C Grundlagen

Andreas Wagner

9. März 2025

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Vorwort

In diesem Buch möchten wir (Autor, Lektoren, Testleser) Sie befähigen, C-Quellcode zu lesen und selbst zu schreiben.

Ich stelle Ihnen nun die einzelnen Abschnitte des Buches vor.

Der Abschnitt „Rundreise durch die Sprache“ erklärt wenige grundlegende Operationen und soll helfen, die Beispiele in den restlichen Abschnitten besser zu verstehen. In dem Abschnitt „Grundlagen“ gibt es alle Informationen, welche Sie für grundlegende, einfache Programme be-

nötigen. Ich haben das Kapitel in „Grundlegende Sprachelemente“ und „Weiterführende Sprachelemente“ unterteilt, um kleinere Teile des Textes zusammenfassen zu können.

Der Abschnitt „Beispiele“ vertieft die Kenntnisse und gibt Ideen weiter, welche sich häifig in Programmen wiederholen. Darauf folgt ein Abschnitt „Gefahren“, welcher beschreibt, worauf man beim Nutzen der Grundlagen achten sollte. Der Abschnitt „Fortgeschrittenes“ enthält Informationen über Multithreading, UTF-8 und andere fortgeschrittene Themen.

1.2 Konventionen

Dieses Buch nutzt die folgende Zeichen um Anmerkungen, Tipps und Warnungen hervorzuheben:

Dies ist eine Anmerkung.

Dies ist ein Tip.

Dies ist eine Warnung.

C-Quellcode sieht folgendermaßen aus (Sie müssen ihn noch nicht verstehen):

```
1  // Dies ist ein Kommentar; er wird nicht ausgefü
       hrt.
2  #include <stdio.h>  // Standard-I/O einbinden
3
4  int main(void)       // Programm akzeptiert keine
       Argumente
5  {
6      // Definiert eine Zeichenkette.
7    char *string = "Welt!";
8
9      // Gibt Zeichenkette aus.
10   printf("Hallo %s", string);
11
12   // Beendet ohne Fehlerstatus.
13   return EXIT_SUCCESS;
14 }
```

1.3 Der Build-Vorgang

Mit „Build“ wird die Arbeit des Computers bezeichnet, welche aus Quelltext ausführbaren Maschinencode erzeugt.

Aus dem Quelltext in den „.c“-Dateien macht der „Compiler“ entweder

- Assembler-Code, welcher vom „Assembler“ zu Maschinencode gemacht wird oder
- direkt Maschinencode, welcher vom Prozessor ausgeführt werden kann.

Diesen Vorgang nennt man „kompilieren“.

Ein Programm namens „Linker“ fügt die Standardbibliothek und weitere angeforderte Bibliotheken zu dem Maschinencode hinzu. Zu Zeiten der dynamisch geladenen Bibliotheken werden inzwischen nur die Lade-Anweisungen für die Bibliotheken hinzugefügt.

Kapitel 2

Ein einführendes Beispiel

2.1 Zweck des Abschnittes

Dieser Abschnitt soll einen groben Überblick darüber geben, was die einzelnen Sprachelemente bedeuten. Ich zeige ein Beispiel, welches Texte auf dem Bildschirm ausgibt und von Tastatur einliest. Ich erkläre sie im Detail um eine Grundlage dafür zu schaffen, die Beispiele in den folgenden Kapiteln zu verstehen. So erwarte ich, dass Sie den Befehl printf() grundlegend verstanden haben, wenn sie den Abschnitt durchgearbeitet haben. Die späteren Beispiele

bauen darauf auf.

2.2 Ein menügesteuertes Programm

Hier zeige ich Ihnen ein Programm, welches ein Zeichen von der Tastatur einliest und abhängig von der gewählten Option eine Aktion durchführt:

- Die Option 1 gibt „Hallo Welt!“ aus,
- die Option 2 zählt die Flaschen Apfelsaft auf dem Tresen und
- Option 3 beendet das Programm.

```
1  #include <stdio.h>
2
3  // Hier wird eine Funktion definiert, welche die
       Zeichenkette "Hallo Welt!" ausgibt.
4  void print_hallo_welt()
5  {
6    printf("Hallo Welt!\n");
7  }
8
9  // Hier wird eine Funktion definiert, welche 99
       Zeilen ausgibt.
10 void print99()
11 {
12   for(int i = 99; i > 0 ; i--)
```

```
13    {
14      printf("%d Flaschen Apfelsaft stehen auf dem
        Tresen.\n", i);
15    }
16  }
17
18  // Machen Sie sich bitte noch keine Gedanken darü
      ber, was die folgende Zeile im Detail macht. "
      main()" ist der Einsprungspunkt für das
      Programm.
19  int main(int argc, char * argv[])
20  {
21    // Wir teilen dem Compiler mit, dass wir Platz
22    // für eine Variable brauchen.
23    int option;
24    do
25    {
26      printf("Bitte wählen Sie aus:\n");
27      printf("1. \"Hallo Welt!\" ausgeben\n");
28      printf("2. \"99 Flaschen...\" ausgeben\n");
29      printf("3. Programm beenden\n");
30
31      // Gewählte Option von Tastatur einlesen.
32      option = getchar();
33
34      // Return-Taste ignorieren
35      while(getchar() != '\n');
36
37      if('1' == option)
38      {
39        print_hallo_welt();
```

```
40      }
41      else if('2' == option)
42      {
43        print99();
44      }
45      else if('3' == option)
46      {
47        printf("Programm wird beendet\n");
48      }
49      else
50      {
51        printf("Ungültige Eingabe!\n");
52      }
53    }
54    while('3' != option);
55    return 0;
56  }
```

Kapitel 3

Grundlagen

3.1 Statements

In C werden alle auszuführenden Operationen mit einem Semikolon abgeschlossen.

```
1 int i;  // Reserviere Speicher um einen Integer
      halten zu können.
2 printf("Test!");  // Gib "Test!" auf der Konsole
      aus.
```

3.2 Präprozessoranweisungen

Bevor der Quellcode in Maschinensprache übersetzt wird, wird er mit einer programmierbaren Textersetzung überarbeitet. Es ist ein Vorverarbeiter, genannt „Präprozessor“.

```
1 #include <stdio.h>
2 #define ERROR(x) fprintf(stderr, "Error: %s", x);
3 #undef ERROR(x)
4 #ifdef STDIO_H
5 #endif
```

#include <stdio.h> kopiert den Inhalt der Datei „stdio.h“ an die Stelle des Befehls. Im Kapitel 8 „Modularisierung“ wird genauer erklärt, was es damit auf sich hat.

#define ERROR(x) fprintf(stderr, "Error: \%s", x) definiert ein sogenanntes Macro. Diese werden vor dem Kompilieren ersetzt. So steht an der Stelle der Zeichenkette

```
1 ERROR("Mein Fehler.")
```

anschließend die folgende Zeichenkette:

```
1 fprintf(stderr, "Error :\%s", "Mein Fehler.");
```

Sie müssen noch nicht verstehen, was der Befehl im Detail macht, ich will es aber kurz erläutern: fprintf(stderr, ...) sagt aus, dass auf dem Standard-Fehlerkanal etwas ausgegeben werden soll. "Error: \%s" teilt dem Befehl mit, dass die Zeichenkette „Error:“ ausgegeben werden soll, gefolgt von einem Leerzeichen und einer Zeichenkette (im

Englischen „String“), welche in einem separaten Parameter übergeben wird. Das Ganze wird in Kapitel 6.1, „printf() und Verwandte“ genauer erkläutert.

