ulm university universität uulm

Vorlesungsbegleiter zu Systemnahe Software I WS 2011/2012

Andreas F. Borchert
Matthias Grabert
Johannes Mayer
Franz Schweiggert

Fakultät für Mathematik und Wirtschaftswissenschaften
Institut für Angewandte Informationsverarbeitung

Hinweise:

- Auf eine detaillierte Unterscheidung zwischen *BSD-Unix*, *System-V-Unix* oder *Linux* wird hier verzichtet. Stattdessen dient der IEEE Standard 1003.1 (POSIX) weitgehend als Grundlage.
- Die enthaltenen Beispiel-Programme wurden zum großen Teil unter Linux entwickelt und sind weitestgehend unter Solaris getestet (für konstruktive Hinweise sind die Autoren dankbar).
- Die Beispiele sollen jeweils gewisse Aspekte verdeutlichen und erheben nicht den Anspruch von Robustheit und Zuverlässigkeit. Man kann alles anders und besser machen.
- Details zu den behandelten bzw. verwendeten Systemaufrufen sollten jeweils im **Manual** bzw. den entsprechenden Header-Files nachgelesen werden.
- Die Sprache C dient in erster Linie als *Werkzeug* zur Darstellung systemnaher Konzepte!

Dennis M. Ritchie, 1941–2011
Ausschnitt eines Fotos von Denise Panyik-Dale, CC-BY-2.0

Dieses Skript ist Dennis M. Ritchie gewidmet, ohne dessen Beiträge es nicht denkbar wäre. Dennis M. Ritchie hat nicht nur die Programmiersprache C entworfen und implementiert, sondern auch zusammen mit Ken Thompson Unix entwickelt. Mit C wurde erstmals eine höhere Programmiersprache statt Assembler für die Systemprogrammierung verwendet. Dennis Ritchie und seinen Kollegen gelang es bei Unix, die Betriebsystemsschnittstellen und die Systemwerkzeuge in einem bis dahin unbekanntem Maß zu vereinfachen. Auf sie geht der Grundsatz zurück, dass jedes Werkzeug genau eine Aufgabe zu erfüllen habe und das möglichst gut.

Inhaltsverzeichnis

Kapitel 1

Entstehungsgeschichte

Die Abb. 1.1 zeigt eine vereinfachte Darstellung der Entwicklungsbeziehungen einiger Programmiersprachen.

Die Programmiersprache C

- C wurde 1972-73 von Dennis Ritchie bei den Bell Laboratories von AT&T entwickelt.
- Vorbilder waren Algol, Fortran und BCPL.
- Zu den Zielsetzungen gehörte es, eine recht einfache portable maschinennahe Sprache zu entwickeln, die ohne aufwändige Laufzeitunterstützung leicht in effizienten Maschinen-Code übersetzt werden kann. Damit gelang es, UNIX weitgehend in C zu schreiben, was die spätere Portabilität von UNIX sehr erleichtert hat.
- Ähnlich wie Assembler bot C damals nur sehr wenig Überprüfungen an (praktisch keine Typüberprüfungen, keine Kontrolle von Array-Indizes oder Zeigern) und wenig Komfort – so liessen sich nur elementare Basistypen einander zuweisen.
- Viele maschinennahe Elemente aus Assembler wurden in C übernommen wie beispielsweise der Umgang mit Arrays, die als Speicherflächen betrachtet werden, auf die Zugriffe mit Hilfe von Zeigerarithmetik erfolgen.
- 1978 erfolgten einige Erweiterungen von C (*enum*, *void*, *structure assignment*, ...), die mit in das erste Buch über C von Kernighan und Ritchie integriert wurden und damit den sogenannten *K&R-Standard* begründeten.
- 1983 begannen Standardisierungsbemühungen für C, die 1989 zum ANSI-Standard X3.159-1989 führten, der auch kurz ANSI-C oder C89 genannt wird. Die wichtigste Änderung war die Einführung von Funktionsprototypen, die es bei C nun erlaubten Funktionsaufrufe gegen die Deklaration einer Funktion zu überprüfen. Ein Jahr später wurde dieser Standard mit nur minimalen Veränderungen auch von ISO (als Standard 9899:1990) übernommen.
- Weitere Erweiterungen und Überprüfungsmöglichkeiten flossen in den 1999 veröffentlichten Standard ISO 9899:1999. Diese Version wird kurz C99 genannt und bildet die Grundlage für diese Vorlesung.
- In der Vorlesung und in den Übungen wird primär mit dem *GNU-C(++)-Compiler* gearbeitet, der auch für Windows erhältlich ist. Hinweise dazu gibt es auf den Webseiten zur Vorlesung.

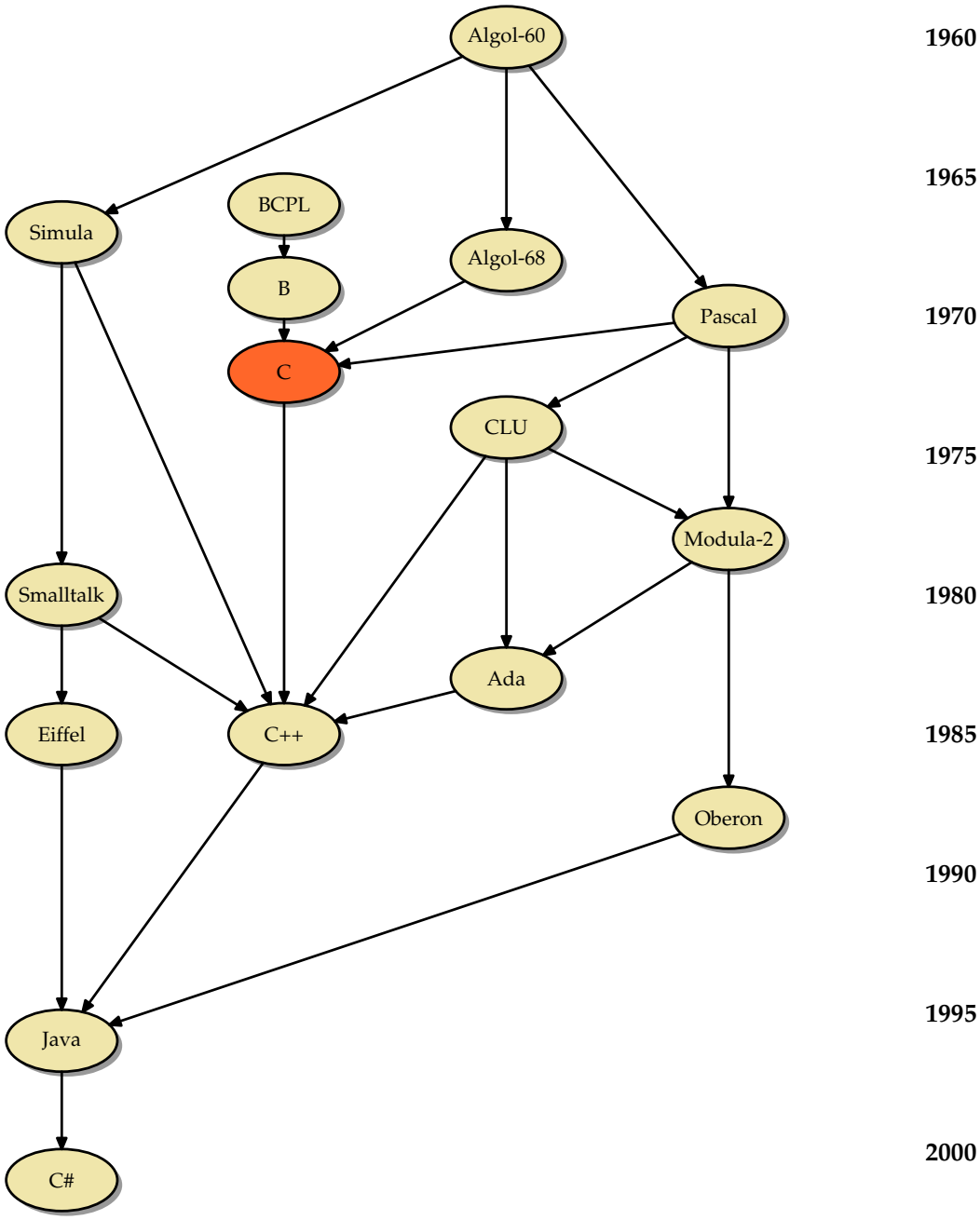


Abbildung 1.1: Entwicklungsbeziehungen einiger Programmiersprachen

- *Literatur:* Jedes Buch, das sich mit C direkt beschäftigt wie beispielsweise [Kernighan 1990] oder [Harbison 2002]. Bücher über C++ sind aufgrund der Komplexität der objektorientierten Konzepte für die Vorlesung nicht empfehlenswert.

Andere Programmiersprachen

- C und grammatikalische Elemente aus der Programmiersprache C wurden prägend für zahlreiche weitere Programmiersprachen.
- Sehr nahe an C im Sinne der Aufwärtskompatibilität blieben die objektorientierten Erweiterungen *C++* und *Objective C*.
- Andere Sprachen haben nur Teile der Syntax übernommen (wie beispielsweise *Java*, *C#* oder *Perl*), ohne kompatibel zu sein oder die maschinennahe Denkweise zu übernehmen. So hat insbesondere Java inhaltlich nicht wenige Techniken von Eiffel oder BETA übernommen, obwohl diese beiden Sprachen syntaktisch völlig anders aussehen.

Kapitel 2

Erste Schritte mit C

Bevor wir uns mit dem Aufbau und der Syntax eines C-Programms beschäftigen, folgen zunächst einige Beispiele, um einen ersten Eindruck der Sprache C zu gewinnen.

2.1 Einige Beispiel-Programme

2.1.1 Unser erstes C-Programm

Folgendes Programm ist ein minimales „Hello World"-Beispiel in C:

Programm 2.1: Hello World – Erste Version (*hallo.c*)

```
main() {
    /* puts: Ausgabe einer Zeichenkette nach stdout */
    puts("Hallo␣zusammen!");
}
```

Übersetzung und Ausführung:

```
doolin$ gcc -Wall hallo.c
hallo.c:1: warning: return type defaults to 'int'
hallo.c: In function 'main':
hallo.c:3: warning: implicit declaration of function 'puts'
hallo.c:4: warning: control reaches end of non-void function
doolin$ a.out
Hallo zusammen!
doolin$
```

Der *gcc* ist der *GNU-C-Compiler*, mit dem wir unsere Programme übersetzen. Ist kein Name für das zu generierende ausführbare Programm angegeben, so wird dieses *a.out* genannt. Die Option *-Wall* bedeutet, dass alle Warnungen ausgegeben werden sollen.

Voreinstellungsgemäß geht *gcc* von *C89* aus. Es ist auch möglich, den aktuellen Standard *C99* zu verwenden, wenn dies in einer entsprechenden Option verlangt wird:

```
doolin$ gcc -Wall -std=c99 hallo.c
hallo.c:1: warning: return type defaults to 'int'
hallo.c: In function 'main':
hallo.c:3: warning: implicit declaration of function 'puts'
doolin$
```

Interessanterweise führt das hier dazu, dass die Warnung über das fehlende **return** weggefallen ist, da dies für *main()* in *C99* nicht mehr vorgeschrieben ist.

Im Vergleich zu *Java* fällt auf, dass bei *main()* die Angabe der Kommandozeilenparameter fehlt. In C ist es zulässig, dies wegzulassen, wenn diese nicht benötigt werden. Ferner fehlt die Angabe einer Klasse, eines Pakets oder eines Moduls, da all dies in C nicht existiert. Stattdessen bestehen C-Quellen im wesentlichen nur aus einer Ansammlung von Variablen- und Funktionsvereinbarungen.

2.1.2 Eine bessere Welt

An den verbliebenen Warnungen bei der Ausführung von Programm 2.1 ist zu erkennen, dass das erste Beispiel nicht ganz vollständig war. Folgendes Beispiel ist nun eine erweiterte und verbesserte Version:

Programm 2.2: Hello World – Verbesserte Version (*hallo1.c*)

```
#include <stdio.h> /* Standard−I/O−Bibliothek einbinden */

int main() {
    /* puts: Ausgabe eines Strings nach stdout */
    puts("Hallo␣zusammen!");
    /* Programm explizit mit Exit−Status 0 beenden */
    return 0;
}
```

Folgende Änderungen sind (gegenüber Programm 2.1) erfolgt:

- Da die Ausgabefunktion *puts()* nicht bekannt war, hat der Übersetzer geraten. Nun ist diese Funktion durch das Einbinden der Standard-I/O-Bibliothek (siehe **#include** <stdio.h>) bekannt.
- Der *Typ des Rückgabewertes* der *main()*-Funktion ist nun als **int** (Integer) angegeben (der Übersetzer hat vorher auch **int** geraten.)
- Der Rückgabewert der *main()*-Funktion, welcher durch **return** 0 gesetzt wird, ist der *Exit-Status* des Programms. Fehlt dieser, führt dies ab *C99* implizit zu einem ein Exit-Status von 0.

  Dieser wird von der *Shell* unmittelbar nach der Ausführung des Programms in der Variablen *$?* (genauer: die Variable hat den Bezeichner *?*) bereit gestellt und kann durch das Kommando **echo** *$?* angezeigt werden (das Dollarzeichen vor dem Variablennamen veranlasst die Shell, den Wert dieser Variablen zu substituieren).

  Eine normale (d.h. erfolgreiche) Beendigung wird durch den *Exit-Status* 0 signalisiert; alles andere steht für „nicht erfolgreich“ (oft: Fehler) bei der Ausführung.

Die Übersetzung und Ausführung von Programm 2.2 liefert nun:

```
doolin$ gcc -Wall -o hallo1 hallo1.c
doolin$ hallo1
Hallo zusammen!
doolin$
```

Mit der Option *-o* kann der Name des Endprodukts beim Aufruf des *gcc* spezifiziert werden.

2.1.3 Quadratisch, praktisch, gut

Programm 2.3 berechnet die ersten 20 Quadratzahlen und gibt sie auf die Standardausgabe aus:

Programm 2.3: Berechnung von Quadratzahlen mit einer for-Schleife (*quadrate.c*)

```
#include <stdio.h>

const int MAX = 20; /* globale Integer−Konstante */

int main() {
    int n; /* lokale Integer−Variable */

    puts("Zahl␣|␣Quadratzahl");
    puts("−−−−−+−−−−−−−−−−−−−");
    for (n = 1; n <= MAX; n++) {
        printf("%4d␣|␣%7d\n", n, n * n); /* formatierte Ausgabe */
    }
}
```

An obigem Programm-Beispiel lässt sich erkennen, wie *globale* Variablen und *lokale* Variablen vereinbart werden können. Die globale Variable wurde als *Konstante* definiert. Außerdem wird die Funktion *printf()* zur formatierten Ausgabe verwendet. Mit einer **for**-Schleife werden die ersten zwanzig natürlichen Zahlen durchlaufen.

Programm 2.4 ist mit einer **while**-Schleife implementiert, die äquivalent zur zuvor vorgestellten **for**-Schleife aus Programm 2.3 ist.

Programm 2.4: Berechnung von Quadratzahlen mit einer while-Schleife (*quadrate1.c*)

```
#include <stdio.h>

const int MAX = 20;

int main() {
    int n;

    puts("Zahl␣|␣Quadratzahl");
    puts("−−−−−+−−−−−−−−−−−−−");
    n = 1; /* wird vor dem ersten Durchlauf ausgefuehrt */
    while (n <= MAX) { /* Bedingung wird vor jedem Durchlauf getestet */
        printf("%4d␣|␣%7d\n", n, n * n);
        n = n + 1; /* wird nach jedem Durchlauf ausgefuehrt */
    }
}
```

2.1.4 Euklidischer Algorithmus

Programm 2.5 implementiert den bekannten Euklidischen Algorithmus zur Bestimmung des *größten gemeinsamen Teilers* zweier natürlicher Zahlen; hier wird die Funktion *scanf()* zum Einlesen der beiden Zahlen von der Standardeingabe benutzt.

Programm 2.5: Euklidischer Algorithmus (*euklid.c*)

```
#include <stdio.h>

int main() {
    int x, y, x0, y0;

    printf("Geben␣Sie␣zwei␣positive␣ganze␣Zahlen␣ein:");
    /* das Resultat von scanf ist die
       Anzahl der eingelesenen Zahlen
    */
    if (scanf("%d␣%d", &x, &y) != 2) { /* &−Operator konstruiert Zeiger */
        return 1; /* Exit−Status ungleich 0 => Fehler */
    }

    x0 = x; y0 = y;

    while (x != y) {
        if (x > y) {
            x = x − y;
        } else {
            y = y − x;
        }
    }

    printf("ggT(%d,␣%d)␣=␣%d\n", x0, y0, x);

    return 0;
}
```

Die Programmiersprache C kennt nur die *Werteparameter-Übergabe* (*call by value*). Daher stehen auch bei *scanf()* nicht direkt die Variablen *x* und *y* als Argumente. Mit dem Operator & wird hier jeweils ein *Zeiger* auf die folgende Variable „konstruiert“. Der Wert eines Zeigers ist die *Hauptspeicher-Adresse* der Variablen, auf die er zeigt (daher wird in diesem Zusammenhang der Operator & auch als *Adressoperator* bezeichnet).

Damit ist der Zeigerwert (= Adresse) zwar lokal zu *scanf()*, jedoch kann dadurch (innerhalb von *scanf()*) auf die lokalen Variablen in *main()* „durchgegriffen“ werden. Auf diese Weise kann *scanf()* die eingelesenen Zahlen in *x* und *y* ablegen. Dies mag hier genügen – später beschäftigen wird uns noch ausführlich mit Zeigern.

Programm 2.6 demonstriert die Erstellung und Verwendung von Funktionen in C.

Programm 2.6: Euklidischer Algorithmus als Funktion (*euklid1.c*)

```
#include <stdio.h>

int ggt(int x, int y) {
    while (y != 0) {
        int tmp = x % y; /* Divisionsrest == wiederholte Subtraktion */
        x = y; y = tmp;
```

```
    }
    return x;
}

int main() {
    int x, y;

    printf("Geben␣Sie␣zwei␣positive␣ganze␣Zahlen␣ein:");
    if (scanf("%d␣%d", &x, &y) != 2) /* &−Operator konstruiert Zeiger */
        return 1; /* Exit−Status ungleich 0 => Fehler */

    printf("ggT(%d,␣%d)␣=␣%d\n", x, y, ggt(x, y));

    return 0;
}
```

Die Berechnung des ggT wurde einfach vom Hauptprogramm in die (neu angelegte) Funktion *ggt()* „ausgelagert“. Aufgrund der Werteparameter-Semantik bei Funktionsaufrufen müssen wir die Eingaben x und y nicht mehr kopieren (wie im vorigen Beispiel).

2.2 Aufbau eines C-Programms

Eine Übersetzungseinheit (*translation unit*) in C ist eine Folge von *Vereinbarungen*, zu denen Funktionsdefinitionen, Typ-Vereinbarungen und Variablenvereinbarungen gehören:

⟨translation-unit⟩	⟶	⟨top-level-declaration⟩
	⟶	⟨translation-unit⟩ ⟨top-level-declaration⟩
⟨top-level-declaration⟩	⟶	⟨declaration⟩
	⟶	⟨function-definition⟩
⟨declaration⟩	⟶	⟨declaration-specifiers⟩
		⟨initialized-declarator-list⟩ „;“
⟨declaration-specifiers⟩	⟶	⟨storage-class-specifier⟩ [⟨declaration-specifiers⟩]
	⟶	⟨type-specifier⟩ [⟨declaration-specifiers⟩]
	⟶	⟨type-qualifier⟩ [⟨declaration-specifiers⟩]
	⟶	⟨function-specifier⟩ [⟨declaration-specifiers⟩]

Hinweis: Die hier und im Folgenden vorgestellten Auszüge der Grammatik wurden weitgehend [Harbison 2002] entnommen und entsprechen dem Stand von *C99*.

2.2.1 Anweisungsblöcke

Wie in Java unterstützt C eine *Blockstruktur* in Form eines *Anweisungsblocks* (*compound statement*). Variablen-Vereinbarungen dürfen an beliebiger Stelle eines *Anweisungsblocks* stehen und sind dann bis zum Ende des jeweiligen Blocks sichtbar. (Zu beachten ist, dass dies erst ab *C99* gilt, bei *C89* sind Vereinbarungen nur zu Beginn des Blocks zulässig.) Anweisungsblöcke erlauben es, mehrere Anweisungen zusammenzufassen und *Sichtbarkeits-/Lebensdauerbereiche* zu definieren; siehe Abb. 2.1 .

⟨statement⟩	⟶	⟨expression-statement⟩
	⟶	⟨labeled-statement⟩
	⟶	⟨compound-statement⟩
	⟶	⟨conditional-statement⟩
	⟶	⟨iterative-statement⟩
	⟶	⟨switch-statement⟩
	⟶	⟨break-statement⟩
	⟶	⟨continue-statement⟩
	⟶	⟨return-statement⟩
	⟶	⟨goto-statement⟩
	⟶	⟨null-statement⟩
⟨compound-statement⟩	⟶	„{“ [⟨declaration-or-statement-list⟩] „}“
⟨declaration-or-statement-list⟩	⟶	⟨declaration-or-statement⟩
	⟶	⟨declaration-or-statement-list⟩
		⟨declaration-or-statement⟩
⟨declaration-or-statement⟩	⟶	⟨declaration⟩
	⟶	⟨statement⟩

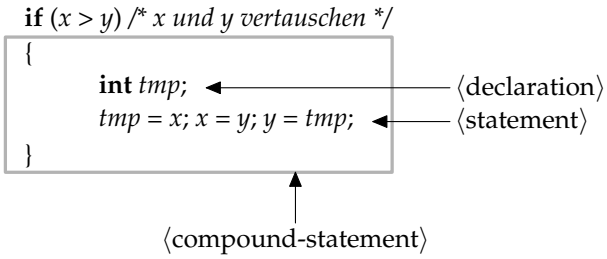


Abbildung 2.1: Anweisungsblock

Anmerkungen zu Abb. 2.1 :

- Die *Gültigkeit* von *tmp* erstreckt sich auf den umrandeten Anweisungsblock.
- Mit **int** *tmp*; wird eine lokale Variable mit dem Datentyp **int** deklariert. **int** ist ein Schlüsselwort und steht für *integer*, also eine ganze Zahl.
- Für *Zuweisungen* wird in C analog zu Java der Operator = verwendet. Als *Vergleichsoperator* kommt (wie auch in Java) zum Einsatz.

Auf die Verwendung eines Anweisungsblocks könnte bei Schleifen verzichtet werden, wenn der Schleifeninhalt ohnehin nur aus einer einzigen Anweisung besteht. Dennoch empfiehlt sich die konsequente Benutzung geschweifter Klammern, um die Lesbarkeit zu erhöhen, Mehrdeutigkeiten zu vermeiden (wie beispielsweise beim *dangling else*) und das Hinzufügen weiterer Anweisungen zu vereinfachen. Für Java wird dies aus den gleichen Gründen analog empfohlen [Sun 1999].

2.2.2 Kommentare

Kommentare beginnen mit „/*“, enden mit „*/“, und dürfen nicht geschachtelt werden. Alternativ kann seit C99 ein Kommentar auch mit „//“ begonnen werden, der sich bis zum Zeilenende erstreckt.

2.2.3 Namen/Bezeichner

Namen bzw. *Bezeichner* bestehen aus Buchstaben und Ziffern, wobei das erste Zeichen ein Buchstabe sein muss. Zu den Buchstaben wird auch der *Unterstrich* „_“ gezählt.

2.2.4 Schlüsselworte

Die folgende Tabelle enthält alle *Schlüsselworte* von C99:

auto	**double**	**inline**	**sizeof**	**volatile**
break	**else**	**int**	**static**	**_Bool**
case	**enum**	**long**	**struct**	**_Complex**
char	**extern**	**register**	**switch**	**_Imaginary**
const	**float**	**restrict**	**typedef**	
continue	**for**	**return**	**union**	
default	**goto**	**short**	**unsigned**	
do	**if**	**signed**	**void**	

Einige der Schlüsselwörter wie etwa **auto** oder **register** existieren nur noch aus Kompatibiltätsgründen zu früheren Versionen und einige (**_Bool**, **_Complex** und **_Imaginary**) werden nur intern verwendet.

2.2.5 Leerzeichen

Zu den als Trenner dienenden *Leerzeichen* (zusammenfassend auch Leerraum bzw. im Englischen *white-space characters* genannt) gehören neben dem eigentlichen Leerzeichen auch Tabulatoren (horizontal und vertikal), Zeilentrenner (*line feed*) und der Seitenvorschub (*form feed*).

Diese werden in C nicht weiter voneinander unterschieden, d.h. jede Sequenz von Leerzeichen ist äquivalent zu einem einzelnen Leerzeichen. Die Ausnahmen hiervon sind nur die Kommentare, die mit „//“ beginnen und die (noch einzuführenden) Direktiven des Präprozessors, die jeweils durch das Zeilenende beendet werden.

Kapitel 3

Ein erster Blick auf den Präprozessor

3.1 Makroprozessoren

Makroprozessoren sind Programme, die weitgehend den Eingabetext in den Ausgabetext unverändert kopieren, jedoch selektiv einzelne Zeichenfolgen durch andere Texte ersetzen. Die Ersetzungen werden über (üblicherweise parametrisierbare) Makros definiert.

Makros erlauben es, den Schreibaufwand zu reduzieren, die Duplikation von Texten zu vermeiden, Programmtexte übersichtlicher zu gestalten und künftige Änderungen zu erleichtern. Aus diesem Grunde waren Makroprozessoren schon lange vor C in Verbindung mit Assembler und anderen Programmiersprachen im Einsatz. Neben auf bestimmte Sprachen spezialisierten Makroprozessorn gibt es auch sprachunabhängige Prozessoren wie etwa *m4*:

```
doolin$ cat morgen.m4
define(`gm', `Guten Morgen, $1!')dnl
gm(Anne)
Guten Abend, Marianne!
gm(Heinz)
doolin$ m4 morgen.m4
Guten Morgen, Anne!
Guten Abend, Marianne!
Guten Morgen, Heinz!
doolin$
```

3.2 Integration eines Makroprozessors

Die Integration eines Makroprozessors in die Programmiersprache C erlaubte es, in der eigentlichen Sprache auf fortgeschrittene Techniken zu verzichten, die teilweise aufwändig zu implementieren gewesen wären:

- Die ursprüngliche Programmiersprache (entsprechend dem K&R-Standard) verzichtete auf Konstantendefinitionen. Stattdessen wurden entsprechende Makros verwendet.
- C verzichtet auf ein Modulkonzept. Stattdessen werden die Funktionsdeklarationen in separate Dateien ausgelagert, die dann mit *include*-Anweisungen des Makropro-

zessors in alle C-Quellen hineinkopiert werden, die diese benötigen. Entsprechend bleibt dem Übersetzer die Aufgabe erspart, Bibliotheken ausfindig zu machen.

- Generische Programmiertechniken (*templates*) und eingebettete Funktionen (*inline functions*) lassen sich ebenfalls in rudimentärer Form auf Basis von Makros realisieren.

Die vereinfachte Implementierung dieser Techniken zieht jedoch auch wesentliche Nachteile mit sich:

- Statt sich auf eine in sich konsistente Sprache zu beschränken, werden zwei voneinander unabhängige Sprachen miteinander vermischt: Die „Wirtssprache" (wie etwa C) und die Sprache des Makroprozessors. Das schafft viele Probleme, da sich beispielsweise die Fehlermeldungen des Übersetzers auf die Ausgabe des Makroprozessors beziehen und nicht etwa auf die ursprüngliche Quelle.
- Die Semantik der Parameterübergabe bei einem Makro (*call by text*) weicht dramatisch ab von der Semantik der Parameterübergabe der Wirtssprache (bei C *call by value*).

Bjarne Stroustrup, der Autor der nachfolgenden Programmiersprache C++, identifiziert in [Stroustrup 1994] den in C integrierten Makroprozessor (kurz Cpp genannt) als eines der Hauptprobleme:

> The character and file orientation of the preprocessor is fundamentally at odds with a programming language designed around the notions of scopes, types, and interfaces.
> ...
> I'd like to see Cpp abolished.

Die neueren Standards für C bemühen sich, einige der Techniken direkt in C anzubieten, die zuvor nur über den Cpp zur Verfügung standen:

- C89 führte Konstantendefinitionen ein.
- C99 unterstützt eingebettete Funktionen.

Entsprechend sollte die Benutzung des Präprozessors minimiert werden.

3.3 Cpp – der C-Präprozessor

Dem eigentlichen C-Compiler ist der Cpp vorgeschaltet. Dieser wird automatisch mit dem Aufruf von *gcc* (und jedem anderen C-Compiler) aktiviert. Er kann auch direkt aufgerufen werden mit *cpp* oder *gcc -E* (siehe *man gcc* bzw. *man cpp*).

Direktiven des Cpp beginnen direkt am Zeilenanfang (ohne führende Leerzeichen!) mit einem #. Danach folgen der Name der Direktive und die zugehörigen Parameter. Zwischen dem führenden # und dem Namen der Direktive sind (abgesehen vom Zeilentrenner) beliebig viele Leerzeichen zulässig.

3.4 define-Direktive

Makros werden mit der Direktive *define* definiert. Makrodefinitionen selbst werden durch den Cpp durch eine Leerzeile ersetzt.

Programm 3.1: Verwendung der define-Direktive (*makros.c*)

```
#define MAX 10 /* so kann eine Konstante definiert werden */
#define MAX1 = 10 /* so nicht! */
#define MAX2 10; /* so auch nicht! */

int main() {
    int x = MAX; /* OK: int x = 10; */
    int y = MAX1; /* FALSCH: int y = = 10; */
    int z = MAX2 + 1; /* FALSCH: int z = 10; + 1; */
}
```

Das Beispiel Programm 3.1 zeigt einige der Fallstricke beim Umgang mit der *define*-Direktive. Der *Cpp* liefert zu diesem Programm folgende Ausgabe:

```
doolin$ gcc -E makros.c
# 1 "makros.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "makros.c"




int main() {
    int x = 10;
    int y = = 10;
    int z = 10; + 1;
}
doolin$
```

Wenn die Kommentare bei der Ausgabe erhalten bleiben sollten, so empfiehlt sich die Option -C: *gcc -E -C makros.c*. Die Angaben der Zeilennummern lassen sich mit der Option *-P* unterdrücken:

```
doolin$ gcc -E -P makros.c







int main() {
    int x = 10;
    int y = = 10;
    int z = 10; + 1;
}
doolin$
```

Welche *Unterschiede* gibt es zwischen *Variablen* und *Makros*?

Beispiele:

- *Variable:* **const int** *MAX* = 3;
- *Makro:* **#define** MAX 3

Eine Variable ist ein Name für eine Speicherstelle. Die 3 steht also irgendwo im Speicher. Dagegen wird beim Makro nur die 3 an der Stelle eines Makroaufrufs eingesetzt. Entsprechend findet sich das Makro auch nicht mehr zur Laufzeit.

3.5 include-Direktive

Der Ersatztext einer *include*-Direktive ist der Inhalt der genannten Datei. Damit werden i. A. Vereinbarungen oder andere Direktiven in die zu übersetzende Quelle „hereinkopiert". Diese Dateien heißen im Kontext von C-Programmen *Header-Dateien* (bzw. *header files*) und haben üblicherweise die Endung *.h*.

Es gibt eine ganze Reihe solcher Header-Dateien im Verzeichnis */usr/include* – diese sind die Schnittstellen der C-Bibliothek und entsprechen (im Rahmen der beschränkten Möglichkeiten von C) den öffentlichen Teilen einer Java-Bibliothek. Dies ist, wie später gezeigt wird, auch der Weg zur Modularisierung in C.

Die folgenden beiden Dateien sind ein Beispiel für die Verwendung der *include*-Direktive:

Programm 3.2: Verwendung der *include*-Direktive (*makros1.c*)

```
#include "defs.h" /* Einfuegen von defs.h */

int main() {
    int x = MAX;
}
```

Programm 3.3: Eine winzige Header-Datei (*defs.h*)

```
#define MAX 3
int y = MAX;
```

Der *Cpp* liefert zu diesem Programm nun die folgende Ausgabe:

```
doolin$ gcc -E makros1.c
# 1 "makros1.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "makros1.c"
# 1 "defs.h" 1

int y = 3;
# 2 "makros1.c" 2

int main() {
    int x = 3;
}
doolin$
```

Kapitel 4

Ein- und Ausgabe

4.1 *stdin, stdout* und *stderr*

Standardmäßig gibt es drei Kanäle für die Ein- und Ausgabe. Die *Standardeingabe* (*stdin*) entspricht der Eingabe auf der Konsole. Entsprechend ist die *Standardausgabe* (*stdout*) die „normale“ Ausgabe auf der Konsole. Mit *stderr* wird die *Fehler- bzw. Diagnoseausgabe* bezeichnet. In der Shell (= Kommandozeile) kann *stdin* mittels <, *stdout* mittels > und *stderr* mittels 2>*err* umgelenkt werden:

Programm 4.1: Ausgabe mit *puts()* und *fputs()* (*out.c*)

```
#include <stdio.h>

int main() {
    /* puts gibt am Ende einen Zeilentrenner aus */
    puts("Ich komme nach stdout ...");
    /* fputs fuegt keinen Zeilentrenner hinzu */
    fputs("Ich komme nach stderr ...\n", stderr);
}
```

```
thales$ gcc -Wall out.c
thales$ a.out
Ich komme nach stdout ...
Ich komme nach stderr ...
thales$ a.out > out.stdout 2> out.stderr
thales$ cat out.stdout
Ich komme nach stdout ...
thales$ cat out.stderr
Ich komme nach stderr ...
thales$
```

4.2 Ausgabe nach stdout

Die Funktion *puts()* gibt eine Zeichenkette auf die Standardausgabe aus:

int *puts*(**const char*** *s*);

Der *Rückgabewert* von *puts()* ist die Anzahl der geschriebenen Zeichen. Nach der Ausgabe der Zeichenkette gibt *puts()* noch einen zusätzlichen Zeilentrenner aus – im Gegensatz zur Funktion *printf()*.

Die Funktion *printf()* kann formatiert in die Standardausgabe schreiben:

int *printf*(**const char*** *format*, */*args */...*);

- *printf()* liefert die *Anzahl* der ausgegebenen Zeichen zurück.

- Ein *Format* ist eine Zeichenfolge, die aus direkt auszugebendem Text und eingestreuten Platzhaltern besteht. Beispiel:
  printf("Ich␣bin␣%d␣Jahre␣alt.", 3);
  Hier ist *%d* der Platzhalter.

- Von Ausnahmen abgesehen benötigen Platzhalter jeweils einen oder auch mehrere Parameter. Diese werden nacheinander der Parameterliste hinter dem Format entnommen. Fehlen am Ende die Parameter zu einem der Platzhalter, so ist das Resultat nicht definiert.

- Platzhalter bestehen aus dem Zeichen %, den Optionen (*Flags*), einer Mindestbreite (*MinWidth*), der Genauigkeit (*Precision*) und einem Zeichen, das die Art des Platzhalters bzw. der durchzuführenden Konvertierung beschreibt (*ConvChar*):

$$
\begin{array}{rcl}
\langle \text{ConvSpec} \rangle & \longrightarrow & \text{„\%“ } \{ \langle \text{Flag} \rangle \} \ [\langle \text{MinWidth} \rangle] \ [\text{„.“ } \langle \text{Precision} \rangle] \\
 & & \langle \text{SizeModifier} \rangle \ \langle \text{ConvChar} \rangle \\
\langle \text{Flag} \rangle & \longrightarrow & \text{„-“} \mid \text{„+“} \mid \text{„ “} \mid \ldots \\
\langle \text{MinWidth} \rangle & \longrightarrow & \langle \text{Digit} \rangle \ \{ \langle \text{Digit} \rangle \} \mid \text{„*“} \\
\langle \text{Precision} \rangle & \longrightarrow & \langle \text{Digit} \rangle \ \{ \langle \text{Digit} \rangle \} \mid \text{„*“} \\
\langle \text{SizeModifier} \rangle & \longrightarrow & \text{„ll“} \mid \text{„l“} \mid \text{„L“} \mid \text{„h“} \mid \text{„hh“} \mid \text{„j“} \mid \text{„z“} \mid \text{„t“} \\
\langle \text{ConvChar} \rangle & \longrightarrow & \text{„d“} \mid \text{„i“} \mid \text{„o“} \mid \text{„u“} \mid \text{„x“} \mid \text{„X“} \mid \ldots
\end{array}
$$

  Beispiele für Platzhalter: *%-3s*, *%d*, *%3.2f*

- * als minimale Breite oder Genauigkeit bedeutet, dass der nächste Parameter aus der verbliebenen Parameterliste als ganze Zahl behandelt wird, die die minimale Breite bzw. Genauigkeit in variabler Weise angibt.

Es gibt unter anderem folgende *Konvertierungszeichen* (*ConvChar*):

Zeichen für die Konvertierung	Effekt
`d,i,o,u,x,X`	ganze Zahl: `d`, `i` und `u` für Dezimaldarstellung, wobei `u` von einer Darstellung ohne Vorzeichen ausgeht (*unsigned*); `x`, `X` für eine hexadezimale Darstellung, wobei `x` die Kleinbuchstaben „a"–„f" und `X` die Großbuchstaben „A"–„F" benutzt; unmittelbar davor kann ein `h` (für **short int**) oder `l` (für **long int**) stehen
`f`	Reelle Zahlen in Gleitkommadarstellung ([-]mmm.nnnnnnn): die Anzahl der Nachkommastellen wird durch die Genauigkeitsangabe festgelegt; Voreinstellung: 6
`e,E`	Reelle Zahlen in Exponentialdarstellung
`c`	Zeichen: der zugehörige Parameter ist eine ganze Zahl, die implizit zu einem **unsigned char** konvertiert wird
`s`	Zeichenketten: der Parameter muß eine durch ein Null-Byte terminierte Zeichenfolge sein; eine Genauigkeitsangabe wird als maximal auszugebende Zeichenzahl interpretiert
`p`	Zeiger: die Adresse wird numerisch in einem systemabhängigen Format (typischerweise hexadezimal) ausgegeben
`%`	die Folge %% gibt ein % aus

Für die die Optionen (*Flags*) gibt u. a. folgende Wahlmöglichkeiten:

Option	Bedeutung
-	linksbündige Ausgabe
+	auch ein positives Vorzeichen wird ausgegeben
Leerzeichen	statt dem pos. Vorzeichen wird ein Leerzeichen ausgeben

Bei numerischen Datentypen ist es notwendig, den genauen Parametertyp zu spezifizieren (*SizeModifier*), wenn es nicht um **int** oder **double** handelt:

Zeichen	Datentyp
ll	**long long int** oder **unsigned long long int**
l	**long int** oder **unsigned long int**
L	**long double**
h	**short int** oder **unsigned short int**
hh	**char** oder **unsigned char** (ab *C99*)
j	*intmax_t* oder *uintmax_t*
z	*size_t* oder *size_t*
t	*ptrdiff_t* oder *ptrdiff_t*

Programm 4.2 veranschaulicht die Verwendung von *printf()* und *puts()*:

Programm 4.2: Ausgabe mit *puts()* und *printf()* (*stdout.c*)

```
#include <stdio.h> /* enthaelt die Deklarationen aller Ein-/Ausgabe-Funktionen */

int main() {
    puts("----+----+----+----+----+----+"); /* Lineal ;-) (ohne "\n
        "!) */

    printf("%s_\n", "Donaudampfschiff"); /* "\n" erzeugt Zeilenumbruch */
    printf("%20s_\n", "Donaudampfschiff"); /* min. Breite */
    printf("%-20s_\n", "Donaudampfschiff"); /* min. Breite + linksbuendig */
    printf("%.10s_\n", "Donaudampfschiff"); /* max. Breite */
    printf("%-10.10s_\n", "Donau");  /* min. & max. Breite + linksbuendig */
```

```
    puts("----+----+----+----+----+----+");

    printf("%d_\n", 254); /* dezimal */
    printf("%5d_\n", 254); /* dezimal (mit min. Breite) */
    printf("%x_\n", 254); /* hexadezimal */

    puts("----+----+----+----+----+----+");

    printf("%f_\n", 3.1415926);  /* Fliesskomma */
    printf("%10f_\n", 3.1415926); /* min. Breite */
    printf("%.3f_\n", 3.1415926); /* Anzahl der Nachkommastellen */
    printf("%10.3f_\n", 3.1415926); /* min. Breite + Anz. d. Nachkommast. */
    printf("%+10.3f_\n", 3.1415926); /* Vorzeichen immer anzeigen */
}
```

```
thales$ gcc -Wall -std=c99 stdout.c
thales$ a.out
----+----+----+----+----+----+
Donaudampfschiff_
    Donaudampfschiff_
Donaudampfschiff    _
Donaudampf_
Donau     _
----+----+----+----+----+----+
254_
  254_
fe_
----+----+----+----+----+----+
3.141593_
  3.141593_
3.142_
     3.142_
    +3.142_
thales$
```

Weitere Hinweise zu *printf()* und *puts()* finden sich in den zugehörigen Manualseiten (*man -s 3c printf* (Solaris), *man 3 printf* (Linux) bzw. *man puts*).

4.3 Ausgabe nach stderr

Die Funktion *fputs()* gibt die Zeichenkette *s* in die Dateiverbindung *stream* – in unserem Fall *stderr* – aus:

int *fputs*(**const char*** *s*, *FILE** *stream*);

Der einzige Unterschied zu *puts()* – abgesehen von dem zusätzlichen Argument – ist, dass von *fputs()* nur die übergebene Zeichenkette ausgegeben wird, wohingegen *puts()* noch einen Zeilenumbruch anhängt.

Bsp.: *fputs*("Hallo", *stderr*);

Die Funktion *fprintf()* kann analog zu *printf()* formatiert ausgeben.

int *fprintf*(*FILE** *stream*, **const char*** *format*, */*args */...*);

Im Unterschied zu *printf()* erwartet *fprintf()* noch die Angabe einer Dateiverbindung. Für Diagnosemeldungen ist in <*stdio.h*> die Variable *stderr* definiert (manchmal auch als Makro).

