Reihung angefordert wird. Hierfür muss die Anzahl der Elemente bekannt sein.

Man kann die Anzahl der Elemente in den eckigen Klammern bei einer Arraydeklaration angeben. Im folgenden Beispiel wird ein Array mit 15 Elementen angelegt.

KnownSize.c

```c
#include <stdio.h>

int main(){
  int xs [15];
  printf("%i\n",xs[2]);
  return 0;
}
```

1.1 Die Größe einer Reihung

Einer Reihung kann man im allgemeinen nicht ansehen, wie viele Elemente sie enthält. Das ist recht unangenehm, denn somit kann es z.B. passieren, das versucht wird bei einer Reihung mit 10 Elementen, auf das 100. Element zuzugreifen.

WrongIndex.c

```c
#include <stdio.h>
int main(){
  int xs [10];
  printf("%i\n",xs[100]);
  return 0;
}
```

Es gibt niemanden, der uns in diesem Programm auf die Finger haut. Wir können es compilieren und laufen lassen. Es läuft sogar ohne Fehlermeldung und Programmabbruch:

Shell

```
sep@pc305-3:~/fh/c/student> bin/WrongIndex
-1073744294
sep@pc305-3:~/fh/c/student>
```

Allerdings wird hier irgendwo in den Speicher gegriffen und der zufällig dort gespeicherte Wert gelesen.

1.2 Reihungen sind nur Zeiger

Wir können Funktionen schreiben, die Reihungen als Parameter übergeben bekommen. Diese Reihungen lassen sich im Funktionsrumpf ganz normal als Reihungen behandeln:

AddInts.c

```c
#include <stdio.h>

int add(int xs [],int argc){
  int result=0;
  int i=0;
```

```
  for (;i<argc;i++){
    result=result+xs[i];
  }
  return result;
}

int main(){
  int xs [] = {1,2,3,4};
  printf("%i\n", add(xs,4));
  return 0;
}
```

Wir können jetzt einmal in diesem Programm einen Fehler einbauen, um zu sehen, als was für einen Typ der Compiler unseren Reihungsparameter betrachtet:

WrongArrayParameterError.c

```
#include <stdio.h>

int add(int xs [],int argc){
  int result=0;
  int i=0
  for (;i<argc;i++){
    result=result+xs[i];
  }
  return result;
}

int main(){
  int xs [] = {1,2,3,4};
  printf("%i\n", add(xs,xs));
  return 0;
}
```

Auf diese Weise erzwingen wir eine Fehlermeldung des Compilers, in der die erwarteten Parametertypen der Funktion `add` angegeben werden. Der Kompilierversuch führt zu folgender Fehlermeldung:

Shell

```
sep@pc305-3:~/fh/c/student> gcc -Wall src/WrongArrayParameter.c
src/WrongArrayParameter.c: In function `main':
src/WrongArrayParameter.c:14: warning: passing arg 2 of `add' makes integer from pointer without a
sep@pc305-3:~/fh/c/student>
```

Der Übersetzer ist der Meinung, der zweite Parameter, den wir der Funktion `add` übergeben, ist ein Zeiger. Der Übersetzer macht aus einem Reihungsparameter implizit einen Zeiger. Und zwar einen Zeiger auf das erste Element des Reihungsparameters. Ein Array `int xs []` ist also nichts weiter als ein Zeiger des Typs `int* xs`.

Die Schreibweise der eckigen Klammern für Reihungen für den Elementzugriff, lässt sich somit als eine versteckte Zeigerarithmetik interpretieren. `xs[i]` bedeutet dann: `*(xs+i)`.

Tatsächlich können wir ohne die Syntax der eckigen Klammern mit Reihungen arbeiten. Obiges Programm lässt sich auch wie folgt formulieren:

NoBrackets.c

```c
#include <stdio.h>

int add(int xs [],int argc){
  int result=0;
  int i=0;
  for (;i<argc;i++){
    result=result + *(xs+i);
  }
  return result;
}

int main(){
  int xs [] = {1,2,3,4};
  printf("%i\n", add(xs,4));
  return 0;
}
```

Auch diese Version hat als Ergebnis die 10.