Der Befehl #undef ERROR(x) entfernt die Anweisung wieder aus dem Präprozessor, so dass die Zeichenkette nicht mehr ersetzt wird.

```
1 #ifndef STDIO_H
2 #else
3 #endif
```

#ifdef STDIO_H überprüft, ob das Macro oder die Präprozessorvariable STDIO_H definiert ist und

- wenn ja, dann wird der Quelltext eingefügt, welcher zwischen dem #ifdef und den nächsten #else oder #endif steht.
- Folgt ein #else auf das #ifdef, so wird der Bereich zwischen #else und #endif genau dann genutzt, wenn STDIO_H *nicht* definiert ist.

#ifndef ist das Gegenstück zu #ifdef, es fügt den folgenden Block in den Quelltext ein, wenn das Macro oder die Präprozessorvariable nicht definiert ist.

```
1 #if BEDINGUNG
2 #elif BEDINGUNG
3 #elif BEDINGUNG
4 #else
5 #endif
```

#if BEDINGUNG fügt den Code bis zum nächsten

- #elif,
- #else oder
- #endif

ein, wenn die angegebene Bedingung wahr ist. Ist die Bedingung falsch und es folgt ein #elif BEDINUNG, dessen Bedingung wahr ist, so wird der Block bis zum nächsten

- #elif,
- #else oder
- #endif

genutzt. Genauso funktioniert es auch mit #else und #endif.

```
1 #pragma omp parallel for
```

#pragma wird für Compiler-Erweiterungen genutzt, welche nicht im Sprachstandard definiert sind. Eine sehr bekannte Erweiterung ist OpenMP, welches Vereinfachungen für die parallele Verarbeitung mittels mehrerer Threads und/oder SIMD[1]-Befehlen zur Verfügung stellt.

[1]Single Instruction Multiple Data – Das sind Prozessorbefehle, welche auf größere Datensätze angewendet werden.

3.3 Blöcke

Codeblöcke sind zum Strukturieren des Quellcodes da. Der Codeblock, welcher der Funktion main() zugeordnet wird, ist der Block, welcher zu Beginn des Programmes ausgeführt wird.

```
1 int main(void)
2 {
3   printf("Hallo Welt!");
4 }
```

Siehe auch das Kapitel 4.7, „Gültigkeitsbereiche von Variablen“.

3.4 Kommentare

Quellcode wird erst mir Kommentaren wirklich gut lesbar. In C werden Kommentare auf zwei verschiedene Arten gemacht:

- Mittels zweiter Schrägstriche wird ein Kommentar bis zum Zeilenende ausgezeichnet. (`int i; // Iterator`)
- Textblöcke werden mittels Schrägstrich und Sternchen als Kommentare ausgezeichnet (`int i; /* Iterator */`.) Diese Blöcke dürfen sich über mehrere Zeilen erstrecken.

Es gibt eine Software namens *Doxygen*, welche aus C/C++-Quellcode- Kommentaren HTML-Dokumentation erzeugt. Ich rate dazu, sich diese Schreibweise anzugewöhnen!

```
1  /** Einsprungspunkt für das Programm.
2   * @return EXIT_SUCCESS, falls alles
       funktioiniert.
3  */
4  int main(
5    int argc,      //< Anzahl der Programmparameter
6    char *argv[]) //< Array von Zeigern auf die
       Programmparameter
7  {
8    printf("Hallo Welt!");  // Wir grüßen die Welt!
9    return EXIT_SUCCESS;  // Und beenden ohne
       Fehlermeldung.
10 }
```

3.5 Deklaration und Definition

Folgender Quellcode *deklariert* eine Variable mit der Bezeichnung „zahl“.

```
1  int zahl;
```

Ab diesem Zeitpunkt kann auf sie zugegriffen werden. Man kann ihr Werte zuweisen und ihren Wert abfragen. Deklarierte, undefinierte Variablen enthalten undefinierte Werte. Es wird davon abgeraten, Variablen abzufragen, bevor sie einen Wert zugewiesen bekommen haben.

Der nachstehende Quellcode *definiert* eine Variable mit dem Namen zahl und dem Wert 5.

```
1 int zahl = 5;
```

Kapitel 4

Grundlegende Sprachelemente

In diesem Kapitel werden die Datentypen beschrieben, mit denen man täglich Umgang hat. Zusätzlich werden Operationen aufgelistet, welche man mit den Datentypen durchführen kann. Atomizität und volatile werden in späteren Kapiteln beschrieben.

4.1 Ganzzahlige Typen

Ganzzahlige Datentypen[1] werden als Bitfelder abgespeichert. Ein Bit hat den Wert 0 oder 1. Ihre Wertigkeit steigt – je nach Systemtyp – von Links nach Rechts oder von Rechts nach Links. Diesen Unterschied nennt man „Endianess“. Es gibt „Big Endian“-Systeme und „Little Endian“-Systeme. Letztere sind häufiger anzutreffen und werden hier zur Veranschaulichung genutzt.

Die folgende Tabelle zeigt auf, wie die Zahl 58 auf einem Little-Endian-System dargestellt wird.

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7
1	2	4	8	16	32	64	128
0	1	0	1	1	1	0	0

todo: Ich kann mich bei Big-Endian und Little-Endian vertun, es kann genau falschherum sein.

4.1.1 Darstellung negativer Zahlen

Es gibt zwei Möglichkeiten negative Zahlen darzustellen:

- Einerkomplement, bei welchem die Bits invertiert interpretiert werden und das erste Bit ein Vorzeichen-

[1] Ganzzahlige DAtentypen werden im Englischen „Integer“ genannt.

bit ist; 1 ist negativ, 0 ist positiv. Bei dem Einerkomplement sind 0000 und 1111 auf die Null abgebildet.

- Zweierkomplement, welches wie das Einerkomplement funktioniert, aber aus mathematischen Gründen bei negativen Zahlen +1 gerechnet wird.

Zweierkomplement ist die übliche Darstellungsweise, da es mathematisch besser passt (zeigen wir gleich).

Rechnen wir zunächst 5 + (-3) im Zweierkomplemet. 5 ist binär, little-endian 0101 und (-3) ist im Einerkomplement, binär, little-endian 1100[2]. Im Zweierkomplement also 1101.

Vorzeichenbit	2^0	2^1	2^2
-	1	2	4
0	1	0	1
1	1	0	1
0	0	1	0

Bei den Werigkeiten 2^2 und 2^0 kommt es zum Übertrag und wegen des Überlaufs bei bit 2^2 muss das Vorzeichenbit gekippt werden. Somit bekommt man das richtige Ergebnis, 2.

Durch die Addition der 1 bei der negativen Zahl kommt also bei einer normalen Addition genau der richtige Wert

[2] 3 ist 0011

heraus. Zudem gibt es die Null im Zweierkomplement nur einmal.

4.1.2 Schlüsselwörter zur Deklaration und Definition

Folgendes Beispiel reserviert Speicher für eine Integer-Konstante und initialisiert ihn mit Null. Nach der Reservierung kann der Speicher genutzt werden.

```
1 int main(int argc, char * argv[])
2 {
3   int i = 0;
4 }
```

Es gibt viele Datentypen. Hier ist eine Liste der alten Namen (C Standard C89 und spätere), welche noch häufig in Quellcodes zu finden sind.

