Sichere Strings und Speicher in C

Pablo Yánez Trujillo

Poolmanager Team
Universität Freiburg

8. August, 2006

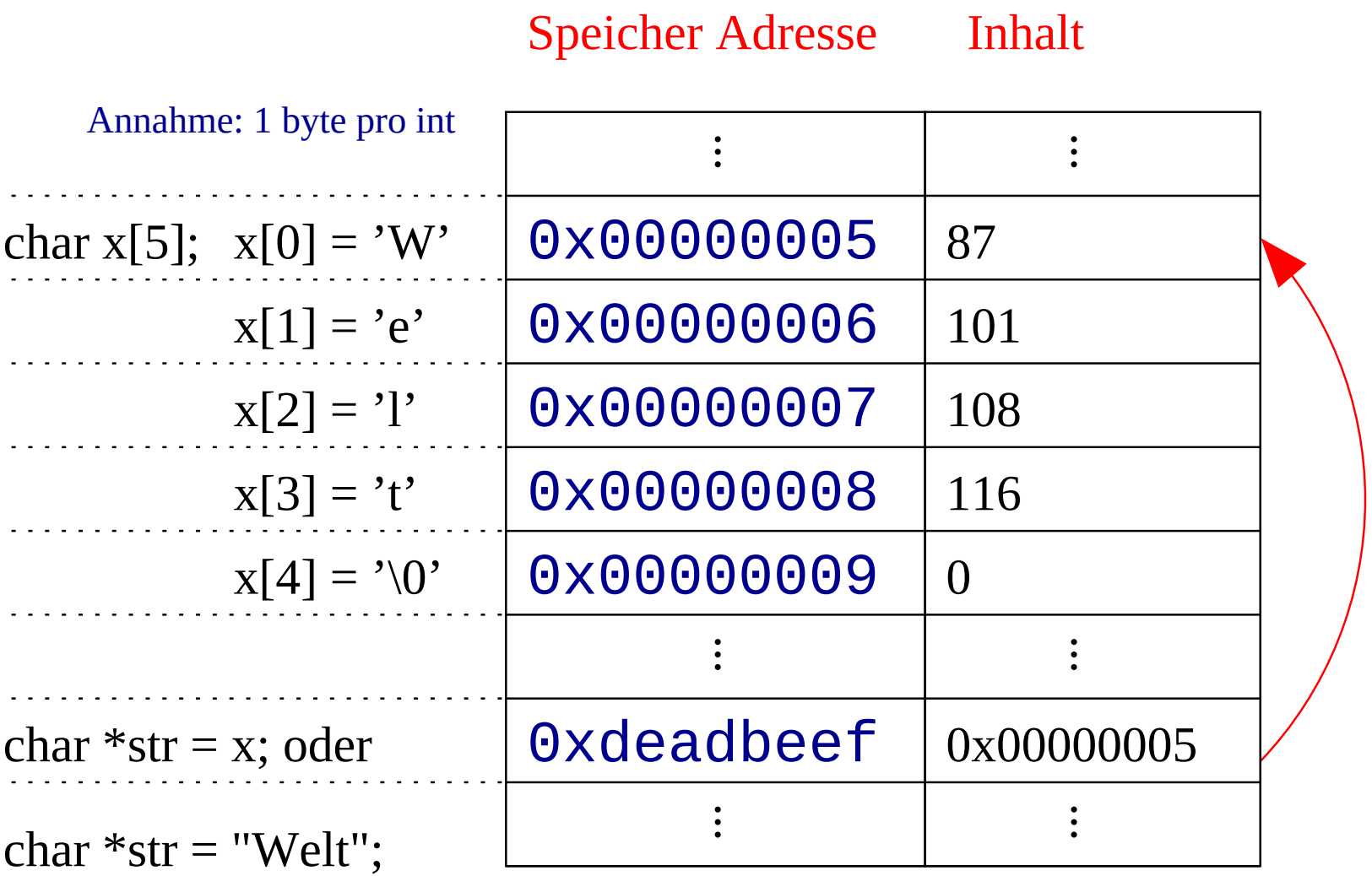
C-Strings

- Die einfachste Form eines Strings zu deklarieren ist:

