Funktionen, Zeiger und Arrays in C

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

1 Funktionen

Mit Hilfe von Funktionen kann man (wie mit Prozeduren bzw. Methoden in Java) eine Problemstellung in Teilprobleme abstrahieren. In C muss jeder ausführbare Code innerhalb einer Funktion stehen.
Hier das Beispiel zur Berechnung des Absolutwerts, das wir auch in Java diskutiert haben:

```
1  #include <stdio.h>
2
3  int abs(int n) {
4      if (n >= 0) {
5        return n;
6      } else {
7        return -n;
8      }
9    }
10
11 int main(void){
12     int x;
13     int i;
14
15     scanf("%d", &i);
16     x = abs(i);
17     printf("%d\n", x);
18 }
```

Funktionen können rekursiv sein.

```
1  #include<stdio.h>
2
3  int fac(int n)
4  {
5    if (n == 0) {
6      return 1;
7    } else {
```

```
8       return n * fac(n-1);
9     }
10  }
11
12  int main(void)
13  {
14    int n = 5;
15    printf("fac(5) = %d\n", fac(5));
16  }
```

Frage 1: Bis zu welchem n lässt sich so die Fakultät berechnen?
Wie muss man das Programm anpassen, damit man auch die Fakultät für größere n berechnen kann?

Funktionen, die keinen Rückgabewert liefern, haben `void` als Rückgabetyp. Bei Funktionen, die keine Parameter nehmen, verwenden wir `void` anstelle der Parameterdeklaration.
Enthält eine Funktionsdeklaration keinerlei Parameterinformation (also: leere Klammern), wird implizit angenommen, dass die Funktion beliebig viele Parameter nimmt. Dies kann dazu führen, dass der Compiler weniger Typchecks durchführen kann.

```
1   int f(int x) {
2     return x;
3   }
4
5   int g() {
6     return 8;
7   }
8
9   int h(void) {
10      return 3;
11  }
12
13  int main(void)
14  {
15      g(5);
16      f(5);
17      h(3);
18      return 0;
19  }
```

In obigem Beispiel liefert der `clang` Compiler[1] folgende Ausgabe:

```
$ clang noparams.c -o noparams
noparams.c:18:8: warning: too many arguments in call to 'g'
    g(5);
```

[1] Version Apple LLVM version 7.0.0

```
    ~  ^
noparams.c:20:7: error: too many arguments to function call, expected 0, have 1
    h(3);
    ~ ^
noparams.c:11:1: note: 'h' declared here
int h(void) {
^
1 warning and 1 error generated.
```

Die Verwendung von `g(5)` liefert lediglich eine Warnung, `h(3)` einen Fehler, der das Compilieren verhindert. Andere Compiler können eine solche Warnung auch unterlassen, so dass man auf die falsche Verwendung der Funktion **nicht immer** hingewiesen wird.

Reihenfolge von Funktionen Die Reihenfolge der Definition von Funktionen ist in C von Bedeutung. Eine Funktion muss **vor** ihrer Verwendung deklariert werden, entweder mit Implementierung oder Funktionsprototyp.

```
1   #include <stdio.h>
2
3   /* Funktionsprototyp */
4   double square(double x);
5
6   int main(void)
7   {
8     double x = 2.;
9     printf("%f %f\n", x, square(x));
10    return 0;
11  }
12
13  /* Funktionsdefinition */
14  double square(double x)
15  {
16    return x * x;
17  }
```

Die Deklaration der Funktion `sqrt` in Zeile 4, auch ***Funktionsprototyp*** genannt, muss mit der Definition der Funktion in Zeile 16ff übereinstimmen.[2] Ein Funktionsprotoyp besteht aus Rückgabetyp, Funktionsname und Typen sowie Anzahl der Parameter; er wird durch ein Semikolon abgeschlossen.
Bei verschränkt rekursiven Funktionen ist es unvermeidbar, Funktionsprototypen zu verwenden.

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 #include <stdlib.h>
```

[2] Parameternamen müssen nicht übereinstimmen und sind optional, zu Dokumentationszwecken aber sinnvoll.

