Allgemeine Informatik III
(Systemnahe Software I)

F. Schweiggert, A. Borchert, M. Grabert und J. Mayer

13. Februar 2006

Fakultät Mathematik u. Wirtschaftswissenschaften
Abteilung Angewandte Informationsverarbeitung
Vorlesungsbegleiter (gültig ab WS 2003/2004)

Hinweise:

- Auf eine detaillierte Unterscheidung zwischen *BSD Unix* und *System V Unix* wird hier verzichtet.
- 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 System-Calls sollten jeweils im **Manual** bzw. den entsprechenden Header-Files nachgelesen werden.
- Die Sprache C dient in erster Linie als *Werkzeug* zur Darstellung systemnaher Konzepte!

Inhaltsverzeichnis

Teil I

Die Programmiersprache C

Kapitel 1

Entstehungsgeschichte

Abb. 1.1 zeigt eine vereinfachte Darstellung der Entwicklung einiger Programmiersprachen.

Die Programmiersprache C

- Wurde 1972-73 von Dennis Ritchie bei den Bell Laboratories von AT&T entwickelt.
- 1978 erfolgten einige Erweiterungen von C (*enum, void, structure assignment, ...*), die zum sog. Kernighan&Ritchie-Standard (**K&R-Standard**) führten.
- Der *ANSI-Standard* beinhaltet eine Reihe von Erweiterungen und Aufräumarbeiten.
- *UNIX ist zum größten Teil in C* geschrieben, ein UNIX-Kern besteht nur aus etwa 5-10% Assemblertext.
- C ist eine maschinennahe Sprache. Array's sind Speicherflächen im Hauptspeicher; Array-Namen werden als Zeiger auf das erste Element aufgefasst; der Zugriff auf Elemente erfolgt via Zeigerarithmetik (Adress-Rechnung).
- C wurde mit UNIX verbreitet und ist damit eine der „portabelsten“ Plattformen. (*Achtung bei der Portabilität der C-Bibliotheken!*)
- In der Vorlesung und in den Übungen wird primär mit dem *GNU-C(++)-Compiler* gearbeitet, der auch für Windows und DOS erhältlich ist (siehe Homepage zur Vorlesung).
- **Literatur:** Jedes Buch, das sich mit C direkt beschäftigt (z. B. [Kernighan90]). Bücher über C++ sind aufgrund der Komplexität der objektorientierten Konzepte für die Vorlesung nicht empfehlenswert.

Andere Programmiersprachen

- *C++* und *Objective C* sind sog. objektorientierte Erweiterungen von C.
- *Java* ist eine objektorientierte Programmiersprache, die auf C und Oberon basiert und von Sun Microsystems Inc. entwickelt wurde. Sie ist im Prinzip plattformunabhängig.
- *C#* (*C Sharp*) ist eine aktuelle Neuentwicklung (Mitte 2000) von Microsoft und vereint Konzepte von Visual Basic, Java und C++, ist aber stark mit der Windows-Plattform verbunden (*.NET*).