Ein String

```
char *string = "Ich bin ein String";
```

- Es gibt kein nativer Datentyp für String (wie C++ oder Java)
- Ein C-String ist also ein Zeiger auf ein `char`
- Eigentlich auf ein Array von `char` Elemente


Speicher Adresse
Inhalt
Annahme: 1 byte pro int
char x[5]; x[0] = 'W'
x[1] = 'e'
x[2] = 'l'
x[3] = 't'
x[4] = '\0'
char *str = x; oder
char *str = "Welt";
0x00000005
87
0x00000006
101
0x00000007
108
0x00000008
116
0x00000009
0
0xdeadbeef
0x00000005

- Ein C-String (Zeichenkette) ist eine Folge von Zeichen, die mit dem '\0' Zeichen abgeschlossen wird
- C-String werden auch 0-terminierende Strings genannt
- Das '\0' Zeichen ist notwenidig, denn damit kann man herausfinden, wo die Zeichenkette aufhört

C-Strings Deklaration

C-Strings Deklaration

```
char *str = "Ich bin ein konstantes String";
char str[1024] = "Bin nicht konstant, benutzer aber wenig Speicher";
char str[] = "Die Array Größe wird mir nach angepasst";
char str[1024]; strcpy(str, "Für mich muss string.h eingebunden sein");

/* Folgendes ist nicht möglich */
char str[1024]; str = "Haha";
char *str; strcpy(str, "Das geht immer schief");
```

"Dynamische Strings"

- Sehr oft muss man mit C-Strings arbeiten, deren Länge erst während der Laufzeit bekannt ist
- Eine Möglichkeit wäre, ein `char`-Array mit genügend Platz zu deklarieren
- Ist aber nicht sinnvoll, denn wenn man die Länge nicht abschätzen kann, kann es zu Pufferüberläufen führen und diese Bugs ausnutzen
- Es ist deswegen wichtig, genug Platz zu reservieren, so dass man keinen Überlauf hat. Da helfen `malloc` & co.

"Dynamische Strings"

- Immer daran denken: Die C-String müssen 0-terminierend sein. Wenn man dynamischen Platz reserviert, muss auch Platz für die terminierende 0 vorhanden sein

C-Strings mit malloc

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


char *str_to_dyn_str(const char *src)
{
    size_t len;
    char *dest;
    len = strlen(src); /* undef.  Verhalten, wenn src nicht 0-terminierend ist */
    dest = malloc(len + 1);


    if(!dest)
        return NULL;
    srtcpy(dest, src); /* \0 von strcpy gesetzt */
    return dest; /* wichtig:  der Aufrufer muss free aufrufen */
}
```

Inside of string.h

- Die C Bibliothek bietet eine Reihe von Funktionen, die mit C-Strings arbeiten
- Die Funktionen müssen aber richtig benutzt werden, sonst gibt es Bugs oder undefiniertes Verhalten
- Es ist immer Arbeit des Programmierers dafür zu sorgen, dass die Puffer genug Speicher haben oder die Strings \0-terminierend sind
- Jetzt werden die wichtigsten Funktion von `string.h` vorgestellt

- Wenn man bereits Erfahrung mit C++/Java/usw. hat, wird einiges seltsam finden. Wie hängt man z.b. ein String in ein anderes an?

So bestimmt nicht

```
char *str = "Hallo " + "Welt!";
```

- Oder wie prüft man, dass ein String ein bestimmtest Inhalt hat?

So bestimmt auch nicht

