Kapitel 7

Datentypen

7.1 Überblick

Datentypen legen

- den *Speicherbedarf*,
- die *Interpretation* des Speicherplatzes sowie
- die *erlaubten Operationen* fest.

Abbildung 7.1 gibt einen Überblick über die verschiedenen Datentypen in C. Die Nähe der Zeiger zu den Vektoren ist dabei kein Zufall, da in C Vektoren als Zeiger betrachtet werden können und umgekehrt.

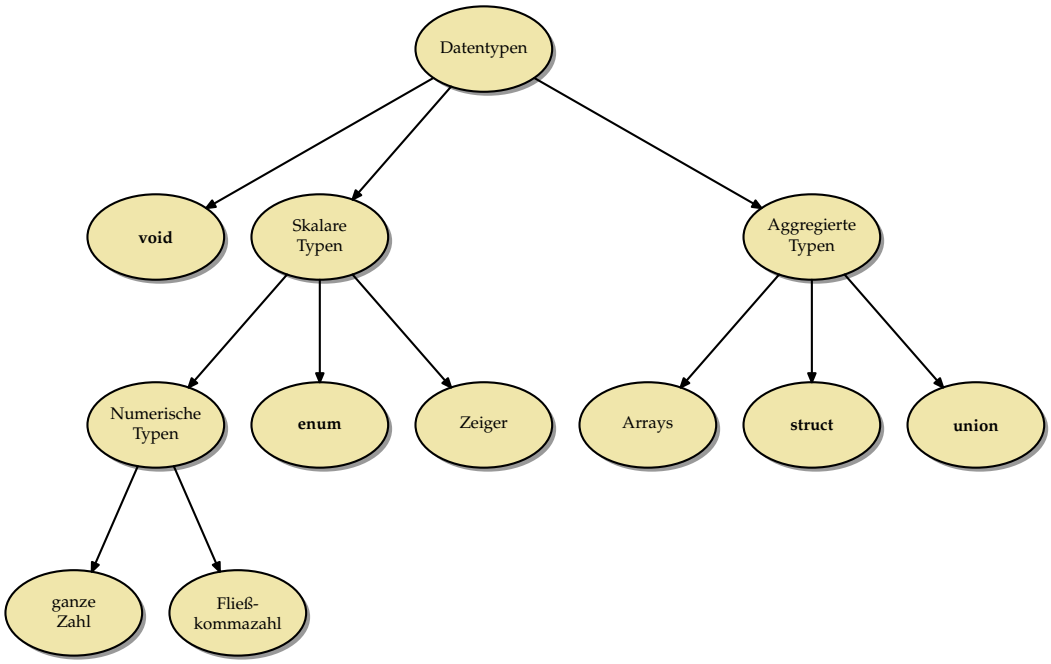

Abbildung 7.1: Datentypen – Eine Übersicht

Aus grammatikalischer Sicht vermischen sich Deklarationen mit Typ-Spezifikationen. So lässt sich ein Zeigertyp nur im Rahmen einer Deklaration konstruieren, jedoch nicht innerhalb einer Typ-Spezifikation:

⟨type-specifier⟩ ⟶ ⟨void-type-specifier⟩
⟶ ⟨integer-type-specifier⟩
⟶ ⟨floating-point-type-specifier⟩
⟶ ⟨enumeration-type-specifier⟩
⟶ ⟨structure-type-specifier⟩
⟶ ⟨union-type-specifier⟩
⟶ ⟨typedef-name⟩

7.2 Skalare Datentypen

Zu den skalaren Datentypen gehören alle Typen, die entweder numerisch sind oder sich zu einem numerischen Typ konvertieren lassen. Ein Wert eines skalaren Datentyps kann beispielsweise ohne weitere Konvertierung in einer Bedingung verwendet werden. Entsprechend wird die 0 im entsprechenden Kontext auch als Null-Zeiger (in Java: **null**) interpretiert oder umgekehrt wird ein Null-Zeiger als äquivalent zu *false* betrachtet und entsprechend ein Nicht-Null-Zeiger innerhalb einer Bedingung als *true* interpretiert. Ferner liegt die Nähe zwischen Zeigern und ganzen Zahlen auch in der von C unterstützten Adressarithmetik begründet.

7.2.1 Ganzzahlige Datentypen

⟨integer-type-specifier⟩ ⟶ ⟨signed-type-specifier⟩
⟶ ⟨unsigned-type-specifier⟩
⟶ ⟨character-type-specifier⟩
⟶ ⟨bool-type-specifier⟩
⟨signed-type-specifier⟩ ⟶ [**signed**] **short** [**int**]
⟶ [**signed**] **int**
⟶ [**signed**] **long** [**int**]
⟶ [**signed**] **long long** [**int**]
⟨unsigned-type-specifier⟩ ⟶ **unsigned short** [**int**]
⟶ **unsigned** [**int**]
⟶ **unsigned long** [**int**]
⟶ **unsigned long long** [**int**]
⟨character-type-specifier⟩ ⟶ **char**
⟶ **signed char**
⟶ **unsigned char**
⟨bool-type-specifier⟩ ⟶ **_Bool**

Die Spezifikation eines ganzzahligen Datentyps besteht aus einem oder mehreren Schlüsselworten, die die Größe festlegen, und dem optionalen Hinweis, ob der Datentyp vorzeichenbehaftet ist oder nicht. Fehlt die Angabe von **signed** oder **unsigned**, so wird grundsätzlich **signed** angenommen. Die einzigen Ausnahme hiervon sind **char** und **_Bool**. Bei **char** darf der Übersetzer selbst eine Voreinstellung treffen, die sich daran orientiert, was sich am effizientesten auf der Zielarchitektur umsetzen lässt.

Auch wenn Angaben wie **short** oder **long** auf eine gewisse Größe hindeuten, so legt keiner der C-Standards die damit verbundenen tatsächlichen Größen fest. Stattdessen gelten nur folgende Regeln:

- Der jeweilige „größere“ Datentyp in der Reihe **char**, **short**, **int**, **long**, **long long** umfasst den Wertebereich der kleineren Datentypen, d.h. **char** ist nicht größer als **short**, **short** nicht größer als **int** usw.
- Für jeden der ganzzahligen Datentypen gibt es Mindestintervalle, die abgedeckt sein müssen. (Die zugehörige Übersichtstabelle folgt auf Seite 54.)
- Die korrespondierenden Datentypen mit und ohne Vorzeichen (etwa **signed int** und **unsigned int**) belegen exakt den gleichen Speicherplatz und verwenden die gleiche Zahl von Bits. (Entsprechende Konvertierungen erfolgen entsprechend der Semantik des Zweier-Komplements.)

In C werden alle ganzzahligen Datentypen durch Bitfolgen fester Länge repräsentiert: $\{a_i\}_{i=1}^{n}$ mit $a_i \in \{0, 1\}$. Bei ganzzahligen Datentypen ohne Vorzeichen ergibt sich der Wert direkt aus der binären Darstellung:

$$a = \sum_{i=1}^{n} a_i 2^{i-1}$$

Daraus folgt, dass der Wertebereich bei n Bits im Bereich von 0 bis $2^n - 1$ liegt. Bei ganzzahligen Datentypen mit Vorzeichen übernimmt a_n die Rolle des Vorzeichenbits. Für die Repräsentierung gibt es bei C99 nur drei zugelassene Varianten:

- **Zweier-Komplement**:

  $$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n 2^{n-1}$$

  Wertebereich: $\left[-2^{n-1}, 2^{n-1} - 1\right]$
  Diese Darstellung hat sich durchgesetzt und wird von fast allen Prozessor-Architekturen unterstützt.

- **Einer-Komplement**:

  $$a = \sum_{i=1}^{n-1} a_i 2^{i-1} - a_n \left(2^{n-1} - 1\right)$$

  Wertebereich: $\left[-2^{n-1} + 1, 2^{n-1} - 1\right]$
  Vorsicht: Es gibt zwei Repräsentierungen für die Null. Es gilt: $-a == \sim a$
  Diese Darstellung gibt es auf einigen historischen Architekturen wie etwa der PDP-1, der UNIVAC 1100/2200 oder der 6502-Architektur.

- **Trennung zwischen Vorzeichen und Betrag**:

$$a = (-1)^{a_n} \sum_{i=1}^{n-1} a_i 2^{i-1}$$

Wertebereich: $\left[-2^{n-1}+1, 2^{n-1}-1\right]$
Vorsicht: Es gibt zwei Repräsentierungen für die Null.
Diese Darstellung wird ebenfalls nur von historischen Architekturen verwendet wie etwa der IBM 7090.

Der ISO-Standard sieht für die einzelnen ganzzahligen Datentypen Intervalle vor, die in jedem Falle abgedeckt werden müssen. Den einzelnen Implementierungen steht es frei, diese Bereiche zu vergrößern. Die Konstanten, die den tatsächlich zur Verfügung stehenden Bereich spezifizieren, finden sich in <*limits.h*>:

Datentyp	Bits	Intervall	Konstanten
signed char	8	$[-127, 127]$	*SCHAR_MIN*, *SCHAR_MAX*
unsigned char	8	$[0, 255]$	*UCHAR_MAX*
char	8		*CHAR_MIN*, *CHAR_MAX* – entspricht entweder **signed char** oder **unsigned char**
short	16	$[-32767, 32767]$	*SHRT_MIN*, *SHRT_MAX*
unsigned short	16	$[0, 65535]$	*USHRT_MAX*
int	16	$[-32767, 32767]$	*INT_MIN*, *INT_MAX*
unsigned int	16	$[0, 65535]$	*UINT_MAX*
long	32	$\left[-2^{31}+1, 2147483647\right]$	*LONG_MIN*, *LONG_MAX*
unsigned long	32	$[0, 4294967295]$	*ULONG_MAX*
long long	64	$\left[-2^{63}+1, 2^{63}-1\right]$	*LLONG_MIN*, *LLONG_MAX*
unsigned long long	64	$\left[0, 2^{64}-1\right]$	*ULLONG_MAX*

7.2.2 Datentypen für Zeichen

Der Datentyp **char** orientiert sich in seiner Größe typischerweise an dem Byte, der kleinsten adressierbaren Einheit. In <*limits.h*> findet sich die Konstante *CHAR_BIT*, die die Anzahl der Bits bei **char** angibt. Dieser Wert muss mindestens 8 betragen und weicht davon auch normalerweise nicht ab.

Der Datentyp **char** gehört mit zu den ganzzahligen Datentypen und entsprechend können Zeichen wie ganze Zahlen und umgekehrt behandelt werden. Der C-Standard überlässt den Implementierungen die Entscheidung, ob **char** vorzeichenbehaftet ist oder nicht. Wer sicher gehen möchte, spezifiziert dies explizit mit **signed char** oder **unsigned char**.

Da für die Kodierung einiger internationaler Zeichensätze mehrere Bytes benötigt werden, gibt es neben **char** einen in <*wchar.h*> definierten Datentyp *wchar_t*. Dies ist ebenfalls ein ganzzahliger Datentyp, der groß genug ist, um alle denkbaren Zeichensätze der lokalen Plattform zu unterstützen. Auch bei *wchar_t* ist nicht festgelegt, ob der Datentyp vorzeichenbehaftet ist oder nicht. Der für Zeichen zulässige Wertebereich steht über die Konstanten *WCHAR_MIN* und *WCHAR_MAX* zur Verfügung. Um bei *getwc()* (dem *wchar_t*-Äquivalent zur Funktion *getc()*) Zeichen und Fehlern bzw. dem Eingabe-Ende unterscheiden zu können, gibt es den ganzzahligen Datentyp *wint_t*, der groß genug ist, um neben den zulässigen Zeichen zwischen *WCHAR_MIN* und *WCHAR_MAX* auch *WEOF* repräsentieren zu können.

Zeichenkonstanten werden in einfache Hochkommata eingeschlossen, etwa 'a' (vom Datentyp **char**) oder *L*'a' (vom Datentyp *wchar_t*).

Für eine Reihe von nicht druckbaren Zeichen gibt es *Ersatzdarstellungen*:

\b	BS	*backspace*
\f	FF	*formfeed*
\n	LF	*newline*, Zeilentrenner
\r	CR	*carriage return*, „Wagenrücklauf“
\t	HT	Horizontaler Tabulator
\\	\	„Fluchtsymbol“
\’	’	einfaches Hochkomma
\a		*audible bell*, Signalton
\0	NUL	Null-Byte
\ddd		ASCII-Code (oktal)

Zeichen können ohne explizite Konvertierungen wie ganzzahlige Werte verwendet werden, wie folgendes Beispiel zeigt:

Programm 7.1: Zeichen als ganzzahlige Werte (*rot13.c*)

```
1  #include <stdio.h>
2
3  const int letters = 'z' − 'a' + 1;
4  const int rotate = 13;
5  int main() {
6      int ch;
7      while ((ch = getchar()) != EOF) {
8          if (ch >= 'a' && ch <= 'z') {
9              ch = 'a' + (ch − 'a' + rotate) % letters;
10         } else if (ch >= 'A' && ch <= 'Z') {
11             ch = 'A' + (ch − 'A' + rotate) % letters;
12         }
13         putchar(ch);
14     }
15 }
```

```
doolin$ gcc -Wall -std=c99 rot13.c
doolin$ echo Hallo | a.out
Unyyb
doolin$ echo Hallo | a.out | a.out
Hallo
doolin$
```

7.2.3 Gleitkommazahlen (float und double)

⟨floating-point-type-specifier⟩ ⟶ **float**
⟶ **double**
⟶ **long double**
⟶ ⟨complex-type-specifier⟩
⟨complex-type-specifier⟩ ⟶ **float _Complex**
⟶ **double _Complex**
⟶ **long double _Complex**

In der Vergangenheit gab es eine Vielzahl stark abweichender Darstellungen für Gleitkommazahlen, bis 1985 mit dem Standard IEEE-754 (auch IEC 60559 genannt) eine Vereinheitlichung gelang, die sich rasch durchsetzte und von allen heute üblichen Prozessor-Architekturen unterstützt wird. Der C-Standard bezieht sich ausdrücklich auf IEEE-754, auch wenn die Einhaltung davon nicht für Implementierungen garantiert werden kann, bei denen die Hardware-Voraussetzungen dafür fehlen.

Bei IEEE-754 besteht die binäre Darstellung einer Gleitkommazahl aus drei Komponenten,

- dem Vorzeichen s (ein Bit),
- dem aus q Bits bestehenden Exponenten $\{e_i\}_{i=1}^{q}$,
- und der aus p Bits bestehenden Mantisse $\{m_i\}_{i=1}^{p}$.

Für die Darstellung des Exponenten e hat sich folgende verschobene Darstellung als praktisch erwiesen:

$$e = -2^{q-1} + 1 + \sum_{i=1}^{q} e_i 2^{i-1}$$

Entsprechend liegt e im Wertebereich $\left[-2^{q-1}+1, 2^{q-1}\right]$. Da die beiden Extremwerte für besondere Kodierungen verwendet werden, beschränkt sich der reguläre Bereich von e auf $[e_{min}, e_{max}]$ mit $e_{min} = -2^{q-1} + 2$ und $e_{max} = 2^{q-1} - 1$. Bei dem aus insgesamt 32 Bits bestehenden Format für den Datentyp **float** mit $q = 8$ ergibt das den Bereich $[-126, 127]$.

Wenn e im Intervall $[e_{min}, e_{max}]$ liegt, dann wird die Mantisse m so interpretiert:

$$m = 1 + \sum_{i=1}^{p} m_i 2^{i-p-1}$$

Wie sich dieser sogenannten normalisierten Darstellung entnehmen lässt, gibt es ein implizites auf 1 gesetztes Bit, d.h. m entspricht der im Zweier-System notierten Zahl $1, m_p m_{p-1} \ldots m_2 m_1$.

Der gesamte Wert ergibt sich dann aus $x = (-1)^s \times 2^e \times m$.

Um die 0 darzustellen, gilt der Sonderfall, dass $m = 0$, wenn alle Bits des Exponenten gleich 0 sind, d.h. $e = -2^{q-1} + 1$, und zusätzlich auch alle Bits der Mantisse gleich 0 sind. Da das Vorzeichenbit unabhängig davon gesetzt sein kann oder nicht, gibt es zwei Darstellungen für die Null: -0 und $+0$.

Ferner unterstützt IEEE-754 auch die sogenannte denormalisierte Darstellung, bei der alle Bits des Exponenten gleich 0 sind, es aber in der Mantisse mindestens ein Bit mit $m_i = 1$ Mantisse gibt. In diesem Falle ergibt sich folgende Interpretation:

$$\begin{aligned} m &= \sum_{i=1}^{p} m_i 2^{i-p-1} \\ x &= (-1)^s \times 2^{e_{min}} \times m \end{aligned}$$

Der Fall $e = e_{max} + 1$ erlaubt es, ∞, $-\infty$ und *NaN* (*not a number*) mit in den Wertebereich der Gleitkommazahlen aufzunehmen. ∞ und $-\infty$ werden bei Überläufen verwendet und NaN bei undefinierten Resultaten (Beispiel: Wurzel aus einer negativen Zahl).

Zusammenfassend:

$$
\begin{aligned}
e &= -2^{q-1} + 1 + \sum_{i=1}^{q} e_i 2^{i-1} \\
m &= \begin{cases} 0 & \text{falls } \forall i \in [1,p] : m_i = 0 \\ 1 + \sum_{i=1}^{p} m_i 2^{i-p-1} & \text{falls } e \in [e_{min}, e_{max}] \wedge \exists i \in [1,p] : m_i \neq 0 \\ \sum_{i=1}^{p} m_i 2^{i-p-1} & \text{falls } e < e_{min} \vee e > e_{max} \end{cases} \\
x &= \begin{cases} (-1)^s \times 2^e \times m & \text{falls } e \in [e_{min}, e_{max}] \\ (-1)^s \times 2^{e_{min}} \times m & \text{falls } e = e_{min} - 1 \\ (-1)^s \infty & \text{falls } e = e_{max} + 1 \wedge m = 0 \\ NaN & \text{falls } e = e_{max} + 1 \wedge m \neq 0 \end{cases}
\end{aligned}
$$

IEEE-754 gibt Konfigurationen für einfache, doppelte und erweiterte Genauigkeiten vor, die auch so von C übernommen wurden. Allerdings steht nicht auf jeder Architektur **long double** zur Verfügung, so dass in solchen Fällen ersatzweise nur eine doppelte Genauigkeit verwendet wird. Umgekehrt rechnen einige Architekturen grundsätzlich mit einer höheren Genauigkeit und runden dann, wenn eine Zuweisung an eine Variable des Typs **float** oder **double** erfolgt. Dies alles ist entsprechend IEEE-754 zulässig – auch wenn dies zur Konsequenz hat, dass Ergebnisse selbst bei elementaren Operationen auf verschiedenen konformen Architekturen voneinander abweichen können. Hier ist die Übersicht:

Datentyp	Bits	q	p
float	32	8	23
double	64	11	52
long double		≥ 15	≥ 63

Der Umgang mit Gleitkomma-Zahlen ist nicht immer ganz einfach, weil selbst kleine Rundungsfehler katastrophale Ausmasse nehmen können. Folgendes Beispiel demonstriert am Beispiel der Flächenberechnung eines Dreiecks wie problematisch selbst Subtraktionen sein können, wenn die Operanden nicht exakt sind. Im Falle extrem schmaler Dreiecke wird der Unterschied fatal:

Programm 7.2: Problematik von Rundungsfehlern (*triangle.c*)

```
1  #include <math.h>
2  #include <stdio.h>
3
4  double triangle_area1(double a, double b, double c) {
5      double s = (a + b + c) / 2;
6      return sqrt(s*(s-a)*(s-b)*(s-c));
7  }
8
9  #define SWAP(a,b) {int tmp; tmp = a; a = b; b = tmp;}
10 double triangle_area2(double a, double b, double c) {
11     /* sort a, b, and c in descending order, applying a bubble-sort */
12     if (a < b) SWAP(a, b); if (b < c) SWAP(b, c); if (a < b) SWAP(a, b);
13     /* formula by W. Kahan */
14     return sqrt((a + (b + c)) * (c - (a - b)) *
15                 (c + (a - b)) * (a + (b - c)))/4;
16 }
```

```
17
18 int main() {
19     double a, b, c;
20     printf("triangle␣side␣lenghts␣a␣b␣c:␣");
21     if (scanf("%lf␣%lf␣%lf", &a, &b, &c) != 3) {
22         printf("Unable␣to␣read␣three␣floats!\n");
23         return 1;
24     }
25     double a1 = triangle_area1(a, b, c);
26     double a2 = triangle_area2(a, b, c);
27     printf("Formula␣#1␣delivers␣%.16lf\n", a1);
28     printf("Formula␣#2␣delivers␣%.16lf\n", a2);
29     printf("Difference:␣%lg\n", fabs(a1−a2));
30     return 0;
31 }
```

```
dublin$ gcc -Wall -std=c99 triangle.c -lm
dublin$ a.out
triangle side lenghts a b c: 1e10 1e10 1e-10
Formula #1 delivers 0.0000000000000000
Formula #2 delivers 0.5000000000000000
Difference: 0.5
dublin$
```

In diesem Beispiel verwendet *triangle_area1* die übliche Flächenberechnungsformel. Das Problem ist hierbei, dass bei der Addition von $a + b + c$ bei einem schmalen Dreieck die kleine Seitenlänge verschwinden kann, wenn die Größenordnungen weit genug auseinander liegen. Wenn dann später die Differenz zwischen s und der kleinen Seitenlänge gebildet wird, kann der Fehler katastrophal werden. In der von William Kahan vorgeschlagenen Berechnungsformel wird diese Problematik vermieden (siehe dazu auch [Goldberg 1991]).

Die Frage, ob zwei Gleitkommazahlen gleich sind, lässt sich nicht allgemein beantworten. Gilt beispielsweise $(x/y)*y == x$? Interessanterweise garantiert hier IEEE-754 die Gleichheit, falls x und y beide in doppelter Genauigkeit repräsentierbar sind (also **double**), $|m| < 2^{52}$ und $n = 2^i + 2^j$ (siehe Theorem 7 aus [Goldberg 1991]). Aber beliebig verallgemeinern lässt sich dies nicht:

Programm 7.3: Problematik der Gleichheit bei Gleitkommazahlen (*equality.c*)

```
1  #include <stdio.h>
2  int main() {
3      double x, y;
4      printf("x␣y␣=␣");
5      if (scanf("%lf␣%lf", &x, &y) != 2) {
6          printf("Unable␣to␣read␣two␣floats!\n");
7          return 1;
8      }
9      if ((x/y)*y == x) {
10         printf("equal\n");
11     } else {
12         printf("not␣equal\n");
13     }
14     return 0;
15 }
```

```
dublin$ gcc -Wall -std=c99 equality.c
dublin$ a.out
x y = 3 10
equal
dublin$ a.out
x y = 2 0.77777777777777
not equal
dublin$
```

Gelegentlich wird nahegelegt, statt dem ==-Operator auf die Nähe zu testen, d.h. $x \sim y \Leftrightarrow |x - y| < \epsilon$, wobei ϵ für eine angenommene Genauigkeit steht. Wie [Goldberg 1991] jedoch darlegt, lässt dies Fragen offen, wie etwa ϵ gewählt werden solle und ob der Wegfall der (bei == selbstverständlichen) Äquivalenzrelation zu verschmerzen ist, da aus $x \sim y$ und $y \sim z$ sich nicht $x \sim z$ folgern lässt. Zudem bleibt auch die Frage, ob auch dann $x \sim y$ gelten soll, wenn beide genügend nahe an der 0 sind, aber die Vorzeichen sich voneinander unterscheiden. Insofern lässt sich die Frage nach einem Äquivalenztest nie allgemein beantworten, sondern muss unter Berücksichtigung der konkreten Problemstellung gelöst werden.

7.2.4 Aufzählungsdatentypen

⟨enumeration-type-specifier⟩	⟶	⟨enumeration-type-definition⟩
	⟶	⟨enumeration-type-reference⟩
⟨enumeration-type-definition⟩	⟶	**enum** [⟨enumeration-tag⟩] „{“
		⟨enumeration-definition-list⟩ [„,“] „}“
⟨enumeration-tag⟩	⟶	⟨identifier⟩
⟨enumeration-definition-list⟩	⟶	⟨enumeration-constant-definition⟩
	⟶	⟨enumeration-definition-list⟩ „,“
		⟨enumeration-constant-definition⟩
⟨enumeration-constant-definition⟩	⟶	⟨enumeration-constant⟩
	⟶	⟨enumeration-constant⟩ „=“ ⟨expression⟩
⟨enumeration-constant⟩	⟶	⟨identifier⟩
⟨enumeration-type-reference⟩	⟶	**enum** ⟨enumeration-tag⟩

- Aufzählungsdatentypen sind grundsätzlich ganzzahlig und entsprechend auch kompatibel mit anderen ganzzahligen Datentypen.
- Welcher vorzeichenbehaftete ganzzahlige Datentyp als Grundtyp für Aufzählungen dient (etwa **int** oder **short**) ist nicht festgelegt.
- Steht zwischen **enum** und der Aufzählung ein Bezeichner (⟨identifier⟩), so kann dieser Name bei späteren Deklarationen (bei einer ⟨enumeration-type-reference⟩) wieder verwendet werden.
- Sofern nichts anderes angegeben ist, erhält das erste Aufzählungselement den Wert 0.

- Bei den übrigen Aufzählungselementen wird jeweils der Wert des Vorgängers genommen und 1 dazuaddiert.
- Diese standardmäßig vergebenen Werte können durch die Angabe einer Konstante verändert werden. Damit wird dann auch implizit der Wert der nächsten Konstante verändert, sofern die nicht ebenfalls explizit gesetzt wird.
- Gegeben sei folgendes (nicht nachahmenswerte) Beispiel:

  enum *msglevel* {
  notice, *warning*, *error* = 10,
  alert = *error* + 10, *crit*, *emerg* = *crit* ∗ 2,
  debug = −1, *debug0*
  };

  Dann ergeben sich daraus folgende Werte: *notice* = 0, *warning* = 1, *error* = 10, *alert* = 20, *crit* = 21, *emerg* = 42, *debug* = −1 und *debug0* = 0. C stört es dabei nicht, dass zwei Konstanten (*notice* und *debug0*) den gleichen Wert haben.

Programm 7.4: Verwendung von Aufzählungstypen (*days.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <time.h>
5
6  enum days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
7  char* dayname[] = {
8      "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
9  };
10
11 int main() {
12     enum days day;
13     for (day = Monday; day <= Sunday; ++day) {
14         printf("Day␣%d␣=␣%s\n", day, dayname[day]);
15     }
16     /* seed the pseudo-random generator */
17     unsigned int seed = time(0); srand(seed);
18     /* select and print a pseudo-random day */
19     enum days favorite_day = rand() % 7;
20     printf("My␣favorite␣day:␣%s\n", dayname[favorite_day]);
21 }
```

```
dublin$ gcc -Wall -std=c99 days.c
dublin$ a.out
Day 0 = Monday
Day 1 = Tuesday
Day 2 = Wednesday
Day 3 = Thursday
Day 4 = Friday
Day 5 = Saturday
Day 6 = Sunday
My favorite day: Tuesday
dublin$
```

7.2.5 Zeigertypen

⟨declaration⟩ ⟶ ⟨declaration-specifiers⟩ [⟨init-declarator-list⟩]
⟨declaration-specifiers⟩ ⟶ ⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨type-specifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨function-specifier⟩ [⟨declaration-specifiers⟩]
⟨init-declarator-list⟩ ⟶ ⟨init-declarator⟩
⟶ ⟨init-declarator-list⟩ „,“ ⟨init-declarator⟩
⟨init-declarator⟩ ⟶ ⟨declarator⟩
⟶ ⟨declarator⟩ „=“ ⟨initializer⟩
⟨declarator⟩ ⟶ [⟨pointer⟩] ⟨direct-declarator⟩
⟨pointer⟩ ⟶ „*“ [⟨type-qualifier-list⟩]
⟶ „*“ [⟨type-qualifier-list⟩] ⟨pointer⟩

Die Grammatik weist auf einen wichtigen subtilen Punkt hin. Im Rahmen der linken Seite einer Deklaration (konkret: ⟨type-specifier⟩) können keine Zeigertypen deklariert werden. Stattdessen ist dies nur im Rahmen der rechten Seite möglich (konkret: ⟨init-declarator⟩). Das hat zur Konsequenz, dass bei

```
int* p, i; /* verwirrend */
```

die Variable *p* den Datentyp **int*** hat, während *i* den Datentyp **int** erhält. Der Stern (in der Grammatik ⟨pointer⟩) bindet also nur zu *p* und nicht zu *i*. Da dies verwirrend ist, sollten gemischte Deklarationen vermieden werden:

```
int* p; int i; /* besser */
```

Folgendes Programm demonstriert einen Zeigertyp an einem einfachen Beispiel:

Programm 7.5: Verwendung von Zeigern (*zeiger.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int i = 13;
5      int* p = &i; /* Zeiger p zeigt auf i; &i = Adresse von i */
6
7      printf("i=%d, p=%p (Adresse), *p=%d (Wert)\n", i, p, *p);
8
9      ++i;
10     printf("i=%d, *p=%d\n", i, *p);
11
12     ++*p; /* *p ist ein Links-Wert */
13     printf("i=%d, *p=%d\n", i, *p);
14 }
```

```
thales$ gcc -Wall zeiger.c
thales$ a.out
i=13, p=bffff418 (Adresse), *p=13 (Wert)
i=14, *p=14
i=15, *p=15
thales$
```

Wenn – wie hier in diesem Beispiel – Zeiger auf lokale Variablen gesetzt werden, besteht die Gefahr, dass der Zeigerwert noch zur Verfügung steht, nachdem die lokale Variable nicht mehr existiert. In diesem Falle ist der Effekt einer Dereferenzierung undefiniert.

Es ist zulässig, ganze Zahlen zu einem Zeiger zu addieren oder davon zu subtrahieren. Dabei wird jedoch der zu addierende oder zu subtrahierende Wert implizit mit der Größe des Typs multipliziert, auf den der Zeiger zeigt. Weiter ist es zulässig, Zeiger des gleichen Typs voneinander zu subtrahieren. Das Resultat wird dann implizit durch die Größe des referenzierten Typs geteilt.