Bsp.: *fprintf*(*stderr*, "Hallo");

4.4 Eingabe von stdin

Die Funktion *scanf()* liest formatiert von der Standardeingabe ein:
int *scanf*(**const char*** *format*, ...);

- Das Format besteht aus Zeichen für die Konvertierung und weiteren Zeichen.
- Konvertierung:
  `%` gefolgt von optionalen Zeichen zur Modifikation, gefolgt von einem Konvertierungszeichen
- Andere Zeichen (außer Konvertierungszeichen und Leerzeichen) müssen mit den Zeichen im Eingabestrom übereinstimmen. Leerzeichen, Tabs (\t) und Zeilentrenner (\n), veranlassen *scanf()* alle Leerzeichen der Eingabe zu überlesen.
- Zu beachten ist, dass C nur die Werteparameterübergabe unterstützt. Aus diesem Grund muss bei *scanf()* der *Adressoperator* vor dem einzulesenden Parameter eingefügt werden. (Bei Arrays und Zeigern fällt das weg – dazu später mehr.) Damit wird ein Zeiger auf die Variable übergeben und *scanf()* greift über diesen Zeiger auf die Variable durch.

Bsp.: *scanf*("%d", &*n*); liest in die ganzzahlige Variable *n* einen Wert von der Standardeingabe (*stdin*) ein.
scanf() hat einen ganzzahligen Rückgabewert, der die Anzahl der tatsächlich erfolgten Variablenzuweisungen wiedergibt.

Konvertierungs-zeichen	Wirkung
d	Dezimal-Konstante
o	Oktal-Konstante
x,X	Hexadezimal-Konstante
f	Gleitkommazahl
s	Zeichenfolge bis zum nächsten Leerzeichen (einschließlich '\t' oder '\n'); ein Null-Byte ('\0') wird angefügt
c	Nächstes Zeichen; um das nächste nicht-leere Zeichen zu lesen: %1c
%	Liest %-Zeichen ohne Zuweisung

Vor dem Konvertierungszeichen kann mit einer Dezimalzahl die maximale Feldlänge spezifiziert werden. Wenn ein Stern vor dem Konvertierungszeichen angegeben wird, unterbleibt die Zuweisung.

Programm 4.3: Eingabe mit *scanf()* (*scanf.c*)

```
#include <stdio.h>

int main() {
    int anzahl, i, j;
    float f;
    char s[50];

    anzahl = scanf("i=%d␣%f␣%s", &i, &f, s);
    puts("----------------------------------");
```

```
    printf("Anzahl:␣%d␣|␣i=%d,␣f=%f,␣s=%s\n", anzahl, i, f, s);
    puts("------------------------------------");

    anzahl = scanf("%2s␣%*d␣%2d", s, &i);
    puts("------------------------------------");
    printf("Anzahl:␣%d␣|␣s=%s,␣i=%d\n", anzahl, s, i);
    puts("------------------------------------");
}
```

```
thales$ gcc -Wall -std=c99 scanf.c
thales$ a.out
i= 1 .2 hallo welt
------------------------------------
Anzahl: 3 | i=1, f=0.200000, s=hallo
------------------------------------
------------------------------------
Anzahl: 1 | s=we, i=1
------------------------------------
thales$ a.out
j= 4711 0815
------------------------------------
Anzahl: 0 | i=4, f=0.000000, s=
------------------------------------
------------------------------------
Anzahl: 2 | s=j=, i=8
------------------------------------
thales$ a.out
1234 a
------------------------------------
Anzahl: 0 | i=4, f=0.000000, s=
------------------------------------
------------------------------------
Anzahl: 1 | s=12, i=4
------------------------------------
thales$
```

Die Funktion *gets()* liest eine Zeile von der Standardeingabe (*stdin*) in eine Zeichenfolge ein:

char* *gets*(**char*** *s*);

Wurde ein Zeilentrenner in der Eingabe gefunden, so wird dieser von *gets()* entfernt – im Gegensatz zu *fgets()*. Außerdem kann *gets()* nicht überprüfen, ob hinter dem Zeiger *s* genügend Speicherplatz vorhanden ist. Ist die Eingabezeile länger als der zur Verfügung stehende Platz, kommt es zu einem *Puffer-Überlauf*.

Die Funktion *fgets()* hingegen liest aus einer beliebigen Dateiverbindung (*file pointer*) und zwar max. soviele Zeichen, wie im Puffer untergebracht werden können:

char* *fgets*(**char*** *s*, **int** *n*, *FILE** *stream*);

Mit n wird dabei die Größe des Puffers angegeben, wobei dann maximal $n-1$ Zeichen gelesen werden (es muss ja noch das abschließende Null-Byte in den Puffer geschrieben werden – dazu aber später mehr). Folgendes Beispiel illustriert die Verwendung dieser beiden Funktionen:

Programm 4.4: Eingabe mit *gets()* und *fgets()* (*in.c*)

```
#include <stdio.h>

const int BUFSIZE = 10;

int main() {
    char s[BUFSIZE];

    fputs("Geben␣Sie␣eine␣Zeile␣ein:␣", stdout);
    /* eine Zeile von stdin einlesen */
    gets(s);
    printf("ECHO:␣>%s<\n", s);

    /* SICHERERE VARIANTE */
    fputs("Geben␣Sie␣eine␣Zeile␣ein:␣", stdout);
    /* eine Zeile von stdin einlesen,
       max. aber nur BUFSIZE−1 Zeichen */
    fgets(s, BUFSIZE, stdin);
    printf("ECHO:␣>%s<\n", s);
}
```

```
thales$ gcc in.c
thales$ a.out
Geben Sie eine Zeile ein: Wow, C macht ja richtig Spass!
ECHO: >Wow, C macht ja richtig Spass!<
Geben Sie eine Zeile ein: Naja, ein Versuch war's wert!
ECHO: >Naja, ein<
thales$
```

4.5 Weitere Ein- und Ausgabe-Funktionen

Zum Einlesen und Ausgeben von einzelnen Zeichen gibt es die Bibliotheksfunktionen *getc()*, *fgetc()*, *getchar()*, *ungetc()*, *putc()*, *fputc()*, *putchar()*. Nähere Infos dazu gibt es in den zugehörigen Manualseiten (*man getc*, ...).

Kapitel 5

Kontrollstrukturen

5.1 Übersicht

⟨expression-statement⟩	⟶	⟨expression⟩ „;“
⟨labeled-statement⟩	⟶	⟨label⟩ „:“ ⟨statement⟩
⟨label⟩	⟶	⟨named-label⟩
	⟶	⟨case-label⟩
	⟶	⟨default-label⟩
⟨case-label⟩	⟶	**case** ⟨constant-expression⟩
⟨default-label⟩	⟶	**default**
⟨iterative-statement⟩	⟶	⟨while-statement⟩
	⟶	⟨do-statement⟩
	⟶	⟨for-statement⟩
⟨conditional-statement⟩	⟶	**if** „(“ ⟨expression⟩ „)“ ⟨statement⟩
	⟶	**if** „(“ ⟨expression⟩ „)“ ⟨statement⟩
		else ⟨statement⟩
⟨while-statement⟩	⟶	**while** „(“ ⟨expression⟩ „)“ ⟨statement⟩
⟨do-statement⟩	⟶	**do** ⟨statement⟩ **while** „(“ ⟨expression⟩ „)“
⟨for-statement⟩	⟶	**for** „(“ [⟨initial-clause⟩] „;“ [⟨expression⟩] „;“
		[⟨expression⟩] „)“ ⟨statement⟩
⟨initial-clause⟩	⟶	⟨expression⟩
	⟶	⟨declaration⟩
⟨switch-statement⟩	⟶	**switch** „(“ ⟨expression⟩ „)“ ⟨statement⟩
⟨break-statement⟩	⟶	**break** „;“
⟨continue-statement⟩	⟶	**continue** „;“
⟨return-statement⟩	⟶	[⟨expression⟩] „;“
⟨goto-statement⟩	⟶	**goto** ⟨named-label⟩ „;“
⟨null-statement⟩	⟶	„;“

Anmerkung 1: In C ist das Semikolon „;“ der *Abschluss* einer Anweisung (*statement*) und nicht *Trenner* zwischen zwei Anweisungen (wie in einigen anderen Programmiersprachen).

Anmerkung 2: *Ausdrücke* (*expressions*) sind deswegen als Anweisungen sinnvoll, da sie auch Nebeneffekte haben können, wie zum Beispiel der Aufruf einer Funktion, die Inkrementierung einer Variablen oder eine Zuweisung.

Bsp.: *puts*("Hallo␣Welt!"); bzw. $x = y$++;

5.2 if-Anweisung

Erst in C99 wurde ein Boolean-Datentyp eingeführt. Zuvor wurde stattdessen **int** verwendet, wobei 0 für *false* stand und alle anderen Werte für *true*. Beginnend mit C99 wird der Datentyp *bool* in <*stdbool.h*> definiert zusammen mit den zugehörigen Konstanten *true* und *false*. Dessen ungeachtet blieb die Semantik erhalten, dass generell ganzzahlige Werte bei Bedingungen akzeptiert und entsprechend der alten Regeln interpretiert werden.

bool	int-Wert
true	1 bzw. ungleich 0
false	0

Eine typische Fehlerquelle:

if (j = 5) *tu_etwas*();

Gemeint war aber folgendes:

if (j == 5) *tu_etwas*();

Die erste Version ist syntaktisch korrekt. *tu_etwas*() wird aber immer aufgerufen, denn die Bedingung j = 5 hat als Nebeneffekt die Wertzuweisung von 5 an die Variable j und der Wert der Bedingung ist der zugewiesene Wert, also 5, d. h. ungleich 0, also *true*.

Die Anweisung im (möglicherweise nicht vorhandenen) *else-Zweig* wird ausgeführt, wenn die Bedingung nicht zutrifft. Das **else** wird jeweils dem „nächsten“ **if** zugeordnet:

Programm 5.1: Geschachtelte **if**-Anweisungen mit **else** (*if.c*)

```
#include <stdio.h>

int main() {
    int n;

    /* ganze Zahl einlesen */
    printf("n␣=␣");
    if (scanf("%d", &n) != 1)
        return 1;

    if (n >= 0)
        if (n >= 5)
            puts("n␣>=␣5");
        else /* zu wem gehoert dieses else wohl? */
            puts("else");
    return 0;
}
```

```
thales$ gcc -Wall if.c
thales$ a.out
n = 10
n >= 5
thales$ a.out
n = 4
else
thales$ a.out
n = -1
thales$
```

Im Folgenden ist das obige Programm durch einen Anweisungsblock übersichtlicher und eindeutiger gestaltet:

Programm 5.2: Sauber geklammerte **if**-Anweisungen mit **else** (*if1.c*)

```
#include <stdio.h>

int main() {
    int n;

    /* ganze Zahl einlesen */
    printf("n␣=␣");
    if (scanf("%d", &n) != 1)
        return 1;

    if (n >= 0) { /* macht die Sache klarer! */
        if (n >= 5) {
            puts("n␣>=␣5");
        } else { /* keine Frage mehr! */
            puts("else");
        }
    }
    return 0;
}
```

Wie in diesem Beispiel sollten generell die bedingt auszuführenden Anweisungsteile in geschweifte Klammern gesetzt werden, auch wenn dies bei nur einer einzelnen Anweisung wegfallen könnte. (Natürlich ist dies nur eine sinnvolle Richtlinie, die in Einzelfällen wie bei der Verwendung von **break** oder **continue** auch verletzt werden kann.)

Programm 5.3: **else-if**-Kette (*elseif.c*)

```
/*
    Lese ganze Zahlen von der Standardeingabe und fasse
    hintereinanderfolgende Zahlen des gleichen Werts in der Ausgabe zusammen
*/
#include <stdio.h>
#include <stdbool.h>

int main() {
    bool first = true; /* noch nichts eingelesen? */
    int current; /* aktuell eingelesene Zahl */
    int last; /* zuvor eingelesene Zahl (falls !first) */
```

```
    int count; /* Anzahl der aufeinanderfolgenden gleichen Zahlen */

    while (scanf("%d", &current) > 0) {
        if (first) {
            first = false;
            count = 1;
        } else if (current == last) {
            ++count;
        } else {
            printf("%d␣x␣%d\n", count, last);
            count = 1;
        }
        last = current;
    }
    if (!first) {
        printf("%d␣x␣%d\n", count, last);
    }
}
```

C bietet kein Schlüsselwort für **else-if**. Stattdessen beginnt der **else**-Fall sofort mit einem **if**. Bei **else-if**-Ketten sollte auf ein zusätzliches Einrücken und das Verwenden geschweifter Klammern verzichtet werden, so dass die Kette als solche gut erkennbar bleibt.

5.3 while-Schleife

Die **while**-Schleife führt die Anweisung bzw. den Anweisungsblock solange aus, wie die Bedingung *true* ist, d.h. der Wert des Bedingungsausdrucks ungleich 0 ist. Die Überprüfung der Bedingung findet jeweils *vor* einem Schleifendurchlauf statt. (Es ist also auch möglich, dass der Schleifenrumpf überhaupt nicht durchlaufen wird.)

Beispiel: Zeichenweises Lesen von *stdio* und Zählen der Leerzeichen (Die Bibliotheks-Funktion *getchar()* aus <*stdio.h*> liest ein Zeichen von *stdin* und liefert es als **int**-Wert. Wenn dieser Wert nicht-negativ ist, war die Einlese-Operation erfolgreich. Wird hingegen *EOF* zurückgeliefert (in <*stdio.h*> als -1 definiert), konnte kein Zeichen mehr eingelesen werden.

Programm 5.4: **while**: Zählen von Leerzeichen (*getchar.c*)

```
#include <stdio.h>

int main() {
    int ch, anzahl = 0;

    while ((ch = getchar()) != EOF) {
        if (ch == '␣') {
            anzahl++;
        }
    }

    printf("Anzahl␣der␣Leerzeichen:␣%d\n", anzahl);
}
```

Hinweise zu diesem Programm:

- Zu beachten ist hier, dass die Variable *ch* hier als **int** deklariert wird und nicht etwa als **char**, da nur der Wertebereich des Datentyps **int** ausreicht, um alle möglichen Zeichenwerte und *EOF* voneinander unterscheiden zu können. In C wird **char** als kleiner ganzzahliger Datentyp betrachtet und entsprechend sind **char** und **int** miteinander kompatibel. Wenn hier **char** verwendet wird, ist nicht einmal definiert, in welcher Form es schief gehen kann, da bei **char** nicht feststeht, ob es mit oder ohne Vorzeichen kommt. Kommt es ohne Vorzeichen, erhalten wir hier eine Dauerschleife. Andernfalls könnte es sein, dass etwa das Zeichen '\0377' mit *EOF* zusammenfällt.
- Der Inkrement-Operator ++ erhöht die Variable um 1. (Dazu später noch mehr.)

5.4 do-while-Schleife

Die **do-while**-Schleife führt die Anweisung bzw. den Anweisungsblock solange aus, wie die Bedingung *true* ist, d.h. der Wert des Bedingungsausdrucks ungleich Null ist. Die Überprüfung der Bedingung findet jeweils *nach* einem Schleifendurchlauf statt. (Es gibt also immer mindestens einen Schleifendurchlauf.)

Programm 5.5: **do-while**: Zählen von Leerzeichen bis zum Zeilenende (*getchar1.c*)

```
#include <stdio.h>

int main() {
    int ch, anzahl = 0;

    do {
        if ((ch = getchar()) == '␣') {
            anzahl++;
        }
    } while (ch != '\n' && ch != EOF);

    printf("Anzahl␣der␣Leerzeichen␣in␣der␣ersten␣Zeile:␣%d\n", anzahl);
}
```

Programm 5.6: **do-while**: Überzählige Leerzeichen herausfiltern (*ignore.c*)

```
#include <stdio.h>
#include <ctype.h> /* wg. isspace() */

void skip_spaces() {
    int ch;

    do {
        ch = getchar();
    } while (ch != EOF && isspace(ch));

    if (ch != EOF) {
        /* wir haben ein Zeichen zu weit gelesen
            => wieder zurueck in die Eingabe damit
            VORSICHT: nur fuer ein Zeichen garantiert! */
        ungetc(ch, stdin);
    }
}
```

```
int main() {
    int ch;

    while ((ch = getchar()) != EOF) {
        putchar(ch);
        if (ch == '␣') {
            skip_spaces();
        }
    }
}
```

Anmerkungen:

- Die Angabe von **void** als Datentyp für den Rückgabewert einer Funktion bedeutet, dass diese Funktion keine Werte zurückgibt.
- Die Funktion *ungetc*() stellt ein Zeichen zurück in den Eingabepuffer. Allerdings können nicht beliebig viele Zeichen auf diese Weise zurückgegeben werden. Garantiert ist dies für nur jeweils ein Zeichen – bis dieses gelesen ist (dann für das nächste usw.).

5.5 for-Schleife

Die **for**-Anweisung hat folgende Grundstruktur:

```
for (/* Initialisierung */; /* Bedingung */; /* Inkrementierung */)
    /* Anweisung */
```

Ein typisches Beispiel für die Verwendung einer **for**-Schleife ist das „Hochzählen“ einer Zähl-Variable in einem bestimmten Bereich:

```
int i;
for (i = 1; i <= 10; i++) {
    printf("%d\n", i);
}
```

Alternativ zu einer **for**-Schleife kann auch die äquivalente **while**-Schleife verwenden werden:

```
/* Initialisierung */
while (/* Bedingung */) {
    /* Anweisung */
    /* Inkrement */
}
```

Bei obigem Beispiel sieht das dann wie folgt aus:

```
int i;
i = 1;
while (i <= 10) {
    printf("%d\n", i);
    i++;
}
```

Beginnend mit C99 ist es auch zulässig, die Schleifenvariable innerhalb des Initialisierungsteils zu deklarieren:

```
for (int i = 1; i <= 10; i++) {
    printf("%d\n", i);
}
```

In diesem Falle ist die Schleifenvariable *i* nur innerhalb der Schleife sichtbar. Dies ist vorzuziehen, da dies die Lesbarkeit und Wartbarkeit des Programmtexts erhöht. Sonst tendieren Schleifenvariablen dazu, die Liste der lokalen Variablen unübersichtlich zu machen und Konflikte können nicht sicher ausgeschlossen werden.

Jeder der drei Ausdrücke in einer **for**-Schleife kann auch *leer* sein; eine „leere" Bedingung stellt eine stets erfüllte Bedingung dar. Somit wird die Schleife in diesem Fall zur *Endlosschleife*, die nur mit **return** oder **break** wieder verlassen werden kann!

Endlosschleife:

```
while (1) { /* ... */ };
```

oder

```
for(;;) { /* ... */ };
```

5.6 continue-Anweisung

Die **continue**-Anweisung dient dazu, vorzeitig den nächsten Schleifendurchlauf zu starten, d.h. die restlichen Anweisungen des Anweisungsteils werden übersprungen, so dass bei der **for**-Schleife noch die inkrementierende Anweisung durchgeführt wird und danach der nächste Schleifentest erfolgt.

Programm 5.7: Verwendung von **continue** (*continue.c*)

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 20; i++) {
        if (i % 4 == 0) continue;
        printf("%d\n", i);
    }
}
```

Programm 5.8: Zusammenhang zwischen **for**, **while** und **continue** (*continue1.c*)

```
#include <stdio.h>

int main() {
    /* Folgendes Beispiel zeigt, dass die Umformulierung einer
       for— in eine while—Schleife bzgl. continue nicht ganz identisch ist!
    */
    int i = 1;
    while (i <= 20) {
        // FALSCH: Endlosschleife mit i=4,4,4,...
        if (i % 4 == 0) continue;
        printf("%d\n", i);
        i++;
    }
}
```

5.7 break-Anweisung

Die **break**-Anweisung dient zum vorzeitigen Verlassen der innersten Schleife oder **switch**-Anweisung (analog zu Java).

Programm 5.9: Verwendung von **break** (*break.c*)

```
#include <stdio.h>

int ggt(int x, int y) {
    while (y != 0) {
        int tmp = x % y;
        x = y; y = tmp;
    }
    return x;
}

int main() {
    for(;;) {
        printf("Geben␣Sie␣zwei␣positive␣ganze␣Zahlen␣ein:␣");
        int x, y;
        if (scanf("%d␣%d", &x, &y) != 2) break;
        printf("ggT(%d,␣%d)␣=␣%d\n", x, y, ggt(x, y));
    }
}
```

5.8 switch-Anweisung

Die **switch**-Anweisung kann zu einer Fallunterscheidung verwendet werden. Programm 5.10 zeigt eine erste Verwendung der *switch*-Anweisung. Programm 5.11 ist eine Anwendung, bei der verschiedene Fälle gemeinsam behandelt werden.

Programm 5.10: Beispiel für die **switch**-Anweisung (*switch.c*)

```
#include <stdio.h>

int main() {
    int i;

    printf("Geben␣Sie␣eine␣ganze␣Zahl:\n");
    while (scanf("%d", &i) > 0) {
        switch (i) {
            case 0:
                printf("0␣eingegeben\n");
                break; /* springt ans Ende von switch */
            case 1:
                printf("1␣eingegeben\n");
                break; /* dito */
            default: /* fuer alle anderen Faelle, also nicht 0 oder 1 */
                printf("Weder␣0␣noch␣1\n");
        } /* Ende von switch */
        printf("Noch␣eine␣Zahl?\n");
    }
```

}

Zur Semantik und Verwendung:

- Nach der Auswertung des **switch**-Ausdrucks wird bei der Anweisung des passenden **case** fortgefahren.
- Auch die folgenden „Fälle“ werden abgearbeitet, falls dies nicht durch ein explizites **break** verhindert wird.
- Der **switch**-Ausdruck muss einen ganzzahligen Typ oder einen Aufzählungstyp (**enum**) haben.
- Der Ausdruck bei einem **case** muß ein *konstanter* Ausdruck sein. Zulässig sind ganzzahlige Werte, Zeichenkonstanten oder Konstanten eines Aufzählungstyps (**enum**).
- Die bei den einzelnen Fällen angegebenen Konstanten müssen zueinander disjunkt sein.
- Trifft keiner der Fälle zu, geht es bei **default** weiter, falls dieser existiert. Ansonsten wird die gesamte **switch**-Anweisung übersprungen.
- Aus Gründen der Lesbarkeit empfiehlt es sich, den **default**-Fall immer explizit (am Ende der **switch**-Anweisung) mit aufzuführen. Und auch dieser Fall sollte sicherheitshalber mit einem **break** verlassen werden.

Programm 5.11: Beispiel für die **switch**-Anweisung, bei der mehrere Fälle gemeinsam behandelt werden (*switch1.c*)

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

bool ispunc(char arg) {
    switch (arg) {
        case '.': /* gemeinsamer Fall ... */
        case ',': /* ... da break fehlt! */
        case ':':
        case ';':
        case '!':
            return true; /* fuer all die obigen Faelle! */
        default:
            return false;
    }
}

int main() {
    char ch;

    printf("Geben␣Sie␣ein␣Zeichen␣ein:␣");

    if (scanf("%c", &ch) > 0) {
        if (ispunc(ch)) {
            puts("Interpunktion");
        } else {
            puts("keine␣Interpunktion");
        }
```

```
    } else {
        puts("\nNichts␣eingegeben!");
    }
}
```

Kapitel 6

Ausdrücke

Eine *Ausdruck* besteht aus *Operatoren* und *Operanden*, wie zum Beispiel zu einer Addition der Operator + und die Summanden gehören.

6.1 Operanden

6.1.1 Links- und Rechts-Werte

⟨expression⟩	⟶	⟨comma-expression⟩
⟨comma-expression⟩	⟶	⟨assignment-expression⟩
	⟶	⟨comma-expression⟩ „,“
		⟨assignment-expression⟩
⟨assignment-expression⟩	⟶	⟨conditional-expression⟩
	⟶	⟨unary-expression⟩ ⟨assignment-op⟩
		⟨assignment-expression⟩

Zuweisungen gehören in C zu den Ausdrücken. Wegen der Asymmetrie einer Zuweisung, bei der links etwas Veränderbares stehen muss, während auf der rechten Seite ein beliebiger Ausdruck stehen kann, wird zwischen Links- und Rechts-Werten unterschieden bzw. zwischen *unary-expression* und *assignment-expression*.

In C ist ein Objekt eine Speicherfläche, deren Inhalt ausgelesen und verändert werden kann. Ein *Links-Wert* ist ein Ausdruck, der ein Objekt identifiziert. Ob ein Links-Wert zum Auslesen oder zum Verändern des Objekts dient, hängt vom Kontext ab. Steht ein Links-Wert auf der linken Seite einer Zuweisung, so wird das Objekt entsprechend verändert und der Wert des Zuweisungsausdrucks (*assignment-expression*) ergibt sich aus dem neuen Wert des Objekts.

Beispiele für Links-Werte sind Variablen, dereferenzierte Zeiger oder indizierte Arrays:

```
int i; int a[10]; int* p = &i;
struct int2 { int i, j; } s; struct int2* sp = &s;
i = 1; /* Links−Wert ist eine Variable */
*p = 2; /* Links−Wert ist ein derefenzierter Zeiger */
a[2] = 3; /* Links−Wert ist ein indiziertes Array */
```

```
s.i = 4; /* Links−Wert ist ein Feld eines Verbundstyps */
sp−>j = 5; /* Links−Wert ist ein Feld eines Verbundstyps */
```

Allerdings ist dabei zu beachten, dass in einigen Fällen auch syntaktisch zulässige Links-Werte nicht links von einer Zuweisung stehen dürfen wie beispielsweise im Falle von Konstantenvariablen:

```
const int i = 1;
i = 2; /* syntaktisch korrekt, jedoch semantisch nicht zulaessig */
```

Mehrfachzuweisungen sind zulässig:

```
int i, j, k;
i = j = k = 1; /* entspricht i = (j = (k = 1)) */
```

Eine Zuweisung liefert jedoch keinen Links-Wert:

```
int i, j, k;
(i += j) += k; /* ist nicht zulaessig */
```

Rechts-Werte können Links-Werte sein oder beliebige andere Ausdrücke, bei denen es nicht mehr darauf ankommt, dass ein konkretes Objekt damit verbunden sein muss. So ist beispielsweise das Ergebnis einer Addition nur noch ein Rechts-Wert.

6.1.2 Operanden im Einzelnen

⟨unary-expression⟩	⟶	⟨postfix-expression⟩
	⟶	⟨sizeof-expression⟩
	⟶	⟨unary-minus-expression⟩
	⟶	⟨unary-plus-expression⟩
	⟶	⟨logical-negation-expression⟩
	⟶	⟨bitwise-negation-expression⟩
	⟶	⟨address-expression⟩
	⟶	⟨indirection-expression⟩
	⟶	⟨preincrement-expression⟩
	⟶	⟨predecrement-expression⟩
⟨postfix-expression⟩	⟶	⟨primary-expression⟩
	⟶	⟨subscript-expression⟩
	⟶	⟨component-selection-expression⟩
	⟶	⟨function-call⟩
	⟶	⟨postincrement-expression⟩
	⟶	⟨postdecrement-expression⟩
	⟶	⟨compound-literal⟩
⟨primary-expression⟩	⟶	⟨identifier⟩
	⟶	⟨constant⟩
	⟶	⟨parenthesized-expression⟩
⟨subscript-expression⟩	⟶	⟨postfix-expression⟩ „[“ ⟨expression⟩ „]“
⟨component-selection-expression⟩	⟶	⟨direct-component-selection⟩
	⟶	⟨indirect-component-selection⟩

⟨direct-component-selection⟩	⟶	⟨postfix-expression⟩ „.“ ⟨identifier⟩
⟨indirect-component-selection⟩	⟶	⟨postfix-expression⟩ „->“ ⟨identifier⟩
⟨function-call⟩	⟶	⟨postfix-expression⟩
		„(“ [⟨expression-list⟩] „)“
⟨compound-literal⟩	⟶	„(“ ⟨type-name⟩ „)“
		„{“ ⟨initializer-list⟩ [„,“] „}“
⟨constant⟩	⟶	⟨integer-constant⟩
	⟶	⟨floating-constant⟩
	⟶	⟨character-constant⟩
	⟶	⟨string-constant⟩

Namen (*identifier*) können im Rahmen eines *primary-expression* sich auf eine Variablenvereinbarung beziehen, eine Funktion oder einen der Werte eines Aufzählungstyps:

- Variablennamen sind in der Regel zulässige Links-Werte. Es gibt jedoch eine wichtige Ausnahme: Der Name einer Array-Variablen steht für die Adresse des ersten Feldes. Da die Adresse konstant ist, kann ihr auch nichts zugewiesen werden. Folgende Konstruktion ist also nicht zulässig:

  ```
  int a[10], b[10];
  a = b; /* FALSCH: a ist kein Links−Wert */
  ```

- Interessanterweise geht dies aber bei Verbundtypen, bei denen der Variablenname jeweils das vollständige Objekt repräsentiert:

  ```
  struct int2 { int i, j; } a, b;
  a = (struct int2){1, 2}; /* Aggregrate zulaessig ab C99 */
  a = b; /* zulaessig */
  ```

  (Dies entspricht der *deep-copy*-Semantik in Modula-2 oder Oberon. Es weicht allerdings von der Semantik des *shallow-copy* in Java ab.)

- Konsequenterweise bedeutet dies, dass auch Arrays einander zugewiesen werden können, wenn diese in Verbundtypen eingepackt werden:

  ```
  struct int10 { int i[10]; } a, b;
  a = b; /* zulaessig */
  ```

- Funktionsnamen *ohne* Parameterliste, d.h. auch ohne Klammern, werden als konstante Zeiger auf die Funktion interpretiert:

  ```
  int (*writestring)(const char* s); /* Funktionszeiger */
  writestring = puts; /* Zeiger auf die Funktion kopieren ... */
  (*writestring)("Hallo␣zusammen!"); /* ... und aufrufen */
  ```

  (Funktionszeiger können in C dazu dienen, OO-Techniken rudimentär nachzubilden.)

Der Typ einer Konstanten (*constant*) ergibt sich direkt aus der lexikalischen Analyse:

```
int i = 1; /* integer−constant: Datentyp int */
double d = 1.23e−45 /* floating−constant: Datentyp double */
char c = 'a'; /* character−constant: Datentyp char */
char* s = "hello␣world"; /* string−constant: Datentyp char* */
```

Bei Bedarf kann der Datentyp auch feiner variiert werden. Auch gibt es Alternativen bei der Darstellung:

```
unsigned long ul = 4294967295UL; /* Datentyp unsigned long: 2^32 − 1 */
long long ll = 1099511627776LL; /* Datentyp long long */
int hex = 0x1ffee; /* Datentyp int in hexadezimaler Notation */
float f = 1.23e−45F; /* Datentyp float */
char newline = '\n'; /* Datentyp char: benanntes Sonderzeichen */
char bell = '\007'; /* Datentyp char in oktaler Darstellung */
char* s = "Hallo␣zusammen!\007\n";
```

Zeichenketten-Konstanten werden immer implizit durch ein Null-Byte abgeschlossen. Zu beachten ist, dass Zeichenketten-Konstanten nicht die gesamte Zeichenkette repräsentieren, sondern nur einen konstanten Zeiger auf das erste Zeichen. Eine Initialisierung eines Zeichen-Arrays mit einer Zeichenketten-Konstanten ist jedoch zulässig. Dies erlaubt auch eine automatisierte Festlegung der Array-Größe:

```
char greeting[] = "Hallo␣zusammen!\n";
```

Bei einem Funktionsaufruf ist die Reihenfolge, in der die aktuellen Parameter bewertet werden, nicht definiert. Jedoch darf davon ausgegangen werden, dass sich die Auswertungen der einzelnen Parameter nicht vermischen, d.h. die Auswertung beginnt mit einem der Parameter, diese wird vollständig ausgeführt, dann wird ein weiterer Parameter ausgewählt, dieser ebenfalls vollständig bewertet usw. bis die Auswertung aller Parameter abgeschlossen ist.

6.2 Operatoren

6.2.1 Übersicht

Die folgende Tabelle (die weitgehend [Harbison 2002] entnommen wurde) gibt einen Überblick der Operatoren, der Vorränge und der Bindungsreihenfolgen (Assoziativität):

Symbole	Art	Typ	Vorrang	Assoziativität
Namen und Konstanten	einfache Symbole	primär	16	—
a[*i*]	Indizierung	Postfix	16	von links
f()	Funktionsaufruf	Postfix	16	von links
.	Feldauswahl	Postfix	16	von links
−>	indirekte Feldauswahl	Postfix	16	von links
++ −−	Inkrement, Dekrement	Postfix	16	von links
(*Typ*) {*Init*}	Aggregrat	Postfix	16	von links
sizeof	Speichergröße	Präfix	15	von rechts
~	bitweise Negation	Präfix	15	von rechts
!	logische Negation	Präfix	15	von rechts
− +	arithmetische Vorzeichen	Präfix	15	von rechts
&	Adress-Operator	Präfix	15	von rechts
*	Dereferenzierung	Präfix	15	von rechts
(*Typ*)	Typ-Konvertierung	Präfix	14	von rechts
* / %	multiplikative Operatoren	dyadisch	13	von links
+ −	additive Operatoren	dyadisch	12	von links
<< >>	Bit-Verschiebungen	dyadisch	11	von links

Fortsetzung auf der nächsten Seite

Symbole	Art	Typ	Vorrang	Assoziativität
< > <= >=	Vergleiche	dyadisch	10	von links
== !=	Äquivalenz	dyadisch	9	von links
&	bitweises Und	dyadisch	8	von links
^	bitweises Exklusiv-Oder	dyadisch	7	von links
|	bitweises Inklusiv-Oder	dyadisch	6	von links
&&	logisches Und	dyadisch	5	von links
||	logisches Oder	dyadisch	4	von links
? :	Auswahl	triadisch	3	von rechts
= += −= *= /= %= <<= >>= &= ^= |=	Zuweisung	dyadisch	2	von rechts
,	sequentiell	dyadisch	1	von links

6.2.2 Monadische Postfix-Operatoren

Monadische Operatoren, auch unäre Operatoren genannt, stehen entweder vor oder hinter dem Operanden. Letztere werden Postfix-Operatoren genannt. Prominentes Beispiel hierfür sind die Postfix-Inkrement- und Dekrement-Operatoren, die als Operanden einen Links-Wert erwarten, diesen um eins erhöhen oder verringern. Sie liefern jedoch als Ausdrucks-Wert den alten Links-Wert zurück:

```
int i = 0; int j;
j = i++; /* i = 1, j = 0 */
```

6.2.3 Monadische Präfix-Operatoren

Die Mehrheit der monadischen Operatoren geht dem zugehörigen Operanden voran:

⟨sizeof-expression⟩	⟶	**sizeof** „(“ ⟨type-name⟩ „)“
	⟶	**sizeof** ⟨unary-expression⟩
⟨unary-minus-expression⟩	⟶	„-“ ⟨cast-expression⟩
⟨unary-plus-expression⟩	⟶	„+“ ⟨cast-expression⟩
⟨logical-negation-expression⟩	⟶	„!“ ⟨cast-expression⟩
⟨bitwise-negation-expression⟩	⟶	„~“ ⟨cast-expression⟩
⟨address-expression⟩	⟶	„&“ ⟨cast-expression⟩
⟨indirection-expression⟩	⟶	„*“ ⟨cast-expression⟩
⟨preincrement-expression⟩	⟶	„++“ ⟨unary-expression⟩
⟨predecrement-expression⟩	⟶	„–“ ⟨unary-expression⟩
⟨cast-expression⟩	⟶	⟨unary-expression⟩
	⟶	„(“ ⟨type-name⟩ „)“ ⟨unary-expression⟩

Wie sich der Grammatik entnehmen lässt, gibt es die Operatoren ++ und −− auch als Präfix-Operatoren. Im Unterschied zu den gleichnamigen Postfix-Operatoren liefern sie als Ausdruck den Links-Wert nach der Inkrementierung bzw. Dekrementierung zurück:

```
int i = 0; int j;
j = ++i; /* i = 1, j = 1 */
```

Anders als in anderen Programmiersprachen mit expliziten Zeigern ist der Dereferenzierungs-Operator $*$ in C ein Präfix-Operator. Das hat zur Konsequenz, dass Klammern unvermeidlich sein können, wenn Post- und Präfix-Operatoren gemischt verwendet werden:

```
struct int2 { int i, j; } a; struct int2* p;
p = &a;
*p.i = 1; /* FALSCH: p.i ist kein Zeiger */
(*.p).i = 1; /* OK: Zuerst den Zeiger dereferenzieren, dann selektieren */
```

Da $(*.p).i$ so umständlich aussieht, gibt es den Operator $->$, der es erlaubt, auf die Klammern zu verzichten: $p->i$. Dem Operator $->$ muss immer ein Feldname folgen.

Der Adressoperator & liefert die Adresse eines Links-Werts. Der Datentyp ist dann jeweils der zugehörige Zeigertyp. Aus **int** wird so **int***. Aus **int*** würde, falls der Operator ein weiteres Mal zum Einsatz kommt, **int****. Beispiele:

```
int i; int* ip; int** ipp;
ipp = &ip; *ipp = &i; /* ip = &i */
**ipp = 2; /* i = 2 */
```

Der Name eines Feldes steht bereits für die Anfangsadresse des Felds:

```
int a[10]; int* ip;
ip = &a; /* FALSCH: a ist bereits eine konstante Adresse */
ip = a; /* OK: ip zeigt jetzt auf a[0] */
ip = &a[0]; /* OK: umstaendlichere Alternative */
ip = &a[3]; /* OK: ip zeigt auf a[3] */
```

Das logische Komplement liefert *true*, falls der Operand 0 oder *false* ist, ansonsten *false*. Wie bei allen anderen logischen Operatoren werden beliebige skalare Operanden akzeptiert und das Resultat ist sowohl zu den ganzzahligen Typen einschließlich *bool* kompatibel.

Der **sizeof**-Operator liefert die benötigte Speicherfläche für den angegebenen Typ. Der Typ wird entweder explizit angegeben (und muss dann in Klammern stehen) oder implizit durch den Typ des folgenden Ausdrucks. Im letzteren Fall wird der Ausdruck nicht bewertet, sondern nur dessen Typ vom Übersetzer bestimmt.

Üblicherweise wird die benötigte Speicherfläche in Bytes gemessen, aber das muss nicht überall der Fall sein. Zumindest gab es (mittlerweile historische) Architekturen mit so „exotischen" Konstellationen wie 5-Bit-Bytes und 36-Bit-Wörtern, für die auch C-Übersetzer entwickelt wurden, so dass C nicht festlegt, in welchen Einheiten Speicher gemessen wird. Deswegen sollten auch nicht die üblichen ganzzahligen Datentypen wie etwa **int** oder **unsigned long** dafür verwendet werden, sondern *size_t* aus <*stddef.h*>.

