Strings und Strukturen in C

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

1 Strings

Strings sind in C als Arrays von Zeichen realisiert, wobei das Ende eines Strings durch ein Nullzeichen `\0` markiert wird.
Beispiel:

```
char name[30]; /* max. 29 Zeichen plus Nullzeichen */
name[0] = 'H';
name[1] = 'u';
name[2] = 'g';
name[3] = 'o';
name[4] = '\0';
printf("%s",name);
```

Bei der Ausgabe werden nur die Zeichen bis zum Nullzeichen berücksichtigt; die übrigen Array-Elemente sind nicht relevant.
Ein häufiger Fehler im Umgang mit Strings ist, dass das entsprechende Array vollständig mit Zeichen gefüllt wird, sodass kein Platz mehr für das Nullzeichen `\0` bleibt. Auch wird beim zeichenweise Kopieren von Zeichenketten `\0` oft vergessen zu kopieren, da das Zeichen zum Testen bei der Abbruchbedingung genutzt wird.
Im Folgenden stellen wir einen Auszug der wichtigsten Funktionen aus der String-Bibliothek `string.h` vor. Man unterscheidet dabei Stringfunktionen (beginnen mit `str`) und Funktionen, die ein Array byteweise bearbeiten (beginnen mit `mem`).

Modifizieren von Strings Stringliterale werden vom C-Compiler aus dem Programm beim Compilieren extrahiert und in den Datenbereich des Programmspeichers geschrieben. Das entsprechende `char`-Array erhält automatisch die richtige Größe (inkl. `\0`). Allerdings ist der Datenbereich typischerweise ein schreibgeschützter Speicherbereich (*read-only*). String**literale** sind in C daher nicht modifizierbar.

```
1 #include <string.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6   char *s1 = "Das ist ein String Literal";
```

```
7     char s2[] = {'D','a','s',' ','a','u','c','h'};
8     char s3[] = {'S','t','u','d','i','\0','e','r','e','r'};
9
10    printf("%s\n",s1);  // "Das ist ein String Literal"
11    printf("%s\n",s2);  // "Das auch"
12    printf("%s\n",s3);  // "Studi"
13
14    s1[1] = 'x'; // undefiniert!!
15
16    return 0;
17  }
```

Um einen String zu modifizieren, muss er zunächst an eine geeignete Stelle kopiert werden (Stack oder Heap). Die Funktion `void *memcpy (void *to, const void *from, size_t n)` kopiert `n` Bytes von Anfang von `from` an den Anfang von `to`. Das Verhalten der Funktion ist undefiniert, wenn die beiden Speicherbereiche von `from` und `to` überlappen. Der Rückgabewert der Funktion ist der Pointer `to`.

```
1   #include <string.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   int main(void)
6   {
7     char *old = "C sucks";
8     char *new = malloc(8 * sizeof(char));
9     memcpy (new, old, 8 * sizeof(char));
10
11    new[2] = 'r';
12    new[3] = 'o';
13    printf("\"%s\" wird zu \"%s\"\n",old, new);
14    free(new);
15    return 0;
16  }
```

Frage 1: Was ist die Ausgabe des Programms?

Länge von Strings Die Funktion `size_t strlen(const char *s)` liefert die Länge des adressierten Strings `s` **ohne** das Nullzeichen, d.h. die Anzahl der Character bis zum ersten Nullzeichen.[1]

```
1   #include <stdio.h>
2   #include <string.h>
3
```

[1] Der Typ `size_t` ist ein vorzeichenloser Integer-Typ zur Bezeichnung von Elementgrößen. Er wird u.A. als Rückgabetyp von `sizeof` verwendet. Der Typmodifikator `const` beim Parameter zeigt an, dass `s` als ein String behandelt wird, dessen Elemente nicht modifiziert werden dürfen.

```c
int main(void)
{
  char s[] = "I <3 C";
  int length = strlen(s);
  printf("Der String \"%s\" hat %d Zeichen\n", s, length);

  // Achtung: strlen liefert die Position des ersten \0
  //  und nicht die Groesse des Arrays
  char s3[] = {'S','t','u','d','i','\0','e','r','e','r'};
  printf("Der String \"%s\" hat %lu Zeichen\n", s3, strlen(s3));
  printf("Das Array s3 hat %lu Bytes\n", sizeof(s3));

  return 0;
}
```

Frage 2: Was ist die Ausgabe des Programms?

