6.11 Extras/Umwandlungen zwischen Zahl und String

6.11.1 Unterschied zwischen Zahl und String

Häufig muss man eine Zahl in einen String umwandeln oder umgekehrt. Aber was ist eigentlich der Unterschied zwischen der Zahl `int i=1234;` und dem String `char s[]="1234"`?

Zuerst die Theorie:

- In `i` ist der Inhalt nach dem Dualsystem, also dem Stellenwertsystem zur Basis zwei sortiert. Das niederwertigste Bit hat den Wert eins. Das Bit davor hat den Wert zwei, davor das Bit den Wert vier usw.
In `s` ist das anders: Die letzte Ziffer hat den Wert eins (=Einerstelle), die Ziffer davor den Wert zehn (=Zehnerstelle), die Stelle davor den Wert hundert usw. Man hat hier das Dezimalsystem.
- In `s` hat jedes Ziffernzeichen nicht nur die vier Ziffernbits, die man braucht um zehn (maximal 16 Ziffern) zu unterscheiden. Jedes Zeichen belegt acht Bit, wovon bei ASCII sieben belegt sind. Man bekommt 128 mögliche Kombinationen, wovon die Ziffern nur zehn belegen. Diese zehn haben die Code-Nummern 48 bis 57, hexadezimal 0x30 bis 0x39 oder binär gesehen 00110000 bis 00111001. Das heißt, ein Byte mit der Code-Nummer 48+n entspricht einer Ziffer mit dem Wert n.
- In `i` ist klar, wie viele Bits die Variable belegt (je nach System meist 16, 32 oder 64).
In `s` ist das nicht klar. Es gibt kurze ("`3`") und lange ("`123456789`"). Deshalb wird jeder String durch einen Terminator ("`\0`", ASCII-Code-Nummer 0) abgeschlossen.
- In `i` gehört zu jedem möglichen Bitmuster eine gültige Zahl.
In `s` gibt es allerhand Möglichkeiten, den String so zu befüllen, dass keine Zahl erkannt wird: "`"`", "`Baum`", "`Muell+7`" usw.

Und nun ein (etwas langes) praktisches Beispiel: Zuerst soll gesehen werden, wie die Zahl `i=1234` im Speicher aussieht (`1234_als_int.c`):

```

1 #include <stdio.h>
2 #include <stdint.h>
3 int main(void)
4 {
5     int32_t i=1234; // int32_t ist eine Ganzzahl mit 4 Byte
6     char *p=(void *)&i;
7     printf("Adresse %u: Inhalt %hhx\n", p+3, *(p+3));
8     printf("Adresse %u: Inhalt %hhx\n", p+2, *(p+2));
9     printf("Adresse %u: Inhalt %hhx\n", p+1, *(p+1));
10    printf("Adresse %u: Inhalt %hhx\n", p+0, *(p+0));
11    return 0;
12 }
```

Terminal

```

schueler@debian964:~$ gcc 1234_als_int.c
schueler@debian964:~$ a.out
Adresse 123834935: Inhalt 0x0
Adresse 123834934: Inhalt 0x0
Adresse 123834933: Inhalt 0x4
Adresse 123834932: Inhalt 0xd2
```

Woher bekommt man diese Inhalte? Der Datentyp `int` (hier `int32_t`) funktioniert nach dem natürlichen Binärcode bzw. dem Dualzahlensystem (also dem Stellenwertsystem zur Basis zwei). Und 1234 entspricht der Dualzahl $10011010010_{(2)}$:

```

1024 | 512 | 256 | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0

```

Zur Kontrolle: $1024+128+64+16+2=$ Als Hexadezimalzahl ist das $4d2_{(16)}$ (in C als $0x4d2$ geschrieben).