Datentyp	kleinster Wert	größter Wert	Anmerkungen
unsigned char	0	255	Meist 8 Bit, bei manchen Systemen 16 Bit breit.
char	-128	127	Meist 8 Bit, bei manchen Systemen 16 Bit breit. Wird für Schriftzeichen genutzt.
unsigned short	0	65.535	16 Bit breit.
short	-32.768	32.767	16 Bit breit.

Datentyp	kleinster Wert	größter Wert	Anmerkungen
unsigned int	0	65.535	Mindestens 16 Bit breit, darf auch unsigned long (32 Bit) sein.
int	-32.768	32.767	Mindestens 16 Bit breit, darf auch long (32 Bit) sein.
unsigned long	0	$4 \cdot 10^{10}$	32 Bit breit.
long	$-2 \cdot 10^{10}$	$2 \cdot 10^{10}$	32 Bit breit.
unsigned long long	0	$1.8 \cdot 10^{20}$	64 Bit breit. Auf manchen Embedded-Systemen nicht vorhanden.
long long	$-9 \cdot 10^{19}$	$9 \cdot 10^{19}$	64 Bit breit. Auf manchen Embedded-Systemen nicht vorhanden.

Seit Standard C99 kann man den Schlüsselwörtern ansehen, wieviele Bit sie breit sind. Sie müssen mit #include <inttypes.h> eingebunden werden.

Beginnen wir mit den vorzeichenbehafteten Typen.

Datentyp	kleinster Wert	größter Wert
Vorzeichenbehaftet		
int8_t	-128	127
int16_t	-32.768	32.767
int32_t	$-2 \cdot 10^{10}$	$2 \cdot 10^{10}$
int64_t	$-9 \cdot 10^{19}$	$9 \cdot 10^{19}$
Vorzeichenbehaftet, schnell		
int_fast8_t	-128	127
int_fast16_t	-32.768	32.767
int_fast32_t	$-2 \cdot 10^{10}$	$2 \cdot 10^{10}$
int_fast64_t	$-9 \cdot 10^{19}$	$9 \cdot 10^{19}$
Vorzeichenbehaftet, Mindestgröße		
int_least8_t	-128	127
int_least16_t	-32.768	32.767
int_least32_t	$-2 \cdot 10^{10}$	$2 \cdot 10^{10}$
int_least64_t	$-9 \cdot 10^{19}$	$9 \cdot 10^{19}$
Besondere		
intmax_t	größter vorzeichenbehaftete Integertyp	
intptr_t	Zeigertyp	

Die mit „schnell“ gekennzeichneten Datentypen dürfen größer sein als die geforderte Mindestgröße und sollen die schnellsten sein, welche mindestens die geforderte Bitanzahl haben.

Ähnliches gilt für die mit „Mindestgröße“ gekennzeichneten Datentypen. todo: mehr

Hier nun die vorzeichenlosen Typen.

Datentyp	kleinster Wert	größter Wert
Vorzeichenlos		
uint8_t	0	255
uint16_t	0	65.535
uint32_t	0	$4 \cdot 10^{10}$
uint64_t	0	$1.8 \cdot 10^{20}$
Mindestgröße, die schnell funktioniert		
uint_fast8_t	0	255
uint_fast16_t	0	65.535
uint_fast32_t	0	$4 \cdot 10^{10}$
uint_fast64_t	0	$1.8 \cdot 10^{20}$
Mindestgröße		
uint_least8_t	0	255
uint_least16_t	0	65.535
uint_least32_t	0	$4 \cdot 10^{10}$
uint_least64_t	0	$1.8 \cdot 10^{20}$
Besondere		
uintmax_t	größter vorzeichenloser Integertyp	
uintptr_t	Zeigertyp	

Hier gilt das selbe wie bei den Vorzeichenbehafteten Datentypen.

4.1.3 Muster für Konstanten

Man kann bei Konstanten einen Datentyp angeben. Wenn man im Quelltext konstante Ausdrücke verwendet, sollte man stets aufpassen, dass man einen sinnvollen Datentyp nutzt. Bei ganzzahliger Division ist $1/2 = 0$. Wir zeigen im nächsten Abschnitt Fließkommazahlen; bei deren Division ist $1.0/2 = 0.5$.

Datentyp	
unsigned char	—
char	'A'
unsigned short	—
short	—
unsigned int	1u, 1U
int	-1
unsigned long	123UL, 234ul
long	123L, 234l
unsigned long long	123ULL, 234ull
long long	123LL, 234ll

4.2 Fließkommazahlen

Gelegentlich benötigt man Zahlen mit Nachkommastellen. Bei Computern werden diese meist nach IEEE 754-Standard gespeichert. Dabei wird die Zahl aus zwei Komponenten

zusammengesetzt:

$$m \cdot 2^e$$

Vorfaktor m und Exponent e werden dabei getrennt gespeichert. Der Faktor m wird „Mantisse“ genannt. Bei negativem Exponenten werden bestimmte Rationale Zahlen (Bruchzahlen wie $\frac{3}{4}$) darstellbar.

Datentyp	Bits in Mantisse	Bits in Exponent
float	24	8
double	53	11
long double	64	16

Bitte beachten Sie, dass bei dieser Bauweise irgendwann das Addieren von 1 die Zahl nicht mehr ändert.

Einige Zahlen wie $\frac{1}{3}$, $\frac{2}{5}$, $\frac{1}{7}$ sind damit nicht darstellbar und werden nur näherungsweise abgespeichert.

4.2.1 Muster für Konstanten

Auch bei Fließkommazahlen kann man den Datentyp angeben:

Datentyp	
float	1.0f
double	1.0
long double	1.0L

4.3 Zeiger und Felder

4.3.1 Felder

Ein „Feld von Werten“ bedeutet, dass mehrere Werte des selben Datentypes hintereinander im Speicher abgelegt werden. Der folgende Quellcode zeigt Möglichkeiten, Felder zu deklarieren und zu definieren.

```
1 int main(void)
2 {
3   float feld1[5]; //uninitialisiertes Feld für 5
      Elemente allokieren
4   int feld2[] = {1, 2, 3, 4}; // 4-elemetniges
      Feld mit Werten initialisieren
5   return EXIT_SUCCESS;
6 }
```

Die erste Zeile deklariert ein Feld aus fünf float-Werten. Die zweite Zeile zeigt, wie ein Feld initialisiert und damit definiert wird.

Die Indexierung der Felder beginnt bei 0. Der folgende Quellcode zeigt, wie die Zahl 3 aus dem Feld in eine

Variable i kopiert wird.

```
1 int main(void)
2 {
3   int feld2[] = {1, 2, 3, 4};
4   int i = feld2[2];    // 3 wird kopiert.
5   return EXIT_SUCCESS;
6 }
```

Im Folgenden zeigen wir, wie der erste Wert (Index 0) des Feldes überschrieben wird:

```
1 int main(void)
2 {
3   int feld2[] = {1, 2, 3, 4};
4   feld2[0] = 5;    // 1 wird durch 5 überschrieben
      .
5   return EXIT_SUCCESS;
6 }
```

4.3.2 Zeiger

Ein „Zeiger“ ist ein Verweis auf eine Speicherstelle im Arbeitsspeicher. Er wird im Quelltext mit einem Sternchen nach dem Datentyp und vor dem Variablennamen kenntlich gemacht. Der folgende Quellcode zeigt, wie

1. ein Zeiger „zeiger“ deklariert wird,

2. ein Wert „wert“ definiert wird,

3. dem Zeiger mittels des Adressoperators & ein Ziel gegeben wird und

4. das Ziel mittels des Referenzoperators * ein neuer Wert gegeben wird.

```
1 int main(void)
2 {
3   int *zeiger;
4   int wert = 5;
5   zeiger = &wert;
6   *zeiger = 6;
7   return EXIT_SUCCESS;
8 }
```

Wenn man mehr Speicher benötigt als eine einzelne Variable, kann man mittels malloc() oder calloc() mehr Speicher reservieren. Das folgende Beispiel

1. erzeugt ein dynamisches Feld, in welches 1000 int-Werte hineinpassen.

