C kompakt für Java-Programmierer

Prof. Dr. Carsten Vogt, FH Köln, Institut für Nachrichtentechnik, www.nt.fh-koeln.de/vogt/

Stand: Februar 2012

In der Übung und im Praktikum "Betriebssysteme und verteilte Systeme" sollen Programmieraufgaben an der UNIX-C-Schnittstelle gelöst werden. Hierzu müssen Programme in der Programmiersprache C geschrieben werden. Viele Kontrollstrukturen und grundlegende Datentypen in C entsprechen denen von Java. Es gibt aber auch Unterschiede, von denen hier diejenigen kurz dargestellt werden sollen, die für das Praktikum am wichtigsten sind.
Viele weitere Details findet man im Buch "C für Java-Programmierer" [1]. Ein gutes eigenständiges Lehrbuch zu C ist beispielsweise [2], während der Klassiker [3] (der von den „Erfindern“ von C stammt) eher zum Nachschlagen von Details geeignet ist. Eine Sammlung von C-Beispielprogrammen findet man unter [4].

Aufbau eines einfachen C-Programms

Das folgende Beispiel zeigt ein vollständiges C-Programm, das in dieser Form in eine Datei eingegeben und übersetzt werden kann:

```
#include <stdio.h>
main() {
 /* Subtraktionsprogramm */
 unsigned int a, b;
 int differenz;
 printf("Bitte zwei nichtnegative ganze Zahlen eingeben: ");
 scanf("%u %u", &a, &b);
 differenz = a-b;
 printf("%u - %u = %d\n",a,b,differenz);
}
```

Erläuterungen zum Beispiel:

`#include <stdio.h>`
Einbinden der "Header-Datei" `stdio.h` in das Programm. Die Datei enthält unter anderem die Köpfe (= Schnittstellen = "Prototypen") der Ein- und Ausgabefunktionen `scanf()` bzw. `printf()`, die damit im Programm bekannt gemacht werden.

`main() {`
Kopf des Hauptprogramms.

`/* Subtraktionsprogramm */`
Kommentar.

`unsigned int a, b`
Deklaration zweier Variablen des Typs `unsigned int` (= nichtnegative ganze Zahlen).

`scanf("%u %u", &a, &b)`
Einlesen zweier Werte in die Variablen `a` und `b`. `%u` ist dabei eine "Formatangabe", die eine

Zahl des Typs `unsigned int` bezeichnet. Andere Formatangaben sind beispielsweise `%d` für `int`, `%f` für `float`, `%c` für `char` und `%s` für eine Zeichenkette (String). `&a` und `&b` bezeichnen die "Adressen" der Variablen `a` und `b`, also die Bezeichnungen der Speicherzellen, in die die eingelesenen Werte gebracht werden sollen.

`printf("Bitte zwei nichtnegative ganze Zahlen eingeben: ")`
Ausgabe eines Texts.

`printf("%u - %u = %d\n",a,b,differenz)`
Ausgabe eines Texts mit eingebetteten Werten. Die Positionen der Werte im Text werden dabei durch Formatangaben (siehe oben) definiert, die Werte selbst durch weitere Parameter von `printf()`. Das Steuerzeichen `\n` bewirkt einen Zeilenvorschub.

Achtung: In einem C-Block folgt der Anweisungsteil dem Deklarationsteil; Deklarationen und Anweisungen dürfen nicht gemischt werden!

C-Programm mit Funktionen

C-Funktionen entsprechen den (statischen, also nicht objektbezogenen) Methoden von Java. Sie werden in einer Datei textuell hintereinander definiert. Eine Funktion muss im Text zuerst definiert werden, bevor sie aufgerufen werden kann.