Nehmen wir an, die Variable `i` steht im Speicher ab der Adresse 700. Dann sieht das im Speicher so aus:

```

Adr. | Inhalt
-----+-----
703 | 0000 0000
702 | 0000 0000
701 | 0000 0100
700 | 1101 0010

```

Nun machen wir das Gleiche mit dem String `s` (`1234_als_string.c`):

```

1 #include <stdio.h>
2 int main(void)
3 {
4     char s[]="1234";
5     char *p=s;
6     printf("Adresse %u: Inhalt %hhx\n", p+4, *(p+4));
7     printf("Adresse %u: Inhalt %hhx\n", p+3, *(p+3));
8     printf("Adresse %u: Inhalt %hhx\n", p+2, *(p+2));
9     printf("Adresse %u: Inhalt %hhx\n", p+1, *(p+1));
10    printf("Adresse %u: Inhalt %hhx\n", p+0, *(p+0));
11    return 0;
12 }

```

```

Terminal
schueler@debian964:~$ gcc 1234_als_string.c
schueler@debian964:~$ a.out
Adresse 2634470743: Inhalt 0x0
Adresse 2634470742: Inhalt 0x34
Adresse 2634470741: Inhalt 0x33
Adresse 2634470740: Inhalt 0x32
Adresse 2634470739: Inhalt 0x31

```

Woher kommen die Inhalte 31 bis 34? Im ASCII wird die Ziffer '1' durch das Bitmuster 0011 0001 dargestellt. Das ist die Codenumber 49, hexadezimal ist das der Wert $31_{(16)}$ (in C als $0x31$ geschrieben). Genauso muss man jetzt die anderen Ziffern decodieren. $0x32$ ist die Ziffer '2', $0x33$ die Ziffer '3', und $0x34$ ist die Ziffer '4'. Die Zahl $0x0$ ist das Bitmuster 0000 0000, die Codenumber 0, also der Terminator `'\0'`.

Wieder bei Adresse 700 sähe der Inhalt des Speichers jetzt so aus:

```

Adr. | Inhalt      | Bedeutung
-----+-----+-----
704 | 0000 0000 | Terminator
703 | 0011 0100 | Ziffer '4'
702 | 0011 0011 | Ziffer '3'
701 | 0011 0010 | Ziffer '2'
700 | 0011 0001 | Ziffer '1'

```

Die Einerstelle ist hier die Stelle direkt vor dem Terminator, die Zehnerstelle die Stelle davor usw. Mit diesem Wissen kann man jetzt den Wert der Zahl berechnen.

6.11.2 Umwandlung Zahl nach String

6.11.2.1 Standard: `sprintf()` Manchmal hat man Daten als Zahl, möchte sie aber in String-Form haben. Typischerweise passiert das bei der Ausgabe. Für die Ausgabe auf der Konsole (dem Terminalfenster) gibt es die Funktion `printf()`, die keine Wünsche offen lässt. Mit den entsprechenden Platzhaltern (z. B. `%d` für Dezimalausgabe von `int`) kann man Zahlen so bequem ausgeben, ohne sich Gedanken um ASCII-Codenummern zu machen. Für die Ausgabe in Dateien ist `fprintf()` zuständig. Wenn man nun innerhalb eines Programms eine Zahl in einen String umwandeln will, kann man die Funktion `sprintf()` verwenden. Ähnlich wie `fprintf()` erwartet auch `sprintf()` vor dem Formatstring (der mit den Platzhaltern) einen Parameter, der angibt, wohin geschrieben werden soll. Aber hier ist es kein Dateimerker, sondern die Startadresse des Strings:

```
1  int i=1234;
2  char s[81];
3  sprintf(s, "%d", i);
```

Es gibt einen großen Unterschied zwischen `sprintf()` auf der einen Seite und `printf()` und `fprintf()` auf der anderen Seite. Bei `printf()` muss man sich nicht um die Position des Cursors auf dem Bildschirm kümmern. Wenn man zehn Mal ein "x" mit `printf()` ausgibt, erscheinen zehn x nebeneinander auf dem Bildschirm. Gibt man zehn Mal ein "x" mit `fprintf()` in eine Datei, werden zehn Bytes in die Datei geschrieben. Das ist bei `sprintf()` anders. `sprintf()` führt keinen Zeiger mit, der die Position speichert, an die geschrieben wurde. Deshalb muss man selbst aktiv werden, wenn man mit `sprintf()` Strings aneinanderhängen will: `sprintf()` gibt als Rückgabewert die Anzahl der geschriebenen Bytes (ohne den Terminator!) aus. Damit kann man aus der alten Adresse und dem Rückgabewert die neue Adresse ermitteln:

```
1 #include <stdio.h> /* auch sprintf() wird hier deklariert */
2 int main(void)
3 {
4     char s[81];
5     char *p=s;
6     int rc;
7     rc=sprintf(p, "Hello, ");
8     p=p+rc;
9     rc=sprintf(p, "World!\n");
10    printf("%s", s); /* Ausgabe zur Kontrolle */
11    return 0;
12 }
```