```
4
5  bool ist_ungerade(int n);  /* Funktionsprototyp */
6
7  bool ist_gerade(int n)
8  {
9    if (n > 0) {
10     return ist_ungerade(n-1);
11   } else {
12     return true;
13   }
14 }
15
16 bool ist_ungerade(int n)   /* Funktionsdefinition */
17 {
18   if (n > 0) {
19     return ist_gerade(n-1);
20   } else {
21     return false;
22   }
23 }
24
25 /* Ermittelt, ob ein Integer gerade oder ungerade ist */
26 int main(void)
27 {
28   int n = -987;
29   if (ist_gerade(abs(n)))  /* verwendet den Absolutwert */
30     printf("%d ist gerade!\n",n);
31   else
32     printf("%d ist ungerade!\n",n);
33   return 0;
34 }
```

1.1 Lokale vs. globale Variablen

Für den Gültigkeitsbereich und die Sichtbarkeit von Variablen gelten in C folgende Regeln:

- Wird eine Variable in einem Anweisungsblock () deklariert, reichen der Gültigkeitsbereich und die Lebensdauer dieser Variablen von der Deklarationsstelle bis zum Ende des Anweisungsblocks.
- Wird eine Variable außerhalb von Funktionen deklariert, reichen Gültigkeitsbereich und Lebensdauer von der Deklaration bis zum Dateiende.

Es gilt außerdem, dass eine äußere Deklaration nicht mehr sichtbar (***verschattet***) ist, wenn eine Variable mit demselben Namen in einem inneren Block deklariert wird. Beim Verlassen des inneren Blocks ist die Variable des äußeren Blocks wieder sichtbar. Ebenso können globale Variablen durch lokale Variablen verschattet werden.

Frage 2: Was ist die Ausgabe des folgenden Programms?

```c
#include <stdio.h>

int i = 90; /* globale Variable */

int main (void)
{
  int j = 25;  /* lokale Variable */
  {
    int j = 3; /* lokale Variable */
    printf("%d ",j);
  }
  printf("%d ",j);
  printf("%d\n",i);
  return 0;
}
```

2 Zeiger

Eine Variable umfasst die folgenden Aspekte:

- Der *Bezeichner* erlaubt es, die Variable innerhalb ihres Gültigkeitsbereichs zu verwenden.
- Der aktuelle *Wert* kann gelesen oder durch einen anderen Wert ersetzt werden.
- Der *Datentyp* bestimmt, welche Werte eine Variable repräsentieren kann.
- Die *Speicheradresse* gibt den Ort im Speicher an, wo der Wert abgespeichert wird.

Wir haben Variablen bisher immer über ihren Namen angesprochen, um ihr einen Wert zuzuweisen bzw. den Wert auszulesen. Eine Variable kann in C aber auch direkt über die Adresse angesprochen werden, die sie im Rechner repräsentiert. [3]
Ein Zeiger (engl. *pointer*) ist ein Wert, der die Speicheradresse bezeichnet. Man sagt, ein Zeiger ***referenziert*** einen Wert, wenn dieser Wert an der entsprechenden Speicheradresse gespeichert ist. Zeiger können beliebige Datentypen, ja sogar Funktionen referenzieren.

```c
#include <stdio.h>

int main(void) {
   int x = 7;
```

[3] Dies ist in Java **nicht** möglich!