2. reserviert genug Speicher für 100 double-Werte und

3. deallokiert das dynamische Feld.

Beachten Sie, dass feld2 nicht deallokiert wird. Das ist in vielen Fällen ein markanter Fehler. Das wird als „Speicherleck“ bezeichnet und kann den gesamten Arbeitsspeicher

aufbrauchen. Solche Fehler kann man mit dem Programm „Valgrind“ einfach finden.[3]

```
1 include <malloc.h>
2
3 int main(void)
4 {
5   int *dynamisches_feld = malloc(1000 * sizeof(
      int));
6   double *feld2 = calloc(sizeof(double), 100);
7   free(dynamisches_feld);
8   return EXIT_SUCCESS;
9 }
```

4.3.3 Zeigerarithmetik

```
1 #include <malloc.h>
2
3 int main(void)
4 {
5   char * ptr1 = malloc(1000);
6   int * ptr2 = malloc(1000);
7   long * ptr3 = malloc(1000);
8
9   printf("%p %p %p\n", ptr1, ptr2, ptr3);
10  ptr1++; ptr2++; ptr3++;
11  printf("%p %p %p\n", ptr1, ptr2, ptr3);
12  printf("%x %x %x\n", ptr1-ptr2, ptr2-ptr3, ptr3
      -ptr1 );
```

[3] Versuchen Sie es einmal mit diesem Beispiel!

```
13 }
```

4.3.4 Zeichen und Zeichenketten

Üblicherweise sind Zeichen 8 Bit breit und werden mit char deklariert. Auf UTF-8-Zeichen variabler Breite gehen wir in Kapitel 11.1 ein. Diese haben den Vorteil, dass sie den gesamten Unicode-Bereich abdecken. Das sind mehr als eine Million Zeichen.

8-Bit-Zeichen haben den Vorteil, dsss sie einfach zu verarbeiten sind. Viele Compiler nutzen den ASCII- oder den ANSI-Standard, um die Zahlenwerte Zeichen zuzuordnen. So ist 65 dort der dezimale Zahlenwert des Zeichens „A“.

Zeichenketten werden in C als *null-terminierte* char-Felder[4] implementiert und werden mit doppelten Anführungszeichen " markiert. Einzelne Zeichen stehen in einzelnen Anführungszeichen '.[5] Wie Sie in

```
1 char * zeichenkette = "Hallo Welt!";
2 char zeichen = 'H';
```

sehen können, ist zeichenkette ein Zeiger auf etwas. Konkret zeigt er auf die Stelle in der ausführbaren Datei, welche den Text „Hallo Welt!“ enthält.

[4] Das heißt, der Zeichenkette wird automatisch ein Null-Wert angehängt.

[5] Anfänger verwechseln diese gelegentlich mit den Akzent-Zeichen ' und `.

Zeichenketten werden auch als *String* bezeichnet. todo: visualisieren

4.4 logische Operationen

Wahrheitswerte werden in C mit „0“ und „ungleich 0“ dargestellt. 0 bedeutet Falsch, alles andere Wahr. Dies wird in Kapitel 5.7, „Kontrollstrukturen“ benötigt.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    if(!0)
6      printf("0 ist falsch.\n");
7    if(1)
8      printf("1 ist wahr.\n");
9    return 0;
10 }
```

Operator	Bedeutung
(a && b)	logisch „UND“; beide Operanden müssen wahr sein, um „wahr“ zurückzuliefern.
(a || b)	logisch „ODER“; einer der beiden Operanden muss wahr sein, um „wahr“ zurückzuliefern.
!a	logisch „NICHT“; der Operand muss falsch sein, um„wahr“ zurückzuliefern.
(a == b)	Prüft zwei Werte auf Gleichheit. WARNUNG! Verschiedene als wahr interpretierte Werte ergeben auch „falsch“!
(a != b)	Prüft zwei Werte auf Ungleichheit.

Verschiedene als wahr interpretierte Werte ergeben auch „falsch“, wenn man sie mittels == vergleicht!

Es sei erwähnt, dass bei den Auswertungen verschiedener logischer Operationen sicherheitshalber immer Klammern verwendet werden sollten. Mehr dazu im Kapitel 5.7.1, if()-else.

C nutzt „Lazy Evaluation“, um && und || auszuwerten. Das heißt, der nächste Operand wird nicht mehr ausgewertet, falls es am Ergebnis nichts mehr ändern würde.

4.5 binäre Operationen

Manche Operationen sind mit „Bitmasken“ einfacher zu implementieren. Diese kann man mittels binärer Operationen zusammenbauen:

- Wenn man eine 1 in der Bitmaske braucht, kann man sie mittels binär ODER (|) in die Bitmaske einfügen.
- Wenn man eine 0 in der Bitmaske braucht, kann man sie mittels binär UND (&) aus der Bitmaske löschen.

Operator	Bedeutung
(a & b)	binär „UND“; beide Bits müssen 1 sein, um 1 zurückzuliefern.
(a | b)	binär „ODER“; mindestens eines der beiden Bits muss 1 sein, um 1 zurückzuliefern.
~a	Negation der Bits (aus 0 wird 1, aus 1 wird 0).
(a == b)	binär „GLEICH“; beiden Operanden müssen gleich sein, um „wahr“ zurückzuliefern.
(a != b)	binär „UNGLEICH“; beide Operanden müssen ungleich sein, um „wahr“ zurückzuliefern.
(a ^ b)	binär „EXKLUSIV ODER“, bekannt als „XOR“; beide Bits müssen unterschiedlich sein, um 1 zurückzuliefern.
(a << b)	a um b Bits nach Links verschieben.
(a >> b)	a um b Bits nach Rechts verschieben.
a <<= b	a um b Bits nach Links verschieben und a zuweisen
a >>= b	a um b Bits nach Rechts verschieben und a zuweisen

Mit „XOR“ kann man eine einfache, symmetrische Verschlüsselung implementieren, denn es gilt ((a XOR b) XOR b) == a. Das heißt, wenn man zwei Mal mit dem selben

Wert XORt, kommt wieder der selbe Wert heraus. Manche Leute sehen das aber schon nicht mehr als Verschlüsselung an. Es gibt mit RSA, DSA und elliptischen Kurven sicherere Verfahren.

4.6 mathematische Operationen

Operator	Bedeutung
(a + b)	Summe aus a und b.
(a - b)	Differenz aus a und b.
-a	Vorzeichenwechsel.todo: geht das so überhaupt oder muss man * (-1) rechnen?
(a * b)	Produkt aus den Faktoren a und b.
(a / b)	Quotient aus a und b.
(a % b)	Rest der Division von a/b
a++	a inkrementieren. Also a += 1;
a–	a dekrementieren. Also a -= 1;
a += b	b zu a addieren und in a speichern.
a -= b	b von a abziehen und in a speichern.
a *= b	a mit b multiplizieren und in a speichern.

Bei der Division von zwei Ganzzahlen werden die Nachkommastellen weggelassen. So ist $1/2 = 0$. Andererseits ist $1.0/2 = 0.5$ denn der Datentyp eines Operanden ist fouble. Das Ergebnis wird auch double.

todo: irgendwo LValue erklären!

4.7 Gültigkeitsbereiche von Variablen

Der Gültigkeitsbereich einer Variable ist der Bereich, in dem die Variable wie gewünscht funktioniert. Er wird auch „Scope“ genannt.

4.7.1 Lokaler Gültigkeitsbereich