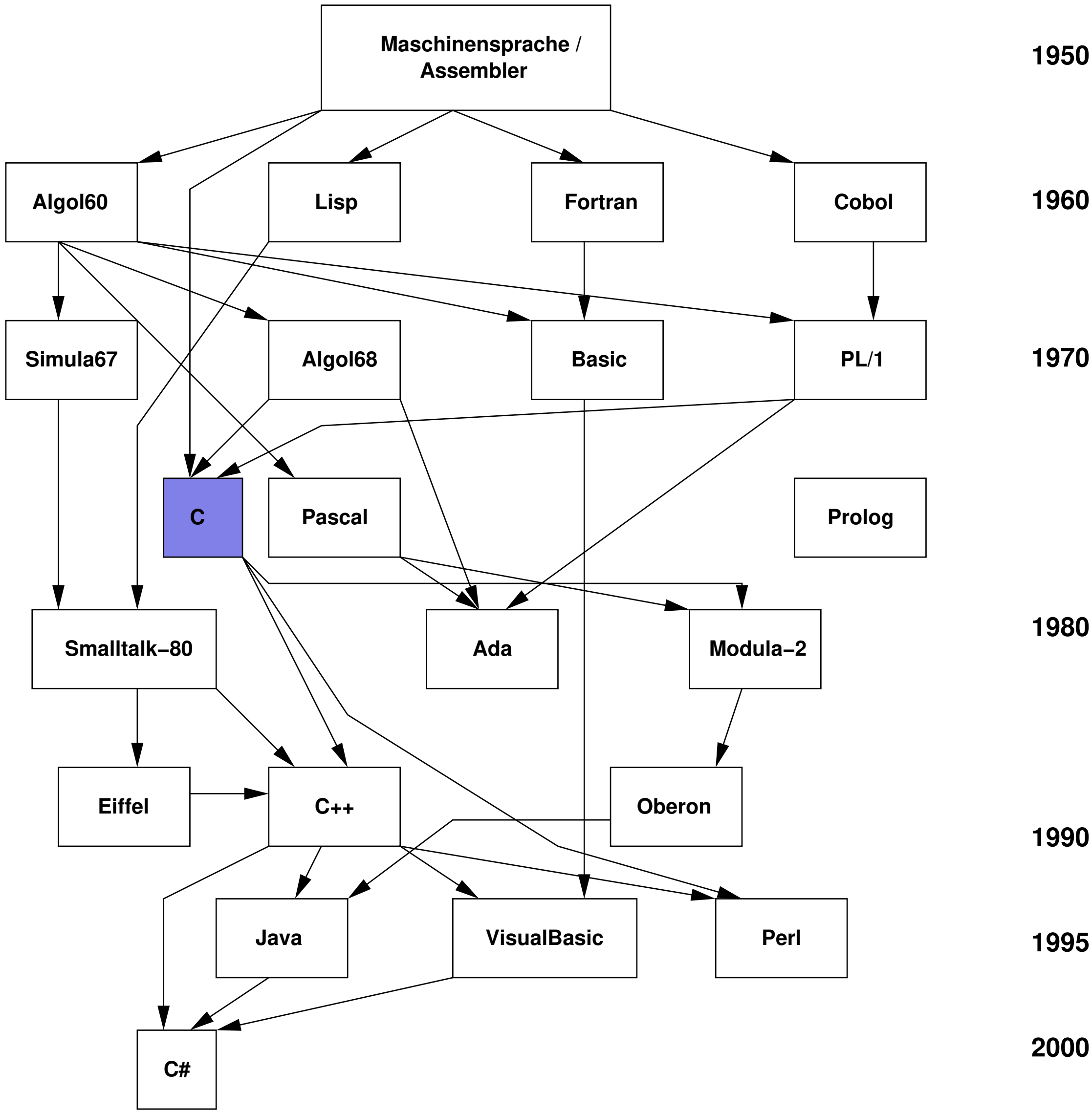


Abbildung 1.1: Entwicklung einiger Programmiersprachen

Kapitel 2

Erste Schritte mit C

Bevor wir uns mit dem Aufbau und der Syntax eines C-Programms beschäftigen folgen nun erst einmal ein paar Beispiele, um ein „Gefühl“ für die Sprache C zu bekommen.

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*)

```c
main() {
        /* puts = Ausgabe eines Strings nach stdout */
    puts("Hallo zusammen!");
}
```

Übersetzung und Ausführung:

```
thales$ 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
thales$ a.out
Hallo zusammen!
thales$
```

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

2.1.2 Eine bessere Welt

An den Warnungen bei der Ausführung von Programm 2.1 erkennt man, 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*)

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

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

Folgende Änderungen sind (gegenüber Programm 2.1) erfolgt:

- Da die Ausgabefunktion *puts()* nicht bekannt war, hat der Compiler 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 Integer angegeben (der Compiler hat vorher auch **int** geraten.)
- Der Rückgabewert der main()-Funktion, welcher durch *return 0* gesetzt wird, ist der *Exit-Status* des Programms.

  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).

  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:

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

Mit der Option *-o* kann man den Namen der Ausgabedatei beim Aufruf des gcc angeben.

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*)

```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 */

    return 0;
}
```

An obigem Programm-Beispiel kann man erkennen, wie *globale Variablen und lokale Variablen vereinbart* werden können. Die globale Variable wurde als *Konstante* vereinbart. Außerdem wird die Funktion *printf()* zur formatierten Ausgabe verwendet. Mit einer *for-Schleife* werden die ersten 20 natürlichen Zahlen durchlaufen.

Programm 2.4 ist mit einer *while-Schleife* implementiert. Hier sieht man, was bei der for-Schleife in Programm 2.3 genau passiert.

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

```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 */
    }

    return 0;
}
```

2.1.4 Euklidscher Algorithmus

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

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

```c
#include <stdio.h>

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

    printf("Geben Sie zwei pos. 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)      /* kommt nur ein Statement, so koennen */
            x = x - y;  /* die { } weggelassen werden */
        else
            y = y - x;
    }

    printf("ggT von %d und %d ist %d\n", x0, y0, x);

    return 0;
}
```

Die Programmiersprache C kennt nur *Wertparameter-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. Soweit dazu – später beschäftigen wird uns noch ausführlich mit Zeigern.

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

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

```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 pos. ganze Zahlen ein:");
    if (scanf("%d %d", &x, &y) != 2)     /* &-Operator konstruiert Zeiger */
        return 1;                        /* Exit-Status ungleich 0 => Fehler */

    printf("ggT von %d und %d ist %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 Call-By-Value-Semantik von Funktionsaufrufen müssen wir die Eingaben x und y nicht mehr kopieren (wie in vorigem Beispiel).

2.2 Aufbau eines C-Programms

Ein C-Programm ist eine *Folge von Definitionen*:

```
C_Program  ::= { Definition }
Definition ::=   FunctionDefinition
               | DataDefinition
               | TypeDefinition
               | ExternDeclaration
```

Zu den Vereinbarungen gehören Funktionsdefinitionen, Typ-Vereinbarungen und Variablenvereinbarungen.

2.2.1 Anweisungsblöcke

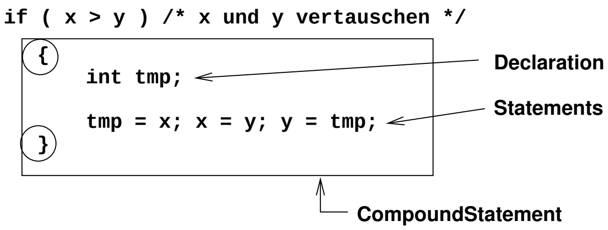


Abbildung 2.1: Anweisungsblock (compound statement)

Die *Blockstruktur* ist in C anders als in Modula-2 oder Oberon; siehe Abb. 2.1. Statt *begin* und *end* steht in C ein Paar von geschweiften Klammern (*{}*). Außerdem ist in C ein *Anweisungsblock* (*compound statement*) nur eine spezielle Anweisung. Somit kann überall dort, wo eine Anweisung stehen kann, auch ein Anweisungsblock verwendet werden:

```
Statement          ::=   [ Expression ] ";"
                       | CompoundStatement
                       | ...
CompoundStatement ::= "{" { Declaration } { Statement } "}"
```

Zu Beginn eines jeden *Anweisungsblocks* dürfen Variablen-Vereinbarungen getroffen werden. Anweisungsblöcke erlauben es, *mehrere Anweisungen zusammenzufassen* und *Sichtbarkeits-/Lebensdauerbereiche zu definieren*; siehe Abb. 2.1.

Anmerkungen zu Abb. 2.1:

- Die *Gültigkeit von tmp* erstreckt sich auf den umrandeten Anweisungsblock.
- Mit „*int tmp;*“ wird eine (lokale) Variable mit Datentyp *int* deklariert. *int* ist ein Schlüsselwort und steht für *Integer* (d. h. ganze Zahlen).
- *Zuweisungen* werden in C mit = und nicht mit := wie in Modula-2 und Oberon notiert. Daher ist == der *Vergleichsoperator* in C.

2.2.2 Kommentare

Kommentare beginnen mit „/*“, enden mit „*/“, und können (im ANSI-Standard) nicht geschachtelt werden.

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 *Underscore* („_“) gezählt.

2.2.4 Schlüsselworte

Die folgende Tabelle enthält die wichtigsten *Schlüsselworte* von C:

break	case	char	const	continue
default	do	double	else	enum
extern	float	for	goto	if
int	long	return	short	signed
sizeof	static	struct	switch	typedef
union	unsigned	void	while	

Einige Schlüsselworte wie *auto* oder *register* sind heute i. A. ohne Bedeutung.

2.2.5 Whitespaces

Whitespace ist eine Sammelbezeichnung für Leerzeichen, Tabulator und Zeilenumbruch. Diese Zeichen dienen als *Trenner*, sie werden ansonsten aber ignoriert.

Kapitel 3

Der Preprozessor – vorab

3.1 Motivation

Eine Technik zur Realisierung von *Unterprogrammen* ist der *offene Einbau* via *Makro* (*Textersatz*). Bei Aufruf eines „normalen“ Unterprogramms wird der zugehörige Anweisungsteil im compilierten Programm angesprungen. Beim „Aufruf“ durch ein Makro wird der Text des Unterprogramms an der Stelle des „Aufrufs“ einfach *vor dem Compilieren* des Programms eingefügt. Der „Aufruf“ des Unterprogramms erfolgt durch Nennung des Makronamens (ggf. mit Parametern).

Vorteil: Die Abarbeitung des Aufrufs ist *schneller* (kein Anspringen, keine Stack-Verarbeitung).

Nachteil: Der *Programmtext* wird mit jedem Aufruf *größer*; Rekursion ist nicht möglich.

Das Programm, das diesen Textersatz durchführt, heißt **Preprozessor** (zu deutsch eigentlich Präprozessor – diese Form hat sich allerdings nicht durchgesetzt) und ist dem eigentlichen Compiler vorgeschaltet (daher auch der Name *Prä*prozessor!). Bei Textverarbeitungsprogrammen wird diese Technik als Makro-Technik sehr häufig verwendet (*Makro-Prozessor*)

Viele Makros haben den Charakter von *Direktiven* in dem Sinne, dass sie Definitionen oder Anweisungen darstellen – es sind trotz allem auch Makros mit Ersatztext (s.u.).

3.2 cpp – der C-Preprozessor

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

Die von ihm erkannten *Direktiven* müssen als erstes (Nicht-Whitespace-)Zeichen in der Zeile mit einem # beginnen, dem meist unmittelbar danach der Name einer Direktive mit evtl. Argumenten folgt.

3.3 define-Direktive

define ist eine Direktive, die eine Makrodefinition erledigt (genauer: ein Makro, das eine implizit eine Makrodefinition erledigt und dessen Ersatztext “Nichts” ist)..

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

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

int main() {
    int x = MAX; /* Initialisieren */
    int y = MAX1; /* Initialisieren ? */
    int z = MAX2; /* Initialisieren ? */

    return 0;
}
```

Programm 3.1 zeigt ein Beispiel zur Verwendung der *define*-Direktive. Der *cpp* liefert zu diesem Programm folgende Ausgabe:

```
thales$ 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;;

    return 0;
}
thales$
```

Will man die Kommentare bei der Ausgabe erhalten, dann kann man das durch Zuschalten der Option *-C* erreichen: *gcc -E -C makros.c*.