1.3 Dynamische Arraygrößen

Ursprünglich ist C so konzipiert, dass die Anzahl der Elemente einer Reihung während der Kompilierung feststehen muss und nicht dynamisch im Programmablauf errechnet werden kann.

Betrachten wir folgende kleine (unsinnige) Funktion.

WrongDynArray.c

```c
 void f(int n){
  int xs[n];
  *xs = 42;
}
```

Übersetzen wir diese mit den entsprechend strengen Optionen `--ansi` und `--pedantic`, dann bekommen wir eine entsprechende Meldung.

Shell

```
$ gcc -c --ansi --pedantic WrongDynArray.c
WrongDynArray.c: In function ‘f’:
WrongDynArray.c:2:3: warning: ISO C90 forbids variable length array ‘xs’ [-Wvla]
    2 |   int xs[n];
      |   ^~~
```

Will man also mit dynamisch zu ermittelnden Elementanzahlen vonb Reihungen arbeiten, geht eigentlich kein Weg an der Speicherallokation mit `malloc` vorbei. Es zeigt sich, dass in C ein

Array immer nur ein Zeiger auf das erste Element ist. Nun erklärt sich auch, warum Informatiker mit dem Index 0 beginnen. Es handelt sich bei einem Index um die Anzahl der Elemente, die zu überspringnen sind.

Aufgabe 1 Schreiben sie eine Funktion die die Reihenfolge der Elemente des Arrays gerade einmal umdreht.

Hierzu zunächst der Beginn der Kopfdatei:

ZeigerInC.h

```
#ifndef AL__H__
#define AL__H__

#include "MemoryTest.h"
#include <stdbool.h>

void reverse(int* xs,int length);
```

Die passenden Includes der Implementierungsdatei:

ZeigerInC.c

```
#include "ZeigerInC.h"
#include <stdlib.h>
#include <stdbool.h>
```

Und nun ist die Imopkemengtiuerung zu schreiben:

ZeigerInC.c

```
void reverse(int* xs,int length){
}
```

1.4 Reihungen als Rückgabewert

Im letzem Abschnitt haben wir bereits gesehen, dass bei der Parameterübergabe von Reihungen diese implizit als Zeiger auf das erste Element behandelt werden. Gleiches gilt auch für die Rückgabewerte von Funktionen. Dadurch lassen sich einige Probleme einhandeln. Versuchen wir einmal eine einfache Funktion zu schreiben, die eine neue Reihungsvariable anlegt und diese als Ergebnis zurückgibt:

WrongArrayReturnError.c

```
#include <stdio.h>

int*  newArray(int length){
  int result [length];
```

```
  int i=0;
  for (;i<5;i++)result[i]=i;
  return result;
}

int main(){
   int* xs = newArray(5);
  int i=0;
   for (;i<5;i++){
     printf("%i ",xs[i]);
   }
   printf("\n");
   return 0;
}
```

Die Übersetzung dieser Klasse glückt zwar, aber der Übersetzer gibt uns eine ernst zu nehmende Warnung:

Shell

```
sep@pc305-3:~/fh/c/student> gcc -std=c99 src/WrongArrayReturn.c
src/WrongArrayReturn.c: In function `newArray':
src/WrongArrayReturn.c:7: warning: function returns address of local variable
sep@pc305-3:~/fh/c/student> .
```

Die Warnung ist insofern ernst zu nehmen, als dass wir tatsächlich ein undefiniertes Laufzeitverhalten bekommen:

Shell

```
sep@pc305-3:~/fh/c/student> ./a.out
0 134518380 808989496 134513155 1074069176
sep@pc305-3:~/fh/c/student>
```

Den Grund hierfür kennen wir bereits aus unseren Kapitel über Zeiger. Implizit behandelt der Übersetzer die Zeile `return result` als eine Rückgabe des Zeigers auf die lokale Variable `result`. Eine lokale Variable innerhalb einer Funktion, wird im Speicher nach Beendigung der Funktion wieder freigegeben. Zeigen wir noch auf diese Variable, so werden wir an ihrer Speicherzelle undefinierte Werte sehen.