Folgendes Beispiel illustriert die Verwendung einiger unärer Operatoren:

Programm 6.1: Verwendung unärer Operatoren (*unaer.c*)

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

int main() {
    int i = 3; /* vorinitialisierte Integervar. */
    int* p; /* Zeiger auf eine Integervar. */
    int a[50]; /* Integer-Array */
    size_t sizeof_a; /* Groesse des Arrays */
    size_t sizeof_p; /* Groesse des Zeigers p */

    p = &i; /* Adresse von i wird p zugewiesen */
    printf("i=%d, p=%p (Adresse), *p=%d (Wert)\n", i, p, *p);
```

```
    printf("i++=%d,␣", i++); /* nachher inkrementieren */
    printf("i=%d\n", i);

    printf("++i=%d,␣", ++i); /* vorher inkrementieren */
    printf("i=%d\n", i);

    printf("!0=%d,␣!1=%d,␣!2=%d\n", !0, !1, !2); /* log. Negation */

    printf("i=%08x,␣~i=%08x\n", i, ~i); /* bitweises (Einer−)Komplement */

    p = a; sizeof_a = sizeof(a); sizeof_p = sizeof(p);
    /* Format %zd: size_t dezimal */
    printf("sizeof(a)=%zd,␣sizeof(p)=%zd\n", sizeof_a, sizeof_p);
}
```

```
doolin$ gcc -Wall -std=c99 unaer.c
doolin$ a.out
i=3, p=ffbff65c (Adresse), *p=3 (Wert)
i++=3, i=4
++i=5, i=5
!0=1, !1=0, !2=0
i=00000005, ~i=fffffffa
sizeof(a)=200, sizeof(p)=4
doolin$
```

Hinweis: Bei älteren C-Bibliotheken, die noch nicht den C99-Standard vollständig unterstützen, kann es sein, dass *%zd* noch nicht funktioniert. Dann sollte notfalls *%lu* mit einer zugehörigen Typ-Konvertierung verwendet werden. Beispiel:

```
printf("sizeof(int)␣=␣%lu\n", (unsigned long) sizeof(int));
```

6.2.4 Dyadische Operatoren

Dyadische Operatoren, auch binäre Operatoren genannt, werden in C durchweg in der Infix-Notation verwendet:

⟨logical-or-expression⟩	⟶	⟨logical-and-expression⟩
	⟶	⟨logical-or-expression⟩ „||“ ⟨logical-and-expression⟩
⟨logical-and-expression⟩	⟶	⟨bitwise-or-expression⟩
	⟶	⟨logical-and-expression⟩ „&&“ ⟨bitwise-or-expression⟩
⟨bitwise-or-expression⟩	⟶	⟨bitwise-xor-expression⟩
	⟶	⟨bitwise-or-expression⟩ „|“ ⟨bitwise-xor-expression⟩
⟨bitwise-xor-expression⟩	⟶	⟨bitwise-and-expression⟩
	⟶	⟨bitwise-xor-expression⟩ „|“ ⟨bitwise-and-expression⟩

⟨bitwise-and-expression⟩	⟶	⟨equality-expression⟩
	⟶	⟨bitwise-and-expression⟩ „^“ ⟨equality-expression⟩
⟨equality-expression⟩	⟶	⟨relational-expression⟩
	⟶	⟨equality-expression⟩ ⟨equality-op⟩ ⟨relational-expression⟩
⟨equality-op⟩	⟶	„==“ | „!=“
⟨relational-expression⟩	⟶	⟨shift-expression⟩
	⟶	⟨relational-expression⟩ ⟨relational-op⟩ ⟨shift-expression⟩
⟨relational-op⟩	⟶	„<“ | „<=“ | „>“ | „>=“
⟨shift-expression⟩	⟶	⟨additive-expression⟩
	⟶	⟨shift-expression⟩ ⟨shift-op⟩ ⟨additive-expression⟩
⟨shift-op⟩	⟶	„<<“ | „>>“
⟨additive-expression⟩	⟶	⟨multiplicative-expression⟩
	⟶	⟨additive-expression⟩ ⟨add-op⟩ ⟨multiplicative-expression⟩
⟨add-op⟩	⟶	„+“ | „-“
⟨multiplicative-expression⟩	⟶	⟨cast-expression⟩
	⟶	⟨multiplicative-expression⟩ ⟨mult-op⟩ ⟨cast-expression⟩
⟨mult-op⟩	⟶	„*“ | „/“ | „%“

Seit *C89* wird zugesichert, dass Ausdrücke in der genannten Reihenfolge ausgewertet werden. Das bedeutet, dass die scheinbare Kommutativität einiger Operatoren wie + oder $*$ durch den Übersetzer nicht für Optimierungszwecke ausgenutzt werden darf, wenn die Reihenfolge in Bezug auf Überläufe oder die Genauigkeit (bei Gleitkommazahlen!) relevant sein könnte. Eine Änderung der Reihenfolge ist somit für den Übersetzer nur dann zulässig, wenn dies auf keinen Fall das Resultat ändert. Das bedeutet jedoch nicht, dass Seiteneffekte in einer garantierten Reihenfolge erfolgen. Diese bleiben undefiniert. Beispiel:

```
int i = 1, j;
j = (i += 10) % ++i;
/* Welchen Wert hat j? 0 oder 11? */
```

Bei den logischen Operatoren wird zunächst der linke Operand ausgewertet und dann nur für den Fall, dass daraus noch nicht das Resultat folgt, der zweite Operand bewertet. Die einzelnen Operanden müssen jeweils einen beliebigen skalaren Typ haben, der mit 0 vergleichbar ist. Das Resultat ist 0 (*false*) oder 1 (*true*) entsprechend dem Wertebereich von *bool*.

Die Operanden eines bitweisen Operators müssen ganzzahlig sein und werden zunächst aneinander angeglichen. Danach werden die Operanden als (inzwischen gleich lange) Bitfelder betrachtet, bei denen die Operation dann paarweise für alle Bits durchgeführt wird (das erste Bit des ersten Operanden mit dem ersten Bit des zweiten Operanden ergibt das erste resultierende Bit usw.).

Die Schiebe-Operatoren << und >> multiplizieren bzw. dividieren ganzzahlige Werte mit Zweier-Potenzen, d.h. $i << n$ entspricht $i * 2^n$ und $i >> n$ entspricht der ganzzahligen Division $i/2^n$. Die Schiebe-Operationen sind nicht definiert, wenn n negativ oder größer oder gleich der Zahl der zur Verfügung stehenden Bits ist. Ferner ist es der Implementierung überlassen, wie $i >> n$ umgesetzt wird, wenn i negativ ist.

Die bitweisen Operatoren können in Kombination mit den Schiebe-Operationen dazu genutzt werden, um Mengen als Bitfelder darzustellen, indem die einzelnen Elemente i durch die Werte 2^i dargestellt werden:

Mengen-notation	in C
$A \cup B$	$A \mid B$
$A \cap B$	$A \,\&\, B$
$A \setminus B$	$A \,\&\, \sim B$
$\{i\}$	$1 << i$
$i \in A$	$(1 << i) \& A$

Beim Divisions-Operator / werden zunächst die Operanden aneinander angeglichen und dann entsprechend des größten gemeinsamen Typs durchgeführt. D.h. bei ganzzahligen Operanden wird eine ganzzahlige Division durchgeführt (mit einer Rundung in Richtung zur 0, falls das Resultat nicht exakt ist) und bei Fließkommazahlen bzw. gemischten Operanden kommt es zu einer Fließkomma-Division.

Folgendes Beispiel illustriert die Verwendung binärer Operatoren:

Programm 6.2: Verwendung binärer Operationen (*binaer.c*)

```
#include <stdio.h>

/* Ausgabe einer nicht−negativen ganzen Zahl in Binaerdarstellung;
   die Zahl der ausgegebenen Zeichen wird zurueckgegeben
*/
unsigned int print_binary(unsigned int n, unsigned int minwidth) {
   unsigned int rn; /* n in bit−maessig umgedrehter Reihenfolge */
   unsigned int printed = 0; /* Anzahl der ausgegebenen Zeichen */
   unsigned int count = 0; /* Anzahl der relevanten Ziffern */

   /* Reihenfolge der Bits umdrehen */
   for (rn = 0; n != 0; n >>= 1) {
      rn = (rn << 1) | (n & 1);
      ++count;
   }
   /* Ausgabe der fuehrenden Nullen, falls notwendig */
   if (minwidth > count) {
      minwidth −= count;
      while (minwidth > 0) {
         putchar('0'); ++printed; −−minwidth;
      }
   }
   /* Ausgabe der relevanten Ziffern */
   for (; count > 0; rn >>= 1) {
      if (rn & 1) {
         putchar('1');
      } else {
         putchar('0');
```

```
        }
        ++printed; --count;
    }
    return printed;
}

int main() {
    unsigned int i = 17, j = 3;

    printf("␣␣␣␣␣i␣=␣"); print_binary(i, 5); puts("");
    printf("␣␣␣␣␣j␣=␣"); print_binary(j, 5); puts("");

    /* Bitweises UND, ODER und XOR (= exklusives ODER) */
    printf("␣␣␣i&j␣=␣"); print_binary(i & j, 5);
    printf(",␣␣i|j␣=␣"); print_binary(i | j, 5);
    printf(",␣␣i^j␣=␣"); print_binary(i ^ j, 5);
    puts("");

    /* Bit-Verschiebungen (nicht zyklisch!) */
    printf("␣␣i>>1␣=␣"); print_binary(i >> 1, 5);
    printf(",␣i<<1␣=␣"); print_binary(i << 1, 5);
    puts("");

    /* Bit-Verschiebungen bei negativen Zahlen (undefiniert!)
        (bei der Konvertierung von signed zu unsigned garantiert
        C hier das Zweier-Komplement bei negativen Zahlen)
    */
    printf("␣␣␣␣-1␣=␣");
    unsigned int bits = print_binary(-1, 0); puts("");
    printf("␣-1>>1␣=␣"); print_binary(-1 >> 1, bits); puts("");
    printf("␣-1<<1␣=␣"); print_binary(-1 << 1, bits); puts("");
}
```

```
dublin$ gcc -Wall -std=c99 binaer.c
dublin$ a.out
     i = 10001
     j = 00011
   i&j = 00001,  i|j = 10011,  i^j = 10010
  i>>1 = 01000, i<<1 = 100010
    -1 = 11111111111111111111111111111111
 -1>>1 = 11111111111111111111111111111111
 -1<<1 = 11111111111111111111111111111110
dublin$
```

Anmerkung: Wenn der Schiebe-Operator >> für negative ganze Zahlen verwendet wird, bleibt es der jeweiligen Implementierung überlassen, ob Nullen oder Einsen an der höchstwertigen Stelle „nachrücken“. Wenn (wie fast durchgängig üblich) das Zweier-Komplement zur Darstellung negativer Zahlen verwendet wird, bedeutet dies, dass der Wert des Ausdrucks $-1 >> 1$ entweder -1 oder 2^{n-1} ergeben kann, wobei n für die Zahl der verwendeten Bits steht.

Der Modulo-Operator % liefert bei ganzzahligen Operanden den Rest der entsprechenden ganzzahligen Division, so dass $(a/b)*b + a\%b == a$ gilt, falls a/b repräsentierbar ist.

Ferner gilt $0 \leq (a\%b) < b$ für $a >= 0$ und $b > 0$. Abweichende Definitionen des Modulo-Operators existieren jedoch, falls a oder b negativ werden:

- **T-Definition:** a/b == (**int**)((**double**)a / (**double**)b)
  d.h. es wird als Resultat der ganzzahligen Division die größte ganzzahlige Zahl genommen, deren Betrag kleiner oder gleich als $\frac{a}{b}$ ist und deren Differenz weniger als 1 beträgt (Rundung in Richtung zur 0).

- **F-Definition:** a/b == (**int**)*floor*((**double**)a / (**double**)b)
  d.h. es wird die größtmögliche ganze Zahl genommen, die kleiner oder gleich $\frac{a}{b}$ ist. (Rundung in Richtung $-\infty$). Hier gilt:
  $0 <= a\%b$ && $a\%b < b$ || $b < a\%b$ && $a\%b <= 0$

- **Nach Euklid:** Diese stimmt mit der F-Definition überein, falls y positiv ist. Es gilt jedoch
  $a/(-b)== -(a/b)$&& $a\%(-b)== a\%b$ und daraus folgt:
  $0 <= a\%b$ && $a\%b < abs(b)$

Praktisch alle gängigen Prozessor-Architekturen unterstützen die T-Definition, die aus diesem Grunde auch von vielen Programmiersprachen übernommen wurde wie beispielsweise C, C++, Modula-2 oder Java. (Im Falle von C blieb die konkrete Semantik jedoch vor C99 noch undefiniert.) Es gibt jedoch gute Gründe, die anderen Definitionen zu verwenden. So unterstützten beispielsweise Oberon und Scheme die F-Definition. Folgendes Beispiel demonstriert die Unterschiede an Beispielen:

Programm 6.3: Der Modulo-Operator (*modulo.c*)

```
#include <stdio.h>

/* nach der T−Definition */
int tmod(int a, int b) {
    return a % b;
}
int tdiv(int a, int b) {
    return a / b;
}

/* nach der F−Definition */
int fmod(int a, int b) {
    int r = a % b;
    if ((a >= 0) == (b >= 0) || r == 0) {
        return r;
    } else {
        return r+b;
    }
}
int fdiv(int a, int b) {
    int r = a % b;
    int q = a / b;
    if ((a >= 0) == (b >= 0) || r == 0) {
        return q;
    } else {
        return q−1;
    }
}
```

```
/* nach Euklid */
int emod(int a, int b) {
    if (b > 0) {
        return fmod(a, b);
    } else {
        return fmod(a, −b);
    }
}
int ediv(int a, int b) {
    if (b > 0) {
        return fdiv(a, b);
    } else {
        return −fdiv(a, −b);
    }
}

int main() {
    int a, b;
    printf("a␣=␣"); scanf("%d", &a);
    printf("b␣=␣"); scanf("%d", &b);
    printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣T−Def.␣F−Def.␣Euklid\n");
    printf("%4d/%4d␣=␣%6d␣%6d␣%6d\n",
        a, b, tdiv(a, b), fdiv(a, b), ediv(a, b));
    printf("%4d%%%4d␣=␣%6d␣%6d␣%6d\n",
        a, b, tmod(a, b), fmod(a, b), emod(a, b));
}
```

```
dublin$ gcc -Wall -std=c99 modulo.c
dublin$ a.out
a = 13
b = -4
              T-Def. F-Def. Euklid
  13/  -4 =     -3     -4     -3
  13%  -4 =      1     -3      1
dublin$ a.out
a = -13
b = -4
              T-Def. F-Def. Euklid
 -13/  -4 =      3      3      4
 -13%  -4 =     -1     -1      3
dublin$
```

6.2.5 Auswahl-Operator

$$
\begin{aligned}
\langle\text{conditional-expression}\rangle &\longrightarrow \langle\text{logical-or-expression}\rangle \\
&\longrightarrow \langle\text{logical-or-expression}\rangle \text{ „?“ } \langle\text{expression}\rangle \\
&\qquad \text{„:“ } \langle\text{conditional-expression}\rangle
\end{aligned}
$$

Eine *Auswahl* bzw. ein bedingter Ausdruck besteht aus einer Bedingung, der zwei Ausdrücke folgen. Die Bedingung wird auf die bekannte Weise bewertet: Ist der Wert ungleich Null (also *true*), so wird der erste Ausdruck als Ergebnis genommen, ist er gleich Null (also *false*), so der zweite.

Ein Beispiel ist die folgende Auswahl:

```
i = a > 7? x + y: y − 2;
```

Im Prinzip ist der Auswahl-Operator eine Kurzschreibweise für eine **if**-Anweisung. Man kann obiges Beispiel ausführlicher auch wie folgt umschreiben:

```
if (a > 7) {
    i = x + y;
} else {
    i = y − 2;
}
```

6.2.6 Komma-Operator

⟨comma-expression⟩	⟶	⟨assignment-expression⟩
	⟶	⟨comma-expression⟩ „,“ ⟨assignment-expression⟩

Durch die Verwendung des *Komma-Operators* lassen sich mehrere Ausdrücke schreiben, wo eigentlich nur ein Ausdruck erlaubt ist. Alle durch Komma voneinander getrennten Teilausdrücke werden bewertet. Der Wert des Gesamtausdrucks ist gleich dem Wert des letzten Teilausdrucks; siehe folgendes Beispiel:

Programm 6.4: Verwendung des Komma-Operators (*komma.c*)

```
#include <stdio.h>

int main() {
    int a, b, c;

    /* Wert eines Komma−Ausdrucks ist gleich
        dem Wert des LETZTEN Teil−Ausdrucks */
    c = (a = 1, b = 2);
    printf("a=%d,␣b=%d,␣c=%d\n", a, b, c);

    puts("−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−");

    /* sinnvollere Verwendung innerhalb einer for−Schleife
        und zum Sparen von Anweisungsbloecken ({...}) */
    for (a = 0, b = 0, c = 0; a <= 10; a++) {
        b −= 1, c += 2, printf("a=%2d,␣b=%3d,␣c=%2d\n", a, b, c);
    }
}
```

```
dublin$ gcc -Wall -std=c99 komma.c
dublin$ a.out
a=1, b=2, c=2
-------------------------
a= 0, b= -1, c= 2
a= 1, b= -2, c= 4
a= 2, b= -3, c= 6
a= 3, b= -4, c= 8
a= 4, b= -5, c=10
a= 5, b= -6, c=12
a= 6, b= -7, c=14
a= 7, b= -8, c=16
a= 8, b= -9, c=18
a= 9, b=-10, c=20
a=10, b=-11, c=22
dublin$
```

6.2.7 Zuweisungen

⟨assignment-expression⟩	⟶	⟨conditional-expression⟩
	⟶	⟨unary-expression⟩ ⟨assignment-op⟩ ⟨assignment-expression⟩
⟨assignment-op⟩	⟶	„=“ | „*=“ | „/=“ | „%=“ | „+=“ | „-=“ | „«=“ | „»=“ | „&=“ | „^=“ | „|=“

Die Zuweisungs-Operatoren liefern (wie die anderen Operatoren auch) einen Wert und gleichzeitig mit der Zuweisung auch einen Nebeneffekt. Für alle Zuweisungs-Operatoren gilt dabei, dass der zugewiesene Wert gleich dem resultierenden Wert ist.

Die doppelte Funktionalität der Zuweisungs-Operatoren ermöglicht eine Mehrfachzuweisung folgender Form:

int x, y;
$x = y = 1$;

Entsprechend der Assoziativität aller Zuweisungs-Operatoren von rechts nach links wird zuerst $y = 1$ bewertet, d.h. y erhält den Wert 1, und dann wird der resultierende Wert von 1 an x zugewiesen. Dieser Wert ist auch gleichzeitig wiederum der Wert des gesamten Ausdrucks, der hier ignoriert wird.

Neben der einfachen Zuweisung gibt es eine Reihe von Zuweisungs-Operatoren, die mit einem der regulären dyadischen Operatoren verbunden sind. So kann die Zuweisung $a = a$ **op** b generell zu a **op**= b verkürzt werden. Dies hat den Vorteil, dass a nur einmal bewertet werden muss und entsprechende Optimierungen dem Übersetzer erleichtert werden.

Programm 6.5: Zuweisungen (*zuweisung.c*)

```
#include <stdio.h>

int main() {
    int a, b;
```

```
    /* Kurzform fuer b = 13; a = b; */
    a = b = 13;
    printf("a=%d,␣b=%d\n", a, b);

    /* a += 2 als Kurzform fuer a = a + 2
       ACHTUNG: Die Reihenfolge der Bewertung ist nicht definiert */
    printf("a=%d,␣(a+=2)=%d\n", a, a += 2);
}
```

```
zuweisung.c: In function 'main':
zuweisung.c:12: warning: operation on 'a' may be undefined
dublin$ a.out
a=13, b=13
a=15, (a+=2)=15
dublin$
```

Hinweis: Wie bereits erwähnt, ist die Reihenfolge der Parameterbewertungen nicht definiert. Deswegen liefern hier aktuellere C-Übersetzer auch eine entsprechende Warnung. Konkret wäre hier statt der Ausgabe „a=15, (a+=2)=15“ auch die Ausgabe „a=13, (a+=2)=15“ möglich gewesen.

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.

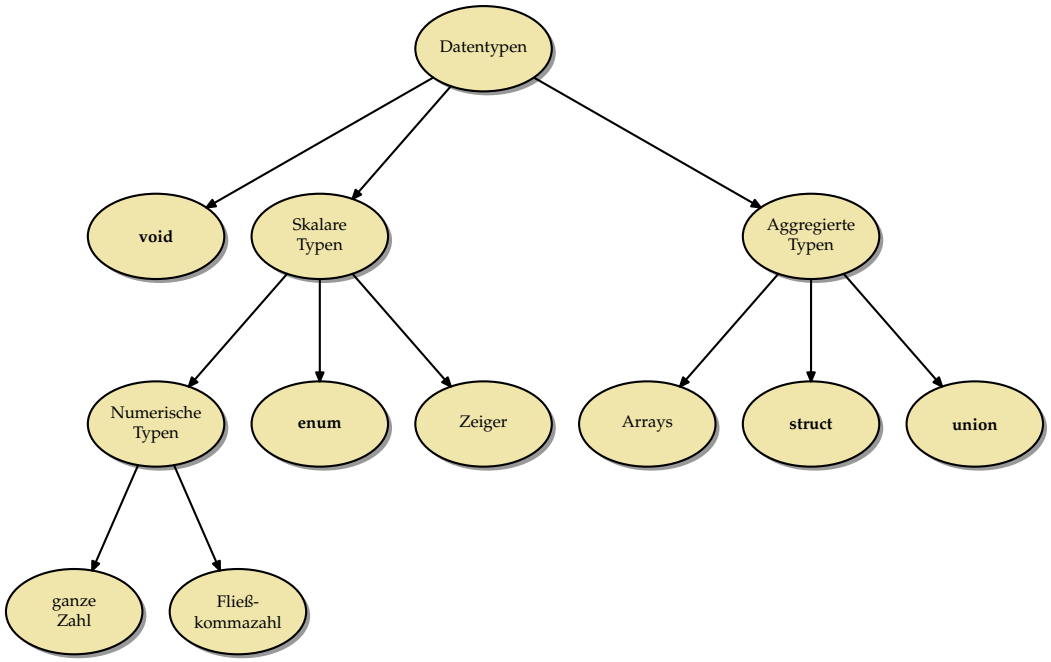


Abbildung 7.1: Datentypen – Eine Übersicht

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:

$$
\begin{array}{lcl}
\langle\text{type-specifier}\rangle & \longrightarrow & \langle\text{void-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{integer-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{floating-point-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{enumeration-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{structure-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{union-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{typedef-name}\rangle
\end{array}
$$

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

$$
\begin{array}{rcl}
\langle\text{integer-type-specifier}\rangle & \longrightarrow & \langle\text{signed-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{unsigned-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{character-type-specifier}\rangle \\
 & \longrightarrow & \langle\text{bool-type-specifier}\rangle \\
\langle\text{signed-type-specifier}\rangle & \longrightarrow & [\ \textbf{signed}\ ]\ \textbf{short}\ [\ \textbf{int}\ ] \\
 & \longrightarrow & [\ \textbf{signed}\ ]\ \textbf{int} \\
 & \longrightarrow & [\ \textbf{signed}\ ]\ \textbf{long}\ [\ \textbf{int}\ ] \\
 & \longrightarrow & [\ \textbf{signed}\ ]\ \textbf{long long}\ [\ \textbf{int}\ ] \\
\langle\text{unsigned-type-specifier}\rangle & \longrightarrow & \textbf{unsigned short}\ [\ \textbf{int}\ ] \\
 & \longrightarrow & \textbf{unsigned}\ [\ \textbf{int}\ ] \\
 & \longrightarrow & \textbf{unsigned long}\ [\ \textbf{int}\ ] \\
 & \longrightarrow & \textbf{unsigned long long}\ [\ \textbf{int}\ ] \\
\langle\text{character-type-specifier}\rangle & \longrightarrow & \textbf{char} \\
 & \longrightarrow & \textbf{signed char} \\
 & \longrightarrow & \textbf{unsigned char} \\
\langle\text{bool-type-specifier}\rangle & \longrightarrow & \textbf{\_Bool}
\end{array}
$$

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**:

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

  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**:

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

  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*)

```
#include <stdio.h>

const int letters = 'z' − 'a' + 1;
const int rotate = 13;
int main() {
    int ch;
    while ((ch = getchar()) != EOF) {
        if (ch >= 'a' && ch <= 'z') {
            ch = 'a' + (ch − 'a' + rotate) % letters;
        } else if (ch >= 'A' && ch <= 'Z') {
            ch = 'A' + (ch − 'A' + rotate) % letters;
        }
        putchar(ch);
    }
}
```

```
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*)

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

double triangle_area1(double a, double b, double c) {
    double s = (a + b + c) / 2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

#define SWAP(a,b) {int tmp; tmp = a; a = b; b = tmp;}
double triangle_area2(double a, double b, double c) {
    /* sort a, b, and c in descending order, applying a bubble-sort */
    if (a < b) SWAP(a, b); if (b < c) SWAP(b, c); if (a < b) SWAP(a, b);
    /* formula by W. Kahan */
    return sqrt((a + (b + c)) * (c - (a - b)) *
                (c + (a - b)) * (a + (b - c)))/4;
}
```

```
int main() {
    double a, b, c;
    printf("triangle side lenghts a b c: ");
    if (scanf("%lf %lf %lf", &a, &b, &c) != 3) {
        printf("Unable to read three floats!\n");
        return 1;
    }
    double a1 = triangle_area1(a, b, c);
    double a2 = triangle_area2(a, b, c);
    printf("Formula #1 delivers %.16lf\n", a1);
    printf("Formula #2 delivers %.16lf\n", a2);
    printf("Difference: %lg\n", fabs(a1−a2));
    return 0;
}
```

```
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*)

```
#include <stdio.h>
int main() {
    double x, y;
    printf("x y = ");
    if (scanf("%lf %lf", &x, &y) != 2) {
        printf("Unable to read two floats!\n");
        return 1;
    }
    if ((x/y)*y == x) {
        printf("equal\n");
    } else {
        printf("not equal\n");
    }
    return 0;
}
```

```
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*)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>

enum days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
char* dayname[] = {
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
};

int main() {
    enum days day;
    for (day = Monday; day <= Sunday; ++day) {
        printf("Day␣%d␣=␣%s\n", day, dayname[day]);
    }
    /* seed the pseudo−random generator */
    unsigned int seed = time(0); srand(seed);
    /* select and print a pseudo−random day */
    enum days favorite_day = rand() % 7;
    printf("My␣favorite␣day:␣%s\n", dayname[favorite_day]);
}
```

```
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*)

```
#include <stdio.h>

int main() {
    int i = 13;
    int* p = &i; /* Zeiger p zeigt auf i; &i = Adresse von i */

    printf("i=%d, p=%p (Adresse), *p=%d (Wert)\n", i, p, *p);

    ++i;
    printf("i=%d, *p=%d\n", i, *p);

    ++*p; /* *p ist ein Links-Wert */
    printf("i=%d, *p=%d\n", i, *p);
}
```

```
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.

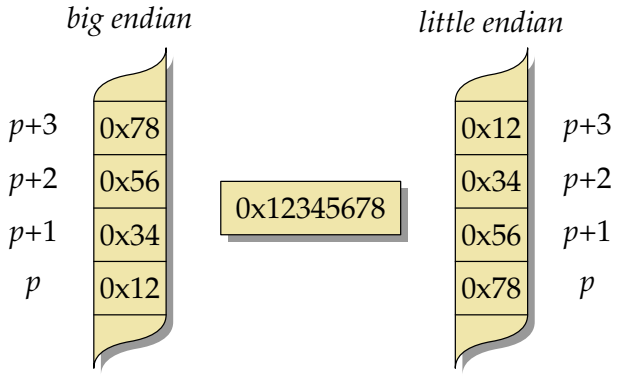


Abbildung 7.2: *big* vs. *little endian*

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

```
#include <stdio.h>

int main() {
    unsigned int value = 0x12345678;
    unsigned char* p = (unsigned char*) &value;

    for (int i = 0; i < sizeof(unsigned int); ++i) {
        printf("p+%d␣−−>␣0x%02hhx\n", i, *(p+i));
    }
}
```

```
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.

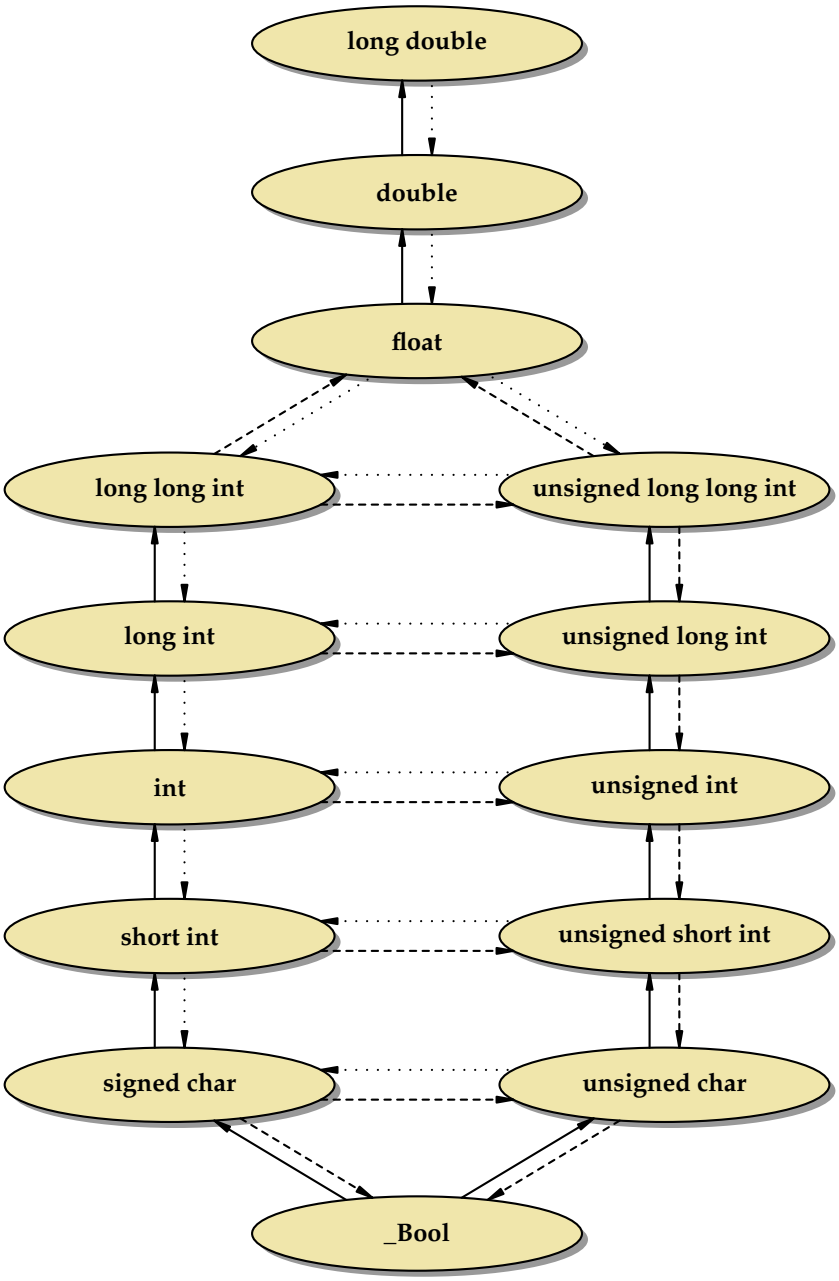


Abbildung 7.3: Konvertierungen zwischen numerischen Datentypen

- 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*)

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>

void print_bool(bool boolval) {
    if (boolval) {
        puts("true");
    } else {
        puts("false");
    }
}

int main() {
    /* unsigned int --> unsigned short int */
    unsigned int i1 = UINT_MAX; unsigned short int i2 = i1;
    printf("%x␣->␣%hx\n", i1, i2);
    /* int op unsigned int --> unsigned int */
    int i3 = -1; unsigned int i4 = UINT_MAX;
    printf("%d␣==␣%u:␣", i3, i4); print_bool(i3 == i4);
    /* int --> float --> int */
    int i5 = 123456789; float f = i5; int i6 = f;
    printf("%d␣->␣%g␣->␣%d\n", i5, f, i6);
}
```

```
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**:

$$
\begin{array}{rcl}
\langle\text{declaration-specifiers}\rangle & \longrightarrow & \langle\text{storage-class-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
 & \longrightarrow & \langle\text{type-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
 & \longrightarrow & \langle\text{type-qualifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
 & \longrightarrow & \langle\text{function-specifier}\rangle\ [\ \langle\text{declaration-specifiers}\rangle\ ] \\
\langle\text{type-qualifier}\rangle & \longrightarrow & \textbf{const} \\
 & \longrightarrow & \textbf{volatile} \\
 & \longrightarrow & \textbf{restrict}
\end{array}
$$

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*)

```
#include <stdio.h>

int main() {
    const int i = 1;

    i++; /* das geht doch nicht, oder?! */
    printf("i=%d\n", i);
}
```

```
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*)

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

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    const size_t SIZE = sizeof(a) / sizeof(a[0]); /* Groesse des Arrays bestimmen */
    int* p = a; /* kann statt a verwendet werden */

    /* aber: a weiss noch die Gesamtgroesse, p nicht */
    printf("SIZE=%zd, sizeof(a)=%zd, sizeof(p)=%zd\n",
        SIZE, sizeof(a), sizeof(p));

    for (int i = 0; i < SIZE; ++i) {
```

```
        *(a + i) = i+1; /* gleichbedeutend mit a[i] = i+1 */
    }

    /* Elemente von a aufsummieren */
    int sum = 0;
    for (int i = 0; i < SIZE; i++) {
        sum += p[i]; /* gleichbedeutend mit ... = a[i]; */
    }
    printf("Summe:␣%d\n", sum);
}
```

```
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*)

```
#include <stdio.h>

const int SIZE = 10;

int main() {
    int x = 1000, y = 2000;
    int a[SIZE];

    /* wir wollen mal sehen, wo denn das alles im Speicher liegt */
    printf("&x␣=␣%p,␣&y␣=␣%p\n&(a[0])␣=␣%p,␣&(a[SIZE+1])␣=␣%p\n",
        &x, &y, &(a[0]), &(a[SIZE+1]));

    /* Initialisierung von a */
    for (int i = 0; i < SIZE; i++) {
        a[i] = i * i;
    }

    /* Elemente von a aufsummieren */
    int sum = 0;
    for (int i = 0; i < SIZE; i++) {
        sum += a[i];
    }
    printf("Summe:␣%d\n", sum);

    /* typischer Fehler: ein Element zuviel */
    sum = 0;
    for (int i = 0; i <= SIZE; i++) {
```

```
        sum += a[i];
    }
    printf("Summe:␣%d\n", sum);

    /* VORSICHT – Lebensgefahr! */
    x = y = 0;
    printf("x=%d,␣y=%d\n", x, y);
    a[SIZE+1]++;
    printf("x=%d,␣y=%d\n", x, y);

    a[SIZE+10000]++; /* => segmentation fault */
}
```

```
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*)

```
#include <stdio.h>

const int SIZE = 10;

/* Array wird veraendert, naemlich mit
   0, 1, 2, 3, ... initialisiert! */
void init(int a[], int length) {
    for (int i = 0; i < length; i++) {
        a[i] = i;
    }
}

int summe1(int a[], int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += a[i];
    }
    return sum;
}
```

```
int summe2(int* a, int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += *(a+i); /* aequivalent zu ... += a[i]; */
    }
    return sum;
}

int main() {
    int array[SIZE];

    init(array, SIZE);

    printf("Summe:␣%d\n", summe1(array, SIZE));
    printf("Summe:␣%d\n", summe2(array, SIZE));
}
```

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 0*x1000* 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 $*(*(matrix + 1)+ 2)$.

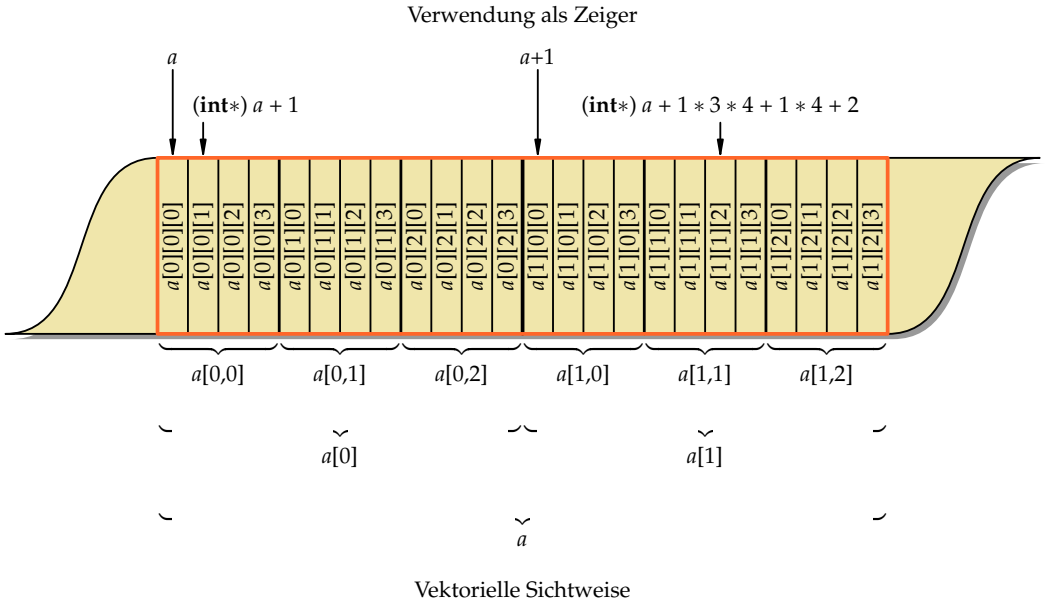


Abbildung 7.4: Repräsentierung eines dreidimensionalen Vektors im Speicher

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: $*($(**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 $*((\mathbf{int}*)b + i * 6*7 + j * 7 + k)$ 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*)

```
#include <stdio.h>

int main() {
    char array[10];
    char string[] = "Hallo!"; /* Groesse wird vom Compiler bestimmt */
    char* s1 = "Welt";
    char* s2;

    /* array = "not OK"; */ /* nicht zulaessig; array ist konstanter Zeiger
                                    * => Adresse nicht veraenderbar */

    array[0] = 'A';  /* zulaessig */
    array[1] = '\0';  /* ... und noch mit Null-Byte terminieren */
    printf("array: %s\n", array);

    /* s1[5] = 'B'; */  /* nicht zulaessig, da konstante
                                    Zeichenketten nicht veraendert werden
                                    duerfen */

    s1 = "ok";   /* zulaessig, da Zeiger nicht konstant */
    printf("s1: %s\n", s1);

    s2 = s1;   /* zulaessig, da Zeiger nicht konstant */
    printf("s2: %s\n", s2);

    string[0] = 'X';  /* zulaessig, da es sich bei string um einen
                                    nicht-konstanten Vektor handelt */
    printf("string: %s\n", string);
    printf("sizeof(string): %zd\n", sizeof(string));
}
```

```
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*)

```
#include <stdio.h>

/* Laenge einer Zeichenkette bestimmen */
int my_strlen1(char s[]) {
    int i;
    /* bis zum abschliessenden Null−Byte laufen */
    for (i = 0; s[i] != '\0'; i++); /* leere Anweisung! */
    return i;
}

/* Laenge einer Zeichenkette bestimmen */
int my_strlen2(char* s) {
    char* t = s;
    while (*t++);
    return t−s−1;
}

/* Kopieren einer Zeichenkette von s nach t
    Vorauss.: genuegend Platz in t */
void my_strcpy1(char t[], char s[]) {
    for (int i = 0; (t[i] = s[i]) != '\0'; i++);
}

/* Kopieren einer Zeichenkette von s nach t
    Vorauss.: genuegend Platz in t */
void my_strcpy2(char* t, char* s) {
    for (; (*t = *s) != '\0'; t++, s++);
}

/* Kopieren einer Zeichenkette von s nach t
    Vorauss.: genuegend Platz in t */
void my_strcpy3(char* t, char* s) {
    while ((*t++ = *s++) != '\0');
}

/* Kopieren einer Zeichenkette von s nach t
    Vorauss.: genuegend Platz in t */
void my_strcpy4(char *t, char *s) {
    /* die doppelten Klammern unterdruecken eine gcc−Warnung,
        der sonst meint, dass = mit == verwechselt worden waere */
    while ((*t++ = *s++));
}

/* Vergleich zweier Zeichenketten
    Ergebnis: 0 fuer s = t, > 0 fuer s > t und < 0 fuer s < t */
int my_strcmp1(char s[], char t[]) {
    int i;
    for (i = 0; s[i] == t[i] && s[i] != '\0'; i++);
```

```
    return s[i] − t[i];
}

/* Vergleich zweier Zeichenketten
    Ergebnis: 0 fuer s = t, > 0 fuer s > t und < 0 fuer s < t */
int my_strcmp2(char* s, char* t) {
    for (; *s == *t && *s != '\0'; s++, t++);
    return *s − *t;
}

int main() {
    char str1[] = "Hallo␣Welt!";
    char str2[20];

    printf("my_strlen1(str1)␣=␣%d\n", my_strlen1(str1));

    my_strcpy1(str2, str1);
    printf("my_strcpy1(str2,␣str1)␣=>␣str2␣=␣%s\n", str2);

    my_strcpy1(str2, "Hallo␣Tom!");
    printf("my_strcmp1(str1,␣str2)␣=␣%d\n", my_strcmp1(str1, str2));
    printf("my_strcmp1(str2,␣str1)␣=␣%d\n", my_strcmp1(str2, str1));
    printf("my_strcmp1(str1,␣str1)␣=␣%d\n", my_strcmp1(str1, str1));
}
```

```
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*)

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

```
int main() {
    char s1[30];
    char s2[] = "Welt!";
    char s3[] = "HALLO␣WELT!";
    char s4[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
    char s5[] = "␣.,:;!?";
    char s6[] = "lo";

    printf("strcpy(s1,␣\"Hallo␣\")␣=␣\"%s\",␣s1␣=␣\"%s\"\n",
        strcpy(s1, "Hallo␣"), s1);

    printf("strlen(s1)␣=␣%d\n", strlen(s1));

    printf("strcat(s1,␣s2)␣=␣\"%s\",␣s1␣=␣\"%s\"\n", strcat(s1, s2), s1);

    printf("strcmp(s1,␣s3)␣=␣%d\n", strcmp(s1, s3));
    printf("strcasecmp(s1,␣s3)␣=␣%d\n", strcasecmp(s1, s3));

    printf("strspn(s1,␣s4)␣=␣%d\n", strspn(s1, s4));
    printf("strcspn(s1,␣s5)␣=␣%d\n", strcspn(s1, s5));

    printf("strchr(s1,␣'e')␣=␣\"%s\"\n", strchr(s1, 'e'));

    printf("strpbrk(s1,␣s5)␣=␣\"%s\"\n", strpbrk(s1, s5));

    printf("strstr(s1,␣s6)␣=␣\"%s\"\n", strstr(s1, s6));
}
```

```
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*)

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

int main() {
    char text[] = "Das␣macht␣doch␣riesigen␣Spass,␣oder?!";
    char trenner[] = "␣,?!";
    char* token;
    int len = strlen(text); /* Laenge von text fuer spaeter sichern */

    /* String in Abschnitte (tokens) zerlegen */

    token = strtok(text, trenner); /* erster Aufruf: s1 = text */
    while (token != 0) { /* oder auch nur: while (token) */
        puts(token);
        token = strtok(0, trenner); /* weitere Aufrufe: s1 = 0 */
    }

    /* Inhalt des Vektors text nach strtok() ausgeben */
    puts("text[]␣=␣");
    for (int i = 0; i < len; i++) {
        if (text[i]) {
            putchar(text[i]);
        } else {
            putchar('\\'); putchar('0');
        }
    }
    putchar('\n');
}
```

```
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*)

```
struct s {
    /* ... */
    struct s* p; /* Zeiger auf die eigene Struktur ist ok */
    /* struct s elem; */ /* nicht erlaubt! */
};
```

```
struct s1 {
    /* ... */
    struct s2* p; /* Zeiger als Vorwaertsverweis ist ok */
    /* struct s2 elem;*/ /* nicht erlaubt! */
};

struct s2 {
    /* ... */
    struct s1* p; /* Zeiger als Rueckwaertsverweis ok */
    struct s1 elem; /* ok */
};
```

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*)

```
#include <stdio.h>

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

int main() {
    struct datum vorl_beginn = {16, 10, 2006};
    struct datum ueb_beginn = {24, 10, 2006};

    printf("vorher:␣%hd.%hd.%hd\n",
        vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);

    vorl_beginn = ueb_beginn;

    printf("nachher:␣%hd.%hd.%hd\n",
        vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);
}
```

```
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*)

```
#include <stdio.h>

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

/* Werteparameter−Semantik */
void ausgabe1(struct datum d) {
    printf("%hd.%hd.%hd\n", d.tag, d.monat, d.jahr);
}

/* Referenzparameter−Semantik (wirkt sich hier nicht aus) */
void ausgabe2(struct datum* d) {
    printf("%hd.%hd.%hd\n", d−>tag, d−>monat, d−>jahr);
}

/* Werteparameter−Semantik: Verbund des Aufrufers aendert sich nicht */
void setJahr1(struct datum d, int jahr) {
    d.jahr = jahr;
}

/* Referenzparameter−Semantik erlaubt die Aenderung */
void setJahr2(struct datum* d, int jahr) {
    d−>jahr = jahr;
}

int main() {
    struct datum start = {16, 10, 2006};

    ausgabe1(start);
    setJahr1(start, 2007); /* keine Aenderung! */
    ausgabe2(&start); /* aequivalent zu ausgabe1(...) */
    setJahr2(&start, 2007); /* setzt das Jahr auf 2004 */
    ausgabe1(start);
}
```