Die Option *-P* ist nützlich, will man den nach der Makro-Verarbeitung entstandenen Text nicht dem eigentlichen C-Compiler "weiterfüttern":

```
thales$ gcc -E -P makros.c
int main() {
    int x = 10;
    int y = = 10;
    int z = 10;;

    return 0;
}
thales$
```

Da Direktiven zu einem *inline-Text* führen, kann man damit auch (nicht-rekursive) Prozeduren realisieren, bei denen „kein Ansprung“ erfolgt, die somit i. A. schneller sind. Dazu können Makros auch parametrisiert werden (dazu und zu den weiteren Direktiven später mehr).

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 des Makros eingesetzt. Das Makro existiert also nach dem Kompilieren gar nicht mehr und hat somit auch keinen Ort im Speicher. Variablen und Makros sind also grundverschieden!

3.4 include-Direktive

include ist eine Direktive, die den Inhalt der angegebenen Datei einfügt (genauer: *include* ist ein Makro, dessen Ersatztext der Inhalt der angegebenen Datei ist).

Damit werden i. A. Vereinbarungen oder andere Direktiven „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 Katalog */usr/include* – diese sind die Schnittstellen der C-Bibliothek und entsprechen im Prinzip den *DEFINITION MODULE*s von Modula-2 (bzw. Oberon). 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"          /* einbinden von defs.h im selben Verzeichnis */

#define MAX 10

int main() {
    int x = MAX;

    return 0;
}
```

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

```
int y = 3;
```

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

```
thales$ 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 = 10;

    return 0;
}
thales$
```

Kapitel 4

Ein- und Ausgabe

4.1 stdin, stdout und stderr

Standardmäßig gibt es drei Kanäle für die Ein-/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 man stdin mittels <, stdout mittels > und stderr mittels 2>*err* umlenken:

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

```c
#include <stdio.h>

int main() {
        /* puts fuegt ein Newline an */
    puts("Ich komme nach stdout ...");
        /* fputs fuegt KEIN Newline an */
    fputs("Ich komme nach stderr ...\n", stderr);

    return 0;
}
```

```
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 einen String auf die Standardausgabe aus.

*int puts(const char *s);*

Der *Rückgabewert* von *puts()* ist die Anzahl der geschriebenen Zeichen. Nach der Ausgabe des Strings gibt *puts()* noch einen zusätzlichen Zeilenumbruch (newline) 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 ausgegeben Zeichen zurück.
- Ein *Formatstring* (*format*) ist eine in Doppelapostrophen eingeschlossene Zeichenfolge (konstanter String) oder Variable (vom Typ „Zeichenkette“). Er besteht aus Text und eingestreuten Formatierungsangaben.
  Bsp.: *printf(„Ich bin %d Jahre alt.“, 3);*
- *Formatierungsangaben* führen zur Verarbeitung von (null oder mehr) Argumenten (*args*). Die nächste fängt da (bei den Argumenten) an, wo die letzte aufgehört hat. Wenn die Anzahl der Argumente nicht ausreicht, so ist das Resultat nicht definiert.
- *Formatierungsangaben* beginnen mit einem *%* evtl. gefolgt *Flags*, einer *minimalen Breite*, einer *Genauigkeit* und einem Zeichen zur *Konvertierung*:

  ```
  ConvSpec  ::= "%" {Flags} [MinWidth] [ "." Precision ] ConvChar
  Flags     ::= "-" | "+" | " " | ...
  MinWidth  ::= Digit { Digit } | "*"
  Precision ::= Digit { Digit } | "*"
  ConvChar  ::= "d" | "i" | "o" | "u" | "x" | "X" | ...
  ```

  Bsp.: *%-3s*, *%d*, *%3.2f*
- *** als minimale Breite oder Genauigkeit bedeutet, dass das nächste Argument als Integer behandelt wird und als minimale Breite bzw. Genauigkeit für diese Formatierungsangabe verwendet wird.

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

Zeichen für die Konvertierung	Effekt
`d,i,o,u,x,X`	Integer: `d` und `i` für Dezimaldarstellung; `u` für unsigned; `x`, `X` für Hexadezimal, wobei `x` die Kleinbuchstaben „a“–„f“, `X` die Großbuchstaben „A“–„F“ benutzt; unmittelbar davor kann ein `h` (für short int) oder `l` (für long int) stehen
`f`	Reelle Zahlen in Dezimalform ([-]mmm.nnnnnnn): Anzahl der Nachkommastellen durch Genauigkeitsangabe; Default: 6
`e,E`	Reelle Zahlen in Exponentialdarstellung
`c`	Zeichen: das zugehörige Datenargument wird als int übergeben, das letzte (= niederwertigste) Byte wird ausgegeben
`s`	Strings: das Argument muß ein Character-Array oder konstanter String mit jeweils abschließendem Null-Byte sein; Genauigkeitsangabe wird als maximal auszugebende Zeichenzahl interpretiert Bsp.: printf(“dieser String ist %s Zeichen lang“, “fuenfunddreissig“);
`%`	die Folge %% gibt ein % aus