```
1  #include <stdio.h>
2
3  int * meinFehler()
4  {
5    int i = 0;
6    return &i;  //Schwerwiegender Fehler, i ist
       wieder weg, wenn die Funktion zurückspringt.
7  }
8
9  int main(void)
10 {
```

```
11   int *j = meinFehler();
12   printf("%d\n", *j);
13   return EXIT_FAILURE;
14 }
```

todo: Findet Valgrind solche Fehler?

Variablen sind von der Deklaration ab bekannt und sind gültig, bis der Codeblock verlassen wird, in welchem sie deklariert wurden.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   {
6     int i = 1;
7     printf("%d\n", i);
8   }
9   {
10    int i = 2;
11    printf("%d\n", i);
12  }
13  return EXIT_SUCCESS;
14 }
```

Man kann unauffällige Fehler machen, wenn man Variablennamen in untergeordneten Codeblöcken nochmal deklariert:

```
1 #include <stdio.h>
2
3 int main(void)
```

```
4  {
5    int i = 100;
6    {
7      int i = 1;
8      printf("%d\n", i); // gibt 1 aus.
9    }
10   return EXIT_SUCCESS;
11 }
```

4.7.2 Globaler Gültigkeitsbereich

Möchte man Variablen in verschiedenen Funktionen zur Verfügung haben, kann man sie im „globalen Scope“ deklarieren oder auch definieren.

```
1 #include <stdio.h>
2
3 double pi = 3.1415926;
4
5 int main(void)
6 {
7   printf("Pi = %f", pi);
8   return EXIT_SUCCESS;
9 }
```

4.8 Zusammenfassung

In diesem Kapitel wurden grundlegende Sprachelemente besprochen. Wir wissen jetzt, was ein „Integer“ ist, ken-

nen Fließkommazahlen und können logische und bitweise Operationen damit dürchführen. Zeiger, Zeichenketten und Felder stellen auch kein Mysterium mehr für uns dar und wir können mit mathematischen Funktionen umgehen.

Ich erinnere daran, dass `1 / 2` wegen ganzzahliger Division Null ergibt. `3 / 2` ergibt 1. Es wird also bei Ganzzahl-Division immer abgerundet.

4.9 Visualisierungs-Übung zu Zeigern

Viele Leute haben Probleme, sich Zeiger richtig vorzustellen. Dazu stelle ich Ihnen hier eine kleine Übung vor, welche Sie machen können, um plastisches Verständnis dafür zu bekommen: Sie brauchen

- 20cm bis 30cm Schnur und
- drei Karteikarten

Legen Sie die drei Karteikarten auf den Tisch. Das sind die Speicherzellen, welche die Varialben enthalten. Wählen Sie eine beliebeige Karte aus und legen Sie die Schnur darauf. Das ist die Variable, welche den Zeiger enthält.

Verbinden Sie nun die Schnur mit einer weiteren Karteikarte. Der Zeiger zeigt nun auf eine Speicherstelle. Setzen wir nun gedanklich den Zeiger neu: Nehmen Sie die Schnur und verbinden Sie die erste Karteikarte mit der bisher unbenutzten. Der Zeiger zeigt nun auf ein anderes Speicherelement.

Ich habe diese Übung mal mit Mitschülern gemacht, als ich am Gymnasium war. Manchen ging dabei ein Licht auf, aber nicht jedem. Viel Erfolg!

Kapitel 5

Weiterführende Sprachelemente

5.1 Konstanten (const)

Wenn man nicht möchte, dass eine Variable überschrieben wird, kann man sie als const definieren. Das Schlüsselwort wird der Deklaration vorangestellt. Dies kann der Compiler an manchen Stellen für Optimierungen nutzen. Es schützt auch vor manschen Programmierfehlern.

```
1 const double pi = 3.1415926;
2 const int Raeder_am_Auto = 4;
```

5.2 Datentypen von Ausdrücken

Ein Ausdruck wie

```
1 pi * Raeder_am_auto
```

hat auch einen Datentyp. Es wird immer derjenige Datentyp verwendet, der im Ausdruck den größten Wertebereich hat. In diesem Fall hat der Ausdruck den Datentyp „double“.

todo: L-Value und R-Value erklären!

5.3 Funktionen

Man kann sich Tipparbeit und Wartungsaufwand sparen, wenn man häufig gebrauchte Codeblöcke in Funktionen auslagert. Funktionen können einen Rückgabewert haben. Dessen Typ wird dem Funktionsnamen vorangestellt. Ist kein Rückgabewert erforderlich, kann man die Funktion als void deklarieren.

Wenn eine Funktion aus Daten etwas machen soll, dann kann man sie als Parameter übergeben. Diese stehen in Klammern. Klammern sind bei Funktionen verpflichtend, man kann sie aber leer lassen.

Funktionen können einen Wert zurückgeben; der Datentyp wird dafür der Funktion vorangestellt. Benötigt man keinen Rückgabewert, kann man eine Fuinktion mit dem

Typ void deklarieren. Rückgabewerte muss man nicht auswerten. Wenn die Funktion einen Fehlerstatus zurückgibt, sollte man ihn inner auswerten.

```
1  #include <stdio.h>
2
3  char * programmname = "Fehler-Demo";
4
5  void fehlermeldung(const char * meldung)
6  {
7    fprintf(stderr, "Fehler in %s: %s\n",
       programmname, meldung);
8  }
9
10 int gib5()
11 {
12   return 5;
13 }
14
15 int main(void)
16 {
17   gib5(); // Man darf Rückgabewerte ignorieren.
       Meist ist das eine schlechte Idee.
18   felermeldung("return vergesssen");
19   // return;
20 }
```

5.4 Funktionszeiger

Funktionszeiger sind unter Anderem nützlich, um Programme zu modularisieren. Mit ihnen kann man während der Laufzeit des Programmes das Verhalten von Funktionen ändern.

```
1  void f1a(int arg1)
2  {
3    printf("a: %d", arg1);
4  }
5
6  void f1b(int arg1)
7  {
8    printf("b: %d", arg1);
9  }
10
11 void (*fn1)(int arg1);
12
13 int main(void)
14 {
15   fn1 = &f1a;
16   fn1(1);
17   fn1 = &f1b;
18   fn1(1);
19 }
```

5.5 Enumerationen

Enumerationen helfen, den Überblick über die Bedeutung von Konstanten zu behalten. Im folgenden Beispiel wird eine Enumeration definiert und eine Variable my_open_enum definiert, die einen solchen Enum hält.