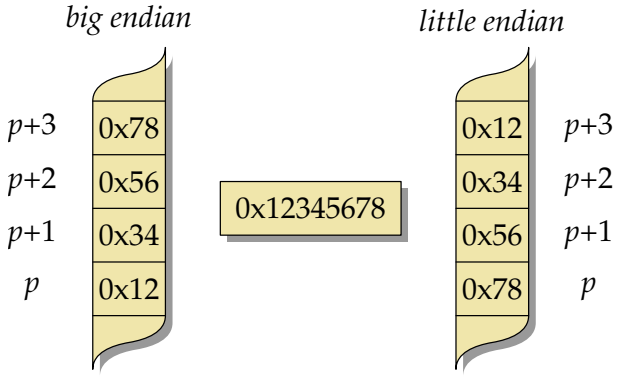

Abbildung 7.2: *big* vs. *little endian*

Programm 7.6: Zeiger-Arithmetik (*zeiger1.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      unsigned int value = 0x12345678;
5      unsigned char* p = (unsigned char*) &value;
6
7      for (int i = 0; i < sizeof(unsigned int); ++i) {
8          printf("p+%d -> 0x%02hhx\n", i, *(p+i));
9      }
10 }
```

```
doolin$ uname -m
sun4u
doolin$ gcc -Wall -std=c99 zeiger1.c
doolin$ a.out
p+0 --> 0x12
p+1 --> 0x34
p+2 --> 0x56
p+3 --> 0x78
doolin$ scp zeiger1.c zeus.rz.uni-ulm.de:
[...]
doolin$ ssh zeus.rz.uni-ulm.de
[...]
zeus$ uname -m
x86_64
zeus$ gcc -Wall -std=c99 zeiger1.c
zeus$ a.out
p+0 --> 0x78
p+1 --> 0x56
p+2 --> 0x34
p+3 --> 0x12
zeus$
```

Im Programm 7.6 wird der Speicher byteweise „durchleuchtet“. Hierbei fällt auf, dass die interne Speicherung einer ganzen Zahl bei unterschiedlichen Architekturen (SPARC vs. Intel x86) verschieden ist: *big endian* vs. *little endian* (siehe Abbildung 7.2). Bei

- *little endian* wird das niedrigstwertige Byte an der niedrigsten Adresse abgelegt, während bei
- *big endian* das niedrigswertige Byte sich bei der höchsten Adresse befindet.

7.2.6 Typ-Konvertierungen

Typ-Konvertierungen können in C sowohl implizit als auch explizit erfolgen. Implizite Konvertierungen werden angewendet bei Zuweisungs-Operatoren, Parameterübergaben und Operatoren. Letzteres schließt auch die monadischen Operatoren mit ein. Explizite Konvertierungen erfolgen durch den sogenannten Cast-Operator.

7.2.6.1 Konvertierungen zwischen numerischen Datentypen

Bei einer Konvertierung zwischen numerischen Typen gilt der Grundsatz, dass – wenn irgendwie möglich – der Wert zu erhalten ist. Falls das jedoch nicht möglich ist, gelten folgende Regeln:

- Bei einer Konvertierung eines vorzeichenbehafteten ganzzahligen Datentyps zum Datentyp ohne Vorzeichen *gleichen Ranges* (also etwa von **int** zu **unsigned int**) wird eine ganze Zahl $a < 0$ zu b konvertiert, wobei gilt, dass $a \bmod 2^n = b \bmod 2^n$ mit n der Anzahl der verwendeten Bits, wobei hier der mod-Operator entsprechend der F-Definition bzw. Euklid gemeint ist. Dies entspricht der Repräsentierung des Zweier-Komplements.
- Der umgekehrte Weg, d.h. vom ganzzahligen Datentyp ohne Vorzeichen zum vorzeichenbehafteten Datentyp gleichen Ranges (also etwa von **unsigned int** zu **int**) hinterlässt ein *undefiniertes* Resultat, falls der Wert nicht darstellbar ist.

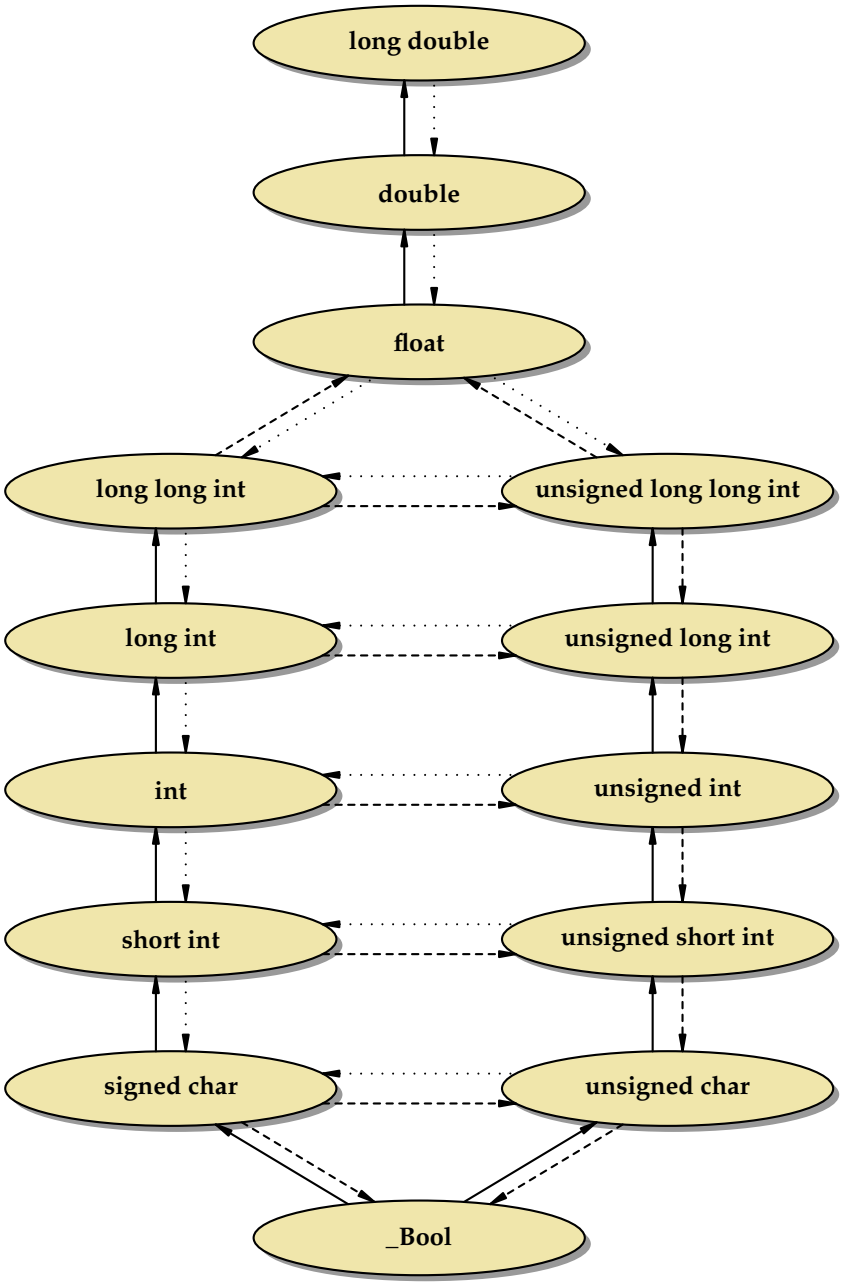

Abbildung 7.3: Konvertierungen zwischen numerischen Datentypen

- Bei einer Konvertierung von größeren ganzzahligeren Datentypen zu entsprechenden kleineren Datentypen werden die nicht mehr darstellbaren höherwertigen Bits weggeblendet, d.h. es gilt wiederum $a \bmod 2^n = b \bmod 2^n$, wobei n die Anzahl der Bits im kleineren Datentyp ist. (Das Resultat ist aber nur bei ganzzahligen Datentypen ohne Vorzeichen wohldefiniert.)
- Bei Konvertierungen zu *_Bool* ist das Resultat 0 (*false*), falls der Ausgangswert 0 ist, ansonsten immer 1 (*true*).
- Bei Konvertierungen von Gleitkommazahlen zu ganzzahligen Datentypen wird der ganzzahlige Anteil verwendet. Ist dieser im Zieltyp nicht darstellbar, so ist das Resultat undefiniert.
- Umgekehrt (beispielsweise auf dem Wege von **long long int** zu **float**) ist einer der beiden unmittelbar benachbarten darstellbaren Werte zu nehmen, d.h. es gilt entweder $a = b$ oder $a < b \wedge \nexists x : a < x < b$ oder $a > b \wedge \nexists x : a > x > b$ mit x aus der Menge des Zieltyps.

Die Abbildung 7.3 zeigt die Konvertierungsbeziehungen der numerischen Datentypen untereinander. Pfeile mit durchgezogenen Linien stehen dabei für werterhaltende Konvertierungen. Bei gestrichelten Linien bleibt der Wert unter Umständen nicht erhalten, aber die Konvertierung ist in jedem Falle wohldefiniert. Bei gepunkteten Linien kann es Fälle mit undefinierten Resultaten geben. Datentypen, die in dem Diagramm auf der gleichen Höhe stehen, haben den gleichen Rang.

7.2.6.2 Konvertierungen anderer skalarer Datentypen

Jeder Aufzählungsdatentyp ist einem der ganzzahligen Datentypen implizit und implementierungsabhängig zugeordnet. Eine Konvertierung hängt von dieser (normalerweise nicht bekannten) Zuordnung ab.

Zeiger lassen sich in C grundsätzlich als ganzzahlige Werte betrachten. Allerdings garantiert C nicht, dass es einen ganzzahligen Datentyp gibt, der den Wert eines Zeigers ohne Verlust aufnehmen kann. C99 hat hier die Datentypen *intptr_t* und *uintptr_t* in <*stdint.h*> eingeführt, die für die Repräsentierung von Zeigern als ganze Zahlen den geeignetsten Typ liefern. Selbst wenn diese groß genug sind, um Zeiger ohne Verlust aufnehmen zu können, so lässt der Standard dennoch offen, wie sich die beiden Typen *intptr_t* und *uintptr_t* innerhalb der Hierarchie der ganzzahligen Datentypen einordnen. Aber die weiteren Konvertierungsschritte und die damit verbundenen Konsequenzen ergeben sich aus dieser Einordnung.

Die Zahl 0 wird bei einer Konvertierung in einen Zeigertyp immer in den Null-Zeiger konvertiert.

7.2.6.3 Implizite Konvertierungen

Bei Zuweisungen wird der Rechts-Wert in den Datentyp des Links-Wertes konvertiert. Dies geschieht analog bei Funktionsaufrufen, wenn eine vollständige Deklaration der Funktion mit allen Parametern vorliegt. Wenn diese fehlt oder (wie beispielsweise bei *printf*) nicht vollständig ist, dann wird **float** implizit zu **double** konvertiert.

Die monadischen Operatoren !, −, +, ~ und ∗ konvertieren implizit ihren Operanden:

- Ein vorzeichenbehafteter ganzzahliger Datentyp mit einem Rang niedriger als **int** wird zu **int** konvertiert,
- Ganzzahlige Datentypen ohne Vorzeichen werden ebenfalls zu **int** konvertiert, falls sie einen Rang niedriger als **int** haben und ihre Werte in jedem Falle von **int** darstellbar sind. Ist nur letzteres nicht der Fall, so erfolgt eine implizite Konvertierung zu **unsigned int**.

- Ranghöhere ganzzahlige Datentypen werden nicht konvertiert.

Die gleichen Regeln werden auch getrennt für die beiden Operanden der Schiebe-Operatoren << und >> angewendet.

Bei dyadischen Operatoren mit numerischen Operanden werden folgende implizite Konvertierungen angewendet:

- Sind die Typen beider Operanden vorzeichenbehaftet oder beide ohne Vorzeichen, so findet eine implizite Konvertierung zu dem Datentyp mit dem höheren Rang statt (siehe Abbildung 7.3). So wird beispielsweise bei einer Addition eines Werts des Typs **short int** zu einem Wert des Typs **long int** der erstere in den Datentyp des zweiten Operanden konvertiert, bevor die Addition durchgeführt wird.

- Ist bei einem gemischten Fall (**signed** vs. **unsigned**) in jedem Falle eine Repräsentierung eines Werts des vorzeichenlosen Typs in dem vorzeichenbehafteten Typ möglich (wie etwa typischerweise bei **unsigned short** und **long int**), so wird der Operand des vorzeichenlosen Typs in den vorzeichenbehafteten Typ des anderen Operanden konvertiert.

- Bei den anderen gemischten Fällen werden beide Operanden in die vorzeichenlose Variante des höherrangigen Operandentyps konvertiert. So wird beispielsweise eine Addition bei **unsigned int** und **int** in **unsigned int** durchgeführt.

Programm 7.7: Implizite Konvertierungen (*conversions.c*)

```
1  #include <stdio.h>
2  #include <limits.h>
3  #include <stdbool.h>
4
5  void print_bool(bool boolval) {
6      if (boolval) {
7          puts("true");
8      } else {
9          puts("false");
10     }
11 }
12
13 int main() {
14     /* unsigned int --> unsigned short int */
15     unsigned int i1 = UINT_MAX; unsigned short int i2 = i1;
16     printf("%x -> %hx\n", i1, i2);
17     /* int op unsigned int --> unsigned int */
18     int i3 = -1; unsigned int i4 = UINT_MAX;
19     printf("%d == %u: ", i3, i4); print_bool(i3 == i4);
20     /* int --> float --> int */
21     int i5 = 123456789; float f = i5; int i6 = f;
22     printf("%d -> %g -> %d\n", i5, f, i6);
23 }
```

```
doolin$ gcc -Wall -std=c99 conversions.c
doolin$ a.out
ffffffff -> ffff
-1 == 4294967295: true
123456789 -> 1.23457e+08 -> 123456792
doolin$
```

7.3 Typen für unveränderliche Werte

C sieht einige spezielle Attribute bei Typ-Deklarationen vor. Darunter ist auch **const**:

⟨declaration-specifiers⟩ ⟶ ⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨type-specifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
⟶ ⟨function-specifier⟩ [⟨declaration-specifiers⟩]
⟨type-qualifier⟩ ⟶ **const**
⟶ **volatile**
⟶ **restrict**

Die Verwendung dieses Attributs hat zwei Vorteile:

- Der Programmierer wird davor bewahrt, eine Konstante versehentlich zu verändern. (Dies funktioniert aber nur beschränkt.)
- Besondere Optimierungen sind für den Übersetzer möglich, wenn bekannt ist, dass sich bestimmte Variablen nicht verändern dürfen.

Wie folgendes Beispiel demonstriert, beschränkt sich der gcc nur auf Warnungen, wenn Konstanten verändert werden:

Programm 7.8: Arbeiten mit Konstanten (*const.c*)

```
1 #include <stdio.h>
2
3 int main() {
4     const int i = 1;
5
6     i++; /* das geht doch nicht, oder?! */
7     printf("i=%d\n", i);
8 }
```

```
doolin$ gcc -Wall -std=c99 const.c
const.c: In function `main':
const.c:6: warning: increment of read-only variable `i'
doolin$ a.out
i=2
doolin$
```

7.4 Aggregierte Typen

7.4.1 Vektoren

⟨direct-declarator⟩	⟶	⟨simple-declarator⟩
	⟶	„(“ ⟨simple-declarator⟩ „)“
	⟶	⟨function-declarator⟩
	⟶	⟨array-declarator⟩
⟨array-declarator⟩	⟶	⟨direct-declarator⟩ „[“ [⟨array-qualifier-list⟩]
		[⟨array-size-expression⟩] „]“
⟨array-qualifier-list⟩	⟶	⟨array-qualifier⟩
	⟶	⟨array-qualifier-list⟩ ⟨array-qualifier⟩
⟨array-qualifier⟩	⟶	**static**
	⟶	**restrict**
	⟶	**const**
	⟶	**volatile**
⟨array-size-expression⟩	⟶	⟨assignment-expression⟩
	⟶	„*“
⟨simple-declarator⟩	⟶	⟨identifier⟩

Wie bei den Zeigertypen erfolgen die Typspezifikationen eines Vektors nicht im Rahmen eines ⟨type-specifier⟩. Stattdessen gehört eine Vektordeklaration zu dem ⟨init-declarator⟩. Das bedeutet, dass die Präzisierung des Typs zur genannten Variablen unmittelbar gehört. Entsprechend deklariert

int *a*[10], *i*;

eine Vektorvariable *a* und eine ganzzahlige Variable *i*.

Vektoren und Zeiger sind eng miteinander verwandt. Der Variablenname eines Vektors (im obigen Beispiel *a*) ist ein konstanter Zeiger auf den zugehörigen Element-Typ (im Beispiel **int**), der auf das erste Element verweist. Allerdings liefert – wie das folgende Beispiel zeigt – **sizeof** *a* die Größe des gesamten Vektors und nicht etwa nur die des Zeigers:

Programm 7.9: Vektoren und Zeiger (*array.c*)

```
1  #include <stdio.h>
2  #include <stddef.h>
3
4  int main() {
5      int a[5] = {1, 2, 3, 4, 5};
6      const size_t SIZE = sizeof(a) / sizeof(a[0]); /* Groesse des Arrays bestimmen */
7      int* p = a; /* kann statt a verwendet werden */
8
9      /* aber: a weiss noch die Gesamtgroesse, p nicht */
10     printf("SIZE=%zd, sizeof(a)=%zd, sizeof(p)=%zd\n",
11         SIZE, sizeof(a), sizeof(p));
12
```

```
13      for (int i = 0; i < SIZE; ++i) {
14          *(a + i) = i+1; /* gleichbedeutend mit a[i] = i+1 */
15      }
16
17      /* Elemente von a aufsummieren */
18      int sum = 0;
19      for (int i = 0; i < SIZE; i++) {
20          sum += p[i]; /* gleichbedeutend mit ... = a[i]; */
21      }
22      printf("Summe: %d\n", sum);
23  }
```

```
doolin$ gcc -Wall -std=c99 array.c
doolin$ a.out
SIZE=5, sizeof(a)=20, sizeof(p)=4
Summe: 15
doolin$
```

Grundsätzlich beginnt die Indizierung bei 0. Ein Vektor mit 5 Elementen hat entsprechend zulässige Indizes im Bereich von 0 bis 4. Wird der zulässige Index-Bereich verlassen, so ist der Effekt undefiniert. Es ist dann damit zu rechnen, dass irgendeine andersweitig belegte Speicherfläche adressiert wird oder es zu einer harten Unterbrechung kommt, weil eine unzulässige Adresse dereferenziert wurde. Was tatsächlich passiert, hängt von der jeweiligen Adressraumbelegung ab:

Programm 7.10: Indizierungsfehler bei Vektoren (*array1.c*)

```
1   #include <stdio.h>
2
3   const int SIZE = 10;
4
5   int main() {
6       int x = 1000, y = 2000;
7       int a[SIZE];
8
9       /* wir wollen mal sehen, wo denn das alles im Speicher liegt */
10      printf("&x = %p, &y = %p\n&(a[0]) = %p, &(a[SIZE+1]) = %p\n",
11          &x, &y, &(a[0]), &(a[SIZE+1]));
12
13      /* Initialisierung von a */
14      for (int i = 0; i < SIZE; i++) {
15          a[i] = i * i;
16      }
17
18      /* Elemente von a aufsummieren */
19      int sum = 0;
20      for (int i = 0; i < SIZE; i++) {
21          sum += a[i];
22      }
23      printf("Summe: %d\n", sum);
24
25      /* typischer Fehler: ein Element zuviel */
```

```
26      sum = 0;
27      for (int i = 0; i <= SIZE; i++) {
28          sum += a[i];
29      }
30      printf("Summe: %d\n", sum);
31
32      /* VORSICHT -- Lebensgefahr! */
33      x = y = 0;
34      printf("x=%d, y=%d\n", x, y);
35      a[SIZE+1]++;
36      printf("x=%d, y=%d\n", x, y);
37
38      a[SIZE+10000]++; /* => segmentation fault */
39  }
```

```
dublin$ a.out
&x = ffbff77c, &y = ffbff778
&(a[0]) = ffbff750, &(a[SIZE+1]) = ffbff77c
Summe: 285
Summe: 2285
x=0, y=0
x=1, y=0
Segmentation Fault(coredump)
dublin$
```

7.4.1.1 Parameterübergabe

Der Name eines Vektors ist ein (konstanter) Zeiger auf das erste Element. Wird der Name eines Vektors als aktueller Parameter angegeben (siehe Programm 7.11), so wird der Zeigerwert als Werteparameter übergeben, d.h. die aufgerufene Funktion arbeitet nicht mit einer Kopie und kann direkt auf den originalen Vektor zugreifen.

Programm 7.11: Parameterübergabe bei Feldern (*array2.c*)

```
1   #include <stdio.h>
2
3   const int SIZE = 10;
4
5   /* Array wird veraendert, naemlich mit
6      0, 1, 2, 3, ... initialisiert! */
7   void init(int a[], int length) {
8       for (int i = 0; i < length; i++) {
9           a[i] = i;
10      }
11  }
12
13  int summe1(int a[], int length) {
14      int sum = 0;
15      for (int i = 0; i < length; i++) {
16          sum += a[i];
17      }
18      return sum;
```

```
19 }
20
21 int summe2(int* a, int length) {
22     int sum = 0;
23     for (int i = 0; i < length; i++) {
24         sum += *(a+i); /* aequivalent zu ... += a[i]; */
25     }
26     return sum;
27 }
28
29 int main() {
30     int array[SIZE];
31
32     init(array, SIZE);
33
34     printf("Summe: %d\n", summe1(array, SIZE));
35     printf("Summe: %d\n", summe2(array, SIZE));
36 }
```

Hinweis: Das Element $a[i]$ kann in dazu äquivalenter Weise auch über $*(a+i)$ mit Hilfe der *Zeigerarithmetik* referenziert werden. So wird auch der Zugriff auf Vektor-Elemente auch tatsächlich intern umgesetzt.

7.4.1.2 Mehrdimensionale Vektoren

Ein *zweidimensionaler Vektor* kann beispielsweise wie folgt anlegt werden:

```
int matrix[2][3];
```

Mit folgender Definition wird ein *initialisierter zweidimensionaler Vektor* angelegt:

```
int matrix[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

Angenommen die Anfangsadresse des Vektors liege bei $0x1000$ und eine ganze Zahl vom Typ **int** würde vier Bytes belegen, dann würde die Repräsentierung des Vektors *matrix* im Speicher folgendermaßen aussehen:

Element	Adresse	Inhalt
matrix[0][0]	0x1000	0
matrix[0][1]	0x1004	1
matrix[0][2]	0x1008	2
matrix[1][0]	0x100C	3
matrix[1][1]	0x1010	4
matrix[1][2]	0x1014	5

Die Variable *matrix* ist ein 2-elementiger Vektor, dessen Elemente 3-elementige Vektoren sind, deren Elemente wiederum ganze Zahlen sind. Nach wie vor gilt, dass der Name einer Vektor-Variablen als konstanter Zeiger auf das erste Element interpretiert wird. Somit ist *matrix* ein konstanter Zeiger auf 3-elementige Vektoren ganzer Zahlen. (Der Elementtyp dieses Zeigers ist also ein 3-elementiger Vektor aus ganzen Zahlen.) Somit bewirkt ein Inkrementieren dieses Zeigers um Eins eine Erhöhung der Adresse um $3 \cdot 4 = 12$ (3 Vektor-Elemente à 4 Bytes). Wird dieser Zeiger dereferenziert, so liegt ein 3-elementiger Vektor aus ganzen Zahlen vor, also ein Zeiger auf das erste Element dieses Vektors. Der Typ dieses Zeigers ist nun **int***, so dass ein Inkrementieren dieses Zeigers eine Erhöhung der Adresse um 4 (4 Bytes pro **int**) zur Folge hat. Somit kann der Zugriff auf das Element $matrix[i][j]$ wieder abgebildet werden auf den äquivalenten Zugriff $*(*(matrix+i)+j)$. Zunächst wird *matrix*

um i erhöht und dereferenziert. Danach wird der resultierende Zeiger um j erhöht und ebenfalls dereferenziert. Jetzt ist man beim Vektor-Element angelangt. Man beachte, dass die Adresse – wie oben bereits besprochen – bei diesen beiden Inkrement-Operationen *nicht mit derselben Schrittweite* verändert wird!

Ein Beispiel für den *Zugriff auf ein Element* von **matrix** ist *matrix*[1][2], wobei dies interpretiert wird als $*(matrix[1]+2)$ und dies wiederum erweitert wird zu $*(*(matrix + 1)+ 2)$.

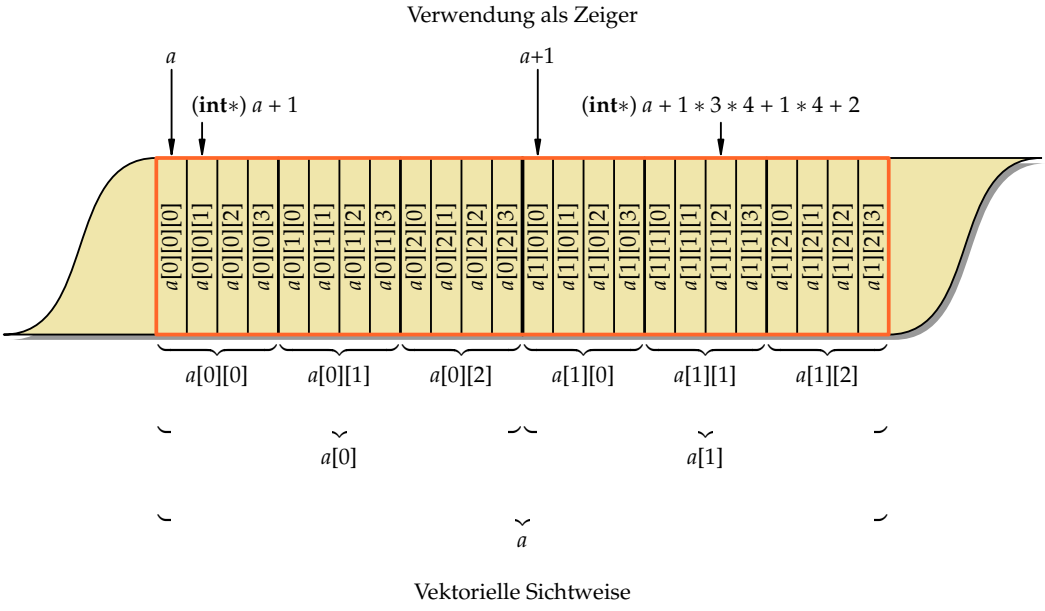

Abbildung 7.4: Repräsentierung eines dreidimensionalen Vektors im Speicher

Für die weitere Betrachtung sei folgendes Beispiel eines drei-dimensionalen Vektors gegeben (siehe auch Abbildung 7.4):

int *a*[2][3][4];

Die Variable a ist ein 2-elementiger Vektor, dessen Elemente 3-elementige Vektoren sind, dessen Elemente wiederum 4-elementige Vektoren ganzer Zahlen sind. Eine andere Sichtweise, die im Hinblick auf die Adressierung sehr nützlich ist: a ist ein Zeiger auf einen 3-elementigen Vektor, dessen Elemente 4-elementige Vektoren ganzer Zahlen sind. Der Zugriff auf ein Element $a[i][j][k]$ wird umgesetzt in $*(a[i][j] + k)$, dann in $*(*(a[i] + j)+ k)$ und schließlich in $*(*(*(a + i)+ j)+ k)$. Man beachte wie zuvor, dass die Typen der Elemente, auf die die verschiedenen Zeiger zeigen, unterschiedlichen Speicherplatzbedarf haben. So ist a ein Zeiger auf einen 3-elementiges Vektor, dessen Elemente 4-elementige Vektoren ganzer Zahlen sind. Bei einem Inkrement dieses Zeigers wird die Adresse also um $3{\cdot}4{\cdot}4 = 48$ inkrementiert. Der Zeiger $*(a + i)$ hingegen ist ein Zeiger auf einen 4-elementigen Vektor ganzer Zahlen. Ein Inkrement dieses Zeigers erhöht somit die Adresse um $4{\cdot}4 = 16$. Schließlich ist $*(*(a + i)+ j)$ ein Zeiger auf eine ganze Zahl und ein Inkrement erhöht folglich die Adresse um 4. Somit erhöhen die drei Inkrements i, j und k die Adresse in ganz unterschiedlichen Schrittweiten. Dies lässt sich auch „per Hand“ nachbilden: $*((\textbf{int}*)a + i * 3 * 4 + j * 4 + k)$ ist somit äquivalent zu $a[i][j][k]$. Die explizite Typkonvertierung (*type cast*) auf einen Zeiger für ganze Zahlen war nötig, damit die Additionen in Schritten passend für ganze Zahlen erfolgen. Dementsprechend mussten die Indizes i und j noch mit den Schrittweiten multipliziert werden. Der Faktor 4 fiel jeweils weg, da bei einem Zeiger auf **int** die Adresse bei jedem Inkrement um 4 erhöht wird.

Im folgenden Quelltext-Fragment wird die Parameterübergabe eines mehrdimensionalen Vektors vorgeführt:

```
void f1(int b[][6][7], int size) {
    /* ... */

    b[3][2][3] = /* ... */;

    /* ... */
}
void f2() {
    int a[5][6][7];
    /* ... */
    f1(a, 5);
    /* ... */
}
```

Bei der Funktionsdeklaration müssen die Größen aller Dimensionen bis auf die erste angegeben werden.

b wird dabei interpretiert als konstanter Zeiger vom Typ **int** (∗*b*)[6][7]. *b* ist also ein Zeiger auf einen 6-elementigen Vektor, dessen Elemente 7-elementige Vektoren ganzer Zahlen sind.

Bei obigem Beispiel ist jedoch vorausgesetzt, dass die Dimensionen 2 und 3 bekannt und fest sind, nämlich 6 und 7. Auf dieser Basis kann der Übersetzer einen Zugriff *b*[*i*][*j*][*k*] intern in ∗((**int**∗)*b* + *i* ∗ 6∗7 + *j* ∗ 7 + *k*) umsetzen. Fehlt diese Information jedoch, d. h. die Funktion soll mehrdimensionale Vektoren beliebiger Größe verarbeiten können, dann lässt sich der intuitive Element-Zugriff *b*[*i*][*j*][*k*] leider nicht mehr verwenden, da dann die Dimensionen 2 und 3 zur *Übersetzungszeit* nicht bekannt sind. In diesem Fall muss man leider etwas komplizierter verfahren:

```
void f1(int b[][][], int s1, int s2, int s3) {
    /* ... */
    /* Zugriff auf b[i][j][k] nun etwas komplizierter: */

    *((int*)b + i * s2*s3 + j * s3 + k) = /* ... */;

    /* ... */
}
void f2() {
    int a[5][6][7];
    /* ... */
    f1(a, 5, 6, 7);
    /* ... */
}
```

Dabei müssen natürlich die Größen aller Dimensionen angegeben werden.

Anmerkung: Der Vektor *a* im letzten Beispiel ist vom Typ **int** (∗*a*)[6][7]. In diesem Typ steckt noch Größeninformation, die zur Adressierung verwendet wird. Um nun den Parameter *b*, der keine Größeninformation mehr enthält, zu verwenden, muss er zunächst mit einer Typkonvertierung (*type cast*) in einen Zeigertyp für ganze Zahlen verwandelt werden. Das ermöglicht einen unstrukturierten und direkten Zugriff auf die Speicherfläche des Vektors.

7.4.1.3 Zeichenketten

Zeichenketten werden in C als *Vektoren von Zeichen* des Typs **char**[] repräsentiert. Das Ende der Zeichenkette wird durch ein sogenanntes *Null-Byte* ('\0') gekennzeichnet. Da es sich bei Zeichenketten um Vektoren handelt, werden bei der Parameterübergabe nur die Zeiger als Werteparameter übergeben. Die Zeichenkette (also der Inhalt des Vektors) kann entsprechend von der aufgerufenen Funktion verändert werden.

Zeichenketten-Konstanten können durch von Doppelapostrophen eingeschlossene Zeichenfolgen wie zum Beispiel "Hallo" spezifizert werden. Dies ist eine Kurzform für {'H', 'a', 'l', 'l', 'o', '\0'}. Zeichenketten-Konstanten dürfen nicht verändert werden. Sie werden, falls die zugrundeliegende Architektur dies ermöglicht, in einem Speicherbereich abgelegt, der nur Lesezugriffe zulässt.