```
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*)

```
#include <stdio.h>

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

void ausgabe(struct datum d) {
    printf("%hd.%hd.%hd\n", d.tag, d.monat, d.jahr);
}

struct datum init1() {
    struct datum d = {1, 1, 1900};
    return d; /* ok, denn es wird eine Kopie erzeugt */
}

struct datum* init2() {
    struct datum d = {1, 1, 1900};
    return &d; /* nicht zulaessig, da Zeiger auf lokale Variable! */
}

int main() {
    struct datum d;
    struct datum* p;

    d = init1();
    ausgabe(d);

    p = init2(); /* Zeiger auf Variable, die nicht mehr existiert! */
    ausgabe(*p); /* wenn's klappt ... dann ist das Glueck! */
    ausgabe(*p); /* sollte eigentliche dasselbe ausgeben :-( */
}
```

```
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$
```

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

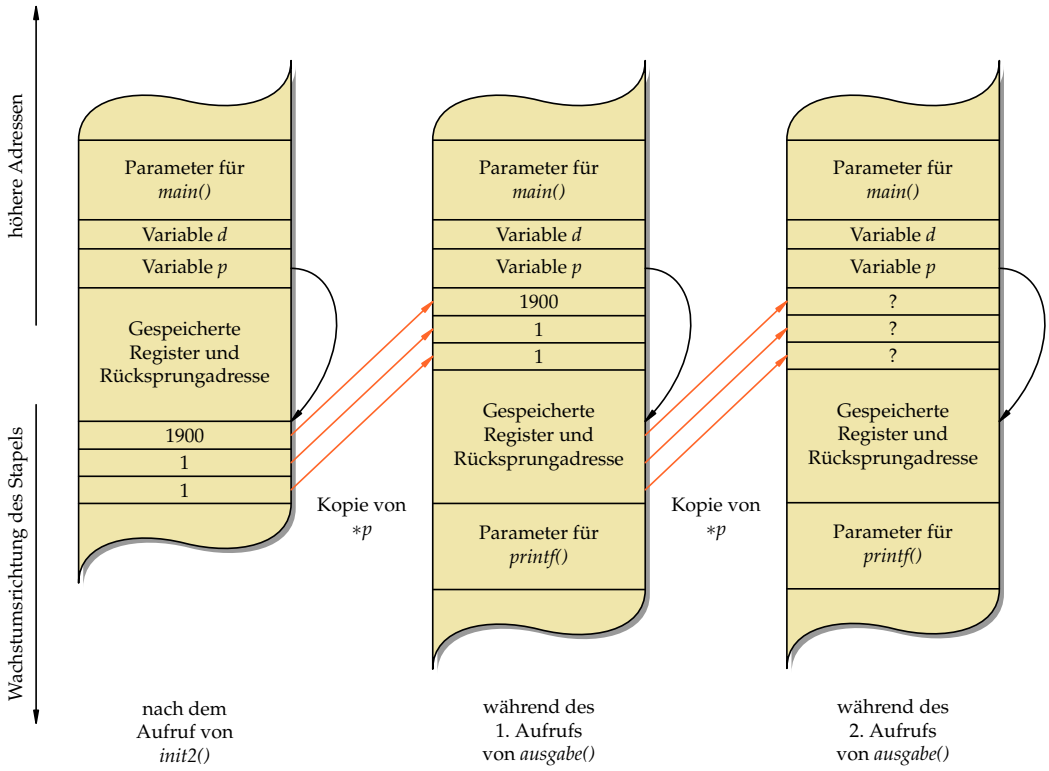


Abbildung 7.5: Funktionsaufrufe und lokale Variablen

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*)

```
#include <stdio.h>

/* Repraesentation von IP−Adressen */
union IPAddr {
    unsigned int ip;
    unsigned char b[4];
};

int main() {
    union IPAddr a;

    a.ip = 0x863c4205; /* bel. IP−Adresse in int−Darst. zuweisen */

    /* Zugriff auf a ueber die Komponente ip */
    printf("%u␣[%x]\n", a.ip, a.ip);

    /* Zugriff auf a ueber die Komponente b */
    printf("%hhu.%hhu.%hhu.%hhu␣", a.b[0], a.b[1], a.b[2], a.b[3]);
    printf("[%02hhx.%02hhx.%02hhx.%02hhx]\n",
        a.b[0], a.b[1], a.b[2], a.b[3]);

    puts("");
    printf("Speicherplatzbedarf:␣%zd\n", sizeof(a));

    puts(""); /* Anordung im Speicher analysieren */
    puts("Position␣im␣Speicher:");

    printf("a:␣␣␣␣%p\n", &a);

    printf("ip:␣␣␣%p\n", &a.ip);

    printf("b[0]:␣%p\n", &a.b[0]);
    printf("b[1]:␣%p\n", &a.b[1]);
    printf("b[2]:␣%p\n", &a.b[2]);
    printf("b[3]:␣%p\n", &a.b[3]);
```

}

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

$$\langle \text{typedef-name} \rangle \quad \longrightarrow \quad \langle \text{identifier} \rangle$$

⟨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*)

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

/* Berechnung der n-ten Fibonacci-Zahl:
   F_0 = 1
   F_1 = 1
   F_n = F_(n-1) + F_(n-2) fuer n > 1
*/
int fib(int n) {
   assert(n >= 0);
   if (n == 0 || n == 1) {
      return 1;
   } else {
      return fib(n-1) + fib(n-2);
   }
}

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

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*)

```
#include <stdio.h>

/* Vertausche die Werte der beiden Variablen,
   worauf die Zeiger a und b zeigen
*/
void swap(int* a, int* b) {
   int tmp; /* temp. Variable zum Vertauschen */
   tmp = *a; *a = *b; *b = tmp;
}

int main() {
   int x = 1, y = 2;

   printf("x␣=␣%d,␣y␣=␣%d\n", x, y);

   /* Zeiger auf x und y als Referenz uebergeben */
   swap(&x, &y);

   printf("x␣=␣%d,␣y␣=␣%d\n", x, y);
}
```

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*)

```
#include <stdio.h>

float square(); /* traditionelle Deklaration ohne Parameter */

int main() {
    float x, y;
    printf("x␣=␣");
    if (scanf("%f", &x) != 1) return 1;
    /* Vorsicht: Parameterliste ist nicht bekannt */
    y = square(x);
    printf("x^2␣=␣%f\n", y);
}

/* erst hier wird verraten, wie der Parameter von square aussieht */
float square(float x) {
    return x * x;
}
```

```
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*)

```
#include <stdio.h>

/* Male−Female−Folge von Douglas Hofstadter:
   F(0) = 1
   M(0) = 0
   F(n) = n − M(F(n−1))
   M(n) = n − F(M(n−1))
*/

/* Vorab−Deklaration von male */
int male(int n);

int female(int n) {
   if (n == 0) return 1;
   return n − male(female(n − 1));
}

/* Definition von male */
int male(int n) {
   if (n == 0) return 0;
   return n − female(male(n − 1));
}

int main() {
   puts("␣i␣F(i)␣M(i)");
   for (int i = 0; i < 10; ++i) {
      printf("%2d␣%4d␣%4d\n", i, female(i), male(i));
   }
}
```

```
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:

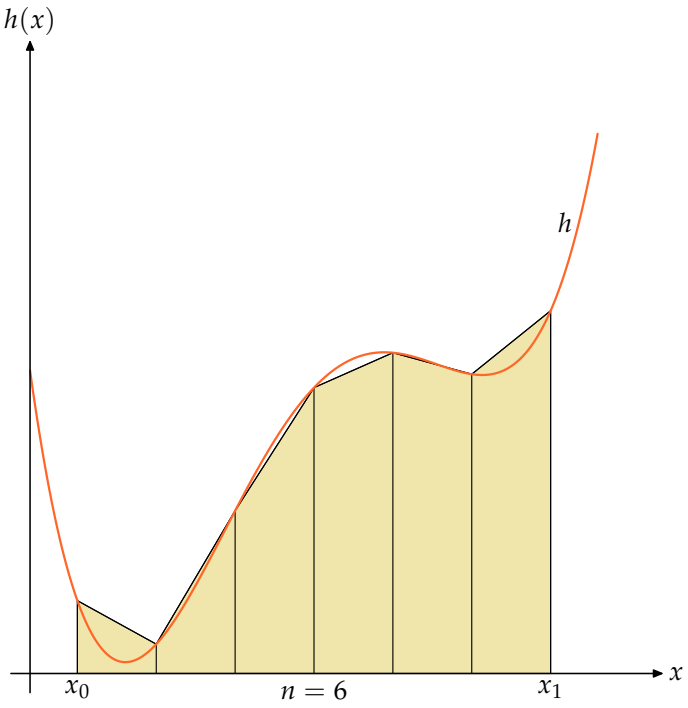


Abbildung 8.1: Integration nach der Trapezregel

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

```
#include <stdio.h>

/* Zeigertyp fuer Funktionen der folgenden Art:
   double FunktionsName(double param); */
typedef double (*Function)(double);

/* Numerische Integration der Funktion f nach der Trapezregel:
   f: zu integrierende Funktion
   [x0,x1]: Integrationsbereich
   n: Anzahl der Intervalle im Integrationsbereich */
double integrate(Function h, double x0, double x1, int n) {
   double delta = (x1 - x0) / n;
   double sum = 0.0;
   double x = x0;
   for (int i = 0; i <= n; i++, x += delta) {
      sum += (*h)(x);
   }
   return (sum - ((*h)(x0) + (*h)(x1)) / 2.0) * delta;
}

/* zwei Beispiel-Funktionen */
double f(double x) {
   return x * x;
}
double g(double x) {
```

```
    return 2.0 * x * x - 3.0 * x + 1.0;
}

int main() {
    puts("Numerische␣Integration:");
    printf("f:␣%lf\n", integrate(&f, 0, 1, 100));
    printf("g:␣%lf\n", integrate(&g, 0, 1, 100));
}
```

```
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*)

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

/* lineare Liste ganzer Zahlen */
typedef struct element {
    int i;
    struct element* next;
} element;

int main() {
    element* head = 0;
    int i;

    /* Zahlen einlesen und in der Liste
        in umgekehrter Reihenfolge ablegen */
    while ((scanf("%d", &i)) == 1) {
        element* last = (element*) calloc(1, sizeof(element));
        if (last == 0) {
            fprintf(stderr, "out␣of␣memory!\n"); exit(1);
        }
        last->i = i; last->next = head; head = last;
    }
    /* Zahlen aus der Liste wieder ausgeben */
    while (head != 0) {
        printf("%d\n", head->i);
        head = head->next;
    }
}
```

```
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, "out␣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*)

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

int main() {
    printf("page␣size␣=␣%d\n", getpagesize());
}
```

```
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

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

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.

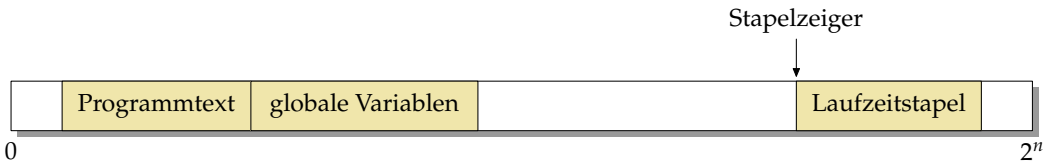


Abbildung 9.1: Aufteilung des Adressraums zu Beginn

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*)

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

```
extern void etext;
extern void edata;
extern void end;

int main() {
    int local;
    int* ip = &local;

    printf("location␣of␣main():␣␣␣␣␣␣␣␣%8p\n", &main);
    printf("end␣of␣program␣text:␣␣␣␣␣␣␣%8p\n", &etext);
    printf("end␣of␣initialized␣data:␣␣␣%8p\n", &edata);
    printf("end␣of␣uninitialized␣data:␣%8p\n", &end);
    printf("address␣of␣local␣variable:␣%8p\n", ip);
}
```

```
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).

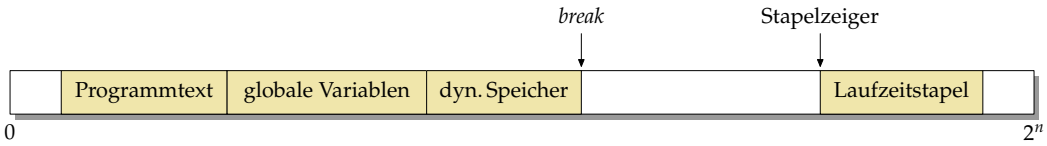


Abbildung 9.2: Dynamisch belegter Speicher im Adressraum

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*)

```
#include <stdio.h>
#include <stdlib.h> // fuer exit
#include <unistd.h> // fuer getpagesize und sbrk

int main() {
    // aktuelle Kachelgroesse ermitteln
    int pagesize = getpagesize();
    // eine Kachel dynamisch allokieren,
    // im Erfolgsfalle zeigt text auf die
    // die alte Position des break
    char* buf = (char*) sbrk(pagesize);
    if (buf == (void*) -1) {
        // Fehlermeldung ausgeben und Schluss
        perror("sbrk"); exit(1);
    }

    // Zeile einlesen und reversiert ausgeben
    int ch; char* cp = buf+pagesize;
    *--cp = '\0';
    while ((ch = getchar()) != EOF && ch != '\n' && cp > buf) {
        *--cp = ch;
    }
    printf("%s\n", cp);
}
```

```
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 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

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

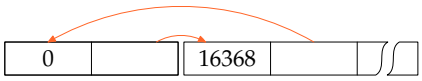


Abbildung 9.3: Ring der freien Speicherflächen zu Beginn

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.

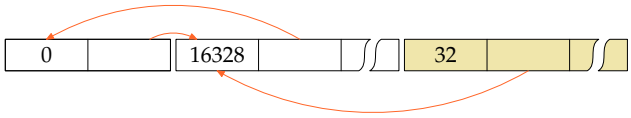


Abbildung 9.4: Speicherverwaltungsstruktur nach der ersten Belegung einer Speicherfläche

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.

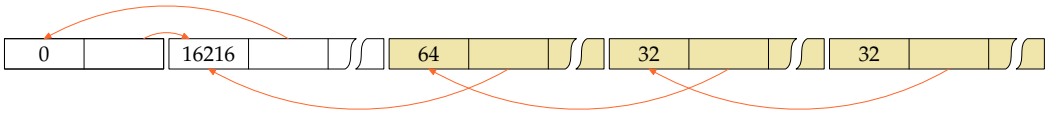


Abbildung 9.5: Speicherverwaltungsstruktur nach drei Speicherbelegungen

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.

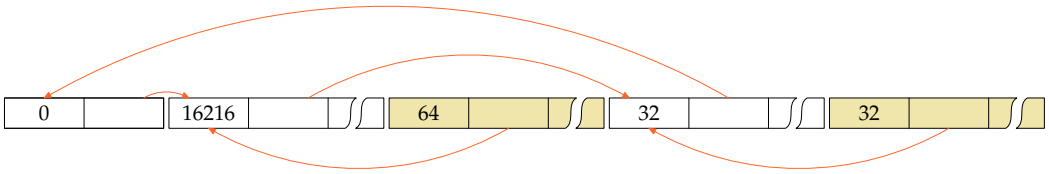


Abbildung 9.6: Speicherverwaltungsstruktur nach einer Speicherfreigabe

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

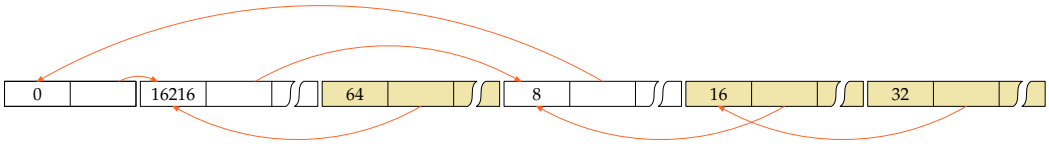


Abbildung 9.7: Suche nach freien Flächen entsprechend des *circular first fit*-Verfahrens (Teil 1)

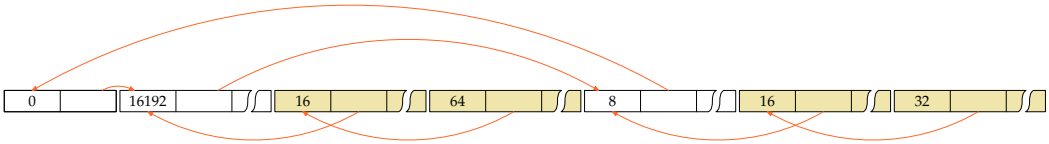


Abbildung 9.8: Suche nach freien Flächen entsprechend des *circular first fit*-Verfahrens (Teil 2)

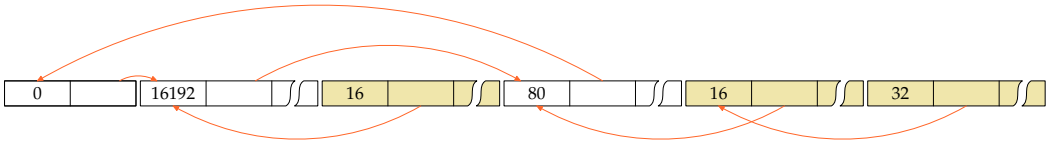


Abbildung 9.9: Zusammenlegung benachbarter freier Speicherflächen

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*)

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#define MEM_SIZE 2048 /* zu multiplizieren mit sizeof(memnode) */
/* Verbund, der einer freien oder
   belegten Speicherflaeche vorangeht */
typedef struct memnode {
   size_t size;
      /* Groesse der Speicherflaeche, die diesem Verbund
         unmittelbar folgt, jedoch ohne Einberechnung
         des Speicherbedarfs fuer diesen Verbund */
   struct memnode* next;
      /* verweist bei freien Speicherflaechen zum naechsten
         Ringelement der freien Speicherflaechen;
         bei belegten Speicherflaechen wird dieser Verweis
         geaendert zu dem unmittelbar vorangehenden Speicherelement,
```

```
            egal ob dieses frei oder belegt ist
        */
} memnode;

/* Speicherbereich, der von den folgenden Funktionen
    verwaltet wird */
memnode dynmem[MEM_SIZE] = {
    /* bleibt immer im Ring der freien Speicherflaechen */
    {0, &dynmem[1]},
    /* enthaelt zu Beginn den gesamten freien Speicher */
    {sizeof dynmem − 2*sizeof(memnode), dynmem}
};
memnode* node = dynmem; /* durchlaeuft den Ring der freien Speicherflaechen */
memnode* root = dynmem; /* verweist auf das erste Element im Ring */

/* um nicht zu sehr zu fragmentieren, werden alle zu belegenden
    Speicherflaechen auf ein Vielfaches von ALIGN aufgerundet;
    dies sollte eine Zweierpotenz sein, die gross genug ist,
    um die Ausrichtungsanforderungen aller elementaren Datentypen
    zu befriedigen */
const int ALIGN = sizeof(memnode);

/* ermittelt, ob die Speicherflaeche bei ptr belegt ist */
bool is_allocated(memnode* ptr) {
    /* Das Wurzelelement ist immer frei */
    if (ptr == root) return false;
    /* Wenn der Zeiger vorwaerts zeigt, dann ist es frei */
    if (ptr−>next > ptr) return false;
    /* Der Zeiger zeigt jetzt rueckwaerts, denn
        ptr−>next == ptr ist nur beim Wurzelelement moeglich */
    /* Wenn nicht auf Wurzel verwiesen wird, ist ptr belegt */
    if (ptr−>next != root) return true;
    /* Wenn ptr−>next auf die Wurzel verweist, gibt es zwei
        Faelle:
            − Es ist belegt und ptr liegt unmittelbar hinter
              dem Wurzelelement
            − Es ist frei und ptr zeigt auf die freie Speicherflaeche
              mit der hoechsten Adresse
    */
    /* Wenn root−>next jenseits von ptr zeigt, dann ist ptr belegt */
    if (root−>next > ptr) return true;
    /* Wenn die Wurzel das einzige freie Element ist,
        dann ist ptr ebenfalls belegt */
    return root−>next == root;
}

/* liefert einen Zeiger auf die p folgende Speicherflaeche,
    unabhaengig davon, ob sie belegt oder frei ist;
    zu beachten ist, dass bei der hoechstgelegenen Speicherflaeche
    dynmem + MEM_SIZE zurueckgeliefert wird */
memnode* successor(memnode* p) {
    return (memnode*) ((char*)(p+1) + p−>size);
}
```

```
/* entspricht malloc(), aber bedient sich nur aus dynmem[] */
void* my_malloc(size_t size) {
    assert(size >= 0);
    if (size == 0) return 0;
    /* runde die gewuenschte Groesse auf
        das naechste Vielfache von ALIGN */
    if (size % ALIGN) {
        size += ALIGN − size % ALIGN;
    }
    /* beginnend mit node−>next suchen wir nach
        einem Element aus dem Ring der freien Speicherflaechen,
        das genuegend Kapazitaet fuer size Bytes bietet;
        dabei verweist ptr auf das Element, das wir
        gerade untersuchen, und prev auf das vorherige Element */
    memnode* prev = node; memnode* ptr = prev−>next;
    do {
        if (ptr−>size >= size) break; /* passendes Element gefunden */
        prev = ptr; ptr = ptr−>next;
    } while (ptr != node); /* bis der Ring durchlaufen ist */
    if (ptr−>size < size) return 0; /* Speicher ist ausgegangen */
    if (ptr−>size < size + 2*sizeof(memnode)) {
        node = ptr−>next; /* "circular first fit" */
        /* entferne ptr aus dem Ring der freien Speicherflaechen */
        prev−>next = ptr−>next;
        /* suche nach der unmittelbar vorangehenden Speicherflaeche;
            zu beachten ist hier, dass zwischen prev und ptr noch
            einige belegte Speicherflaechen liegen koennen
        */
        for (memnode* p = prev; p < ptr; p = successor(p)) {
            prev = p;
        }
        ptr−>next = prev;
        /* node muss immer auf ein freies Element zeigen */
        if (ptr == node) node = root;
        return (void*) (ptr+1);
    }
    node = ptr; /* "circular first fit" */
    /* die bisherige Speicherflaeche wird zerlegt
        in einen neue passende Speicherflaeche, die jetzt belegt wird
        und die alte verkleinerte Speicherflaeche, die uebrigbleibt
    */
    /* lege das neue Element an */
    memnode* newnode = (memnode*)((char*)ptr + ptr−>size − size);
    newnode−>size = size; newnode−>next = ptr;
    /* korrigiere den Zeiger der folgenden Speicherflaeche,
        falls sie belegt sein sollte */
    memnode* next = successor(ptr);
    if (next < dynmem + MEM_SIZE && next−>next == ptr) {
        next−>next = newnode;
    }
    /* reduziere die Groesse des alten Elements
        aus dem Ring der freien Speicherflaechen */
```

```
    ptr->size -= size + sizeof(memnode);
    return (void*) (newnode+1);
}

/* Lege die beiden freien Speicherflaechen bei prev und ptr
    zusammen, falls dies moeglich sein sollte */
void join_if_possible(memnode* prev, memnode* ptr) {
    memnode* p = (memnode*)((char*) prev + sizeof(memnode) + prev->size);
    /* Wenn die beiden Speicherflaechen nicht direkt
        hintereinander liegen, koennen wir sie nicht vereinigen */
    if (p != ptr) return;
    /* Das Wurzelelement muss in Ruhe gelassen werden,
        damit immer mindestens ein Element im Ring der freien
        Speicherflaechen verbleibt */
    if (prev->size == 0) return;
    /* Vereinigung der beiden freien Speicherflaechen */
    prev->next = ptr->next;
    prev->size += ptr->size + sizeof(memnode);
    /* korrigiere den Zeiger der folgenden Speicherflaeche,
        falls sie belegt sein sollte */
    memnode* next = successor(ptr);
    if (next < dynmem + MEM_SIZE && next->next == ptr) {
        next->next = prev;
    }
    /* setze node auf prev, falls sie zuvor auf ptr zeigte */
    if (node == ptr) node = prev;
}

/* gibt die Speicherflaeche bei ptr wieder frei;
    ptr muss zuvor von my_malloc() zurueckgeliefert worden sein */
void my_free(void* ptr) {
    if (!ptr) return;
    memnode* node = (memnode*) ((char*) ptr - sizeof(memnode));
    /* Ueberpruefe, ob node tatsaechlich belegt ist */
    assert(is_allocated(node));
    /* Wir suchen nach dem vorangehenden Element aus
        dem Ring der freien Speicherflaechen */
    memnode* prev = node->next;
    while (is_allocated(prev)) {
        prev = prev->next;
    }
    /* fuege node wieder in den Ring an passender Stelle ein */
    node->next = prev->next; prev->next = node;
    /* Lege zusammenhaengende freie Speicherflaechen wieder
        zusammen, falls dies moeglich ist */
    join_if_possible(node, node->next);
    join_if_possible(prev, node);
}

/* Gibt zur Analyse den gesamten Status von dynmem[] auf
    der Standardausgabe aus;
    hier wird angenommen, dass Zeiger bequem als int
    darstellbar sind --> das ist nicht portabel */
```

```
void debug_alloc(void) {
    for (memnode* ptr = root; ptr < dynmem + MEM_SIZE; ptr = successor(ptr)) {
        if (ptr == node) {
            putchar('*');
        } else {
            putchar('␣');
        }
        bool allocated = is_allocated(ptr);
        if (allocated) {
            putchar('A');
        } else {
            putchar('F');
        }
        putchar('␣');
        printf("%d␣%5lu␣-->␣%d", (int) ptr,
            (unsigned long) ptr->size, (int)ptr->next);
        if (allocated) {
            /* Ausgabe der freigebbaren Adresse */
            printf("␣(f␣%d)", (int)(ptr+1));
        }
        putchar('\n');
    }
}

/* Interaktives Testprogramm */
int main() {
    char command;
    int intval;
    void* p;

    while (printf(":␣"), scanf("␣%c", &command) == 1) {
        switch (command) {
            case 'a':
                if (scanf("%d", &intval) != 1) break;
                p = my_malloc(intval);
                if (p) {
                    printf("got␣%d␣bytes␣at␣%d\n", intval, (int) p);
                } else {
                    printf("failed\n");
                }
                break;
            case 'f':
                if (scanf("%d", &intval) != 1) break;
                my_free((void*) intval);
                break;
            case 'd':
                debug_alloc();
                break;
            default:
                printf("supported␣commands:\n");
                printf("a␣n␣␣␣␣␣␣␣allocate␣n␣bytes\n");
                printf("d␣␣␣␣␣␣␣␣␣debugging␣output\n");
                printf("f␣p␣␣␣␣␣␣␣free␣area␣at␣p\n");
```

```
            break;
        }
    }
}
```

```
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*)

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <time.h>

typedef struct {
    int* array; /* Zeiger auf das erste Element des Vektors */
    int size; /* Groesse des belegten Speichers */
    int used; /* tatsaechliche Groesse des Vektors; used <= size! */
} IntArray;

void* my_calloc(size_t nelem, size_t elsize) {
    assert(nelem > 0 && elsize > 0);
    void* ptr = calloc(nelem, elsize);
    if (ptr) return ptr; /* alles OK */
    /* Fehlerbehandlung: */
    fprintf(stderr, "out␣of␣memory␣−−␣aborting!\n");
    /* Termination mit core dump */
    abort();
}
```

```
void* my_realloc(void* ptr, size_t size) {
    ptr = realloc(ptr, size);
    if (ptr) return ptr; /* alles OK */
    /* Fehlerbehandlung: */
    fprintf(stderr, "out␣of␣memory␣--␣aborting!\n");
    /* Termination mit core dump */
    abort();
}

/*
 * Neues Integer-IntArray mit der angegebenen
 * Groesse erzeugen und zurueckgeben
 *
 * Parameter "IntArray **a" notwendig, um IntArray-Zeiger
 * als Referenz-Parameter zu uebergeben!
 */
void arrayNew(IntArray** a, int len) {
    assert(len >= 0);
    *a = (IntArray*) my_calloc(1, sizeof(IntArray));
    (*a)->size = (*a)->used = len;
    if (len > 0) {
        (*a)->array = (int*) my_calloc(len, sizeof(int));
    } else {
        (*a)->array = 0;
    }
}

/* Integer-IntArray wieder freigeben */
void arrayDelete(IntArray** a) {
    if ((*a)->array) free((*a)->array);
    free(*a);
    *a = 0;
}

/* Groesse des Integer-Arrays veraendern */
void arrayResize(IntArray* a, int len) {
    assert(len >= 0);
    if (a->size < len) {
        /* da realloc durch das Umkopieren recht
           teuer sein kann, empfiehlt es sich,
           gleich etwas mehr Speicher zu reservieren
        */
        int newlen = len + (len << 3) + 10;
        a->array = (int*) my_realloc(a->array, sizeof(int) * newlen);
        a->size = newlen;
    }
    a->used = len;
}

/* Laenge/Groesse des Arrays ermitteln */
int arrayLength(IntArray* a) {
    return a->used;
}
```

```
/* Element des Arrays lesen */
int arrayGet(IntArray* a, int i) {
    assert(i >= 0 && i < a->used);
    return a->array[i];
}

/* Element des Arrays neu setzen */
void arraySet(IntArray* a, int i, int value) {
    assert(i >= 0 && i < a->used);
    a->array[i] = value;
}

int main() {
    IntArray* a;
    int value;

    /* Pseudo-Zufallszahlengenerator initialisieren */
    unsigned int seed = time(0); srand(seed);

    arrayNew(&a, 0);

    /* Alle Zahlen aus der Standardeingabe einlesen */
    for (int index = 0; scanf("%d", &value) == 1; ++index) {
        arrayResize(a, index + 1); arraySet(a, index, value);
    }

    /* Misch-Algorithmus von Fisher & Yates, 1938 */
    for (int i = arrayLength(a) - 1; i >= 0; --i) {
        /* Ein Element aus dem Bereich 0..i auswaehlen */
        int j = rand() % (i + 1);
        /* Vertausche die Elemente mit den Indizes i und j */
        if (i != j) {
            int tmp = arrayGet(a, i);
            arraySet(a, i, arrayGet(a, j));
            arraySet(a, j, tmp);
        }
    }

    /* Alle Zahlen aus dem Vektor ausgeben */
    for (int i = 0; i < arrayLength(a); ++i) {
        printf("%d\n", arrayGet(a, i));
    }
    arrayDelete(&a);
}
```

```
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*)

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

int main() {
    char s[] = "Ratatouille";
    char* s1 = strdup(s);

    memset(s1+1, 'i', 3);
    printf("s1 = \"%s\"\n", s1);

    printf("memcmp(s, s1, strlen(s)) = %d\n", memcmp(s, s1, strlen(s)));

    memcpy(s1, s, strlen(s)+1);
    printf("s1 = \"%s\"\n", s1);

    memmove(s1+2, s1, 4);
    printf("s1 = \"%s\"\n", s1);

    printf("memchr(s1, 't', strlen(s1)) = %s\n",
        (char*) memchr(s1, 't', strlen(s1)));

    free(s1);
}
```

```
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[]) {
    // ...
}
```

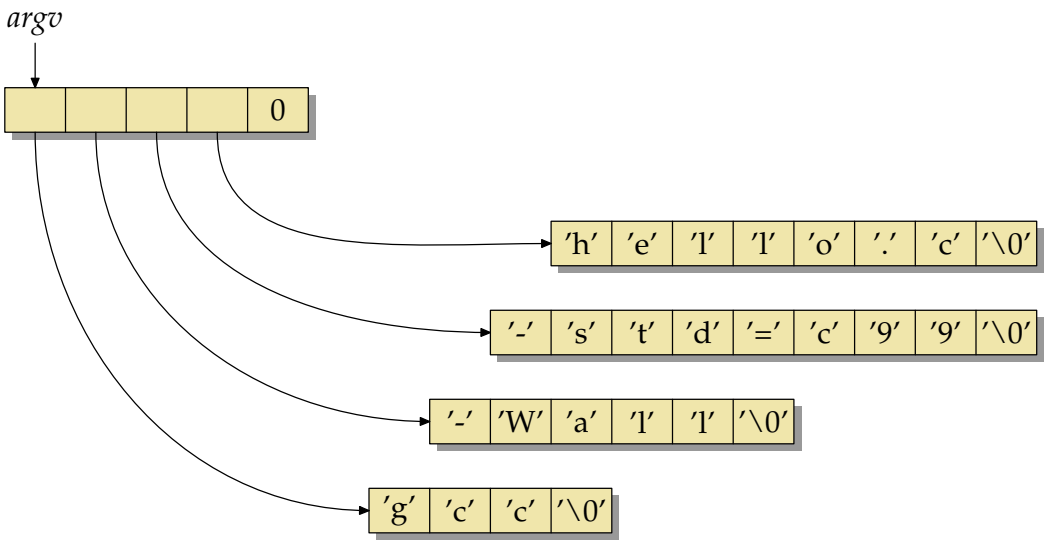


Abbildung 10.1: Die Repräsentierung von *argv* am Beispiel von *„gcc -Wall -std=c99 hello.c“*

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=c99 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*)

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("Kommandoname:␣%s\n", argv[0]);
    printf("Anzahl␣der␣Kommandozeilenargumente:␣%d\n", argc−1);
    for (int i = 1; i < argc; i++) {
        printf("Argument␣%d:␣%s\n", i, argv[i]);
    }
}
```

```
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*)

```
#include <stdio.h>

int main(int argc, char** argv) {
    char** argp = argv;
    printf("Kommandoname:␣%s\n", *argp++);
    while (*argp) {
        printf("Argument␣%d:", argp - argv);
        printf("␣%s\n", *argp++);
    }
}
```

```
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.

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*)

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

int main(int argc, char *argv[]) {
    char line[256];

    if (argc != 2) {
        fprintf(stderr, "Usage:␣%s␣pattern\n", argv[0]);
        exit(1);
    }

    while (fgets(line, sizeof line, stdin)) {
        if (strstr(line, argv[1])) {
            fputs(line, stdout);
        }
    }
}
```

```
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*)

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

[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.

```
#include <string.h>

/* Lese eine beliebig lange Zeile von fp ein und
   liefere einen Zeiger auf eine dynamisch organisierte
   Speicherflaeche zurueck, in dem die Zeile abgelegt
   worden ist
*/
char* readline(FILE* fp) {
    int len = 32;
    char* cp = malloc(len);
    int i = 0;
    int ch;
    while ((ch = getc(fp)) != EOF && ch != '\n') {
        cp[i++] = ch;
        if (i == len) {
            /* verdoppele die Speicherflaeche */
            len *= 2;
            char* newcp = realloc(cp, len);
            if (!newcp) {
                free(cp);
                return 0;
            }
            cp = newcp;
        }
    }
    if (i == 0 && ch == EOF) {
        free(cp);
        return 0;
    }
    cp[i++] = 0;
    return realloc(cp, i);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage:␣%s␣pattern\n", argv[0]);
        exit(1);
    }

    for (char* line; (line = readline(stdin)); free(line)) {
        if (strstr(line, argv[1])) {
            puts(line);
        }
    }
}
```

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*)

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

/* Lese eine beliebig lange Zeile von fp ein und
   liefere einen Zeiger auf eine dynamisch organisierte
   Speicherflaeche zurueck, in dem die Zeile abgelegt
   worden ist
*/
char* readline(FILE* fp) {
    int len = 32;
    char* cp = malloc(len);
    int i = 0;
    int ch;
    while ((ch = getc(fp)) != EOF && ch != '\n') {
        cp[i++] = ch;
        if (i == len) {
            /* verdoppele die Speicherflaeche */
            len *= 2;
            char* newcp = realloc(cp, len);
            if (!newcp) {
                free(cp);
                return 0;
            }
            cp = newcp;
        }
    }
    if (i == 0 && ch == EOF) {
        free(cp);
        return 0;
    }
    cp[i++] = 0;
    return realloc(cp, i);
}

int main(int argc, char *argv[]) {
    const char* cmdname = argv[0]; /* Kommando-Name */
    const char usage[] = "Usage:␣%s␣[-nv]␣pattern\n";
    bool opt_v = false; /* Option -v gesetzt? */
    bool opt_n = false; /* Option -n gesetzt? */

    /* Optionen verarbeiten */
    while (--argc > 0 && **++argv == '-') {
        /* nach "--" kommen keine Optionen mehr (per Definition) */
        if (argv[0][1] == '-' && argv[0][2] == '\0') {
            argc--; argv++; break;
        }
        /* steht nur "-", dann fehlt etwas ;-) */
        if (argv[0][1] == '\0') {
            fprintf(stderr, "%s:␣empty␣option\n", cmdname);
            fprintf(stderr, usage, cmdname);
```

```
            exit(1);
        }
        for (char* s = *argv + 1; *s != '\0'; s++) {
            switch (*s) { /* jede einzelne Option abarbeiten ... */
                case 'v' : opt_v = true; break;
                case 'n' : opt_n = true; break;
                default: /* unbekannte Option! */
                    fprintf(stderr, "%s:␣illegal␣option␣'%c'\n", cmdname, *s);
                    fprintf(stderr, usage, cmdname);
                    exit(1);
            }
        }
    }

    /* Jetzt sollte nur noch das Suchmuster uebrig bleiben */
    if (argc != 1) {
        fprintf(stderr, usage, cmdname);
        exit(1);
    }
    char* substring = argv[0];

    int lineno = 1;
    for (char* line; (line = readline(stdin)); free(line)) {
        if (!strstr(line, substring) == opt_v) {
            if (opt_n) {
                printf("%d:␣", lineno);
            }
            puts(line);
        }
        lineno++;
    }
}
```

```
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*)

```
#include <stdio.h>

#define CONST 3
#define MAX(x,y) ((x) > (y)? (x): (y))

int main() {
    int a = 4, b = CONST;

    printf("Maximum␣von␣%d␣und␣%d:␣%d\n", a, b, MAX(a, b));
}
```

```
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*)

```
#include <stdio.h>

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

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*)

```
#include <stdio.h>

#define X 1
int x1 = X;
#define X 2
int x2 = X;

int main() {
    printf("x1 = %d, x2 = %d\n", x1, x2);
}
```

```
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*)

```
#include <stdio.h>

#define X 1
int x1 = X;
#undef X
#define X 2
int x2 = X;

int main() {
    printf("x1 = %d, x2 = %d\n", x1, x2);
}
```

```
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*)

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

#define ASSERT(cond) \
    ( \
        (cond)? 1: ( \
            fprintf(stderr, "%s␣(%d):␣assertion␣\"%s\"␣failed\n", \
                __FILE__, __LINE__, #cond), \
            abort() \
        ) \
    )

int main() {
    int a = 3;
    ASSERT(a < 0);
}
```

```
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*)

```
#include <stdio.h>

#ifndef DEBUG
# define DEBUG(x)
#endif

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

```
#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*)

```
#include <stdio.h>

#ifdef DBG
# ifndef DEBUG
# define DEBUG(x) (puts((x)))
# endif
#else
# undef DEBUG
# define DEBUG(x)
#endif

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

}

```
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*)

```
#include <stdio.h>

int main() {
#if x == 1
    puts("1");
#elif x == 2
    puts("2");
#else
    puts("anderer␣Wert␣oder␣nicht␣definiert");
#endif
}
```

```
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*)

```
int f(int a, int b); /* eine Funktions−Deklaration */

/* eine Funktions−Definition */
int max(int a, int b) {
    return a > b? a: b;
}

/* eine Variablendefinition */
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*)

```
/* eine Variablen−Deklaration */
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*)

```
#include <stdio.h>

/* Deklarationen */
extern int i;
extern void f();

int main() {
    printf("Wert␣von␣i␣vor␣dem␣Aufruf:␣%d\n", i);
    f();
    printf("Wert␣von␣i␣nach␣dem␣Aufruf:␣%d\n", i);
}
```

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

```
int i = 1; /* Definition */

void f() { /* Definition */
    ++i;
}
```

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

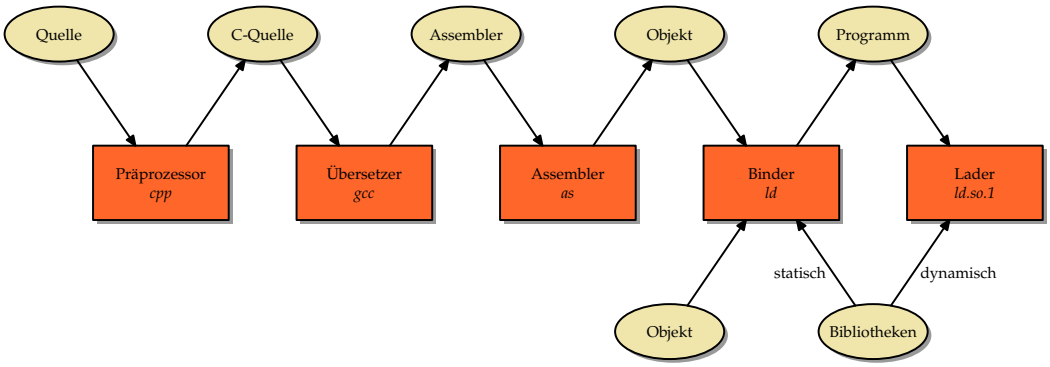


Abbildung 12.1: Der Weg von der Quelle zum ausführbaren Programm

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 abweichender 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*)

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

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

```
#include "ggt.h"

int ggt(int a, int b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
#include "ggt.h"

int main(int argc, char* argv[]) {
    char* cmdname = *argv++; --argc;
    char usage[] = "Usage: %s a b\n";
    if (argc != 2) {
        fprintf(stderr, usage, cmdname);
        exit(1);
    }
    int a = atoi(argv[0]);
    int b = atoi(argv[1]);
    if (a > 0 && b > 0) {
        printf("%d\n", ggt(a, b));
    } else {
        fprintf(stderr, usage, cmdname);
        exit(1);
    }
```

}

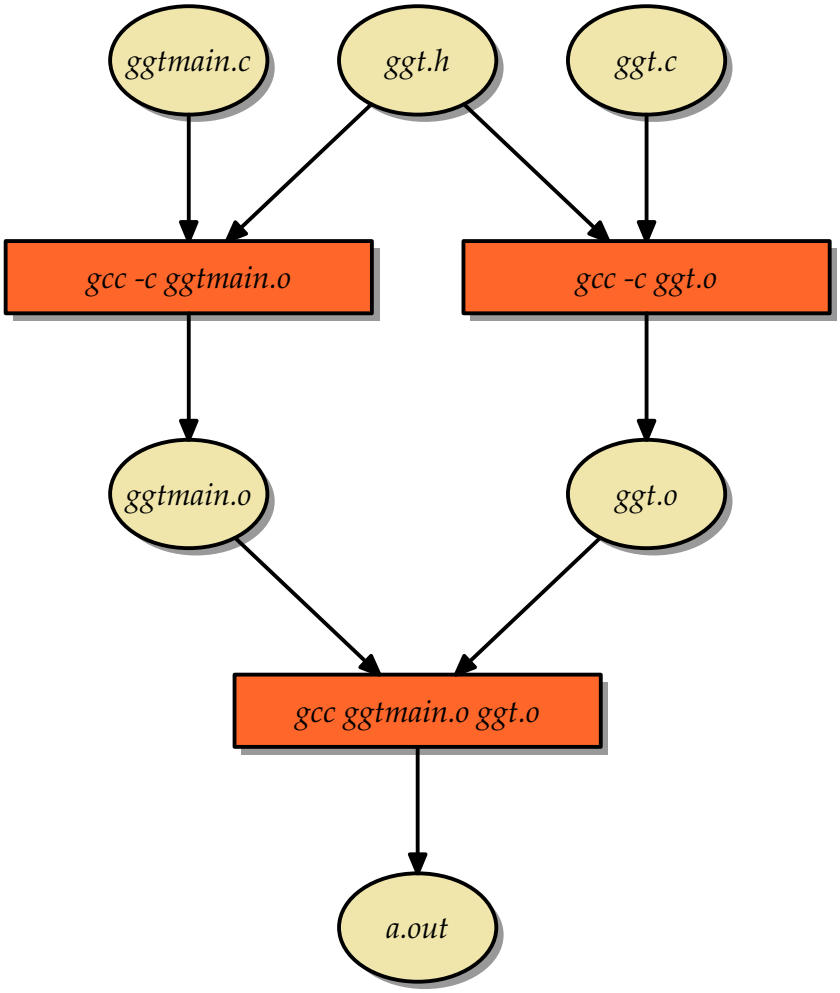


Abbildung 12.2: Übersetzungsvorgang bei ausgelagerten Deklarationen

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*)

```
a.out:  ggt.o ggtmain.o
                gcc ggt.o ggtmain.o
ggt.o:  ggt.h ggt.c
                gcc -c -Wall -std=c99 ggt.c
ggtmain.o: ggt.h ggtmain.c
                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* (*Makefile*)

```
Sources := $(wildcard *.c)
Objects := $(patsubst %.c,%.o,$(Sources))
Target := ggt
CC := gcc
CFLAGS := -Wall -std=c99
$(Target): $(Objects)
                gcc -o $(Target) $(Objects)
.PHONY: depend
depend:
                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 Schlüsselword **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*)

```
#include <stdio.h>

void work() {
    static int counter = 0;
    printf("␣%d", counter++);
}

int main() {
    for (int i = 0; i < 10; i++) {
        work();
    }
    puts("");
}
```

```
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.

13.2 Der Übergang von ANSI C / C90 zu C99

Der gcc unterstützt in der Version 3.2, die momentan bei uns installiert ist, einen Großteil der Erweiterungen von *C99*, wenn auch nicht alle. (Unter *http://gcc.gnu.org/c99status.html* ist der aktuelle Stand der C99-Konformität von gcc dokumentiert.) Um diesen C-Dialekt zu verwenden, muss man den gcc mit der Kommandozeilen-Option *-std=c99* aufrufen. Im Folgenden werden wir die *wesentlichen Neuerungen in C99* (gegenüber C90) kennen lernen, wobei es in diesem Rahmen unmöglich ist, auf Vollständigkeit Wert zu legen.

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*)

```
#include <stdio.h>

int main() { // Hauptprogramm
    puts("Hallo!"); // Ausgabe
    return 0;
```

```
}
```

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*)

```
#include <stdio.h>

int main() {
    int x = 17;
    printf("x␣=␣%d\n", x);

    int y = x * x - 2;
    printf("y␣=␣%d\n", y);

    return 0;
}
```

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*)

```
#include <stdio.h>

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

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 (ganz ohne Zeiger-Arithmetik ;-).