Die einzige vernünftigen Möglichkeiten der Rückgabe von Reihungen in C sind die Rückgabe eines als Parameter überreichten Zeiger auf eine Reihung, oder aber der Zeiger auf eine dynamisch neu erzeugte Reihung:

ReturnArray.c

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

int* newArray(int length){
  int* result =  (int*)malloc(sizeof(int[length]));
  int i=0;
```

```
  for (;i<5;i++)result[i]=i;
  return result;
}

int main(){
  int* xs = newArray(5);
  int i=0;
  for (;i<5;i++){
    printf("%i ",xs[i]);
  }
  printf("\n");
  return 0;
}
```

Dieses Programm übersetzt ohne Warnung und liefert die erwartete Ausgabe.

Shell

```
sep@pc305-3:~/fh/c/student> ./bin/ReturnArray
0 1 2 3 4
sep@pc305-3:~/fh/c/student>
```

Tatsächlich findet man es relativ selten in C Programmen, dass eine Funktion eine Reihung als Rückgabewert hat. Zumeist bekommen Funktionen Reihungen als Parameter übergeben und manipulieren diese.

1.5 Funktionen höherer Ordnung für Reihungen

Wir haben bereits gesehen, dass Funktionen über Funktionszeigern an andere Funktionen als Parameter übergeben werden können. Im Zusammenhang mit Reihungen eröffnet dieses ein mächtiges Programmierprinzip. Fast jede Funktion, die sich mit Reihungen beschäftigt, wird einmal jedes Element der Reihung durchgehen und dazu eine `for`-Schleife benutzen. Wie wäre es denn, wenn man dieses Prinzip verallgemeinert und eine allgemeine Funktion schreibt, die einmal jedes Element einer Reihung betrachtet und etwas mit ihm macht. Was mit jedem Element zu tun ist, wird dieser Funktion als Funktionszeiger übergeben.

Funktionen, die als Parameter Funktionen übergeben bekommen, werden auch als Funktionen höherer Ordnung bezeichnet.

Beispiel 1.3 Wir schreiben für Reihungen ganzer Zahlen die Funktion `map`, die jedes Element einer Reihung mit einer übergebenen Funktion umwandelt:

IntMap.c

```
#include <stdio.h>

void map(int xs [],int l,int (f) (int)){
  int i=0;
  for (;i<l;i++)  xs[i]=f(xs[i]);
```

```
}
```

Wir stellen drei Funktionen, die eine ganze Zahl als Parameter und Ergebnis haben zur Verfügung:

IntMap.c

```
int add5(int x){return x+5;}
int square(int x){return x*x;}
int doubleX(int x){return 2*x;}
```

Jetzt können wir die Funktion `map` verwenden, um für jedes Element einer Reihung, eine derartige Funktion anzuwenden. Zur Ausgabe von Reihungen schreiben wir eine kleine Hilfsfunktion:

IntMap.c

```
void printArray(int xs [],int l){
  printf("[");
  int i=0;
  for (;i<l;i++){
    printf("%i",xs[i]);
    if (i!=l-1) printf(", ");
  }
  printf("]\n");
}

int main(){
  int xs [] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
  map(xs,15,add5);
  printArray(xs,15);
  map(xs,15,square);
  printArray(xs,15);
  map(xs,15,doubleX);
  printArray(xs,15);
  return 0;
}
```

Und hier die Ausgabe des Programms:

Shell

```
sep@pc305-3:~/fh/c/student> bin/IntMap
[6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
[36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
[72, 98, 128, 162, 200, 242, 288, 338, 392, 450, 512, 578, 648, 722, 800]
sep@pc305-3:~/fh/c/student>
```

Zunächst wird auf jedes Element der Reihung 5 aufaddiert. Dann werden die Elemente quadriert und schließlich verdoppelt.

Aufgabe 2 Schreiben Sie eine Funktion `fold` mit folgender Signatur:

ZeigerInC.h

```
int fold(int xs [],int length,int op(int,int),int startV);
```

Die Spezifikation der Funktion sei durch folgende Gleichung gegeben:

fold(xs,l,op,st) = op(…op(op(op(st,xs[0]),xs[1]),xs[2])…,xs[l-1])

Oder bei Infixschreibweise der Funktion op gleichbedeutend über folgende Gleichung:

fold($\{x_0, x_1,,\ldots,x_{l-1}\}$,l,op,st) = st op x_0 op x_1 op x_2 op …op x_{l-1}

ZeigerInC.c