```
5       int *y = &x;
6       *y = 10;
7       return 0;
8   }
```

Die tatsächliche Adresse einer Variablen wird bei der Deklaration vom System bestimmt; darauf hat ein Programmierer keinen Einfluss. Verschiedene Programmausführungen können daher auch verschiedene Adressen liefern.

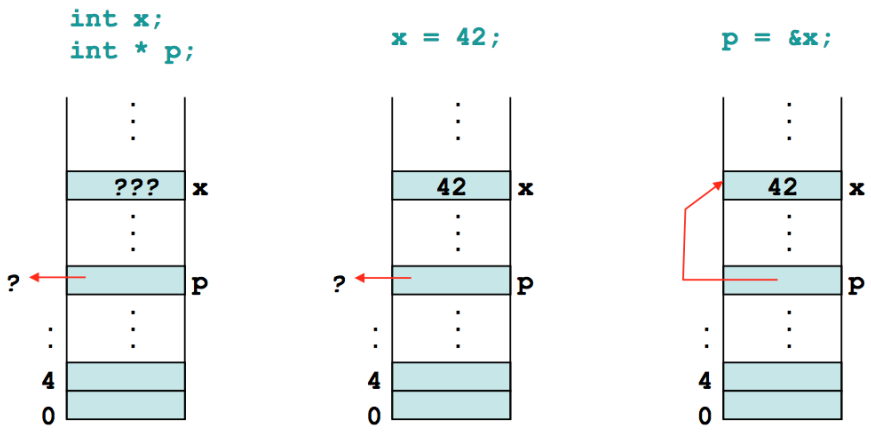
Deklaration von Zeigern Der Typbezeichner für einen Zeiger auf einen Wert vom Typ `t` ist `t*` . Bei der Deklaration von Zeigern muss man also angeben, Werte welchen Typs an der Speicheradresse zu finden sind.

```
int *a, *b;  // a und b Zeiger auf int
int *y, x ;  // y Zeiger und x int
int* f, g;   // f Zeiger und g int !!
```

Die letzte Variante suggeriert fälschlicherweise, dass auch g eine Zeigervariable sei; sie sollte vermieden werden. Der Asterisk * bei der Typbezeichnung wird daher – analog zur Benutzung zur Ermittlung des Werts – in der Regel bei der Deklaration an den Variablen vorne anhängt, nicht an den Typen (Variante 1 und 2).

Zuweisung von Zeigern Der ***Adressoperator*** & liefert einen Zeiger; d.h. eine Adresse.

```
int x  = 42;  // x ist ein Integer mit Wert 42
int *p = &x;   // p ist ein Zeiger auf x
```



Der "Null-Zeiger" ist ein spezieller Wert, der angibt, dass ein Zeiger **nicht** auf eine gültige Speicheradresse verweist.

```
int *p1 = NULL;            // Null pointer
```

`NULL` ist eine Null-Pointer-Konstante; in C99 wird sie als Integer-Wert 0 implementiert, der als Zeiger interpretiert wird.
Der Adress-Operator `&` liefert **niemals** einen Null-Zeiger.

Frage 3: Was ist der Unterschied zwischen einem Null-Zeiger und einem nicht initialisierten Zeiger?

Dereferenzieren von Zeigern Der Wert an Adresse `y` wird über den Indirektionsoperator `*`, z.B. `*y`, ermittelt. [4]

```
1  #include <stdio.h>
2
3  int main(void) {
4     int x = 7;
5     printf("Die Adresse von x ist %p \n",&x);
6     printf("Der Wert von x ist %d \n",    x);
7
8     int *y = &x;
9     printf("Die Adresse von y ist %p \n",&y);
10    printf("Der Wert von y ist %p \n",    y);
11    printf("Der Wert an y ist %d \n",    *y);
12
13    *y = 10;
14    printf("Nach der Zuweisung:  *y = 10 \n");
15    printf("Die Adresse von x ist %p \n",&x);
16    printf("Der Wert von x ist %d \n",    x);
17    printf("Die Adresse von y ist %p \n",&y);
18    printf("Der Wert von y ist %p \n",    y);
19    printf("Der Wert an y ist %d \n",    *y);
20
21    return 0;
22 }
```

Das obige Programm liefert beispielsweise folgende Ausgabe; die konkreten Adressen können sich aber je nach Ausführung unterscheiden:

```
Die Adresse von x ist 0x7fff57c3b7d8
Der Wert von x ist 7
Die Adresse von y ist 0x7fff57c3b7d0
Der Wert von y ist 0x7fff57c3b7d8
Der Wert an y ist 7
Nach der Zuweisung:  *y = 10
Die Adresse von x ist 0x7fff57c3b7d8
Der Wert von x ist 10
Die Adresse von y ist 0x7fff57c3b7d0
Der Wert von y ist 0x7fff57c3b7d8
Der Wert an y ist 10
```

[4] Genauer: Der Indirektionsoperator `*` liefert das Datenobjekt, dessen Startadresse sich an der Speicherstelle eines Zeigers befindet.

Das Derefenzieren eines Zeigers, die zuvor nicht mit einer gültigen Adresse initialisiert wurde, ist undefiniert. Die Gefahr ist, dass einfach auf irgendeine Adresse im Arbeitsspeicher zurückgegriffen wird. Häufig bricht die Programmausführung mit einem ***Segmentation Fault*** ab, wenn die Speicherhardware einen unerlaubten Zugriff auf geschützte Bereiche des Speichers an das Betriebssystem meldet.
Typische Ursachen für Segmentation Faults sind:

- Dereferenzieren von Null-Zeigern
- Versuchter Zugriff auf nicht-existierende Speicheradressen bzw. Adressen, die außerhalb des Speicherbereichs liegen, der vom System zur Ausführung des Programm bereitgestellt wird
- Versuchter Zugriff auf Speicherbereiche, die geschützt sind (z.B. Schreibzugriff auf schreibgeschützte Bereiche)

Programme in C sind besonders anfällig für Segmentation Faults, da Programmierer hier selbst den Speicher verwalten müssen (⇒ Abschnitt Speichermanagement).

2.1 Parameterübergabe bei Funktionen

Bei Funktionsaufrufen werden in C immer (Kopien von) Werten als Parameter übergeben. Diese Vorgehensweise nennt man ***Call-by-value***. Veränderungen an den Parametern haben keinen Effekt an der Aufrufstelle.

Beispiel Folgende Funktion tauscht die Werte der Parameter `x` und `y`.

```
void swap1(int x, int y) {
   int temp;

   temp = x; /* Speichere den Wert von x in temp */
   x = y;    /* Speichere den Wert von y in x */
   y = temp; /* Speichere den Wert von temp in xy */

   return;
}
```

An der Aufrufstelle `swap1(a,b)` werden die Werte von `a` und `b` dabei nicht verändert. Wenn man will, dass tatsächlich die Werte von `a` und `b` durch den Aufruf modifiziert werden, müssen wir die jeweiligen Zeiger übergeben und die Werte an den Speicheradressen vertauschen:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void swap2 (int *x, int *y)
5 {
6   int temp = *x;  /* Speichere den Wert an x in temp */
7   *x = *y;        /* Speichere den Wert an y an x */
```

```
8     *y = temp;      /* Speichere den Wert von temp an y */
9  }
10
11 int main(void) {
12     int a = 86;
13     int b = 42;
14     printf("Vor swap:  a = %d, b = %d \n", a, b);
15     swap2(&a, &b);
16     printf("Nach swap: a = %d, b = %d \n", a, b);
17     return 0;
18 }
```

Auch in dieser Variante handelt es sich um Parameterübergabe über Call-by-Value: Die übergebenen Werte sind in diesem Fall die Adressen. Diese werden in der Funktion nicht geändert!

Zeiger als Parameter sind insbesondere dann nützlich, wenn eine Funktion mehr als ein Ergebnis liefern soll. In C findet man diese Verwendung von Zeigern häufig. So liefert beispielsweise die Funktion `scanf` mehrere Ergebnisse: Der Rückgabewert liefert, wie viele Elemente aus dem Eingabestrom gelesen wurden bzw. EOF, falls ein Problem auftrat, bevor ein Element gelesen wurde (z.B. Ende der Eingabe erreicht); die Ergebnisse des Einlesens werden über Zeiger an den Aufrufer zurückgegeben.

```
int i, j;
j = scanf("%d", &i);
/* schreibt den eingelesenen Wert an die Adresse von i */
```

Falls nach dem Aufruf $j > 0$, dann enthält `i` den eingelesenen int-Wert. Andernfalls ist `i` nicht initialisiert!