Konkatenation von Strings Das Aneinanderhängen (Konkatenieren) von Strings kann in C mit Hilfe von `char* strcat(char* s1, const char* s2)` implementiert werden. Dabei wird das Nullzeichen `\0` von des ersten Strings `s1` überschrieben. Voraussetzung für das korrekte Konkatenieren ist, dass der für `s1` reservierte Speicherbereich ausreichend groß zur Aufnahme von `s2` ist. Andernfalls ergibt sich undefiniertes Verhalten. Die Alternative `char* strncat(char* s1, const char* s2, size_t n)` konkateniert nur die ersten `n` Elemente von `s2` an `s1`. Dabei ist `n` so zu wählen, dass der für `s1` reservierte Speicherbereich nicht überschritten wird. Auch hier wird `\0` an das Ende der Resultats angehängt. Für überlappende Speicherbereiche ist das Verhalten nicht definiert.

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

int main(void)
{
  const char x[] = "Urin";
  const char y[] = "stinkt";

  char* z = malloc(strlen(x) + strlen(y) + 1);
  memcpy (z, x, strlen(x));
  strncat(z, y, strlen(y));
  printf("%s\n",z);
  free(z);
  return 0;
}
```

Frage 3: Was ist die Ausgabe des Programms?

Vergleichen von Strings Mit der Funktion `int strcmp(char* s1, char* s2)` kann man zwei Strings vergleichen. Dabei werden die Strings Zeichen für Zeichen durchlaufen und die ASCII-Codes der Zeichen verglichen. Sind die beiden Strings identisch, liefert die Funktion den Wert 0. Ist der Rückgabewert kleiner als 0 (größer als 0), dann hat der erste ungleiche Buchstabe in `s1` einen größeren (kleineren) ASCII-Code als der entsprechende Buchstabe in `s2`.

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

int main(void)
{
  char s1[] = "C rocks";
  char s2[] = "C sucks";

  int not_equal = strcmp(s1,s2);
  if(not_equal) {
    printf("\"%s\" und \"%s\" sind unterschiedlich",s1,s2);
  } else {
    printf("\"%s\" und \"%s\" sind gleich",s1,s2);
  }

  return 0;
}
```

Falls in den jeweiligen Char-Arrays kein Nullzeichen `\0` ist, wird bei `strcmp` unkontrolliert über die Arraygrenze hinweggelesen!
Die Funktion `int strncmp(const char *x, const char *y, size_t n)` arbeitet wie `strcmp` mit dem Unterschied, dass die ersten `n` Zeichen von `x` und `y` miteinander verglichen werden. Da bei `strncmp` die Anzahl der zu vergleichenden Character mit angegeben ist, gilt sie als sicherere Variante.

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

int main(void)
{
  const char x[] = "abce";
  const char y[] = "abcd";

  for(int i = strlen(x); i > 0; --i) {
    if(strncmp( x, y, i) != 0){
      printf("Die ersten %d Zeichen der beiden Strings sind nicht
     gleich\n", i);
    } else {
      printf("Die ersten %d Zeichen der beiden Strings sind
    gleich\n", i);
      break;
    }
```

```
16    }
17    return 0;
18  }
```

Ausgabe:

```
Die ersten 4 Zeichen der beiden Strings sind nicht gleich
Die ersten 3 Zeichen der beiden Strings sind gleich
```

Umwandlung von Strings in Zahlenwerte Hilfsfunktionen zur Umwandlung von Strings in Zahlenwerte finden Sie in `stdlib.h`. So wandelt `double atof(const char *s)` einen String `s` in einen Double- und `int atof(const char *s)` einen String in einen Integerwert um.

2 Strukturen

Um Daten unterschiedlicher Typen gemeinsam zu verwalten, gibt es in C ***Strukturen*** (Verbund, engl. *structures, structs*). Im folgenden Beispiel wird eine Struktur `datum` definiert und mittels `typedef` als Typ `datum_t` verfügbar gemacht [2]:

```
struct datum
 {
   int tag;
   char monat[10];
   int jahr;
 } ;
typedef struct datum datum_t;
```

Variablen für einzelne Instanzen der Struktur lassen sich dann folgendermaßen definieren:

```
// statische Initialisierung
datum_t geburtstag = {5, "Juli", 1987};
```

Eine dynamische Initialisierung während des Programmlaufs muss komponentenweise erfolgen. Werden Strings als Komponenten verwendet, müssen diese mittels `strcpy()` etc. initialisiert werden.