Terminal

```
schueler@debian964:~$ gcc sprintf.c
schueler@debian964:~$ ./a.out
Hello, World!
```

6.11.2.2 Mehr Sicherheit: `snprintf()` Wenn man komplizierte Formatstrings hat (z. B. `%d Volt %s. %i, %i\n`), dann ist es mühsam, zu gewährleisten, dass nicht zu viele Bytes nach `s` geschrieben werden. Deshalb gibt es seit C99 die Funktion `snprintf()`, bei der der zweite Parameter die Länge des Puffers angibt (ähnlich wie bei `fgets()` oder `strftime()`):

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int i=1234;
5     char s[31];
```

```

6     snprintf(s, 31, "Meine_Lieblingszahl:_%d._Wirklich!", i);
7     printf("Ergebnis-String: >>%s<<\n", s);
8 }

```

```

Terminal
schueler@debian964:~$ gcc snprintf1.c
schueler@debian964:~$ ./a.out
Ergebnis-String: >>Meine Lieblingszahl: 1234. Wir<<

```

Beim Rückgabewert gibt es hier eine Besonderheit: `snprintf()` gibt die Anzahl der Zeichen zurück, die hätten geschrieben werden müssen (ohne den Terminator), wenn da nicht die Begrenzung durch den zweiten Parameter gewesen wäre. Anders gesagt: Falls in diesem Fall der Rückgabewert höchstens 30 ist, dann ist nichts abgeschnitten worden, und der Rückgabewert gibt an, wie viele Zeichen geschrieben wurden. Andernfalls (Rückgabewert 31 oder größer) wurden auch nur 30 Zeichen geschrieben und der Rest wurde abgeschnitten. Wenn einen die Frage interessiert, ob es mit der Stringgröße gepasst hat, muss man eben eine Fallunterscheidung vornehmen:

```

1 #include <stdio.h>
2 int main(void)
3 {
4     int rc, i=0;
5     char s[7]="";
6     printf("Eingabe:_");
7     scanf("%d", &i);
8     rc=snprintf(s, 7, "%d", i); /* schreibt. max. 7 Bytes incl. '\0' */
9     if(rc>=7)
10    {
11        printf("Wollte_%d_Zeichen_schreiben, _konnte_aber_nur_6_schreiben.\n", rc);
12    }
13    else
14    {
15        printf("Habe_wie_gewuenscht_%d_Zeichen_geschrieben.\n", rc);
16    }
17    printf("Ergebnis-String: >>%s<<\n", s);
18    return 0;
19 }

```

```

Terminal
schueler@debian964:~$ gcc snprintf2.c
schueler@debian964:~$ ./a.out
Eingabe: 123
Habe wie gewuenscht 3 Zeichen geschrieben.
Ergebnis-String: >>123<<
schueler@debian964:~$ ./a.out
Eingabe: 123456789
Wollte 9 Zeichen schreiben, konnte aber nur 6 schreiben.
Ergebnis-String: >>123456<<

```

6.11.3 Umwandlung String nach Zahl

Oft ist es auch umgekehrt, dass man einen String hat, in dem eine Zahl eingebaut ist und man an die Zahl herankommen möchte. Hierfür gibt es die Funktion `sscanf()`, mit der man – wie bei `sprintf()` – statt der Konsole (dem Terminalfenster) einen String bedient. Außerdem gibt es in `<stdlib.h>` einige Hilfsfunktionen, die einfacher zu handhaben sind.

6.11.3.1 Gewohnheit: `sscanf()` Die Funktion `sscanf()` ist so zu benutzen wie `scanf()`: Man hat einen Formatstring mit (hoffentlich nur einem) Platzhalter, dahinter die Adresse(n) der Variable(n), in die geschrieben wird. Und wie bei `fscanf()` gibt es einen zusätzlichen Parameter am Anfang der Parameterliste, der die Quelle angibt: Hier eine Adresse, an der ein String steht mit Zahlen oder sonstigen Dingen, aus denen gelesen werden soll:

```

1 #include <stdio.h>
2 int main(void)
3 {
4     char s[]="230.5_Volt";
5     double ueff=0.0, us;
6     sscanf(s, "%lf", &ueff);
7     us=ueff*1.4142; /* mal was Sinnvolles */
8     printf("Spitzenwert: Us=%g_V\n", us);
9     return 0;
10 }
```

Zeile 6 Erster Parameter: Quellstring `s`, zweiter Parameter: Platzhalter, dritter Parameter: Adresse des Ziels `ueff`

Wie `sprintf()` führt auch `sscanf()` leider die Anzahl der eingelesenen Bytes nicht mit. Bei jedem Aufruf von `sscanf()` fängt man daher wieder beim Anfang des Strings an. Leider ist der Rückgabewert von `sscanf()` (wie der von `scanf()`) nicht die Anzahl der eingelesenen Zeichen, sondern die Anzahl der umgewandelten Platzhalter. Deshalb kann man ihn nicht gebrauchen, um innerhalb eines Strings mit dem Lesen voranzukommen. Stattdessen gibt es den Platzhalter `%n`, mit dem man die Anzahl der bisher eingelesenen Zeichen in eine weitere Variable schreiben kann.

6.11.3.2 Einfachheit: `atoi()` und Kollegen Wenn man einfach nur einen String in eine Zahl umwandeln will, reicht eine der folgenden Funktionen aus `<stdlib.h>`:

- a) `atoi()`: Wandelt String in `int`-Variable (*ascii-to-int*)
- b) `atol()`: Wandelt String in `long`-Variable (*ascii-to-long*)
- c) `atoll()`: Wandelt String in `long long`-Variable (*ascii-to-long-long*) – seit C99
- d) `atof()`: Wandelt String in `double`-Variable (*ascii-to-float*) – seit C99

Diese Hilfsfunktionen (`<stdlib.h>` enthält das, was sonst nirgends hineinpasst) gibt es offenbar nur für die wichtigsten Datentypen. Ihre Anwendung ist absolut einfach (`atof1.c`):

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(void)
4 {
5     char s[]="115_Volt";
6     double ueff, us;
7     ueff=atof(s); /* das ist alles */
8     us=ueff*1.4142.;
9     printf("Us=%f_V\n", us);
10    return 0;
11 }
```

Man sieht, dass `atof()` nur so lange liest, bis es kein Zeichen mehr findet, welches zu einer Gleitkommazahl passt. Dahinterliegende Zeichen (auch Ziffernzeichen) werden nicht beachtet. Falls schon zu Beginn nichts Sinnvolles gefunden wird (also nichts, was Teil einer Gleitkommazahl sein

könnte), ist der Rückgabewert 0.0. Fehler kann man so nicht finden: `atof("0")` hat denselben Rückgabewert wie `atof("Hallo")`. Wenn das nichts ausmacht, ist `atof()` ausreichend. Für `atoi()`, `atol()` und `atoll()` gilt sinngemäß dasselbe.

6.11.3.3 Sonderfall Überlauf Nun kann es passieren, dass im String eine so große oder eine so kleine Zahl steht, dass sie nicht mehr in den passenden Datentyp umgewandelt werden kann. Bei `double`-Variablen passiert dann folgendes:

- Der Rückgabewert ist bei zu großen Werten `HUGE_VAL` und `-HUGE_VAL`, bei zu kleinen Werten `+0` oder `-0`
- Die globale Fehlervariable der Standardbibliothek mit dem Namen `errno` erhält einen bestimmten Wert ungleich null, der der symbolischen Konstante `ERANGE` entspricht; die Bedeutung des Wertes kann mit `perror()` ausgegeben werden

Das Beispiel `atof2.c` zeigt vier solcher Sonderfälle:

```

1 #include <errno.h>
2 #include <math.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 int main(void)
7 {
8     char s[81];
9     double zahl;
10
11     errno=0;
12     strcpy(s, "123.0"); /* Fall 0: Zahl o.k. */
13     zahl=atof(s);
14     printf("String=>>>%s<<<\n", s);
15     printf("Zahl=%g\n", zahl);
16     printf("errno=%i\n", errno);
17     perror("");
18     printf("\n");
19
20     errno=0;
21     strcpy(s, "-123.0e1000"); /* Fall 1: Zahl zu gross */
22     zahl=atof(s);
23     printf("String=>>>%s<<<\n", s);
24     printf("Zahl=%g\n", zahl);
25     printf("errno=%i\n", errno);
26     perror("");
27     printf("\n");
28
29     errno=0;
30     strcpy(s, "123.0e-1000"); /* Fall 2: Zahl zu klein */
31     zahl=atof(s);
32     printf("String=>>>%s<<<\n", s);
33     printf("Zahl=%g\n", zahl);
34     printf("errno=%i\n", errno);
35     perror("");
36     printf("\n");
37
38     errno=0;