Die geschweiften Klammern werden mit einem Semikolon abgeschlossen! Es gibt Compiler, welche sehr ungewöhnliche Fehlermeldungen ausgeben, wenn solch ein Semikolon fehlt.

```
1 enum open_enum {
2   O_INVALID,   // automatisch 0
3   O_READ,      // automatisch 1
4   O_WRITE,     // automatisch 2
5   O_RDWR       // automatisch 3
6 }; //Semikolon!
7
8 enum open_enum my_open_enum = O_INVALID;
```

Das folgende Beispiel zeigt, wie man den einzelnen Konstanten Werte zuweist. Es wird bei der auch gleich eine Variable namens mode deklariert, welche eine Enumeration des definierten Types enthält.

```
1 enum drive_mode {
2   DM_NEUTRAL = 0,
```

```
3   DM_VORWAERTS = 2,
4   DM_RUECKWAERTS = 6,
5   DM_PARKEN = 125
6 } mode;
```

Bitte beachten Sie, dass die Variable noch nicht definiert wurde. Sie kann einen beliebigen Wert haben; auch außerhalb des Wertebereiches.

5.6 Strukturen

Strukturen werden mit dem Schlüsselwort `struct` deklariert und sind sehr nützlich um den Überblick im Quellcode zu behalten. Mit ihrer Hilfe kann man Daten strukturieren und leichter lesbar bekommen.

Die geschweiften Klammern werden mit einem Semikolon abgeschlossen! Es gibt Compiler, welche sehr ungewöhnliche Fehlermeldungen ausgeben, wenn solch ein Semikolon fehlt.

5.6.1 Forward Deklarationen

Um auf eine Struktur innerhalb einer Struktur mittels eines Zeigers verweisen zu können, muss der Name der Struktur vorher bekannt sein. Im folgenden Beispiel machen wir das

mit der Struktur „Auto“, um eine einfach verkettete Liste zu implementieren. Dies wird im Kapitel 9.1 genauer erklärt.

```
1  struct Auto;
2
3  struct Auto
4  {
5    Auto * naechstes;
6    int ps;
7    int reifen;
8    void (*fahre_los)(struct Auto * auto);
9  };  // Semikolon!!!
10
11 void fahre1(struct Auto * auto)
12 {
13   printf("%d PS geben gas.", auto->ps);
14 }
15
16 void fahre2(struct Auto * auto)
17 {
18   printf("%d Räder setzen sich in Bewegung!",
       auto->reifen);
19 }
20
21 int main(void)
22 {
23   struct Auto auto1
24   {
25     .ps = 100,
26     .reifen = 4,
27     .fahre_los = &fahre1;
```

```
28    };
29
30    auto1.fahre_los(&auto1);
31    // Gibt "100 PS geben gas." aus.
32
33    struct Auto *auto2 = malloc(sizeof(Auto));
34    auto1->naechstes = &auto2;
35    auto2->naechstes = NULL;
36    auto2->ps = 500;
37    auto2->reifen = 6;
38    auto2->fahre_los = &fahre2;
39
40    auto2->fahre_los(auto2);
41    // Gibt "6 Reifen setzen sich in Bewegung!" aus
        .
42 }
```

5.7 Kontrollstrukturen

Es reicht normalerweise nicht, Variablen zu deklarieren, meist muss man auch Dinge wiederholen oder nur bedingt machen. Dafür gibt es Schleifen oder if()-Blöcke.

5.7.1 if()-else

Um einen Codeblock nur bedingt auszuführen, gibt es das Schlüsselwort `if`. Das folgende Beispiel zeigt, wie man es benutzt:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int test = 0;
6    if(0 == test)
7    {
8      printf("test == 0");
9    }
10   else if(0 != test)
11   {
12     printf("test != 0");
13   }
14   else
15   {
16     printf("Unmöglicher Fall!");
17   }
18 }
```

Das Programm sollte `test == 0` ausgeben, da wir die Variable `test` mit 0 initialisieren und anschließend prüfen, ob diese gleich 0 ist.

5.7.2 for(), while(), do while()

Um Codeblöcke wiederholt auszuführen, kann man die Schlüsselworte `for`, `while` und `do while` nutzen.

Das Schlüsselwort `for` muss mit drei Ausdrücken gefüttert werden:

1. Einen Startwert, im Beispiel `int = 0`,

2. die Bedingung, weche wahr sein muss, um den nächsten Schritt auszuführen, im Beispiel `i < 10` und

3. der Schritt, welcher bei jedem Schleifendurchlauf ausgeführt wird; im Beispiel `i++`.

```c
#include <stdio.h>

int main(void)
{
  for(int i = 0; i < 10; i++)
  {
    printf("%d\t", i);
  }
  printf("\n");

  int i = 0;
  while(i < 10)
  {
    printf("%d\t", i);
    i++;
  }
  printf("\n");

  i = 0;
  do
  {
    printf("%d\t", i);
    i++;
  }
  while(i < 10);
```

```
26   printf("\n");
27
28   return EXIT_SUCCESS;
29 }
```

Die while-Schleife hat nur die Bedingung in der Klammer, während welcher der Block wiederholt wird. do while ist sehr ähnlich, die Bedingung wird nur zum Ende des Blockes überprüft. Sie wird also mindestens einmal ausgeführt.

5.7.3 switch() & case

Wenn man nicht mehrere if-Blöcke haben will, kann man mittels switch(variable) mehrere Fälle übersichtlich behandeln.

```
1 #include <stdio.h>
2
3 void print_hallo_welt()
4 {
5   printf("Hall Welt!\n");
6 }
7
8 void print99()
9 {
10   for(int i = 99; i > 0 ; i--)
11   {
12     printf("%d Flaschen Apfelsaft stehen auf dem
     Tresen.\n");
```

```
13    }
14  }
15
16  int main(int argc, char * argv[])
17  {
18    // Wir teilen dem Compiler mit, dass wir Platz
19    // für eine Variable brauchen.
20    int option;
21    do
22    {
23      printf("Bitte wählen Sie aus:\n");
24      printf("1. \"Hallo Welt!\" ausgeben\n");
25      printf("2. \"99 Flaschen...\" ausgeben\n");
26      printf("3. Programm beenden\n");
27
28      // Gewählte Option von Tastatur einlesen.
29      option = getchar();
30
31      switch(option)
32      {
33      case '1':
34        print_hallo_welt();
35        break;
36      case '2':
37        print99();
38        break;
39      case '3'
40        printf("Programm wird beendet");
41        break;
42      default:
43        printf("Ungültige Eingabe!");
```