Beispiel:

```
#include <stdio.h>
int differenz(unsigned x, unsigned y) {
 return x-y;
}
main() {
 /* Subtraktionsprogramm */
 unsigned int a, b;
 printf("Bitte zwei nichtnegative ganze Zahlen eingeben: ");
 scanf("%u %u", &a, &b);
 printf("%u - %u = %d",a,b,differenz(a,b));
}
```

Wahrheitswerte

C kennt keinen eigenen Typ für Wahrheitswerte, sondern stellt sie durch Zahlenwerte dar. Der Zahlenwert 0 wird dabei als "false" interpretiert und jeder Zahlenwert ungleich 0 als "true". Aussagenlogische Operationen liefern den Ganzzahlwert 0 für "false" und den Ganzzahlwert 1 für "true". So liefert beispielsweise der Ausdruck "4>3" den Wert 1, "4<3" den Wert 0 und "4>3 && 6<7" den Wert 1.

Felder (Arrays)

Felder (Arrays) speichern in C wie in Java mehrere Werte desselben Typs. Im Unterschied zu Java werden Felder nicht durch einen Konstruktor erzeugt, sondern durch eine Variablendeklaration, bei der die Länge des Felds angegeben wird.

Beispiel:

```
int feld[10];
int i;
for (i=0;i<=9;i++)
 feld[i]=0;
```

Erläuterung zum Beispiel:

`int feld[10]`
Deklaration eines Felds zur Aufnahme von zehn Ganzzahlwerten. Bei der Bearbeitung dieser Deklaration stellt das System Speicherplatz für das Feld bereit, so dass unmittelbar danach damit gearbeitet werden kann.

Achtung:

- Die Laufvariable einer for-Schleife in C kann nicht im Schleifenkopf deklariert werden; die Deklaration muss im Deklarationsteil davor stehen.
- Die Größe eines Felds muss durch eine Konstante angegeben werden. Sie muss also schon zum Zeitpunkt der Übersetzung des Programms feststehen und kann nicht erst während des Programmablaufs berechnet werden.
- In C wird bei einem Feldzugriff, der die Feldlänge überschreitet, keine Fehlermeldung ausgegeben!
- Es gibt in C keine Möglichkeit, aus dem Programm heraus die Länge eines Felds festzustellen.

Strings

Strings, also Zeichenketten, werden in C nicht durch einen eigenen Typ, sondern durch Felder mit dem Komponententyp `char` dargestellt. Jedes Zeichen des Strings steht dabei in einer eigenen Feldkomponenten. Das letzte Zeichen ist stets ein `'\0'`, das das Ende des Strings markiert.

Beispiel:

```
#include <string.h>
...
char name[12];
strcpy(name,"Schmitz");
printf("Name: %s,  Laenge: %d",name,strlen(name));
```

Erläuterungen zum Beispiel:

`char name[12]`
Deklaration eines Felds, das Zeichenketten mit maximal 11 Zeichen (plus abschließendes Zeichen `'\0'`) aufnehmen kann.

`strcpy(name,"Schmitz")`
In das Feld `name` wird die Zeichenkette `"Schmitz"` eingetragen. Die Funktion `str-`

`cpy()` ("String Copy") ist eine Funktion der C-Standardbibliothek. Sie kann i.a. nur benutzt werden, wenn im Programmkopf die Anweisung `#include <string.h>` steht.

`printf("Name: %s,  Länge: %d",name,strlen(name))`
Der String im Feld `name` und seine Länge werden auf den Bildschirm ausgegeben. `%s` ist dabei die Formatangabe für Strings, `%d` für ganze Zahlen. Die Funktion `strlen()` liefert die Länge des Strings, also die Anzahl seiner Zeichen.

Strukturen

C-Strukturen ("Structs") speichern mehrere Werte, die von unterschiedlichen Typen sein können und die über Punktnotation zugreifbar sind. Eine C-Struktur entspricht also, ganz grob gesprochen, einem Java-Objekt, das zwar Attribute ("Werte"), aber keine Methoden besitzt.

Beispiel:

```
struct angestellteninfo {
  char name[12];
  int personalnummer;
  float gehalt;
};
struct angestellteninfo p1, p2;
strcpy(p1.name,"Schmitz");
p1.personalnummer = 4711;
```

Erläuterungen zum Beispiel:

`struct angestellteninfo { ... }`
Deklaration eines Struktur*typs* mit dem Namen `angestellteninfo`, also einer "Bauplanbeschreibung" für Strukturen, die Informationen über eine Person aufnehmen können.