Für die *Flags* gibt u. a. folgende Wahlmöglichkeiten:

Flags	Bedeutung
-	linksbündige Ausgabe
+	auch pos. Vorzeichen wird ausgegeben
Leerzeichen	statt pos. Vorzeichen ein Leerzeichen ausgeben

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

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

```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 */

    return 0;
}
```

```
thales$ gcc -Wall 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 Infos zu *printf()* und *puts()* findet man wie immer in den *Manpages* (*man printf* bzw. *man puts*).

4.3 Ausgabe nach stderr

Die Funktion *fputs()* gibt einen String in die angegebene Dateiverbindung (stream) – in unserem Fall *stderr* – aus:

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

Der einzige Unterschied zu *puts()* – mal abgesehen vom zusätzlichen Argument – ist, dass von *fputs()* nur der übergebene String 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 (file pointer). 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, ...);*

- Format-String besteht aus Zeichen für die Konvertierung und weiteren Zeichen.
- Konvertierung:
  % gefolgt von optionalen Zeichen zur Modifikation, gefolgt von Konvertierungszeichen
- Andere Zeichen (außer Konvertierungszeichen und Leeraum) müssen mit Zeichen im Eingabestrom übereinstimmen. Leerzeichen, Tabs (\t) und Newlines (\n), also Whitespaces, veranlassen *scanf()* zum nächsten Nicht-Whitespace weiterzugehen.
- Beachte : C kennt nur Wertparameterübergabe. Aus diesem Grund muß speziell bei *scanf()* der *Adressoperator* vor den aktuellen Parameter gestellt werden (Ausnahme: Arrays – dazu später mehr). Damit wird ein Zeiger auf die Variable übergeben und *scanf()* greift über diesen Zeiger auf das „Orginal“ durch!

Bsp.: *scanf(„%d“, &n);* liest in die Integer-Variable *n* einen Wert von der *Standardeingabe* (stdin) ein.

scanf() hat einen ganzzahligen *return*-Wert, der die Anzahl der tatsächlich erfolgten Variablenzuweisungen wiedergibt.

Konvertierungs-Zeichen	Wirkung
d	Dezimal-Konstante
o	Oktal-Konstante
x,X	Hexadezimal-Konstante
f	Fließkommazahl
s	String; Zeichenfolge bis zum nächsten Leerraum (Blank, ’\t’, ’\n’); Null-Byte (’\0’) wird angefügt
c	Nächstes Zeichen; um das nächste Nicht-Whitespace-Zeichen zu lesen: %1c
%	Liest %-Zeichen ohne Zuweisung

Maximale Feldlänge: Dezimal-Zahl

Flag: „“* liest, unterdrückt aber 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("-------------------------------------");

    return 0;
}
```

```
thales$ gcc -Wall 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 einen String ein:

*char *gets(char *s);*

Das evtl. abschließende Newline wird von *gets()* entfernt (im Gegensatz zu *fgets()*). Außerdem überprüft *gets()* nicht, ob genügend Speicherplatz vorhanden ist. Es kann also zu einem *Puffer-Überlauf* (buffer overflow) kommen!

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*)

```c
#include <stdio.h>

#define 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);

    return 0;
}
```

```
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-/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 Manpages (*man getc*, ...).

Kapitel 5

Kontrollstrukturen

5.1 Übersicht

```
Statement ::=   Prefix Statement
              | [ Expression ] ";"
              | "break" ";"
              | "continue" ";"
              | "return" [ Expression ] ";"
              | "goto" Identifier ";"
              | CompoundStatement
              | "if" "(" Expression ")" Statement
              | "if" "(" Expression ")" Statement "else" Statement
              | "do" Statement "while" "(" Expression ")" ";"
Prefix    ::=   Identifier ":"
              | "while" "(" Expression ")"
              | "for" "(" [ Expression ] ";" [ Expression ] ";" [ Expression ] ")"
              | "switch" "(" Expression ")"
              | "case" Constant ":"
              | "default" ":"
```

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

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

C kennt keinen Datentyp *BOOLEAN*. Daher wird statt dessen der Typ *int* dafür verwendet. Der Wert 0 steht dabei für *FALSE* und alles andere, also ungleich 0, steht für *TRUE*:

BOOLEAN	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, 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 geschachtelte *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;
}
```

Tipp: Für alle Kontrollstrukturen, die mehr als einen Ausdruck im Anweisungsteil enthalten verwendet man am besten einen *Anweisungsblock* (also Schachtelung in „{" und „}"). Das Ergebnis ist weniger fehleranfällig und deutlich leichter zu lesen! Am besten ist es, wenn man immer die Anweisung(en) in einen Anweisungsblock schachtelt, damit es erst zu keinen Fehlern kommt. Es gibt nämlich die Tendenz, dass im Nachhinein immer eine Anweisung dazu kommt. :-)

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 gar nicht durchlaufen wird.)

Beispiel: Zeichenweises Lesen von *stdio* und Zählen der Leerzeichen (Die Bibliotheks-Funktion *getchar()* aus *stdio.h* liest ein Zeichen von *stdio* und liefert es als *int*-Wert. Das Ende des Eingabestroms ist über das Makro *EOF* (in *stdio.h*) als -1 definiert.)

Programm 5.3: *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);

    return 0;
}
```

5.4 do-while-Schleife

Die **do−while**-Schleife führt die Anweisung bzw. den Anweisungsblock solange aus, wie die Bedingung *TRUE* ist, d. h. ihr der Wert des Bedingungsausdrucks ungleich Null ist. Die Überprüfung der Bedingung findet jeweils *nach* einem Schleifendurchlauf statt. (Es gibt also immer mindestens einen Schleifendurchlauf.) Dies entspricht im Prinzip der *REPEAT*-Schleife in Oberon oder Modula-2 (nicht ganz allerdings!).