Das folgende Programm illustriert das Arbeiten mit Zeichenketten und Zeichenketten-Konstanten:

Programm 7.12: Zeichenketten und Zeichenketten-Konstanten (*strings.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      char array[10];
5      char string[] = "Hallo!"; /* Groesse wird vom Compiler bestimmt */
6      char* s1 = "Welt";
7      char* s2;
8
9      /* array = "not OK"; */ /* nicht zulaessig; array ist konstanter Zeiger
10                                 * => Adresse nicht veraenderbar */
11
12     array[0] = 'A'; /* zulaessig */
13     array[1] = '\0'; /* ... und noch mit Null-Byte terminieren */
14     printf("array: %s\n", array);
15
16     /* s1[5] = 'B'; */ /* nicht zulaessig, da konstante
17                            Zeichenketten nicht veraendert werden
18                            duerfen */
19
20     s1 = "ok";  /* zulaessig, da Zeiger nicht konstant */
21     printf("s1: %s\n", s1);
22
23     s2 = s1;  /* zulaessig, da Zeiger nicht konstant */
24     printf("s2: %s\n", s2);
25
26     string[0] = 'X'; /* zulaessig, da es sich bei string um einen
27                            nicht-konstanten Vektor handelt */
28     printf("string: %s\n", string);
29     printf("sizeof(string): %zd\n", sizeof(string));
30 }
```

```
doolin$ gcc -Wall -std=c99 strings.c
doolin$ a.out
array: A
s1: ok
s2: ok
string: Xallo!
sizeof(string): 7
doolin$
```

Folgendes Beispiel demonstriert, auf welche Weisen typische Standardoperationen für Zeichenketten (Längenbestimmungen, Kopieren und Vergleichen) umgesetzt werden können:

Programm 7.13: Kopieren, Vergleichen, etc. von Zeichenketten (*strings1.c*)

```
1  #include <stdio.h>
2
3  /* Laenge einer Zeichenkette bestimmen */
4  int my_strlen1(char s[]) {
5      int i;
6      /* bis zum abschliessenden Null-Byte laufen */
7      for (i = 0; s[i] != '\0'; i++); /* leere Anweisung! */
8      return i;
9  }
10
11 /* Laenge einer Zeichenkette bestimmen */
12 int my_strlen2(char* s) {
13     char* t = s;
14     while (*t++);
15     return t-s-1;
16 }
17
18 /* Kopieren einer Zeichenkette von s nach t
19     Vorauss.: genuegend Platz in t */
20 void my_strcpy1(char t[], char s[]) {
21     for (int i = 0; (t[i] = s[i]) != '\0'; i++);
22 }
23
24 /* Kopieren einer Zeichenkette von s nach t
25     Vorauss.: genuegend Platz in t */
26 void my_strcpy2(char* t, char* s) {
27     for (; (*t = *s) != '\0'; t++, s++);
28 }
29
30 /* Kopieren einer Zeichenkette von s nach t
31     Vorauss.: genuegend Platz in t */
32 void my_strcpy3(char* t, char* s) {
33     while ((*t++ = *s++) != '\0');
34 }
35
36 /* Kopieren einer Zeichenkette von s nach t
37     Vorauss.: genuegend Platz in t */
38 void my_strcpy4(char *t, char *s) {
```

```
39      /* die doppelten Klammern unterdruecken eine gcc-Warnung,
40          der sonst meint, dass = mit == verwechselt worden waere */
41      while ((*t++ = *s++));
42  }
43
44  /* Vergleich zweier Zeichenketten
45      Ergebnis: 0 fuer s = t, > 0 fuer s > t und < 0 fuer s < t */
46  int my_strcmp1(char s[], char t[]) {
47      int i;
48      for (i = 0; s[i] == t[i] && s[i] != '\0'; i++);
49      return s[i] - t[i];
50  }
51
52  /* Vergleich zweier Zeichenketten
53      Ergebnis: 0 fuer s = t, > 0 fuer s > t und < 0 fuer s < t */
54  int my_strcmp2(char* s, char* t) {
55      for (; *s == *t && *s != '\0'; s++, t++);
56      return *s - *t;
57  }
58
59  int main() {
60      char str1[] = "Hallo Welt!";
61      char str2[20];
62
63      printf("my_strlen1(str1) = %d\n", my_strlen1(str1));
64
65      my_strcpy1(str2, str1);
66      printf("my_strcpy1(str2, str1) => str2 = %s\n", str2);
67
68      my_strcpy1(str2, "Hallo Tom!");
69      printf("my_strcmp1(str1, str2) = %d\n", my_strcmp1(str1, str2));
70      printf("my_strcmp1(str2, str1) = %d\n", my_strcmp1(str2, str1));
71      printf("my_strcmp1(str1, str1) = %d\n", my_strcmp1(str1, str1));
72  }
```

```
doolin$ gcc -Wall -std=c99 strings1.c
doolin$ a.out
my_strlen1(str1) = 11
my_strcpy1(str2, str1) => str2 = Hallo Welt!
my_strcmp1(str1, str2) = 3
my_strcmp1(str2, str1) = -3
my_strcmp1(str1, str1) = 0
doolin$
```

Anmerkung: Der gcc liefert bei einer Bedingung der Form *t++ = *s++ eine Warnung aus, da er von einer Verwechslung des Zuweisungs-Operators = mit dem Vergleichs-Operator == ausgeht. Diese Warnung lässt sich vermeiden, indem zusätzliche (ansonsten überflüssige) Klammern gesetzt werden. Diese Konvention dient auch dem Leser des Programmtexts als Hinweis, dass es sich dabei nicht um ein Versehen handelt.

In der C-Bibliothek gibt es bereits einige Funktionen zum Umgang mit Zeichenketten, deren Deklarationen über <*strings.h*> zugänglich sind. Im Folgenden werden einige ausgewählte Funktionen vorgestellt. Nähere Informationen und weitere Funktionen finden sich in der zugehörigen Manualseite, die beispielsweise mit *man strcpy* abgerufen werden

kann.

size_t strlen(**const char*** *s*)
: Liefert die Länge der Zeichenkette *s*. (Der Datentyp *size_t* ist der passende ganzzahlige Datentyp für Größenangaben.)

char* *strcpy*(**char*** *s1*, **const char*** *s2*)
: Kopiert den Inhalt von *s2* nach *s1* und gibt das Resultat zurück. Hinter *s1* muss genügend Platz vorhanden sein.

char* *strcat*(**char*** *s1*, **const char*** *s2*)
: Hängt eine Kopie des Inhalts von *s2* an *s1* an und gibt das Resultat zurück.

int *strcmp*(**const char*** *s1*, **const char*** *s2*)
: Vergleicht die beiden Zeichenketten *s1* und *s2*. Das Ergebnis ist kleiner 0, wenn *s1* lexikographisch kleiner als *s2* ist, gleich 0, wenn *s1* und *s2* identisch sind, und größer 0 sonst.

int *strcasecmp*(**const char*** *s1*, **const char*** *s2*)
: Vergleicht die beiden Zeichenketten analog zu *strcmp*, wobei zwischen Klein- und Großschreibung nicht unterschieden wird.

size_t strspn(**const char*** *s1*, **const char*** *s2*)
: Gibt die Länge des maximalen Präfixes von *s1* zurück, der nur aus Zeichen von *s2* besteht. (Ein Präfix ist ein Teil einer Zeichenkette, der beim ersten Zeichen beginnt.)

size_t strcspn(**const char*** *s1*, **const char*** *s2*)
: Gibt die Länge des maximalen Präfixes von *s1* zurück, das nur aus Zeichen besteht, die nicht in *s2* vorkommen.

char* *strchr*(**const char*** *s*, **int** *c*)
: Gibt einen Zeiger auf das erste Vorkommen des Zeichens *c* innerhalb von *s1* zurück. Kommt das Zeichen *c* nicht in *s1* vor, so ist das Ergebnis der Null-Zeiger.

char* *strpbrk*(**const char*** *s1*, **const char*** *s2*)
: Gibt einen Zeiger auf das erste Vorkommen eines Zeichen aus *s2* innerhalb von *s1* zurück. Kommt kein Zeichen aus *s2* in *s1* vor, so ist das Ergebnis der Null-Zeiger.

char* *strstr*(**const char*** *s1*, **const char*** *s2*)
: Gibt einen Zeiger auf das erste Vorkommen der Zeichenkette *s2* (bis auf das abschließende Null-Byte) innerhalb von *s1* zurück. Ist *s2* nicht in *s1* enthalten, so ist das Ergebnis der Null-Zeiger. Ist *s2* die leere Zeichenkette (""), so ist das Ergebnis *s1*.

char* *strtok*(**char*** *s1*, **const char*** *s2*)
: Die Funktion *strtok()* dient dazu, die Zeichenkette *s1* in einzelne Symbole (*tokens*) zu zerlegen, die ihrerseits durch ein oder mehrere Zeichen aus *s2* voneinander getrennt sind. Der erste Aufruf, bei dem *s1* angegeben ist, gibt einen Zeiger auf das erste Zeichen des ersten Symbols (der inzwischen mit einem Null-Byte versehen ist – *strtok() verändert also den String s1*) zurück. Um einen Zeiger auf das erste Zeichen des zweiten Symbols (und weiterer Symbole) zu erhalten, muss für *s1* ein Null-Zeiger übergeben werden. Hierbei kann diesselbe Zeichenkette *s2* oder auch eine andere verwendet werden. (*strtok()* speichert also intern die Position innerhalb der Zeichenkette – über verschiedene Funktionsaufrufe hinweg.) Ein Null-Zeiger als Rückgabewert signalisiert, dass es keinen weiteren Abschnitt mehr gibt.

  Anmerkung: Die Position innerhalb der Zeichenkette wird von *strtok()* in einer den Aufruf überdauernden Variable gespeichert. Bei einem Aufruf von *strtok()*, bei dem

s1 nicht der Null-Zeiger ist, geht die Position innerhalb der vorigen Zeichenkette verloren. *strtok()* kann also *nicht gleichzeitig* für zwei verschiedene Zeichenketten verwendet werden.

Programm 7.14: Verwendung diverser Funktionen für Zeichenketten (*strings2.c*)

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <strings.h>
4
5  int main() {
6      char s1[30];
7      char s2[] = "Welt!";
8      char s3[] = "HALLO WELT!";
9      char s4[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
10     char s5[] = " .,:;!?";
11     char s6[] = "lo";
12
13     printf("strcpy(s1, \"Hallo \") = \"%s\", s1 = \"%s\"\n",
14         strcpy(s1, "Hallo "), s1);
15
16     printf("strlen(s1) = %d\n", strlen(s1));
17
18     printf("strcat(s1, s2) = \"%s\", s1 = \"%s\"\n", strcat(s1, s2), s1);
19
20     printf("strcmp(s1, s3) = %d\n", strcmp(s1, s3));
21     printf("strcasecmp(s1, s3) = %d\n", strcasecmp(s1, s3));
22
23     printf("strspn(s1, s4) = %d\n", strspn(s1, s4));
24     printf("strcspn(s1, s5) = %d\n", strcspn(s1, s5));
25
26     printf("strchr(s1, 'e') = \"%s\"\n", strchr(s1, 'e'));
27
28     printf("strpbrk(s1, s5) = \"%s\"\n", strpbrk(s1, s5));
29
30     printf("strstr(s1, s6) = \"%s\"\n", strstr(s1, s6));
31 }
```

```
doolin$ gcc -Wall -std=c99 -D_POSIX_C_SOURCE=200112L strings2.c
doolin$ a.out
strcpy(s1, "Hallo ") = "Hallo ", s1 = "Hallo "
strlen(s1) = 6
strcat(s1, s2) = "Hallo Welt!", s1 = "Hallo Welt!"
strcmp(s1, s3) = 32
strcasecmp(s1, s3) = 0
strspn(s1, s4) = 5
strcspn(s1, s5) = 5
strchr(s1, 'e') = "elt!"
strpbrk(s1, s5) = " Welt!"
strstr(s1, s6) = "lo Welt!"
doolin$
```

Anmerkung: Entsprechend IEEE Std 1003.1 (POSIX) wird *strcasecmp* innerhalb von <*strings.h*> deklariert und die anderen vorgestellten Funktionen innerhalb von <*string.h*>. Dies war allerdings früher anders und aus Kompatibilitätsgründen ist unter Solaris die gezeigte Option „-D_POSIX_C_SOURCE=200112L“, die ein entsprechendes Symbol für den Präprozessor definiert, zu verwenden, damit <*strings.h*> und <*string.h*> dem Standard entsprechen. Andernfalls kommt es zu einer Warnung durch den gcc, dass *strcasecmp()* nicht deklariert sei. Unter Linux oder Cygwin kann diese Option wegfallen.

Folgendes Beispiel demonstriert die Verwendung der Funktion *strtok()*:

Programm 7.15: Verwendung der Funktion *strtok()* (*strings3.c*)

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char text[] = "Das␣macht␣doch␣riesigen␣Spass,␣oder?!";
6      char trenner[] = "␣,?!";
7      char* token;
8      int len = strlen(text); /* Laenge von text fuer spaeter sichern */
9
10     /* String in Abschnitte (tokens) zerlegen */
11
12     token = strtok(text, trenner); /* erster Aufruf: s1 = text */
13     while (token != 0) { /* oder auch nur: while (token) */
14         puts(token);
15         token = strtok(0, trenner); /* weitere Aufrufe: s1 = 0 */
16     }
17
18     /* Inhalt des Vektors text nach strtok() ausgeben */
19     puts("text[]␣=␣");
20     for (int i = 0; i < len; i++) {
21         if (text[i]) {
22             putchar(text[i]);
23         } else {
24             putchar('\\'); putchar('0');
25         }
26     }
27     putchar('\n');
28 }
```

```
doolin$ gcc -Wall -std=c99 strings3.c
doolin$ a.out
Das
macht
doch
riesigen
Spass
oder
text[] =
Das\0macht\0doch\0riesigen\0Spass\0 oder\0!
doolin$
```

Anmerkung: Statt *strtok()* kann auch *strtok_r()* verwendet werden, das einen weiteren

Parameter benötigt und dafür ohne eine interne Variable auskommt, die sich den Status aus dem letzten Aufruf merkt.

7.4.2 Verbundtypen

⟨structure-type-specifier⟩ ⟶ **struct** [⟨identifier⟩] „{“
⟨struct-declaration-list⟩ „}“
⟶ **struct** ⟨identifier⟩
⟨struct-declaration-list⟩ ⟶ ⟨struct-declaration⟩
⟶ ⟨struct-declaration-list⟩ ⟨struct-declaration⟩
⟨struct-declaration⟩ ⟶ ⟨specifier-qualifier-list⟩
⟨struct-declarator-list⟩ „;“
⟨specifier-qualifier-list⟩ ⟶ ⟨type-specifier⟩ [⟨specifier-qualifier-list⟩]
⟶ ⟨type-qualifier⟩ [⟨specifier-qualifier-list⟩]
⟨struct-declarator-list⟩ ⟶ ⟨struct-declarator⟩
⟶ ⟨struct-declarator-list⟩ „,“ ⟨struct-declarator⟩
⟨struct-declarator⟩ ⟶ ⟨declarator⟩
⟶ [⟨declarator⟩] „:“ ⟨constant-expression⟩

7.4.2.1 Einfache Verbundtypen

Ein *Verbundtyp* (in C auch Struktur genannt) fasst mehrere *Elemente* zu einem Datentyp zusammen. Im Gegensatz zu Vektoren können die Elemente *unterschiedlichen* Typs sein.

Mit dem Schlüsselwort **struct** kann ein Verbundtyp wie folgt deklariert werden:

```
struct datum {
    short tag, monat, jahr;
};
```

Hier ist *datum* ist der *Name des Verbundtyps*, der allerdings nur in Verbindung mit dem Schlüsselwort **struct** erkannt wird. Der hier deklarierte Verbundtyp repräsentiert – wie der Name schon andeutet – ein Datum. Jede Variable dieses Verbundtyps besteht aus drei ganzzahligen Komponenten, dem Tag, dem Monat und dem Jahr.

Eine Variable *geburtsdatum* dieses Verbundtyps kann danach wie folgt angelegt werden:

```
struct datum geburtsdatum;
```

Analog zu Aufzählungen lassen sich auch Variablen für namenlose Verbundtypen anlegen:

```
struct {
    short tag, monat, jahr;
} my_geburtsdatum;
```

Ohne den Namen fehlt jedoch die Möglichkeit, weitere Variablen dieses Typs zu deklarieren oder den Typnamen in einer Typkonvertierung oder einem Aggregat zu spezifizieren.

Variablen eines Verbund-Typs können bereits bei ihrer Definition initialisiert werden:

```
struct datum geburtsdatum = {3, 5, 1978};
```

Alternativ kann auch der Wert eines Verbundtyps innerhalb eines Ausdrucks mit Hilfe eines Aggregats konstruiert werden:

```
struct datum geburtsdatum;
geburtsdatum = (struct datum) {3, 5, 1978};
```

Auf die *Komponenten* eines Verbundtyps kann wie folgt zugegriffen werden:

```
struct datum gebdat = ...;

printf("%hd.%hd.%hd", gebdat.tag, gebdat.monat, gebdat.jahr);

struct datum *p = ...;

/* Zeiger zuerst dereferenzieren ... */
printf("%hd.%hd.%hd", (*p).tag, (*p).monat, (*p).jahr);
/* ... oder einfacher (und aequivalent) mit -> ... */
printf("%hd.%hd.%hd", p->tag, p->monat, p->jahr);
```

Vorsicht: Aufgrund der *Vorrang-Regeln* bei Operatoren ist $*p.tag$ äquivalent zu $*(p.tag)$ und nicht zu $(*p).tag$.

Anmerkung: Das Ausgabeformat *%hd* passt genau zu dem verwendeten Datentyp **short**.

7.4.2.2 Verschachtelte Verbundtypen

Die Elemente eines Verbundtyps können (beinahe) beliebigen Typs sein. Insbesondere ist es auch möglich, Verbundtypen ineinander zu verschachteln:

```
struct person {
    char* name;
    char* vorname;
    struct datum geburtsdatum;
};
```

Wenn dann eine Variable *p* als **struct** *person p* vereinbart ist, dann kann wie folgt auf die Elemente zugegriffen werden:

```
p.name = ...;
p.vorname = ...;
p.geburtsdatum.tag = ...;
p.geburtsdatum.monat = ...;
p.geburtsdatum.jahr = ....;
```

Es wäre auch möglich gewesen, einen unbenannten Verbundtyp einzubetten:

```
struct my_person {
    char* name;
    char* vorname;
    struct {
        short tag, monat, jahr;
    } geburtsdatum;
};
```

7.4.2.3 Rekursive Verbundtypen

Zeiger auf Verbundtypen können bereits verwendet werden, auch wenn die zugehörigen Strukturen noch nicht (bzw. nicht vollständig) deklariert sind. Dies ist eine der wenigen

Ausnahmen von der Regel, dass alles, was verwendet wird, vorher deklariert werden muss. Diese Ausnahme ermöglicht rekursive bzw. zyklisch verkettete Strukturen:

Programm 7.16: Rekursive Strukturen (*struct.c*)

```
1  struct s {
2      /* ... */
3      struct s* p; /* Zeiger auf die eigene Struktur ist ok */
4      /* struct s elem; */ /* nicht erlaubt! */
5  };
6
7  struct s1 {
8      /* ... */
9      struct s2* p; /* Zeiger als Vorwaertsverweis ist ok */
10     /* struct s2 elem;*/ /* nicht erlaubt! */
11 };
12
13 struct s2 {
14     /* ... */
15     struct s1* p; /* Zeiger als Rueckwaertsverweis ok */
16     struct s1 elem; /* ok */
17 };
```

7.4.2.4 Zuweisung von Verbundtypen

Variablen des gleichen Verbundtyps können einander auch zugewiesen werden. Dabei werden die einzelnen *Elemente* der Struktur jeweils *kopiert*. Dies wird durch folgendes Beispiel veranschaulicht:

Programm 7.17: Zuweisung von Verbundtypen (*struct1.c*)

```
1  #include <stdio.h>
2
3  struct datum {
4      short tag, monat, jahr;
5  };
6
7  int main() {
8      struct datum vorl_beginn = {16, 10, 2006};
9      struct datum ueb_beginn = {24, 10, 2006};
10
11     printf("vorher:␣%hd.%hd.%hd\n",
12         vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);
13
14     vorl_beginn = ueb_beginn;
15
16     printf("nachher:␣%hd.%hd.%hd\n",
17         vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);
18 }
```

```
doolin$ gcc -Wall -std=c99 struct1.c
doolin$ a.out
vorher: 16.10.2006
nachher: 24.10.2006
doolin$
```

Anmerkung: Eine gewisse Vorsicht ist geboten, wenn Verbundtypen mit Zeigerwerten auf diese Weise kopiert werden, weil dann zwar der Inhalt des Verbunds kopiert wird, aber nicht die über die Zeiger referenzierten Datenstrukturen.

7.4.2.5 Verbundtypen als Funktionsargumente

Verbunde können als Werteparameter übergeben werden oder – durch die Verwendung von Zeigern – auch als Referenz-Parameter verwendet werden:

Programm 7.18: Verbundtypen als Funktionsargumente (*struct2.c*)

```
1  #include <stdio.h>
2
3  struct datum {
4      short tag, monat, jahr;
5  };
6
7  /* Werteparameter-Semantik */
8  void ausgabe1(struct datum d) {
9      printf("%hd.%hd.%hd\n", d.tag, d.monat, d.jahr);
10 }
11
12 /* Referenzparameter-Semantik (wirkt sich hier nicht aus) */
13 void ausgabe2(struct datum* d) {
14     printf("%hd.%hd.%hd\n", d->tag, d->monat, d->jahr);
15 }
16
17 /* Werteparameter-Semantik: Verbund des Aufrufers aendert sich nicht */
18 void setJahr1(struct datum d, int jahr) {
19     d.jahr = jahr;
20 }
21
22 /* Referenzparameter-Semantik erlaubt die Aenderung */
23 void setJahr2(struct datum* d, int jahr) {
24     d->jahr = jahr;
25 }
26
27 int main() {
28     struct datum start = {16, 10, 2006};
29
30     ausgabe1(start);
31     setJahr1(start, 2007); /* keine Aenderung! */
32     ausgabe2(&start); /* aequivalent zu ausgabe1(...) */
33     setJahr2(&start, 2007); /* setzt das Jahr auf 2004 */
34     ausgabe1(start);
35 }
```

```
doolin$ gcc -Wall -std=c99 struct2.c
doolin$ a.out
16.10.2006
16.10.2006
16.10.2007
doolin$
```

7.4.2.6 Verbunde als Ergebnis von Funktionen

Funktionen können als Ergebnistyp auch einen Verbundtyp verwenden. Hingegen ist Vorsicht angebracht, wenn Zeiger auf Verbunde zurückgegeben werden:

Programm 7.19: Verbunde als Ergebnis von Funktionen (*struct3.c*)

```
1  #include <stdio.h>
2
3  struct datum {
4      short tag, monat, jahr;
5  };
6
7  void ausgabe(struct datum d) {
8      printf("%hd.%hd.%hd\n", d.tag, d.monat, d.jahr);
9  }
10
11 struct datum init1() {
12     struct datum d = {1, 1, 1900};
13     return d; /* ok, denn es wird eine Kopie erzeugt */
14 }
15
16 struct datum* init2() {
17     struct datum d = {1, 1, 1900};
18     return &d; /* nicht zulaessig, da Zeiger auf lokale Variable! */
19 }
20
21 int main() {
22     struct datum d;
23     struct datum* p;
24
25     d = init1();
26     ausgabe(d);
27
28     p = init2(); /* Zeiger auf Variable, die nicht mehr existiert! */
29     ausgabe(*p); /* wenn's klappt ... dann ist das Glueck! */
30     ausgabe(*p); /* sollte eigentliche dasselbe ausgeben :-( */
31 }
```

```
doolin$ gcc -Wall -std=c99 struct3.c
struct3.c: In function `init2':
struct3.c:18: warning: function returns address of local variable
doolin$ a.out
1.1.1900
1.1.1900
0.9.1900
doolin$
```

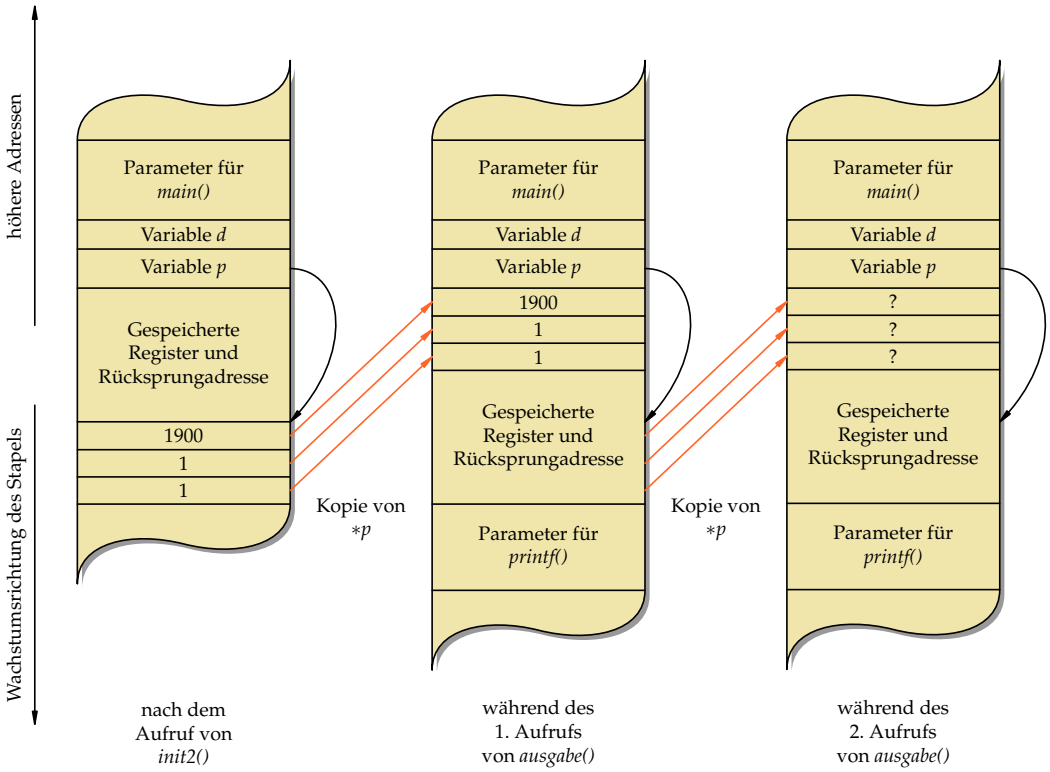

Abbildung 7.5: Funktionsaufrufe und lokale Variablen

Erklärung: Die Variable *d* in der Funktion *init2()* ist eine lokale Variable, die auf dem Laufzeit-Stapel für Funktionen (im Englischen *runtime stack* genannt) lebt. Sie existiert nur solange diese Funktion ausgeführt wird. Danach wird dieser Speicherplatz evtl. anderweitig verwendet (siehe Abbildung 7.5).

Nach dem Aufruf von *init2()* ist zwar die Lebenszeit der Daten hinter *p* zwar vorbei, aber sie liegen typischerweise immer noch intakt auf dem Laufzeit-Stapel. Entsprechend werden beim ersten Aufruf von *ausgabe()* die Daten noch korrekt kopiert. Allerdings werden die von *p* referenzierten Daten dann während des ersten Aufrufs von *ausgabe()* überschrieben. Deswegen werden beim folgenden zweiten Aufruf von *ausgabe()* vollkommen undefinierte Werte bei der Parameterübergabe kopiert.

7.4.2.7 Variante Verbünde

⟨union-type-specifier⟩ ⟶ **union** [⟨identifier⟩] „{“
⟨struct-declaration-list⟩ „}“
⟶ **union** ⟨identifier⟩

Syntaktisch gleichen variante Verbünde den regulären Verbünden – es wird nur das Schlüsselwort **union** an Stelle von **struct** verwendet.

Der Unterschied zwischen regulären Verbünden und varianten Verbünden ist, dass die Komponenten eines regulären Verbunds an *verschiedenen Stellen* im Speicher stehen, wohingegen die Komponenten varianter Verbünde an *derselben Stelle* im Speicher stehen und sich somit überlagern. Entsprechend können bei einem regulären Verbund alle Komponenten gleichzeitig in getrennter Weise verwendet werden, wohingegen bei einem varianten Verbund nur eine der Varianten und damit nur eine Komponente jeweils zu verwenden ist.

Es gibt zwei Gründe, die für die Verwendung eines varianten Verbunds sprechen können:

- Variante Verbünde sparen Speicherplatz ein, wenn immer nur eine Variante benötigt wird. In diesem Falle muss (ausserhalb des varianten Verbunds) ein Status verwaltet werden, der mitteilt, welche Variante gerade in Benutzung ist.
- Durch variante Verbünde sind zwei (oder mehr) Sichten durch verschiedene Datentypen auf ein gemeinsames Stück Speicher möglich, ohne dass hierfür jeweils umständliche Konvertierungen notwendig wären. Allerdings ist hier Vorsicht geboten, da dies sehr von der jeweiligen Plattform abhängen kann.

In folgendem Beispielprogramm wird ein varianter Verbund zur Repräsentation einer IP-Adresse verwendet. Daher enthält dieser variante Verbund eine ganze Zahl und einen 4-elementigen Byte-Vektor, die beide die gleiche Stelle im Speicher referenzieren. Somit kann je nach Wunsch die IP-Adresse als ganze Zahl oder als Sequenz von vier Bytes betrachtet werden.

Programm 7.20: Verwendung eines varianten Verbunds (*union.c*)

```
1  #include <stdio.h>
2
3  /* Repraesentation von IP-Adressen */
4  union IPAddr {
5      unsigned int ip;
6      unsigned char b[4];
7  };
8
9  int main() {
10     union IPAddr a;
11
12     a.ip = 0x863c4205; /* bel. IP-Adresse in int-Darst. zuweisen */
13
14     /* Zugriff auf a ueber die Komponente ip */
15     printf("%u␣[%x]\n", a.ip, a.ip);
16
```

```
17      /* Zugriff auf a ueber die Komponente b */
18      printf("%hhu.%hhu.%hhu.%hhu␣", a.b[0], a.b[1], a.b[2], a.b[3]);
19      printf("[%02hhx.%02hhx.%02hhx.%02hhx]\n",
20          a.b[0], a.b[1], a.b[2], a.b[3]);
21
22      puts("");
23      printf("Speicherplatzbedarf:␣%zd\n", sizeof(a));
24
25      puts(""); /* Anordung im Speicher analysieren */
26      puts("Position␣im␣Speicher:");
27
28      printf("a:␣␣␣␣%p\n", &a);
29
30      printf("ip:␣␣␣%p\n", &a.ip);
31
32      printf("b[0]:␣%p\n", &a.b[0]);
33      printf("b[1]:␣%p\n", &a.b[1]);
34      printf("b[2]:␣%p\n", &a.b[2]);
35      printf("b[3]:␣%p\n", &a.b[3]);
36  }
```

Ausführung auf einer *big-endian*-Maschine:

```
doolin$ uname -m
sun4u
doolin$ gcc -Wall -std=c99 union.c
doolin$ a.out
2252096005 [863c4205]
134.60.66.5 [86.3c.42.05]

Speicherplatzbedarf: 4

Position im Speicher:
a:    ffbff464
ip:   ffbff464
b[0]: ffbff464
b[1]: ffbff465
b[2]: ffbff466
b[3]: ffbff467
doolin$
```

Ausführung auf einer *little-endian*-Maschine:

```
zeus$ uname -m
x86_64
zeus$ gcc -Wall -std=c99 union.c
zeus$ a.out
2252096005 [863c4205]
5.66.60.134 [05.42.3c.86]

Speicherplatzbedarf: 4

Position im Speicher:
a:    0x7fff0034a430
ip:   0x7fff0034a430
b[0]: 0x7fff0034a430
b[1]: 0x7fff0034a431
b[2]: 0x7fff0034a432
b[3]: 0x7fff0034a433
zeus$
```

Hinweis: Wie sich dem Beispiel entnehmen lässt, können die beiden Sichtweisen des gleichen varianten Verbunds auf unterschiedlichen Plattformen sehr verschieden ausfallen. Entsprechend der unterschiedlichen Repräsentierung einer ganzen Zahl im Speicher (*big-endian* vs. *little-endian*), ergeben sich hier unterschiedliche Byte-Reihenfolgen.

7.5 Typdefinitionen

⟨typedef-name⟩ ⟶ ⟨identifier⟩
⟨storage-class-specifier⟩ ⟶ **typedef**
⟶ **extern**
⟶ **static**
⟶ **auto**
⟶ **register**

Einer Deklaration kann das Schlüsselwort **typedef** vorausgehen. Dann wird der Name, der sonst ein Variablenname geworden wäre, stattdessen zu einem neu definierten Typnamen. Dieser Typname kann anschließend überall dort verwendet werden, wo die Angabe eines ⟨type-specifier⟩ zulässig ist.

Ein erstes Beispiel demonstriert dies:

typedef int *Laenge*; /* *Vereinbarung des eigenen Typnames "Laenge"* */

/* ... */

Laenge i, j; /* *Vereinbarung der Variablen i und j vom Typ Laenge* */

In obigem Beispiel ist *Laenge* zu einem Synonym für **int** geworden. Damit sind **int** *i, j*; und *Laenge i, j*; äquivalente Vereinbarungen. Hier bieten Typdefinitionen die Flexibilität, einen Typ an einer zentralen Stelle zu vereinbaren, um ihn dann bequem für das gesamte Programm verändern zu können. Das ist insbesondere sinnvoll bei der Verwendung numerischer Datentypen. Synonyme können auch zur Lesbarkeit beitragen, wenn besonders „sprechende“ Namen verwendet werden.

Typdefinitionen ermöglichen es, komplexere Typen in einen ⟨type-specifier⟩ zu integrieren, die sich sonst nur im Rahmen einer ⟨declaration⟩ formulieren ließen. Das betrifft insbesondere Zeiger und Vektoren wie folgendes Beispiel zeigt:

```
typedef char* CharPointer;
typedef int TenIntegers[10];
CharPointer cp1, cp2; // beide sind vom Typ char*
char* cp3, cp4; // cp4 hat nur den Typ char!
TenIntegers a, b; // beides sind Vektoren
int c[10], d; // d hat nur den Typ int!
```

Bei Verbünden werden ebenfalls Typdefinitionen verwendet, um anschließend nur den Namen ohne das Schlüsselwort **struct** verwenden zu können:

```
typedef struct datum {
    short tag, monat, jahr;
} datum;
datum geburtsdatum; // aequivalent zu struct datum geburtsdatum
datum heute, morgen;
```

Die Verwendung von Typnamen aus Typdefinitionen bleibt – abgesehen von den syntaktischen Unterschieden – äquivalent zur Verwendung des ursprünglichen Datentyps. Entsprechend entsteht durch eine Typdefinition kein neuer Typ, der nicht mehr mit dem alten Typ kompatibel wäre.

7.6 Komplexe Deklarationen

Durch die unglückliche Aufteilung von Typ-Spezifikationen in ⟨type-specifier⟩ (links stehend) und ⟨declarator⟩ (rechts stehend, sich um den Namen anordnend), werden komplexere Deklarationen rasch unübersichtlich. Die Motivation für diese Syntax kam wohl aus dem Wunsch, dass die Deklaration einer Variablen ihrer Verwendung gleichen solle. Entsprechend hilft es, sich bei komplexeren Deklarationen die Vorränge und Assoziativitäten der zugehörigen Operatoren in Erinnerung zu rufen (siehe Abschnitt 6.2.1).

Sei folgendes Beispiel gegeben:

```
char* x[10];
```

Der Vorrangtabelle lässt sich entnehmen, dass der []-Operator einen höheren Vorrang (16) im Vergleich zum ∗-Operator (15) hat. Entsprechend handelt es sich bei x um einen Vektor mit 10 Elementen mit dem Elementtyp Zeiger auf **char**. Im einzelnen:

x[10]	Vektor mit 10 Elementen
$*\,x$[10]	Vektor mit 10 Zeigern
char$*\,x$[10]	Vektor mit 10 Zeigern auf Zeichen

Ein weiteres Beispiel:

```
int* (*(*x)())[5];
```

Die Analyse beginnt hier wieder beim Variablennamen x in der Mitte der Deklaration:

$*x$	ein Zeiger
$(*x)()$	ein Zeiger auf eine Funktion
$*(*x)()$	ein Zeiger auf eine Funktion, die einen Zeiger liefert
$(*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf einen 5-elementigen Vektor liefert
$*\,(*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf einen 5-elementigen Vektor aus Zeigern liefert
int$*\,(*(*x)())[5]$	ein Zeiger auf eine Funktion, die einen Zeiger auf einen 5-elementigen Vektor aus Zeigern auf **int** liefert

An zwei Stellen waren hier Vorränge relevant: Im zweiten Schritt war wesentlich, dass Funktionsaufrufe (Vorrangstufe 16) Vorrang haben vor der Dereferenzierung (Vorrangstufe 15) und im vierten Schritt hatte die Indizierung (Vorrangstufe 16) ebenfalls Vorrang vor der Dereferenzierung. Transparenter wird dies durch einen stufenweisen Aufbau mit Typdefinitionen:

typedef int∗ *intp*; *// intp = Zeiger auf int*
typedef *intp intpa*[5]; *// intpa = Vektor mit 5 Zeigern auf int*
typedef *intpa f*(); *// f = Funktion, die intpa liefert*
typedef *f*∗ *fp*; *// Zeiger auf eine Funktion*
fp x;

Zusammenfassend:

- [] und () haben einen höheren Rang als ∗.
- [] und () assoziieren von *links nach rechts*, während ∗ von *rechts nach links* gruppiert.

Klammern können verwendet werden, um die Operatoren anders zu gruppieren und damit den Typ entsprechend zu verändern. Beispiel:

int (∗*x*[10])();

Hier ist *x* ein 10-elementiger Vektor von Zeigern auf Funktionen mit Rückgabewerten des Typs **int**. Im einzelnen:

x[10]	*x* als 10-elementiger Vektor
(∗*x*[10])	*x* als 10-elementiger Vektor von Zeigern
(∗*x*[10])()	*x* als 10-elementiger Vektor von Zeigern auf Funktionen
int (∗*x*[10])()	*x* als 10-elementiger Vektor von Zeigern auf Funktionen, mit Rückgabewerten des Typs **int**.

Ohne die Klammern sähe es anders aus. In diesem Falle wäre *x* ein 10-elementiger Vektor von Funktionen, mit Rückgabewerten des Typs Zeiger auf **int**. Diese Deklaration wäre jedoch nicht zulässig, da Funktionen als Element-Typ nicht zulässig sind.

Ein weiteres Beispiel, das *x* als einen Zeiger auf einen Vektor von Zeigern auf Funktionen deklariert, deren Rückgabewerte des Typs Zeiger auf Vektoren des Verbundtyps **struct** *S* sind:

struct *S* (∗(∗(∗*x*)[])())[];

Im einzelnen:

(∗*x*)	*x* als Zeiger
(∗*x*)[]	*x* als Zeiger auf einen Vektor
(∗(∗*x*)[])	*x* als Zeiger auf einen Vektor mit Zeigern
(∗(∗*x*)[])()	*x* als Zeiger auf einen Vektor mit Zeigern auf Funktionen
(∗(∗(∗*x*)[])())	*x* als Zeiger auf einen Vektor mit Zeigern auf Funktionen, deren Rückgabewerte Zeiger sind
(∗(∗(∗*x*)[])())[]	*x* als Zeiger auf einen Vektor mit Zeigern auf Funktionen, deren Rückgabewerte Zeiger auf Vektoren sind
struct *S* (∗(∗(∗*x*)[])())[]	*x* als Zeiger auf einen Vektor mit Zeigern auf Funktionen, deren Rückgabewerte Zeiger auf Vektoren des Verbundtyps **struct** *S* sind

Beispiele für unzulässige Deklarationen:

int *af*[]()	Vektor von Funktionen, die Rückgabewerte des Typs **int** liefern
int *fa*()[]	Funktion, die einen Vektor von ganzen Zahlen liefert; hier wäre **int**∗ *fa*() akzeptabel gewesen
int *ff*()()	Funktion, die eine Funktion liefert, welche wiederum **int** liefert

Kapitel 8

Funktionen

⟨function-definition⟩	⟶	⟨function-def-specifier⟩
		⟨compound-statement⟩
⟨function-def-specifier⟩	⟶	[⟨declaration-specifiers⟩] ⟨declarator⟩
		[⟨declaration-list⟩]
⟨function-specifier⟩	⟶	**inline**
⟨function-declarator⟩	⟶	⟨direct-declarator⟩
		„(“ ⟨parameter-type-list⟩ „)“
	⟶	⟨direct-declarator⟩ „(“ [⟨identifier-list⟩] „)“
⟨identifier-list⟩	⟶	⟨identifier⟩
	⟶	⟨parameter-list⟩ „,“ ⟨identifier⟩
⟨parameter-type-list⟩	⟶	⟨parameter-list⟩
	⟶	⟨parameter-list⟩ „,“ „...“
⟨parameter-list⟩	⟶	⟨parameter-declaration⟩
	⟶	⟨parameter-list⟩ „,“ ⟨parameter-declaration⟩
⟨parameter-declaration⟩	⟶	⟨declaration-specifiers⟩ ⟨declarator⟩
	⟶	⟨declaration-specifiers⟩
		[⟨abstract-declarator⟩]
⟨abstract-declarator⟩	⟶	⟨pointer⟩
	⟶	[⟨pointer⟩] ⟨direct-abstract-declarator⟩
⟨direct-abstract-declarator⟩	⟶	„(“ ⟨abstract-declarator⟩ „)“
	⟶	[⟨direct-abstract-declarator⟩]
		„[“ [⟨constant-expression⟩] „]“

- Die Definition einer Funktion besteht aus dem *Typ des Rückgabewertes*, dem *Namen der Funktion*, einer möglicherweise leeren *Liste formaler Argumente* (d. h. jeweils Typ und Argumentname) und dem *Anweisungsblock*.
- Funktionen sind immer *global*, d. h. sie dürfen nicht geschachtelt werden (im Gegensatz zu anderen Programmiersprachen wie etwa Pascal oder Oberon). Es gibt aber

auch Implementierungen (wie etwa der gcc), die dies dennoch in Abweichung vom Standard erlauben. Das ist dann aber nicht mehr portabel.

- Parameter sind in C immer Werteparameter (*call by value*). Allerdings ist hier bei Vektoren die besondere Semantik in C zu beachten, bei der der Name eines Vektors für einen Zeiger steht.

- Wenn die Funktion keine Parameter hat, sollte in der Parameterliste explizit **void** angegeben werden (ohne einen zugehörigen Namen).

- Die Angabe des Typs des Rückgabewertes ist (beginnend mit C99) zwingend. Bei Funktionen, die keinen Wert zurückliefern, ist **void** als Typ des Rückgabewertes anzugeben. Vektoren oder Funktionen können nicht zurückgeliefert werden – stattdessen sind entsprechende Zeigertypen zu verwenden.

- Der *Name einer Funktion* ist zugleich ein *Zeiger* auf diese Funktion. Weil dies in traditionellem C noch nicht galt, ist es auch zulässig, den Adress-Operator & zu verwenden.

Das Programm 8.1 berechnet eine der Fibonacci-Zahlen entsprechend der rekursiven Definition:

Programm 8.1: Rekursive Berechnung der Fibonacci-Zahlen (*fibonacci.c*)

```
1  #include <assert.h>
2  #include <stdio.h>
3
4  /* Berechnung der n-ten Fibonacci-Zahl:
5      F_0 = 1
6      F_1 = 1
7      F_n = F_(n-1) + F_(n-2) fuer n > 1
8  */
9  int fib(int n) {
10     assert(n >= 0);
11     if (n == 0 || n == 1) {
12         return 1;
13     } else {
14         return fib(n-1) + fib(n-2);
15     }
16 }
17
18 int main() {
19     for (int i = 0; i < 10; ++i) {
20         printf("fib(%d) = %d\n", i, fib(i));
21     }
22 }
```

Mit **return** wird die aktuelle Funktion beendet. Der Wert des angegebenen Ausdrucks ist der *Rückgabewert* der Funktion und somit der Wert des Funktionsaufrufs. Ein **return** innerhalb von *main()* hat die *Terminierung* des Programms zur Folge. Bei *main()* erfolgt aber (beginnend mit C99) ein **return** 0; implizit, wenn das Ende der Funktion erreicht wird.

8.1 Umsetzung von Referenzparametern (*call by reference*)

Die Parameterübergabe erfolgt in C – wie bereits erwähnt – über Werteparameter (*call by value*). Jedoch lässt sich die Semantik von Referenzparametern sehr einfach mit Hilfe von Zeigern emulieren, wobei dies jedoch *nicht transparent* ist, denn Referenzparameter müssen – infolge der Emulation mit Zeigern – syntaktisch *anders gehandhabt* werden als Werte-Parameter.

Programm 8.2 enthält eine Funktion *swap()* zum Vertauschen der Werte zweier ganzzahliger Variablen. Hierbei wird durch die Verwendung von Zeigern die Semantik von Referenzparametern emuliert:

Programm 8.2: Vertauschen der Werte zweier Variablen (*swap.c*)

```
1  #include <stdio.h>
2
3  /* Vertausche die Werte der beiden Variablen,
4     worauf die Zeiger a und b zeigen
5  */
6  void swap(int* a, int* b) {
7     int tmp; /* temp. Variable zum Vertauschen */
8     tmp = *a; *a = *b; *b = tmp;
9  }
10
11 int main() {
12    int x = 1, y = 2;
13
14    printf("x␣=␣%d,␣y␣=␣%d\n", x, y);
15
16    /* Zeiger auf x und y als Referenz uebergeben */
17    swap(&x, &y);
18
19    printf("x␣=␣%d,␣y␣=␣%d\n", x, y);
20 }
```

Da bei jedem Zugriff auf die Werte der referenzierten Parameter Derefenzierungen explizit erfolgen müssen, bleiben Referenz-Parameter in C umständlich und eben nicht transparent. In C++ wurden aus diesem Grunde Referenz-Typen eingeführt.

Obiges Beispiel zeigt auch, wie *Prozeduren* in C realisiert werden können. Der Spezial-Typ **void** zeigt dabei an, dass es *keinen Rückgabewert* gibt.

8.2 Vorab-Deklarationen von Funktionen

Traditionellerweise erlaubte C den Verzicht auf Funktionsdeklarationen, so dass auch Funktionen verwendet werden konnten, ohne dass der Übersetzer zuvor einen Hinweis darauf fand, wie die Parameter deklariert sind oder der Rückgabetyp aussieht. Der Übersetzer arbeitete in diesen Fällen mit impliziten Annahmen, die dann später nicht widerlegt werden durften. Dies sollte jedoch bei standard-konformen C-Programmen nicht mehr versucht werden.

Stattdessen sollte jede Funktion vor ihrer Benutzung entweder definiert oder zumindest deklariert sein. Bei den Standard-Funktionen aus der Bibliothek gibt es hierfür die zugehörigen mit **#include** einzukopierenden sogenannten Header-Dateien.

Eine Vorab-Deklaration einer Funktion bestand traditionellerweise nur aus dem Namen der Funktion und der Spezifikation des Typs für die Rückgabewerte. Das sah etwa so aus:

```
int fib(); /* Deklaration von fib */

/* ... */
int f = fib(5); /* Parameterliste ist noch unbekannt */

/* ... */

/* Definition von fib */
int fib(int n) { /* ... */ }
```

Dieser traditionelle Stil gibt dem Übersetzer jedoch keine Gelegenheit, die aktuellen Parameter mit der formalen Parameterliste zu vergleichen. Insbesondere ist es dem Übersetzer dann auch nicht möglich, die aktuellen Parameter jeweils passend zu dem zugehörigen formalen Parametertyp implizit zu konvertieren.

Sofern der Übersetzer Gelegenheit hat, implizite Annahmen mit zu spät kommenden Definitionen zu vergleichen, ist mit entsprechenden Fehlern oder Warnungen zu rechnen, wie folgendes Beispiel zeigt:

Programm 8.3: Konflikt zwischen impliziter Deklaration und expliziter Definition einer Funktion (*badf.c*)

```
1  #include <stdio.h>
2
3  float square(); /* traditionelle Deklaration ohne Parameter */
4
5  int main() {
6      float x, y;
7      printf("x = ");
8      if (scanf("%f", &x) != 1) return 1;
9      /* Vorsicht: Parameterliste ist nicht bekannt */
10     y = square(x);
11     printf("x^2 = %f\n", y);
12 }
13
14 /* erst hier wird verraten, wie der Parameter von square aussieht */
15 float square(float x) {
16     return x * x;
17 }
```

```
dublin$ gcc -Wall -std=c99 badf.c 2>&1 | fmt -s -w60
badf.c:15: conflicting types for `square'
badf.c:15: an argument type that has a default promotion
can't match an empty parameter name list declaration
badf.c:3: previous declaration of `square'
dublin$
```

Falls allerdings die impliziten Annahmen nicht widerlegt werden, weil die explizite Definition sich in einer anderen Übersetzungseinheit befindet, dann unterbleibt die Fehlermeldung und das Resultat ist dann undefiniert:

```
dublin$ gcc -Wall -std=c99 badf1.c badf2.c
dublin$ a.out
x = 1
x^2 = 3.515625
dublin$
```

Das folgende Beispielprogramm zeigt, wie eine *Vorab-Deklaration* korrekt eingesetzt wird. Da die Funktionen *male()* und *female()* sich gegenseitig aufrufen, gibt es keine Anordnung der Funktionen, die ohne eine solche Deklaration auskommen würde:

Programm 8.4: Vorab-Deklarationen bei Funktionen (*fm.c*)

```
1  #include <stdio.h>
2
3  /* Male-Female-Folge von Douglas Hofstadter:
4      F(0) = 1
5      M(0) = 0
6      F(n) = n - M(F(n-1))
7      M(n) = n - F(M(n-1))
8  */
9
10 /* Vorab-Deklaration von male */
11 int male(int n);
12
13 int female(int n) {
14     if (n == 0) return 1;
15     return n - male(female(n - 1));
16 }
17
18 /* Definition von male */
19 int male(int n) {
20     if (n == 0) return 0;
21     return n - female(male(n - 1));
22 }
23
24 int main() {
25     puts(" i  F(i)  M(i)");
26     for (int i = 0; i < 10; ++i) {
27         printf("%2d  %4d  %4d\n", i, female(i), male(i));
28     }
29 }
```

```
dublin$ gcc -Wall -std=c99 fm.c
dublin$ a.out
 i F(i) M(i)
 0    1    0
 1    1    0
 2    2    1
 3    2    2
 4    3    2
 5    3    3
 6    4    4
 7    5    4
 8    5    5
 9    6    6
dublin$
```

8.3 Funktionszeiger

Der *Name einer Funktion* ist zugleich auch ein *Zeiger auf diese Funktion*. Doch wozu sind Zeiger auf Funktionen gut? Darauf gibt das folgende Programmbeispiel eine Antwort. Hierbei soll eine Funktion zur numerischen Integration (mittels Trapezregel) entwickelt werden. Diese Funktion soll jedoch beliebige Funktionen integrieren können. Wie ist das möglich? Ganz einfach: mit Funktionszeigern! Der Integrations-Funktion wird einfach ein Zeiger auf die zu integrierende Funktion (Integrand) mitgegeben. Somit kann eine beliebige Funktion mit demselben Typ, d. h. gleiche Anzahl und gleicher Typ der Parameter und gleicher Typ des Rückgabewertes, übergeben und integriert werden.

In Abbildung 8.1 ist das Integrationsverfahren (*Trapezregel*), das in Programm 8.5 verwendet wird, zum leichteren Verständnis illustriert.

In diesem Beispiel wird das Integral

$$\int_{x0}^{x1} h(x)dx$$

durch

$$\sum_{i=0}^{n-1} \Delta \frac{h(x0 + (i+1)\Delta) + h(x0 + i\Delta)}{2}$$

approximiert, wobei Δ für die Schrittweite steht:

$$\Delta := (x1 - x0)/n$$

Die obige Näherung lässt sich dann wie folgt umschreiben:

$$\begin{aligned} & \sum_{i=0}^{n-1} \Delta \frac{h(x0 + (i+1)\Delta) + h(x0 + i\Delta)}{2} \\ = \quad & \Delta \sum_{i=0}^{n-1} \frac{h(x0 + (i+1)\Delta) + h(x0 + i\Delta)}{2} \\ = \quad & \left(\left(\sum_{i=0}^{n} h(x0 + i\Delta) \right) - \frac{h(x0) + h(x1)}{2} \right) \Delta \end{aligned}$$

Diese letzte Formel wird in Programm 8.5 verwendet:

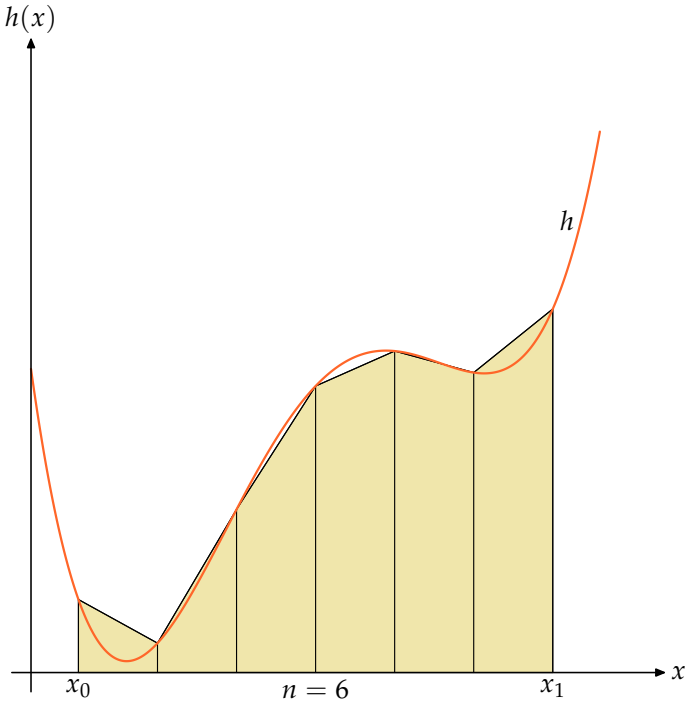

Abbildung 8.1: Integration nach der Trapezregel

Programm 8.5: Funktionszeiger: Integration nach der Trapezregel (*mathfun.c*)

```
1  #include <stdio.h>
2
3  /* Zeigertyp fuer Funktionen der folgenden Art:
4     double FunktionsName(double param); */
5  typedef double (*Function)(double);
6
7  /* Numerische Integration der Funktion f nach der Trapezregel:
8     f: zu integrierende Funktion
9     [x0,x1]: Integrationsbereich
10    n: Anzahl der Intervalle im Integrationsbereich */
11 double integrate(Function h, double x0, double x1, int n) {
12    double delta = (x1 - x0) / n;
13    double sum = 0.0;
14    double x = x0;
15    for (int i = 0; i <= n; i++, x += delta) {
16       sum += (*h)(x);
17    }
18    return (sum - ((*h)(x0) + (*h)(x1)) / 2.0) * delta;
19 }
20
21 /* zwei Beispiel-Funktionen */
22 double f(double x) {
23    return x * x;
24 }
25 double g(double x) {
```

```
26      return 2.0 ∗ x ∗ x − 3.0 ∗ x + 1.0;
27  }
28
29  int main() {
30      puts("Numerische Integration:");
31      printf("f: %lf\n", integrate(&f, 0, 1, 100));
32      printf("g: %lf\n", integrate(&g, 0, 1, 100));
33  }
```

```
dublin$ gcc -Wall -std=c99 mathfun.c
dublin$ a.out
Numerische Integration:
f: 0.333350
g: 0.166700
dublin$
```

Anmerkung: Beim Aufruf einer Funktion über einen Funktionszeiger (siehe oben $(*h)(x)$) muss zunächst der Zeiger dereferenziert werden und danach kann erst die Funktion aufgerufen werden. Da nun aber () Vorrang vor $*$ hat, ist die Klammerung notwendig!

Beginnend mit C89 darf hier auch $(*h)(x)$ zu $h(x)$ verkürzt werden. Allerdings wird dennoch die erste Variante häufig bevorzugt, weil sie sofort verdeutlicht, dass ein Funktionszeiger im Spiel ist.

Kapitel 9

Dynamische Datenstrukturen

9.1 Belegen und Freigeben von Speicher

Analog zu **new** in Java gibt es in C Funktionen, mit denen dynamisch Speicherplatz belegt bzw. wieder frei gegeben werden kann. Eine automatische Freigabe nicht mehr benötigten Speicherplatzes findet in C nicht statt und muss daher selbst organisiert werden.

In C sind diese Funktionen Bestandteil der Standard-Bibliothek, jedoch nicht der Programmiersprache im engeren Sinne (wie etwa Java). Deswegen müssen diese Funktionen unabhängig von den Datentypen der Anwendungen geschrieben werden, so dass ein allgemeiner unspezifischer Zeigertyp benötigt wird, der dann automatisch (mit oder ohne eine explizite Konvertierung) bei einer Zuweisung an andere Zeigertypen angepasst wird. Dieser Zeigertyp nennt sich in C **void***.

Ein weiteres Problem einer Bibliothekslösung sind Anforderungen, die sich aus dem dynamisch anzulegenden Datentyp aus der Anwendung ergeben. So müssen beispielsweise auf einer SPARC-Plattform 32-Bit-Zahlen bei durch 4 teilbaren Adressen untergebracht werden und 64-Bit-Gleitkommazahlen benötigen sogar durch 8 teilbare Adressen. Umgekehrt könnte eine Zeichenkette bei einer beliebigen Adresse beginnen. Da den Bibliotheksfunktionen hier jedoch verborgen bleibt, welcher Datentyp verwendet wird, muss von dem ungünstigsten Fall ausgegangen werden.

Dies sind im Einzelnen die folgenden Funktionen, die in <*stdlib.h*> deklariert sind:

void* *calloc*(*size_t nelem*, *size_t elsize*)
: Diese Funktion versucht, genügend Speicher zu belegen, so dass *nelem* Elemente der Größe *elsize* eines Vektors untergebracht werden können. Gelingt dies, wird ein Zeiger auf den Anfang dieser Speicherfläche geliefert, die zuvor vollständig mit Nullen initialisiert worden ist. Wenn dies jedoch fehlschlägt, so ist das Ergebnis der Null-Zeiger. (Diese Funktion kann auch für Datenstrukturen verwendet werden, die keine Vektoren sind. In diesem Falle ist eben $elsize = 1$.)

void* *malloc*(*size_t size*)
: Diese Funktion versucht, Speicher im Umfange von mindestens *size* Bytes zu belegen, so dass ein beliebiger Datentyp dieser Größe untergebracht werden kann. Im Erfolgsfalle wird der Zeiger auf den Anfang der belegten Speicherfläche geliefert. Falls dies nicht klappt, so wird wiederum der Null-Zeiger zurückgegeben. Im Unterschied zu *calloc()* wird die Speicherfläche nicht initialisiert und entsprechend ist damit zu rechnen, dass sich dort noch Daten aus der vorherigen Nutzung dieser Speicherfläche befinden.

void *free*(**void*** *ptr*)
: Hier muss *ptr* auf eine zuvor belegte, jedoch noch nicht freigegebene Speicherfläche

verweisen. Dann gibt *free* diese Fläche zur andersweitigen Nutzung wieder frei.

void* *realloc*(**void*** *ptr*, *size_t size*)
Hier ist *ptr* ein Zeiger, der auf bereits belegten Speicher verweist, der von einem dieser Funktionen zuvor zurückgegeben wurde. Die Funktion *realloc* bemüht sich dann um eine Anpassung der benötigten Speicherfläche an die angegebene Größe *size*, die sowohl größer als auch kleiner als die vorherige Größe sein darf. Falls *realloc* den Wunsch erfüllen kann, besitzt es die Freiheit, den Inhalt der zuvor belegten Speicherfläche (maximal *size* Bytes) an einen neuen Ort umzukopieren. Entsprechend liefert *realloc* im Erfolgsfalle den aktuellen Zeiger auf die Speicherfläche zurück, der sich möglicherweise im Vergleich zu *ptr* geändert hat. Wenn dies nicht klappt, bleibt alles unverändert und das Ergebnis ist der Null-Zeiger.

Die Funktion *realloc()* ist eine Verallgemeinerung der anderen Funktionen: Falls *ptr* der Null-Zeiger ist, entspricht *realloc()* der Funktion *malloc()*. Falls $size = 0$, dann entspricht dies der Funktion *free()*.

Folgendes Beispiel liest beliebig viele ganze Zahlen von der Standardeingabe und gibt sie alle in umgekehrter Reihenfolge wieder aus:

Programm 9.1: Lineare Listen in C (*reverse.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* lineare Liste ganzer Zahlen */
5  typedef struct element {
6      int i;
7      struct element* next;
8  } element;
9
10 int main() {
11     element* head = 0;
12     int i;
13
14     /* Zahlen einlesen und in der Liste
15        in umgekehrter Reihenfolge ablegen */
16     while ((scanf("%d", &i)) == 1) {
17         element* last = (element*) calloc(1, sizeof(element));
18         if (last == 0) {
19             fprintf(stderr, "out of memory!\n"); exit(1);
20         }
21         last->i = i; last->next = head; head = last;
22     }
23     /* Zahlen aus der Liste wieder ausgeben */
24     while (head != 0) {
25         printf("%d\n", head->i);
26         head = head->next;
27     }
28 }
```

```
dublin$ gcc -Wall -std=c99 reverse.c
dublin$ echo 1 2 3 | a.out
3
2
1
dublin$
```

Anmerkungen: Hier wurde der Zeiger des Typs **void***, den *calloc()* zurückliefert explizit in den gewünschten Zeigertyp konvertiert. Diese explizite Konvertierung könnte auch wegfallen, da diese dann auch implizit korrekt vorgenommen wird. Es ist nur aus Gründen der verbesserten Lesbarkeit sinnvoll, den Zeigertyp an dieser Stelle zu nennen.

Keinesfalls sollte der Test auf den Nullzeiger unterbleiben. Wenn dies zu umständlich erscheint, dann kann eine eigene Funktion zwischengeschaltet werden:

```
    void* my_calloc(size_t nelem, size_t elsize) {
        void* ptr = calloc(nelem, elsize);
        if (ptr) return ptr; /* alles OK */
        /* Fehlerbehandlung: */
        fprintf(stderr, öut of memory −− aborting!\n");
␣␣␣␣␣␣/*␣Termination␣mit␣core␣dump␣*/
␣␣␣␣␣␣abort();
␣␣␣}
```

9.2 Der Adressraum eines Programms

Wenn ein Prozess unter UNIX startet, wird zunächst nur Speicherplatz für den Programmtext (also den Maschinen-Code), die globalen Variablen, die Konstanten (etwa die von Zeichenketten) und einem Laufzeitstapel angelegt.

All dies liegt in einem sogenannten virtuellen Adressraum (typischerweise mit 32- oder 64-Bit-Adressen), den das Betriebssystem einrichtet. Es wird hier von einem virtuellen Adressraum gesprochen, da die verwendeten Adressen nicht den physischen Adressen entsprechen, sondern dazwischen eine durch das Betriebssystem konfigurierte Abbildung durchgeführt wird.