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

```
#include <stdio.h>

    // BEACHTE: Die Parameter s1 und s2 muessen
    // vor int m[s1][s2] stehen, wo sie verwendet werden!
void print(int s1, int s2, int m[s1][s2]) {
```

```
    for (int i = 0; i < s1; i++) {
        for (int j = 0; j < s2; j++)
            printf("%4d␣", m[i][j]); // komfortabler Element−Zugriff ;−)
        puts("");
    }
}

void printsum(int len, int a[len], int b[len]) {
    int c[len]; // (eigentlich ueberfluessiges) lokales Array
    for (int i = 0; i < len; i++)
        c[i] = a[i] + b[i];
    printf("(");
    for (int i = 0; i < len; i++)
        printf(i > 0 ? ",␣%d" : "%d", c[i]);
    printf(")\n");
}

int main() {
    int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
    print(2, 3, matrix);
    printsum(3, matrix[0], matrix[1]);
    return 0;
}
```

```
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*)

```
#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*)

```
#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;
}
```

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*)

```
#include <stdio.h>

#define LEN 5

int main() {
    int a[LEN] = {[1 ... 3] = −1, [2] = 2};
    for (int i = 0; i < LEN; i++)
        printf("a[%d]␣=␣%d\n", i, a[i]);
    return 0;
}
```

```
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*)

```
#include <stdio.h>

int main() {
    char c = getchar();
    switch (c) {
        case 'a' ... 'z' : puts("Kleinbuchstabe"); break;
        case 'A' ... 'Z' : puts("Grossbuchstabe"); break;
        case '0' ... '9' : puts("Ziffer"); break;
        default : puts("anderes␣Zeichen"); break;
    }
    return 0;
}
```

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*)

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

int main() {
    _Bool c = true;
    printf("c␣=␣%s\n", c ? "true" : "false");
    printf("c␣=␣%d\n", c);
    c = 1;
    printf("c␣=␣%s\n", c ? "true" : "false");
    printf("c␣=␣%d\n", c);
    return 0;
}
```

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*)

```
#include <stdio.h>

int main() {
```

```
    long long int n = 123123123123;
    printf("n␣=␣%lli\n", n);
    return 0;
}
```

```
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*)

```
#include <stdio.h>

#define LEN 10

int main() {
    char *label = "Bezeichnung";
    int x = 387;
    char s[LEN];

    snprintf(s, LEN, "%s:␣%d", label, x);
    puts(s);

    return 0;
}
```

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

Anmerkung: Bei Verwendung des gcc in der Version 3.2 muss momentan leider noch das Makro *__EXTENSIONS__* definiert werden, damit die Funktion *snprintf() auch deklariert* wird!

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*)

```
#include <stdio.h>

#define DEBUG(...) fprintf(stderr, __VA_ARGS__);
#define QUOTE(...) puts(#__VA_ARGS__);
#define TEST(cond, ...) if (!(cond)) \
                                DEBUG(__VA_ARGS__)

int main() {
    int x = 3;
    DEBUG("x␣=␣%d\n", x)
    QUOTE(x < 2, x > 0, x)
    TEST(x == 0, "failed:␣x␣=␣%d\n", x)
    return 0;
}
```

```
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*)

```
#include <stdio.h>

#define ASSERT(cond,...) \
            if (!(cond)) { \
                fprintf(stderr, "%s␣(%d):␣%s():␣assertion␣\"%s\"␣failed\n", \
```

```
                    __FILE__, __LINE__, __func__, #cond); \
                __VA_ARGS__ \
            }

void work() {
    ASSERT(1 == 3)
}

int main() {
    int a = 3;
    ASSERT(a < 0, fprintf(stderr, "DEBUG:␣a␣=␣%d\n", a);)
    work();
    return 0;
}
```

```
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*)

```
#include <stdio.h>

int count = 0;

void work_slow() {
    count++;
}

inline void work_fast() {
    count++;
}

int main() {
    for (int i = 0; i < 100000000; i++)
#ifdef FAST
        work_fast();
#else
        work_slow();
#endif
    return 0;
```

}

```
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.

Kapitel 14

Sicheres Programmieren mit C

Gerade bei systemnaher Software muss man sich auch über Sicherheit Gedanken machen. Diese Software wird oft mit Superuser-Rechten ausgeführt (z. B. *passwd*) und von „normalen“ Benutzern aufgerufen (bzw. kann zumindest von „normalen“ Benutzern beeinflusst werden). Durch die *Laufzeitumgebung* (z. B. *Umgebungsvariablen*), *Eingaben*, *Kommandozeilen-Argumente* etc. kann u. a. durch den Aufrufer auf das Programm Einfluss genommen werden. Bietet das Programm Angriffspunkte, so wird dies garantiert von Angreifern ausgenutzt, die so im schlimmsten Fall Superuser- bzw. Root-Rechte erhalten.

14.1 Typische Schwachstellen

Um gängige Schwachstellen in C kennen zu lernen, betrachten wir folgendes Beispiel-Programm:

Programm 14.1: C-Programm mit vielen Schwachstellen (*pubcat.c*)

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

#define BUFSIZE 256
#define PUBDIR "/home/thales/jmayer/pub"

int main(int argc, char *argv[]) {
    char s[BUFSIZE];

    if (argc != 2) {
        fprintf(stderr, "Usage:␣%s␣<filename>\n", argv[0]);
        return 1;
    }

    sprintf(s, "cat␣%s/%s", PUBDIR, argv[1]);

    system(s);

    return 0;
}
```

Mit diesem Programm soll es möglich sein, Dateien aus einem öffentlichen Ordner anzeigen zu lassen. Hierzu verwenden wir die Funktion *system()* aus *stdlib.h*, die das über-

gebene Unix-Kommando ausführt. Das Kommando *cat* gibt einfach die Dateien auf die Standardausgabe aus, deren Namen es als Kommandozeilen-Argumente erhält.

Programm 14.1, das *Benutzer jmayer* erzeugt hat, wird nun übersetzt und als Kommando *pubcat* in das Verzeichnis /tmp gestellt:

```
thales$ id
uid=13847(jmayer) gid=200(sai)
thales$ gcc -Wall -o pubcat pubcat.c
thales$ cp pubcat /tmp/pubcat
thales$ ll /tmp/pubcat
-rwxr-x---    1 jmayer   sai          6774 Dec 10 13:08 /tmp/pubcat
thales$ ll /home/thales/jmayer/pub/
-rw-------    1 jmayer   sai            15 Dec 10 13:08 welcome.txt
thales$ cat /home/thales/jmayer/pub/welcome.txt
Sie sind drin!
thales$ /tmp/pubcat welcome.txt
Sie sind drin!
thales$
```

Jetzt kann auch *Benutzer mayer* das Kommando *pubcat* ausführen. Da das Heimatverzeichnis von jmayer aber niemandem Zugriff erlaubt, bleiben alle Leseversuche erfolglos:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ cat /home/thales/jmayer/pub/welcome.txt
cat: /home/thales/jmayer/pub/welcome.txt: Permission denied
thales$ /tmp/pubcat welcome.txt
cat: /home/thales/jmayer/pub/welcome.txt: Permission denied
thales$
```

Der *Benutzer jmayer* setzt aber nun das *Set-User-ID-Bit* des Kommandos *pubcat*:

```
thales$ id
uid=13847(jmayer) gid=200(sai)
thales$ chmod u+s /tmp/pubcat
thales$ ll /tmp/pubcat
-rwsr-x---    1 jmayer   sai          6774 Dec 10 13:08 /tmp/pubcat
thales$
```

Das Setzen des *Set-User-ID-Bits* hat zur Folge hat, dass das Kommando *pubcat* nicht mehr mit den Benutzerrechten des Aufrufers, sondern mit den Benutzerrechten des Datei-Eigentümers ausgeführt wird. Folglich kann nun der *Benutzer mayer* mit dem Kommando *pubcat* die Datei welcome.txt lesen, obwohl er selbst immer noch keinen Zugriff auf diese Datei hat:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ cat /home/thales/jmayer/pub/welcome.txt
cat: /home/thales/jmayer/pub/welcome.txt: Permission denied
thales$ /tmp/pubcat welcome.txt
Sie sind drin!
thales$
```

Nun tritt gleich die erste *Schwachstelle* des Programms zu Tage: Der *Dateiname wird nicht überprüft*. Das ist fatal, denn nun kann einfach mittels „..“ ins Heimatverzeichnis und von dort aus ins *.ssh*-Verzeichnis gewechselt und der *private Schlüssel* ausgegeben werden:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ /tmp/pubcat ../.ssh/id_dsa
-----BEGIN DSA PRIVATE KEY-----
[...]
-----END DSA PRIVATE KEY-----
thales$
```

Es kommt noch schlimmer! Man kann sich einfach der *Metasymbole der Shell* bedienen, um den „Dateinamen“ komplett umzufunktionieren. Zwei Shell-Kommandos kann man einfach durch einen Strichpunkt voneinander trennen. Bauen wir einen Strichpunkt in den Dateinamen ein, so wird alles danach als weiteres Kommando interpretiert. Die gemäßigte Variante ist, dass wir als zweites Kommando *id* verwenden, um die Benutzerkennung, unter der das Programm ausgeführt wird, auszugeben. Wir können aber auch das Programm *sh* verwenden. Dann haben wir eine *Shell mit den Rechten des Benutzers jmayer*:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ /tmp/pubcat 'welcome.txt; id'
Sie sind drin!
uid=15157(mayer) gid=200(sai) euid=13847(jmayer)
thales$ /tmp/pubcat 'welcome.txt; sh'
Sie sind drin!
thales$ id
uid=15157(mayer) gid=200(sai) euid=13847(jmayer)
thales$ ^D
thales$ id
uid=15157(mayer) gid=200(sai)
thales$
```

Angenommen, wir hätten gar keinen Einfluss auf die Argumente des ausgeführten Programms. Dann gibt es immer noch eine gravierende Schwachstelle. Das Kommando *cat* wird nämlich in allen Verzeichnissen, die in der Umgebungsvariable *PATH* angegeben sind gesucht, da *cat* nicht mit dem *absoluten Pfad* angegeben ist. D. h., dass ein Angreifer ganz einfach ein *eigenes Kommando cat* z. B. im Verzeichnis /tmp erzeugen kann und dann die Variable PATH auf den Wert /tmp setzen kann. Danach führt das Kommando pubcat genau das Kommando cat des Angreifers aus:

```
thales$ id
uid=15157(mayer) gid=200(sai)
thales$ ll /tmp/cat
-rwxr-x---   1 mayer    sai           44 Dec 10 14:08 /tmp/cat
thales$ cat /tmp/cat
#!/bin/sh
echo "Mein Programm!"
/usr/bin/id
thales$ /tmp/pubcat welcome.txt
Sie sind drin!
thales$ PATH=/tmp
thales$ /tmp/pubcat welcome.txt
Mein Programm!
uid=15157(mayer) gid=200(sai) euid=13847(jmayer)
thales$
```

Normalerweise würde ein Angreifer natürlich in seinem eigenen cat eine Shell starten, mit der er dann unter den Benutzerrechten von jmayer arbeiten kann.

Selbst wenn man den absoluten Pfad des Kommandos cat angibt, gibt es immer noch Angriffsmöglichkeiten. Wenn z. B. das Kommando */usr/local/bin/cat* innerhalb von *pubcat.c* aufgerufen würde, so könnte ein Angreifer den IFS – eine Umgebungsvariable – neu setzen. Der IFS wird von der Shell verwendet bei der Zerlegung einer Kommandozeile in das Kommando und seine Argumente. Normalerweise enthält er Leerzeichen, Tabulator und Zeilenumbruch. Setzt man den IFS jedoch auf den Wert „/“, so würde *„/usr/local/bin/-cat“* interpretiert als der Aufruf des Kommandos „usr“ mit den Argumenten „local“, „bin“ und „cat“. Dann kann man wie vorher durch Setzen von PATH ein *eigenes Kommando usr* ausführen.

Eine weitere große Sicherheitslücke von Programm 14.1 ist, dass mittels *sprintf()* das Kommando „zusammengebaut“ wird und dabei nicht darauf geachtet wird, ob der Platz in *s* ausreicht. Man kann sich ja vorstellen, dass das Argument beliebig groß werden kann. So schreibt *sprintf()* in einen Speicherbereich außerhalb des Arrays. Der Zugriff kann hier also außerhalb der Grenzen des Arrays erfolgen (*buffer overflow*). Somit kann ein Angreifer Maschinencode in diesem Array plazieren und dann noch die Rücksprungadresse (innerhalb einer Funktion) geeignet setzen, so dass sein Code angesprungen wird. Je größer das Array dimensioniert ist, desto einfacher wird es für den Angreifer, sinnvollen Maschinencode im Array zu plazieren – dann kann es auch ein etwas längeres Programm sein. ;-) Auf dieser Basis existieren viele Angriffe. Dies ging hier jedoch viel einfacher.

Ein weiterer typischer Fehler ist, wenn dem Benutzer die Möglichkeit gegeben wird, auf den Format-String bei *printf()* Einfluss zu nehmen. Dann kann er nämlich die Formatierungsangabe *%n* unterbringen, die bewirkt, dass die Anzahl der bisher ausgegebenen Zeichen in das Argument – das als Zeiger auf eine Integer interpretiert wird – geschrieben wird! Das würde man auf den ersten Blick nicht vermuten, dass *printf()* nicht nur Argumente liest, sondern evtl. auch verändert:

Programm 14.2: printf() und die Formatierungsangabe %n (*printf.c*)

```
#include <stdio.h>

int main() {
    int n;

    printf("%d%n\n", 3, &n);
    printf("n␣=␣%d\n", n); /* Anzahl bis %n ausgegebener Zeichen */
```

```
    printf("%d%n\n", 33, &n);
    printf("n␣=␣%d\n", n); /* Anzahl bis %n ausgegebener Zeichen */

    return 0;
}
```

```
thales$ gcc -Wall printf.c
thales$ a.out
3
n = 1
33
n = 2
thales$
```

Zusammenfassung der typischen Schwachstellen von C-Programmen:

- *Indizierungsfehler bei Arrays:* Diese können einerseits direkt durch den Programmierer passieren, wenn er auf einen Index zugreift, den es gar nicht gibt. Andererseits kann so ein Fehler auch in den Funktionen *gets()*, *sprintf()*, *strcpy()*, *strcat()*, etc. passieren.
- *Unzureichende Überprüfung von Argumenten:* In unserem Fall war das eine nicht ausreichende Überprüfung eines „Dateinamens". Beim Arbeiten mit Dateien, Ausführen von Kommandos oder Arbeiten mit Systembefehlen ist immer Vorsicht geboten, wenn der Benutzer Einfluss auf die Argumente haben kann.
- *Benutzereinfluss auf dem Format-String bei printf():* Wir haben gesehen, dass *printf()* seine Argumente nicht nur zum Lesen, sondern manchmal auch zum Schreiben verwendet. Dieses unerwartete Verhalten ist u. U. fatal, wenn der Benutzer Einfluss auf den Format-String nehmen kann.
- *Benutzung eines Zeigers nach seiner Freigabe*
- *doppelte Speicherfreigabe mit free()*

Fazit: Die Vermeidung dieser Fehler ist alles andere als trivial, wie gefundene Schwachstellen in sicherheitsrelevanter Software wie etwa ssh (*secure shell*) und ssl (*secure socket layer*) belegen. Daher ist es für systemnahe Software ratsam, auf problematische Funktionen der C-Bibliothek zu verzichten und stattdessen auf sicherere Alternativen auszuweichen.

14.2 Dynamische Strings

Es ist zwar sehr einfach möglich – mittels *calloc()* bzw. *strdup()* – einen dynamischen String anzulegen, jedoch bieten die Funktionen *strcpy()*, *strcat()*, *sprintf()*, etc. keine Möglichkeit der Überprüfung, ob die allokierte Länge nicht überschritten wird – was nicht zuletzt an der fehlenden Größeninformation bei C-Arrays liegt.

Es bietet sich also die Verwendung der Bibliothek *libowfat* (http://www.fefe.de/libowfat/) an, die von Felix von Leitner nach einem Vorbild von Dan J. Bernstein nachprogrammiert und unter die GPL (GNU General Public License) gestellt wurde. Diese Bibliothek ist bei uns lokal unter */usr/local/diet* installiert.

Bei C-Arrays fehlt im Wesentlichen die Größeninformation. Dies behebt die folgende Datenstruktur für Strings in der *stralloc-Bibliothek*:

```
typedef struct stralloc {
    char* s;  /* Zeichen des Strings (i.A. ohne '\0' am Ende) */
    unsigned int len; /* Laenge des Strings (len <= a) */
    unsigned int a; /* Laenge des Arrays s */
} stralloc;
```

Hinter *s* verbirgt sich das Zeichenarray, das im Allgemeinen nicht nullterminiert ist. Dies hat den Vorteil, dass Strings auch das Null-Byte enthalten können. Die Komponente *len* enthält die momentane Länge des Strings und *a* ist die momentante Länge des Arrays *s*; also gilt folglich immer $len \leq a$.

Da C lokale Variablen nicht automatisch initialisiert, müssen diese jeweils „von Hand" initialisiert werden:

```
stralloc sa = {0};
```

Man beachte, dass durch obige Initialisierung nicht nur *sa.s*, sondern auch *sa.len* und *sa.a* auf 0 initialisiert werden.

Im Folgenden sind die wesentlichen Funktionen der *stralloc-Bibliothek* kurz beschrieben. Der Rückgabewert dieser Funktionen ist bei Erfolg 1 und sonst 0.

int stralloc_ready(stralloc* sa,unsigned int len)
: Stellt sicher, dass genügend Platz für *len* Zeichen vorhanden ist.

int stralloc_readyplus(stralloc* sa,unsigned int len)
: Stellt sicher, dass genügend Platz für weitere *len* Zeichen vorhanden ist.

int stralloc_copys(stralloc* sa,const char* buf)
: Kopiert den nullterminierten String *buf* nach *sa*.

int stralloc_copy(stralloc* sa,const stralloc* sa2)
: Kopiert *sa2* nach *sa*.

int stralloc_cats(stralloc* sa,const char* buf)
: Hängt den nullterminierten String *buf* an *sa* an.

int stralloc_cat(stralloc* sa,stralloc* sa2)
: Hängt *sa2* an *sa* an.

int stralloc_0(stralloc* sa)
: Hängt ein Null-Byte an *sa* an.

void stralloc_free(stralloc* sa)
: Gibt den von *sa* belegten Speicher wieder frei – also den Speicher von *sa->s* (und nicht den Speicher für die stralloc-Struktur).

Das folgende Programm demonstriert die Verwendung der stralloc-Bibliothek. Dieses Programm enthält auch je eine kleine Funktion, um einen stralloc-String auszugeben und eine (beliebig lange) Zeile von der Standardeingabe zu lesen.

Programm 14.3: Arbeiten mit der stralloc-Bibliothek (*stralloc.c*)

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

    /* sa auf Standardausgabe schreiben */
void print(stralloc *sa) {
    int i;
    for (i = 0; i < sa->len; i++)
        putchar(sa->s[i]);
```

```
}

    /* eine Zeile von der Standardeingabe lesen */
int readline(stralloc *sa) {
    if (!stralloc_copys(sa, "")) return 0;
    while (1) {
        if (!stralloc_readyplus(sa, 1)) return 0;
        if (fread(sa->s + sa->len, sizeof(char), 1, stdin) <= 0) return 0;
        if (sa->s[sa->len] == '\n') break;
        sa->len++;
    }
    return 1;
}

int main() {
    stralloc sa = {0};

    if (readline(&sa)) {
        print(&sa); puts("");

        printf("Laenge␣von␣sa:␣%d\n", sa.len);
        stralloc_0(&sa);
        printf("Laenge␣von␣sa:␣%d\n", sa.len);

        puts(sa.s);

        stralloc_free(&sa);
    }

    return 0;
}
```

```
thales$ gcc -Wall -I /usr/local/diet/include/ -L /usr/local/diet/lib/ \
> stralloc.c -lowfat
thales$ a.out
Meine Eingabe ...
Meine Eingabe ...
Laenge von sa: 17
Laenge von sa: 18
Meine Eingabe ...
thales$
```

Anmerkungen: Bei der Übersetzung (bzw. dem Linken) muss man dem gcc den Pfad der Include-Datei und den Pfad der Bibliothek angeben und die Bibliothek libowfat mit der *Option -lowfat* hinzubinden.

14.3 Zusammenfassung und Fazit

- *C-Arrays (und somit auch Strings) enthalten keine Größeninformation.* Dies kann leicht zu unerlaubten Zugriffen führen, die nicht abgefangen werden. Betroffen sind auch beliebte Funktionen wie etwa *strcpy()*, *strcat()*, *gets()*, *sprintf()*, etc.

- *„Traue nichts und niemandem“* ist dringend anzuraten. Deshalb sollten Eingaben aller

Art (Kommandozeilen-Argumente, Umgebungsvariablen, Standardeingabe, Dateieingabe, etc.) überprüft werden.

- *Positivlisten* (Was ist erlaubt?) sind bei der Überprüfung von Eingaben *besser und sicherer als Negativlisten* (Was ist verboten?)!
- Der *wohldefinierte Bereich einer Programmiersprache* sollte nie verlassen werden (z. B. durch den Zugriff auf nicht vorhandene Array-Elemente).
- Soweit *automatische Überprüfungen* möglich sind, sollten sie verwendet werden.
- Sind die automatische Überprüfungen nicht hinreichend (wie z. B. bei Array-Zugriff), so sollte etwa auf eine *andere Bibliothek* (z. B. stralloc-Bibliothek) ausgewichen werden.

Kapitel 15

Das Aufbau des Betriebssystems Unix

15.1 Betriebssysteme allgemein

15.1.1 Definition

Die *DIN-Norm 44300* definiert wie folgt, was ein *Betriebssystem* ist:

> Zum Betriebssystem zählen die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und die insbesondere die Abwicklung von Programmen steuern und überwachen.

Das Betriebssystem ist das *erste Programm*, welches *nach dem Starten eines Rechners* geladen wird (abgesehen von der Firmware). *Solange der Rechner in Betrieb ist*, läuft das Betriebssystem.

15.1.2 Aufgaben

Das Betriebssystem hat zwei zentrale *Aufgaben*:

- **Ressourcen-Management:**

  Das Betriebssystem kontrolliert alle Hardware- und Software-Komponenten eines Rechners und teilt sie *effizient* den einzelnen Nachfragern zu.

  Das Betriebssystem stellt Basis-Dienstleistungen (Dateizugriff, Prozessmanagement) zur Verfügung, auf denen Anwendungsprogramme aufsetzen können.

- **„Erweiterte Maschine“:** (sog. *Virtual Machine*)

  Das Betriebssystem besteht aus einer (oder mehreren) Schichten von Software, die über der „nackten“ Hardware liegen. Diese *Virtual Machine* ist einfacher zu verstehen und zu programmieren. Komplexe Hardware-Details verbergen sich hinter einfacheren und einheitlichen *erweiterten Instruktionen* (*Systemaufrufe*).

  Der Programmierer erhält vom Betriebssystem eine angenehmere Schnittstelle zur Hardware. Dabei wird von Hardware-Details wir *Speicherorganisation, I/O-Struktur, Bus-Struktur, etc.* abstrahiert. Der Programmierer muss sich nicht mehr um sämtliche maschinennahen Details kümmern.

15.1.3 Schichtenmodell

Das *Schichtenmodell* teilt die *„erweiterte Maschine“* in die folgenden Schichten ein (siehe Abbildung 15.1):

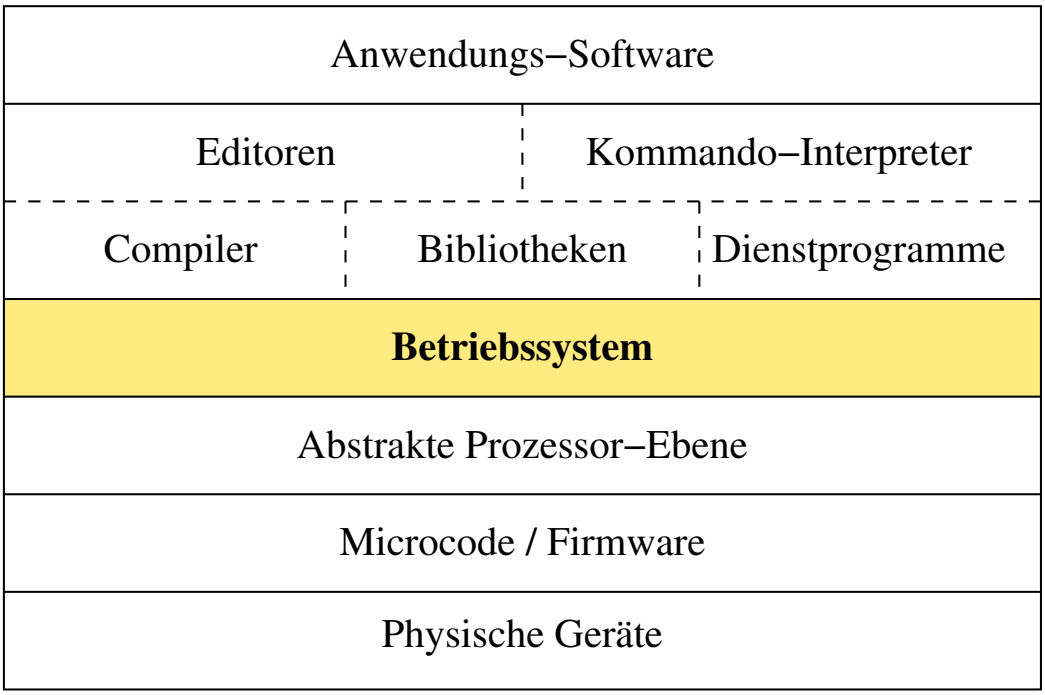


Abbildung 15.1: Das Schichtenmodell der „erweiterten Maschine“

- **Physikalische Geräte:**

  Prozessor, Festplatten, Grafikkarte, Stromversorgung, etc.

- **Microcode / Firmware:**

  Software, die die *physikalischen Geräte direkt kontrolliert* und sich teilweise direkt auf den Geräten befindet. Diese bietet der nächsten Schicht eine einheitlichere Schnittstelle zu den physikalischen Geräten. Dabei werden *einige Details* der direkten Gerätesteuerung *verborgen*.

  Beispiel: Abbildung logischer Adressen auf physikalische Adressen bei Festplatten.

- **Maschinensprache:**

  Stellt die Schnittstelle zwischen Hardware und Software dar. Je nach dem ob *RISC (Reduced Instruction Set Computer)* oder *CISC (Complex Instruction Set Computer)* zwischen 50 und 300 Instruktionen (ADD, MOVE, JUMP, etc.). Die meisten dienen zum *Kopieren von Daten* zwischen den Registern und dem Hauptspeicher, dem Ausführen von (teils bedingten) *Sprüngen* und dem Ausführen von *arithmetischen und Vergleichs-Operationen*.

- **Betriebssystem:**

  Das Betriebssystem vermittelt zwischen Anwender bzw. Anwenderprogramm und Hardware. Es verbirgt die Komplexität der Hardware vor dem Programmierer und bietet angenehmere Instruktionen für das Programmieren (*open(), lseek(), read(), close()* statt „move head of disc 1 to track 231, lower head, ...“).

- **System-Software:**

  Compiler, Editoren, Kommando-Interpreter, Dienstprogramme – gehören nicht zum Kern des Betriebssystems, werden aber meist vom Hersteller des Betriebssystem(-Kern)s mitgeliefert.

- **Anwendungs-Software:**

  Von Benutzern bzw. für Benutzer zur Lösung ihrer Probleme entwickelt (Beispiel: Textverarbeitungsprogramm).

15.2 Unix-Schalenmodell

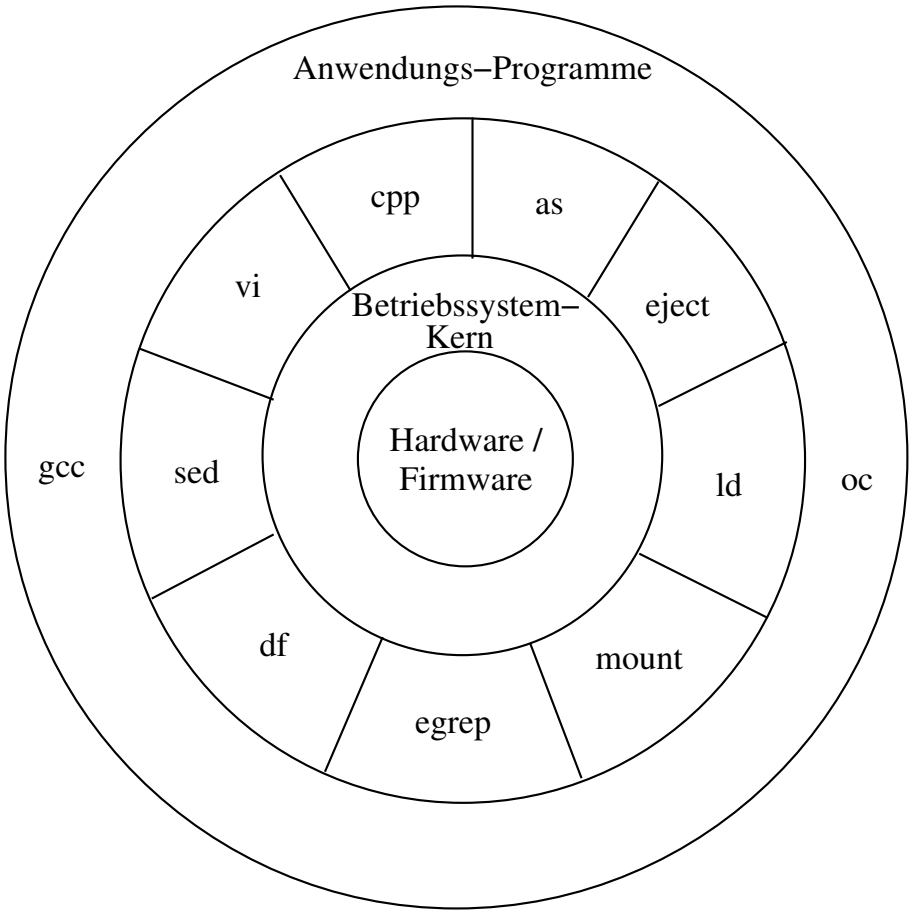


Abbildung 15.2: Das Unix-Schalenmodell

Abbildung 15.2 zeigt das *Unix-Schalenmodell*. Die wesentlichen Punkte sind:

- **Abstraktion:**

  Das Betriebssystem übernimmt die Kommunikation mit der Hardware. Es isoliert die Programme von Hardware-Details. Der Programmierer kann mit abstrakten Objekten (z. B. Filedeskriptor) statt mit hardware-nahen Details (z. B. Plattenadressen) arbeiten. Dadurch werden Programme hardware-unabhängig und somit leichter portierbar.

- **Interaktion:**

  Programme kommunizieren mit dem Betriebssystem-Kern über klar definierte Systemaufrufe (*System Calls*), um Dienstleistungen zu erhalten und Daten auszutauschen.

- **Kombination:**

  Benutzerprogramme und (System-)Kommandos befinden sich in derselben Schicht. Sie können aufeinander aufbauen, sich gegenseitig aufrufen oder Daten miteinander austauschen.

15.3 Interner Aufbau von Unix

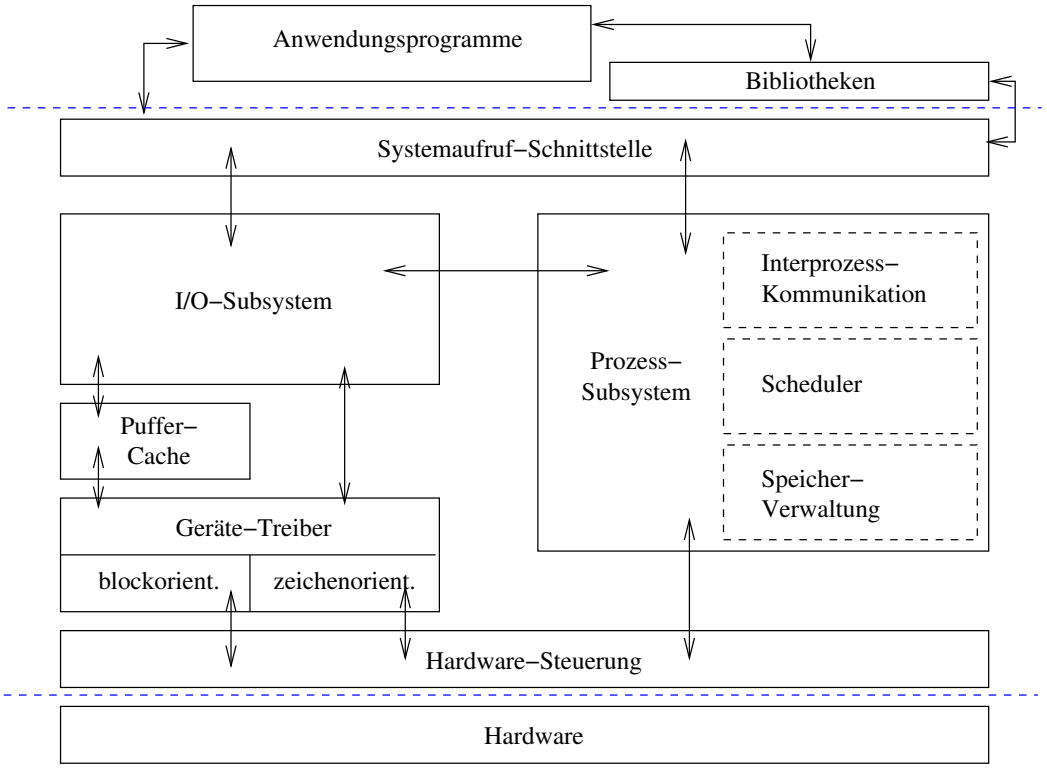


Abbildung 15.3: Der interne Aufbau von Unix

Der interne Aufbau des Betriebssystems Unix ist in Abbildung 15.3 veranschaulicht.

- **Systemaufrufe (System Calls):**

  Die Systemaufrufe bilden die *Schnittstelle* zwischen dem Betriebssystem-Kern und den Anwendungsprogrammen. Über sie können die *Dienstleistungen des Betriebssystems* in Anspruch genommen werden. In einem Programm sind Systemaufrufe einfach *Funktionsaufrufe*.

- **Subsysteme:**

  Intern kann man den Betriebssystem-Kern in *zwei Subsysteme* aufteilen – das *Ein-/Ausgabe-Subsystem* (oder *I/O-Subsystem* oder *I/O Subsystem*) und das *Prozess-Subsystem*.

Ersteres ist verantwortlich für alles, was mit Dateien zu tun hat. Letzteres ist verantwortlich für Programmausführung, Speicherzuteilung, etc. Die Systemaufrufe lassen sich je einem dieser beiden Subsysteme zuordnen.

- **Zugriffskontrolle:**

  Die *Systemaufrufe* bieten den *einzig möglichen Zugang zu den Ressourcen* (Hardware). Es gibt keinen direkten Zugriff der Benutzerprogramme auf die Ressourcen (CPU, Hauptspeicher, Festplatten, etc.).

- **Transparenz:**

  Die Dienstleistungen des Betriebssystem-Kerns sind transparent. (*Beispiel:* Alles ist eine Datei: „Echte Datei“, Gerät, Pipe, Netzwerkverbindung, etc.)

Kapitel 16

Das I/O-Subsystem

16.1 Dateien

16.1.1 Was ist eine Datei?

Was ist eigentlich eine Datei? Unter UNIX ist eine Datei wie folgt definiert: „Every file is a sequence of bytes.“ Also: *Eine Datei ist einfach eine Folge von Bytes*.

Zu einer Datei gehören

- ein oder auch mehrere *Namen*,
- der *Inhalt* und *Aufbewahrungsort* (Menge von Blöcken auf der Platte, etc.) und
- *Verwaltungsinformationen* (Besitzer, erlaubter Zugriff, Zeitstempel, Länge, Dateityp, etc.).

UNIX verlangt und unterstellt bei regulären (d. h. gewöhnlichen) Dateien *keinerlei Struktur* und unterstützt auch keine. Die Konzepte variabel oder konstant langer Datensätze (Records) sind im Kernel von UNIX nicht implementiert. *Eine Datei wird abstrakt als Datenstrom repräsentiert.*

Neuartig unter UNIX ist, dass praktisch *alle Objekte des Systems durch Dateien repräsentiert* und somit abstrahiert sind. Hinter einer Datei kann sich Plattenplatz, ein Gerät, eine Pipeline (zur Prozesskommunikation), etc. „verstecken“.

16.1.2 Aufgaben des Betriebssystems

Im Zusammenhang mit Dateien hat das UNIX-Betriebssystem folgende Aufgaben:

- Speichern und Wiederbeschaffen von Benutzerdaten
- Bereitstellen von Platten-Speicherplatz
- Verwalten von freiem Plattenplatz
- Zugriff auf Geräte ermöglichen
- Kontrolle der Zugriffsrechte auf Dateien

16.1.3 Dateioperationen

Der Zugriff auf Dateien erfolgt immer einheitlich mit denselben Dateioperationen (öffnen, schließen, lesen, schreiben, etc.) via Systemaufruf.

Beispiele für relevante Systemaufrufe:

chdir	Katalog wechseln
chmod	Zugriffsrechte ändern
chown	Besitzer ändern
close	Datei schließen
dup	Filedeskriptor duplizieren
dup2	Filedeskriptor duplizieren
fnctl	Eigenschaften einer offenen Datei ändern
ioctl	Ähnlich zu fcntl, aber Geräte-spezifisch
link	Neuer Name für bestehende Datei
lseek	Positionieren in einer Datei
mknod	Besondere Dateien erzeugen
mount	Dateisystem in bestehendes „einhängen“
open	Öffnen/Erzeugen einer Datei
pipe	Namenlose Pipeline erzeugen
read	Lesen aus einer Datei
stat	Attribute einer Datei lesen
umount	Dateisystem „aushängen“
unlink	Namen entfernen
write	Schreiben in eine Datei

16.1.4 Dateitypen

Obwohl Dateien als Abstraktionen für alle Objekte des Systems dienen, benötigt UNIX nur wenige verschiedene *Dateiarten* (*file types*):

- *Gewöhnliche Datei* (*ordinary file* oder *regular file*):
  Container für jede Art von Daten (sowohl Text als auch binär!)