Programm 5.4: *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);
    return 0;
}
```

Programm 5.5: *do-while*: Whitespaces ignorieren (*ignore.c*)

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

void skip_spaces() {
    int ch;
    do
        ch = getchar();
    while (isspace(ch));

        /* 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)
        if (ch == ' ') {
            putchar(ch);
            skip_spaces();
```

```
        }
        else
            putchar(ch);

    return 0;
}
```

Anmerkungen:

1. Die Funktion *ungetc()* stellt ein Zeichen zurück in die Eingabe. Dies ist jedoch nur für ein Zeichen garantiert – bis dieses gelesen ist (dann für das nächste usw.).
2. Dass hier für Zeichen Variable vom Datentyp *int* verwendet werden liegt daran, dass in C Zeichen (Datentyp *char*) 1-Byte-Integer sind; damit kann auch mit dem Wert -1 (EOF) als Integer verglichen werden!

5.5 for-Schleife

Die *for*-Anweisung hat folgende Grundstruktur:

```
for (/* initialisierung */; /* bedingung */; /* inkrement */)
    /* 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);
```

Äquivalent zu einer *for*-Schleife kann man auch eine *while*-Schleife verwenden:

```
/* 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++;
}
```

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

Dient dazu, vorzeitig an den Schleifenanfang zu springen, wobei die restlichen Anweisungen im Anweisungsteil übersprungen werden.

Programm 5.6: Verwendung von continue (*continue.c*)

```c
#include <stdio.h>

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

Programm 5.7: Zusammenhang zwischen for, while und continue (*continue1.c*)

```c
#include <stdio.h>

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

5.7 break-Anweisung

Dient zum vorzeitigen Verlassen einer Schleife oder zum „Verlassen“ eines Falles in der Fallunterscheidung (*case* in einer *switch*-Anweisung).

Programm 5.8: Verwendung von break (*break.c*)

```c
#include <stdio.h>

int main() {
    int i;
    for (i = 1; i <= 20; i++) {
        if (i % 4 == 0)
```

```
            break;
        printf ("%d\n", i);
    }
    return 0;
}
```

5.8 switch-Anweisung

Die *switch*-Anweisung kann zu einer Fallunterscheidung verwendet werden. Programm 5.9 zeigt eine erste Verwendung der *switch*-Anweisung. Programm 5.10 ist eine Anwendung mit einem sogenannten *fall through*.

Programm 5.9: 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");
    }

    return 0;
}
```

Zur Semantik und Verwendung:

- Nach der Auswertung des „*switch*-Ausdrucks“ wird bei der Anweisung des passenden *case*-Falls fortgefahren.
- Auch die folgenden „Fälle“ werden abgearbeitet, falls dies nicht durch ein explizites *break* verhindert wird.
- Der Ausdruck nach *switch* muß ein *Integer-Typ* sein (d. h. *char*, *short*, *int*, *long*, *enum*) — nicht *float*, *double*, *long double*. (Nach Kernighan&Ritchie war nur *int* erlaubt!)
- Der Ausdruck nach *case* muß ein *konstanter Ausdruck mit Integer-Wert* sein.

- Die *case*-Labels müssen verschieden sein.
- Trifft keiner der Fälle zu, geht es bei *default* weiter (muß zwar nicht, sollte aber grundsätzlich der letzte Fall sein).
- Der *default*-Fall sollte immer vorhanden sein.
- Auch der *default*-Fall sollte mit **break** verlassen werden (⇝ spätere Änderungen).

Programm 5.10: Beispiel für die *switch*-Anweisung mit „fall through" (*switch1.c*)

```
#include <stdio.h>

int ispunc(char arg) {
    switch (arg) {
        case '.':       /* mehrmals fall through ... */
        case ',':       /* ... da break fehlt ! */
        case ':':
        case ';':
        case '!':
            return 1;   /* fuer all die obigen Faelle ! */
        default:
            return 0;
    }
}

int main() {
    char ch;