Diese Abbildung wird nicht für jedes einzelne Byte definiert, sondern für größere Einheiten, die Kacheln (im Englischen: *page*) genannt werden. Die Kacheln haben alle eine einheitliche Größe, die aber von Plattform zu Plattform variieren kann. Typisch sind Werte wie etwa 4 oder 8 Kilobyte:

Programm 9.2: Die Größe einer Kachel (*getpagesize.c*)

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5     printf("page␣size␣=␣%d\n", getpagesize());
6 }
```

```
dublin$ uname -m
sun4u
dublin$ gcc -Wall -std=c99 getpagesize.c
dublin$ a.out
page size = 8192
dublin$
```

```
zeus$ uname -m
x86_64
zeus$ gcc -Wall -std=c99 -D_BSD_SOURCE getpagesize.c
zeus$ a.out
page size = 4096
zeus$
```

Sei $[0, 2^n - 1]$ der virtuelle Adressraum, $P = 2^m$ die Größe einer Kachel und

$$M : [0, 2^{n-m} - 1] \to \mathbb{N}_0$$

die Funktion, die eine Kachelnummer in die korrespondierende physische Anfangsadresse abbildet. Dann lässt sich folgendermassen aus der virtuellen Adresse a_{virt} die zugehörige physische Adresse a_{phys} ermitteln:

$$a_{phys} = M(a_{virt} \textbf{ div } P) + a_{virt} \textbf{ mod } P$$

Die Funktion M wird von der zur Hardware gehörenden MMU (*memory management unit*) implementiert in Abhängigkeit von Tabellen, die das Betriebssystem konfigurieren kann. Für weite Teile des Adressraums bleibt M jedoch undefiniert. Ein Versuch, über einen entsprechenden Zeiger zuzugreifen, führt dann zu einem Abbruch des Programms (*segmentation violation*).

Abbildung 9.1 zeigt die traditionelle Aufteilung des Adressraums zum Zeitpunkt des Programmstarts. Hierbei sind die ungenutzten Teile des Adressraumes weiß (M ist hier undefiniert). Falls der Stapelzeiger des Laufzeitstapels den zu Beginn belegten Bereich verlässt, wird dieser vom Betriebssystem im Rahmen des zur Verfügung stehenden Speichers automatisch vergrössert. Deswegen ist der Laufzeitstapel auch vom Rest des Adressraums abgetrennt, damit der zur Verfügung stehende Spielraum genügend groß ist. Die Umgebung der Adresse 0 bleibt unbelegt, damit Verweise durch den Nullzeiger zu einem Abbruch führen.

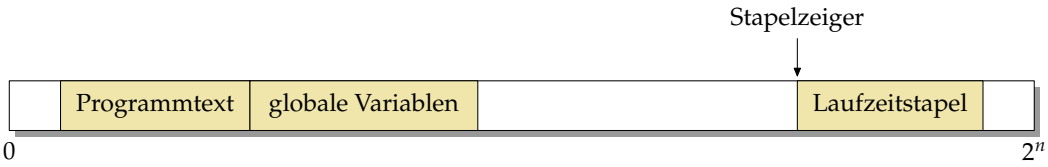

Abbildung 9.1: Aufteilung des Adressraums zu Beginn

Folgendes Programm nutzt die Adressen einiger vordefinierter Symbole und die einer lokalen Variable, um die Anordnung im Adressraum zu ermitteln:

Programm 9.3: Anordnung des Programmtexts, der globalen Variablen und des Laufzeitstapels im Adressraum (*aspace.c*)

```
1  #include <stdio.h>
2
3  extern void etext;
4  extern void edata;
5  extern void end;
6
7  int main() {
8      int local;
9      int* ip = &local;
10
11     printf("location␣of␣main():␣␣␣␣␣␣␣␣%8p\n", &main);
12     printf("end␣of␣program␣text:␣␣␣␣␣␣␣%8p\n", &etext);
13     printf("end␣of␣initialized␣data:␣␣␣%8p\n", &edata);
14     printf("end␣of␣uninitialized␣data:␣%8p\n", &end);
15     printf("address␣of␣local␣variable:␣%8p\n", ip);
16 }
```

```
dublin$ gcc -Wall -std=c99 aspace.c
dublin$ a.out
location of main():           1060c
end of program text:          1071c
end of initialized data:      20960
end of uninitialized data:    20980
address of local variable: ffbff77c
dublin$ size a.out
   text    data     bss     dec     hex filename
   1906     272      32    2210     8a2 a.out
dublin$
```

Das Schlüsselwort **extern** wird hier verwendet, um eine Variable zu deklarieren, die anderswo definiert ist. Die Symbole *etext*, *edata* und *end* sind keine Variablen im üblichen Sinne, da sie von Binder (*ld*, für *linkage editor* stehend, ist das Werkzeug, das aus einzelnen Programmteilen ein vollständiges ausführbares Programm erstellt) automatisch deklariert werden. Entsprechend wurde hier auch der Datentyp **void** verwendet, da sich hinter diesen Namen kein Inhalt verbirgt und somit nur die Verwendung des Adress-Operators & sinnvoll ist. Konkret steht *etext* für das Ende des Programmtexts, *edata* für das Ende der initialisierten globalen Variablen und *end* für das Ende der uninitialisierten Variablen. Das Werkzeug *size* liefert für ausführbare Programme die Größen des Programmtextbereichs (*text*), der initialisierten Daten (*data*) und der uninitialisierten Daten (*bss*, für *block storage segment* stehend).

Die Aufteilung des Adressraums eines laufenden Prozesses kann auch mit dem Kommando *pmap* aufgelistet werden. Wenn das Programm jedoch nur für sehr kurze Zeit läuft oder der Stand zu Anfang interessiert, empfiehlt sich die Verwendung eines Debuggers. Hier wird beispielhaft der *mdb* (*modular debugger*) verwendet, der allerdings nur unter Solaris zur Verfügung steht[1]:

[1] Alternativ kann natürlich der *gdb* (GNU debugger) verwendet werden, jedoch ist dies beim *gdb* etwas umständlicher

```
dublin$ mdb
Loading modules: [ libc.so.1 ]
> main:b
> :r
mdb: stop at main
mdb: target stopped at:
main:           save      %sp, -0x78, %sp
> $m
            BASE            LIMIT             SIZE NAME
           10000            12000             2000
/home/thales/sai/lehre/ws06/ai3/skript/9/progs/a.out
           20000            22000             2000
/home/thales/sai/lehre/ws06/ai3/skript/9/progs/a.out
        ff280000         ff32c000            ac000 /usr/lib/libc.so.1
        ff33c000         ff344000             8000 /usr/lib/libc.so.1
        ff390000         ff392000             2000 [ anon ]
        ff3a0000         ff3a2000             2000
/usr/platform/sun4u-us3/lib/libc_psr.so.1
        ff3b0000         ff3de000            2e000 /usr/lib/ld.so.1
        ff3ee000         ff3f0000             2000 /usr/lib/ld.so.1
        ff3f0000         ff3f2000             2000 /usr/lib/ld.so.1
        ff3fa000         ff3fc000             2000 /usr/lib/libdl.so.1
        ffbfa000         ffc00000             6000 [ stack ]
> $q

dublin$
```

Hier lässt sich sehen, dass der Speicherbereich [0, 0x10000) unbelegt ist, dann das Textsegment ab der Adresse 0x10000 folgt und dann mit Abstand davon bei der Adresse 0x20000 der Bereich der globalen Daten liegt. Im Unterschied zum oben gezeigten Diagramm lässt sich hier ersehen, dass die C-Bibliothek (repräsentiert durch die Datei */usr/lib/libc.so.1*) ebenfalls in den Adressraum gelegt worden ist. Entsprechend dieser Aufteilung stehen für den Laufzeitstapel hier maximal etwa 8 Megabyte zur Verfügung.

9.3 Dynamische Speicherverwaltung

Grundsätzlich ist eine dynamische Speicherverwaltung recht einfach. Jeder bislang unbenutzte Bereich des Adressraums kann mit Hilfe des Betriebssystems belegt werden. Traditionellerweise wird hierfür der Bereich im Adressraum unmittelbar hinter den globalen Daten verwendet. UNIX verwaltet hier einen Zeiger, *break* genannt, der zu Beginn auf die Adresse von *end* zeigt und sukzessive entsprechend des dynamischen Speicherbedarfs zu höheren Adressen verschoben werden kann (siehe Abbildung 9.2).

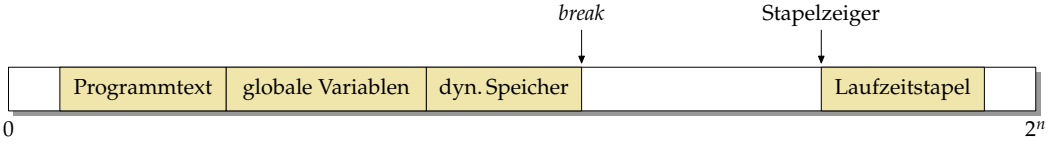

Abbildung 9.2: Dynamisch belegter Speicher im Adressraum

Folgendes Programm zeigt, wie mit dem Systemaufruf *sbrk()* der *break* um eine angegebene Größe verschoben werden kann, um auf diese Weise Speicher dynamisch zu belegen. Die Funktion *sbrk()* liefert dabei im Erfolgsfalle den *alten* Wert des *break* zurück, der dann auf

die neu belegte Speicherfläche verweist. Prinzipiell akzeptiert *sbrk()* beliebig kleine Längen, sinnvoll ist aber nur ein Vielfaches der Kachelgröße.

Programm 9.4: Dynamisches Belegen von Speicher mit Hilfe von *sbrk()* (*sbrk.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h> // fuer exit
3  #include <unistd.h> // fuer getpagesize und sbrk
4
5  int main() {
6      // aktuelle Kachelgroesse ermitteln
7      int pagesize = getpagesize();
8      // eine Kachel dynamisch allokieren,
9      // im Erfolgsfalle zeigt text auf die
10     // die alte Position des break
11     char* buf = (char*) sbrk(pagesize);
12     if (buf == (void*) -1) {
13         // Fehlermeldung ausgeben und Schluss
14         perror("sbrk"); exit(1);
15     }
16
17     // Zeile einlesen und reversiert ausgeben
18     int ch; char* cp = buf+pagesize;
19     *--cp = '\0';
20     while ((ch = getchar()) != EOF && ch != '\n' && cp > buf) {
21         *--cp = ch;
22     }
23     printf("%s\n", cp);
24 }
```

```
dublin$ gcc -Wall -std=c99 break.c
dublin$ echo "Hallo zusammen!" | a.out
!nemmasuz ollaH
dublin$
```

Dieses Verfahren hat jedoch zwei wichtige Nachteile:

- Jeweils ein Systemaufruf wird für das Belegen weiteren Speichers investiert. Dies ist ineffizient.
- Es ist keine Freigabe oder Wiederverwendung nicht mehr genutzter Speicherflächen vorgesehen.

Aus diesem Grunde ist es sinnvoll, in der Bibliothek Funktionen wie *malloc()* und *free()* anzubieten, die kleinere Speicherflächen unterstützen und auch die Wiederverwendung von nicht mehr genutzten Speicherflächen ermöglichen.

Für *malloc()* und *free()* gibt es zahlreiche Implementierungen. Es kann hier sinnvoll sein, eine Entscheidung zu treffen, ob Sparsamkeit mit dem Umgang mit Speicher oder ob die Laufzeiteffizienz wichtiger sind. Je nach dieser Entscheidung bieten sich unterschiedliche Algorithmen an.

Leider ist eine automatische Speicherbereinigung (*garbage collection*) in C nicht im gewohnten Sinne wie bei Java möglich, die es erlauben würde, auf *free()* zu verzichten. Das Problem ist, dass bei Java der Übersetzer spezielle Datenstrukturen für alle verwendeten

Datentypen generiert, die einer Speicherbereinigung die Erfassung sämtlicher Zeiger erlaubt. Ferner müssen die Zeiger immer wohldefiniert sein (also entweder 0 oder auf ein lebendes Objekt verweisend). Dies war in C nie vorgesehen. C-Übersetzer generieren diese Datenstrukturen nicht und Zeiger können grundsätzlich undefinierte Werte enthalten. Dennoch werden automatische Speicherbereinigungen auch für C angeboten[2]. Dies sind sogenannte konservative Speicherbereinigungen, die jeden potentiellen Zeigerwert als solchen erachten. Da ihnen die Unterstützung fehlt, müssen sie dann auch Objekte als noch lebendig betrachten, auf die zufällig ein undefinierter Zeiger verweist. Entsprechend können sie nicht in dem Umfang Speicher automatisch freigeben, wie es sonst erwartet wird.

Das im folgenden vorgestellte Beispiel zeigt eine sehr einfache dynamische Speicherverwaltung. Auf die Möglichkeit, dynamisch Speicherbereiche durch Systemaufrufe zu belegen, wurde aus Gründen der Übersichtlichkeit verzichtet. Stattdessen dient der Vektor *dynmem* als Speicherfläche, die dynamisch vergeben wird.

An dem Beispiel wird eine besondere Charakteristik von *free()* bewusst. Da nur ein Zeiger übergeben wird und somit keine Angabe der Größe der dahinter belegten Speicherfläche als Parameter vorliegt, muss diese Information von *malloc()* irgendwo notiert und dann von *free()* gefunden werden. Die einfachste Lösung besteht darin, diese Information unmittelbar *vor* dem Zeiger abzulegen.

Durch fortlaufendes Belegen und Freigeben einzelner Speicherabschnitte wird im Laufe der Zeit die Speicherbelegung immer weiter gesplittet. Zwischen Abschnitten mit belegten Speicherflächen liegen dann immer Flächen unterschiedlicher Größen, die wieder freigegeben worden sind. Um all die freien Flächen zwischen den belegten Flächen auffindbar zu machen, bietet es sich an, einen Ring freier Speicherflächen anzulegen, wobei jede freie Speicherfläche auf die nächste Fläche verweist.

Für den Ring wird dann neben der Größenangabe noch zusätzlich ein Zeiger auf das nächste Element benötigt. Entsprechend werden für die Verwaltungsstruktur zwei Komponenten benötigt: *size* und *next*.

Wenn mehrere Flächen hintereinander frei werden, ist es sinnvoll, diese zu einer größeren Fläche wieder zusammenzulegen. Damit dies mit vertretbaren Aufwand erfolgen kann, empfiehlt es sich, den Ring der freien Flächen so sortiert zu verwalten, dass jede freie Fläche auf *die nächste* im Speicher liegende freie Fläche verweist – mit Ausnahme der letzten freien Fläche, die dann wieder auf die erste verweist.

Das nächste Problem liegt darin, eine belegte Speicherfläche, die an *free()* übergeben wird, wieder in den Ring der freien Speicherflächen entsprechend der Sortierung einzuordnen. Dies wird in der vorgestellten Lösung erreicht, indem *malloc()* den *next*-Zeiger auf die unmittelbar vorangehende Speicherfläche setzt. Dies erleichtert später bei *free()* das Auffinden der nächsten vorangehenden freien Speicherfläche, hinter die dann die gerade freigewordene Speicherfläche eingehängt werden kann.

Da die freien Speicherflächen in einem Ring organisiert sind, wäre es ärgerlich, wenn beim vollständigen Verbrauch des zur Verfügung stehenden Speichers das letzte Element aus dem Ring verschwinden würde. Um das zu vermeiden, gibt es ein spezielles Ringelement, das nur aus der Verwaltungsstruktur besteht, aber keine Fläche anbietet und somit *size* den Wert 0 hat. Wenn dieses spezielle Ringelement an den Anfang der gesamten zur Verfügung stehenden Speicherfläche gelegt wird, so bringt dies auch den Vorteil, dass wir bei allen regulären Speicherflächen immer davon ausgehen können, das eine unmittelbar davor liegende Speicherfläche existiert.

Abbildung 9.3 zeigt die Ausgangssituation, bei der der Ring aus genau zwei Elementen besteht, dem speziellen Ringelement mit *size* gleich 0 und einem weiteren Ringelement, das die gesamte freie (noch zusammenhängende) Fläche verwaltet.

Wenn danach 32 Bytes angefordert werden, ist es nicht sinnvoll, die gesamte 16368 Bytes bietende freie Fläche dafür wegzugeben. Stattdessen wird die Fläche in zwei Teile

[2]Die bekannteste Lösung ist der Boehm-Demers-Weiser Speicherbereiniger: *www.hpl.hp.com/personal/Hans_Boehm/gc/*

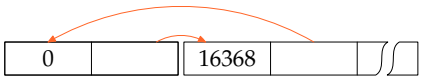

Abbildung 9.3: Ring der freien Speicherflächen zu Beginn

zerlegt, wovon der erste Teil weiterhin frei bleibt und der zweite Teil belegt wird. Abbildung 9.4 zeigt diesen Stand, wobei die freien Speicherflächen weiß bleiben und belegte Speicherflächen farbig unterlegt sind. Ferner zu beachten ist hier, dass der *next*-Zeiger der Verwaltungsstruktur immer auf die unmittelbar vorangehende Speicherfläche verweist – unabhängig davon, ob diese frei oder belegt ist.

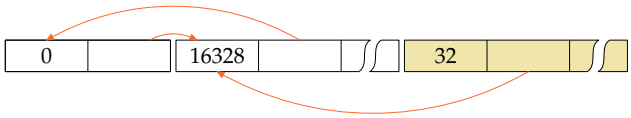

Abbildung 9.4: Speicherverwaltungsstruktur nach der ersten Belegung einer Speicherfläche

Wie Abbildung 9.5 zeigt, wiederholt sich das Verfahren, wenn weitere Speicherflächen belegt werden. In diesem Beispiel wurden noch einmal 32 und dann 64 Bytes angefordert.

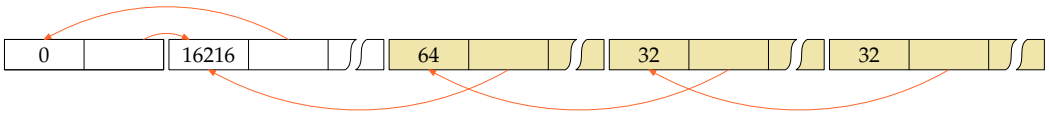

Abbildung 9.5: Speicherverwaltungsstruktur nach drei Speicherbelegungen

Wenn jetzt die mittlere Speicherfläche mit 32 Bytes wieder freigegeben wird, dann kommt sie in den Ring der freien Speicherflächen zurück, wie Abbildung 9.6 zeigt.

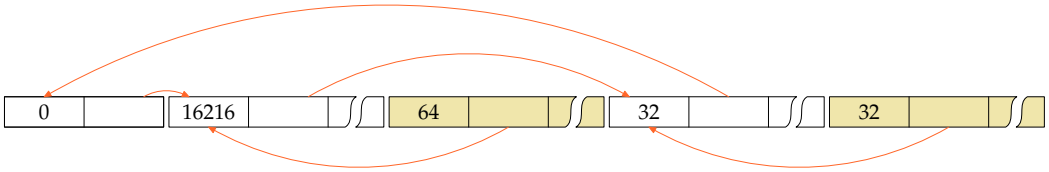

Abbildung 9.6: Speicherverwaltungsstruktur nach einer Speicherfreigabe

Wenn mehrere freie Speicherflächen zur Verfügung stehen, stellt sich die Frage, welche als erstes zur Vergabe bei der nächsten Speicheranforderung in Betracht gezogen wird. Als günstig hat sich hier das Verfahren erwiesen, das der Ring immer relativ zur letzten zur Freigabe verwendeten Fläche nach einer passenden Fläche durchsucht wird. Dieses Verfahren wird in der Literatur *circular first fit* genannt. Abbildung 9.7 zeigt, dass bei der nächsten Speicherbelegung zuerst die kleine gerade frei gewordene Fläche in Betracht gezogen wird und 9.8 zeigt, wie danach wieder die große Freifläche für die darauffolgende Belegung genutzt wird.

Um eine zu hohe Aufsplitterung in winzige Speicherflächen zu vermeiden, ist es wichtig, die Gelegenheit zu nutzen, zusammenhängende freie Flächen zusammenzufassen wie

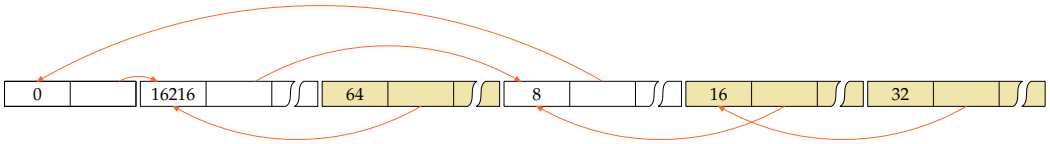

Abbildung 9.7: Suche nach freien Flächen entsprechend des *circular first fit*-Verfahrens (Teil 1)

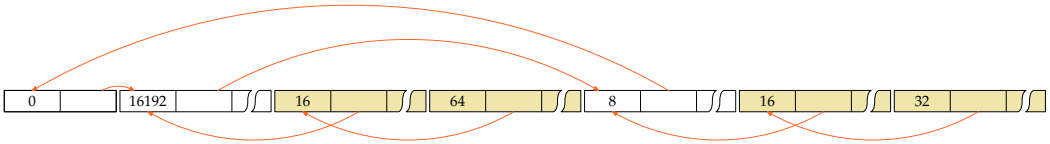

Abbildung 9.8: Suche nach freien Flächen entsprechend des *circular first fit*-Verfahrens (Teil 2)

es in Abbildung 9.9 demonstriert wird nach der Freigabe der belegten Speicherfläche von 64 Bytes.

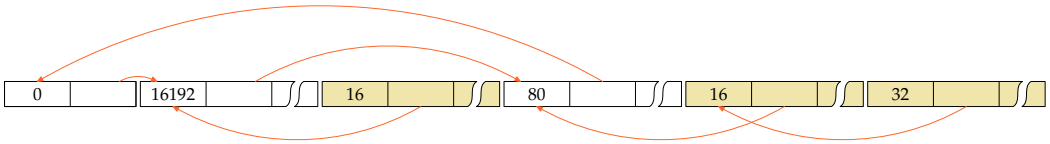

Abbildung 9.9: Zusammenlegung benachbarter freier Speicherflächen

Das Programm 9.5 ist eine beispielhafte Umsetzung der vorgestellten Speicherverwaltung, bei der zur Vermeidung von Konflikten mit der C-Bibliothek die Namen *my_malloc()* und *my_free()* an Stelle von *malloc()* und *free()* verwendet werden. Das Hauptprogramm erlaubt einen interaktiven Test. Prinzipiell ähnelt die Speicherverwaltung der in [Kernighan 1990] publizierten Lösung. Im Vergleich dazu wurde hier der Aufwand zum Einfügen frei gewordener Speicherflächen in den Ring verringert.

Programm 9.5: Beispiel für eine dynamische Speicherverwaltung (*alloc.c*)

```
1  #include <assert.h>
2  #include <stdbool.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  #define MEM_SIZE 2048 /* zu multiplizieren mit sizeof(memnode) */
7  /* Verbund, der einer freien oder
8     belegten Speicherflaeche vorangeht */
9  typedef struct memnode {
10    size_t size;
11       /* Groesse der Speicherflaeche, die diesem Verbund
12          unmittelbar folgt, jedoch ohne Einberechnung
13          des Speicherbedarfs fuer diesen Verbund */
14    struct memnode* next;
```

```
15          /* verweist bei freien Speicherflaechen zum naechsten
16              Ringelement der freien Speicherflaechen;
17              bei belegten Speicherflaechen wird dieser Verweis
18              geaendert zu dem unmittelbar vorangehenden Speicherelement,
19              egal ob dieses frei oder belegt ist
20          */
21  } memnode;
22
23  /* Speicherbereich, der von den folgenden Funktionen
24      verwaltet wird */
25  memnode dynmem[MEM_SIZE] = {
26      /* bleibt immer im Ring der freien Speicherflaechen */
27      {0, &dynmem[1]},
28      /* enthaelt zu Beginn den gesamten freien Speicher */
29      {sizeof dynmem − 2*sizeof(memnode), dynmem}
30  };
31  memnode* node = dynmem; /* durchlaeuft den Ring der freien Speicherflaechen */
32  memnode* root = dynmem; /* verweist auf das erste Element im Ring */
33
34  /* um nicht zu sehr zu fragmentieren, werden alle zu belegenden
35      Speicherflaechen auf ein Vielfaches von ALIGN aufgerundet;
36      dies sollte eine Zweierpotenz sein, die gross genug ist,
37      um die Ausrichtungsanforderungen aller elementaren Datentypen
38      zu befriedigen */
39  const int ALIGN = sizeof(memnode);
40
41  /* ermittelt, ob die Speicherflaeche bei ptr belegt ist */
42  bool is_allocated(memnode* ptr) {
43      /* Das Wurzelelement ist immer frei */
44      if (ptr == root) return false;
45      /* Wenn der Zeiger vorwaerts zeigt, dann ist es frei */
46      if (ptr−>next > ptr) return false;
47      /* Der Zeiger zeigt jetzt rueckwaerts, denn
48          ptr−>next == ptr ist nur beim Wurzelelement moeglich */
49      /* Wenn nicht auf Wurzel verwiesen wird, ist ptr belegt */
50      if (ptr−>next != root) return true;
51      /* Wenn ptr−>next auf die Wurzel verweist, gibt es zwei
52          Faelle:
53              − Es ist belegt und ptr liegt unmittelbar hinter
54                dem Wurzelelement
55              − Es ist frei und ptr zeigt auf die freie Speicherflaeche
56                mit der hoechsten Adresse
57      */
58      /* Wenn root−>next jenseits von ptr zeigt, dann ist ptr belegt */
59      if (root−>next > ptr) return true;
60      /* Wenn die Wurzel das einzige freie Element ist,
61          dann ist ptr ebenfalls belegt */
62      return root−>next == root;
63  }
64
65  /* liefert einen Zeiger auf die p folgende Speicherflaeche,
66      unabhaengig davon, ob sie belegt oder frei ist;
67      zu beachten ist, dass bei der hoechstgelegenen Speicherflaeche
```

```
68      dynmem + MEM_SIZE zurueckgeliefert wird */
69  memnode* successor(memnode* p) {
70      return (memnode*) ((char*)(p+1) + p->size);
71  }
72
73  /* entspricht malloc(), aber bedient sich nur aus dynmem[] */
74  void* my_malloc(size_t size) {
75      assert(size >= 0);
76      if (size == 0) return 0;
77      /* runde die gewuenschte Groesse auf
78          das naechste Vielfache von ALIGN */
79      if (size % ALIGN) {
80          size += ALIGN - size % ALIGN;
81      }
82      /* beginnend mit node->next suchen wir nach
83          einem Element aus dem Ring der freien Speicherflaechen,
84          das genuegend Kapazitaet fuer size Bytes bietet;
85          dabei verweist ptr auf das Element, das wir
86          gerade untersuchen, und prev auf das vorherige Element */
87      memnode* prev = node; memnode* ptr = prev->next;
88      do {
89          if (ptr->size >= size) break; /* passendes Element gefunden */
90          prev = ptr; ptr = ptr->next;
91      } while (ptr != node); /* bis der Ring durchlaufen ist */
92      if (ptr->size < size) return 0; /* Speicher ist ausgegangen */
93      if (ptr->size < size + 2*sizeof(memnode)) {
94          node = ptr->next; /* "circular first fit" */
95          /* entferne ptr aus dem Ring der freien Speicherflaechen */
96          prev->next = ptr->next;
97          /* suche nach der unmittelbar vorangehenden Speicherflaeche;
98              zu beachten ist hier, dass zwischen prev und ptr noch
99              einige belegte Speicherflaechen liegen koennen
100         */
101         for (memnode* p = prev; p < ptr; p = successor(p)) {
102             prev = p;
103         }
104         ptr->next = prev;
105         /* node muss immer auf ein freies Element zeigen */
106         if (ptr == node) node = root;
107         return (void*) (ptr+1);
108     }
109     node = ptr; /* "circular first fit" */
110     /* die bisherige Speicherflaeche wird zerlegt
111         in einen neue passende Speicherflaeche, die jetzt belegt wird
112         und die alte verkleinerte Speicherflaeche, die uebrigbleibt
113     */
114     /* lege das neue Element an */
115     memnode* newnode = (memnode*)((char*)ptr + ptr->size - size);
116     newnode->size = size; newnode->next = ptr;
117     /* korrigiere den Zeiger der folgenden Speicherflaeche,
118         falls sie belegt sein sollte */
119     memnode* next = successor(ptr);
120     if (next < dynmem + MEM_SIZE && next->next == ptr) {
```

```
121         next->next = newnode;
122     }
123     /* reduziere die Groesse des alten Elements
124         aus dem Ring der freien Speicherflaechen */
125     ptr->size -= size + sizeof(memnode);
126     return (void*) (newnode+1);
127 }
128
129 /* Lege die beiden freien Speicherflaechen bei prev und ptr
130     zusammen, falls dies moeglich sein sollte */
131 void join_if_possible(memnode* prev, memnode* ptr) {
132     memnode* p = (memnode*)((char*) prev + sizeof(memnode) + prev->size);
133     /* Wenn die beiden Speicherflaechen nicht direkt
134         hintereinander liegen, koennen wir sie nicht vereinigen */
135     if (p != ptr) return;
136     /* Das Wurzelelement muss in Ruhe gelassen werden,
137         damit immer mindestens ein Element im Ring der freien
138         Speicherflaechen verbleibt */
139     if (prev->size == 0) return;
140     /* Vereinigung der beiden freien Speicherflaechen */
141     prev->next = ptr->next;
142     prev->size += ptr->size + sizeof(memnode);
143     /* korrigiere den Zeiger der folgenden Speicherflaeche,
144         falls sie belegt sein sollte */
145     memnode* next = successor(ptr);
146     if (next < dynmem + MEM_SIZE && next->next == ptr) {
147         next->next = prev;
148     }
149     /* setze node auf prev, falls sie zuvor auf ptr zeigte */
150     if (node == ptr) node = prev;
151 }
152
153 /* gibt die Speicherflaeche bei ptr wieder frei;
154     ptr muss zuvor von my_malloc() zurueckgeliefert worden sein */
155 void my_free(void* ptr) {
156     if (!ptr) return;
157     memnode* node = (memnode*) ((char*) ptr - sizeof(memnode));
158     /* Ueberpruefe, ob node tatsaechlich belegt ist */
159     assert(is_allocated(node));
160     /* Wir suchen nach dem vorangehenden Element aus
161         dem Ring der freien Speicherflaechen */
162     memnode* prev = node->next;
163     while (is_allocated(prev)) {
164         prev = prev->next;
165     }
166     /* fuege node wieder in den Ring an passender Stelle ein */
167     node->next = prev->next; prev->next = node;
168     /* Lege zusammenhaengende freie Speicherflaechen wieder
169         zusammen, falls dies moeglich ist */
170     join_if_possible(node, node->next);
171     join_if_possible(prev, node);
172 }
173
```

```
174 /* Gibt zur Analyse den gesamten Status von dynmem[] auf
175     der Standardausgabe aus;
176     hier wird angenommen, dass Zeiger bequem als int
177     darstellbar sind --> das ist nicht portabel */
178 void debug_alloc(void) {
179     for (memnode* ptr = root; ptr < dynmem + MEM_SIZE; ptr = successor(ptr)) {
180         if (ptr == node) {
181             putchar('*');
182         } else {
183             putchar(' ');
184         }
185         bool allocated = is_allocated(ptr);
186         if (allocated) {
187             putchar('A');
188         } else {
189             putchar('F');
190         }
191         putchar(' ');
192         printf("%d %5lu -->  %d", (int) ptr,
193             (unsigned long) ptr->size, (int)ptr->next);
194         if (allocated) {
195             /* Ausgabe der freigebbaren Adresse */
196             printf(" (f %d)", (int)(ptr+1));
197         }
198         putchar('\n');
199     }
200 }
201
202 /* Interaktives Testprogramm */
203 int main() {
204     char command;
205     int intval;
206     void* p;
207
208     while (printf(": "), scanf(" %c", &command) == 1) {
209         switch (command) {
210             case 'a':
211                 if (scanf("%d", &intval) != 1) break;
212                 p = my_malloc(intval);
213                 if (p) {
214                     printf("got %d bytes at %d\n", intval, (int) p);
215                 } else {
216                     printf("failed\n");
217                 }
218                 break;
219             case 'f':
220                 if (scanf("%d", &intval) != 1) break;
221                 my_free((void*) intval);
222                 break;
223             case 'd':
224                 debug_alloc();
225                 break;
226             default:
```

```
227                 printf("supported commands:\n");
228                 printf("a n          allocate n bytes\n");
229                 printf("d            debugging output\n");
230                 printf("f p          free area at p\n");
231                 break;
232             }
233         }
234 }
```

```
doolin$ gcc -Wall -std=c99 alloc.c
doolin$ a.out
: a 20
got 20 bytes at 151888
: a 40
got 40 bytes at 151840
: a 60
got 60 bytes at 151768
: a 30
got 30 bytes at 151728
: a 70
got 70 bytes at 151648
: a 40
got 40 bytes at 151600
: f 151840
: f 151648
: f 151728
: a 20
got 20 bytes at 151736
: d
 F 135528     0 --> 135536
 F 135536 16048 --> 151640
 A 151592    40 --> 135536 (f 151600)
*F 151640    80 --> 151832
 A 151728    24 --> 151640 (f 151736)
 A 151760    64 --> 151728 (f 151768)
 F 151832    40 --> 135528
 A 151880    24 --> 151832 (f 151888)
: doolin$
```

Zusammenfassend sind folgende Punkte interessant bei einer Speicherverwaltung in C:

- Die Verwaltungsstrukturen sind typischerweise benachbart zu den vergebenen Speicherflächen. Entsprechend kann bereits ein minimaler Index-Fehler (eins zu weit über den zulässigen Bereich) dazu führen, dass diese Strukturen zerstört werden mit unabsehbaren Folgen bei folgenden Aufrufen von *malloc()* oder *free()*.

- Selbst wenn benachbarte Flächen wieder zusammengelegt werden, tendiert die gesamte Struktur in Richtung einer fortlaufenden Zersplitterung. Das bedeutet, dass es zunehmend schwieriger wird, größere Flächen zu belegen, so dass mehr Speicher vom System angefordert werden muss, obwohl im Prinzip genügend Speicher zur Verfügung steht. Das führt dazu, dass langlaufende Prozesse mit sehr variablen Größen bei den Speicherflächen (beispielsweise durch reguläre dynamische Datenstrukturen und größere dynamisch organisierte Puffer) dazu tendieren, im Laufe der Zeit immer mehr Speicher zu belegen.