```
int fold(int xs [],int length,int op(int,int),int startV){
  return startV;
}
```

1.6 Zeichenketten

Bisher hatten wir noch keine Möglichkeit mit Zeichenketten, also Wörtern und Sätzen zu arbeiten. Einzelne Buchstaben ließen sich durch Daten des Typs `char` adequat[1] darstellen. Ein Wort oder längere Texte können nun als ein zusammen allokierten Speicherbereich von `char`-Werten aufgefasst werden. Damit kann ein Zeiger auf diesen Speicherbereich als Zeiger auf einen ganzen Text verstanden werden. So können wir Speicher anfordern und darin Wörter speichern:

Word1.c

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

int main(){
  char* hallo = malloc(5*sizeof(char));
  *(hallo+0) = 'H';
  *(hallo+1) = 'A';
  *(hallo+2) = 'L';
  *(hallo+3) = 'L';
  *(hallo+4) = 'O';

  printf("%c\n",*(hallo+3));
  return 0;
}
```

Wir haben ein Problem: wir können der Variablen `hallo` vom Typ `char*` im obigen Beispiel nicht ansehen, für wieviel Buchstaben dort Platz ist. Diese Information müssten wir seperat in

[1]Nunja, zumindest wenn man sich auf die 26 Buchstaben des lateinischen Alpabets beschränkt.

einer Variablen speichern. Also die eigentliche Wortlänge. Mit dieser Information lässt sich dann auch ein Wort komplett ausdrucken:

Word2.c

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

void printWord(unsigned int l, char* word){
  unsigned int i;
  for (i=0;i<l;i++)
    printf("%c",*(word+i));

  printf("\n");
}

int main(){
  unsigned int laenge = 5;

  char* hallo = malloc(laenge*sizeof(char));
  *(hallo+0) = 'H';
  *(hallo+1) = 'A';
  *(hallo+2) = 'L';
  *(hallo+3) = 'L';
  *(hallo+4) = 'O';

  printWord(laenge,hallo);

  return 0;
}
```

Das ist etwas umständlich, wenn man immer die Länge als Zweitinformation mit sich herumschleppen muss. Deshalb gibt es einen Standardtrick in C, wenn man mit Reihungen von Zeichen arbeitet. Als letztes Zeichen schreibt man noch den Nullbuchstaben, der als `'\0'` notiert wird. Dann kann man auf eine vorgegebene Länge verzichten und bei der Ausgabe abbrechen, wenn das Nullzeichen erreicht wurde.

Word3.c

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

void printWord( char* word){
  int i;
  for (i=0;*(word+i)!='\0';i++)
    printf("%c",*(word+i));

  printf("\n");
}

int main(){
  char* hallo = malloc(6*sizeof(char));
  *(hallo+0) = 'H';
```

```
  *(hallo+1) = 'A';
  *(hallo+2) = 'L';
  *(hallo+3) = 'L';
  *(hallo+4) = 'O';
  *(hallo+5) = '\0';
  printWord(hallo);

  return 0;
}
```

Die oben dargestellte Form Texte zu verarbeiten ist der Standardweg in C. Dieses ist in die Sprache eingebaut. Wenn ein Text in doppelten Anführungszeichen steht, wie wir es ja schon oft in den Funktionsaufrufen von `printf` gesehen haben, dann wird der Text auf eben diese Weise in einen über den Typ `char*` beschriebenen Speicherbereich wie oben von Hand durchgeführt gespeichert. Somit lässt sich die Erzeugung des obigen Textes im Speicher viel einfacher schreiben:

Word4.c

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

void printWord( char* word){
  int i;
  for (i=0;*(word+i)!='\0';i++)
    printf("%c",*(word+i));

  printf("\n");
}

int main(){
  char* hallo = "HALLO";
  printWord(hallo);

  return 0;
}
```

Ein weiterer Luxus ist, dass die Funktion `printf` auch darauf vorbereitet ist, derartige Zeichenketten auszugeben, so dass wir auf ein eigenes `printWord` verzichten können. Hierzu ist der Funktion `printf` als Formatierungsanweisung `%s` mitzugeben:

Word5.c

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

int main(){
  char* hallo = "HALLO";
  printf("%s\n",hallo);

  return 0;
}
```

Aufgabe 3 Schreiben Sie eine Funktion `int length(char* text)`, die angibt, wieviel Zeichen in dem Speicherbereich von `text` gespeichert sind.