```
// Dynamisches Allozieren auf dem Heap
datum_t* g = malloc(sizeof(datum_t));
g->tag = 5;
// Alloziert den String mit den anderen Struktur-Komponenten auf
    dem Heap
strcpy(g->monat, "Juli");
g->jahr = 1987;

// spaeteres Freigeben des Speichers
free(g);
```

[2] Es gibt weitere syntaktische Varianten der Struktur-Definition, auf die wir hier nicht näher eingehen werden.

Die einzelnen Komponenten einer Struktur können über den .-Operator angesprochen werden.

```
geburtstag.jahr = 1986;
```

Auch Strukturen können über Zeiger adressiert werden. Da dies in der Praxis häufig benötigt wird, gibt es für den Komponentenzugriff eine syntaktische Variante mittels ->.

```
datum_t *d = &geburtstag;
//d enthaelt nun die Speicheradresse von geburtstag
(*d).tag = 1950;
//Der addressierte Wert von d ist die Struktur, daher hier
    Zugriff ueber .
d->tag = 1950;
// syntaktische Alternative
```

Strukturen können auch innerhalb von Funktionen definiert werden:

```
#include <stdio.h>

int main (void) {
  // Struktur fuer zweidimensionale Punkte
  struct point {
    float x;
    float y;
  };
  typedef struct point point_t;

  // Punkt p, auf dem Stack alloziert
  point_t p;
  // Variante 1:
  (&p)->x = 3;
  // Variante 2:
  p.y = 4;
  printf ("(%f, %f)\n", p.x, (&p)->y);
  // Punkte q, auch auf dem Stack alloziert und statisch
    initialisiert
  point_t q = {3,4};
  // Vergleich der beiden Punkte
  int equal = (p.x == q.x) && (p.y == q.y);
  // nicht moeglich: p == q !!!
  // Deallokation nicht notwendig
  return 0;
}
```

Ein Vergleich von Strukturen muss komponentenweise erfolgen. Wie wir im folgenden Beispiel sehen werden, können Strukturen als Parameter und Rückgabewerte von Funktionen verwendet werden.

2.1 Beispiel: Einfach verkettete Listen

Auch für C Programme kann man abstrakte Datentypen wie Listen und Bäume implementieren. Hierbei sind Strukturen sehr hilfreich, um beispielsweise Knoten zu repräsentieren. Wir zeigen im Folgenden eine Implementierung einer einfach verketteten Liste in Anlehnung an die Listenimplementierung, die wir in Java gesehen haben.
Wir definieren zunächst eine Struktur für Knoten, die wir unter dem Typ `node_t` einführen. Jeder Knoten hat eine `value`-Komponente sowie einen Zeiger auf den nächsten Knoten `next`. Eine Liste hat als Komponente den Zeiger auf den ersten Listenknoten `first`.

```
1  typedef struct node node_t;
2  struct node
3  {
4    int value;
5    node_t *next;
6  };
7
8  typedef struct linked_list linked_list_t;
9  struct linked_list
10 {
11   node_t *first;
12 };
```

Die Funktion `new_list` erstellt eine leere Liste, die auf dem Heap alloziert wird.

```
1  linked_list_t *new_list(void)
2  {
3    // Alloziere Speicher auf dem Heap fuer die Listen-Struktur
4    linked_list_t *ll = malloc(sizeof(linked_list_t));
5    if (NULL != ll)
6    {
7      // Initialisere die Komponenten der Liste
8      ll->first = NULL;
9      return ll;
10   }
11   else
12   {
13     printf("Couldn't allocate linked_list\n");
14     abort(); // Programmabbruch
15   }
16 }
```

Alternativ kann der Speicher direkt mittels `calloc` initialisiert werden. Der Vergleich `NULL == ll` lässt sich außerdem verkürzen:

```
1  linked_list_t *new_list(void)
2  {
3    // Alloziere und initalisiere Speicher auf dem Heap fuer die
       Listen-Struktur
4    linked_list_t *ll = calloc(1,sizeof(linked_list_t));
```

```
5     if (ll)
6     {
7       return ll;
8     }
9     else
10    {
11      printf("Couldn't allocate linked_list");
12      abort(); // Programmabbruch
13    }
14  }
```

Das Einfügen von Elementen setzen wir folgendermaßen um:

```
1   /* Fuegt ein Element hinten in die Liste ein */
2   void add_elem(linked_list_t *ll, int i)
3   {
4     // Neuer Knoten fuer das einzufuegende Element
5     node_t *new_node;
6     new_node = malloc(sizeof(node_t));
7     if (NULL == new_node)
8     {
9       printf("Couldn't allocate new node\n");
10      exit(-1);
11    }
12    new_node->value = i;
13    new_node->next = NULL;
14
15    // Iteriere bis zum Ende der Liste
16    if (NULL == (ll->first))
17    {
18      // Falls Liste bisher leer, fuege neuen Knoten als ersten
        Knoten ein
19      ll->first = new_node;
20    }
21    else
22    {
23      node_t *n = ll->first;
24      while (NULL != (n->next))
25      {
26        n = n->next;
27      }
28      // Fuege Knoten als Nachfolger des letzten Elements ein
29      n->next = new_node;
30    }
31    return;
32  }
```

In dieser Variante wird zunächst geprüft, ob die Liste noch leer ist und das Element als erstes Element eingefügt werden muss.
Unter der Verwendung von Zeigern kann man auf diesen Spezialfall verzichten. Statt-

dessen iterieren wir über den jeweiligen Speicherplatz, an dem der Zeiger auf das nächste Element verwaltet wird.

```
/* Fuegt ein Element hinten in die Liste ein */
void add_elem(linked_list_t *ll, int i)
{
  // Alloziere neue Knoten-Stuktur
  node_t *new_node = malloc(sizeof(node_t));
  if (!new_node)
  {
    printf("Couldn't allocate new node");
    exit(-1);
  }
  // Initalisiere den neuen Knoten
  new_node->value = i;
  new_node->next = NULL;

  // Iteriere ueber die Liste bis zum Ende
  // current ist ein Zeiger auf einen Zeiger, der auf einen
    Knoten zeigt
  node_t **current = &ll->first;
  while (*current)
  {
    current = &(*current)->next;
  }
  *current = new_node;
}
```

Das Entfernen von Knoten an einer Position lässt sich analog dazu mit Zeigern so umsetzen:

```
/* Entfernt das Element an Position pos, falls vorhanden */
void remove_elem(linked_list_t *ll, int pos)
{
  // Zeiger auf den Zeiger, der auf den naechsten Knoten zeigt
  node_t **prev = &(ll->first);
  // Zeiger auf den naechsten Knoten
  node_t *n = ll->first;
  while (pos > 0 && NULL != (n->next))
  {
    pos--;
    prev = &(n->next);
    n = n->next;
  }
  if (!pos)
  {
    *prev = n->next;
    free(n);
  }
  return;
```

```
}
```

Außerdem implementieren wir noch eine Funktion, die über die Liste iteriert und die Elemente nacheinander auf der Konsole ausgibt.

```
void print_list(linked_list_t *ll)
{
  node_t *n = ll->first;
  while (n)
  {
    // Ausgabe des Knotenwerts
    printf("%d  ", n->value);
    // Weiter mit dem Nachfolger
    n = n->next;
  }
  printf("\n");
}
```

Frage 4: Schreiben Sie eine Variante von `print_list`, die mit Hilfe einer `for`-Schleife über die Elemente iteriert!

Wir benötigen schließlich noch eine Funktion, um die Liste wieder zu deallozieren. Dazu iterieren wir zunächst über die einzelnen Knoten der List und geben diese frei, bevor wir die Listen-Struktur selbst freigeben.

```
void free_list(linked_list_t *ll)
{
  // Entferne zunaechst alle Knoten der Liste
  node_t *n = ll->first;
  while (NULL != n)
  {
    // Merken, was der naechste Knoten ist, da nach dem free die
    Information nicht mehr verfuegbar ist
    node_t *next = n->next;
    // Deallozieren des Knotens
    free(n);
    n = next;
  }
  // Entferne die Listen-Stuktur
  free(ll);
}
```

Wir können die Liste nun folgendermaßen verwenden:

```
  // Entferne die Listen-Stuktur
  free(ll);
}

int main(void)
```

```
6  {
7    // Erstelle leere Liste
8    linked_list_t *ll = new_list();
9    // Fuege Elemente in die Liste ein
10   for (int i = 1; i < 10; i++)
11   {
12     add_elem(ll, i);
13   }
14   // Gebe die Liste aus
15   print_list(ll);
16
17   // Entferne alle geraden Elemente aus der Liste
18   for (int i = 9; i >= 0; i -= 2)
19   {
20     remove_elem(ll, i);
21   }
22   // Gebe die Liste erneut aus
23   print_list(ll);
24
25   // Entferne die Liste vom Heap
26   free_list(ll);
```

Frage 5: In welchen Aspekten unterscheiden sich Strukturen in C und Objekte in Java?

3 Der Heartbleed Bug

Das Verwenden von Strings und anderen Speicherbereichen mit flexibler Länge erfordert in C besonders sorgfältiges Vorgehen. Kleine Programmierfehler können leicht zu schwerwiegenden Problemen führen, die das Programm abstürzen lassen oder zu Sicherheitslücken im Programm führen, was in der Regel noch schwerwiegender ist. Sicherheitslücken lassen sich durch einfaches Testen nur schwer finden und so kann es passieren, dass eine Sicherheitslücke lange Zeit unentdeckt bleibt und ausgenutzt wird.
Ein prominentes Beispiel hierfür ist der “Heartbleed” Bug , der auf einen Fehler in der Implementierung der OpenSSL-Bibliothek zurückgeht. Diese Bibliothek wird von vielen Internet-Anwendungen verwendet, um eine verschlüsselte Verbindung sicherzustellen. Der fehlerhafte Code wurde am 31. Dezember 2011 in die Bibliothek eingebunden, am 14. März 2012 veröffentlicht, aber erst 2 Jahre später, im April 2014, entdeckt und behoben. Es wird geschätzt, dass ca. 17% aller Webserver mit verschlüsselter Verbindung von dem Problem betroffen waren. Eine anschaulichere Erklärung des Bugs als Comic findet sich unter `http://xkcd.com/1354/`.
Der Fehler war Teil einer Komponente, mit der sogenannte “Heartbeat”-Nachrichten zwischen Servern verschickt werden können. Diese Nachrichten werden gesendet, um eine Verbindung auch dann offen zu halten, wenn gerade keine anderen Daten verschickt

werden. Wenn die Gegenseite nicht mehr auf Heartbeat-Nachrichten antwortet, kann die Verbindung geschlossen werden, ansonsten wird sie offen gelassen.
Um auszuschließen, dass Angreifer durch wiederholende Nachrichten etwas über die verschlüsselte Kommunikation erfahren können, wurden die Heartbeat-Nachrichten von OpenSSL so entworfen, dass der Sender einen beliebigen Text an den Empfänger senden kann und dieser dann zur Bestätigung mit dem gleichen Text antwortet.
Im folgenden betrachten wir eine **stark vereinfachte** Form des Quelltexts, um das Problem bei der Implementierung dieser Funktion zu verdeutlichen.
Eine Heartbeat-Nachricht enthält den gesendeten Text als `char`-Array und die Länge des Texts als Zahl.

```
typedef struct message {
  int length;
  char *data;
} message_t;
```

Um den gleichen Text als Antwort zu schicken, wurde im Code das Feld für die Länge ausgelesen und dann so viele Bytes von der ankommenden Nachricht in die Antwort kopiert, wie in der Länge angegeben wurde.

```
message_t *incoming = ...;
message_t *outgoing = ...;

memcpy(outgoing->data, incoming->data, incoming->length);
```

Ein Angreifer konnte nun eine Nachricht, wie die folgende schicken, um den Fehler auszunutzen:

```
message_t *bad_message = ...;
bad_message->length = 65535;
bad_message->data = "Hallo";
```

Diese Nachricht hat zur Folge, dass beim Kopieren des Text aus der eingehenden Nachricht 65535 Bytes kopiert werden, aber nur wenige Daten-Bytes wirklich vorhanden sind. Das Programm liest dann einfach den Speicher, der zufällig hinter der eingehenden Nachricht steht und schreibt diesen dann in die Antwort. Mit diesem Trick konnten Angreifer durch wiederholte Anfragen einen großen Teil des Speichers auf dem betroffenen Server auslesen, wo sich unter anderem auch Sicherheitsschlüssel befinden, mit denen eine noch umfassendere Kontrolle des Servers möglich ist.
Der Fehler wurde noch dadurch verstärkt, dass die Bibliothek OpenSSL ihre eigene Implementierung von `malloc` verwendet hat, welche weniger Überprüfungen enthält als die Implementierung der Standardbibliothek. Die Standardbibliothek hätte bei vielen Angriffsversuchen vermutlich zu einem Segmentation Fault geführt und somit verhindert, dass geheime Daten an den Angreifer geschickt werden.
Zum Beheben des Fehlers wurde letztendlich eine Überprüfung eingebaut, die Nachrichten mit ungültiger Längen-Angabe verwirft. Die Lehre aus diesem Fehler ist, dass man Daten, die von außerhalb des Systems kommen, nie vertrauen sollte und dass man beim Arbeiten mit Speicher in C immer sorgfältig überprüfen muss, dass man im gültigen Bereich bleibt.

Wie die Änderungen am Code zur Fehlerbehebung tatsächlich aussehen, sieht man zum Beispiel in der Versionsverwaltung auf GitHub[3]. Der Artikel "Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug" [4] und der Wikipedia Artikel[5] beschreiben den Bug ausführlicher.
Es wird erwartet, dass das Muster, nach welchem der Heartbleed-Bug operiert, sich in vielen weiteren Programmen findet, neben vielen anderen Bugs, die z.B. durch Überschreiten des gültigen Bereichs die Programmausführung beeinflussen und Schadprogramm auf einem System einschleusen können.

4 Zusammenfassung: Zur Programmierung in C

C ist eine hardwarenahe Sprache, die im Vergleich mit anderen Programmiersprachen wenige Abstraktionsmechanismen besitzt. Wir wollen hier noch einmal die wichtigsten Eigenschaften von C kurz zusammenfassen.

- Kein automatisches Speichermanagement: Der Speicherbereich für dynamisch erzeugte Datenstrukturen muss vom Programmierer angefordert und wieder freigegeben werden. Es liegt dabei in der Verantwortung des Programmierers sicherzustellen, dass freigegebener Speicher nicht versehentlich später noch zugegriffen wird und dass es keine Speicherlecks gibt.

- Keine Unterscheidung von Arrays und Zeigern: Arrays sind lediglich Zeiger auf eine Speicheradresse, die den Beginn des Arrays (das erste Element) enthält. Es ist nicht möglich, anhand dieses Zeigers die Größe eines Arrays zu ermitteln. Die Größe muss daher immer gemeinsam mit dem Array verwaltet werden, um Zugriffe auf Indizes zu validieren und Pufferüberläufe zu verhindern. Diese stellen ein gravierendes Sicherheitsproblem dar und können auch zu Programmabstürzen führen.

- Schwache Typisierung: Das Typsystem von C erlaubt potentiell fehlerhafte Typcasts, die erst zur Laufzeit zu Fehlern führen bzw. unbestimmtes Verhalten provozieren (z.B. Interpretation von beliebigen Integern als Speicheradressen).

- Keine Fehlerbehandlung mittels Exception: Funktionen nutzen häufig designierte Rückgabewerte, um Fehlerfälle zu melden (z.B. liefert `malloc` den Wert `NULL` , wenn kein freier Speicher mehr verfügbar ist). Wenn diese Sonderfälle nicht berücksichtigt werden, passieren bisweilen schwer zu lokalisierende Fehler.

[3] `https://github.com/openssl/openssl/commit/731f431497f463f3a2a97236fe0187b11c44aead?diff=split`
[4] `http://www.theregister.co.uk/2014/04/09/heartbleed_explained/`
[5] `https://en.wikipedia.org/wiki/Heartbleed`

Hinweise zu den Fragen

Hinweise zu Frage 1:
‘‘C sucks’’ wird zu ‘‘C rocks’’

Hinweise zu Frage 2:

```
Der String ‘‘I <3 C’’ hat 6 Zeichen
Der String ‘‘Studi’’ hat 5 Zeichen
Das Array s3 hat 10 Bytes
```

Hinweise zu Frage 3:

```
 Urinstinkt
```

Hinweise zu Frage 4:

```
1   // current ist ein Zeiger auf einen Zeiger, der auf einen
      Knoten zeigt
2   node_t **current = &ll->first;
3   while (*current)
4   {
5     current = &(*current)->next;
6   }
7   *current = new_node;
```

Hinweise zu Frage 5: Sie unterscheiden sich u.a. in den folgenden Aspekten:

- Strukturen sind Werte in C, Objekte sind keine Werte in Java (man kann an eine Variable nur die Referenz auf ein Objekt zuweisen, nicht aber das Objekt selbst).
- In Java wird bei Objekt-Methoden ein impliziter Parameter mit der Referenz auf das aktuelle Objekt (`this`) übergeben; in C muss die Struktur explizit als Parameter übergeben werden.
- Java-Objekte sind immer auf dem Heap alloziert; bei Strukturen kann der Programmierer den Speicherort bestimmen.
- Attribute von Java-Objekten werden immer durch den Konstruktor initialisiert; Strukturen in C muss der Programmierer initialisieren.
- Vererbung und dynamisches Binden gibt es in C nicht (es kann allerdings mit Funktionspointern simuliert werden).
- Information Hiding, Polymorphismus, etc. gibt es in C nicht.