```
char *str1, *str2;
...
if(str1 == "Hallo" || str2 == str1)...
```

strcmp

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

- Wenn der Inhalt von s1 und s2 gleich sind, dann liefert strcmp 0 zurück.
- Wenn der Inhalt von s1 und s2 ungleich sind, dann liefert strcmp etwas ungleich 0 zurück
- strcmp gibt die Distanz zweier Wörter zurück

strncmp

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

- Wenn der Inhalt der ersten `n` Zeichen von `s1` und `s2` gleich sind, dann liefert `strncmp` 0 zurück.
- Wenn der Inhalt der ersten `n` Zeichen von `s1` und `s2` ungleich sind, dann liefert `strncmp` etwas ungleich 0 zurück
- `strncmp` gibt die Distanz zweier Teilwörter der Länge `n` zurück

Beispiel strcmp

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

int main(int argc, char **argv)
{
    if(argc != 3)
    {
        printf("usage:  %s word1 word2\n", argv[0]);
        return 1;
    }

    if(!strcmp(argv[1], argv[2]))
        printf("%s and %s are equal\n", argv[1], argv[2]);
    else
        printf("%s and %s differ\n", argv[1], argv[2]);

    return 0;
}
```

strcat

```
#include <string.h>

char *strcat(char *dest, const char *src);
```

- Hängt die Zeichenkette `src` an die Zeichenkette `dest` an
- `dest` **muss** 0-terminierend sein, sonst ist das Verhalten von `strcat` nicht definiert
- `strcat` setzt das terminierende \0-Zeichen von alleine

strncat

```
#include <string.h>

char *strncat(char *dest, const char *src, size_t n);
```

- Hängt die ersten `n` Zeichen der Zeichenkette `src` an die Zeichenkette `dest` an
- `dest` **muss** 0-terminierend sein, sonst ist das Verhalten von `strncat` nicht definiert
- `strncat` setzt das terminierende \0-Zeichen von alleine (höchstens n+1 Zeichen)

Beispiel strcat/strncat

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

void foo()
{
    char str[1024];

    str[0] = 0; /* Äquivalent *str=0; */
    strcat(str, "Hallo");
    strncat(str, " Welt, die ganz grausam ist", 5);
    printf("%s\n", str); /* Ausgabe:  Hallo Welt */
}
```

Was passiert, wenn es stände `char str[5];`

strcpy

```
#include <string.h>

char *strcpy(char *dest, const char *src);
```

- Kopiert die Zeichenkette `src` (inklusive \0-Zeichen) in `dest`
- Die alte Zeichenkette `dest` wird überschrieben
- `src` muss 0-terminierend sein, sonst ist das Verhalten undefiniert

strncpy

```
#include <string.h>

char *strncpy(char *dest, const char *src, size_t n);
```

- Kopiert die ersten `n` Zeichen der Zeichenkette `src` in `dest`
- Achtung: Sollte das \0-Zeichen nicht unter den ersten `n` Zeichen vorhanden sein, ist das Resultat **nicht** 0-terminierend
- Die alte Zeichenkette `dest` wird überschrieben
- `src` muss 0-terminierend sein, sonst ist das Verhalten undefiniert

Beispiel strcpy/strncpy

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

void foo()
{
    char str[1024] = "Servuzzzz";
    printf("%s\n", str); /* Ausgabe:  Servuzzzz */
    strcpy(str, "Hallo");
    printf("%s\n", str); /* Ausgabe:  Hallo */
    strncpy(str, "Weltuntergang", 4);
    printf("%s\n", str);
    /* Ausgabe:  Welt oder Welto ?  */
}
```

Was passiert, wenn es stände `char str[5];`

strchr/strrchar

```
#include <string.h>

char *strchr(const char *s, int c);

char *strrchr(const char *s, int c);
```

- `strchr` sucht das **erste** Vorkommen des Zeichens `c` in `s`
- `strtcht` sucht das **letzte** Vorkommen des Zeichens `c` in `s`
- Wird das Zeichen gefunden, so wird einen Zeiger auf die resultierende Zeichenkette zurückgeliefert
- Wird das Zeichen nicht gefunden, so wird `NULL` zurückgeliefert
- `s` muss 0-terminierend sein, sonst ist das Verhalten undefiniert

strstr

```
#include <string.h>

char *strstr(const char *haystack, const char *needle);
```

- `strstr` sucht das erste Vorkommen der Teilzeichenkette `needle` in `haystack` und gibt einen Zeiger auf die resultierende Zeichenkette zurück
- Wenn die Teilzeichenkette nicht gefunden wird, so wird `NULL` zurückgeliefert
- `needle` und `haystack` müssen 0-terminierend sein, sonst ist das Verhalten undefiniert