`struct angestellteninfo p1, p2`
Deklaration zweier Variablen `p1` und `p2` des Typs `angestellteninfo`. Jede der Variablen besteht aus drei Komponenten, wie sie in der Typdefinition festgelegt wurden.

`strcpy(p1.name,"Schmitz"); p1.personalnummer = 4711`
Zuweisung zweier Werte an zwei Komponenten der Variablen `p1`.

Unions

Unions ähneln Structs, da auch für sie mehrere Komponenten definiert sind. Anders als bei Structs sind die Komponenten einer Union im Speicher nicht hintereinander angeordnet, sondern sie überlagern sich. Daher kann jeweils nur *eine* Komponente einer Union einen definierten Wert haben.

Beispiel:

```
union angestellteninfo {
  char name[12];
  int personalnummer; };
```

```
union angestellteninfo p1;
strcpy(p1.name,"Schmitz");
p1.personalnummer = 4711;
```

Erläuterungen zum Beispiel:

`union angestellteninfo { ... }`
Deklaration eines Uniontyps mit dem Namen `angestellteninfo`. Eine Union dieses Typs kann *entweder* den Namen eines Angestellten *oder* seine Personalnummer aufnehmen, aber nicht beides gleichzeitig.

`strcpy(p1.name,"Schmitz")`
Der Namenskomponenten der Variablen `p1` wird ein String zugewiesen.

`p1.personalnummer = 4711`
Der Nummernkomponente der Variablen `p1` wird eine Zahl zugewiesen. Damit wird der zuvor zugewiesene String (zumindest teilweise) überschrieben.

Zeiger: Grundoperationen

Variablen in C haben "Adressen": Die Adresse einer Variablen ist, vereinfacht gesagt, die Nummer der Hauptspeicherzelle, in der ihr Wert gespeichert ist (oder, wenn die Variable mehrere Zellen belegt, die Nummer ihrer ersten Zelle). Die Adresse einer Variablen `i` kann in einer anderen Variablen `pt` gespeichert werden. Damit kann auf die Variable nicht nur über ihren Namen `i`, sondern auch über den Zeiger `pt` zugegriffen werden (siehe Skizze unten).

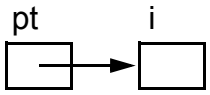
Beispiel:

```
short i;
short *pt;
pt = &i;
*pt = 1;
*pt = *pt +1;
```

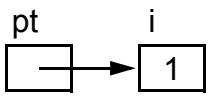
Erläuterungen zum Beispiel:

`short i; short *pt`
Deklaration einer Variablen `i`, die Werte des Typs `short` aufnehmen kann, und einer (Zeiger-)Variablen `pt`, die Adressen von Variablen des Typs `short` aufnehmen kann.

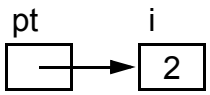
`pt = &i`
Der "Adressoperator" `&` ermittelt die Adresse der Variablen `i`. Diese Adresse wird der Variablen `pt` als Wert zugewiesen, so dass `pt` anschließend auf `i` "zeigt".



`*pt = 1`
Der "Dereferenzierungsoperator" `*` ermittelt die Variable, auf die die Zeigervariable `pt` verweist. Dieser Variablen (hier: `i`) wird dann der Wert 1 zugewiesen.

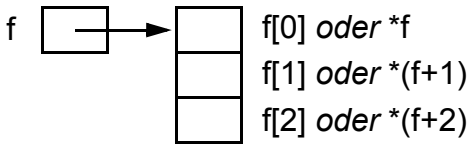


`*pt = *pt + 1`



Der Wert der Variablen, auf die die Zeigervariable `pt` verweist (immer noch `i`), wird ermittelt, und es wird eine 1 addiert (rechte Seite der Zuweisung). Dann wird wiederum die Variable ermittelt, auf die `pt` verweist (linke Seite der Zuweisung, wiederum `i`), und ihr wird der berechnete Wert zugewiesen.