```

```

39     strcpy(s, "INFINITY"); /* Fall 3: Unendlich eingegeben */
40     zahl=atof(s);
41     printf("String=>>%s<<\n", s);
42     printf("Zahl=%g\n", zahl);
43     printf("errno=%i\n", errno);
44     perror("");
45     printf("\n");
46
47     errno=0;
48     strcpy(s, "NAN"); /* Fall 4: Not-A-Number (z.B. 0.0/0.0) */
49     zahl=atof(s);
50     printf("String=>>%s<<\n", s);
51     printf("Zahl=%g\n", zahl);
52     printf("errno=%i\n", errno);
53     perror("");
54     return 0;
55 }

```

```

Terminal
schueler@debian964:~$ gcc atof2.c
schueler@debian964:~$ ./a.out
String=>>123.0<<
Zahl=123
errno=0
Success

String=>>123.0e1000<<
Zahl=inf
errno=34
Numerical result out of range

String=>>123.0e-1000<<
Zahl=0
errno=34
Numerical result out of range

String=>>INFINITY<<
Zahl=inf
errno=0
Success

String=>>NAN<<
Zahl=nan
errno=0
Success

```

Fall 0 ist der Fall ohne Fehler. `errno` ist null, `perror()` gibt Success aus.

Bei Fall 1 ist die Zahl $123 \cdot 10^{1000}$ nicht mehr als double darstellbar. Das Ergebnis ist `HUGE_VAL`, was auf den meisten heutigen Systemen mit ∞ gleich ist. Die Ausgabe ist deshalb `inf` (`printf()` nennt diese Zahl so). `errno` bekommt den Wert 34, das ist die Fehlerkonstante `ERANGE` mit der Bedeutung `Numerical result out of range`. Wenn man will, kann man diese Information benutzen.

Bei Fall 2 ist es genauso. Ergebnis ist `0.0`, `errno` wiederum `ERANGE`.

Bei Fall 3 wird gleich der String `INFINITY` eingelesen (`INF` ginge genauso). Der String könnte ja Ergebnis eines `printf()`-, `fprintf()`- oder `sprintf()`-Aufrufs sein. Er wird wiederum richtig als Überlauf interpretiert mit dem Ergebnis `HUGE_VAL`. Da aber kein aktueller Fehler bei der Umwandlung aufgetreten ist, ist `errno` diesmal null.

Fall 4 ist eine weitere Spezialität: Der String `NAN` symbolisiert das Ergebnis einer mathematisch nicht erlaubten Gleitkomma-Operation, z. B. $\frac{0}{0}$ oder $\sqrt{-1}$. Dieser Wert wird übernommen. Da kein aktueller Fehler bei der Umwandlung aufgetreten ist, ist `errno` auch hier null.

6.11.3.4 Mit Fehlersuche: `strtod()` und Kollegen Wer wissen möchte, wie weit `atof()` gelesen hat und wo das erste nicht passende Zeichen war, der kann auf folgende Funktionen zugreifen:

- a) `strtol()`: Wandelt String in long-Variable (*string-to-long*)
- b) `strtoll()`: Wandelt String in long long-Variable (*string-to-long-long*) – seit C99
- c) `strtoul()`: Wandelt String in unsigned long-Variable (*string-to-unsigned-long*)
- d) `strtoull()`: Wandelt String in unsigned long long-Variable (*string-to-unsigned-long-long*) – seit C99
- e) `strtof()`: Wandelt String in float-Variable (*string-to-float*) – seit C99
- f) `strtod()`: Wandelt String in double-Variable (*string-to-double*)
- g) `strtold()`: Wandelt String in long double-Variable (*string-to-long-double*) – seit C99