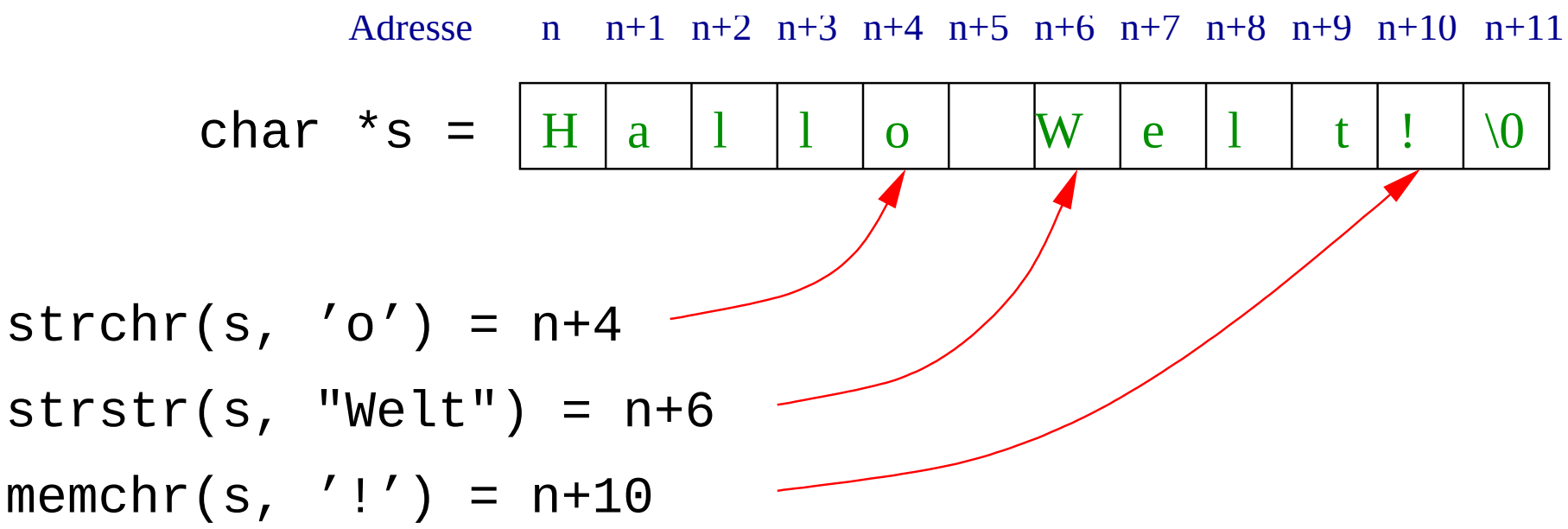
Beispiel strchr/strrchar/strstr

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

void foo()
{
    char str[] = "Wert 1 ; Wert 2; Wert 3; Wert 4";
    char *p_1, *p_2, *p_3;

    p_1 = strchr(str, ';');
    printf("%s\n", p_1); /* Ausgabe:  ; Wert 2; Wert 3; Wert 4 */
    p_2 = strrchr(str, ';');
    printf("%s\n", p_2); /* Ausgabe:  ; Wert 4 */
    p_3 = strstr(str, "Wert 3");
    printf("%s\n", p_3); /* Ausgabe:  Wert 3; Wert 4 */
    p_3 = strstr(str, "wert 3");
    printf("%s\n", p_3); /* Ausgabe:  (null) */
}
```

Suche im Speicher



strlen

```
#include <string.h>

size_t strlen(const char *s);
```

- strlen gib die Länge der Zeichenkette (ohne \0-Zeichen) zurück
- s muss 0-terminierend sein, sonst ist das Verhalten undefiniert

Beispiel strlen

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

int main(int argc, char **argv)
{
    int n;

    for(n = 0; n < argc; ++n)
        printf("Das %d Argument ist:  %s und hat Länge %d\n",
                n, argv[n], strlen(argv[n]));

    return 0;
}
```

Wie sieht die Ausgabe dieses Programms aus?