Zeiger ermöglichen eine "Adressarithmetik": Man kann mit Adresswerten "rechnen", also beispielsweise von einer Variablen zu einer anderen Variablen übergehen, die im Speicher unmittelbar davor oder dahinter liegt. Verweist beispielsweise die Zeigervariable `pt` auf eine Variable `i`, so verweist der Adresswert `pt+2` auf die Variable, die im Speicher um zwei Positionen hinter `i` liegt.



Die Adressarithmetik wird in C insbesondere zur Realisierung von Feldern genutzt: Ein Feldname `f` wird in einem C-Programm intern behandelt wie ein (konstanter) Zeiger auf die erste Hauptspeicherzelle des Felds. Eine Zuweisung an die i-te Komponente von `f` lässt sich dann wahlweise schreiben als `f[i]=0` oder als `*(f+i)=0`. Der Programmierer wird natürlich meist die bequeme Indexschreibweise wählen; intern wird sie allerdings immer auf die Adressarithmetik zurückgeführt.

Bei Zeigern auf Strukturen gibt es zwei alternative Schreibweisen: Ist beispielsweise `pt` ein Zeiger auf eine Struktur des Typs `angestelltenifo` (siehe oben), so kann man auf deren Komponente `gehalt` entweder mit `(*pt).gehalt` oder mit `pt->gehalt` zugreifen.

Zeiger zur dynamischen Speicherverwaltung

Den größten Nutzen bringen Zeiger bei der Arbeit mit Speicherbereichen, die der Programmierer selbst verwaltet: C definiert eine Funktion `malloc()` (= "Memory Allocation" = Anlegen von Speicher), mit der das Programm vom Betriebssystem einen zusammenhängenden Speicherbereich einer bestimmten Größe anfordern kann. Diese Anforderung geschieht während der Laufzeit des Programms, also "dynamisch". Der Vorteil dabei ist, dass die Größe des Speichers bei Bedarf erst während des Programmablaufs berechnet werden kann, während beispielsweise die Größe von Feldern schon zur Übersetzungszeit festliegen muss.

Beispiel 1: Dynamisch erzeugter Speicherbereich für eine Folge von `int`-Werten

```
#include <stdio.h>
#include <alloc.h> (auf manchen Systemen stattdessen malloc.h oder stdlib.h)
main() {
 unsigned int bereichslaenge;
 int *bereichsanfang;
 int i;
 printf("Bitte gewuenschte Laenge des Bereichs eingeben: ");
 scanf("%u", &bereichslaenge);
 bereichsanfang = (int *) malloc(bereichslaenge*sizeof(int));

 for (i=0;i<bereichslaenge;i++) {
```

```
  printf("Wert %d: ",i);
  scanf("%d",bereichsanfang+i);
 }
 printf("Inhalt des Speicherbereichs:");
 for (i=0;i<bereichslaenge;i++)
  printf(" %d ",*(bereichsanfang+i));
 free(bereichsanfang);
}
```

Erläuterungen zum Beispiel:

`#include <alloc.h>`

Einbinden der "Header-Datei" `alloc.h`, in der die Funktion `malloc()` deklariert ist. (In manchen Systemen muss stattdessen `malloc.h` oder `stdlib.h` eingebunden werden.)

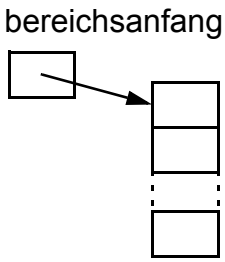
`int *bereichsanfang`

Deklaration einer Zeigervariablen, die anschließend auf den Anfang des dynamisch belegten Speicherbereichs verweisen soll. Dieser Speicherbereich soll dann zur Speicherung einer Folge von `int`-Werten benutzt werden, so dass `bereichsanfang` mit dem Typ "Zeiger auf `int`" deklariert werden muss.