```
44          break;
45        }
46      }
47      while('3' != option)
48      return 0;
49  }
```

5.8 Zusammenfassung

In diesem Kapitel haben wir uns mit const-Konstanten beschäftigt, haben festgestellt, dass Ausdrücke wie 1/2.0f auch einen Datentyp haben und wir haben gelernt, Funktionen zu deklarieren und zu definieren. Wir haben auch Funktionszeiger gesehen und können Aufzählungen (Enumerationen) und Strukturen definieren. Mittels Kontrollstrukturen wie if(), for(;;), while() und switch() können wir den Programmfluss steuern.

Kapitel 6

Standardfunktionen

6.1 printf(), fprintf(), snprintf() und Verwandte

Es gibt verschiedene Varianten von Befehlen, um eine Zeichenkette auszugeben oder zusammenzubauen.

- printf(char *format, ...) Gibt aus den erzeugten String auf der Standard-Ausgabe aus.
- fprintf(FILE *stream, char *format, ...) Gibt den erzeugten String in dem Stream stream wieder.
- sprintf(char *puffer, char *format, ...) Legt die erzeugte

Zeichenkette in dem Puffer puffer ab. Es erfolgt keine Längenbegrenzung!

- snprintf(char *puffer, size_t n, char *format, ...) erzeugt einen höchstens n Zeichen langen String und legt ihn in puffer ab.

Die Funktionen

- vprintf(char *format, va_list arg),
- vfprintf(FILE *f, char *format, va_list arg),
- vsprintf(char *puffer, char *format, va_list arg),
- vsnprintf(char *puffer, size_t n, char *format, va_list arg)

sind Varianten der obigen Befehle, welche eine va_list aus stdarg.h als Argument nutzen – für Details verweise ich auf viel Dokumentation im Internet.

sprintf() und vsprintf() geben keine Garantie, dass der erzeugte String in den Puffer passt!

Auf die Nutzung von FILE *-Variablen gehen wir später ein. Es handelt sich dabei um Zeiger auf Strukturen, welche den Zustand einer

- Datei,
- eines Eingabekanals oder
- eines Ausgabekanals

beschreiben. Zwei Standard-Ausgabekanäle des Type FILE * sind stdout, die Standardausgabe und stderr, der Standard-Fehlerausgabe. Die Standardeingabe nent sich stdin.

6.1.1 Formatangaben

Die Formatangabe in printf() und seinen Varianten sieht wie folgt aus:

%[flags][breite][. präzision][länge] spezifizierer

Der Spezifizierer gibt dabei grob an, welcher Datentyp in welcher Weise ausgegeben werden soll. Die folgende Tabelle gibt Aufschluss über die möglichen Angaben. flags , breite, präzision und länge sind dabei optional, was hier durch die rechteckigen Klammern angedeutet werden soll.

d	vorzeichenbehafteter Integer	-123
i	vorzeichenbehafteter Integer	-234
u	vorzeichenloser Integer	123
o	vorzeichenlse Oktalzahl	750
x	vorzeichenlose Hexadezimalzahl	f82
X	vorzeichenlose Hexadezimalzahl in Großschreibweise	F82
f	dezimale Floatingpoint-Schreibweise (klein)	16.25
F	dezimale Floatingpoint-Schreibweise (groß)	16.25
e	wissenschaftliche Exponentialdarstellung (Mantisse, Exponent)	3.1212e-2
E	wissenschaftliche Exponentialdarstellung (Mantisse, Exponent)	3.1212E-2
g	kürzeste Darstellung von %f und %e	1.2345e-16
G	kürzeste Dasstellung von %F und %E	1.2345E-16

a	hexadezimale Floatingpoint-Darstellung (klein)	-0xc.afeep-32
A	hexadezimale Floatingpoint-Darstellung (groß)	-0XC.AFEEP32
c	einzelnes Zeichen	r
s	null-terminirter String (Zeichenkette)	test
p	Zeiger auf eine Adresse	0000a000
n	Siehe ausführliche Beschreibung	
%	Zwei % hintereinander geben ein einzelnes % auf dem Stream aus	%

todo: n beschreiben

Bei den Floatingpoint-Schreibweisen kann es passieren, dass die Locale[1] aus dem Punkt ein Komma macht!

todo: Prüfen, ob das stimmt!

Folgende Flags sind bei im Format-String definiert:

[1] Gebietseinstellung

-	Ausgabe linksbündig machen (rechtsbündig ist der Standard).
+	erzwingt die Ausgabe eines Vorzeichens (Plus oder Minus). Standardmäßig werden nur negative Vorzeichen angezeigt.
(Leerzeichen)	Falls kein Vorzeichen ausgegebenw erden sollte, wird ein Leerzeichen vorneangestellt.
#	Falls man dies einem o, x oder X voranstellt, wird der Ausgabe ein 0x oder 0X vorangestellt. Falls man a, A, e, E, f, F, g oder G nutzt, wird ein Dezimalpunkt genutzt, egal ob er gebraucht wird, oder nicht.
0	Füllt die Breite von Links aus mit Nullen (0) auf.

Für die Breite kann man eine Zahl oder ein Sternchen (∗) angeben. Das Sternchen bedeutet, dass nicht in dem Format-String sondern in der Argumentenliste (vor der auszugebenden Zahl) eine Integer-Zahl übergeben wird, welche spezifiziert, wie viele Stellen mindestens ausgegeben werden sollen. Analog gilt es für eine Zahl, welche in dem Format-Ausdruck stehen darf: Das Resultat mit Leerzeichen von links aufgefüllt, bis die gewünschte Breite erreicht ist. Ist das Resultat länger, wird die das Resultat

nicht abgeschnitten.

Bei der . precision-Angabe wird bestimmt, wie viele Stellen hinter dem Komma ausgegeben werden sollen. Bei dem Spezfizierer s wird damit die Länge des Strings begrenzt. Ansonsten wird bis zum Null-Byte geschrieben.

Der Längen-Spezifizierter gibt an, in welchem Format der auszugebende Wert im Speicher abgelegt ist:

	Spezifizierer	
Länge	d, i	u, o , x, X
(keiner)	int	unsigned int
hh	signed char	unsigned char
h	short int	unsigned short int
l	long int	unsigned long int
ll	long long int	unsigned long long int
j	intmax_t	uintmax_t
z	size_t	size_t
t	ptrdiff_t	ptrdiff_t
L		

	Spezifizierer				
Länge	f, F, e, E, g, G, a, A	c	s	p	n
(keiner)	double	int	char *	void *	int *
hh					signed char *
h					short int *
l		wint_t	wchar_t *		long int *
ll					long long int *
j					intmax_t
z					size_t *
t					ptrdiff_t *
L	long double				

6.2 scanf() und seine Varianten

Die Funktion scanf()u und seine varianten stelle ich hier nur vor, um alten Code verständlich zu machen, Vermeiden Sie bitte die Nutzung dieses Befehls, da er unsicher ist.

scanf() kann zu Pufferüberläufen führen, wodurch fremder Schadcode ausgeführt werden kann!

Mit fread() und atoi(), atol(), atof()...kann man bessere Ergebnisse erzielen. Dazu später mehr

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int my_int;
6    char buf[20];
7    const char *data = "10 1.3e-1 test";
8    double my_double;
9
10   sscanf(data, "%i %f %20s", &my_int, &my_double,
       buf);
11
12   printf("buf: %s, my_double: %f, my_int: %i",
       buf, my_double, my_int);
```

```
13
14   return EXIT_SUCCESS;
15 }
```

- scanf(char *format, ...) liest einen String von der Standardeingabe und extrahiert die in dem format-String angegebenen Daten.
- sscanf(const char *s, char *format, ...) extrahiert aus dem String s die Daten nach dem in format angegebenen Muster.
- fscanf(FILE *f, char *format, ...) liest aus der in f angegebenen Datei nach dem in format angegebenen Muster.
- vscanf(char *format, va_lsit), vsscanf(char *format, va_list) und vfscanf(FILE *f, char *format, va_list) sind die va_list -Varianten der oben angegebenen Befehle. Ich verweise hierzu auf Dokumentation aus dem Internet zur Nutzung von va_list aus stdarg.h.

Der Formatstring darf aus Formatierungsbefehlen, Leerzeichen und Zeichen bestehen. Lautet der String „%d ist % f“, so wird eine dezimale Ganzzahl, ein Leerzeichen, die Zeichenkette „ist“ und eine Floatingpoint-Zahl eingelesen.

Wird die genannte Zeichenkette nicht eingeben, schlägt die Funktion fehl. [2]

Der Formatstring wird wie folgt gebildet: %[∗][breite][][länge] spezifizierer .

breite und länge sind optionale Parameter, genauso wie das Sternchen hinter dem Prozentzeichen.