- *Verzeichnis* bzw. *Katalog* (*directory file*):
  Verzeichnis für die Namen von Dateien und zugehörige Verweise auf „Verwaltungsinformation“ (*Inode*). Ein Verzeichnis hat immer eine fest vorgegebene Struktur.

- *Gerätedatei* (*device special file*: character special file / block special file):
  Geräte und Hardware allgemein; zu Dateien abstrahiert (Terminal, Drucker, Platten, Hauptspeicher, etc.)

- *Symbolischer Link* (*symbolic link*):
  Verweis auf den Pfadnamen einer anderen (u. U. nicht existierenden) Datei

- *Benannte Pipeline* (*named pipe*; *FIFO*):
  Zur unidirektionalen Prozesskommunikation

- *Socket*:
  Kommunikationsstruktur; insb. zur Kommunikation in Rechnernetzen

- und einige wenige mehr (System V, BSD)

16.1.5 Gerätedateien

Es gibt zwei Arten von Geräten:

- **Zeichengeräte** (*character devices*):
  Dies sind Geräte auf die ungepuffert zugegriffen werden kann, gemäß dem Modell „unformatierter sequentieller Bytestrom“.
  Beispiele: Terminal, Drucker

- **Blockgeräte** (*block devices*):
  Dies sind Geräte, die Daten in beliebig adressierbaren Blöcken speichern und übertragen können.
  Beispiel: Festplatte

Das I/O-Subsystem benutzt einen Puffermechanismus beim Zugriff auf Daten, die sich auf einem Blockgerät (z. B. Festplatte) befinden, den sog. *Puffer-Cache*.

Die Module des Kernels, welche unmittelbar auf die physikalischenn Geräte zugreifen, heißen *Gerätetreiber*. Sie haben eine *Geräte-unabhängige Schnittstelle* „nach oben“ zu den übrigen Kernel-Routinen und eine *Geräte-abhängige Schnittstelle* „nach unten“ zur Hardware hin.

16.2 Dateisysteme

16.2.1 Arten von Dateisystemen

Man unterscheidet prinzipiell drei Arten von Dateisystemtypen:

- **Plattenbasierte Dateisysteme:**
  Die wichtigsten Typen (im folgenden weiter betrachtet) sind die plattenbasierenden wie z. B. das *UFS* (*Unix File System*), *FAT* (*File Allocation Table* – von DOS oder Windows) und *NTFS* (*New Technology File System*), die auf Festplatten, Disketten oder CDs (ISO-Dateisystem) angelegt werden.

- **Netzwerk-Dateisysteme:**
  Andere Dateisysteme basieren auf Netzwerken; dazu zählen die Typen *NFS* (*Network File System*) oder *RFS* (*Remote File System*). *NFS* ist das Dateisystem, welches im *WiMa-Netz (!)* (z. B. für Heimatverzeichnisse) Verwendung findet.

- **Pseudo-Dateisysteme:**
  Schließlich gibt es eine Reihe spezieller Dateisystemtypen, die unter dem Begriff *Pseudodateisysteme* (*Pseudo Filesystems*) zusammengefasst werden.
  Beispiel: Das Prozess-Dateisystem (*procfs*)

16.2.2 Netzwerk-Dateisysteme

16.2.2.1 Allgemeines

Über ein Netzwerk verteilte Dateisysteme basieren auf dem sogenannten *Client/Server-Prinzip*. Der Server ist dabei der Rechner, der die Dateisysteme, die er lokal verwaltet, anderen Rechner verfügbar macht. Der Client ist der Rechner, der diese Leistung auf einem Server nutzt. Mischformen sind durchaus üblich: Rechner, die selbst anderen Rechnern Teile ihres Dateisystems zur Verfügung stellen, selbst auch welche von anderen Rechnern erhalten (Beispiel: unsere Server thales, turing, etc.). Es können nur lokale Dateisysteme weitergegeben werden. Abbildung 16.1 zeigt ein Beispiel für ein verteiltes Dateisystem (nach [Handschuch 1993]).

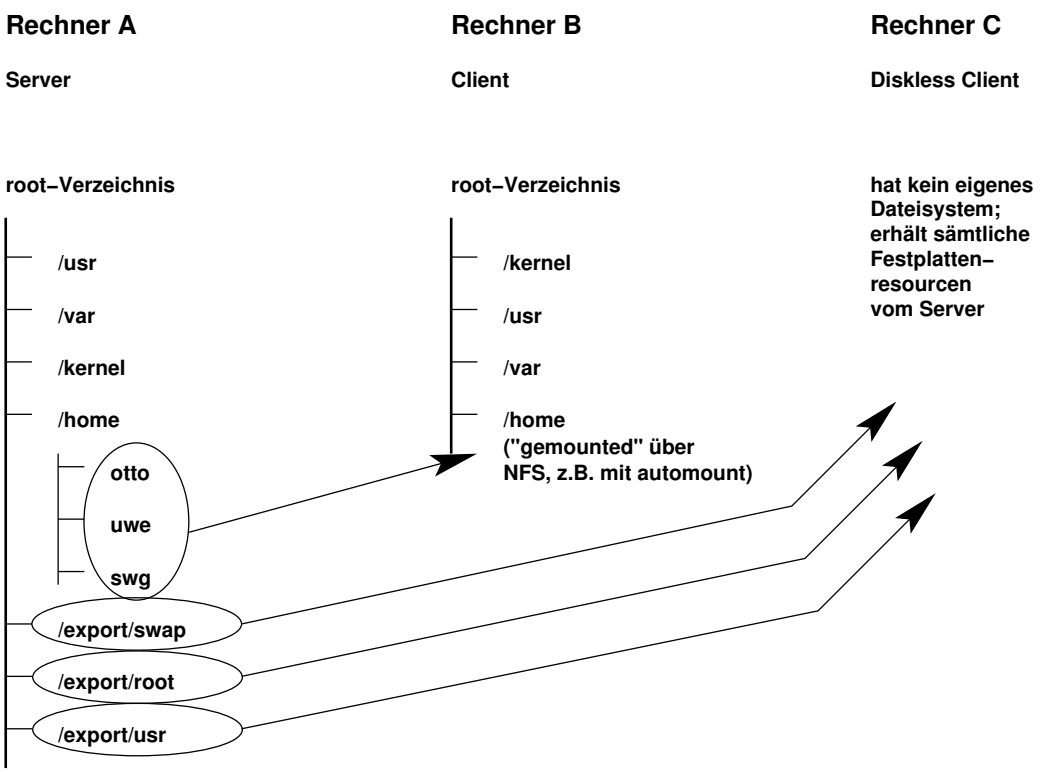


Abbildung 16.1: Beispiel für ein Netzwerk-Dateisystem

16.2.2.2 Network File System (NFS)

Das *Network File System* (*NFS)* wurde von *Sun Microsystems Inc.* für die Kopplung von Rechnern mit unterschiedlicher Hardware und unterschiedlichem Betriebssystem entwickelt. Damit sollte über das Netzwerk der Zugriff eines Rechners auf die Festplatten eines anderen Rechners ermöglicht werden: Daten lesen und verarbeiten, auch wenn diese in einem Format abgelegt sind, die nur das dortige Betriebsystem „kennt". Die notwendige Umsetzung der Datenstruktur wird vom *XDR-Protokoll* (*eXternal Data Representation*) vorgenommen; die Steuerung des Zugriffs erfolgt über *RPC* (*Remote Procedure Call*). Der Zugriff wird transparent durchgeführt, also ohne Eingabe von besonderen Befehlen. Systemaufrufe wie *read()* oder *write()*, die einem remote-System gelten, werden *auf dem Client wie lokale Operationen behandelt*, aber über das Netzwerk weitergeleitet.

Die NFS-Implementierung von Sun unterstützt sog. zustandslose Server:

- Das Öffnen einer Datei mittels *open()* wird zunächst im Client zurückgehalten, falls sich diese Datei auf einem remote-Dateisystem befindet.
- Die Übertragung der für *open()* spezifischen Daten geschieht erst bei einem nachfolgenden *read()*- oder *write()*-Aufruf, zusammen in einer Transaktion.
- Ein „Absturz" des remote-Dateisystems zwischen *open()* und z. B. *read()* führt so zu keinen Problemen, die Zustände sind lokal.
- NFS nimmt keine Verwaltung des Zugriffs von Clients vor, den Clients ist ebenso der Zustand des Servers nicht bekannt.
- Die Anforderungen nach einem NFS-Zugriff durch einen Client sind alle unabhängig voneinander, und mit sämtlichen Informationen ausgestattet, die der Server für

die Erfüllung der Aufgabe braucht.

- Problematische Zustände pflanzen sich nicht im Netzwerk fort.
- Dadurch ist eine konsistente Arbeitsweise des Servers gewährleistet, da ein Auftrag immer vollständig abgearbeitet werden kann, sobald er über das Netz beim Server eintrifft.

Funktionsweise von NFS-Anforderungen:
Wird von einem Prozess (NFS-Client) eine Zugriffsanforderung auf eine Datei abgesetzt, so wird dies als Betriebssystemaufruf an die Dateisystem-Schnittstelle interpretiert. Hier wird mit Hilfe des *VFS* (*Virtual File System*) festgestellt, wie die gewünschte Datei erreichbar ist: lokal oder über das Netz. Wenn lokal erreichbar, so wird sie mit Hilfe des entsprechenden Plattentreibers von einer lokalen Platte geladen. Ist die Datei nur über das Netz erreichbar, so wird der Betriebssystemaufruf des Client über NFS an die Dateisystemschnittstelle des Servers weitergeleitet und dort behandelt. Auf dem Server ist die Datei dann lokal erreichbar, die Datei kann gelesen und das Ergebnis an den Client über das Netz zurückgesandt werden. Dabei werden immer nur die Daten übertragen, die auch direkt zur Ausführung eines Befehls benötigt werden. Bei einem Zugriff auf eine große Datei wird diese bei einem Zugriff nicht in voller Länge übertragen, sondern nur der Teil, der für die Verarbeitung gerade benötigt wird.
Konsequenzen:

- Über NFS können entfernte Dateien schnell gelesen und geschrieben werden, andere Operationen wie das *Sperren von Dateien* (*File Locking*) sind tunlichst zu vermeiden, da das NFS-Protokoll selbst kein Sperren von Dateien vorsieht.
- Änderungen in Dateien wie in deren Inodes erfolgen zeitverzögert – so kann es durchaus sein, dass zwei verschiedene Prozesse ein- und dieselbe Datei Löschen können, ohne dass einer von ihnen eine Fehlermeldung „Datei nicht vorhanden“ bekommt!
- Das *Sperren von Dateien* wird bei NFS über den *NFS lock daemon* realisiert; dies ist ein Prozess, der im Auftrag von Prozessen auf anderen Rechnern das Sperren von Dateien übernimmt. (Bei der Verwendung von *lockf()*, das bei entfernten Dateien den NFS lock daemon verwendet, kann es zu Prozessen kommen, die temporär – selbst mit `kill -9` – nicht beendet werden können!)

16.2.2.3 Remote File System (RFS)

Das *RFS* (*Remote File System*) ist ähnlich dem NFS. Es kann im Gegensatz dazu aber nur in homogenen Netzen verwendet werden (alle Systeme müssen Unix System V als Betriebssystem haben, da keine Datentyp-Konvertierungen durchgeführt werden). Dieses System von *Sun Microsystems Inc.* hat sich allerdings in der Praxis nicht durchgesetzt und scheint vom Markt verschwunden zu sein.

16.2.2.4 AFS

AFS ist ein verteiltes Dateisystem, das Rechnern in einem Netzwerk ermöglicht, effizient ihre Dateisystem-Ressourcen zu teilen. Es basiert auf einem verteilten Dateisystem, das ursprünglich als „*Andrew File System*“ an der *Carnegie Mellon Universität* für den Einsatz als hochschulweites Dateisystem für ca. 2000 Arbeitsplätze entwickelt wurde. „Andrew“ war dort die Bezeichnung eines entsprechenden Forschungsprojekts. Das System wird (im Gegensatz zu NFS) inzwischen kommerziell weiterentwickelt und vertrieben (Fa. Transarc) und ist für alle gängigen UNIX-Plattformen verfügbar. Die AFS-Entwicklung ist als Basis

für das *Distributed File System* (*DFS*) in den OSF/DCE-Standard eingebracht worden (*OSF*: *Open Software Foundation*, *DCE*: *Distributed Computing Environment*).

Der einheitliche Namensraum von AFS erlaubt es, Dateien unter gleichem Namen von allen Rechnern anzusprechen. Weiterhin ist es möglich, dass die Administrationseinheiten (eine Abteilung oder andere organisatorische Einheit), *Zellen* (*cells*) genannt, ihre lokalen Datenräume in einem globalen Datenraum zuammenführen. Damit können Benutzer vollkommen transparent auf Datenbestände auf entfernt liegenden Rechnern zugreifen – ohne Kenntnisse über die Lokation der Rechner, nur über den entsprechenden Pfadnamen. AFS folgt ebenfalls dem *Client/Server-Konzept*.

16.2.3 Pseudo-Dateisysteme

16.2.3.1 Das tmpfs-Dateisystem

Ein temporäres Dateisystem kann im Hauptspeicher des Rechners (und damit lokal!) angelegt werden. Diese Art von Dateisystem wird oft auch als *RAM-Disk* bezeichnet. Der Katalog */tmp* wird typischerweise als *tmp-Dateisystem* (oder auch *tmpfs*) angelegt.

Dieses Dateisystem kann wie ein gewöhnliches Dateisystem auch mit Dateien belegt und mit Unterkatalogen organisert werden. Man beachte, dass auf diese Dateien der Zugriff lokal und nicht über NFS erfolgt! Zu beachten ist aber auch, dass alles in diesem Dateisystem nach einem Neustart des Rechners oder nach einem *umount* nicht mehr vorhanden ist, entsprechend der Philosophie „temporär“. Man kann es aber sehr wohl zur Synchronisation mittels *link* oder *open* (also Sperren durch Nutzen von dateiorientierten Systemaufrufen) nutzen, weil kein NFS-Zugriff erfolgt!

Der Vorteil des *tmp-Dateisystems* liegt in der *Performance* (Hauptspeicherzugriff!).

16.2.3.2 Das proc-Dateisystem

Das *proc-Dateisystem* (kurz: *procfs*) dient zur Verwaltung von Prozessen. Es ist ein virtuelles Dateisystem, d. h. es belegt keinen „echten“ Platz auf der Platte, sondern existiert im Hauptspeicher. Die Prozesse sind nach ihrer Prozess-ID (PID) geordnet darin abgelegt. Programme wie *ps* greifen darauf zu.

16.2.4 Das Unix-Dateisystem (UFS)

16.2.4.1 Prinzipieller Aufbau

Ein UNIX-System enthält in der Regel mehrere Unix-Dateisysteme, von denen eines das *Wurzeldateisystem* (*root file system*) ist, welches beim Wurzelkatalog „/“ eingehängt ist.

Abbildung 16.2 zeigt die Grobstruktur eines Unix-Dateisystems mit den folgenden

Abbildung 16.2: Grobstruktur des Unix-Dateisystems

wesentlichen Elementen:

- **Bootblock**:
  Beim Wurzeldateisystem enthält dieser Block den sog. *bootstrap code*, der beim „Hochfahren“ des Betriebssystems als erstes geladen wird. Bei anderen Dateisystemen ist er zwar vorhanden, wird aber nicht verwendet.

- **Superblock**:
  Enthält die folgenden Verwaltungsinformationen des Dateisystems:
  - Größe des Dateisystems
  - Anzahl freier Blöcke
  - Liste der freien Blöcke
  - Zeiger auf den nächsten freien Block in der Liste der freien Blöcke (hier: Zeiger = Index!)
  - Größe der Inode-Liste
  - Anzahl freier Inodes
  - Liste der freien Inodes
  - Zeiger auf nächste freie Inode in der Liste der freien Inodes (hier wiederum: Zeiger = Index!)
  - „Sperr-Felder" für Liste der freien Blöcke / Inodes
  - Anzeigefeld, ob Superblock verändert wurde
- **Inode-Liste**:
  Zu jeder Datei gehört genau eine *Inode* und eine (fast) beliebige Anzahl an Datenblöcken. Die Inode enthält die Verwaltungsinformationen, welche zu dieser Datei gehören. Eine besondere Inode ist die *Wurzel-Inode* (*root inode*), die den Wurzelkatalog repräsentiert und beim Zusammenfügen (⇝ *mount*) von Dateisystemen wichtig ist.
- **Datenblöcke**:
  Ein Datenblock kann entweder zur Adressierung (⇝ *indirekte Adressierung*) oder zum Speichern von *Dateiinhalt* (auch *Verzeichniseinträge*) verwendet werden.

Alle anderen Dateisysteme (außer dem Wurzel-Dateisystem) werden explizit mit dem Kommando *mount* – normalerweise dem Superuser vorbehalten – in das Wurzel-Dateisystem eingefügt (oder in bereits so eingefügte Dateisysteme eingefügt). Deshalb muss beim Arbeiten z. B. mit Disketten das darauf enthaltene Dateisystem (i. A. vom Typ FAT) explizit „ge*mount*et" werden. Den Wurzelkatalog eine Dateisystems kann man nur auf einen Katalog (⇝ *mount point*) eines bereits „ge*mount*eten" Dateisystems „*mount*en". Mit *umount* kann ein Dateisystem wieder ausgehängt werden, wenn es nicht mehr in Benutzung und nicht das Wurzel-Dateisystem ist.

Aus Performance-Gründen werden Schreiboperationen über einen Puffer im Hauptspeicher abgewickelt. Das System synchronisiert selbständig in gewissen Zeitabständen den Inhalt dieses Puffer mit den Dateisystemen auf den Festplatten – dies geschieht mit dem Kommando *sync*.

Das Kommando *df* gibt die Anzahl der freien Plattenblöcke und Indexverweise in angemeldeten Dateisystemen oder Verzeichnissen aus. Das Kommando *du* (*disk usage*) gibt die Plattenbelegung durch Dateien in Einheiten von 512-Byte-Blöcken an. Bei Verwendung der Option „-k" wird die Dateigröße in Kilobytes angegeben. (Weitere Infos zu diesen Kommandos gibt's wie immer auf den Manual-Seiten.)

Beispiel:

```
chomsky$ ls
kap.tex  pics  progs
chomsky$ du -k
4       ./progs
36      ./pics
64      .
chomsky$
```

16.2.4.2 Inodes

Die Datenstruktur *Inode* ist die interne Repräsentation einer Datei. Die *Inode-Nummer* ist ein Index in der Inode-Liste und ist die *eindeutige Identifikation* einer Datei (innerhalb eines Dateisystems), *nicht* der Dateiname.

In der *Inode* sind sind sämtliche Informationen enthalten, die eine Datei beschreiben (siehe z. B. *Kommando ls -lai*):

Inode-Inhalt	**Beispiel**
Eigentümer (owner)	swg
Gruppe (group)	sai
Typ (type)	„normale“ Datei (regular file)
Rechte (rights)	`rw-r--r--`
Letzter Lesezugriff auf den Dateiinhalt (last access)	Wed Jun 21 09:25:11 2000
Letzte Änderung am Dateiinhalt (last modification)	Sun May 21 11:50:15 2000
Letzte Änderung an der Inode (last change)	Wed Jun 21 05:24:20 2000
Anzahl der Dateinamen (link count)	2
Größe (size)	600 Bytes
Blockadressen (block addresses)	...

Adressierung Die Datenblöcke, die zu einer Datei gehören werden direkt bzw. (bis zu dreifach) indirekt adressiert. In der Inode selbst gibt es eine Reihe Einträgen für direkte Adressierung und je einen Eintrag für einfach, zweifach und dreifach indirekte Adressierung (siehe Abbildung 16.3). Bei indirekter Adressierung stehen in Datenblöcken, sog.

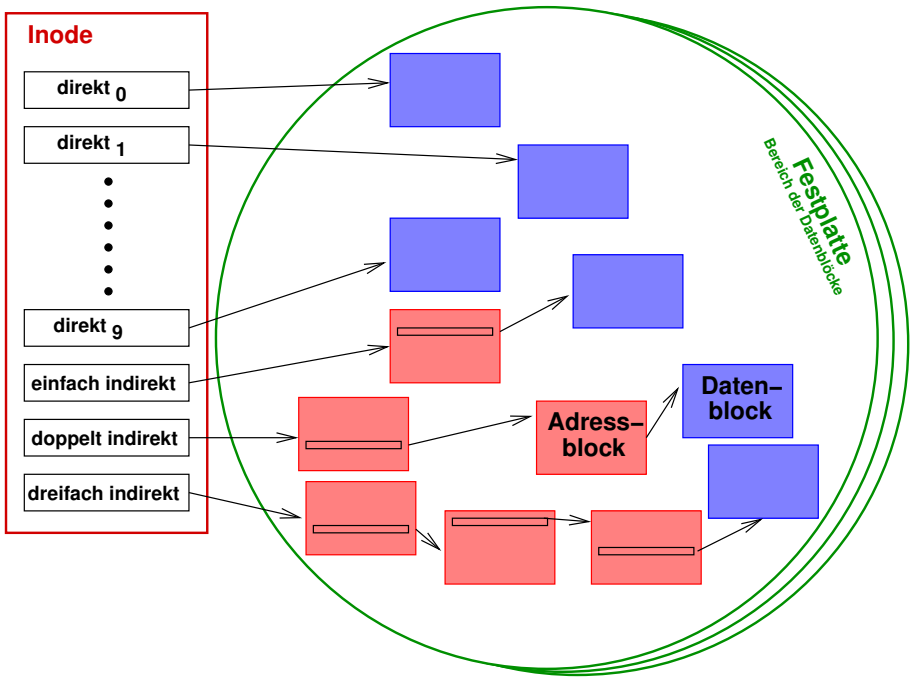


Abbildung 16.3: Adressierung der Datenblöcke

Adressblöcke, wiederum Adressen weiterer Daten- oder Adressblöcke. (Datenblöcke sind im Prinzip Arrays von Blockadressen.)

Durch die einfach, doppelt und dreifach indirekte Adressierung können – je nach gewählter Blockgröße – theoretisch Dateigrößen von bis zu 100 GB erreicht werden. Die eigentliche *Beschränkung* liegt allerdings in der *Angabe der Dateigröße*: Mit 32 Bits können nur Dateigrößen bis 4 GB spezifiziert werden! (Ab Solaris 8 wird die Dateigröße mit 64 Bit beschrieben und dadurch könnten die Dateigrößen theoretisch im Terabytebereich liegen.)

Zugriffsschutz

- **Benutzergruppen**:
  Zur Identifikation eines Benutzers gehört der *Benutzername* (*user name*) und der *Name der Gruppe* (*group name*).

  Während der Benutzername pro System eindeutig ist, fassen die Gruppen jeweils mehrere Benutzer (zum Beispiel alle Mitarbeiter eines bestimmten Projekts oder alle Mitarbeiter einer Abteilung) zu einer Einheit zusammen.

  Ein Benutzer kann eine eigene Gruppe bilden oder zusammen mit anderen Benutzern in einer oder in mehreren verschiedenen Gruppen eingetragen sein. Die Gruppenzuordnung jedes Benutzers lässt sich dynamisch verändern. Die jeweils aktuelle Gruppenzugehörigkeit wirkt sich auf die Zugriffsrechte des Benutzers aus.

- **Benutzerklassen**:
  Die Zugriffsrechte für alle Objekte (d. h. Dateien) des Dateisystems sind nach drei Klassen gestaffelt und für jede Klasse individuell definierbar.

  - Klasse 1: *Eigentümer* (*user*)
    Die Rechte des Besitzers der Datei
  - Klasse 2: *Gruppe* (*group*)
    Die Rechte für die Mitglieder in dieser Gruppe (außer dem Besitzer der Datei)
  - Klasse 3: *Andere* (*other*)
    Die Rechte aller nicht durch die Klassen 1 und 2 erfassten Benutzer

  Wie die Benutzer des Systems haben auch alle Dateien einen Benutzernamen und eine Gruppe. Der Benutzername definiert den Besitzer. Der Gruppenname bestimmt die Gruppe für den Zugriffsschutz.

- **Individueller Schutz**:
  Der Schutzmechanismus in UNIX ermöglicht das individuelle Festlegen von Zugriffsrechten für jedes Objekt (d. h. Datei) im Dateibaum. Der Besitzer kann bei jedem Objekt die Rechte für jede der drei Benutzerklassen separat festlegen – mit dem *Kommando chmod*.

  Für jede der drei Benutzerklassen kann jedes der folgenden drei *Zugriffsrechte* gesetzt bzw. nicht gesetzt sein:

  - **r** (read)
    Berechtigt zum Lesen einer Datei oder der Einträge eines Katalogs.
  - **w** (write)
    Berechtigt zum Verändern einer Datei oder eines Katalogs. Bei Katalogen erlaubt dies primär das Hinzufügen von neuen Einträgen. Details zum Löschen von Einträgen s.u.
  - **x** (execute)
    Berechtigt zum Ausführen einer Datei oder zum Wechseln in einen Katalog (*Kommando cd*).

- **Zugriff auf eine Datei**:
  Ein Prozess (= Ausführung eines Programms) hat vier IDs:

  - reale Benutzer-ID (real user ID)
  - real Gruppen-ID (real group ID)
  - effektive Benutzer-ID (effective user ID)
  - effektive Gruppen-ID (effective group ID)

  Für den Zugriff auf Dateien sind die effektiven IDs wichtig.

  Ein Prozess erhält nun unter folgenden Bedingungen Zugriff auf eine Datei:

  1. Ist die effektive Benutzer-Id 0 (superuser), so ist ein Zugriff immer erlaubt.
  2. Stimmt die effektive Benutzer-ID des Prozesses mit dem Besitzer der Datei überein und sind die Zugriffsrechte entsprechend dem Zugriff (read, write) richtig gesetzt, so ist Zugriff möglich.
  3. Stimmt die effektive Benutzer-ID des Prozesses *nicht* mit dem Besitzer der Datei überein
     und
     stimmt die effektive Gruppen-ID des Prozesses mit der Gruppen-ID des Besitzers der Datei überein
     und
     und stimmen Zugriffswunsch und Zugriffsrechte überein,
     so ist der Zugriff erlaubt.
  4. Stimmt die effektive Benutzer-ID des Prozesses *nicht* mit dem Besitzer der Datei überein
     und
     stimmt die effektive Gruppen-ID des Prozesses *nicht* mit der Gruppen-ID des Besitzers der Datei überein
     und
     und stimmen Zugriffswunsch und Zugriffsrechte für Andere überein,
     so ist der Zugriff erlaubt.

- **Konsequenzen**:
  Dieser Mechanismus kann sowohl einzelne Dateien als auch ganze Teilbäume vor dem Zugriff einiger, vieler oder aller Benutzer schützen.

  In früheren UNIX Versionen genügte bereits das Schreib-Recht für einen Katalog, um alle darin befindlichen Dateien löschen zu können. In neueren UNIX Versionen kann der Besitzer des Katalogs durch Setzen des sog. *Sticky-Bits* (mit dem Kommando *chmod +t*) verlangen, dass zusätzlich mindestens eine der folgenden Bedingungen erfüllt ist, bevor ein Benutzer eine Datei aus dem Katalog löschen kann:

  - Benutzer ist Besitzer der Datei
  - Benutzer ist Besitzer des Katalogs
  - Benutzer hat Schreib-Recht auf die Datei
  - Benutzer ist Super-User

  Will der Katalogbesitzer diese Sicherheitsmaßnahme aktivieren, muss er das Kommando *chmod +t* auf seinen Katalog anwenden.

  Der *Superuser* ist ein besonders ausgezeichneter Benutzer (Benutzername *root*, id=0), bei dem sämtliche Mechanismen des Zugriffsschutzes *nicht* greifen.

Formen einer Inode Abhängig vom aktuellen Speicherort gibt es die Datenstruktur „Inode“ in zwei verschiedenen Ausprägungen:

- *Arrayelement in der Inode-Liste* eines Dateisystems
- Inode als *Element der Kernel-Inode-Tabelle* im Hauptspeicher (*in-core inode*)

Beide Ausprägungen sind funktional identisch, nur im Aufbau etwas unterschiedlich.

Zusätzliche Einträge der *in-core inode*:

- Status
  - Inode gesperrt
  - Prozess wartet auf Freigabe
  - Inhalt ist anders als bei *disk inode*
  - Datei ist ein *mount point*
- Logische Geräte-Nummer
- Eigene *Inode-Nummer*
- Referenz-Zähler – gibt die Zahl der aktiven Instanzen dieser Datei an (⇝ in der OFT, kommt noch!)

Auf diesen (öffentlichen) Teil der Inode (siehe */usr/include/sys/stat.h*) kann man vom Programm aus zugreifen:

```
/* sys/stat.h: stat structure, used by stat(2) and fstat(2) */
struct stat {
  dev_t   st_dev;   /*device number*/
  ino_t   st_ino;   /*inode number*/
  mode_t  st_mode;  /*mode and type of file*/
  nlink_t st_nlink; /*number of links to file*/
  uid_t   st_uid;   /*user ID of the file's owner*/
  gid_t   st_gid;   /*group ID of the file's group*/
  dev_t   st_rdev;  /*ID of (raw)device if block/char special*/
  off_t   st_size;  /*file size in bytes*/
  time_t  st_atime; /*time of last access*/
  time_t  st_mtime; /*time of last data modification*/
  time_t  st_ctime; /*time of last file status change*/
};
```

Die Komponente *st_mode* enthält sowohl den Dateityp, als auch die Zugriffsrechte – bitweise kodiert in einer Integer (siehe Abb. 16.4 und Abb. 16.5).

Außer den Zugriffsrechten und dem Dateityp gibt es noch drei weitere Bits:

- **Set-UID-Bit** (UID = user id)
  Wir ein Programm ausgeführt, bei dem dieses Bit gesetzt ist, so erhält der Prozess (= Ausführung des Programms) die effektive Benutzer-ID (effective user id) des Besitzers der Programm-Datei. (Beispiel: Kommando *passwd*)
- **Set-GID-Bit** (GID = group id)
  Hier muss man zwischen Verzeichnissen und „normalen“ Dateien unterscheiden. Handelt es sich um eine „normale“ Datei, also um ein Programm, so wird – analog zum Set-UID-Bit – bei der Ausführung die effektive Gruppen-ID auf die Gruppe der Programm-Datei gesetzt. Im Falle eines Verzeichnisses erhalten alle in diesem Verzeichnis angelegten Dateien die Gruppe des Verzeichnisses als Gruppe.

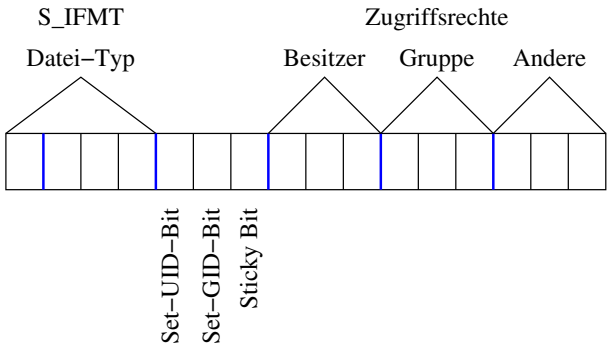
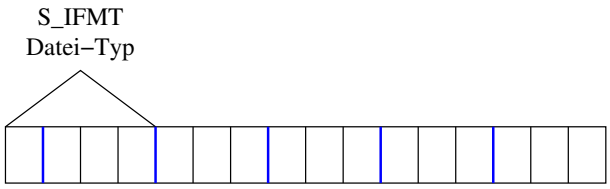


Abbildung 16.4: Kodierung der Komponente `st_mode`



1	1	0	0	Socket
1	0	1	0	symbolischer Link (symbolic link)
1	0	0	0	"normale" Datei (regular file)
0	1	1	0	Blockorient. Gerät (block device)
0	1	0	0	Verzeichnis (directory)
0	0	1	0	Zeichenorient. Gerät (char device)
0	0	0	1	Benannte Pipeline / FIFO

Abbildung 16.5: Kodierung des Dateityps in `st_mode`

- **Sticky-Bit**
  Bewirkt bei Verzeichnissen das Einschalten einer zusätzlichen „Sicherung“ gegen das unerlaubte Löschen von Dateien – wie bereits besprochen. Handelt es sich um ein ausführbares Programm (als *read-only*, d. h. der Programm-Text darf nicht verändert werden), so verbleibt eine Kopie des Programm-Textes im Swap-Bereich (nach Beendigung des Programms) und kann beim nächsten Start schneller geladen werden.

Kommando / Bibliotheksfunktion zum Ändern der Datei-Zugriffsrechte:
chmod / ***int chmod(const char *path, mode_t mode);***

Etwas Spielerei mit den Rechten:

```
spatz$ mkdir tmp
spatz$ ls -l
total 12
-rwxrwxr-x  1 swg users 11276 2005-01-16 17:40 a.out
drwxrwxr-x  2 swg users    48 2005-01-16 17:47 tmp
spatz$ chmod +t tmp
spatz$ ls -l
total 12
-rwxrwxr-x  1 swg users 11276 2005-01-16 17:40 a.out
drwxrwxr-t  2 swg users    48 2005-01-16 17:47 tmp
spatz$ cd tmp
spatz$ > xyz
spatz$ ls -l
total 0
-rw-rw-r--  1 swg users 0 2005-01-16 17:48 xyz
spatz$ cd ..
spatz$ chmod g+s tmp
spatz$ ls -l
total 12
-rwxrwxr-x  1 swg users 11276 2005-01-16 17:40 a.out
drwxrwsr-t  2 swg users    72 2005-01-16 17:48 tmp
spatz$ groups
users uucp dialout audio video
spatz$ chown :dialout tmp
spatz$ ls -l
total 12
-rwxrwxr-x  1 swg users   11276 2005-01-16 17:40 a.out
drwxrwsr-t  2 swg dialout    72 2005-01-16 17:48 tmp
spatz$ id
uid=1000(swg) gid=100(users) groups=14(uucp),16(dialout),
17(audio),33(video),100(users)
spatz$ cd tmp
spatz$ ls -l
total 0
-rw-rw-r--  1 swg users 0 2005-01-16 17:48 xyz
spatz$ > uvw
spatz$ ls -l
total 0
-rw-rw-r--  1 swg dialout 0 2005-01-16 17:51 uvw
-rw-rw-r--  1 swg users   0 2005-01-16 17:48 xyz
spatz$
```

Folgendes Programm verwendet den Systemaufruf *stat()*:
int stat(const char *file_name, struct stat *buf);

stat() erwartet als ersten Parameter einen Dateinamen und liefert dann im zweiten Parameter vom Typ *struct stat* die Informationen über die Datei. Im Fehlerfall ist der Rückgabewert negativ.

Programm 16.1: Informationen über Dateien ermitteln – mittels stat() (*stat.c*)

```
#include <sys/stat.h> // fuer struct stat
#include <unistd.h> // fuer stat(.)
#include <time.h> // fuer ctime(.)
#include <stdlib.h> // fuer exit(.)
#include <stdio.h>

int main(int argc, char **argv) {
    int i; char *s;
    struct stat statbuf;
    for (i = 1; i < argc; i++) {
        printf("Datei␣\"%s\":\n", argv[i]);

        if (stat(argv[i], &statbuf) < 0)
            perror("stat"), exit(1); // Abbruch im Fehlerfall

        /* ctime(.) erzeugt aus der Datenstruktur time_t einen String */
        printf("␣␣␣␣␣␣␣␣letzter␣Zugriff:␣%s",
                ctime(&statbuf.st_atime));
        /* statbuf.st_mtime und statbuf.st_ctime
         * koennen analog behandelt werden. */

        switch (statbuf.st_mode & S_IFMT) {
            case S_IFREG : s = "normale␣Datei"; break;
            case S_IFDIR : s = "Verzeichnis"; break;
            case S_IFCHR : s = "zeichenorient.␣Geraet"; break;
            case S_IFBLK : s = "blockorient.␣Geraet"; break;
            case S_IFLNK : s = "symbolischer␣Link"; break;
            case S_IFIFO : s = "FIFO"; break;
            case S_IFSOCK : s = "Socket"; break;
            default: s = "_unbekannt_"; break;
        }
        printf("␣␣␣␣␣␣␣␣Datei−Typ:␣%s\n", s);
    }
    return 0;
}
```

```
chomsky$ gcc stat.c
chomsky$ a.out stat.c /dev/tty /dev/hda .
Datei "stat.c":
        letzter Zugriff: Sun Jan 11 23:14:34 2004
        Datei-Typ: normale Datei
Datei "/dev/tty":
        letzter Zugriff: Sun Jan 11 23:14:17 2004
        Datei-Typ: zeichenorient. Geraet
Datei "/dev/hda":
        letzter Zugriff: Sat Jul 29 14:48:22 2000
        Datei-Typ: blockorient. Geraet
Datei ".":
        letzter Zugriff: Sun Jan 11 23:16:04 2004
        Datei-Typ: Verzeichnis
chomsky$
```

Anmerkungen: Mit der Funktion *ctime()* (⇝ *time.h*) kann ein Zeitstempel (Datum und Uhrzeit) als String formatiert werden – mit fest vorgegebenem Format. Außerdem stehen Konstanten *S_IFMT*, *S_IFREG*, etc. bereit, um den Typ einer Datei auf einfache Art und Weise mit einer bitweisen Und-Operation zu ermitteln. Beim Fehlschlagen eines Systemaufrufs kann mittels *perror()* eine ausführliche Fehlermeldung ausgegeben werden – später mehr dazu. Mit der Bibliotheksfunktion *exit()* kann man ein Programm mit dem angegebenen Exit-Status beenden (was innerhalb von *main()* gleichbedeutend einem return ist).

16.2.4.3 Verzeichnisse

Verzeichnisse oder *Kataloge* (*Directories*) sind besondere Dateien mit einer *festen Struktur*. Sie enthalten Einträge mit den Elementen *Dateiname* und *Inode-Nummer*. (Jede Datei hat genau eine Inode, aber evtl. mehrere Dateinamen.)

Beispiel:

Lokaler Dateiname	Verweis auf Inode-Liste (Index)
.	4711
..	4709
onefile	47119
otherfile	471110
subdir_1	471130

Jeder Katalog enthält mindestens zwei besondere Einträge, nämlich „.“ und „..“. „.“ ist ein Verweis auf den aktuellen Katalog; „..“ ist ein Verweis auf den Vaterkatalog.

Damit ergibt sich die bekannte Benutzer-Sicht: Hierarchisches Dateisystem oder Dateibaum! Die Blätter im Baum sind *normale Dateien*, *benannte Pipelines* oder *Gerätedateien*, die anderen Knoten *Verzeichnisse*.

Mit den Systemaufrufen *opendir()*, *readdir()* (und *closedir()*) kann man die Einträge in einem Verzeichnis wie folgt ermitteln:

Programm 16.2: Dateien in einem Verzeichnis ermitteln (*dir.c*)

```
#include <sys/types.h>
#include <dirent.h>
#include <stdlib.h>
#include <stdio.h>
```

```
int main(int argc, char **argv) {
    DIR * dir;
    struct dirent *entry;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <directory>\n", argv[0]);
        exit(1);
    }

    if (!(dir = opendir(argv[1])))
        perror("opendir"), exit(2);
    while ((entry = readdir(dir)))
        printf("%ld : %s\n", entry->d_ino, entry->d_name);
    closedir(dir);

    return 0;
}
```

```
thales$ gcc dir.c
thales$ a.out .
2497008 : .
2496977 : ..
2497010 : dir.c
2496000 : a.out
thales$
```

Anmerkungen: Der Rückgabewert von *opendir()* ist genau dann ungleich dem Nullzeiger, wenn das Öffnen erfolgreich war. Andernfalls liegt ein Fehler vor. Die Funktion *readdir()* signalisiert im Falle eines Nullzeigers als Rückgabewert entweder einen Fehler oder, dass kein weiterer Eintrag mehr vorhanden ist. Mit Hilfe von *stat()* könnte man nun zu jedem Verzeichniseintrag wie gehabt weitere Informationen „beschaffen“.

Dateinamen Dateinamen bestehen aus Zahlen, Buchstaben und Sonderzeichen. Erlaubt sind im Prinzip alle Zeichen außer „/“ (Slash) und „\0“ (Null-Byte). Verschiedene Zeichen tragen jedoch eine Sonderbedeutung bei der Kommandoeingabe (Shell) und eignen sich daher nur bedingt für Dateinamen.

Dateinamen sind in älteren UNIX-Versionen auf eine Länge von 14 Zeichen begrenzt, in neueren Versionen auf 256 (genauer: 255 + Null-Byte) Zeichen oder mehr.

Standard-Verzeichnisse

- **root directory**:
  Der Katalog am oberen Ende des Dateibaums / heißt Wurzel (*root*). Dieser Katalog ist direkter oder indirekter Vorgänger aller Dateien und Verzeichnisse im gesamten Dateibaum und hat selbst keinen Vorgänger.

- **top level directories**:
  Im oberen Bereich des Dateibaums befinden sich Kataloge mit verschiedenen Systemdaten.

  - */bin* — ausführbare Programme (*binaries*)
  - */lib* — Bibliotheken und Hilfsdateien

- */tmp* — Platz für temporäre Dateien aller Benutzer – bei einem *Netzwerk-Dateisystem* wie hier im WiMa-Net ist dieser Unterkatalog immer lokal auf der jeweiligen Maschine, liegt also (aus Performance-Gründen) auf einer an dieser Maschine angehängten Platte oder im Hauptspeicher (⇝ tmpfs)
- */home* — Heimatverzeichnisse
- */etc* — Konfigurationsdateien und Systemverwaltungskommandos
- */proc* — Prozess-Pseudo-Dateisystem (*procfs*); benutzt als Schnittstelle zu den Kernel Datenstrukturen (mehr z.B. `man -S5 proc`)
- */dev* — Gerätedateien (*device special files*)
- */usr* — Benutzerdaten und eher lokale Systemdateien (/usr/bin, /usr/lib, /usr/tmp /usr/man)
- */usr/spool* — Zwischenablage für Druckaufträge, externe Post, usw. (*spooling area*)
- */usr/local* — lokale Software, meist ebenfalls unterteilt in Kataloge mit den Namen *bin, lib, man* und *src*.