`scanf("%u", &bereichslaenge)`

Einlesen der vom Benutzer gewünschten Größe des Speicherbereichs von der Tastatur. Die Größe steht also zum Zeitpunkt der Übersetzung des Programms noch nicht fest, sondern ergibt sich erst zur Laufzeit des Programms.

```
bereichsanfang =
      (int *) malloc(bereichslaenge*sizeof(int));
```



Aufruf der `malloc()`-Funktion zur Belegung eines bisher freien Speicherbereichs durch das Betriebssystem. Die Funktion erhält als Parameter eine nichtnegative ganze Zahl, die die Größe des gewünschten Speichers in Byte angibt. Der Speicher im Beispielprogramm soll mehrere ganze Zahlen aufnehmen, deren Anzahl durch `bereichslaenge` angegeben ist. Die Anzahl der benötigten Bytes ergibt sich also als der Wert von `bereichslaenge` multipliziert mit der Anzahl von Bytes, die für einen `int`-Wert gebraucht werden (`sizeof(int)`).

`malloc()` liefert die Anfangsadresse des neu belegten Speicherbereichs zurück. Diese Adresse wird durch die Typumwandlung `(int *)` in die Adresse eines Bereichs mit `int`-Zahlen umgewandelt und der Zeigervariablen `bereichsanfang` zugewiesen.

`scanf("%d",bereichsanfang+i)`

Einlesen einer ganzen Zahl, die an die i-te Stelle des neuen Speicherbereichs gebracht wird. Die Adresse dieser Stelle ergibt sich durch die Adressrechnung `bereichsanfang+i`.

`printf(" %d ",*(bereichsanfang+i))`
Ausgabe der ganzen Zahl, die an der i-ten Stelle des neuen Speicherbereichs steht. Die Adressrechnung `bereichsanfang+i` liefert die Adresse dieser Stelle; der `*`-Operator davor liefert den `int`-Wert, der an dieser Stelle steht.

`free(bereichsanfang)`
Der belegte Speicherbereich wird wieder freigegeben. Dieser Aufruf darf auch fehlen, dann wird der Speicherplatz am Ende der Programmausführung freigegeben.

Beispiel 2: Dynamisch erzeugter Speicherbereich für eine Folge von Structs

```
#include <stdio.h>
#include <alloc.h> (auf manchen Systemen stattdessen malloc.h oder stdlib.h)
main() {
 struct angestellteninfo {
   char name[12];
   int personalnummer;
   float gehalt; };
 struct angestellteninfo *bereichsanfang;
 bereichsanfang = (struct angestellteninfo *)
                  malloc(10*sizeof(struct angestellteninfo));
 bereichsanfang[2].personalnummer = 4711;
 printf("Personalnummer des 3. Angestellten: %d",
                       bereichsanfang[2].personalnummer);
}
```

Erläuterungen zum Beispiel:

`struct angestellteninfo *bereichsanfang`
Deklaration einer Zeigervariablen, die auf einen Speicherbereich verweisen kann, der Strukturen des Typs `angestellteninfo` enthält.

```
bereichsanfang
      = (struct angestellteninfo *) malloc(10*sizeof(...))
```
Belegung eines Speicherbereichs, der 10 Strukturen des Typs `angestellteninfo` aufnehmen kann, und an `bereichsanfang` Zuweisung der Adresse dieses Bereichs.

`bereichsanfang[2].personalnummer = 4711`
Zugriff auf die dritte Struktur im neuen Speicherbereich ("dritte", da die Indizierung bei 0 beginnt) und Zuweisung eines Werts an ihre Komponente `personalnummer`. Zum Zugriff auf die Struktur wird hier die bequeme Indexschreibweise von Feldern benutzt. Dies ist möglich, weil Indizierung und Adressrechnung intern auf dieselbe Weise durchgeführt werden.