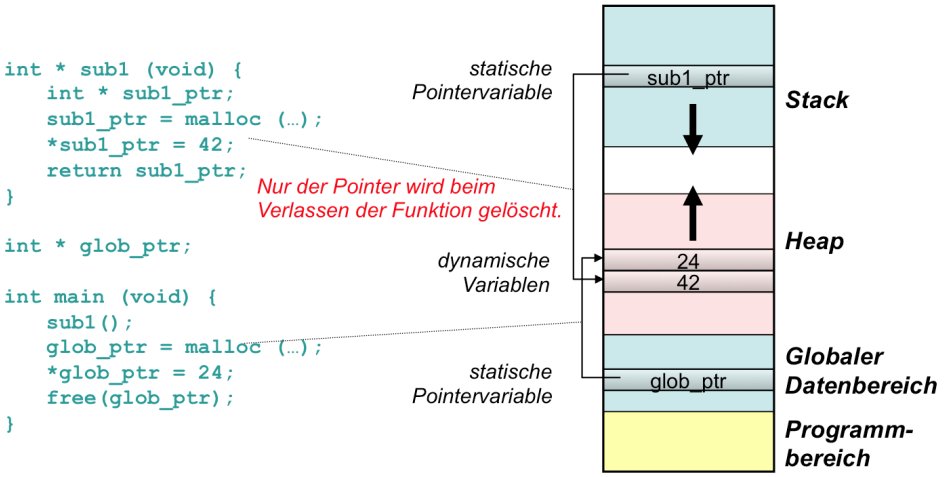
- In Java gibt es nicht die Möglichkeit Adressen von Objekten und Variablen zu ermitteln.
- Die Verwendung von nicht-initialisierten Variablen wird in Java durch den Compiler verhindert.
- Das Dereferenzieren von `null` liefert **in Java immer** eine `NullPointerException`.

3 Dynamische Speicherverwaltung

Für C Programme unterscheiden wir die folgenden Speicherbereiche:

- Der ***Programmbereich*** enthält den kompilierten Programmcode Maschinenbefehle.

- Im ***globalen Datenbereich*** befinden sich alle Daten, die vom Beginn des Programms bis zum Programmende verfügbar sind (u.a. globale Variablen, String-Literale).
- Im ***Stack*** werden die Funktionsinkarnationen mit ihren lokalen Variablen verwaltet.
- Der ***Heap*** stellt Speicherplatz für die dynamischen Speicherverwaltung zur Verfügung.



Daten, die auf dem Stack gespeichert sind, werden automatisch verwaltet. Mit jedem Funktionsaufruf wird ein neuer Stackframe für eine Funktion erzeugt; bei Beenden der Funktion wird der entsprechende Speicherbereich automatisch wieder freigegeben. Die Lebensdauer der lokalen Variablen, wie `sub1_ptr`, endet mit dem Verlassen der Funktion. Die lokal im Stackframe allozierten Daten, wie Variablen, können daher nicht beliebig über Funktionsaufrufe hinweg verwendet werden.
Falls Daten über das Funktionsende hinweg benötigt werden, müssen sie auf dem Heap gespeichert werden; dazu gibt es mehrere Funktionen in der Bibliothek `stdlib.h` Die Funktion `void *malloc(size_t n)` ist eine der Funktionen in C, mit der Speicher auf dem Heap alloziert werden kann. Der Parameter `n` gibt an, wie viele Bytes an Speicher angefordert werden. Ist die Allokation erfolgreich, wird ein Zeiger auf den Beginn des allozierten Speicherbereichs zurückgeliefert. Andernfalls wird `NULL` als Fehlerindikator geliefert.
Das folgende Beispiel alloziert einen Speicherbereich für 10 Integer auf dem Heap.

```
int *p = malloc(10 * sizeof (int));
if (p == NULL) {
} else {
    /* Allokation war erfolgreich; der Speicherbereich kann jetzt
     verwendet werden!  */
```