ZeigerInC.h

```
int length(char* text);
```

ZeigerInC.c

```
int length(char* text){
  return 0;
}
```

Aufgabe 4 Schreiben Sie eine Funktion `bool isPallindrom(char* text)`, die testet, ob der übergebene Text vorwärts und rückwärts gelesen identisch ist.

ZeigerInC.h

```
bool isPallindrom(char* text);
```

ZeigerInC.c

```
bool isPallindrom(char* text){
  return true;
}
```

Aufgabe 5 Schreiben Sie eine Funktion `int leseAlsZahl(char* text)`, die Zeichenkette als Zahl interpretiert. Für die Zeichenkette `"42"` soll also die Zahl 42 als Ergebnis zurückgegeben werden:

ZeigerInC.h

```
int leseAlsZahl(char* text);
```

ZeigerInC.c

```
int leseAlsZahl(char* text){
  int result=0;

}
```

Aufgabe 6 Schreiben Sie eine Funktion `char* concat(char* text1,char* text2)`, die eine neue Zeichenkette erzeugt, die aus den beiden Zeichenketten der Parameter aneinandergehängt besteht.

ZeigerInC.h

```
char* concat(char* text1,char* text2);
```

ZeigerInC.c

```
char* concat(char* text1,char* text2){
  return NULL;
}
```

2 Array-basierte generische Liste

Wir wollen möglichst ähnlich wie in Java die Listen implementieren, die als Datenspeicher einen Array enthalten.

2.1 Datenstruktur

Zunächst einmal, sollen, so wie es im Prinzip in Java auch ist, beliebige Objekte in der Liste gespeichert werden. Hierzu definierebn wir ein Objekt als eine beliebige Referenz:

ZeigerInC.h

```
typedef void* Object;
```

Statt der Typvariablen generischer Typen, definieren wir für diese einen Typ:

ZeigerInC.h

```
typedef Object E;
```

Eine Arrayliste ist eine Struktur aus der Anzahl der Elemente, der Kapazität des internen Speichers und eines Array für die Elemente:

ZeigerInC.h

```
typedef struct{
  unsigned int size;
  unsigned int capacity;
  E* store;
}AL;
```

Für die Arrayliste definieren die folgenden Funktionen:

ZeigerInC.h

```
AL alNew();
void alDelete(AL this);
void alAdd(AL* this, E e);
void alForEach(AL this, void f(E));
E alFold(AL this, E start, void op(E,E));
```

2.2 Konstrukorfunktion

Der Konstruiktzor erzeugt eine leer Liste mit der Kapazität von 5 Elementen:

ZeigerInC.c

```
AL alNew(){
  AL this;
  this.size = 0;
  this.capacity = 5;
  this.store = (E*)malloc(this.capacity*sizeof(E));
  return this;
}
```

2.3 Destruktorfunktion

Der Destruktor löscht die Liste aus dem Speicher, aber nicht die darain referenzierten Objekte:

ZeigerInC.c

```
void alDelete(AL this){
  free(this.store);
}
```

2.4 Einfügen von Elementen

Um ein Element einzufügen, muss geschaut werden, ob die Kapaziät ausreicht. Ansonsten muss zunächst mehr Speicher neu allokiert werden:

ZeigerInC.c

```
void alAdd(AL* this, E e){
  if (this->size>=this->capacity){
    this->capacity =5+this->capacity;
    this->store = (E*)realloc(this->store,(this->capacity)*sizeof(E));
```

```
  }

  this->store[this->size++] = e;
}
```

2.5 Eine foreach-Funktion

Wir können wie gewohnt auch eine einfache foreach-Funktion schreiben:

ZeigerInC.c

```
void alForEach(AL this, void f(E)){
  unsigned int i;
  for (i=0;i<this.size;i++) f(this.store[i]);
}
```

2.6 Eine Faltung

Auch eine Faltung über die Elemente lässt sich definieren:

ZeigerInC.c

```
E alFold(AL this, E start, void op(E,E)){
  unsigned int i;
  for (i=0;i<this.size;i++)
    op(start,this.store[i]);
  return start;
}
```

Aufgabe 7 Implementieren Sie die Header Datei für Array basierte Listen. Sie können dabei jeweils die aus der Javaaufgabe bekannten Lösungen anpassen.