Referenzübergabe von Parametern

Die Parameterübergabe an C-Funktionen erfolgt im Normalfall wie bei Java-Methoden durch einen "Wertaufruf": Es wird Speicherplatz für die formalen Parameter erzeugt, die Werte der

aktuellen Parameter werden in diese Speicherplätze kopiert, und die Funktion arbeitet auf diesen Speicherplätzen. Nach Rückkehr aus der Funktion werden die Speicherplätze wieder gelöscht. Eine C-Funktion arbeitet bei einem Wertaufruf also auf ihren eigenen Speicherplätzen; sie greift nicht auf die Speicherplätze der aufrufenden Funktion (beispielsweise des Hauptprogramms) zu.
In manchen Fällen ist es jedoch erforderlich, dass eine Funktion auf den Variablen der aufrufenden Funktion arbeitet. In diesem Fall verwendet man einen "Referenzaufruf", bei dem *Zeiger* auf die Variablen der aufrufenden Funktion übergeben werden. Die gerufene Funktion kann dann über diese Zeiger auf die Variablen der rufenden Funktion zugreifen.

Beispiel: Funktion zum Vertauschen der Inhalte zweier Variablen der *aufrufenden* Funktion

```
void tausch(int *a, int *b) {
 int hilf;
 hilf=*b; *b=*a; *a=hilf;
}

main() {
 int x=1,y=2;
 tausch(&x,&y);
}
```

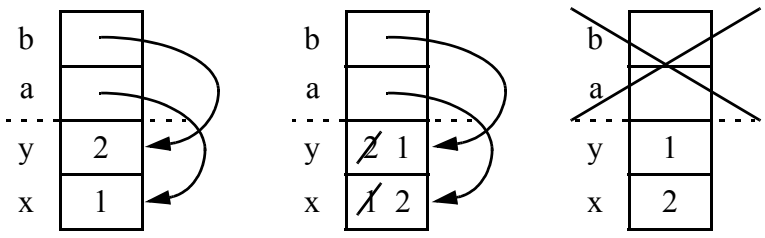
Erläuterungen zum Beispiel:

`void tausch(int *a, int *b)`
Die Funktion `tausch()` erhält als Parameter *Zeiger* auf zwei `int`-Variablen.

`hilf=*b; *b=*a; *a=hilf`
Die Funktion tauscht die beiden Werte, die in den Variablen stehen, auf die die Zeiger `a` und `b` verweisen. `a` und `b` enthalten dabei die Adressen der Variablen der aufrufenden Funktion (siehe nächster Punkt).

`tausch(&x,&y)`
Das Hauptprogramm ruft die Funktion `tausch()` auf und übergibt dabei an die formalen Parameter `a` und `b` die Adressen ihrer Variablen `x` und `y`. Die Funktion vertauscht also die Inhalte dieser beiden Hauptprogrammvariablen (siehe Grafik).

Aufrufstacks: a.) Beim Aufruf: b.) Bei der Ausführung: c.) Nach der Rückkehr:



(Zum Begriff des Aufrufstacks und zur Implementation von Funktionsaufrufen erinnern Sie sich bitte an Praktische Informatik 1.)

Felder (insbesondere Strings) werden in C immer per Referenzaufruf übergeben. Ein entsprechender Funktionsprototyp hat die Form `fct(`*`komponententyp`* `*feld, int laenge)` oder `fct(`*`komponententyp`* `feld[], int laenge)`, wobei *`komponententyp`* je nach Typ des Felds `int`, `char`, `float` usw. sein kann. Da man in C nicht feststellen kann, wie lang ein Array ist (er ist lediglich durch einen Zeiger auf seine erste Speicherzelle charakterisiert), sollte die aufrufende Funktion die Arraylänge in einem zweiten Parameter übergeben, damit die gerufene Funktion bei Bedarf darauf zugreifen kann.

Weiterführende Literatur und Web-Seite

[1] C. Vogt, C für Java-Programmierer, Hanser-Verlag München, 2007
Web-Seite: http://www.fh-koeln.de/cfuerjava/

[2] M. Dausmann, U. Bröckl, J. Goll.: C als erste Programmiersprache – Vom Einsteiger zum Profi, Teubner-Verlag Wiesbaden, 2008

[3] B. Kernighan, D. Ritchie, Programmieren in C, Hanser-Verlag München, 1990

[4] C. Vogt, C-Beispielprogramme, http://www.nt.fh-koeln.de/vogt/dv/c/cbsp.html