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

    if (scanf("%c", &ch) > 0) {
        if (ispunc(ch))
            puts("Interpunktion");
        else
            puts("keine Interpunktion");
        return 0;
    }
    else {
        puts("\nNichts eingegeben!");
        return 1;
    }
}
```

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

Bei Operanden unterscheidet man zwischen zwei Arten:

- *Links-Werte bzw. L-Werte* sind solche Operanden, die konkret ein Objekt im Speicher (also eine Speicherfläche) bezeichnen und deshalb auch veränderbar sind. Beispiele sind der Name einer Variablen (x), ein dereferenzierter Zeiger ($*p$) und ein indiziertes Array ($a[3]$).
- *Rechts-Werte bzw. R-Werte* sind Werte eines Audrucks. Beispielsweise ist der R-Wert der Variablen a gleich 3, wenn a diesen Wert besitzt.

Die Bezeichnungen Links- bzw. Rechts-Wert entstanden im Hinblick auf eine Zuweisung. Da dabei die linke Seite verändert werden muss, muss links ein L-Wert stehen.

Beachte: Jeder L-Wert hat natürlich auch einen R-Wert. Beispiele für Ausdrücke, die keine L-Werte sind, sind Konstanten (3) oder Ergebnisse von Berechnungen (x+y).

6.1.2 Operanden im Einzelnen

```
Operand      ::=   Identifier
                 | Constant
                 | String
                 | "(" Expression ")"
                 | Operand "(" [ ArgumentList ] ")"
                 | Operand "[" Expression "]"
                 | Operand "." Identifier
                 | Operand "->" Identifier
ArgumentList ::= Assignment { "," Assignment }
```

- **Identifier**
  - L-Wert, wenn Name einer Variablen; kein L-Wert, wenn Name einer Konstanten, z. B. in einer Aufzählung (**enum**) definiert
  - Typ folgt aus der Namensvereinbarung
  - bezeichnet er ein Array, so gilt er als Zeiger auf das erste Element (konstanter Zeigerwert, kein L-Wert)
  - bezeichnet er eine Strukturvariable („record/struct“), so ist er ein L-Wert
  - bezeichnet er eine Funktion (nicht beim Aufruf), so ist er ein Zeiger auf diese Funktion (konstanter Zeigerwert)
- **Constant**
  - ihr Typ ergibt sich aus der Definition (kein L-Wert)
  - Zeichen (Characters) sind vom Typ *int*
- **String**
  - repräsentiert einen **konstanten** Zeigerwert auf das erste Zeichen (kein L-Wert)
- **ArgumentList**
  - kann fehlen, runde Klammern aber dennoch notwendig
  - Wertparameter
  - Reihenfolge der Bewertung *nicht* definiert

6.2 Operatoren

6.2.1 Übersicht

Die folgende Tabelle gibt eine Übersicht über die Operatoren mit Vorrang/Priorität (höchster oben) und Bindung/Assoziativität:

Priorität	Klasse	Operatoren	Assoziativität
1	primäre	() [] . ->	links
2	unäre	! ~ - + ++ -- & * (type) sizeof	rechts
3	multiplikative	* / %	links
4	additive	+ -	links
5	Bit-Verschiebungen	<< >>	links
6	Vergleiche	< <= > >=	links
7	Äquivalenztests	== !=	links
8	bitweises UND	&	links
9	bitweises (exkl.) ODER	^	links
10	bitweises (inkl.) ODER	|	links
11	logisches UND	&&	links
12	logisches ODER	||	links
13	Auswahl	?:	rechts
14	Zuweisung	= += -= *= /= %= >>= <<= &= ^= |=	rechts
15	Komma	,	links

6.2.2 Unäre Operatoren

```
Unary ::=   Operand
          | Operand ("++" | "--")
          | ("*" | "&" | "-" | "!" | "~") Unary
          | ("++" | "--" | "(" Type ")" | "sizeof") Unary
          | "sizeof" "(" Type ")"
```

- *Inkrement ++ bzw. Dekrement –:* Argument muß L-Wert sein; a++ bzw. a−− liefern als Ergebnis den Wert *vor* der Inkrementierung bzw. Dekrementierung; ++a bzw. −−a liefern als Ergebnis den Wert *nach* der Inkrementierung bzw. Dekrementierung
- *Dereferenzierung* mit dem Operator *
- *Adressoperator &:* Argument muss ein L-Wert sein
- *Negation !:* Ergebnis ist 1, falls Operand den Wert 0 hatte, 0 sonst
- *sizeof* liefert die Zahl der Bytes, die ein Datentyp T (**sizeof**(*T*)) oder ein Ausdruck x (**sizeof** *x* oder **sizeof**(*x*)) belegt; liefert bei einem Array-Namen die Größe des gesamten Arrays
- *Bitweises Komplement* mit ˜ (Einer-Komplement)

Folgendes Beispiel illustriert die Verwendung unärer Operatoren:

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

```c
#include <stdio.h>

int main() {
    int i = 3;   /* vorinitialisierte Integervar. */
    int *p;      /* Zeiger auf eine Integervar. */
    int a[50];   /* Integer-Array */

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

    return 0;
}
```

```
thales$ gcc -Wall unaer.c
thales$ a.out
i=3, p=ffbef62c (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
thales$
```

6.2.3 Binäre Operatoren