Es fällt auf, dass es keine `strtoi()`-Funktion gibt. Merkmal dieser Funktionen ist der zweite Parameter. Man übergibt hier der Funktion die Adresse eines Zeigers. Die Funktion schreibt dann in den Zeiger die Adresse, ab der das erste Zeichen steht, das nicht mehr zu einer Zahl gehören kann:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h> /* nur fuer strcpy() */
4 int main(void)
5 {
6     char s[81]="";
7     char *fehleradr;
8     double u;
9
10    strcpy(s, "230.7"); /* Fall 1: Nur die Zahl */
11    printf("An Adresse_%u_ steht: >>%s<<\n", s, s);
12    u=strtod(s, &fehleradr);
13    printf("Nach_strtod():_U=%f_V\n", u);
14    printf("An Fehleradr._%u_ steht: >>%s<<\n\n", fehleradr, fehleradr);
15
16    strcpy(s, "230.7_BLABLA"); /* Fall 2: Zahl+Muell */
17    printf("An Adresse_%u_ steht: >>%s<<\n", s, s);
18    u=strtod(s, &fehleradr);
19    printf("Nach_strtod():_U=%f_V\n", u);
20    printf("An Fehleradr._%u_ steht: >>%s<<\n\n", fehleradr, fehleradr);
21
22    strcpy(s, "BLUBB"); /* Fall 3: Nur Muell */
23    printf("An Adresse_%u_ steht: >>%s<<\n", s, s);
24    u=strtod(s, &fehleradr);
25    printf("Nach_strtod():_U=%f_V\n", u);

```

```

26     printf("An Fehleradr. %u steht: >>%s<<\n\n", fehleradr, fehleradr);
27
28     return 0;
29 }

```

```

Terminal
schueler@debian964:~$ gcc strtod.c
schueler@debian964:~$ ./a.out
An Adresse 3219855296 steht: >>230.7<<
Nach strtod(): U=230.700000 V
An Fehleradr. 3219855301 steht: >><<

An Adresse 3219855296 steht: >>230.7 BLABLA<<
Nach strtod(): U=230.700000 V
An Fehleradr. 3219855301 steht: >> BLABLA<<

An Adresse 3219855296 steht: >>BLUBB<<
Nach strtod(): U=0.000000 V
An Fehleradr. 3219855296 steht: >>BLUBB<<

```

Man sieht hier drei verschiedene Fälle:

- Fall 1: Im String ist nur die Zahl enthalten. Die Variable `fehleradr` zeigt auf das erste Zeichen nach dem String, also auf den Terminator. Wenn gewünscht, könnte man diesen Fall abfragen mit `(*fehleradr=='\0')` oder mit `(strlen(s)==fehleradr-s)`.
- Fall 2: Im String ist nach der Zahl noch weiterer Inhalt. Die Variable `fehleradr` zeigt auf das erste Zeichen, das nicht mehr zur Zahl gehört.
- Fall 3: Im String gibt es schon zu Beginn keinen Inhalt, der als Zahl zu interpretieren wäre. Die Variable `fehleradr` ist damit gleich der Startadresse des Strings. Wenn gewünscht, könnte man diesen Fall einfach mit `(s==fehleradr)` abfragen. Der Rückgabewert ist in diesem Fall `0.0`.

6.11.3.5 Mit beliebiger Basis: `strtol()` und Kollegen Die Funktionen `strtol()`, `strtoll()`, `strtoul()` und `strtoull()` haben noch ein kleines Feature eingebaut: Man kann bei ihnen die Basis des Zahlensystems eingeben¹, wobei jede Basis zwischen 2 und 36 erlaubt ist. Die möglichen Ziffernzeichen sind dann je nach Basis 0 bis 9 und A bis Z. Damit ist – anders als bei `sscanf()` – hier auch die Eingabe von Dualzahlen möglich (`strtol.c`):

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(void)
4 {
5     char s[81]="10101010";
6     long zahl;
7     zahl=strtol(s, NULL, 2);
8     printf("%d\n", zahl);
9     return 0;
10 }

```

¹Gibt man null ein, wird (wie bei `scanf()` mit Platzhalter `%i` als Basis 8, 10 oder 16 verwendet: Zahlen, die mit `0x` beginnen, werden hexadezimal interpretiert, sonstige Zahlen, die mit `0` beginnen, oktall und alle anderen dezimal.

Zeile 7 Dritter Parameter ist die Basis 2, das entspricht dem Dualzahlensystem. Zweiter Parameter ist die Fehleradresse. Wenn sie wie hier NULL ist, wird sie nicht benutzt.

Zeile 8 Kontrollausgabe: $10101010_{(2)} = 170$