16.2.4.4 Links

Hardlinks Mit folgendem Kommando kann man einen *Hardlink* erzeugen:

```
ln name neuername
```

Beim Erzeugen eines Hardlinks wird nur ein *neuer Verzeichniseintrag mit derselben Inode-Nummer* angelegt. (Link – genauer Hardlink – ist also ein Synonym für Verzeichniseintrag!)

Die bisher über den Namen `name` erreichbaren Daten sind jetzt auch über den Namen `neuername` erreichbar. Beim Zugriff besteht zwischen `name` und `neuername` keinerlei Unterschied, da letztendlich auf den gleichen physikalischen Speicherplatz auf der Festplatte zugegriffen wird.

Hardlinks können nur auf „normale“ Dateien oder Gerätedateien eingerichtet werden. Beide Objekte, `name` und `neuername`, müssen sich im *gleichen Dateisystem* befinden, da ja über die Inode-Nummer referenziert wird. Der Eintrag `name` muss bereits existieren.

Der Linkzähler (link count) in der Inode gibt an, wieviele Dateinamen dieselbe Datei referenzieren (z. B. „.“ und „..“ in einem Unterverzeichnis beziehen sich auf dieselbe Datei).

Beispiel für das Anlegen eines Hardlinks:

```
chomsky$ ls -lai
total 8
 313752 drwxr-xr-x   2 jmayer   users        4096 Jan 12 00:12 .
 117840 drwx------   3 jmayer   users        4096 Jan 12 00:11 ..
 314005 -rw-r--r--   1 jmayer   users           0 Jan 12 00:12 hallo.txt
chomsky$ ln hallo.txt neu.txt
chomsky$ ls -lai
total 8
 313752 drwxr-xr-x   2 jmayer   users        4096 Jan 12 00:12 .
 117840 drwx------   3 jmayer   users        4096 Jan 12 00:11 ..
 314005 -rw-r--r--   2 jmayer   users           0 Jan 12 00:12 hallo.txt
 314005 -rw-r--r--   2 jmayer   users           0 Jan 12 00:12 neu.txt
chomsky$
```

Anmerkungen: In der ersten Spalte der Ausgabe von *ls -lai* steht die *Inode-Nummer* und in der dritte Spalte steht der *Link-Zähler* (*link count*).

Softlinks Mit folgendem Kommando kann man einen *Softlink* bzw. *symbolischen Link* erzeugen:

```
ln -s name neuername
```

Beim Erzeugen eines Softlinks wird eine *neue Datei* (vom *Typ „Softlink“*) erzeugt, deren *Inhalt der Name der referenzierten Datei* ist.

Das bisher über den Namen `name` erreichbare Daten-Objekt ist jetzt auch über `neuername` erreichbar. Falls das Daten-Objekt `name` existiert, so bestehen beim Zugriff auf diese Daten via `neuername` aus Sicht des Benutzer keinerlei Unterschiede zum Zugriff über `name`. Im Kernel laufen jedoch zusätzliche Operationen ab (für `neuername` wird eine Inode angelegt!). Softlinks sind folglich „teurer“ als Hardlinks.

Der Eintrag `name` muss nicht unbedingt existieren. Softlinks können außerdem auch auf Verzeichnisse eingerichtet werden und können auch auf andere Dateisysteme verweisen. Der Link-Zähler (link count) wird logischerweise nicht erhöht.

Beispiel für das Anlegen eines Softlinks:

```
chomsky$ ls -lai
total 8
 313752 drwxr-xr-x   2 jmayer   users        4096 Jan 12 00:14 .
 117840 drwx------   3 jmayer   users        4096 Jan 12 00:13 ..
 314005 -rw-r--r--   1 jmayer   users           0 Jan 12 00:12 hallo.txt
chomsky$ ln -s hallo.txt neu.txt
chomsky$ ls -lai
total 8
 313752 drwxr-xr-x   2 jmayer   users        4096 Jan 12 00:14 .
 117840 drwx------   3 jmayer   users        4096 Jan 12 00:13 ..
 314005 -rw-r--r--   1 jmayer   users           0 Jan 12 00:12 hallo.txt
 314007 lrwxrwxrwx   1 jmayer   users           9 Jan 12 00:14 neu.txt -> hal
chomsky$
```

Konsequenzen

- Durch Hardlinks wird aus dem Dateibaum ein zyklenfreier gerichteter Graph.
- Durch Softlinks können auch Zyklen im Dateisystem entstehen (durch Softlinks auf Verzeichnisse)!

16.3 Systemaufrufe für I/O-Verbindungen – Erster Teil

16.3.1 Öffnen von Dateiverbindungen – open()

int open(const char *path, int oflag);
int open(const char *path, int oflag, mode_t mode);

Der Systemaufruf *open()* mit *zwei* Parametern öffnet eine existierende Datei. Die Variante mit *drei* Parametern ist für den Fall bestimmt, dass eine Datei beim Öffnen zusätzlich angelegt wird.

Der erste Parameter ist der *Name der zu öffnenden Datei*.

Der zweite Parameter enthält die *oflags*, die die Art der I/O-Verbindung und ihre Eigenschaften spezifizieren. *open()* kennt drei Arten von I/O-Verbindungen:

oflag	Verbindungsart
O_RDONLY	nur lesen
O_WRONLY	nur schreiben
O_RDWR	lesen und schreiben

Die *oflags* können nicht nur die Werte *O_RDONLY*, *O_WRONLY* oder *O_RDWR* annehmen, sondern differenziertere Flagwerte. Diese entstehen durch (bitweises) Oder-Verknüpfung von einem oder mehreren der folgenden Flagwerte zur Verbindungsart:

oflag	Bedeutung
O_CREAT	Falls die Datei nicht existiert, wird sie angelegt. Nur bei diesem Flagwert findet das dritte Argument Verwendung.
O_TRUNC	Falls die Datei bereits existiert, wird sie auf Länge 0 verkürzt, d. h., der alte Inhalt wird „gelöscht“.
O_APPEND	Dieses Flag stellt sicher, dass jede Schreib-Operation am Ende der Datei erfolgt, auch bei konkurrierenden Schreib-Operationen verschiedener Prozesse.
O_EXCL	Zusammen mit dem *O_CREAT*-Flag stellt dieses Flag sicher, dass *open()* fehlschlägt, wenn `path` bereits existiert.
O_NONBLOCK *O_NDELAY*	Prozess blockiert nicht beim Öffnen eines noch nicht bereiten IPC-Kanals (FIFO). Zusätzlich wirkt sich dieses Flag auf nachfolgende *read()* und *write()* Systemaufrufe aus.
O_SYNC	Bei jedem *write()* auf diese Dateiverbindung blockiert der Prozess bis die Schreib-Operation physikalisch abgeschlossen ist.

Beliebige Kombinationen sind möglich, aber nicht alle sind sinnvoll! *fcntl.h* enthält für *oflag* die Definitionen der symbolischen Konstanten (Makros).

Der dritte Parameter von *open()* ist nur relevant, wenn eine Datei geöffnet werden soll, die noch nicht existiert. In diesem Fall soll die Datei ggf. angelegt werden (über den zweiten Parameter festzulegen – durch Oder-Verknüpfung mit *O_CREAT*); dabei müssen dann die Zugriffsrechte auf diese neue Datei durch den dritten Parameter festgelegt werden (in der Form `rwxrwxrwx` für Eigentümer, Gruppe und Andere und zwar binär kodiert – wie gehabt bei *stat()*, nur ohne Dateityp etc.).

Eine erfolgreiche Ausführung des Systemaufrufs liefert einen *Filedeskriptor* zurück (genauer: den *kleinsten freien* Filedeskriptor). Dieser ist eine kleine, nicht-negative Zahl, die bei allen nun folgenden I/O-Operationen (*read()*, *write()*, ..., *close()*) diese I/O-Verbindung identifiziert. Sie stellt einen Index in eine entsprechende Tabelle (UFDT) dar. *stdin* steht für den Filedeskriptor 0, *stdout* für 1 und *stderr* für 2. Diese sind beim Start eines Prozesses typischerweise bereits geöffnet.

Im Falle eines Fehlers liefert *open()* einen negativen Wert.

Beispiele:

- Eine Datei zum Schreiben anlegen. Wenn sie bereits existiert, dann soll sie auf Länge 0 verkürzt werden:

```
int fd;
if ((fd = open(dateiname, O_WRONLY | O_CREAT | O_TRUNC, perms)) < 0) {
    /* Fehlerbehandlung, z.B. */
    perror("open()");
    exit(1);
}
```

- Eine Datei, die bereits existiert, zum Lesen und Schreiben öffnen. Alle Schreiboperationen sollen am Dateiende erfolgen.

```
int fd;
if ((fd = open(dateiname, O_RDWR | O_APPEND, perms)) < 0) {
    /* Fehlerbehandlung, z.B. */
    perror("open()");
    exit(1);
}
```

- Eine Datei, die noch nicht existieren darf, zum Schreiben öffnen. (Existiert die Datei bereits, so führt dies zu einem Fehler!)

```
int fd;
if ((fd = open(dateiname, O_WRONLY | O_CREAT | O_EXCL, perms)) < 0) {
    /* Fehlerbehandlung, z.B. */
    perror("open()");
    exit(1);
}
```

Programm 16.3 öffnet ein Datei names „myfile.txt“ zum Lesen und Schreiben. Existiert diese Datei noch nicht, so wird sie angelegt, andernfalls auf Länge 0 verkürzt.

Programm 16.3: Öffnen einer Datei – mittels open() (*open.c*)

```
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
    int fd;

    if ((fd = open("myfile.txt", O_RDWR | O_CREAT | O_TRUNC, 0664)) < 0)
        perror("open"), exit(1);

    /* Arbeiten mit der Dateiverbindung fd ... */

    close(fd);

    return 0;
}
```

Anmerkungen: Die *Zugriffsrechte* werden am einfachsten als *Oktalzahl* (wie hier geschehen) angegeben. Hier wurden die Zugriffsrechte `rw-rw-r--` gesetzt, die aber noch durch die sogenannte *umask* modifiziert werden. Die Zugriffsrechte, die bei open gesetzt werden sollen, werden mit dem *Komplement der umask mit bitweisem Und verknüpft*. (**Beispiel:** In vorigem Programm sollten die Zugriffsrechte 0664 gesetzt werden; 0664 & ~*umask* ist 0664 & ~022 = 0644 im Fall, dass *umask* = 022 gesetzt ist, was bewirkt, dass die Schreibrechte für die Gruppe und Andere entfernt werden). Dabei können höchstens Rechte weggenommen werden, aber keine dazu kommen. Die *umask* (*file mode creation mask*) kann mit dem Systemaufruf *umask()* und dem Kommando *umask* gesetzt werden (siehe Manpages).

16.3.2 Schließen von Dateiverbindungen – close()

int close(int fd);

Beendet die I/O-Verbindung, die durch den übergebenen Filedeskriptor repräsentiert wird, sofern kein weiterer Filedeskriptor für dieselbe I/O-Verbindung existiert. Im Fehlerfall liefert *close()* einen negativen Wert.

Da der Kernel die Zahl der I/O-Verbindungen pro Prozess limitiert ist es ratsam, nicht mehr gebrauchte I/O-Verbindungen mit *close()* zu schließen.

Terminiert ein Prozess, so führt der Kernel implizit einen *close()* Systemaufruf für alle noch offenen I/O-Verbindungen aus.

16.3.3 Duplizieren von Filedeskriptoren – dup(), dup2()

int dup(int fd);
int dup2(int oldfd, int newfd);

Der Systemaufruf *dup()* erhält als Argument einen Filedeskriptor und liefert als Rückgabewert einen neuen Filedeskriptor als Duplikat zurück; dieser ist der kleinste freie Filedeskriptor! Das Duplikat bezeichnet exakt dieselbe I/O-Verbindung. Im Fehlerfall liefert *dup()* einen negativen Wert. Der Systemaufruf *dup2()* kopiert den Filedeskriptor *oldfd* nach *newfd* und schließt ggf. vorher den Filedeskriptor *newfd*, falls er eine I/O-Verbindung repräsentierte. *dup2()* liefert ebenfalls den neuen Filedeskriptor und im Fehlerfall -1.

Das folgende Programm lenkt mit Hilfe von *dup()* die Standardausgabe (*stdout*) in eine Datei namens „stdout.txt“ um.

Programm 16.4: Umlenkung der Standardausgabe – mittels dup() (*dup.c*)

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd;

    /* Datei, auf die die Standardausgabe umgelenkt werden soll, oeffnen */
    if ((fd = open("stdout.txt", O_WRONLY | O_CREAT | O_TRUNC, 0664)) < 0)
        perror("open"), exit(1);

    close(1); /* stdout schliessen */

    dup(fd); /* fd duplizieren => 1 ist der kleinste freie! */
    close(fd); /* fd schliessen, da doppelt */

    /* Ausgabe auf stdout => geht in die Datei stdout.txt! */
    puts("Hallo␣Welt!");

    return 0;
}
```

Erläuterung: Zunächst wird eine neue Dateiverbindung eröffnet (zum Schreiben). Danach wird Filedeskriptor 1 (*stdout*) geschlossen und dies ist jetzt der kleinste freie. Somit liefert der folgende *dup()*-Aufruf als Ergebnis den Filedeskriptor 1. Alle Schreiboperationen nach *stdout* erfolgen danach in die Datei mit Namen „stdout.txt“. Der Filedeskriptor fd ist nun ein unnötiges Duplikat und kann nun geschlossen werden. Es gibt ja immer noch einen Filedeskriptor für diese Dateiverbindung, nämlich den Filedeskriptor 1. (Die Aufgabe ließe sich auch kürzer mit einem *close(1)* gefolgt von *open()* lösen. ;-)

16.3.4 Informationen über Dateien und I/O-Verbindungen – stat(), etc.

int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);

Der Systemaufruf *stat()* – der in Programm 16.1 auf Seite 186 bereits verwendet und zuvor besprochen wurde – liefert Informationen zu einer Datei in dem übergebenen Puffer `buf`. Der Rückgabewert ist im Fehlerfall negativ.

Die beiden Systemaufrufe *stat()* und *lstat()* sind nahezu identisch. Sie verhalten sich bei *symbolischen Links* jedoch unterschiedlich. Mit *stat()* erhält man bei einem symbolischen Links über die Datei, auf die der Link „zeigt", wohingegen man von *lstat()* Informationen über den symbolischen Link selbst erhält.

Mit dem Systemaufruf *fstat()*, der als erstes Argument nicht einen Dateinamen, sondern einen Filedeskriptor erwartet, kann man Informationen über eine offen Dateiverbindung erhalten.

Das folgende Programm ermittelt – mit Hilfe des Systemaufrufs *fstat()* – den Typ der Datei, die hinter der Standardeingabe steckt:

Programm 16.5: Typ der Standardeingabe ermitteln – mittels fstat() (*fstat.c*)

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {
    char *s;
    struct stat statbuf;

    if (fstat(0, &statbuf) < 0)
        perror("fstat"), exit(1);

    switch (statbuf.st_mode & S_IFMT) {
        case S_IFREG : s = "normale␣Datei"; break;
        case S_IFDIR : s = "Verzeichnis"; break;
        case S_IFCHR : s = "zeichenorient.␣Geraet"; break;
        case S_IFBLK : s = "blockorient.␣Geraet"; break;
        case S_IFLNK : s = "symbolischer␣Link"; break;
        case S_IFIFO : s = "FIFO"; break;
        case S_IFSOCK : s = "Socket"; break;
        default: s = "_unbekannt_"; break;
    }
    printf("Datei−Typ:␣%s\n", s);

    return 0;
}
```

```
thales$ gcc -Wall fstat.c
thales$ a.out
Datei-Typ: zeichenorient. Geraet
thales$ a.out < stat.c
Datei-Typ: normale Datei
thales$
```

16.3.5 Zugriff auf Verzeichnisse – readdir(), etc.

DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);

Mit den Systemaufrufen *opendir()*, *readdir()* und *closedir()* können die Einträge eines Verzeichnisses gelesen werden. Dies wurde in Abschnitt 16.2.4.3 auf Seite 187 bereits erläutert und mit dem Programm 16.2 demonstriert. Im Fehlerfall liefern diese Systemaufrufe einen Null-Zeiger oder einen negativen Wert.

Jeder Verzeichniseintrag wird in einer Struktur vom Typ *struct dirent* geliefert:

```
struct  dirent {
    ino_t           d_ino;      /* Inode-Nummer */
    off_t           d_off;      /* (interne Bedeutung) */
    unsigned short  d_reclen;   /* Groesse des Eintrags */
    char            d_name[1];  /* Dateiname */
};
```

Diese Struktur enthält in der Komponente *d_ino* die *Inode-Nummer* und in der Komponente *d_name* den *Dateinamen*. Die Komponente *d_reclen* enthält die Größe der Struktur (die nicht fest ist, bedingt durch die variable Größe von *d_name*). Die Komponente *d_off* ist nur von interner Bedeutung für das Dateisystem.

16.3.6 Schreiben in I/O-Verbindungen – write()

ssize_t write(int fd, const void *buf, size_t nbyte);

write() schreibt aus dem Hauptspeicher in eine I/O-Verbindung. *nbyte* bestimmt die Datenmenge (Anzahl der zu schreibenden Bytes), *buf* ist die Startadresse im Hauptspeicher und der Filedeskriptor *fd* identifiziert die I/O-Verbindung.

Eine *write()*-Operation ist ein 1:1 Kopiervorgang. Es findet keinerlei Daten-Konvertierung oder Veränderung statt. Führt die I/O-Verbindung zu einer gewöhnlichen Datei (regular file), so bestimmt der sog. *File Offset Pointer* im Open-File-Table-Slot (OFT – siehe weiter hinten) das Ziel des Kopiervorgangs. Die übertragenen Bytes überschreiben bereits vorher gespeicherte Daten. Anschließend zeigt der File Offset Pointer hinter das letzte übertragene Byte.

Der Rückgabewert von *write()* ist die Anzahl der geschriebenen Bytes bzw. -1 im Fehlerfall.

Achtung: Die Modellannahme, dass die Daten nach Abschluß des *write()* Systemaufrufs physikalisch auf die Platte geschrieben wurden, ist nicht vollständig korrekt. Das I/O-Subsystem kopiert die Daten in den Puffer-Cache und gibt die Kontrolle zurück, etwa mit folgender Meldung:

> *„I've taken note of your request, and rest assured that your file deskriptor is OK, I've copied your data successfully, and there's enough disk space. Later, when it's convenient for me, and if I'm still alive, I'll put your data on the disk where it belongs. If I discover an error then I'll try to print something on the console, but I won't tell you about it (indeed, you may have terminated by then). If you, or any other process, tries to read this data before I've written it out, I'll give it to you from the buffer cache, so, if all goes well, you'll never be able to find out when and if I've completed your request. You may ask no further questions. Trust me. And thank me for the speedy reply."* [Marc J. Rochkind, Seite 29]

Mit dem Flag *O_SYNC* könnte man dies verhindern. Ein deutlicher Performanceverlust wäre jedoch die Folge.

Folgendes Programm verwendet *write()*, um einen String (und einen abschließenden Zeilenumbruch) auf die Standardausgabe (Filedeskriptor 1) auszugeben:

Programm 16.6: Ausgabe eines Strings mit Hilfe von write() (*write.c*)

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

int main() {
    char *s = "Hallo␣Welt!";
    int len = strlen(s);

    /* Ausgabe von s mit abschliessendem Newline aus stdout */
    write(1, s, len);
    write(1, "\n", 1);

    return 0;
}
```

Da *write()* unter Umständen auch weniger Bytes schreiben kann, als gefordert ist, führt folgendes Programm solange *write()* Systemaufrufe aus, bis alle Daten geschrieben sind.

Programm 16.7: Ausgabe eines Strings mit Hilfe von write() (*write1.c*)

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

int dowrite(int fd, const void *buf, size_t nbyte) {
    int ret, n = nbyte;
    const char *cbuf = buf;
    /* solange write() aufrufen, bis alle Daten geschrieben sind */
    do {
        ret = write(fd, cbuf, nbyte);
        cbuf += ret; /* Adresse erhoehen um Anzahl der geschr. Bytes */
        nbyte -= ret; /* und Anzahl zu schreibender Bytes entspr. verringern */
    } while (ret > 0 && nbyte > 0);
    return ret < 0 ? ret : n;
}

int main() {
    char *s = "Hallo␣Welt!";
    int len = strlen(s);

    /* Ausgabe von s mit abschliessendem Newline aus stdout */
    dowrite(1, s, len);
    dowrite(1, "\n", 1);

    return 0;
}
```

16.3.7 Lesen aus I/O-Verbindungen – read()

ssize_t read(int fd, void *buf, size_t nbyte);

read() liest (maximal) *nbyte* viele Bytes von der I/O-Verbindung, die der Filedeskriptor *fd* bezeichnet, in den Puffer mit der Startadresse *buf*. Die Datenübertragung beginnt mit dem Byte, auf das der *File Offset Pointer* im OFT-Slot zeigt. Nach Abschluß des *read()* Systemaufrufs zeigt der *File Offset Pointer* auf das Byte, welches auf das letzte übertragene Byte folgt.

Der Rückgabewert 0 zeigt *EOF* (*End Of File*) und −1 einen Fehler an. Andernfalls gibt *read()* die Anzahl der gelesenen Bytes (nicht mehr als *nbyte*) zurück.

Folgendes Programm verwendet *read()*, um (wie *fgets()*) eine Zeile (aber maximal *buflen*−1 Zeichen) von der Standardeingabe zu lesen (*readline()* ist ein *fgets()*-Variante für Filedeskriptoren):

Programm 16.8: Einlesen einer Zeile von der Standardeingabe – mittels read() (*read.c*)

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

#define BUFLEN 128

/* liest eine Zeile von der I/O−Verbindung, die fd bezeichnet,
 * aber maximal buflen−1 viele Zeichen
 */
int readline(int fd, char *buf, int buflen) {
    int res, len;

    /* eine Zeile (aber max. BUFLEN−1 Zeichen) von stdin lesen */
    for (len = 0; len < buflen−1; len++) {
        /* ein Zeichen von stdin in den Puffer lesen */
        if ((res = read(fd, buf+len, 1)) < 0)
            return −1;
        /* bei EOF oder '\n' ist Schluss mit dem Einlesen */
        if (res == 0 || buf[len] == '\n')
            break;
    }

    /* String noch sauber mit einem Null−Byte terminieren */
    buf[len] = '\0';

    return len;
}

int main() {
    char buf[BUFLEN];

    if (readline(0, buf, BUFLEN) < 0)
        perror("readline"), exit(1);

    /* ... und auf stdout ausgeben */
    puts(buf);

    return 0;
}
```

Besonderheit: Nicht-blockierendes Lesen (non-blocking read)

Durch die Abstraktion aller Datenquellen/-ziele zu Dateien kann mit *read()* nicht nur von Plattendateien, sondern auch von Terminals, Pipes und Stream-Devices gelesen werden.

Wenn über eine derartige I/O-Verbindung gerade keine Daten zur Verfügung stehen und ein Prozess führt darauf einen *read()* Systemaufruf aus, dann blockiert der Prozess bis Daten eintreffen. Wurde jedoch beim *open()* oder durch *fcntl()* das Flag *O_NDELAY* / *O_NONBLOCK* für diese I/O-Verbindung gesetzt, so kehrt *read()* ohne zu blockieren zurück. Bei Terminals und Pipes produziert *read()* als Rückgabewert 0, was ein Entdecken von **EOF** unmöglich macht. Dieses Problem vermeidet *read()* bei den neueren Stream-Devices, hier kommt der Wert -1 zurück und *errno* enthält den Wert *EAGAIN*.

16.3.8 Fehlerbehandlung bei Systemaufrufen – perror()

Jeder Systemaufruf zeigt mit seinem *Rückgabewert* an, ob die Ausführung erfolgreich war oder ob eine Fehlersituation aufgetreten ist. Normalerweise liefert ein Systemaufruf im Fehlerfall entweder einen *negativen Wert (oder sogar genau* -1*)* oder einen *Null-Zeiger* – abhängig vom Typ des Rückgabewertes. Ein stabil kodiertes Programm überprüft den Rückgabewert bei jedem Systemaufruf.

Programm 16.9: Fehlerbehandlung bei Systemaufrufen – Die Erste (*error1.c*)

```
#include <stdio.h> /* wg. perror() */
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    int fd;
    if ((fd = open("file", O_RDWR | O_CREAT | O_EXCL | O_TRUNC, 0600)) < 0)
        perror("open"), exit(1);
    close(fd);
    return 0;
}
```

```
chomsky$ gcc -Wall error1.c
chomsky$ ls
a.out  error1.c
chomsky$ a.out
chomsky$ ls
a.out  error1.c  file
chomsky$ a.out
open: File exists
chomsky$
```

Wenn der System Call *open()* (exklusives Erzeugen einer Datei, siehe *O_EXCL*) bei der Ausführung im Kernel Mode in irgendwelche Probleme läuft, liefert er *-1* als Ergebnis. Dies tritt z. B. auf, wenn die Datei bereits existiert.

Fragen:

Auf welches Problem traf *open()*?
Woher weiß *perror()*, welches Problem aufgetreten ist?

Antwort: *errno*

Tritt bei der Ausführung eines Systemaufrufs ein Fehler auf, liefert der Aufruf einen eigentlich unmöglichen Wert zurück. Dies ist meist *-1* oder ein *Null-Zeiger*. Die Details

beschreibt die Definition der einzelnen Systemaufrufe im Kapitel 2 der *Reference Manuals* (siehe auch *man -s 2 intro ↪ Introduction to system calls and error numbers*).

Neben diesem Fehlerindikator macht der Kernel in der globalen Variable *errno* eine Fehlernummer verfügbar. Mit Hilfe dieser Fehlernummer kann *perror()* bzw. *strerror()* eine entsprechende Fehlermeldung ausgeben bzw. erzeugen.

Programm 16.10: Fehlerbehandlung bei Systemaufrufen – Die Zweite (*error2.c*)

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h> /* wg. errno */
#include <string.h> /* wg. strerr() */

int main() {
    int fd;
    if ((fd = open("file", O_RDWR | O_CREAT | O_EXCL | O_TRUNC, 0600)) < 0) {
        perror("open");
        printf("errno␣=␣%d\n", errno);
        printf("Fehler␣=␣%s\n", strerror(errno));
        exit(1);
    }
    close(fd);
    return 0;
}
```

```
chomsky$ gcc -Wall error2.c
chomsky$ ls
a.out  error2.c
chomsky$ a.out
chomsky$ ls
a.out  error2.c  file
chomsky$ a.out
open: File exists
errno = 17
Fehler = File exists
chomsky$
```

Die Datei */usr/include/errno.h* definiert symbolische Konstanten (Makros) für alle vom Kernel benutzten Fehlernummern.

errno	Makro	Fehlermeldung
1	EPERM	Operation not permitted
2	ENOENT	No such file or directory
3	ESRCH	No such process
4	EINTR	Interrupted system call
	...	
19	ENODEV	No such device
20	ENOTDIR	Not a directory
	...	
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Not a typewriter
	...	
64	ENONET	Machine is not on the network
65	ENOPKG	Package not installed
	...	

Anwendung von errno:
Wird ein "langsamer" Systemaufruf (z. B. *read()*) durch ein Interrupt-Signal unterbrochen, so liefert er -1 als Fehlerindikator zurück, setzt aber *errno* auf den Wert *EINTR*. Dies bedeutet, dass eigentlich *kein Fehler* aufgetreten ist, sondern der Systemaufruf nur „gestört" worden war. Das Programm kann nun entscheiden, ob es den Systemaufruf erneut aufsetzt oder auf die Unterbrechung hin anders weiterarbeitet.

```
/* ... */

do {
    nread = read(fd, buf, BUFSIZE);
} while (nread < 0 && errno == EINTR); /* ... nur unterbrochen? */

if (nread < 0)
    perror("read"), exit(1); /* ... also echter Fehler */

/* ... */
```

16.4 Datenstrukturen für I/O-Verbindungen

16.4.1 UFDT, OFT und KIT

Der Kernel verwaltet in seinem Bereich des Hauptspeichers eine Reihe von Datenstrukturen, die die Systemaufrufe beim Arbeiten mit Dateien benutzen (siehe Abbildung 16.6).

- **UFDT: User File Descriptor Table**
  Die *UFDT* (*User File Descriptor Table*) identifiziert alle offenen I/O-Verbindungen eines Prozesses. Der Kernel legt diese Tabelle für jeden Prozeß (aber im Kernel-Adressraum) an. Sie gehört zum Kontext des Prozesses. Jeder *Filedeskriptor* eines Prozesses ist ein Index in seine UFDT. Als einzige Komponente enthalten die Slots der UFDT einen Zeiger in die *OFT* (*Open File Table*) des Kernels.

- **OFT: Open File Table**
  Die *OFT* (*Open File Table*) existiert nur einmal im Adressraum des Kernels. Alle Slots der *OFT* enthalten

  – einen Zeiger in die *KIT* (*Kernel Inode Tabelle*),

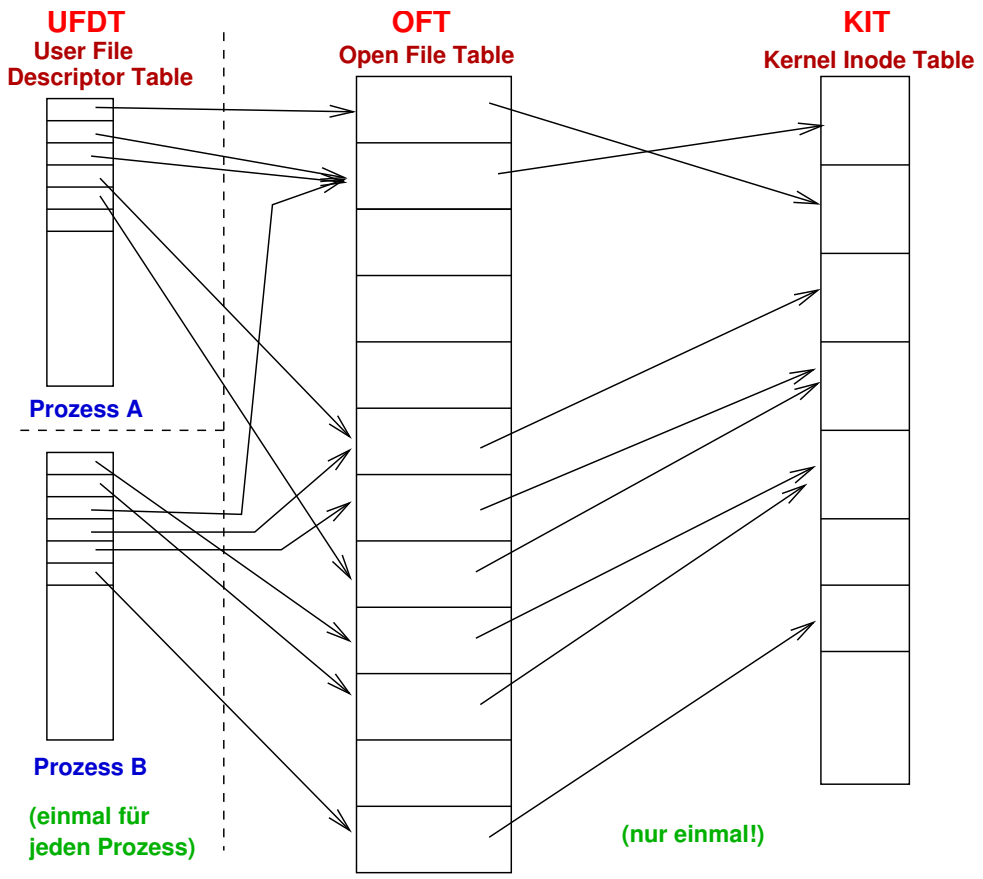


Abbildung 16.6: Datenstrukturen für I/O-Verbindungen

- die *oflags*,
- einen *File Offset Pointer*
- und einen Referenz-Zähler, der angibt, wieviele UFDT-Slots auf diesen OFT-Slot verweisen.

Die *oflags* stammen aus dem *open()*-Aufruf (bzw. aus dem letzten *fcntl()*-Aufruf). Der *File Offset Pointer* legt fest, bei welchem Byte die nächste *read()*- oder *write()*-Operation stattfindet.

- **KIT: Kernel Inode Table**
  Die *KIT* (*Kernel Inode Table*) existiert nur einmal im System. Der Kernel koordiniert darüber sämtliche Zugriffe auf Dateien. Die *in-core* Version der Inodes enthält alle Komponenten der Plattenversion plus Informationen, die nur bei offenen I/O-Verbindungen von Bedeutung sind: ein Referenz-Zähler (der angibt wieviele OFT-Slots auf diesen KIT-Slot verweisen), die Gerätenummer, ...
  Jede offene Datei belegt genau einen Slot, unabhängig wie oft und von wievielen Prozessen sie geöffnet wurde.

16.4.2 Interne Abläufe bei den Systemaufrufen

Um die Integrität der I/O-Datenstrukturen zu gewährleisten, behält der Kernel sämtliche Tabellen in seinem Adreßraum. Prozesse können deshalb nur durch Systemaufrufe auf

diese Tabellen einwirken.

16.4.2.1 Systemaufruf open()

open() belegt immer je einen Slot in der UFDT und der OFT. Der Referenz-Zähler im OFT-Slot wird mit 1 initialisiert. Bestand keine Verbindung zu der Datei, belegt *open()* auch einen Slot in der KIT und initialisiert diese neue in-core Inode (und deren Referenz-Zähler auf 1). Besteht bereits eine Verbindung zu der Datei (vom selbem oder einem anderen Prozess), wird der schon für die Datei reservierte KIT-Slot verwendet. Der Kernel installiert nur den Zeiger vom OFT-Slot auf den KIT-Slot und inkrementiert den Referenz-Zähler in dieser in-core Inode.

Folge: Mehrere OFT-Slots können auf den selben KIT- Slot zeigen. Der Kernel hält von jeder Platten-Inode maximal eine Kopie „*in-core*".

16.4.2.2 Systemaufruf close()

close() gibt einen Slot in der UFDT frei und dekrementiert den Referenz-Zähler im OFT-Slot. Wird dieser Referenz-Zähler dadurch zu 0, kann auch der OFT-Slot freigegeben und der Referenz-Zähler im KIT-Slot dekrementiert werden. Wird nun dieser Referenz-Zähler (in der KIT) zu 0, so gibt der Kernel auch den KIT-Slot frei. Sollte in der Inode der *Link Count* 0 sein, so kann der Kernel jetzt auch die Platten-Inode und den Plattenplatz freigeben.

16.4.2.3 Systemaufruf dup()

Der Systemaufruf *dup()* (siehe auch *dup2()*) belegt den niedersten, noch freien Slot in der UFDT, kopiert den Inhalt eines bereits belegten Slots dorthin und inkrementiert den Referenz-Zähler im OFT-Slot, auf den nun beide UFDT-Slots zeigen.

Folge: Mehrere UFDT-Slots eines Prozesses können auf den selben OFT-Slot zeigen.

16.4.2.4 Systemaufruf fork()

Der Systemaufruf *fork()* erzeugt einen neuen Prozess, indem er den kompletten Kontext des ausführenden Prozesses verdoppelt. Dabei muss der Kernel u. A. sowohl die UFDT kopieren als auch die Referenz-Zähler in den entsprechenden OFT-Slots inkrementieren.

Folge: Mehrere UFDT-Slots aus verschiedenen aber verwandten Prozessen können auf den selben OFT-Slot zeigen (Vererbung). Diese Prozesse können sich dadurch einen File Offset Pointer teilen.

16.4.2.5 Beispiel

Im Folgenden soll die Belegung der **Kerneltabellen** über eine Folge von Systemaufrufen gezeigt werden. *Der Ausgangszustand ist in Abbildung 16.7 dargestellt.*

Vater:

fd1 = *open*("file1", *O_RDONLY*);
fd2 = *open*("file2", *O_RDWR*);

Dabei wird jeweils ein UFDT-Slot und ein OFT-Slot angelegt und der Referenz-Zähler im OFT-Slot auf 1 gesetzt. Da beide Dateien noch nicht offen waren wir auch je ein Eintrag in die KIT hinzugefügt (mit Referenz-Zähler 1). *Danach sieht die Situation wie in Abbildung 16.8.*

Vater:

fork();

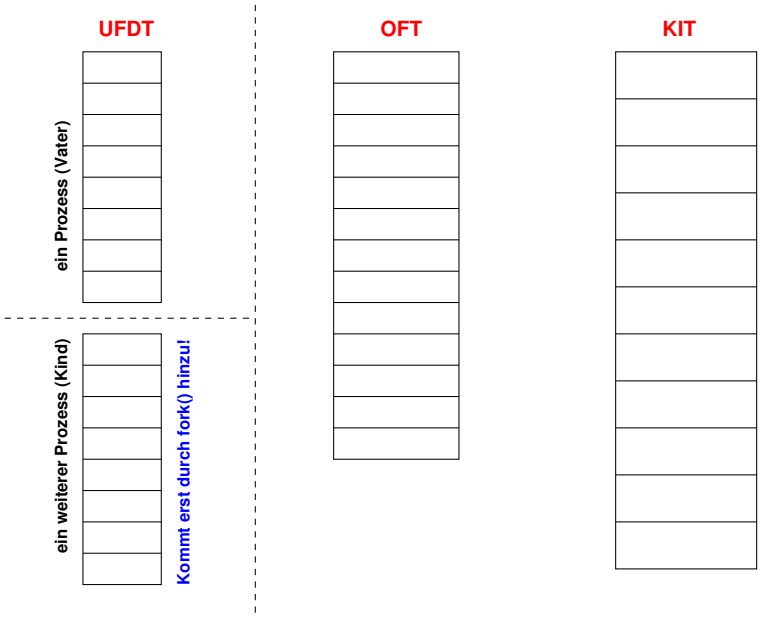


Abbildung 16.7: Kerneltabellen – Ausgangszustand

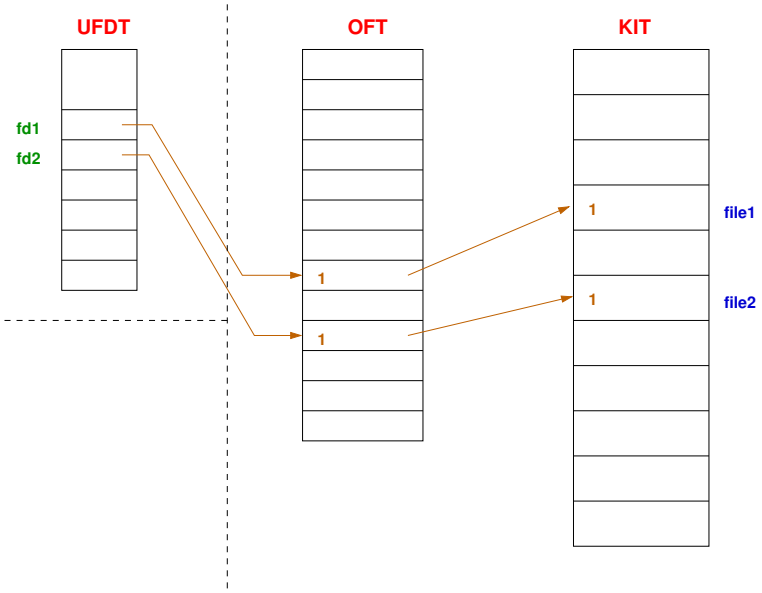


Abbildung 16.8: Kerneltabellen (1)

Durch den Systemaufruf *fork()* wird der Kontext des Prozesses dupliziert und es entsteht so ein weiterer Prozess. Dessen UFDT-Einträge verweisen auf dieselben OFT-Slots, deren Referenzähler dabei dementsprechend erhöht wurden. *Danach sieht die Situation wie in Abbildung 16.9.*

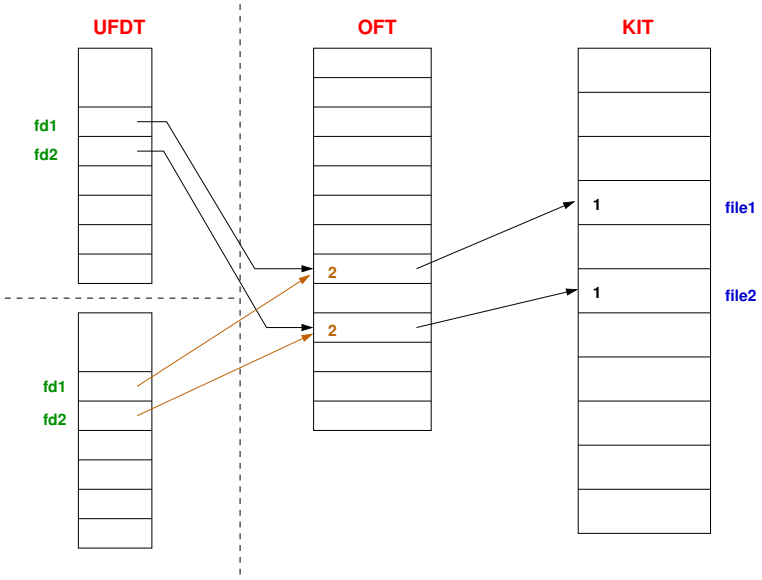


Abbildung 16.9: Kerneltabellen (2)

Vater:

fd3 = *open*("file3", *O_WRONLY*);

Kind:

fd3 = *open*("file3", *O_RDONLY*);

Durch jedes der beiden *open()* Systemaufrufe wird bei Vater und Kind je ein neuer UFDT- und eine neuer OFT-Slot allokiert und der Referenz-Zähler im OFT-Slot auf 1 gesetzt. Beim ersten *open()* wird die Inode in den Speicher geholt und von der OFT ein Verweis auf diesen KIT-Eintrag gesetzt – und schließlich der Referenz-Zähler im KIT-Eintrag auf 1 gesetzt. Beim zweiten *open()* wird nur ein weiterer Verweis auf denselben KIT-Slot erzeugt und der Referenz-Zähler im KIT-Slot inkrementiert. *Danach sieht die Situation wie in Abbildung 16.10.*

Vater:

fd4 = *open*("file4", *O_RDONLY*);

Kind:

fd4 = *open*("file5", *O_RDONLY*);

Danach sieht die Situation wie in Abbildung 16.11.