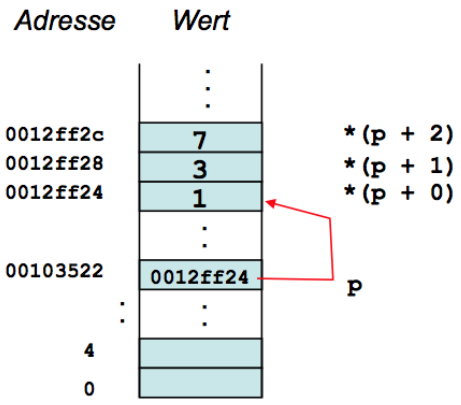
```
    ...
}

/* Wenn der Speicherbereich nicht mehr benötigt wird, geben wir
    ihn frei */
free(p);
```

Auf dem Heap allozierter Speicher ist vom Zeitpunkt der Allokation bis zum Deallokation (oder Programmende) verfügbar. Um Speicherlecks (engl. *memory leaks*) zu vermeiden, sollten Programmierer nicht mehr benötigten Speicher nach der Verwendung wieder freigeben. Dazu dient die Funktion `void free(void *pointer)`. Der Semantik von Zugriff auf freigegebenen Speicher und das mehrmalige Freigeben des gleichen Speicherbereichs ist nicht definiert!
`malloc` und `free` operieren mit Zeigern vom Typ `void *`; Ein Zeiger vom Typ `void*` verweist auf Daten unbekannten Typs und kann nicht dereferenziert werden. Es ist aber möglich, dass nach der Zuweisung an eine Zeiger-Variable vom Typ `(int *)` die einzelnen Elemente zugegriffen werden können.
`sizeof` liefert die jeweils notwendige Größe des Speicherbereichs in Byte, die im System für einen Datenwert benötigt wird. Die Verwendung von `sizeof` unterstützt daher die Portierbarkeit von Programmen auf verschiedene Systeme.

```
1  #include <stdio.h>
2
3  int main(void) {
4     printf("char      : %ld Byte\n" , sizeof(char));
5     printf("int       : %ld Bytes\n", sizeof(int));
6     printf("long      : %ld Bytes\n", sizeof(long int));
7     printf("float     : %ld Bytes\n", sizeof(float));
8     printf("double    : %ld Bytes\n", sizeof(double));
9     return 0;
10 }
```

Zugriff Auf die einzelnen Elemente im allozierten Speicherbereich kann folgendermaßen zugegriffen werden:

```
int *p =  malloc(3 * sizeof (int));
*(p+0) =  1;
*(p+1) =  3;
*(p+2) =  7;
```



Die Adressen der aufeinanderfolgenden int-Elemente differieren nicht um 1, sondern um die Größe eines Integers. Bei der "Berechnung" der Adresse spielt der Typ der Elemente eine wichtige Rolle: Er gibt an, wieviel Speicher benötigt wird, um ein Element abzuspeichern und damit auch, wo das nächste Element zu finden ist. Ist z.B. ein int-Wert 4 Bytes (= 32 Bit) groß, so wird bei int-Zeigern der Abstand mit dem Faktor 4 skaliert.

Man beachte die Klammerung:

- Der Ausdruck `*(pa + 1)` liefert den Wert an Adresse `pa` mit Abstand 1.
- Der Ausdruck `*pa + 1` liefert den Wert an Adresse `pa` und addiert zu diesem Wert 1.

Weitere Varianten zur dynamischen Speicherallokation Die Funktion `malloc` alloziert den Speicher, initialisiert ihn aber nicht, da dies in vielen Kontexten nicht notwendig ist. Die Variante `void *calloc(size_t n, size_t elementsize)` alloziert einen Speicherbereich der Größe `n` × `elementsize` und initialisiert ihn mit 0.
Beispiel:

```
/* Speicherplatz fuer 200 chars */
char *w = calloc(200, sizeof(char));
/* Speicherplatz fuer 10 int */
int *p = calloc(10, sizeof(int));
```

Die Funktion `void *realloc(void *pointer, size_t size)` erlaubt es, die Größe von Speicherbereichen zu erweitern bzw. zu verkleinern. Sie liefert einen Zeiger auf einen Speicherbereich der Größe `size`, der die gleichen Daten wie der durch `pointer` referenzierte Bereich enthält (bis zum Minimum der Größe von ursprünglichen und des neuen Bereichs). Neue Speicherregionen werden dabei nicht initialisiert. Falls die Allokation fehlschlägt, wird der ursprüngliche Bereich nicht verändert und `NULL` zurückgegeben.