[2]Ich bezweifle, dass alle Implementierungen von scanf() richtig funktionieren. Mir sind schon welche untergekommen, die einfach nicht terminierten.

i	Beliebige Anzahl von Ziffern, optional mit vorangestelltem Vorzeichen. Grundsätzlich werden DEzimalziffern erwartet, eine vorangestellte 0 ergibt Oktalwerte, ein vorangestelltes 0x hexadezimalwerte.
d, u	Dezimaler Integer. d für vorzeichenbehaftete, u für vorzeichenlose Zahlen.
o	oktaler Integer.
x	hexadezimaler Integer.
f, e, g, a	Floatingpoint-Zahl im Format 1.3e27 oder 2.555E-1.
c	Einzelnes Zeichen.
s	Zeichenkette.
p	Zeigeradresse
n	Anzahl der bisher gelesenen Zeichen.
%	%.
[chars], [ĉhars]	TODO

Ein Sternchen in dem Formatspezifiziere gibt an, dass die gelesenen Zeichen nicht in ein Argument gespeichert werden. breite gibt die höchste Zahl von Zeichen an, welche eingelesen werden sollen. länge ist wieder hh, h, l, ll, j, z, t, oder L.

todo: richtig anpassen

	Spezifizierer	
Länge	d, i	u, o , x, X
(keiner)	int *	unsigned int *
hh	signed char *	unsigned char +
h	short int *	unsigned short int *
l	long int *	unsigned long int *
ll	long long int *	unsigned long long int *
j	intmax_t *	uintmax_t *
z	size_t *	size_t *
t	ptrdiff_t *	ptrdiff_t *
L		

todo: richtig anpassen

	Spezifizierer			
Länge	f, F, e, E, g, G, a, A	c	s	p
(keiner)	double *	int *	char *	void *
hh				
h				
l		wint_t *	wchar_t *	
ll				
j				
z				
t				
L	long double *			

Hier ein Beispiel, wie man mit scanf() einen Fehler

macht, indem man bei der Puffer-Allokation eine andere Größe nimmt als bei dem scanf()-Format-Parameter \%s.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    char *buff[20];
6    int i;
7    char null1 = 0; // Schutzmagie, damit der
       String endet.
8    char null2 = 0;
9    char null3 = 0;
10   char null4 = 0;
11
12   printf("Machen Sie das Programm kaputt!");
13   scanf("%d %25s", &i, buff);
14
15   printf("%i, %25s", i, buff);
16 }
```

scanf() kann zu Pufferüberläufen führen, wodurch fremder Schadcode ausgeführt werden kann!

6.3 String-Befehle

6.3.1 strlen()

6.3.2 strncat()

6.3.3 strstr()

6.4 Kopierbefehle

memcpy() benutzen, wenn möglich (benutzt optimierte Befehle).

6.5 Dateibefehle

Wir hatten im Kapitel 6.1, „printf(), fprintf(), snprintf() und Verwandte“ bereits mit der Struktur FILE zu tun; es werden für gewöhnlich Zeiger auf solche Strukturen verwendet, um mit Dateien zu hantieren.

Die übliche Herangehensweise, eine Datei zu nutzen ist, sie mittels

- fopen() zu öffnen,
- fread() zu lesen,
- fwrite() zu schreiben und

- fclose () wieder zu schließen.

Zur Navigation in den Dateien werden ftell () und fseek () genutzt. fflush () leert die Zwischenspeicher auf Festplatte.

6.5.1 fopen(), fclose()

6.5.2 fread(), fwrite()

6.5.3 ftell(), fseek()

6.5.4 fflush()

Wenn man die Daten auf dem Datenträger ablegen will, muss man die Dateisystem-Zwischenspeicher „flushen“, also auf Platte „spülen“. Das kann auch bei Ausgaben auf stdout Sinn ergeben, wenn man bemerkt, dass man lange Zeit keine Ausgabe bekommt, obwohl die Anweisung dafür gegeben wurde. Manche Betriebssysteme stellen das Zwischenspeicher so ein, dass die zwischengespeicherten Ausgaben erst angezeigt werden, wenn ein Zeilenvorschub „\n“ ausgegeben wurde.

Kapitel 7

Programmparameter

Parameter werden dem Programm als Feld von Zeigern übergeben, welche auf nullterminiterte Zeichenketten zeigen. Die Anzehl der Felder wird in argc übergeben, das Feld der Zeiger steht in argv[].

Hier sehen Sie ein Programm, welches die übergebenen Parameter durch ein Leerzeichen getrennt ausgibt.

```
1 int main(int argc, char *argv[])
2 {
3   for(int i = 0; i < argc; i++)
4   {
5     printf("%s ", argv[i]);
6   }
7 }
```

Kapitel 8

Modularisierung

Kapitel 9

Beispiele

9.1 Einfach verkettete Liste

todo: Einfach verkettete Liste

9.2 Überblick über weitere Algorithmen

todo: AVL-Baum, RB-Baum, B*-Baum erwähnen todo: Grafische Algorithmen erwähnen: Sweep, todo, welche todo: Rucksack-Problem todo: Programmierparadigmen erwähnen: Dynamische Programmierung, Divide & Conquer todo: ggf. hier ausmisten

Kapitel 10

Gefahren

10.1 Technische Grundlagen

todo: Stack und Heap erklären.

10.2 Pufferüberlauf

todo: Globale Puffer-Variablen verdammen. Funktionspointer nach Puffern sind gefährlich; zB Rücksprungadresse bei Aufruf von scanf()

Kapitel 11

Fortgeschrittenes

11.1 UTF-8-Zeichen

11.2 Threads

11.2.1 Funktionen

11.2.2 volatile

Nur benutzen, wenn nur ein einzelner, schreibender Thread exisitiert.

11.2.3 restricted

11.2.4 Atomizität

Bei mehreren schreibenden Threads benutzen. Ist langsamer als "volatile"

```
1  #include <stdio.h>
2  #include <threads.h>
3  #include <stdatomic.h>
4  #include <stdlib.h>
5
6  int a = 0;
7  volatile int b = 0;
8  atomic_int c = 0;
9
10 int max = 1000000;
11
12 int thread1(void* dummy)
13 {
14   a=0;
15   while(a < max)
16   {
17     a++;
18   }
19   return 0;
20 }
21
22 int thread2(void* dummy)
23 {
24   b=0;
25   while(b < max)
```

```
26    {
27      b++;
28    }
29    return 0;
30  }
31
32  int thread3(void* dummy)
33  {
34    c= 0;
35    while(c < max)
36    {
37      c++;
38    }
39    return 0;
40  }
41
42  int thread4(void* dummy)
43  {
44    do
45    {
46      printf("%d, %d, %d\n", a, b, c);
47    } while(a < max && b < max && c < max);
48    return 0;
49  }
50
51  int main(void)
52  {
53    thrd_t
54      t1,
55      t2,
56      t3,
```

```
57      t4
58      ;
59    if(thrd_create(&t4, &thread4, NULL)
60      || thrd_create(&t3, &thread3, NULL)
61      || thrd_create(&t2, &thread2, NULL)
62      || thrd_create(&t1, &thread1, NULL))
63    {
64      printf("threads failed");
65      return EXIT_FAILURE;
66    }
67    thrd_join(t1, NULL);
68    thrd_join(t2, NULL);
69    thrd_join(t3, NULL);
70    thrd_join(t4, NULL);
71    return EXIT_SUCCESS;
72  }
```