Kind:

close(*fd1*);

Vater:

close(*fd2*);

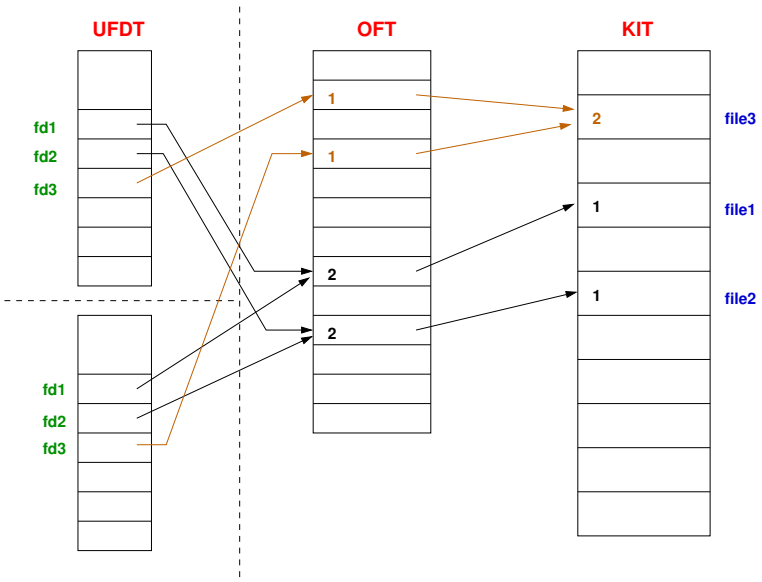


Abbildung 16.10: Kerneltabellen (3)

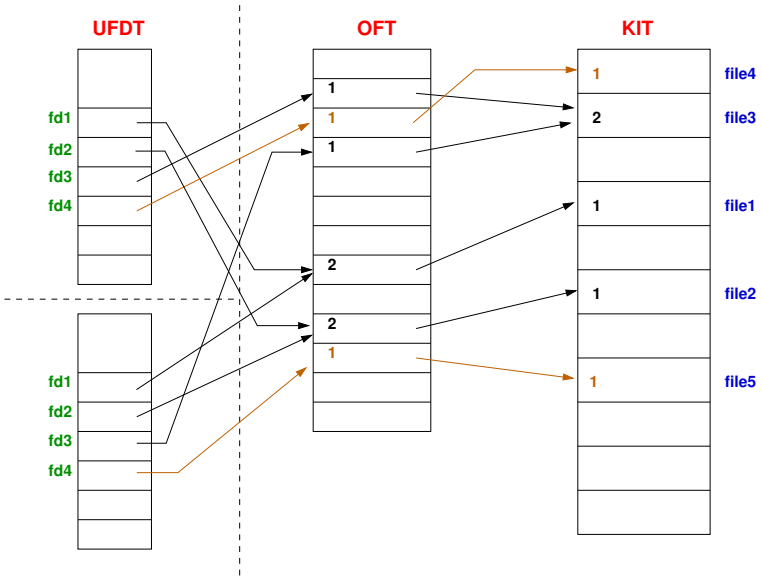


Abbildung 16.11: Kerneltabellen (4)

Nach jedem *close()* wird ein UFDT-Slot frei und der Referenz-Zähler im OFT-Slot dekrementiert. Da der Referenz-Zähler jeweils noch größer als 0 ist, bleibt der OFT-Slot bestehen. *Danach sieht die Situation wie in Abbildung 16.12.*

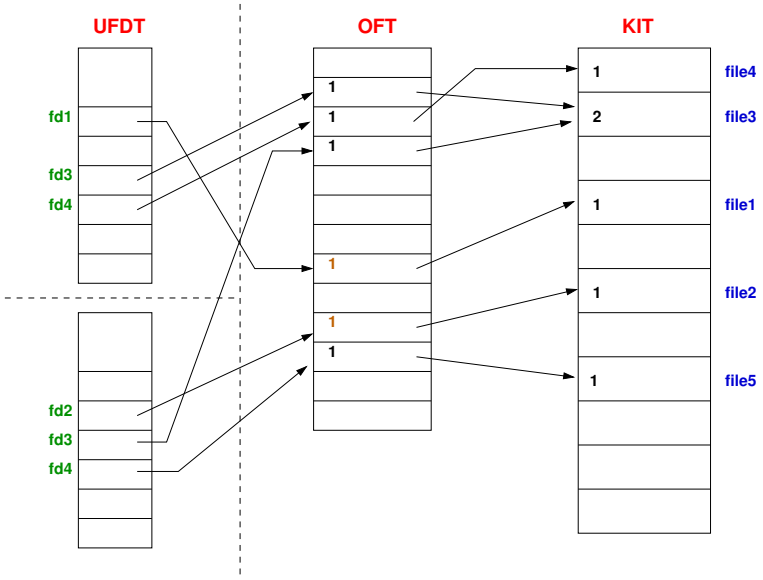


Abbildung 16.12: Kerneltabellen (5)

Kind:

fd5 = *dup*(*fd4*);

dup() liefert den kleinsten freien Filedeskriptor. Dieser ist ja fd1, den wir eben geschlossen haben. Der neue UFDT-Slot zeigt auf den selben OFT-Slot. Deshalb muss auch der Referenz-Zähler im OFT-Slot erhöht werden. *Danach sieht die Situation wie in Abbildung 16.13.*

Kind:

close(*fd4*);
close(*fd5*);

Beim ersten *close()* wird nur der UFDT-Slot freigegeben, da der Referenz-Zähler im OFT-Slot immer noch größer 0 (nämlich 1) ist. Nach dem zweiten *close()* ist der Referenz-Zähler im OFT-Slot gleich 0 und somit wird der OFT-Slot freigegeben und der Referenz-Zähler im KIT-Slot dekrementiert. Dieser Referenz-Zähler geht nun auch auf 0 und somit wird der KIT-Slot freigegeben ($\rightsquigarrow$ auf Platte, wenn Link Count größer 0 und sonst Datei freigeben).

Vater:

close(*fd1*);
close(*fd4*);

Nach jedem dieser *close()*-Aufrufe werden alle Slots freigegeben, da alle Referenz-Zähler auf 0 gehen. *Danach sieht die Situation wie in Abbildung 16.14.*

Kind:

close(*fd2*);
close(*fd3*);

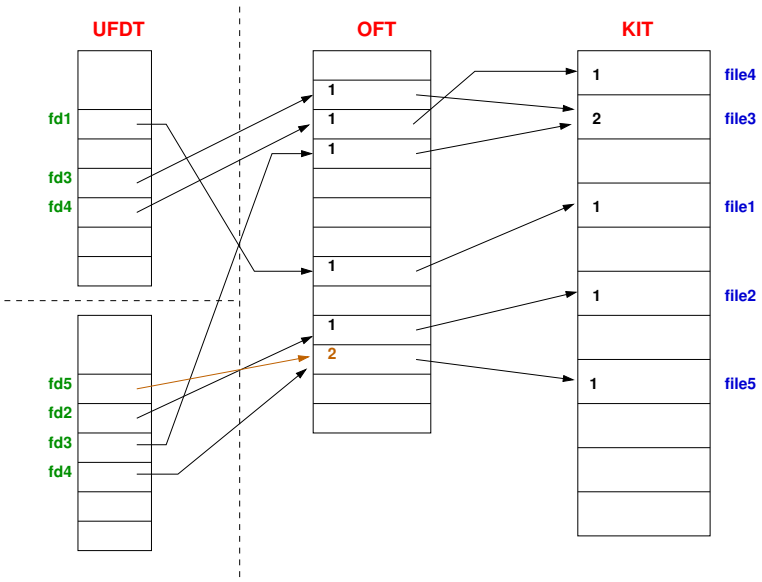


Abbildung 16.13: Kerneltabellen (6)

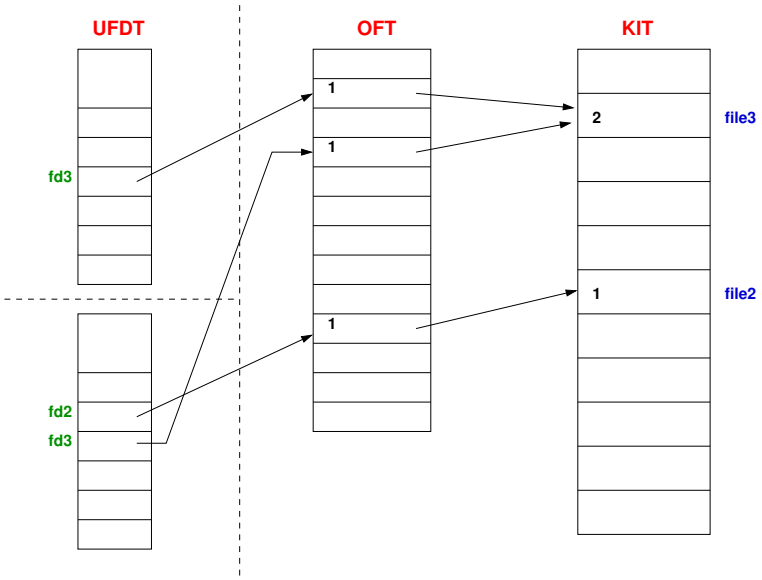


Abbildung 16.14: Kerneltabellen (7)

Alle UFDT-, OFT-, und KIT-Slot werden nach dem ersten *close()* entfernt, da alle Referenz-Zähler auf 0 absinken. Bei dem zweiten *close()* fällt der UFDT-Slot weg und auch der OFT-Slot, da dieser Referenz-Zähler auf 0 absinkt. Der Referenz-Zähler im KIT-Eintrag ist jedoch noch 1, so dass der KIT-Eintrag bleibt.

Vater:

close(*fd3*);

Bei diesem *close()* werden nun alle Slots freigegeben, da der Referenz-Zähler sowohl im OFT-Slot als auch im KIT-Slot auf 0 absinkt.

Jetzt sind alle Tabellen wieder leer (siehe Ausgangszustand in Abbildung 16.7).

16.5 Systemaufrufe für I/O-Verbindungen – Zweiter Teil

16.5.1 Positionieren in Dateien – lseek()

off_t lseek(int fd, off_t offset, int whence);

lseek() verändert den *File Offset Pointer* OFT-Slot. Diesen neuen Startpunkt für die nächste *read()*- oder *write()*-Operation bestimmt *lseek()* aus *offset* und *whence*:

whence	Neue Position
SEEK_SET	Dateianfang plus *offset*
SEEK_CUR	momentane Position plus *offset*
SEEK_END	Dateiende plus *offset*

offset kann auch eine negative Zahl sein. Die symbolischen Konstanten (Makros) für *whence* sind in der Header-Datei *unistd.h* definiert.

lseek() kann nicht auf alle I/O-Verbindungen angewendet werden. Bei einem Terminal oder einer Pipe hat der *File Offset Pointer* keine sinnvolle Bedeutung. *lseek()* signalisiert deshalb einen Fehler mit dem Rückgabewert -1. Im Erfolgsfall gibt *lseek()* den neuen Wert des *File Offset Pointer* zurück.

Das folgende Programm demonstriert die Verwendung von *lseek()*:

Programm 16.11: Positionieren in Dateien – mittels lseek() (*lseek.c*)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    char buf[11];
    int fd, pos;
    if ((fd = open("file", O_RDWR | O_CREAT | O_TRUNC, 0664)) < 0)
        perror("open"), exit(1);

    pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
    printf("Position:␣%d\n", pos);

    if (write(fd, "0123456789\n", 11) != 11) /* String schreiben */
        perror("write"), exit(2);

    pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
    printf("Position:␣%d\n", pos);
```

```
    lseek(fd, 2, SEEK_SET);  /* 2 Bytes hinter den Anfang */

    if (write(fd, "xx", 2) != 2)  /* String schreiben */
        perror("write"), exit(2);

    pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
    printf("Position:␣%d\n", pos);

    lseek(fd, −4, SEEK_END);  /* 4 Bytes vor das Ende */

    if (write(fd, "aa", 2) != 2)  /* String schreiben */
        perror("write"), exit(2);

    pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
    printf("Position:␣%d\n", pos);

    lseek(fd, 0, SEEK_SET); /* an den Anfang positionieren */
    read(fd, buf, 10);  /* 10 Zeichen lesen */
    buf[10] = '\0';  /* String mit Null−Byte terminieren */
    puts(buf);   /* ... und schliesslich ausgeben */

    pos = lseek(fd, 0, SEEK_CUR); /* Position ermitteln */
    printf("Position:␣%d\n", pos);

    return 0;
}
```

```
chomsky$ gcc -Wall lseek.c
chomsky$ a.out
Position: 0
Position: 11
Position: 4
Position: 9
01xx456aa9
Position: 10
chomsky$ cat file
01xx456aa9
chomsky$
```

Implizites lseek:

Enthalten die *oflags* im OFT-Slot einer I/O-Verbindung *O_APPEND*, so verursacht jeder *write()* Systemaufruf vor dem Datentransfer eine implizite *lseek()*-Operation, die den *File Offset Pointer* zuverlässig ans Ende der Datei positioniert.

Zwei explizite Systemaufrufe, ein *lseek()*, gefolgt von einem *write()*, sind in einem Multiuser-/Multitasking-Betriebssystem wie UNIX nicht notwendig eine *atomare* Operation. Ein konkurrierender Prozess könnte seinen *write()* Systemaufruf zwischen dem *lseek()* und dem *write()* Systemaufruf des ersten Prozesses ausführen. Dadurch wäre der *File Offset Pointer* des ersten Prozesses nicht mehr auf das neue Ende der Datei positioniert und der erste Prozess würde die Daten des zweiten Prozesses überschreiben. Das *O_APPEND* Flag ermöglicht dagegen das sichere und geordnete Anfügen an eine Datei. Der implizite *lseek()* und der Systemaufruf *write()* laufen als eine atomare Operation ab.

16.5.2 Erzeugen von Links – link(), symlink()

int link(const char *oldpath, const char *newpath);
int symlink(const char *oldpath, const char *newpath);

Mit *link()* kann man einen Hardlink erzeugen – analog zum Kommando *ln*. Mit *symlink()* kann man – analog zum Kommando *ln* mit der Option *-s* – einen Softlink erzeugen. Der Rückgabewert ist jeweils 0 bei Erfolg und -1 bei Misserfolg.

Das Anlegen eines Links funktioniert nur dann, wenn noch keine Datei mit diesem Namen bereits existiert ($\rightsquigarrow$ zur Synchronisation geeignet, genauso wie *open()* mit *O_EXCL* – später mehr dazu).

16.5.3 Entfernen von Dateinamen – unlink()

int unlink(const char *pathname);

Der Systemcall *unlink()* entfernt den angegebenen Dateinamen. Konsequenterweise wird der Link Count in der Inode dekrementiert. Sinkt der Link Count dabei auf 0, so werden die Inode und die Datenblöcke sofort freigegeben, wenn die Inode nicht in der KIT ist. Andernfalls werden die Inode und die Datenblöcke dann freigegeben, wenn die Inode aus der KIT entfernt wird (nachdem der Referenz-Zähler auf 0 gesunken ist). Im Erfolgsfall liefert *unlink()* 0 und -1 bei einem Fehler.

Besonderheit:
Das I/O-Subsystem verzögert das Freigeben von Inode und Plattenplatz solange noch ein Prozess die Datei geöffnet hat. Da jedoch der Verzeichnis-Eintrag entfernt wurde, ist ein weiterer Zugriff auf die Datei, mit *open()* oder *stat()*, über den Namen nicht mehr möglich. Die Filedeskriptoren der Prozesse bilden die „letzte“ Verbindung zu der Datei. Sie können für *read()*, *write()*, *fstat()*, ..., und *close()* benutzt werden. Nach dem letzten expliziten oder impliziten *close()* Systemaufruf mit einem Filedeskriptor, der noch auf die Datei verwiesen hat, gibt das I/O-Subsystem die Daten endgültig frei.

Dieses Verhalten des Kernels vereinfacht den Gebrauch (und das ordentliche Entfernen) von *temporären Dateien*. Das folgende Programm demonstriert das Arbeiten mit temporären Dateien:

Programm 16.12: Erzeugen einer temporären Datei (*temp.c*)

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

#define TMPNAME "/tmp/jmayerXXXXXX" /* XXXXXX am Ende wg. mktemp! */
#define BUFSIZE 128

int main() {
    char tmp[BUFSIZE];
    int fd;

    /* temp. Dateiname mit mktemp() erzeugen;
     * "XXXXXX" muss am Ende des Templates stehen und wird ersetzt */
    strcpy(tmp, TMPNAME);
    if (!mktemp(tmp)) /* Parameter von mktemp() wird veraendert! */
        perror("mktemp"), exit(1);

    /* temporaere Datei oeffnen -- aber nur, wenn sie noch nicht existiert */
    if ((fd = open(tmp, O_RDWR | O_CREAT | O_TRUNC | O_EXCL, 0600)) < 0)
```

```
        perror("open"), exit(2);

    /* danach Dateinamen loeschen -- I/O-Verb. zur Datei immer noch via fd! */
    unlink(tmp);

    /* Arbeiten mit der temp. Datei */

    /* nun aber wirklich die Inode und den Plattenplatz freigeben */
    close(fd);

    return 0;
}
```

Zum Erzeugen des Dateinamens wird die Funktion *mktemp()* verwendet:
char *mktemp(char *template);
Diese Funktion erwartet, dass am Ende von *template* der String „XXXXXX“ steht. Dieser wird ersetzt durch eine Zeichenfolge, so dass der Dateiname noch nicht existiert. Dabei wird der Parameter verändert! Der Rückgabewert ist ungleich dem Null-Zeiger, wenn alles ohne Schwierigkeiten verlief. Im Falle eines Fehlers liefert diese Funktion den Null-Zeiger.

Der Prozess kann die temporäre Datei benutzen, wenn er jedoch – egal aus welchem Grund – terminiert, gibt das I/O-Subsystem automatisch allen Plattenplatz frei, den die Datei belegt hat. Es entstehen keine unliebsamen „Überbleibsel“.

Temporäre Dateien legt man (aus Performance-Gründen) immer im Katalog */tmp* an, da dieser lokal auf der Maschine ist und meist sogar im Hauptspeicher (⇝ *tmpfs*).

16.5.4 Ändern der oflags – fcntl()

int fcntl(int fd, int cmd, long arg);
fcntl() verändert die Eigenschaften von bereits offenen I/O-Verbindungen. Er manipuliert die *oflag*-Bits im OFT-Slot. Damit können nachträglich Flagbits wie *O_APPEND* oder *O_SYNC* für eine I/O-Verbindung gesetzt oder gelöscht werden.

Der Parameter *cmd* kann entweder den Wert *F_GETFL* oder den Wert *F_SETFL*. Im ersten Fall liefert *fcntl()* die oflags als Rückgabewert und ignoriert den dritten Parameter. Im zweiten Fall setzt *fcntl()* die oflags auf den in *arg* angegebenen Wert. Der Rückgabewert −1 signalisiert eine Fehlersituation.

Programm 16.13: oflags verändern – mittels fcntl() (*fcntl.c*)

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

void clear_append(int fd) {
    int oflags;
        /* oflags mittels fcntl() ermitteln */
    if ((oflags = fcntl(fd, F_GETFL, 0)) < 0)
        perror("fcntl"), exit(1);

    /* Ist O_APPEND gesetzt? Wenn nein, dann fertig! */
    if ((oflags & O_APPEND) != O_APPEND)
        return;

    oflags &= ~O_APPEND; /* O_APPEND loeschen */
```

```
    /* veraenderte oflags setzen */
    if (fcntl(fd, F_SETFL, oflags) < 0)
        perror("fcntl"), exit(2);
}

int main() {
    char buf[20];
    int fd, n;

    if ((fd = open("file", O_RDWR | O_CREAT | O_TRUNC | O_APPEND, 0644)) < 0)
        perror("open"), exit(3);

    write(fd, "aaaaa", 5); /* "aaaaa" in die Datei schreiben */

    lseek(fd, 0, SEEK_SET);
    write(fd, "bbbb", 4); /* haengt "bbbb" an -- wg. O_APPEND! */

    clear_append(fd);  /* O_APPEND loeschen */

    lseek(fd, 0, SEEK_SET);
    write(fd, "ccc", 3); /* jetzt wird "ccc" tatsaechlich am Anfang geschr. */

    lseek(fd, 0, SEEK_SET); /* vom Anfang an ... */
    n = read(fd, buf, 19); /* auslesen ...*/
    buf[n] = '\0';
    puts(buf);   /* ... und ausgeben */

    close(fd);   /* wuerde auch implizit passieren */

    return 0;
}
```

```
chomsky$ gcc -Wall fcntl.c
chomsky$ a.out
cccaabbbb
chomsky$ cat file; echo
cccaabbbb
chomsky$
```

16.5.5 ioctl()

int ioctl(int fd, int cmd, long arg);

ioctl() verändert die Eigenschaften von offenen I/O-Verbindungen zu Geräten oder IPC-Kanälen. Ein Prozess kann damit direkt auf einen *Gerätetreiber* Einfluss nehmen. Deshalb sind auch die erlaubten Kommandos sehr abhängig von den einzelnen Gerätetreibern und davon, welche Treibermodule in den Kernel konfiguriert wurden (siehe dazu die entsprechenden Manpages!). Die Verwendung ist analog zu *fcntl()*.

Typischerweise will man bei Passworteingaben nicht haben, dass der eingegebene Text auch gleichzeitig ausgegeben wird. Die wir mit folgendem Programm, das *ECHO* aus- und wieder einschaltet, erreicht.

Programm 16.14: Echo aus- und anschalten – mittels ioctl() (*echo.c*)

```
#include <unistd.h>
#include <termio.h>
#include <stdio.h>

void echo_off(int fd) {
    struct termio buf;
    /* Flags lesen ... */
    if (ioctl(fd, TCGETA, &buf) < 0)
        perror("ioctl"), exit(1);

    /* ECHO nicht gesetzt? => nichts zu tun */
    if ((buf.c_lflag & ECHO) != ECHO)
        return;

    buf.c_lflag &= ~ECHO; /* ECHO loeschen */

    /* Flags setzen ... */
    if (ioctl(fd, TCSETA, &buf) < 0)
        perror("ioctl"), exit(2);
}

void echo_on(int fd) {
    struct termio buf;
    /* Flags lesen ... */
    if (ioctl(fd, TCGETA, &buf) < 0)
        perror("ioctl"), exit(1);

    /* ECHO gesetzt? => nichts zu tun */
    if ((buf.c_lflag & ECHO) == ECHO)
        return;

    buf.c_lflag |= ECHO; /* ECHO setzen */

    /* Flags setzen ... */
    if (ioctl(fd, TCSETA, &buf) < 0)
        perror("ioctl"), exit(2);
}

int main() {
    char buf[128];

    printf("Name:␣");
    fgets(buf, 128, stdin);
    buf[strlen(buf)-1] = '\0'; /* Newline entfernen */
    printf("Ihr␣Name:␣%s\n", buf);

    printf("Passwort:␣");
    echo_off(0);  /* Echo ausschalten */
    fgets(buf, 128, stdin);
    buf[strlen(buf)-1] = '\0'; /* Newline entfernen */
    echo_on(0);   /* ... und Echo wieder einschalten */
    puts("");
```

```
    printf("Ihr␣Passwort:␣%s\n", buf);

    return 0;
}
```

16.6 Synchronisation

16.6.1 Generelles

Problem: Nebenläufigkeit

Mehrere Prozesse arbeiten lesend und schreibend auf einer Datei. Dabei kann es durchaus passieren, dass ein Prozess einen Datensatz liest, der während dessen von einem anderen Prozess in Teilen verändert wird. Mit einem einfachen Beispiel soll das Problem verdeutlicht werden.

Beispiel:

Mehrere Prozesse benötigen einen fortlaufenden Zähler, z. B. um für Datensätze einen internen Schlüssel zu vergeben; dazu lesen sie die Zahl für den nächsten Schlüssel aus einer Datei und schreiben ihn inkrementiert wieder zurück. In der folgenden Implementierung wird zwischen dem Lesen der Zahl und dem Zurückschreiben eine zufällig gewählte Zeit gewartet. Dieses Warten (Suspendieren) des Prozesses kann mit *sleep()* realisiert werden; die Zufallszahl für die Anzahl der zu wartenden Sekunden kann mit dem Pseudo-Zufallszahlengenerator *random()*, dessen Startwert mit der Funktion *srandom()* gesetzt werden kann, bestimmt werden:

Programm 16.15: Inkrementieren ohne Synchronisation (*nolock.c*)

```
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>

#define MAXDELAY 5
#define MAXREPEAT 5
#define BUFSIZE  128
#define SEQFILE  "seqno"

/* Zaehler lesen, delay Sekunden schlafen und dann erhoehen */
void incr(int delay) {
    int fd, seqno, n;
    char buf[BUFSIZE];

    if ((fd = open(SEQFILE, O_RDWR)) < 0)
        perror("open"), exit(2);

        /* Dateiinhalt in den Puffer einlesen */
    if ((n = read(fd, buf, BUFSIZE)) < 0)
        perror("read"), exit(3);
        /* ... und mit Null−Byte terminieren */
    buf[n] = '\0';
```

```
        /* Integer parsen */
    if (sscanf(buf, "%d", &seqno) != 1)
        fprintf(stderr, "sscanf␣error\n"), exit(4);

    printf("gelesen:␣%d\n", seqno);

    seqno++; /* Zaehler erhoehen */

    sleep(delay); /* delay Sekunden schlafen */

    sprintf(buf, "%03d\n", seqno); /* neuen Zaehler als String */
    n = strlen(buf);

    lseek(fd, 0, SEEK_SET);  /* an den Dateianfang gehen */

    if (write(fd, buf, n) != n)  /* ... und den Zaehler schreiben */
        perror("write"), exit(5);

    close(fd);
}

int main(int argc, char **argv) {
    int i, delay;

    if (argc != 2) {
        fprintf(stderr, "Usage:␣%s␣<seed>\n", argv[0]);
        exit(1);
    }

    /* Pseudo−Zufallszahlengenerator initialisieren */
    srandom(atoi(argv[1]));

    /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
    sleep(delay = random() % (MAXDELAY+1));

    for (i = 1; i <= MAXREPEAT; i++) {
        /* gemeinsamen Zaehler erhoehen */
        incr(delay);
        /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
        sleep(delay = random() % (MAXDELAY+1));
    }

    return 0;
}
```

```
thales$ gcc -Wall -o nolock nolock.c
thales$ cat seqno
001
thales$ nolock 2 > out1 & nolock 5 > out2 &
[1] 14289
[2] 14290
thales$
[1]-  Done                    nolock 2 >out1
[2]+  Done                    nolock 5 >out2
thales$ cat out1
gelesen: 1
gelesen: 3
gelesen: 4
gelesen: 6
gelesen: 7
thales$ cat out2
gelesen: 1
gelesen: 2
gelesen: 4
gelesen: 5
gelesen: 6
thales$ cat seqno
008
thales$
```

Wie man hier leicht sieht, steht der gemeinsame Zähler nach der Ausführung nicht auf 11. Jeder der beiden Prozesse sollte aber um 5 weiterzählen. Folgendes Fehlerszenario ist hierbei (mehrmals) aufgetreten:

1. Prozess A liest den Zähler
2. Prozess B liest den Zähler
3. Prozess A schreibt den inkrementierten Zähler
4. Prozess B schreibt den inkrementierten Zähler

Dabei erhalten A und B denselben Zählerwert und B überschreibt die Änderung von A (hier nicht tragisch). Eigentlich müsste das Lesen des Zählers und das Schreiben des inkrementierten Zählerwertes „auf einmal“ erfolgen, so dass kein anderer Prozess in der Zeit etwas schreiben oder lesen kann. Prozess A müsste also die Datei vor dem Lesen für sich sperren können und würde dann erst nach dem Schreiben des inkrementierten Wertes diese Sperre wieder aufheben.

Lösung: Semaphoren

Eine Semaphore ist eine „Ampel“, die immer nur *einem* Prozess grün (Datei benutzen erlaubt) und allen anderen Prozessen „rot“ (Benutzung z. Zt. nicht erlaubt) signalisiert.

Alle Prozesse verpflichten sich, nur bei „grün“ bestimmten Code auszuführen, z. B. auf eine gemeinsame Datei zuzugreifen. Dadurch, dass immer nur ein Prozess „grün“ erhält (*mutual exclusion*), synchronisieren sich die Prozesse über die Semaphore.

Eine Semaphore bewacht die Anweisungen des sog. *kritischen Bereichs*. Bevor ein Prozess den kritischen Bereich betritt, muss er sich von der Semaphore „grün“ signalisieren lassen (warten, bis „grün“ gesetzt ist). Bei Betreten des kritischen Bereichs wird die Semaphore auf „rot“ gesetzt, wenn der Prozess den kritischen Teil verläßt, teilt er dies der Semaphore mit (wieder auf „grün“ setzen). Anschließend kann die Semaphore einem anderen Prozess „grün“ signalisieren. Der Notation des Informatikers *Dijkstra* folgend nennt

man diese Operationen *P(sema)* (Semaphore für sich reservieren – *protect*) und *V(sema)* (Semaphore freigeben – holl. *vrej*).

In einem Programm könnte dies etwa folgendermaßen aussehen:

```
/* ... */

P(sem); /* Semaphore reservieren */

    /*
     * kritischer Bereich
     */

V(sem); /* Semaphore freigeben */

/* ... */
```

Anmerkung: Ist die Semaphore auf „rot“, so wartet der Prozess (Operation *P()*) so lange, bis sie (von wem auch immer) auf „grün“ gesetzt wird.

Problem: Deadlock

- Zwei Prozesse arbeiten z. B. gleichzeitig auf zwei Dateien.
- Jeder hat eine Datei gesperrt und wartet darauf, dass der andere seine Datei freigibt, um weiterarbeiten zu können.

16.6.2 Synchronisation mit open() und O_EXCL

In folgendem Programm sind die beiden Funktionen *my_lock()* und *my_unlock()* neu hinzugekommen. Außerdem wurde um die Anweisungen in *incr()* eine *P()*- und eine *V()*-Operation eingebaut. Der Rest ist identisch mit vorigem Beispiel. (Die *P()*-Operation besteht aus einem *open()*-Aufruf mit *O_EXCL*. Die *V()*-Operation besteht aus einem *unlink()*-Aufruf.)

Programm 16.16: Inkrementieren mit Synchronisation (*lock.c*)

```
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#define MAXDELAY 5
#define MAXREPEAT 5
#define BUFSIZE  128
#define SEQFILE  "seqno"
#define SEMA  "/tmp/mysem"
#define MAXTRIES 5
#define WAITTIME 1
#define P(sem)  if (!my_lock(sem)) \
                        fprintf(stderr, "my_lock␣failed\n"), exit(6);
#define V(sem)  if (!my_unlock(sem)) \
                        fprintf(stderr, "my_unlock␣failed\n"), exit(7);

/* Zugriff ueber die Semaphore sem (Datei!) synchronisieren
```

```
 * Rueckgabewert: != 0 bei Erfolg
 */
int my_lock(char *sem) {
    int ret, tries = 0;
        /* Datei exklusiv anlegen => falls sie existiert gibt's einen Fehler;
         * das Ueberpruefen, ob die Datei existiert, und das Anlegen ist eine
         * _atomare_ Operation bei open() mit O_EXCL! */
    while ((ret = open(sem, O_RDWR | O_CREAT | O_EXCL, 0664)) < 0 && errno ==
        EEXIST) {
        if (++tries > MAXTRIES) /* ... das ganze nur MAXTRIES Mal versuchen */
            return 0;
        sleep(WAITTIME); /* ... und dazwischen immer ein kleines Schlaefchen */
    }
    return ret >= 0;
}

/* Semaphore wieder freigeben (d.h. Datei loeschen)
 * Rueckgabewert: != 0 bei Erfolg
 */
int my_unlock(char *sem) {
    return unlink(sem) == 0;
}

/* Zaehler lesen, delay Sekunden schlafen und dann erhoehen */
void incr(int delay) {
    int fd, seqno, n;
    char buf[BUFSIZE];

    P(SEMA) /* kritischen Bereich beginnen; Lock anfordern */

    puts("BEGIN␣kritischer␣Bereich");

    if ((fd = open(SEQFILE, O_RDWR)) < 0)
        perror("open"), exit(2);

        /* Dateiinhalt in den Puffer einlesen */
    if ((n = read(fd, buf, BUFSIZE)) < 0)
        perror("read"), exit(3);
        /* ... und mit Null-Byte terminieren */
    buf[n] = '\0';

        /* Integer parsen */
    if (sscanf(buf, "%d", &seqno) != 1)
        fprintf(stderr, "sscanf␣error\n"), exit(4);

    printf("gelesen:␣%d\n", seqno);

    seqno++; /* Zaehler erhoehen */

    sleep(delay); /* delay Sekunden schlafen */

    sprintf(buf, "%03d\n", seqno); /* neuen Zaehler als String */
    n = strlen(buf);
```

```
    lseek(fd, 0, SEEK_SET);  /* an den Dateianfang gehen */

    if (write(fd, buf, n) != n)  /* ... und den Zaehler schreiben */
        perror("write"), exit(5);

    close(fd);

    puts("END␣kritischer␣Bereich");

    V(SEMA) /* kritischen Bereich verlassen; Lock wieder freigeben */
}

int main(int argc, char **argv) {
    int i, delay;

    if (argc != 2) {
        fprintf(stderr, "Usage:␣%s␣<seed>\n", argv[0]);
        exit(1);
    }

    /* Pseudo−Zufallszahlengenerator initialisieren */
    srandom(atoi(argv[1]));

    /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
    sleep(delay = random() % (MAXDELAY+1));

    for (i = 1; i <= MAXREPEAT; i++) {
        /* gemeinsamen Zaehler erhoehen */
        incr(delay);
        /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
        sleep(delay = random() % (MAXDELAY+1));
    }

    return 0;
}
```

```
thales$ gcc -Wall -o lock lock.c
thales$ cat seqno
001
thales$ lock 2 > out1 & lock 5 > out2 &
[1] 27992
[2] 27993
thales$
[1]-  Done                    lock 2 >out1
[2]+  Done                    lock 5 >out2
thales$ cat out1
BEGIN kritischer Bereich
gelesen: 1
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 4
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 6
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 8
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 10
END kritischer Bereich
thales$ cat out2
BEGIN kritischer Bereich
gelesen: 2
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 3
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 5
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 7
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 9
END kritischer Bereich
thales$ cat seqno
011
thales$
```

Anmerkung: Die zur Synchronisation verwendete Datei wird im Katalog */tmp* angelegt, um sicher zu gehen, dass sie lokal auf dem Rechner ist und kein NFS im Spiel ist.

16.6.3 Synchronisation mit lockf()

int lockf(int fd, int function, off_t size);

File Locking und *Record Locking* wurde erst relativ spät in UNIX eingeführt. Einige ältere System V Versionen erhielten einen System Call *lockf()*, neuere (aktuelle) Versionen

erweitern die Funktionalität von *fcntl()* und stellen eine Bibliotheksfunktion *lockf()* zur Verfügung.

Im einfachsten Fall kann *lockf()* eine Datei komplett vor simultanem Zugriff schützen und somit als Semaphore brauchbar machen. *lockf()* bietet aber zusätzlich die Möglichkeit, auf genau definierten Teilen einer Datei exklusiven Zugriff durchzusetzen.

lockf() unterstützt folgende Kommandos (Parameter *function*):

Kommando	*Bedeutung*
F_LOCK	Einen Bereich der Datei für exklusiven Gebrauch nur durch den ausführenden Prozess reservieren.
F_ULOCK	Vorher reservierten Bereich der Datei wieder freigeben.
F_TEST	Einen Bereich der Datei auf Reservierungen durch andere Prozesse überprüfen.
F_TLOCK	Falls in dem Bereich keine Reservierung durch einen anderen Prozess besteht, den Bereich für den ausführenden Prozess reservieren, sonst Rückkehr mit Fehleranzeige. Prozess blockiert nicht!

Der Bereich beginnt beim *File Offset Pointer* und hat die Länge *size*. Ein Wert von 0 steht für unendlich. In diesem Fall reicht der Bereich also immer bis zum Ende der Datei – egal wie groß die Datei wird.

Der Rückgabewert von *lockf()* ist 0 im Erfolgsfall und -1, falls ein Fehler aufgetreten ist. Die beiden Werte *EACCES* und *EAGAIN* signalisieren bei *F_TEST* bzw. *F_TLOCK*, dass eine Reservierung nicht möglich ist.

Ein Aufruf mit *F_LOCK* blockiert so lange, bis die Reservierung gesetzt werden kann. Um das Blockieren zu verhindern, könnte man eine Sequenz von *lockf()*-Aufrufen mit *F_TEST* und *F_LOCK* benutzen. Zwischen den beiden Aufrufen kann aber ein anderer Prozess erfolgreich *F_LOCK* anwenden ($\rightsquigarrow$ keine *atomare Operation*), worauf der erste Prozess doch blockiert. Abhilfe schafft hier *F_TLOCK*, was *F_TEST* und *F_LOCK* zu einer *atomaren, nicht blockierenden Operation* zusammenfasst.

Die *P()*-Operation besteht aus (evtl. wiederholten) *lockf()*-Aufrufen mit *F_TLOCK* als Kommando.

Die *V()*-Operation besteht aus einem *lockf()*-Aufruf mit *F_ULOCK* als Kommando.

Das folgende Programm demonstriert die Verwendung von *lockf()* an dem bekannten Beispiel:

Programm 16.17: Inkrementieren mit Synchronisation – mittels lockf() (*lockf.c*)

```
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

#define MAXDELAY 5
#define MAXREPEAT 5
#define BUFSIZE  128
#define SEQFILE  "seqno"
#define MAXTRIES 5
#define WAITTIME 1
#define P(sem)  if (!my_lock(sem)) \
                        perror("my_lock"), exit(6);
```

```
#define V(sem)  if (!my_unlock(sem)) \
                                perror("my_unlock"), exit(7);

/* Zugriff ueber die Semaphore sem (Filedeskriptor) synchronisieren
 * Rueckgabewert: != 0 bei Erfolg
 */
int my_lock(int sem) {
    int ret, tries = 0;
    /* zuerst auf den Anfang der Datei positionieren und dann mittels
     * lockf() bis zum Dateiende alles sperren
     */
    while ((ret = lseek(sem, 0, SEEK_SET)) == 0  /* an den Anfang gehen ... */
            && (ret = lockf(sem, F_TLOCK, 0)) < 0 /* ... und bis zum Ende sperren */
            && (errno == EACCES || errno == EAGAIN)) {
        if (++tries > MAXTRIES) /* ... das ganze nur MAXTRIES Mal versuchen */
            return 0;
        sleep(WAITTIME); /* ... und dazwischen immer ein kleines Schlaefchen */
    }
    return ret == 0;
}

/* Semaphore wieder freigeben (d.h. Datei loeschen)
 * Rueckgabewert: != 0 bei Erfolg
 */
int my_unlock(int sem) {
    return lseek(sem, 0, SEEK_SET) == 0  /* an den Anfang gehen ... */
            && lockf(sem, F_ULOCK, 0) == 0; /* ... und bis zum Ende freigeben */
}

/* Zaehler lesen, delay Sekunden schlafen und dann erhoehen */
void incr(int delay) {
    int fd, seqno, n;
    char buf[BUFSIZE];

    if ((fd = open(SEQFILE, O_RDWR)) < 0)
        perror("open"), exit(2);

    P(fd) /* kritischen Bereich beginnen; Lock anfordern */

    puts("BEGIN␣kritischer␣Bereich");

        /* Dateiinhalt in den Puffer einlesen */
    if ((n = read(fd, buf, BUFSIZE)) < 0)
        perror("read"), exit(3);
        /* ... und mit Null-Byte terminieren */
    buf[n] = '\0';

        /* Integer parsen */
    if (sscanf(buf, "%d", &seqno) != 1)
        fprintf(stderr, "sscanf␣error\n"), exit(4);

    printf("gelesen:␣%d\n", seqno);
```

```
    seqno++; /* Zaehler erhoehen */

    sleep(delay); /* delay Sekunden schlafen */

    sprintf(buf, "%03d\n", seqno); /* neuen Zaehler als String */
    n = strlen(buf);

    lseek(fd, 0, SEEK_SET);  /* an den Dateianfang gehen */

    if (write(fd, buf, n) != n)  /* ... und den Zaehler schreiben */
        perror("write"), exit(5);

    puts("END␣kritischer␣Bereich");

    V(fd) /* kritischen Bereich verlassen; Lock wieder freigeben */

    close(fd);
}

int main(int argc, char **argv) {
    int i, delay;

    if (argc != 2) {
        fprintf(stderr, "Usage:␣%s␣<seed>\n", argv[0]);
        exit(1);
    }

    /* Pseudo-Zufallszahlengenerator initialisieren */
    srandom(atoi(argv[1]));

    /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
    sleep(delay = random() % (MAXDELAY+1));

    for (i = 1; i <= MAXREPEAT; i++) {
        /* gemeinsamen Zaehler erhoehen */
        incr(delay);
        /* zufaellig zwischen 0 und MAXDELAY Sekunden schlafen */
        sleep(delay = random() % (MAXDELAY+1));
    }

    return 0;
}
```

Anmerkung: Die wesentlichen Änderungen sind die andere Implementierung von *my_lock()* und *my_unlock()* – unter Verwendung von *lockf()* und *lseek()* (um die gesamte Datei zu sperren) – und die Verwendung des Filedeskriptors zum Sperren. Deswegen müssen jetzt die *P()*- und die *V()*-Operation innerhalb von *open()* und *close()* sein.

```
thales$ gcc -Wall -o lockf lockf.c
thales$ cat seqno
001
thales$ lockf 2 > out1 & lockf 5 > out2 &
[3] 16357
[4] 16358
thales$
[3]-  Done                    lockf 2 >out1
[4]+  Done                    lockf 5 >out2
thales$ cat out1
BEGIN kritischer Bereich
gelesen: 1
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 4
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 6
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 8
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 10
END kritischer Bereich
thales$ cat out2
BEGIN kritischer Bereich
gelesen: 2
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 3
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 5
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 7
END kritischer Bereich
BEGIN kritischer Bereich
gelesen: 9
END kritischer Bereich
thales$ cat seqno
011
thales$
```

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 für 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 für 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