Speicherverwaltung

- Die folgenden Funktionen werden ebenfalls in `string.h` definiert.
- Nicht nur für Strings geeignet sondern im Prinzip für jeden Datentyp
- Sie helfen dabei, den Speicher sicher zu verwalten, wenn man sich an den Regeln hält

memchr/memrchr

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);

void *memrchr(const void *s, int c, size_t n);
```

- `memchr` sucht in den ersten `n` Bytes von `s` nach dem **ersten** Vorkommen vom Zeichen `c` in `s`. `memchr` gibt einen Zeiger auf das erste passende Byte (interpretiert als Zeichen)
- `memrchr` sucht in den letzten `n` Bytes von `s` nach dem **ersten** Vorkommen vom Zeichen `c` in `s`. `memchr` gibt einen Zeiger auf das erste passende Byte (interpretiert als Zeichen)
- Wenn das Zeichen nicht gefunden wird, wird `NULL` zurückgegeben

Beispiel memchr

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

void foo()
{
    int integers[] = {2, -12, 120, 3, 0 }, i = 0, *p;

    p = memchr(integers, 'x', sizeof(integers));

    while(p && p[i] != 0)
        printf("Adresse:  %x, Zahl:  %d\n", &p[i], p[i++]);
}
```

memcopy/memmove

```
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);

void *memmove(void *dest, const void *src, size_t n);
```

- Beide Funktionen kopieren `n` Bytes von `src` in `dest`
- Bei `memcpy` dürfen sich die Speicherbereiche von `src` und `dest` **nicht** überlappen
- Bei `memmove` dürfen sich die Speicherbereiche von `src` und `dest` überlappen, die Funktion ist sicherer als `memcpy` aber langsamer

Beispiel memcpy: Ohne Überlappung

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

char str1[] = "hallo WELT", str2[] = "HALLO welt, wie geht es dir?";

printf("str1 = %s, str2 = %s\n", str1, str2);
/* Ausgabe:  str1 = hallo WELT, str2 = HALLO welt, wie geht es dir?  */

memcpy(str1, str2, 5);

printf("str1 = %s, str2 = %s\n", str1, str2);
/* Ausgabe:  str1 = HALLO WELT, str2 = HALLO welt, wie geht es dir?  */
```

Beispiel memcpy: Mit Überlappung

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

char str[1024] = "abcdefghijklmnopqrstuvxyz";
char *p = strchr(str, 'c');

printf("str = %s, p = %s\n", str, p);

memcpy(p, str, 5);

printf("str = %s, p = %s\n", str, p);

/* str = abcdefghijklmnopqrstuvxyz, p = cdefghijklmnopqrstuvxyz */
/* str = ababadehijklmnopqrstuvxyz, p = abadehijklmnopqrstuvxyz */
```

Beispiel memmove: Mit Überlappung

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

char str[1024] = "abcdefghijklmnopqrstuvxyz";
char *p = strchr(str, 'c');

printf("str = %s, p = %s\n", str, p);

memmove(p, str, 5);

printf("str = %s, p = %s\n", str, p);

/* str = abcdefghijklmnopqrstuvxyz, p = cdefghijklmnopqrstuvxyz */
/* str = ababcdehijklmnopqrstuvxyz, p = abcdehijklmnopqrstuvxyz */
```

- `memmove` ist zwar langsamer als `memcpy`, dafür ist sie aber sicherer
- Wenn man weiß, dass die Speicherbereiche sich nicht überlappen, kann man ruhig `memcpy` nehmen, sonst `memmove`.
- Wir haben viele Beispiele bezogen auf C-Strings gesehen. Aber mit diesen Funktionen kann man beliebigen Speicher manipulieren
- Diese Funktionen setzen nicht von alleine das \0-Zeichen, wie z.B. `strcpy`, deswegen muss man als Programmierer immer darauf achten, wenn man mit C-Strings arbeitet
- Morgen werden wir das besser verstehen, wenn wir versuchen eine simple String Bibliothek für C zu schreiben