- Die wenigsten Implementierungen sind in der Lage, Speicher zur Laufzeit eines Prozesses wieder an das Betriebssystem zurückzugeben. Voraussetzung dazu wäre, dass vollständige Kacheln wieder frei würden. Dies ist bei der üblichen Versplitterung so unwahrscheinlich, dass Implementierungen dies typischerweise überhaupt nicht vorsehen.

9.4 Dynamische Vektoren

Dank *realloc()* ist es möglich, Vektoren zur Laufzeit dynamisch zu vergrößern. Da C selbst nicht die aktuelle Länge eines dynamischen Vektors verwaltet, liegt es nahe, dies mit Hilfe eines entsprechenden Verbunds zu tun.

Neben dem eigentlichen Vektor gibt es dann noch Komponenten wie *size* und *used*, die die tatsächliche Länge des Vektors und die in Benutzung befindliche Länge notieren, wobei natürlich immer $used \leq size$ gelten sollte. Es ist sinnvoll, diese beiden Größen zu trennen, da dies einem die Flexibilität gibt, den Vektor in größeren Schüben zu vergrößern und dies nicht etwa elementweise zu tun, da jede echte Vergrößerung mit Hilfe von *realloc()* das Risiko des teuren Umkopierens des gesamten Vektors trägt. Das verschwendet zwar Speicherplatz, ist aber sehr gut für das Laufzeitverhalten. Ohne diese Technik kann es passieren, dass ein sonst linearer Aufwand für das schrittweise Vergrößern des Vektors quadratisch wird.

Prinzipiell wäre es möglich, direkt auf den Vektor zuzugreifen. Hier in diesem Beispiel erfolgt der Zugriff aber über die Funktionen *arrayGet()* und *arraySet()*, die die Gelegenheit nutzen, die Indizes zu überprüfen. Solche Überprüfungen erfolgen am besten mit dem Makro *assert* aus <*assert.h*>. Das Makro *assert* erhält jeweils eine Bedingung. Hat diese den Wert *false*, so wird die Ausführung abgebrochen mit einer Fehlermeldung, die den Text der Bedingung und die Stelle des Aufrufs nennt.

Das folgende Beispiel liest beliebig viele ganze Zahlen aus der Standardeingabe ein und mischt sie mit einem Misch-Algorithmus von Fisher und Yates (siehe Algorithmus P auf Seite 145 in [Knuth 1997]).

Programm 9.6: Zufälliges Mischen ganzer Zahlen mit Hilfe eines dynamischen Vektors (*shuffle.c*)

```
1  #include <assert.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <time.h>
6
7  typedef struct {
8      int* array; /* Zeiger auf das erste Element des Vektors */
9      int size; /* Groesse des belegten Speichers */
10     int used; /* tatsaechliche Groesse des Vektors; used <= size! */
11 } IntArray;
12
13 void* my_calloc(size_t nelem, size_t elsize) {
14     assert(nelem > 0 && elsize > 0);
15     void* ptr = calloc(nelem, elsize);
16     if (ptr) return ptr; /* alles OK */
17     /* Fehlerbehandlung: */
18     fprintf(stderr, "out␣of␣memory␣--␣aborting!\n");
19     /* Termination mit core dump */
```

```
20      abort();
21  }
22
23  void* my_realloc(void* ptr, size_t size) {
24      ptr = realloc(ptr, size);
25      if (ptr) return ptr; /* alles OK */
26      /* Fehlerbehandlung: */
27      fprintf(stderr, "out of memory --- aborting!\n");
28      /* Termination mit core dump */
29      abort();
30  }
31
32  /*
33   * Neues Integer-IntArray mit der angegebenen
34   * Groesse erzeugen und zurueckgeben
35   *
36   * Parameter "IntArray **a" notwendig, um IntArray-Zeiger
37   * als Referenz-Parameter zu uebergeben!
38   */
39  void arrayNew(IntArray** a, int len) {
40      assert(len >= 0);
41      *a = (IntArray*) my_calloc(1, sizeof(IntArray));
42      (*a)->size = (*a)->used = len;
43      if (len > 0) {
44          (*a)->array = (int*) my_calloc(len, sizeof(int));
45      } else {
46          (*a)->array = 0;
47      }
48  }
49
50  /* Integer-IntArray wieder freigeben */
51  void arrayDelete(IntArray** a) {
52      if ((*a)->array) free((*a)->array);
53      free(*a);
54      *a = 0;
55  }
56
57  /* Groesse des Integer-Arrays veraendern */
58  void arrayResize(IntArray* a, int len) {
59      assert(len >= 0);
60      if (a->size < len) {
61          /* da realloc durch das Umkopieren recht
62              teuer sein kann, empfiehlt es sich,
63              gleich etwas mehr Speicher zu reservieren
64          */
65          int newlen = len + (len << 3) + 10;
66          a->array = (int*) my_realloc(a->array, sizeof(int) * newlen);
67          a->size = newlen;
68      }
69      a->used = len;
70  }
71
72  /* Laenge/Groesse des Arrays ermitteln */
```

```
73  int arrayLength(IntArray* a) {
74      return a->used;
75  }
76
77  /* Element des Arrays lesen */
78  int arrayGet(IntArray* a, int i) {
79      assert(i >= 0 && i < a->used);
80      return a->array[i];
81  }
82
83  /* Element des Arrays neu setzen */
84  void arraySet(IntArray* a, int i, int value) {
85      assert(i >= 0 && i < a->used);
86      a->array[i] = value;
87  }
88
89  int main() {
90      IntArray* a;
91      int value;
92
93      /* Pseudo-Zufallszahlengenerator initialisieren */
94      unsigned int seed = time(0); srand(seed);
95
96      arrayNew(&a, 0);
97
98      /* Alle Zahlen aus der Standardeingabe einlesen */
99      for (int index = 0; scanf("%d", &value) == 1; ++index) {
100         arrayResize(a, index + 1); arraySet(a, index, value);
101     }
102
103     /* Misch-Algorithmus von Fisher & Yates, 1938 */
104     for (int i = arrayLength(a) - 1; i >= 0; --i) {
105         /* Ein Element aus dem Bereich 0..i auswaehlen */
106         int j = rand() % (i + 1);
107         /* Vertausche die Elemente mit den Indizes i und j */
108         if (i != j) {
109             int tmp = arrayGet(a, i);
110             arraySet(a, i, arrayGet(a, j));
111             arraySet(a, j, tmp);
112         }
113     }
114
115     /* Alle Zahlen aus dem Vektor ausgeben */
116     for (int i = 0; i < arrayLength(a); ++i) {
117         printf("%d\n", arrayGet(a, i));
118     }
119     arrayDelete(&a);
120 }
```

```
dublin$ gcc -Wall -std=c99 shuffle.c
dublin$ echo 1 2 3 4 5 6 7 8 9 10 | a.out
3
9
4
8
1
10
5
6
7
2
dublin$
```

9.5 Dynamische Zeichenketten

Wenn eine Zeichenkette in einem lokalen Puffer liegt und dieser die Lebenszeit der Funktion überdauern soll, dann ist es sinnvoll, für diesen dynamisch Speicher zu belegen und die Zeichenkette umzukopieren. Dies geschieht sinnvollerweise mit genau der Länge, die benötigt wird unter Berücksichtigung des Null-Bytes.

Hierfür bietet sich die Funktion *strdup()* an, die für eine Kopie der übergebenen Zeichenkette Speicher belegt, die Kopieraktion selbst durchführt und im Erfolgsfalle einen Zeiger auf die neue Kopie zurückgibt. Entsprechend ist

char∗ *s1* = *strdup*(*s*);

eine praktische Kurzform für folgende Lösung:

char∗ *s1* = *malloc*(*strlen*(*s*)+1); *// Nullbyte nicht vergessen*
if (*s1*) *strcpy*(*s1*, *s*); *// nur im Erfolgsfalle kopieren*

9.6 Speicher-Operationen

Die C-Bibliothek bietet Funktionen an, die einige Operationen wie Kopieren, Initialisieren und Vergleichen für Speicherflächen effizient umsetzen. Die Deklarationen für folgende Funktionen sind alle über <*string.h*> erhältlich. Im Einzelnen:

void∗ *memset*(**void**∗ *s*, **int** *c*, *size_t n*)
: Diese Funktion konvertiert *c* in den Datentyp **unsigned char** und initialisiert, beginnend mit der von *s* adressierten Speicherzelle *n* Bytes mit dem Wert von *c*. Der Zeigerwert von *s* wird zurückgegeben.

void∗ *memcpy*(**void**∗ *s1*, **const void**∗ *s2*, *size_t n*)
: Hier werden *n* Bytes von *s2* nach *s1* kopiert. Die beiden Bereiche dürfen sich nicht überlappen. Der Zeigerwert von *s1* wird zurückgegeben.

void∗ *memmove*(**void**∗ *s1*, **const void**∗ *s2*, *size_t n*)
: Diese Funktion arbeitet analog zu *memcpy()*. Im Vergleich zu *memcpy()* lässt sie auch sich überlappende Speicherflächen zu.

int *memcmp*(**const void**∗ *s1*, **const void**∗ *s2*, *size_t n*)
: Hier werden bis zu *n* Bytes (jeweils als **unsigned char**) der Speicherflächen hinter *s1* und *s2* verglichen. Der Rückgabewert ist analog zu dem von *strcmp()* < 0, falls $s1 < s2$, $= 0$, falls $s1 = s2$ und > 0, falls $s1 > s1$. Anders als bei *strcmp()* wird das Null-Byte nicht als ein Terminierungszeichen interpretiert.

void* *memchr*(**const void*** *s*, **int** *c*, *size_t n*)*memchr()*
Diese Funktion sucht das erste Vorkommen von *c* (als **unsigned char**) in den ersten *n* Bytes des Speicherbereichs, auf den *s* zeigt. Der Rückgabewert ist im Erfolgsfalle ein Zeiger auf das erste Vorkommen und andernfalls der Null-Zeiger.

Programm 9.7: Arbeiten mit Speicher-Operationen (*mem.c*)

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main() {
6      char s[] = "Ratatouille";
7      char* s1 = strdup(s);
8
9      memset(s1+1, 'i', 3);
10     printf("s1 = \"%s\"\n", s1);
11
12     printf("memcmp(s, s1, strlen(s)) = %d\n", memcmp(s, s1, strlen(s)));
13
14     memcpy(s1, s, strlen(s)+1);
15     printf("s1 = \"%s\"\n", s1);
16
17     memmove(s1+2, s1, 4);
18     printf("s1 = \"%s\"\n", s1);
19
20     printf("memchr(s1, 't', strlen(s1)) = %s\n",
21         (char*) memchr(s1, 't', strlen(s1)));
22
23     free(s1);
24 }
```

```
dublin$ gcc -Wall -D__EXTENSIONS__ -std=c99 mem.c
dublin$ a.out
s1 = "Riiitouille"
memcmp(s, s1, strlen(s)) = -8
s1 = "Ratatouille"
s1 = "RaRatauille"
memchr(s1, 't', strlen(s1)) = tauille
dublin$
```

Kapitel 10

Kommandozeilenparameter

10.1 Parameter der Funktion *main()*

Bislang wurde die Funktion *main()* immer ohne Parameter deklariert. Dabei werden (analog zu Java) durchaus Parameter übergeben, die den Zugriff auf die Kommandozeilenparameter erlauben. In C liegt die Liste der Kommandozeilenparameter als Vektor von Zeichenketten vor, d.h. als Vektor von Zeigern auf **char**. Das erste Element dieses Vektors enthält den Namen des Programms und die weiteren die zusätzlich auf der Kommandozeile angegebenen Parametern. Da die Zahl der Parameter variabel ist, gibt es dafür einen weiteren Parameter:

```
int main(int argc, char* argv[]) {
    // ...
}
```

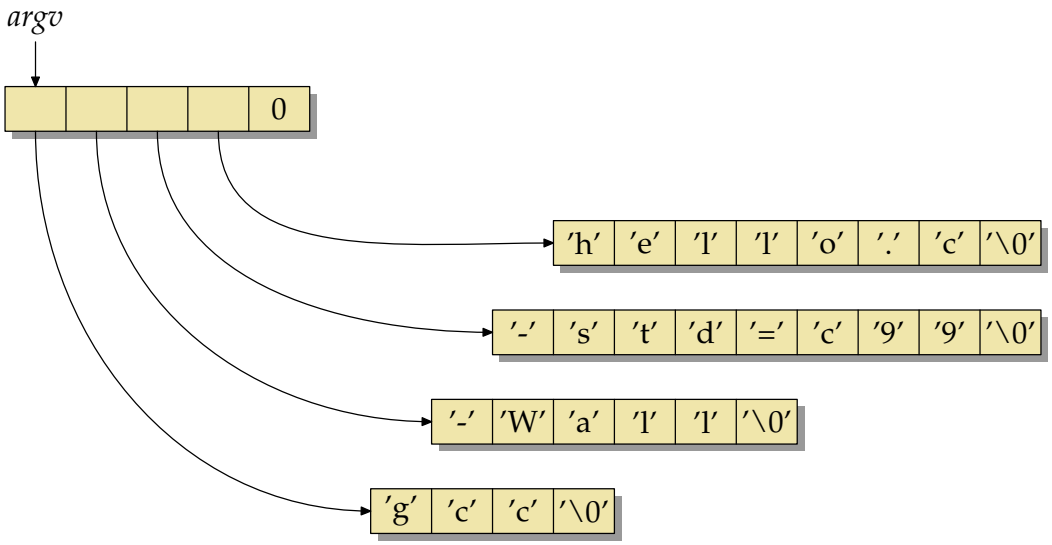

Abbildung 10.1: Die Repräsentierung von *argv* am Beispiel von *„gcc -Wall -std=gnu99 hello.c“*

Die beiden Parameter im Einzelnen:

int *argc*
: Dieser Parameter enthält die Anzahl der Kommandozeilen-Parameter, wobei der

Kommandoname auch dazu gerechnet wird. D. h. falls *argc* gleich 1 ist, gibt es kein Kommandozeilenparameter, sondern nur den Kommandonamen. Und im Falle, dass *argc* den Wert 2 hat, gibt es nur einen Kommandozeilenparameter.

char* *argv*[]
: Dieser Parameter ist ein Vektor von Zeigern auf **char**, also ein Vektor von Zeichenketten. In *argv*[0] ist der Kommandoname enthalten. In *argv*[1] steht das erste Argument, in *argv*[2] das zweite, usw.

Die Abbildung 10.1 zeigt die Repräsentierung von *argv* für die beispielhafte Kommandozeile *„gcc -Wall -std=gnu99 hello.c“*. Wie die Darstellung zeigt, sind nicht nur die einzelnen Zeichenketten durch ein Null-Byte terminiert, sondern auch der Vektor der Zeiger hat am Ende (bei *argv*[*argc*]) einen Null-Zeiger. (Dabei sind die Speicherflächen für Zeiger und für Zeichen aus Gründen der Einfachheit gleich groß dargestellt, obwohl Zeiger größer sein dürften als Zeichen.)

Die gesamte Datenstruktur liegt üblicherweise hintereinander im Speicher, d.h. unmittelbar hinter dem Vektor der Zeiger folgt die Zeichenkette des Kommandonamens, dahinter die des ersten Arguments usw. Eine Implementierung hat aber das Recht, dies zu ändern.

10.2 Ausgabe der Kommandozeilenargumente

Das folgende Beispiel-Programm illustriert die Verwendung von *argc* und *argv* mit der Ausgabe des Kommandonamens, der Zahl der Kommandozeilenargumente und deren Inhalte.

Programm 10.1: Ausgabe der Kommandozeilenargumente und des Kommandonamens (*args.c*)

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]) {
4     printf("Kommandoname:␣%s\n", argv[0]);
5     printf("Anzahl␣der␣Kommandozeilenargumente:␣%d\n", argc−1);
6     for (int i = 1; i < argc; i++) {
7         printf("Argument␣%d:␣%s\n", i, argv[i]);
8     }
9 }
```

```
dublin$ gcc -Wall -std=c99 args.c
dublin$ a.out ein paar Argumente...
Kommandoname: a.out
Anzahl der Kommandozeilenargumente: 3
Argument 1: ein
Argument 2: paar
Argument 3: Argumente...
dublin$ a.out "ein paar" Argumente...
Kommandoname: a.out
Anzahl der Kommandozeilenargumente: 2
Argument 1: ein paar
Argument 2: Argumente...
dublin$ a.out
Kommandoname: a.out
Anzahl der Kommandozeilenargumente: 0
dublin$ gcc -Wall -std=c99 args.c -o args
dublin$ args
Kommandoname: args
Anzahl der Kommandozeilenargumente: 0
dublin$
```

Anmerkung: Wie bereits Abbildung 10.1 andeutet, ist *argv*[*argc*] der Null-Zeiger. Somit gibt es also noch eine alternative Möglichkeit, das Ende des Vektors *argv* zu bestimmen:

Programm 10.2: Alternative Bearbeitung der Kommandozeilenargumente (*args2.c*)

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv) {
4      char** argp = argv;
5      printf("Kommandoname:␣%s\n", *argp++);
6      while (*argp) {
7          printf("Argument␣%d:", argp − argv);
8          printf("␣%s\n", *argp++);
9      }
10 }
```

```
dublin$ gcc -Wall -std=c99 args2.c
dublin$ a.out Hallo zusammen
Kommandoname: a.out
Argument 1: Hallo
Argument 2: zusammen
dublin$
```

10.3 Verarbeiten von Optionen

Als Beispiel möge ein vereinfachter Nachbau des *grep*-Kommandos[1] dienen, das es erlaubt, Zeilen die ein gegebenes reguläres Muster erfüllen aus der Eingabe herauszufiltern.

[1] Der Name *grep* steht für *g/re/p*, einem Kommando des zeilenorientierten Editors *ed*, wobei „g“ für *global* steht (also den gesamten Textinhalt betreffend, „p“ für *print* (um die Ausgabe der Zeilen bittend) und *re* eine Abkürzung bzw. Platzhalter für einen beliebigen regulären Ausdruck (*regular expression*) ist. Das Kommando *grep* wurde gebaut, um den Aufruf von *ed* zu ersparen, wenn es nur darum geht, die Eingabe nach Zeilen mit bestimmten Mustern zu durchsuchen.

Vereinfacht wird der Nachbau insbesondere dadurch, dass auf die Möglichkeit regulärer Ausdrücke verzichtet wird. Stattdessen findet nur eine Suche nach enthaltenen Zeichenketten statt:

Programm 10.3: Ein vereinfachtes *grep* (*mygrep.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char *argv[]) {
6      char line[256];
7
8      if (argc != 2) {
9          fprintf(stderr, "Usage: %s pattern\n", argv[0]);
10         exit(1);
11     }
12
13     while (fgets(line, sizeof line, stdin)) {
14         if (strstr(line, argv[1])) {
15             fputs(line, stdout);
16         }
17     }
18 }
```

```
doolin$ gcc -Wall -std=c99 mygrep.c
doolin$ a.out strstr <mygrep.c
      if (strstr(line, argv[1])) {
doolin$ a.out line <mygrep.c
   char line[256];
   while (fgets(line, sizeof line, stdin)) {
      if (strstr(line, argv[1])) {
         fputs(line, stdout);
doolin$
```

An dieser Stelle ist es noch ärgerlich, dass mit einer maximalen Zeilenlänge von 256 Zeichen gearbeitet wird. Dies lässt sich – *realloc()* sei Dank – besser lösen mit einer kleinen Funktion *readline*, die beliebig lange Zeilen einlesen kann und für diese dynamisch ausreichend Speicher belegt:

Programm 10.4: Eine verbesserte Fassung von *grep* mit beliebig langen Eingabezeilen (*mygrep2.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  /* Lese eine beliebig lange Zeile von fp ein und
6      liefere einen Zeiger auf eine dynamisch organisierte
7      Speicherflaeche zurueck, in dem die Zeile abgelegt
8      worden ist
9  */
```

```
10 char* readline(FILE* fp) {
11     int len = 32;
12     char* cp = malloc(len);
13     if (!cp) return 0;
14     int i = 0;
15     int ch;
16     while ((ch = getc(fp)) != EOF && ch != '\n') {
17         cp[i++] = ch;
18         if (i == len) {
19             /* verdoppele die Speicherflaeche */
20             len *= 2;
21             char* newcp = realloc(cp, len);
22             if (!newcp) {
23                 free(cp);
24                 return 0;
25             }
26             cp = newcp;
27         }
28     }
29     if (i == 0 && ch == EOF) {
30         free(cp);
31         return 0;
32     }
33     cp[i++] = 0;
34     return realloc(cp, i);
35 }
36
37 int main(int argc, char *argv[]) {
38     if (argc != 2) {
39         fprintf(stderr, "Usage: %s pattern\n", argv[0]);
40         exit(1);
41     }
42
43     for (char* line; (line = readline(stdin)); free(line)) {
44         if (strstr(line, argv[1])) {
45             puts(line);
46         }
47     }
48 }
```

Nun liegt es nahe, den *grep*-Nachbau mit einigen Optionen zu bereichern. An dieser Stelle seien hier „-n"' und "'-v" ausgewählt, wobei „-n" die Ausgabe von Zeilennummern anfordert und „-v" darum bittet, alle Zeilen auszugeben, die die angegebene Zeichenkette *nicht* enthalten. Hinzu kommt noch die spezielle Option „–", die die Sequenz der Optionen beendet. Das folgende Argument wird so auch dann als Suchmuster akzeptiert, falls es mit einem Bindestrich beginnen sollte.

Programm 10.5: Ein vereinfachtes *grep* mit Optionen (*mygrep3.c*)

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

```
 5
 6  /* Lese eine beliebig lange Zeile von fp ein und
 7     liefere einen Zeiger auf eine dynamisch organisierte
 8     Speicherflaeche zurueck, in dem die Zeile abgelegt
 9     worden ist
10  */
11  char* readline(FILE* fp) {
12      int len = 32;
13      char* cp = malloc(len);
14      int i = 0;
15      int ch;
16      while ((ch = getc(fp)) != EOF && ch != '\n') {
17          cp[i++] = ch;
18          if (i == len) {
19              /* verdoppele die Speicherflaeche */
20              len *= 2;
21              char* newcp = realloc(cp, len);
22              if (!newcp) {
23                  free(cp);
24                  return 0;
25              }
26              cp = newcp;
27          }
28      }
29      if (i == 0 && ch == EOF) {
30          free(cp);
31          return 0;
32      }
33      cp[i++] = 0;
34      return realloc(cp, i);
35  }
36
37  int main(int argc, char *argv[]) {
38      const char* cmdname = argv[0]; /* Kommando-Name */
39      const char usage[] = "Usage: %s [-nv] pattern\n";
40      bool opt_v = false; /* Option -v gesetzt? */
41      bool opt_n = false; /* Option -n gesetzt? */
42
43      /* Optionen verarbeiten */
44      while (--argc > 0 && **++argv == '-') {
45          /* nach "--" kommen keine Optionen mehr (per Definition) */
46          if (argv[0][1] == '-' && argv[0][2] == '\0') {
47              argc--; argv++; break;
48          }
49          /* steht nur "-", dann fehlt etwas ;-) */
50          if (argv[0][1] == '\0') {
51              fprintf(stderr, "%s: empty option\n", cmdname);
52              fprintf(stderr, usage, cmdname);
53              exit(1);
54          }
55          for (char* s = *argv + 1; *s != '\0'; s++) {
56              switch (*s) { /* jede einzelne Option abarbeiten ... */
57                  case 'v' : opt_v = true; break;
```

```
58                  case 'n' : opt_n = true; break;
59                  default: /* unbekannte Option! */
60                      fprintf(stderr, "%s: illegal option '%c'\n", cmdname, *s);
61                      fprintf(stderr, usage, cmdname);
62                      exit(1);
63              }
64          }
65      }
66
67      /* Jetzt sollte nur noch das Suchmuster uebrig bleiben */
68      if (argc != 1) {
69          fprintf(stderr, usage, cmdname);
70          exit(1);
71      }
72      char* substring = argv[0];
73
74      int lineno = 1;
75      for (char* line; (line = readline(stdin)); free(line)) {
76          if (!strstr(line, substring) == opt_v) {
77              if (opt_n) {
78                  printf("%d: ", lineno);
79              }
80              puts(line);
81          }
82          lineno++;
83      }
84  }
```

```
doolin$ gcc -Wall -std=c99 mygrep3.c
doolin$ a.out -?
a.out: illegal option '?'
Usage: a.out [-nv] pattern
doolin$ a.out -n <mygrep3.c
Usage: a.out [-nv] pattern
doolin$ a.out -- -n <mygrep3.c
   const char usage[] = "Usage: %s [-nv] pattern\n";
   int opt_n = 0;           /* Option -n gesetzt? */
doolin$ a.out -n include <mygrep3.c
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
doolin$
```

Anmerkung: Bei Unix-Kommandos ist es üblich, dass man, anstatt zwei Optionen „-v"' und "'-n" getrennt anzugeben, diese auch kombiniert in der Form „-nv"' bzw. "'-vn" angeben kann.

Kapitel 11

Der Präprozessor

Der Präprozessor, der die Quelle noch vor der eigentlich Übersetzung bearbeitet, erledigt im Wesentlichen folgende Aufgaben:

- Einbinden von (Header-)Dateien
- Makro-Expansion
- Bedingte Übersetzung

Anweisungen für den Präprozessor werden *Präprozessor-Direktiven* genannt. Sie bestehen aus einem "'**#**'" ganz am Anfang einer Zeile, einer beliebigen Folge von Leerzeichen (kann auch entfallen), dem Namen der Direktive (wie etwa *define* oder *include*) und weiteren Parametern.

Bei der Betrachtung der einzelnen Direktiven ist in Erinnerung zu rufen, dass der Präprozessor als Filter arbeitet, d.h. aus der textuellen Eingabe des Quelltexts entsteht ein anderer Text, der an den Übersetzer weitergeht. Wenn keine Direktiven vorliegen, wird Programmtext unverändert weitergegeben. Abgesehen von den Hinweisen auf Dateinamen und Zeilennummern, die die Erzeugung von Fehlermeldungen erleichtern, werden keine Direktiven an den Übersetzer weitergereicht. Daher sind zwei Punkte wichtig:

- Welchen Text bekommt der Übersetzer an genau der Stelle zu sehen, an der zuvor die Direktive stand? Der Text, der an Stelle der Direktive steht, wird als *Ersatztext* bezeichnet.
- Welchen Einfluss hat die Direktive auf den übrigen Text?

11.1 Einbinden von Dateien

Die Direktive **#include** gibt es in zwei Varianten, je nach der Art der Einklammerung des Dateinamens. In Abhängigkeit davon ergibt sich der Suchpfad für die genannte Datei:

#include <dateiname>
: Suche nach *dateiname* nur in den Standardverzeichnissen wie etwa */usr/include*. Dieser Suchpfad kann auf der Kommandozeile des Übersetzers durch die Option „-I“ ergänzt werden.

#include "dateiname"
: Suche neben den Standardverzeichnissen zunächst im aktuellen Verzeichnissen und dann in den (möglicherweise durch „-I“ ergänzten) Standardverzeichnissen.

In beiden Fällen ist der Inhalt der gefundenen Datei der *Ersatztext* der Direktive. Ferner wird der Ersatztext ergänzt um Hinweise an den Übersetzer, von wo der Text stammt, damit Fehlermeldungen richtig zugeordnet werden können:

```
doolin$ cat a.h
int a();
doolin$ cat b.h
int b1();
int b2();
doolin$ cat ab.c
#include "a.h"
#include "b.h"

int main() {}
doolin$ gcc -E ab.c
# 1 "ab.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "ab.c"
# 1 "a.h" 1
int a();
# 2 "ab.c" 2
# 1 "b.h" 1
int b1();
int b2();
# 3 "ab.c" 2

int main() {}
doolin$
```

Es ist dabei allerdings zu beachten, dass Fehler für den Übersetzer gelegentlich zu spät und damit in der falschen Quelle erkannt werden. Im folgenden Beispiel gibt es zahlreiche Fehlermeldungen, die sich auf *ab.c* und *b.h* beziehen, obwohl nur ein Semikolon in *a.h* fehlt:

```
doolin$ sed 's/;//' <a.h >a.h.tmp && mv a.h.tmp a.h
doolin$ cat a.h
int a()
doolin$ gcc -Wall ab.c
ab.c: In function `a':
ab.c:4: warning: `main' is usually a function
ab.c:4: parse error before '{' token
ab.c:4: declaration for parameter `main' but no such parameter
b.h:2: declaration for parameter `b2' but no such parameter
b.h:1: declaration for parameter `b1' but no such parameter
doolin$
```

11.2 Makros

11.2.1 Definition und Verwendung von Makros

Mit der **#define**-Direktive können Makros mit und ohne textuelle Parameter definiert werden. Im Anschluss einer Makrodefinition wird überall, wo im folgenden Text der Makroname verwendet wird, ein Textersatz vorgenommen. Da der Präprozessor unabhängig von der eigentlichen Sprache C arbeitet, geschieht dies ohne Rücksicht auf die regulären Sichtbarkeitsregeln und Blockstrukturen von C.

Damit Makro-Aufrufe leichter als solche erkannt werden können, werden für Makronamen per Konvention keine Kleinbuchstaben verwendet. Allerdings gibt es zahlreiche Ausnahmen zu dieser Konvention und selbst die C-Bibliothek aus dem Standard hält sich nicht daran, wie etwa *getchar()* und *assert()* belegen.

Folgendes Programm illustriert die Verwendung von Makros für Konstanten und auch als „Unterprogramme“:

Programm 11.1: Definition und Verwendung von Makros (*makros.c*)

```
1  #include <stdio.h>
2
3  #define CONST 3
4  #define MAX(x,y) ((x) > (y)? (x): (y))
5
6  int main() {
7      int a = 4, b = CONST;
8
9      printf("Maximum von %d und %d: %d\n", a, b, MAX(a, b));
10 }
```

```
doolin$ gcc -Wall -std=c99 makros.c
doolin$ a.out
Maximum von 4 und 3: 4
doolin$
```

Anmerkungen: Die Parameter eines Makros haben keinen zugeordneten Typ. Somit findet bei der Ersetzung keine Typprüfung statt – sehr wohl aber danach durch den Übersetzer, aber eben auf dem Ergebnis der Ersetzung.

Vielfach wurde und wird die Verwendung von Makros im Vergleich zu regulären Funktionen bevorzugt, um durch das Einsparen eines Aufrufs einen Effizienz-Gewinn zu erzielen. Alternativ bietet hier C99 auch sogenannte **inline**-Funktionen an, die den gleichen Effekt erzielen können – aber eben mit den gewohnten Vorteilen von Blockstrukturen und Parameterüberprüfungen.

Da Makroaufrufe durch den Präprozessor vollständig durch Textersetzung durchgeführt werden, ist eine Rekursion nicht möglich.

11.2.2 Fallen und Fehlerquellen

Es gibt eine ganze Reihe von Fehlerquellen und Stolperfallen bei **#define**. Diese werden am folgenden Beispiel erläutert:

```
thales$ cat define.txt
#define SIZE 10;
#define MAXLEN = 10
#define QUADRAT(x)      x * x
#define SQUARE(x)       (x) * (x)
#define QUADRIERE(x)    ((x) * (x))
#define TEST (x)        ((x) != 0)

a1) x = SIZE;
a2) int a[SIZE];
b)  int a[MAXLEN];
c)  QUADRAT(x+1)
d1) SQUARE(x+1)
d2) !SQUARE(0)
e1) !QUADRIERE(0)
e2) QUADRIERE(++x)
f)  TEST(2)
thales$
```

Aus obiger Datei erzeugt der Preprozessor die folgende Ausgabe:

```
thales$ cpp define.txt
# ... verkuerzte Ausgabe ;-)
a1) x = 10;;
a2) int a[10;];
b) int a[= 10];
c) x+1 * x+1
d1) (x+1) * (x+1)
d2) !(0) * (0)
e1) !((0) * (0))
e2) ((++x) * (++x))
f) (x) ((x) != 0)(2)
thales$
```

Erläuterungen:

a1) Obwohl bei der Definition des Makros SIZE wohl der Strichpunkt nicht gewollt war, bewirkt hier der doppelte Strichpunkt nur, dass eine leere Anweisung hinzu kommt.

a2) In diesem Fall führt der versehentlich hinzugefügte Strichpunkt aber zu einem Übersetzungsfehler, der aber möglicherweise schwer zu finden ist, wenn nicht die Ausgabe des Präprozessors analysiert wird.

b) Genauso wie bei a2) fällt das „=“ hier durch einen Übersetzungsfehler auf, der ebenfalls fast nur durch das Betrachten der Ausgabe des Präprozessors zu finden ist.

c) Als Parameter können nicht nur Variablen übergeben werden, sondern beliebiger Text – in diesem Falle ein arithmetischer Ausdruck. Allerdings dürfte die Überraschung gross sein, wenn keine implizite Klammerung erfolgt und somit die Vorränge erst nach dem Textersatz eine Rolle spielen.

d1) Deswegen ist es sinnvoll, alle Parameter innerhalb des Makro-Ersatztextes grundsätzlich zu klammern wie dies *SQUARE* demonstriert.

d2) Aber das Klammern der Argumente reicht noch nicht aus, da der Ersatztext wiederum sich in einen anderen Ausdruck einbetten kann. Hier z. B. hat der Operator ! höhere Prioriät als der Operator $*$. Daher wird nur der erste Faktor komplementiert; der gesamte Ausdruck ist entsprechend äquivalent zu $(!(0))*(0)$.

e1) Das Problem kann vermieden werden, wenn grundsätzlich auch der vollständige Ausdruck des Ersatztextes eingeklammert wird – wie dies hier bei *QUADRIERE* demonstriert wird.

e2) Dessen ungeachtet kann eine Mehrfach-Verwendung eines Makroparameters innerhalb des Ersatztextes einen mehrfachen Seiteneffekt nach sich ziehen.

f) Zwischen dem Namen des Makros und der Liste der formalen Parameter dürfen keine Leerzeichen stehen, da sonst die Parameterliste dem Ersatztext zugeordnet wird.

11.2.3 Makrodefinition auf der Kommandozeile

Grundsätzlich können Makrodefinitionen auch auf der Kommandozeile des Übersetzers erfolgen. Dies ist möglich über die Option *-D*, der unmittelbar die Makrodefinition mit einem Gleichheitszeichen als Trenner folgt. So ist beispielsweise *-DCONST=3* äquivalent zu der Direktive **#define** CONST 3.

Im folgenden Beispiel lassen sich auf diese Weise die Ausgabetexte zur Verfolgung des Programmverlaufs wahlweise ein- und ausschalten in Abhängigkeit von dieser Option auf der Kommandozeile:

Programm 11.2: Verfolgung des Programmverlaufs mit Hilfe von Makros (*debug.c*)

```
1 #include <stdio.h>
2
3 int main() {
4     DEBUG("Start");
5     puts("... ich arbeite ...");
6     DEBUG("Ende");
7 }
```

Wenn die Ausgabetexte nicht erwünscht sind, so kann ganz einfach ein leerer Ersatztext spezifiziert werden:

```
doolin$ gcc -Wall -std=c99 -D'DEBUG(x)=' debug.c
doolin$ a.out
... ich arbeite ...
doolin$
```

Andernfalls kann die Zeichenkette etwa *puts* zur Ausgabe übermittelt werden, so dass diese sichtbar werden:

```
doolin$ gcc -Wall -std=c99 -D'DEBUG(x)=(puts(x))' debug.c
doolin$ a.out
Start
... ich arbeite ...
Ende
doolin$
```

11.2.4 Zurücknahme einer Makrodefinition

Grundsätzlich kann ein Makro durch die Verwendung von Neu-Definitionen verschiedene Werte annehmen. Allerdings führt dies zu Warnungen, wie dies durch folgendes Beispiel belegt wird:

Programm 11.3: Mehrfache Definition eines Makros (*redef.c*)

```
1  #include <stdio.h>
2
3  #define X 1
4  int x1 = X;
5  #define X 2
6  int x2 = X;
7
8  int main() {
9      printf("x1 = %d, x2 = %d\n", x1, x2);
10 }
```

```
doolin$ gcc -Wall -std=c99 redef.c
redef.c:3:1: warning: "X" redefined
redef.c:1:1: warning: this is the location of the previous definition
redef.c: In function `main':
redef.c:7: warning: implicit declaration of function `printf'
doolin$ a.out
x1 = 1, x2 = 2
doolin$
```

In Fällen, bei denen dies ausdrücklich erwünscht wird, kann zur Vermeidung der Warnung zunächst die alte Definition explizit mit Hilfe der **#undef**-Direktive zurückgenommen werden, bevor dann ohne weitere Warnung eine Neu-Definition möglich ist:

Programm 11.4: Zurücknahme einer Makrodefinition (*undef.c*)

```
1  #include <stdio.h>
2
3  #define X 1
4  int x1 = X;
5  #undef X
6  #define X 2
7  int x2 = X;
8
9  int main() {
10     printf("x1 = %d, x2 = %d\n", x1, x2);
11 }
```

```
doolin$ gcc -Wall -std=c99 undef.c
doolin$ a.out
x1 = 1, x2 = 2
doolin$
```

11.2.5 Vordefinierte Makros

Es gibt einige vordefinierte Makros, wie beispielsweise *__FILE__* oder *__LINE__*, die durch den Namen der Quelltextdatei bzw. die Zeilennummer des Makroaufrufs ersetzt werden. Dies wird verwendet, um Makros wie etwa *assert* aus <*assert.h*> zu definiert. Dies kann aber auch selbst erledigt werden wie folgendes Beispiel demonstriert:

Programm 11.5: Zusicherungen überprüfen (*assert.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define ASSERT(cond) \
5      ( \
6          (cond)? 1: ( \
7              fprintf(stderr, "%s␣(%d):␣assertion␣\"%s\"␣failed\n", \
8                  __FILE__, __LINE__, #cond), \
9              abort() \
10         ) \
11     )
12
13 int main() {
14     int a = 3;
15     ASSERT(a < 0);
16 }
```

```
doolin$ gcc -Wall -std=c99 assert.c
doolin$ a.out
assert.c (15): assertion "a < 0" failed
Abort(coredump)
doolin$
```

Anmerkungen: Obiges Beispiel demonstriert auch die Möglichkeit, den Ersatztext eines umfangreichen Makros mit Hilfe des „\“ in mehrere Zeilen zu brechen. Konkret wird ein Zeilentrenner, dem unmittelbar ein „\“ vorausgeht, durch leeren Text ersetzt.

Ein weitere Technik ist das Voranstellen des „#“ vor einem Parameternamen. In diesem Falle wird der zum Parameter gehörende Text in Anführungszeichen gesetzt. Diese Technik ist notwendig, um übergebene Texte in Zeichenketten zu verwandeln, da der Präprozessor keinen Textersatz innerhalb einer Zeichenkettenkonstanten durchführt.

11.3 Bedingte Übersetzung

Die Direktiven **#ifdef**, **#ifndef** und **#if**, die sich bis zum zugehörigen **#endif** erstrecken, erlauben es, Text in Abhängigkeit von Bedingungen an den Übersetzer weiterzuleiten oder dies zu unterlassen.

11.3.1 Test auf Makro-Existenz

Im einfachsten Falle dient dies dazu, eine Definition davon abhängig zu machen, ob bereits eine Definition auf der Kommandozeile angegeben wurde:

Programm 11.6: Überdefinierbare Makros (*debug2.c*)

```
1  #include <stdio.h>
2
3  #ifndef DEBUG
4  # define DEBUG(x)
5  #endif
6
```

```
7  int main() {
8      DEBUG("Start");
9      puts("... ich arbeite ...");
10     DEBUG("Ende");
11 }
```

```
#include <stdio.h>

#ifndef DEBUG
#  define DEBUG(x)
#endif

int main() {
   DEBUG("Start");
   puts("... ich arbeite ...");
   DEBUG("Ende");
}
```

Anmerkungen: Die Direktive **#ifndef** (für *if not defined* stehend) hat als Ersatztext den leeren Text, falls der angegebene Makroname definiert ist. Andernfalls wird als Ersatztext der gesamte Text bis vor dem **#endif** verwendet. Um die durch Bedingungen erzeugte Schachtelstruktur besser erkennen zu können, ist es üblich, bei inneren Direktiven einige Leerzeichen zwischen dem „#“ und dem Namen der Direktive entsprechend der Verschachtelungstiefe einzufügen.

Folgendes Beispiel geht noch einen Schritt weiter und bietet eine Voreinstellung für das Makro *DEBUG* an in Abhängigkeit davon, ob das Makro *DBG* (auf der Kommandozeile) definiert wurde oder nicht. Dabei ist die Konstruktion so tolerant, dass *DEBUG* immer noch überdefiniert werden kann. Falls aber *DBG* undefiniert bleibt, dann wird in jedem Falle sichergestellt, dass *DEBUG* durch den leeren Text ersetzt wird:

Programm 11.7: Makrodefinitionen in Abhängigkeit von der Kommandozeile (*debug3.c*)

```
1  #include <stdio.h>
2
3  #ifdef DBG
4  # ifndef DEBUG
5  # define DEBUG(x) (puts((x)))
6  # endif
7  #else
8  # undef DEBUG
9  # define DEBUG(x)
10 #endif
11
12 int main() {
13     DEBUG("Start");
14     puts("... ich arbeite ...");
15     DEBUG("Ende");
16 }
```

```
doolin$ gcc -Wall -std=c99 debug3.c
doolin$ a.out
... ich arbeite ...
doolin$ gcc -Wall -std=c99 -DDBG debug3.c
doolin$ a.out
Start
... ich arbeite ...
Ende
doolin$ gcc -Wall -std=c99 \
>  -DDBG -D'DEBUG(x)=(fprintf(stderr, "DEBUG: %s\n", (x)))' debug3.c
doolin$ a.out
DEBUG: Start
... ich arbeite ...
DEBUG: Ende
doolin$
```

11.3.2 Weitere Tests

Neben Tests **#ifdef** und **#ifndef**, die nur die Existenz bzw. Nicht-Existenz einer Makrodefinition überprüfen ist es auch möglich, mit **#if** ganzzahlige Ausdrücke unter Verwendung der gewohnten C-Operatoren anzugeben. Analog zu C wird dabei der Wert 0 als **false** und jeder andere Wert als **true** interpretiert. Neben Konstanten können auch andere Makros mit einem numerischen Wert spezifiziert werden. Die Verwendung unbekannter Makronamen ist ebenfalls zulässig. In diesem Fall wird der Wert 0 angenommen. Neben **#else** wird auch für entsprechende Bedingungsketten auch **#elif** unterstützt, das für *else if* steht.

Das folgende Programm illustriert die Verwendung der beschriebenen Direktiven:

Programm 11.8: Bedingte Übersetzung (*if.c*)

```
1  #include <stdio.h>
2
3  int main() {
4  #if x == 1
5      puts("1");
6  #elif x == 2
7      puts("2");
8  #else
9      puts("anderer␣Wert␣oder␣nicht␣definiert");
10 #endif
11 }
```

```
thales$ gcc -Wall if.c
thales$ a.out
anderer Wert oder nicht definiert
thales$ gcc -Wall -Dx=1 if.c
thales$ a.out
1
thales$ gcc -Wall -Dx=2 if.c
thales$ a.out
2
thales$ gcc -Wall -Dx=3 if.c
thales$ a.out
anderer Wert oder nicht definiert
thales$
```

Kapitel 12

Modularisierung

12.1 Deklaration vs. Definition

Bei *Funktionen und Variablen* wird zwischen *Deklarationen* und *Definitionen* unterschieden. Eine Deklaration teilt dem Übersetzer alle notwendigen Informationen mit, so dass die entsprechende Variable oder Funktion anschließend verwendet werden kann. Eine Definition führt im Falle von Variablen dazu, dass entsprechend Speicherplatz belegt wird und, falls erwünscht, eine Initialisierung erfolgt. Eine Funktions-Definition schließt den Programmtext der Funktion mit ein.

Einige Beispiele:

Programm 12.1: Beispiele für Deklarationen und Definitionen (*deklvsdef.c*)

```
1  int f(int a, int b); /* eine Funktions-Deklaration */
2
3  /* eine Funktions-Definition */
4  int max(int a, int b) {
5      return a > b? a: b;
6  }
7
8  /* eine Variablendefinition */
9  int i = 27;
```

Wie kann jedoch eine Variable nur deklariert werden ohne sie zu definieren? Hierfür gibt es das Schlüsselwort **extern**, das einer Deklaration vorausgeht. Prinzipiell dürfte dieses Schlüsselwort auch Funktionsdeklarationen vorausgehen – aber dort ergibt sich der Unterschied aus Deklaration und Definition bereits aus der Abwesen- bzw. Anwesenheit des zugehörigen Programmtexts.

Hier ist ein Beispiel für eine Variablendeklaration:

Programm 12.2: Beispiel für eine Variablendeklaration (*extern.c*)

```
1  /* eine Variablen-Deklaration */
2  extern int i;
```

Im Falle einer Funktions- oder Variablen-Deklaration akzeptiert der Übersetzer eine darauf folgende Verwendung der entsprechenden Funktion oder Variablen. Dennoch muss die jeweilige Funktion oder Variable irgendwo definiert werden. Unterbleibt dies, so gibt es eine Fehlermeldung bei einem Versuch, das Programm zusammenzubauen.

12.2 Aufteilung eines Programms in Übersetzungseinheiten

Grundsätzlich kann ein C-Programm in beliebig viele Übersetzungseinheiten aufgeteilt werden. Dabei ist jede Übersetzungseinheit eine Folge von Deklarationen und Definitionen.

Das folgende Beispiel besteht aus zwei Übersetzungseinheiten *main.c* und *lib.c*. (Konventionellerweise erhält jede getrennt übersetzbare C-Quelle die Dateiendung „.c“.) In *lib.c* finden sich die Definitionen der globalen Funktion *f* und der globalen Variablen *i*. In *main.c* werden sowohl *i* als auch *f* deklariert. Eine Definition ist hier nur für *main* gegeben:

Programm 12.3: Übersetzungseinheit mit Deklarationen fremder Variablen und Funktionen (*main.c*)

```
1  #include <stdio.h>
2
3  /* Deklarationen */
4  extern int i;
5  extern void f();
6
7  int main() {
8      printf("Wert␣von␣i␣vor␣dem␣Aufruf:␣%d\n", i);
9      f();
10     printf("Wert␣von␣i␣nach␣dem␣Aufruf:␣%d\n", i);
11 }
```

Programm 12.4: Übersetzungseinheit mit extern nutzbaren Definitionen (*lib.c*)

```
1  int i = 1; /* Definition */
2
3  void f() { /* Definition */
4      ++i;
5  }
```

12.3 Zusammenbau mehrerer Übersetzungseinheiten

Beide Übersetzungseinheiten können nun in beliebiger Reihenfolge unabhängig voneinander übersetzt werden. Die Option „*-c*“ bittet hier den Übersetzer darum, nur zu Übersetzen und noch kein fertiges ausführbares Programm zu erstellen. Dabei entstehen sogenannte Objekt-Dateien mit der Dateiendung „.o“. Diese enthalten den aus der Quelle erzeugten Maschinen-Code und eine Symboltabelle, die das spätere Zusammenbauen mit anderen Objekten erlaubt. Diese Symboltabellen können mit dem Kommando *nm* betrachtet werden:

```
clonard$ nm lib.o
00000000 a *ABS*
00000000 T f
00000000 D i
clonard$ nm main.o
00000000 a *ABS*
         U f
         U i
00000000 T main
         U printf
clonard$
```

Die Ausgabe von *nm* ist dreispaltig: Links steht der vorläufige dem Symbol zugeordnete Wert, in der Mitte der Typ des Symbols und rechts das Symbol selbst. Bei den Typen steht „T“ für ein in diesem Objekt definitiertes Symbol, das in den Programmtext verweist (typischerweise eine Funktion), „D“ für ein definiertes Symbol, das in den globalen Variablenbereich verweist (typischerweise eine globale Variable) und „U“ für ein deklariertes, aber bislang undefiniert gebliebenes Symbol. Der Ausgabe von *nm* lässt sich in dem Beispiel entsprechend entnehmen, dass in dem Objekt *lib.o* die Symbole *f* und *i* definiert werden, während in *main.o* nur das Symbol *main* definiert wird und *f* und *i* undefiniert bleiben.

Objekte können mit dem Binder, unter UNIX *ld* (*linkage editor*) genannt, zu einem ausführbaren Programm zusammengebaut werden. Dies gelingt nur dann, wenn es zu jedem deklarierten Symbol *genau eine* Definition gibt. Im folgenden Beispiel führt der direkte Aufruf von *ld* nicht zum Ziel, weil keine Definition für *printf* vorliegt. Wird stattdessen *gcc* aufgerufen für den Zusammenbau dann wird zwar ebenfalls nur *ld* aufgerufen, wobei jedoch implizit noch die C-Bibliothek hinzugefügt wird, so dass dann die Definition von *printf* gefunden wird:

```
clonard$ ld main.o lib.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000000010074
main.o: In function `main':
main.o(.text+0x18): undefined reference to `printf'
main.o(.text+0x3c): undefined reference to `printf'
clonard$ gcc main.o lib.o
clonard$ a.out
Wert von i vor dem Aufruf: 1
Wert von i nach dem Aufruf: 2
clonard$
```

Dabei ist zu beachten, dass der Zusammenbau eines Programms durch den Binder nur auf den Symbolen beruht. Eine semantische Information wie etwa der zugehörige Typ liegt dem Binder nicht vor und kann somit auch nicht berücksichtigt werden. Entsprechend gibt es in C keine Sicherheit, dass zusammengebaute Teile auch zusammenpassen. Dies wird durch folgendes Beispiel belegt, bei dem *f* in *m1.c* als Funktion deklariert wird und in *m2.c* als Variable des Typs **double** deklariert wird:

```
clonard$ cat m1.c
extern void f();
int main() {
   f();
}
clonard$ cat m2.c
double f = 1.0;
clonard$ gcc -Wall -std=c99 -c m1.c
clonard$ gcc -Wall -std=c99 -c m2.c
clonard$ gcc m1.o m2.o
clonard$ a.out
Illegal Instruction(coredump)
clonard$
```

Bei keinem der beiden Übersetzungsläufe kann der Übersetzer die Inkonsistenz entdecken und für den Binder sind die Unterschiede irrelevant, so dass ohne jegliche Warnungen ein Zusammenbau möglich ist. Die Ausführung des Programms scheitert in diesem Beispiel an dem Versuch, die *double*-Variable als Funktion auszuführen. Aber ein Absturz ist nicht garantiert – insbesondere, wenn die Fehler sehr viel subtiler sind mit nicht überein stimmenden Parameterzahlen oder Typen bei Variablen, Rückgabewerten und Parametern.

12.4 Herstellung der Schnittstellensicherheit in C

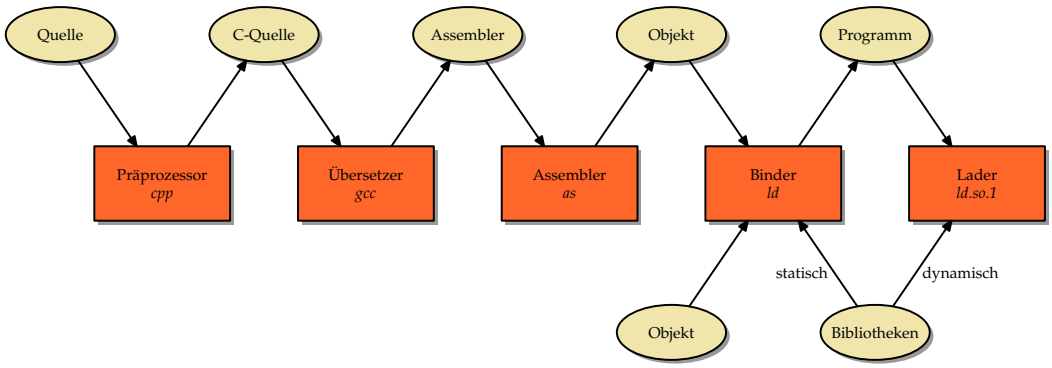

Abbildung 12.1: Der Weg von der Quelle zum ausführbaren Programm

Zusammengefasst zeigt die Abbildung 12.1 den Weg von der Quelle zum ausführbaren Programm. Hierbei ist zu beachten, dass in der langen Kette nur der Übersetzer mit der Semantik von C soweit vertraut ist, dass Überprüfungen von Funktionsaufrufen und Variablenverwendungen möglich sind. Dies steht im deutlichen Kontrast zu Java, Oberon oder Modula-2, die jeweils Schnittstellensicherheit bis zur Laufzeit hin garantieren – selbst bei dynamisch nachgeladenen Modulen.

Dennoch gibt es bei der Verwendung von C einige Techniken, die, wenn sie konsequent und diszipliniert angewendet werden, weitgehend helfen, nicht zusammenpassende Schnittstellen zu erkennen.

12.4.1 Auslagerung von Deklarationen in Header-Dateien

Der erste und wichtigste Schritt ist die Auslagerung aller Deklarationen externer Variablen und Funktionen in Header-Dateien. Dies vermeidet nicht nur die Existenz voneinander ab-

weichender Deklarationen in unterschiedlichen Quellen, sondern ermöglicht die Überprüfung sowohl der Funktionsaufrufe oder Variablenverwendungen als auch der Funktions- oder Variablendefinitionen gegen die gleiche Deklaration. Wenn die Verwendung und auch die Definition einer Funktion oder Variablen zu der gleichen Deklaration konsistent sind, dann passen sie zusammen.

Folgendes Beispiel demonstriert dies für eine Funktion zur Berechnung des größten gemeinsamen Teilers, die von dem Hauptprogramm getrennt wurde:

Programm 12.5: Gemeinsam genutzte Header-Datei mit der Deklaration der ggf-Funktion (*ggt.h*)

```
1  int ggt(int a, int b);
```

Programm 12.6: Getrennt übersetzbare Funktion zur Berechnung des ggT (*ggt.c*)

```
1  #include "ggt.h"
2
3  int ggt(int a, int b) {
4      while (a != b) {
5          if (a > b) {
6              a -= b;
7          } else {
8              b -= a;
9          }
10     }
11     return a;
12 }
```

Programm 12.7: Getrennt übersetzbares Hauptprogramm zur Berechnung des ggT (*ggt-main.c*)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "ggt.h"
4
5  int main(int argc, char* argv[]) {
6      char* cmdname = *argv++; --argc;
7      char usage[] = "Usage: %s a b\n";
8      if (argc != 2) {
9          fprintf(stderr, usage, cmdname);
10         exit(1);
11     }
12     int a = atoi(argv[0]);
13     int b = atoi(argv[1]);
14     if (a > 0 && b > 0) {
15         printf("%d\n", ggt(a, b));
16     } else {
17         fprintf(stderr, usage, cmdname);
18         exit(1);
19     }
20 }
```

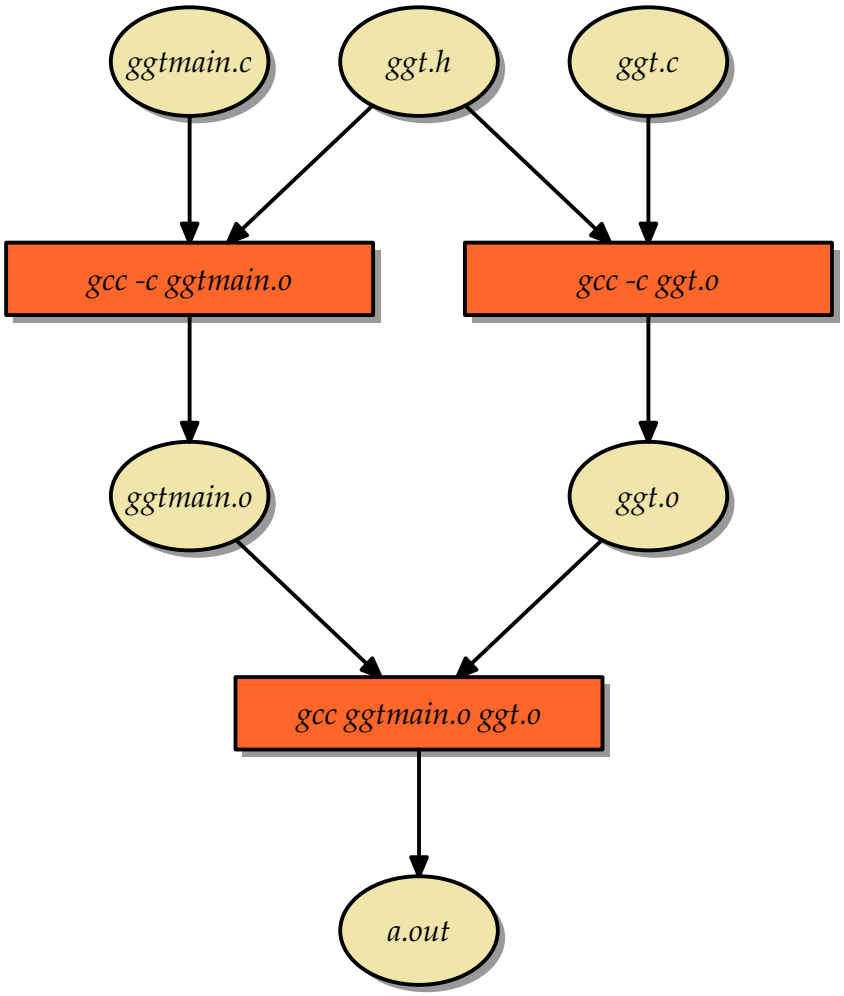

Abbildung 12.2: Übersetzungsvorgang bei ausgelagerten Deklarationen

Obwohl die Übersetzungen von *ggt.c* und *ggtmain.c* unabhängig voneinander erfolgen, gibt es mit *ggt.h* einen gemeinsamen Teil, der bei beiden Übersetzungen vorliegt (siehe Abbildung 12.2). Innerhalb von *ggt.h* wird die Funktion *ggt* deklariert. Wenn *ggt.c* übersetzt wird, so kann diese Deklaration mit der darauf folgenden Definition verglichen werden. Abweichungen führen hier zu einer Fehlermeldung. Analog wird der Aufruf der Funktion *ggt* innerhalb von *ggtmain.c* anhand der vorliegenden Deklaration überprüft. Auf diese Weise lässt sich sicherstellen, dass *ggtmain.o* und *ggt.o* in Bezug auf die Funktion *ggt* zueinander passen, wenn sie vom Binder zu *a.out* zusammengebaut werden:

```
doolin$ gcc -Wall -std=c99 -c ggtmain.c
doolin$ gcc -Wall -std=c99 -c ggt.c
doolin$ gcc ggtmain.o ggt.o
doolin$ a.out 36 48
12
doolin$
```

12.4.2 Neu-Übersetzungen unter Berücksichtigung der Abhängigkeiten

Die vorgestellte Lösung funktioniert nur unter einer wichtigen Voraussetzung: Wenn eine Header-Datei verändert wird, müssen alle Übersetzungseinheiten neu übersetzt werden, die diese Header-Datei direkt oder indirekt über **#include** einbeziehen. Unterbleibt dieses, dann ist die Schnittstellensicherheit nicht mehr gewährleistet, da dann die einzelnen Übersetzungseinheiten mit nicht zueinander kompatiblen Versionen einer Header-Datei übersetzt sein könnten. Die Gefahr ist durchaus gegeben, da bei einer umfangreichen Zahl von Übersetzungseinheiten und Header-Dateien es rasch unübersichtlich wird, welche C-Quellen wegen einer Schnittstellenänderung neu zu übersetzen sind.

Für dieses Problem wurde 1977 in den Bell Laboratories das Werkzeug *make* von Stuart Feldman entwickelt. Alle heutigen Varianten dieses Werkzeugs (wovon es nicht wenige gibt) haben die wesentlichen Eigenschaften des Originals übernommen. Die Grundidee liegt darin, eine Datei namens *makefile* (oder *Makefile*) zusammen mit den Quellen im gleichen Verzeichnis anzulegen, das

- die Abhängigkeiten der Dateien (sowohl Quellen als auch die erzeugbaren Dateien) spezifiziert und
- angibt, mit welchen Kommandos bei Bedarf die erzeugbaren Dateien hergestellt werden können.

Die Abhängigkeiten und Erzeugungskommandos können für das vorgestellte Beispiel wie folgt in einem *makefile* repräsentiert werden:

Programm 12.8: Einfaches makefile für das ggT-Beispiel (*makefile*)

```
1 a.out: ggt.o ggtmain.o
2                 gcc ggt.o ggtmain.o
3 ggt.o: ggt.h ggt.c
4                 gcc -c -Wall -std=c99 ggt.c
5 ggtmain.o: ggt.h ggtmain.c
6                 gcc -c -Wall -std=c99 ggtmain.c
```

Zeilen, die nicht mit einem Tabulator-Zeichen beginnen, nennen eine erzeugbare Datei. Es folgen ein Doppelpunkt, ggf. Leerzeichen und dann (noch auf der gleichen Zeile) die Dateinamen, wovon die erzeugbare Datei abhängt. Im konkreten Fall hängt beispielsweise *a.out* von den Dateien *ggt.o* und *ggtmain.o* ab. Die darauffolgenden Zeilen, die jeweils mindestens mit einem führenden Tabulator-Zeichen beginnen müssen, enthalten die Kommandos mit denen die zuvor genannte Datei erzeugt werden kann. Dabei darf jeweils vorausgesetzt werden, dass die Dateien, wovon die erzeugende Datei abhängt, bereits existieren. Im obigen Beispiel wird so zu Beginn das Kommando *gcc ggt.o ggtmain.o* angegeben, das die Datei *a.out* erzeugen kann, sobald *ggt.o* und *ggtmain.o* vorliegen.

Beliebig vieler solcher erzeugbaren Dateien, deren Abhängigkeiten und die zugehörigen erzeugenden Kommandos können in einem *makefile* aufgeführt werden. Wenn das Kommando *make* aufgerufen wird, kann entweder die gewünschte Datei auf der Kommandozeile angegeben werden oder es wird implizit die erste im *makefile* genannte erzeugbare Datei ausgewählt (im Beispiel wäre das *a.out*).

Das Werkzeug *make* geht dann beginnend mit dem gegebenen Ziel rekursiv wie folgt vor:

1. Sei Z das Ziel. Wenn das Ziel im *makefile* nicht explizit genannt ist, jedoch als Datei existiert, dann ist nichts weiter zu tun. (Falls das Ziel wieder als Datei noch als Regel existiert, dann gibt es eine Fehlermeldung.)

2. Andernfalls ist innerhalb des *makefile* eine Abhängigkeit gegeben in der Form

   $$Z : A_1 \ldots A_n,$$

   wobei die Folge $\{A_i\}_1^n$ leer sein kann ($n = 0$). Dann ist der Algorithmus (beginnend mit Schritt 1) rekursiv aufzurufen für jede der Dateien $A_1 \ldots A_n$.

3. Sobald alle Dateien $A_1 \ldots A_n$ in aktueller Form vorliegen, wird überprüft, ob der Zeitstempel von Z (letztes Schreibdatum) jünger ist als der Zeitstempel jede der Dateien $A_1 \ldots A_n$.

4. Falls es ein A_i gibt, das neueren Datums ist als Z, dann werden die zu Z gehörenden Kommandos ausgeführt, um Z neu zu erzeugen.

So sieht für das obige Beispiel ein Aufruf von *make* aus, wenn alle erzeugbaren Dateien (*a.out*, *ggt.o* und *ggtmain.o*) fehlen:

```
doolin$ make
gcc -c -Wall -std=c99 ggt.c
gcc -c -Wall -std=c99 ggtmain.c
gcc ggt.o ggtmain.o
doolin$ make
make: `a.out' is up to date.
doolin$ a.out 48 112
16
doolin$
```

Hier ist auch zu sehen, dass bei einem unmittelbar folgenden zweiten Aufruf von *make* nichts passiert, da *a.out* immer noch existiert und aktuell ist. Das Werkzeug *make* kann auch darum gebeten werden, die genaue Vorgehensweise zu dokumentieren:

```
doolin$ make --debug | sed -n '/^Updating/,$p'
Updating goal targets....
 File `a.out' does not exist.
   File `ggt.o' does not exist.
  Must remake target `ggt.o'.
gcc -c -Wall -std=c99 ggt.c
  Successfully remade target file `ggt.o'.
   File `ggtmain.o' does not exist.
  Must remake target `ggtmain.o'.
gcc -c -Wall -std=c99 ggtmain.c
  Successfully remade target file `ggtmain.o'.
Must remake target `a.out'.
gcc ggt.o ggtmain.o
Successfully remade target file `a.out'.
doolin$
```

Interessant ist dann der Fall, wenn die durch *ggt.h* repräsentierte Schnittstelle aktualisiert werden, aber alle erzeugbaren Dateien existieren:

```
doolin$ touch ggt.h
doolin$ make --debug | sed -n '/^Updating/,$p'
Updating goal targets....
   Prerequisite `ggt.h' is newer than target `ggt.o'.
  Must remake target `ggt.o'.
gcc -c -Wall -std=c99 ggt.c
  Successfully remade target file `ggt.o'.
   Prerequisite `ggt.h' is newer than target `ggtmain.o'.
  Must remake target `ggtmain.o'.
gcc -c -Wall -std=c99 ggtmain.c
  Successfully remade target file `ggtmain.o'.
 Prerequisite `ggt.o' is newer than target `a.out'.
 Prerequisite `ggtmain.o' is newer than target `a.out'.
Must remake target `a.out'.
gcc ggt.o ggtmain.o
Successfully remade target file `a.out'.
doolin$
```

12.4.3 Extraktion der Abhängigkeiten

Eines der verbliebenen Probleme ist die korrekte Extraktion der Abhängigkeiten, d.h. welche Header-Dateien werden direkt (oder indirekt!) von einer C-Quelle über **#include** einbezogen. Hierfür gibt es mehrere Werkzeuge für C, wovon *makedepend* den größten Verbreitungsgrad hat. Beim *gcc* bietet sich hier auch die Option „*-M*“ an.

Im einfachsten Falle wird *makedepend* mit allen C-Quellen aufgerufen (nur die Dateien, die auch an *gcc* übergeben werden, jedoch nicht die Header-Dateien). Dann werden die Abhängigkeiten an das Ende der *makefile*-Datei gehängt:

```
doolin$ ls
ggt.c  ggt.h  ggtmain.c  makefile
doolin$ cat makefile
a.out:          ggt.o ggtmain.o
                gcc ggt.o ggtmain.o
doolin$ makedepend *.c
doolin$ cat makefile
a.out:          ggt.o ggtmain.o
                gcc ggt.o ggtmain.o
# DO NOT DELETE

ggt.o: ggt.h
ggtmain.o: /usr/include/stdio.h /usr/include/sys/feature_tests.h
ggtmain.o: /usr/include/sys/ccompile.h /usr/include/sys/isa_defs.h
ggtmain.o: /usr/include/iso/stdio_iso.h /usr/include/sys/va_list.h
ggtmain.o: /usr/include/stdio_tag.h /usr/include/stdio_impl.h
ggtmain.o: /usr/include/iso/stdio_c99.h /usr/include/stdlib.h
ggtmain.o: /usr/include/iso/stdlib_iso.h /usr/include/iso/stdlib_c99.h ggt.h
doolin$
```

Im Vergleich zu vorher kommen jetzt auch weitere Header-Dateien aus */usr/include* hinzu. Das ist vorteilhaft, da sich diese Header-Dateien prinzipiell (etwa durch eine Aktualisierung des Systems) zusammen mit den Bibliotheken ebenso verändern können. Die speziell eingefügte Kommentarzeile markiert dann den Teil des *makefile*, der von weiteren Aufrufen von *makedepend* anzupassen ist. Entsprechend sollten alle eigenen Änderungen *davor* erfolgen.

Ferner ist im Vergleich zu zuvor die explizite Nennung der Abhängigkeiten der Objekte von den zugehörigen C-Quellen verschwunden. Dies ist aufgrund der impliziten Regeln von *make* ohne Verlust möglich. Es gibt zahlreiche implizite Regeln bei *make*, um den Tippaufwand zu reduzieren.

Die Prozedur wird üblicherweise vereinfacht, indem ein zusätzliches Ziel für das Aktualisieren des *makefile* eingefügt wird. Zwar könnte das Ziel dann sinnvollerweise auch *makefile* heissen, aber traditionellerweise wird hierfür der Zielname *depend* verwendet, den es nicht als Datei gibt. Grundsätzlich müssen Ziele in *make* nicht unbedingt real existierende Dateien seien. In diesem Falle muss das Ziel immer wieder aufs neue erzeugt werden. Bei GNU-make können solche Pseudo-Ziele auch explizit deklariert werden, indem sie als Abhängigkeit des speziellen Ziels *.PHONY* genannt werden:

Programm 12.9: Allgemeine makefile-Vorlage mit Unterstützung von *makedepend* (*makefilevorlage*)

```
1  Sources := $(wildcard *.c)
2  Objects := $(patsubst %.c,%.o,$(Sources))
3  Target := ggt
4  CC := gcc
5  CFLAGS := -Wall -std=c99
6  $(Target): $(Objects)
7                  gcc -o $(Target) $(Objects)
8  .PHONY: depend
9  depend:
10                 makedepend -- $(CFLAGS) -- $(Sources)
```

Hier stehen jetzt zu Beginn einige Variablenzuweisungen wie etwa an *Sources* und *Objects*, die danach mit vorausgehendem Dollar-Zeichen und eingeklammert referenziert werden können: *$(Sources)*und *$(Objects)*.Die Variablen *CC* und *CFLAGS* werden von den impliziten C-Übersetzungsregeln von *make* benutzt. Sie enthalten den Befehl zum Übersetzen und die Optionen, die beim Übersetzen anzugeben sind. Ferner werden in dieser Vorlage einige spezielle Eigenschaften von GNU-make ausgenutzt. So expandiert etwa *$(wildcard *.c)*zu allen in „.c“ endenden Dateien im aktuellen Verzeichnis und *$(patsubst %.c,%.o,$(Sources*)) bildet die in *$(Sources)*enthaltenen C-Dateinamen in die entsprechenden Dateinamen mit der Endung „.o“ ab. Auf diese Weise sind nicht viele Änderungen an so einer Vorlage notwendig und damit wird es unwahrscheinlicher, dass sich auf diesem Wege Fehler einschleichen. So ließe sich das Beispiel dann übersetzen:

```
doolin$ ls
ggt.c  ggt.h  ggtmain.c  Makefile
doolin$ make depend
makedepend -- -Wall -std=c99 -- ggt.c ggtmain.c
doolin$ make
gcc -Wall -std=c99   -c -o ggt.o ggt.c
gcc -Wall -std=c99   -c -o ggtmain.o ggtmain.c
gcc -o ggt ggt.o ggtmain.o
doolin$ ggt 152 222
2
doolin$
```

12.5 Private Funktionen und Variablen

Voreinstellungsgemäß sind alle globalen Funktionen und Variablen öffentlich und daher auch von anderen Übersetzungseinheiten her frei nutzbar. Dies stellt einen deutlichen Unterschied zu Java, Oberon oder Modula-2 dar, bei denen alles privat ist, was nicht explizit als öffentlich sichtbar markiert wurde.

In C gibt es zwei Techniken zum Einschränken der Sichtbarkeit, die mit dem Schlüsselwort **static** verbunden sind. Dieses Schlüsselwort kann bei Deklarationen und Definitionen als sogenannte Speicherklasse mit angegeben werden. Allerdings hängt die genaue Semantik davon ab, ob die jeweilige Deklaration außerhalb eines Blocks oder lokal innerhalb einer Funktion erfolgt.

12.5.1 Lokale static-Variablen

Lokale Variablen entstehen normalerweise bei einem Aufruf der sie umgebenden Funktion und enden ihre Lebenszeit, wenn der Funktionsaufruf beendet ist. Solche Variablen werden in C-Terminologie der Speicherklasse **auto** zugeordnet.

Wird jedoch einer lokalen Variable das Schlüsselwort **static** vorangestellt, dann wird für diese Variable nur einmalig Speicher belegt, der unabhängig von den einzelnen Aufrufen existiert. D.h. alle Funktionsaufrufe referenzieren über diese Variable die gleiche Speicherfläche, die über die gesamte Lebensdauer des Prozesses zur Verfügung steht.

Das folgende Programm veranschaulicht die Verwendung der Speicherklasse **static** für lokale Variablen:

Programm 12.10: Speicherklasse static für lokale Variablen (*static.c*)

```
1  #include <stdio.h>
2
3  void work() {
4      static int counter = 0;
5      printf("␣%d", counter++);
6  }
7
8  int main() {
9      for (int i = 0; i < 10; i++) {
10         work();
11     }
12     puts("");
13 }
```

```
clonard$ gcc -Wall -std=c99 static.c
clonard$ a.out
 0 1 2 3 4 5 6 7 8 9
clonard$
```

12.5.2 Private nicht-lokale Variablen und Funktionen

Nicht-lokalen Variablen und Funktionen kann ebenfalls das Schlüsselwort **static** vorangestellt werden. In diesem Falle beschränkt sich die Sichtbarkeit dieser Variablen und Funktionen auf die umgebende Übersetzungseinheit. Die Namen dieser Variablen und Funktionen können dann auch nicht mehr in Konflikt stehen zu gleichnamigen Definitionen aus anderen Übersetzungseinheiten.

Entsprechend können solche privaten Variablen und Funktionen auch dann nicht aus anderswo verwendet werden, wenn entsprechende Deklarationen gegeben sind. Dies wird in folgendem Beispiel deutlich:

```
thales$ cat lib1.c
#include <stdio.h>
static float c = 4.5;
static void print() {
    printf("print: c = %f\n", c);
}
thales$ cat test4.c
#include <stdio.h>
extern float c;
extern void print();
int main() {
    printf("c = %f\n", c);
    print();
    return 0;
}
thales$ gcc -Wall test4.c lib1.c
lib1.c:3: warning: `print' defined but not used
/tmp/ccUWIDw6.o: In function `main':
/tmp/ccUWIDw6.o(.text+0x4): undefined reference to `c'
/tmp/ccUWIDw6.o(.text+0x8): undefined reference to `c'
/tmp/ccUWIDw6.o(.text+0x34): undefined reference to `print'
collect2: ld returned 1 exit status
thales$
```

Erklärung: Aufgrund der Definition mit dem Schlüsselwort **static** können *c* und *print()* nicht in *test4.c* verwendet werden – auch nicht nach der Deklaration. Hierbei kommt es zwar zu keinem Fehler bei der Übersetzung, aber beim Binden tritt ein Fehler auf, da die deklarierte Variable bzw. Funktion nicht gefunden wird.

Kapitel 13

Die C-Standards

13.1 Geschichtliche Entwicklung

In der Zeit *zwischen 1969 und 1973* wurde eine erste Version von *C* entwickelt. Der Name C resultierte aus der Tatsache, dass viele Elemente ihren Ursprung in der Sprache *B* hatten. *1973* wurde dann ein Großteil des *UNIX-Kernels* neu in C implementiert – C war inzwischen mächtig genug. *1978* publizierten Dennis Ritchie und Brian Kernighan dann die erste Ausgabe ihres Buches mit dem Titel *„The C Programming Language“*. Dieses Buch war über Jahre hinweg ein de-facto-Standard für das sogenannte *Kernighan&Ritchie C* (*K&R C*). (Die zweite Ausgabe des Buches richtet sich schon nach dem ANSI-Standard!) *1989* wurde C schließlich offiziell standardisiert im *Standard ANSI X3.159-1989* („Programming Language C“). *ANSI C* (oder auch *C89*) enthält eine ganze Reihe zusätzlicher Eigenschaften gegenüber K&R C. Der ANSI-Standard wurde dann *1990* von der ISO als Standard übernommen (*ISO 9899:1990*). Somit wurde es auch oft als *C90* bezeichnet. *1995* kam es zu einigen Änderungen am C-Standard. Von Bedeutung ist jedoch die Revision des C-Standards im Jahre *1999* (*ISO 9899:1999*). Dieser Standard wird oft auch einfach mit *C99* bezeichnet. *2000* wurde dieser Standard schließlich als ANSI-Standard übernommen.

2011 wurde der Standard *ISO/IEC 9899:2011* veröffentlicht (C11), der nach der Einarbeitung einiger Korrekturen durch *ISO/IEC 9899:2018* abgelöst wurde (C18).

13.2 Der Übergang von ANSI C / C90 zu C99

Der gcc unterstützt in der aktuellen Fassung (bei uns sind die Versionen 4.9.2, 5.4.0 und 7.3.0 installiert) alle Erweiterungen von *C99* und *C11* weitgehend. (Unter `https://gcc.gnu.org/c99status.html` ist die historische Entwicklung der C99-Konformität des gcc dokumentiert.) Sehr lange (bis einschließlich der 4er-Serie) war beim gcc der C90-Standard die Voreinstellung, aber ab der 5er-Serie des gcc wird per Voreinstellung von C11 ausgegangen.

Wenn noch ein älterer gcc-Übersetzer verwendet wird, sollte gcc mit *-std=gnu99* aufgerufen werden. Sonst fehlen zahlreiche mittlerweile selbstverständliche gewordene Möglichkeiten von C. Im Folgenden sind einige der wichtigsten Änderungen zwischen C90 und C99 zusammengestellt:

13.2.1 Einzeilige Kommentare

Mit „//“ ist es (wie auch in C++) möglich, *einzeilige Kommentare* einzuleiten, d. h. der Kommentar geht von „//“ *bis zum Zeilenende*.

Programm 13.1: Einzeilige Kommentare (*comment.c*)

```
1  #include <stdio.h>
2
3  int main() { // Hauptprogramm
4      puts("Hallo!"); // Ausgabe
5      return 0;
6  }
```

13.2.2 Mischen von Deklarationen/Definitionen und Anweisungen

Bisher durften Deklarationen und Definitionen nur am Anfang eines Anweisungsblocks (*compound statement*) stehen. Diese strikte Trennung ist nun aufgehoben. Jetzt können Anweisungen und Deklarationen bzw. Definitionen in beliebiger Reihenfolge stehen.

Programm 13.2: Mischen von Deklarationen/Definitionen und Anweisungen (*mixed.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      int x = 17;
5      printf("x␣=␣%d\n", x);
6
7      int y = x * x - 2;
8      printf("y␣=␣%d\n", y);
9
10     return 0;
11 }
```

13.2.3 Variablen in for-Schleifen

Im Initialisierungs-Teil einer *for-Schleife* können nun auch Variablen definiert werden. Der Gültigkeitsbereich solcher Variablen erstreckt sich dann auf alle Teile der Schleife (Initialisierung, Vergleich, Inkrement und Schleifenkörper).

Programm 13.3: Variablen in for-Schleifen (*for.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      for (int i = 0; i < 10; i++)
5          printf("i␣=␣%d\n", i);
6      return 0;
7  }
```

13.2.4 Arrays variabler Länge

Bisher musste die Länge bei der Definition bzw. Deklaration eines Arrays immer eine Konstante sein. Diese Restriktion ist nun entfallen. Dies betrifft sowohl lokale Variablen als auch Parameter. Besonders interessant ist dies im Hinblick auf die Parameterübergabe mehrdimensionaler Arrays, die nun sehr vereinfacht ist.

Da diese Art der Parameterübergabe bei Arrays nicht für C++ übernommen worden ist und es unwahrscheinlich ist, dass dies je geschehen wird, ist davon eher abzuraten, um eine Kompatibilität zwischen C und C++ nicht zu gefährden.

Programm 13.4: Arrays variabler Länge (*varrays.c*)

```
1  #include <stdio.h>
2
3      // BEACHTE: Die Parameter s1 und s2 muessen
4      // vor int m[s1][s2] stehen, wo sie verwendet werden!
5  void print(int s1, int s2, int m[s1][s2]) {
6      for (int i = 0; i < s1; i++) {
7          for (int j = 0; j < s2; j++)
8              printf("%4d ", m[i][j]); // komfortabler Element-Zugriff ;-)
9          puts("");
10     }
11 }
12
13 void printsum(int len, int a[len], int b[len]) {
14     int c[len]; // (eigentlich ueberfluessiges) lokales Array
15     for (int i = 0; i < len; i++)
16         c[i] = a[i] + b[i];
17     printf("(");
18     for (int i = 0; i < len; i++)
19         printf(i > 0 ? ", %d" : "%d", c[i]);
20     printf(")\n");
21 }
22
23 int main() {
24     int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
25     print(2, 3, matrix);
26     printsum(3, matrix[0], matrix[1]);
27     return 0;
28 }
```

```
thales$ gcc -Wall -std=c99 varrays.c
thales$ a.out
   1    2    3
   4    5    6
(5, 7, 9)
thales$
```

13.2.5 Flexibles Array-Element in Strukturen

Es ist jetzt möglich, dass das *letzte Element einer Struktur* einen *unvollständigen Array-Typ* bestitzt, d. h., dass die *Länge* des Arrays *nicht spezifiziert* ist. In diesem Fall trägt dieses Element nichts zum Speicherplatzverbrauch der Struktur bei. Es kann jedoch bei einer *dynamisch allokierten Struktur* verwendet werden, um auf ein variables Array zuzugreifen.

Programm 13.5: Flexibles Array-Element in Strukturen (*struct.c*)

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

struct my {
    int len;
    int a[]; // flexibles Array-Element (am Ende!)
};

int main() {
    printf("sizeof(struct my) = %d\n", sizeof(struct my));
    printf("sizeof(int)       = %d\n", sizeof(int));

    int len = 3;
    struct my *s;

        // simple Rechnung bei der Speicherallokation
    s = calloc(1, sizeof(struct my) + len * sizeof(int));
    s->len = len;

    for (int i = 0; i < s->len; i++)
        printf("s->a[%d] = %d\n", i, s->a[i]);

    return 0;
}
```

```
thales$ gcc -Wall -std=c99 struct.c
thales$ a.out
sizeof(struct my) = 4
sizeof(int)       = 4
s->a[0] = 0
s->a[1] = 0
s->a[2] = 0
thales$
```

13.2.6 Nicht-konstante Initialisierer

In der *Initialisierung eines aggregierten Typs* (Array, Struktur) der Speicherklasse **auto** dürfen nun auch Variablen verwendet werden.

Programm 13.6: Nicht-konstante Initialisierer (*init.c*)

```c
#include <stdio.h>

void work(int a, int b) {
    int c[2] = {a + b, a - b};
    printf("c[0] = %d\n", c[0]);
    printf("c[1] = %d\n", c[1]);
}

int main() {
    work(1, 3);
    return 0;
```

```
12 }
```

13.2.7 Namentliche Element-Initialisierer

Ein Initialisierer (für Arrays und Strukturen) in ANSI C (C89) bzw. C90 musste die Elemente in der selben Reihenfolge enthalten wie sie bei der Definition aufgeführt sind. In C90 kann nun durch Voranstellen von *[Index]=* bei Array- und *.Element=* bei Struktur-Initialisierern diese Restriktion entfallen. Es können auch Elemente oder Indizes bei der Initialisierung fehlen.

13.2.7.1 Arrays

int *a*[6] = {[3] = 1, [1] = −1, [4] = 1};

ist äquivalent zu

int *a*[6] = {[1] = −1, [3 ... 4] = 1};

und äquivalent zu

int *a*[6] = {0, −1, 0, 1, 1, 0};

Programm 13.7: Initialisierer für Arrays (*init1.c*)

```
1  #include <stdio.h>
2
3  #define LEN 5
4
5  int main() {
6      int a[LEN] = {[1 ... 3] = −1, [2] = 2};
7      for (int i = 0; i < LEN; i++)
8          printf("a[%d] = %d\n", i, a[i]);
9      return 0;
10 }
```

```
thales$ gcc -Wall -std=c99 init1.c
thales$ a.out
a[0] = 0
a[1] = -1
a[2] = 2
a[3] = -1
a[4] = 0
thales$
```

Anmerkung: Wird ein *Element mehr als einmal initialisiert*, so hat die *letzte Initialisierung* dieses Elements Erfolg.

13.2.7.2 Strukturen

Gegeben ist die folgende Typ-Definition:

struct *point* { **int** *x*, *y*; };

Dann kann man eine Variable von diesem Typ wie folgt initialisieren:

struct *point p* = {.*y* = 3, .*x* = 2};

In ANSI C hätte das wie folgt aussehen müssen:

struct *point p* = {2, 3};

13.2.8 Bereiche bei switch-Anweisungen

Wollte man bisher in einer switch-Anweisung einen Fall für alle Werte im Bereich beispielsweise von 3 bis 10 haben, so musste man dies sehr kompliziert notieren:

case 3:
case 4:
/* ... */
case 10:

Dies lässt sich nun ganz einfach wie folgt erledigen:

case 3 ... 10:

Hinweis: Man beachte, dass die *Leerzeichen* um „…“ für den Parser sehr wichtig sind. Andernfalls besteht die Gefahr, dass dies bei Integern falsch interpretiert wird!

Programm 13.8: Bereiche bei der switch-Anweisung (*switch.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      char c = getchar();
5      switch (c) {
6          case 'a' ... 'z' : puts("Kleinbuchstabe"); break;
7          case 'A' ... 'Z' : puts("Grossbuchstabe"); break;
8          case '0' ... '9' : puts("Ziffer"); break;
9          default : puts("anderes␣Zeichen"); break;
10     }
11     return 0;
12 }
```

13.2.9 Boolesche Variablen

Für Boolesche Variablen gibt es jetzt den Typnamen *_Bool*. In der Header-Datei *stdbool.h* sind passend dazu die beiden Makros *true* und *false* definiert. Beim Typnamen _Bool handelt es sich jedoch auch nur um einen Integer-Typ, wie folgendes Programm zeigt:

Programm 13.9: Boolesche Variablen (*bool.c*)

```
1  #include <stdio.h>
2  #include <stdbool.h>
3
4  int main() {
5      _Bool c = true;
6      printf("c␣=␣%s\n", c ? "true" : "false");
7      printf("c␣=␣%d\n", c);
8      c = 1;
9      printf("c␣=␣%s\n", c ? "true" : "false");
10     printf("c␣=␣%d\n", c);
```

```
11      return 0;
12  }
```

13.2.10 Große Integer

Neu gibt es auch den Typ *long long int*, der Integer der Größe von mindestens 64 Bit beherbergen kann. Bei *printf()* gibt es die neue Konvertierungsangabe *%lli* mit der Integer von diesem Typ interpoliert werden können.

Programm 13.10: Große Integer (*longlong.c*)

```
1  #include <stdio.h>
2
3  int main() {
4      long long int n = 123123123123;
5      printf("n = %lli\n", n);
6      return 0;
7  }
```

```
thales$ gcc -Wall -std=c99 longlong.c
thales$ a.out
n = 123123123123
thales$
```

13.2.11 Funktion snprintf()

Analog zur Funktion *fprintf()* mit der man auf einen Stream (z. B. stderr, stdout) formatiert schreiben kann, gibt es die Funktion *sprintf()*, die als ersten Parameter ein Zeichen-Array (String) erwartet und in dieses formatiert schreibt. Dabei setzt man sich aber derselben Gefahr wie bei der Verwendung von *gets()* aus. Beim Schreiben wird nicht drauf geachtet, ob der vorhandene Platz ausreicht. (Diese Information ist diesen beiden Funktionen auch nicht bekannt!) Aus diesem Grund entstand das sicherere *fgets()*, bei dem auch die Größe des Puffers angegeben werden kann. Diese Entwicklung hat sich nun auch bei *sprintf()* vollzogen. Herausgekommen ist die Funktion *snprintf()*, die als Parameter den Puffer, die Größe des Puffers, den Format-String und die Format-Argumente erwartet und höchstens ein Zeichen weniger als die Größe des Puffers Zeichen „ausgibt“, da der Puffer ja noch mit dem Null-Byte terminiert werden muss.

Programm 13.11: Funktion snprintf() (*snprintf.c*)

```
1  #include <stdio.h>
2
3  #define LEN 10
4
5  int main() {
6      char *label = "Bezeichnung";
7      int x = 387;
8      char s[LEN];
9
10     snprintf(s, LEN, "%s: %d", label, x);
```

```
11      puts(s);
12
13      return 0;
14  }
```

```
thales$ gcc -Wall -std=c99 -D'__EXTENSIONS__=' snprintf.c
thales$ a.out
Bezeichnu
thales$
```

13.2.12 Variable Anzahl von Argumenten bei Makros

Makros können nun auch eine *variable Anzahl von Argumenten* haben. Dies drückt man bei den *formalen Argumenten* durch „…“ aus. Im Ersatztext kann man dann auf die Argumente, welche für „…“ eingesetzt wurden, mit dem Bezeichner *__VA_ARGS__* Bezug nehmen.

Programm 13.12: Variable Anzahl von Argumenten bei Makros (*makros.c*)

```
1   #include <stdio.h>
2
3   #define DEBUG(...) fprintf(stderr, __VA_ARGS__);
4   #define QUOTE(...) puts(#__VA_ARGS__);
5   #define TEST(cond, ...) if (!(cond)) \
6                                   DEBUG(__VA_ARGS__)
7
8   int main() {
9       int x = 3;
10      DEBUG("x␣=␣%d\n", x)
11      QUOTE(x < 2, x > 0, x)
12      TEST(x == 0, "failed:␣x␣=␣%d\n", x)
13      return 0;
14  }
```

```
thales$ gcc -E -std=c99 makros.c
/* gekuerzte Ausgabe ;-) */
int main() {
    int x = 3;
    fprintf((&__iob[2]), "x = %d\n", x);
    puts("x < 2, x > 0, x");
    if (!(x == 0)) fprintf((&__iob[2]), "failed: x = %d\n", x);
    return 0;
}
thales$ gcc -Wall -std=c99 makros.c
thales$ a.out
x = 3
x < 2, x > 0, x
failed: x = 3
thales$
```

13.2.13 Name der aktuellen Funktion

Analog zu den Makros *__FILE__* und *__LINE__* gibt es jetzt (vom Compiler) einen vordefinierten Bezeichner *__func__*, bei dem es sich um einen String (also Zeichen-Array) handelt, der den Name der aktuellen Funktion enthält.

Programm 13.13: assert mit Name der aktuellen Funktion (*assert.c*)

```
1  #include <stdio.h>
2
3  #define ASSERT(cond,...) \
4                  if (!(cond)) { \
5                      fprintf(stderr, "%s (%d): %s(): assertion \"%s\" failed\n", \
6                          __FILE__, __LINE__, __func__, #cond); \
7                      __VA_ARGS__ \
8                  }
9
10 void work() {
11     ASSERT(1 == 3)
12 }
13
14 int main() {
15     int a = 3;
16     ASSERT(a < 0, fprintf(stderr, "DEBUG: a = %d\n", a);)
17     work();
18     return 0;
19 }
```

```
thales$ gcc -Wall -std=c99 assert.c
thales$ a.out
assert.c (16): main(): assertion "a < 0" failed
DEBUG: a = 3
assert.c (11): work(): assertion "1 == 3" failed
thales$
```

13.2.14 Inline-Funktionen

Funktionen können jetzt auch *„inline“* deklariert werden. Dies bedeutet, dass bei der Definition der Funktion das Schlüsselwort **inline** vor den Typ des Rückgabewertes eingefügt wird und hat zur Folge, dass der Code der *Funktionsimplementierung an der Stelle des Funktionsaufrufes eingesetzt* wird – ähnlich einem Makro. Dabei gibt es aber *keine Probleme mit Seiteneffekten* in den Argumenten – wie dies bei Makros der Fall ist.

Programm 13.14: Inline-Funktionen (*inline.c*)

```
1  #include <stdio.h>
2
3  int count = 0;
4
5  void work_slow() {
6      count++;
7  }
```

```
 8
 9  inline void work_fast() {
10      count++;
11  }
12
13  int main() {
14      for (int i = 0; i < 100000000; i++)
15  #ifdef FAST
16          work_fast();
17  #else
18          work_slow();
19  #endif
20      return 0;
21  }
```

```
thales$ gcc -Wall -std=c99 -O inline.c
thales$ time a.out

real    0m4.836s
user    0m4.180s
sys     0m0.010s
thales$ gcc -Wall -std=c99 -O -D'FAST=' inline.c
thales$ time a.out

real    0m0.820s
user    0m0.750s
sys     0m0.010s
thales$
```

Anmerkung: Damit vom gcc bei inline-Funktionen tatsächlich der Code der Funktionsimplementierung an die Aufrufstellen „kopiert“ wird, muss die Option *-O* angegeben werden, die dem Übersetzer das Optimieren erlaubt.

13.3 Der Übergang von C99 zu C11/C18

Zu den wichtigsten Neuerungen gehören Alignments und Threads.

13.3.1 Alignment

Mit dem neuen Schlüsselwort **_Alignas** ist es möglich, bei der Belegung von Speicher für eine Variable ein Alignment zu verlangen, das über die normale Mindestanforderung herausgeht (*over alignment*). Dies kann z.B. sinnvoll sein, um eine Datenstruktur genau auf eine Cache-Line-Kante auszurichten. Beispiel:

```
#include <stdalign.h>

struct data {
    alignas(64) char cache_line[64];
};
```

Notwendig war hier die Verwendung von **#include** <stdalign.h>, um die übliche Schreibweise **alignas** zuzulassen, die als Makro mit dem Textersatz **_Alignas** definiert ist. Solche Konstrukte sind bei den neueren Standards von C üblich, um Konflikte mit älteren Programmen zu vermeiden.

Darüberhinaus gibt es auch seit C99 **_Alignof** bzw. (wiederum aus **#include** <stdalign.h>) **alignof**, mit dem das Alignment einer Variable oder eines Typs ermittelt werden kann. Dies ist insbesondere in Verbindung mit der Speicherbelegungsfunktion *aligned_alloc* sinnvoll.

13.3.2 Threads

Beginnend mit C11 unterstützt der Standard die POSIX-Threads-Schnittstelle. Variablen können mit **_Thread_local** lokal zum eigenen Thread deklariert werden (bzw. über **#include** <threads.h>) mit **thread_local**. Über **#include** <stdatomic.h> stehen atomare Methoden für den konkurrierenden Speicherzugriff zur Verfügung.

Anhang

Literatur

M. J. Bach: *The Design of the UNIX Operating System.* Prentice Hall, 1986.

D. Comer: *Operating System Design: The XINU Approach.* Prentice Hall, 1984.

P. A. Darnell und P. E. Margolis: *C: A Software Engineering Approach.* Springer, Dritte Auflage, 2001.

D. Goldberg: *What every computer scientist should know about floating-point arithmetic.* ACM Computing Surveys, Jahrgang 23, Heft 1 vom März 1991, Seiten 5-48.

T. Handschuch: *Solaris 2 f"ur den Systemadministrator.* Solaris Galerie, IWT-Verlag, 1993.

S. P. Harbison, G. L. Steele: *C: A Reference Manual.* Fünfte Auflage, Prentice Hall, 2002.

H. Herold: *Linux-Unix-Shells.* Addison-Wesley, 1999.

B. W. Kernighan und R. Pike: *Der UNIX-Werkzeugkasten.* Hanser, 1986.

B. W. Kernighan und D. Ritchie: *Programmieren in C.* Hanser, Zweite Auflage, 1990.

D. E. Knuth: *The Art of Computer Programming*, *Volume 2*, *Seminumerical Algorithms*, Addison-Wesley, Dritte Auflage, 1997.

A. Koenig: *C Traps and Pitfalls.* Prentice Hall, 1989.

M. Rochkind: *UNIX-Programmierung f"ur Fortgeschrittene.* Hanser Verlag, 1988.

W. R. Stevens: *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992.

B. Stroustrup: *The Design and Evolution of C++.* Addison-Wesley, 1994.

Sun Microsystems: *Code Conventions for the Java Programming Language.* Zu finden im Web unter http://java.sun.com/docs/codeconv/.

A. S. Tanenbaum: *Operating Systems - Design and Implementation.* Prentice Hall, 1987.
$\rightsquigarrow$ **Bitte auf die jeweils aktuellste Auflage achten!**

Abbildungsverzeichnis

Beispiel-Programme

Index