Der Rückgabewert von `realloc` sollte immer zunächst einer temporären Variablen zugewiesen werden, da bei fehlgeschlagener Allokation sonst der Zeiger auf den ursprünglichen Bereich mit `NULL` überschrieben wird. Dies führt u.U. dann zu Speicherlecks.

```
int *p = (int *) malloc(initial_size);

/* .... spaeter ... */

void *tmp = realloc(p, bigger_size);
if (tmp != NULL) {
   p = tmp; /* OK */
} else {
   /* Problembehandlung */
   ...
}
```

4 Arrays

Arrays ermöglichen das Zusammenfassen und Verwalten von verwandten Daten gleichen Typs unter *einem* Namen. Diese Daten sind angeordnet und können durch einen Index adressiert werden.
Ein Array kann mit folgender Notation auf dem Stack alloziert werden:

```
int a[6];                  // Array mit Namen a mit 6 Integern
int b[6] = {0,0,1,0,0,0}; // Deklaration und Initialisierung
```

Zur Allokation auf dem Heap muss, wie oben erläutert, `malloc()` (bzw. `calloc`, `realloc`) und `free()` verwendet werden:

```
int* ar;
ar = (int *) malloc(sizeof(int) * 10);
/* Reserviert Speicher für 10 Integer-Variablen und lässt 'ar'
  auf den Speicherbereich zeigen. */

if (ar == NULL) {
  printf("Nicht genug Speicher vorhanden."); // Fehler ausgeben
  exit(1);                  // Programm mit Fehlercode abbrechen
}
free(ar);                            // gibt Speicher wieder frei
```

Die Indizes eines Arrays mit Größe n sind $0, 1, \ldots, n-1$. In C wird der Inhalt von Arrays beim Deklarieren nicht vorinitialisiert, dies muss vom Programmierer selbst getan werden.

```
for (int i = 0; i < 6; i++) {
  a[i] = 2 * i;
}
```

Anders als in Java löst der Zugriff auf einen Index außerhalb der Array-Grenzen nicht immer einen Fehler aus – das Verhalten ist undefiniert!

```
/* Weiterfuehrung des Beispiels */
printf("%d",a[-1]); /* undefiniertes Verhalten */
printf("%d",a[6]);  /* undefiniertes Verhalten */
```

4.1 Adressarithmetik auf Arrays

Zwischen Arrays und Zeigern besteht in C ein enger Zusammenhang: Jede Operation mit Array-Indizes kann man auch mit Zeigern formulieren.
Im obigen Beispiel ist `a` ein Array mit 6 int-Elementen. Dies wird im Speicher realisiert als ein Block von 6 aufeinanderfolgenden Elementen. `a[i]` bezeichnet dabei das Element an Position `i`. Deklariert man nun eine Variable `pa` als Zeiger auf einen int-Wert:

```
int *pa;
```

dann kann `pa` auf das Element an Position 0 von `a` verweisen;

```
pa = &a[0];
```

D.h. `pa` enthält die Adresse von `a[0]`. `&a[0]` ist dabei äquivalent zu `a`.
Die Zuweisung

```
x = *pa;
```

kopiert den Wert von `a[0]` nach `x`.
Die Adresse des Elements `a[i]` ist gegeben durch `pa + i`; und `*(pa + i)` liefert den Wert des Elements `a[i]`. Es ergibt sich außerdem, dass `a[i]` äquivalent zu `*(a + i)` ist; ebenso ist `&a[i]` äquivalent zu `(a + i)`.