Die Kopfdatei hat hierzu die folgenden zusätzlichen Typsynonyme und Signaturen:

ZeigerInC.h

```
typedef bool Predicate(Object);
typedef bool EQ(Object,Object);
typedef int Comparator(Object,Object);

E alGet(AL this, int i);
void alRemove(AL* this, int i);
void alAddAll(AL* this, AL that);
bool alContainsWith(AL this,Predicate pred);
AL alSublist(AL this, int i, int l);
bool alStartsWith(AL this,AL that,EQ eq);
```

```
bool alEndsWith(AL this,AL that,EQ eq);
void alReverse(AL* this);
void alInsert(AL* this,int i, E e);
void alSortBy(AL* this,Comparator cmp);

#endif
```

- Indexzugriff auf die Elemente:

ZeigerInC.c

```
E alGet(AL this, int i) {
  /* ToDo */
  return NULL;
}
```

- Das Element am übergebenen Index soll gelöscht werden, sofern dieser Index existiert. Die Liste wird um 1 kürzer und alle nachfolgenden Elemente rutschen in der Liste eine Position nach vorne.

ZeigerInC.c

```
void alRemove(AL* this, int i) {
  /* ToDo */
}
```

- Alle Elemente der übergenenen Liste sollen in die Liste hinten hinzugefügt werden.

ZeigerInC.c

```
void alAddAll(AL* this, AL that) {
  /* ToDo */
}
```

- Ist genau dann wahr, wenn ein Element das übergebene Prädikat erfüllt.

ZeigerInC.c

```
bool alContainsWith(AL this,Predicate pred) {
  /* ToDo */
  return false;
}
```

- Eine neue Teilliste dieser Liste. Die neue Liste habe die Länge l und besteht aus den Elementen ab Index `i` von dieser Liste. Ist `l` zu lang, sor wird eine kürzere Teilliste erzeugt.

ZeigerInC.c

```
AL alSublist(AL this, int i, int l) {
  /* ToDo */
  return result;
}
```

- Genau dann wahr, wenn die übergebene Liste ein Präfix dieser Liste ist.

ZeigerInC.c

```
bool alStartsWith(AL this,AL that,EQ eq) {
  /* ToDo */
  return true;
}
```

- Genau dann wahr, wenn die übergebene Liste eine Endliste dieser Liste ist.

ZeigerInC.c

```
bool alEndsWith(AL this,AL that,EQ eq) {
  /* ToDo */
  return true;
}
```

- Ändert die Reihenfolge der Liste, indem diese umgedreht wird.

ZeigerInC.c

```
void alReverse(AL* this) {
  /* ToDo */
}
```

- Das Element soll an dem übergebenen Index eingefügt werden. Ist der Index zu groß, wird am Ende der Liste eingefügt. Ist der Index negativ, so ändert sich die Liste nicht.

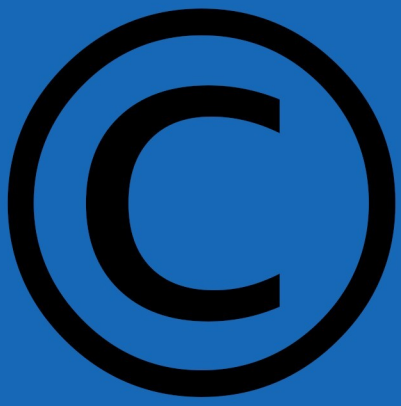
ZeigerInC.c

```
void alInsert(AL* this,int i, E e) {
  /* ToDo */
}
```

- Die Liste wird sortiert. Als Sortierfunktion wird der übergenen Komperator verwendet. Sie können hierzu im Internet nach dem Bubble-Sort Algorithmus suchen, der mit Hilfe zweier Schleifen mehrfach über die Liste geht und benachbarte Element tauscht, falls die in falscher Reihenfolge stehen.

ZeigerInC.c

```
void alSortBy(AL* this,Comparator cmp) {
  /* ToDo */
}
```



Zeigertypen
Adressoperator
Referenzierung und Derefenzierung
dynamische Daten
Allokation und Freigabe von Speicher