> **Frage 4:** Schreiben Sie ein Programm, dass diese Äquivalenz an einem Beispiel zeigt!

Arrays und Zeiger-Variablen sind trotz vieler Gemeinsamkeiten in der Verwendung essentiell verschieden:

- Arrays kann man keinen Wert zuweisen; einer Zeiger-Variablen kann man hingegen eine andere Speicheradresse zuweisen.
- Für Arrays kann man die Größe (in byte) über `sizeof` ermitteln.

Arrays als Parameter Wird ein Array als Parameter an eine Funktion übergeben, so wird tatsächlich die Adresse des Anfangselements, also ein Zeiger, übergeben. [5] Anders als in Java verfügt man in der Funktion dann über keinerlei Information, wie viele Elemente in dem Array vorhanden sind. Es ist daher gute Programmierpraxis

[5] Man kann in C auch nur einen Teil eines Vektors an eine Funktion übergeben. Der Aufruf `f(&a[2])` liefert an eine Funktion `f` die Adresse des Teilarrays, das mit dem Element `a[2]` beginnt.

in C für Arrays eine Variable mit der Größe jeweils mitzuverwalten, die bei einem Funktionsaufruf dann ein weiterer Parameter ist.
Als Beispiel betrachten wir hier die Variante der Main-Funktion, die dem Programm beim Start Kommandozeilenparameter übergibt.

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5     // Ausgabe der Kommandozeilenparameter
6     for(int i = 0; i < argc; i++) {
7        printf("argv[%d] = %s \n", i, argv[i]);
8     }
9     return 0;
10 }
```

- Der Parameter `argc` beinhaltet die Anzahl an Argumenten.
- `argv` liefert die Adresse eines Arrays, das Strings (in C repräsentiert durch char-Arrays) enthält. Die Verwendung `argv[i]` liefert den String an Position i. An Position 0 steht dabei der Programmname.

Beispiel für Ausgabe des Programms:

```
$ ./commandline hello world
argv[0] = ./commandline
argv[1] = hello
argv[2] = world
```

Zusammenfassend:
In C ist der Wert einer Variablen oder eines Ausdrucks von einem Array-Typ die Adresse des Elements 0. Ein Ausdruck aus Array-Namen und Index ist äquivalent zu einem Ausdruck aus Zeiger und Abstand (engl. *offset*).

Hinweise zu den Fragen

Hinweise zu Frage 1: Die Werte sind korrekt bis $n = 12$ für Integer mit 32-Bit Repräsentation.

```
fac(0) = 1
fac(1) = 1
fac(2) = 2
fac(3) = 6
fac(4) = 24
fac(5) = 120
fac(6) = 720
fac(7) = 5040
fac(8) = 40320
fac(9) = 362880
fac(10) = 3628800
fac(11) = 39916800
fac(12) = 479001600
```

Die Fakultät von 13 ist 6.227.020.800 und ist größer als der größte 32-bit Integer-Wert 2.147.483.647. Die Verwendung eines Integer-Datentyps mit größerem Wertebereich (z.B. `long`) sorgt dafür, dass ein Overflow erst bei größeren Eingaben passiert.
Problematisch kann außerdem sein, dass für jeden rekursiven Aufruf eine weitere Funktionsinkarnation erstellt wird; dies führt bei großen Eingaben dazu, dass der verfügbare Speicherplatz für Funktionsinkarnationen irgendwann aufgebraucht ist (falls der Compiler die Rekursion nicht optimieren kann). In der iterativen Variante der Funktion tritt dieses Problem nicht auf.

Hinweise zu Frage 2:

```
3 25 90
```

Hinweise zu Frage 3: Für den Null-Zeiger ist garantiert, dass er nirgendwohin zeigt; ein nicht initialisierter Zeiger hingegen kann überallhin zeigen. Man kann außerdem testen, ob ein Zeiger den Wert `NULL` hat; man kann aber nicht prüfen, ob ein Zeiger initialisiert wurde.