```
Binary ::=  Unary
          | Binary "||" Binary | Binary "&&" Binary
          | Binary "|"  Binary | Binary "^"  Binary
          | Binary "&"  Binary | Binary "==" Binary
          | Binary "!=" Binary | Binary "<"  Binary
          | Binary "<=" Binary | Binary ">"  Binary
          | Binary ">=" Binary | Binary "<<" Binary
          | Binary ">>" Binary | Binary "+"  Binary
          | Binary "-"  Binary | Binary "*"  Binary
          | Binary "/"  Binary | Binary "%"  Binary
```

- *Logisches UND (&&) bzw. ODER (||):* Operanden müssen mit 0 vergleichbar sein; Resultat ist 0 oder 1; 0 entspricht *false*; 1 (ungleich 0) entspricht *true*
- *Bitweises UND (&), ODER (|) und XOR (^):* Operanden müssen Integer sein; Ergebnis ist Integer
- *Vergleichsoperatoren:* Resultat ist (boolesche) Integer; Wert 1 für *true* und 0 sonst
- *Modulo-Operator (%)*
- *Bit-Verschiebungen* können mit den Operatoren << und >> bei Integern durchgeführt werden.

Folgendes Beispiel illustriert die Verwendung binärer Operatoren:

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

```c
#include <stdio.h>

   /* gibt zahl in Binaerdarstellung mit
    * der min. Breite min_breite aus (rekursiv !) */
void print_binaer(int zahl, int min_breite) {
   if (min_breite > 0) {
         /* rekursiv die vorderen Bits der Zahl
          * ausgeben (mit minimaler Breite min_breite-1
          * (Effizienz :-() */
      print_binaer(zahl >> 1, min_breite - 1);
```

```
            /* danach noch das letzte Bit ausgeben */
            printf("%d", zahl & 1);
      }
}

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

        /* Division und Modulo (= Divisionsrest ) */
    printf("i=%d, j=%d, i/j=%d, i%%j=%d\n", i, j, i / j, i % j);

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

    printf("i="); print_binaer(i, 5);
    puts("");
    printf("j="); print_binaer(j, 5);
    puts("");

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

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

        /* Bit-Verschiebungen bei negativen Zahlen */
    i = -1;
    printf("i = -1 = "); print_binaer(i, 32); puts("");
    printf("-1>>1  = "); print_binaer(i >> 1, 32); puts("");
    printf("-1<<1  = "); print_binaer(i << 1, 32); puts("");

    return 0;
}
```

```
thales$ gcc -Wall binaer.c
thales$ a.out
i=17, j=3, i/j=5, i%j=2
---------------------------------------
i=10001
j=00011
i&j=00001, i|j=10011, i^j=10010
i>>1=01000, i<<1=00010
i = -1 = 11111111111111111111111111111111
-1>>1  = 11111111111111111111111111111111
-1<<1  = 11111111111111111111111111111110
thales$
```

Anmerkung: Bei negativen Zahlen ist immer das höchstwertigste Bit 1. Bei den Bit-Verschiebungen nach rechts wird das Vorzeichenbit (eben das höchstwertigste) immer wieder auf 1 gesetzt. Daher auch das verwunderliche Ergebnis, dass sich -1 bei einer Verschiebung um ein Bit nach rechts nicht ändert.

Folgendes Beispiel zeigt, dass der *Operator %* für negative Argumente nicht die Modulo-Funktion berechnet:

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

```c
#include <stdio.h>

/* Die Modulo-Operation ist nicht identisch mit
 * dem Operator %!
 */
int mod(int a, int b) {
    if (a < 0)
        return a % b + b;
    else
        return a % b;
}

int main() {
    printf("%d\n", 33 % 5);
    printf("%d\n", -33 % 5);
    printf("%d\n", mod(33, 5));
    printf("%d\n", mod(-33, 5));
    return 0;
}
```

```
thales$ gcc -Wall modulo.c
thales$ a.out
3
-3
3
2
thales$
```

6.2.4 Auswahl-Operator

```
Selection ::=  Binary
             | Binary "?" Selection ":" Selection
```

Eine *Auswahl* (*selection*) besteht aus einer Bedingung, der zwei Ausdrücke folgen. Die Bedingung wird 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;
```

6.2.5 Komma-Operator

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);
    }

    return 0;
}
```

```
thales$ gcc -Wall komma.c
thales$ 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
thales$
```

6.2.6 Zuweisungen

```
Assignment ::=   Unary "="   Assignment
               | Unary "|="  Assignment
               | Unary "^="  Assignment
               | Unary "&="  Assignment
               | Unary "<<=" Assignment
               | Unary ">>=" Assignment
               | Unary "+="  Assignment
               | Unary "-="  Assignment
               | Unary "*="  Assignment
               | Unary "/="  Assignment
               | Unary "%="  Assignment
```

= ist der Zuweisungsoperator in C (⇝ *Nebeneffekt*). $tmp = x$ ist genau betrachtet ein Ausdruck und hat somit einen Wert, nämlich den zugewiesenen Wert, also den von x.

Damit ist eine Zuweisung der Form $x = y = 1$ möglich, die Bewertung des Ausdrucks erfolgt von rechts nach links: zuerst wird $y = 1$ bewertet, d. h. y bekommt den Wert 1; der Wert dieses Teil-Ausdrucks ist der zugewiesene Wert (also 1); dieser Wert wird dann x zugewiesen; der Wert des gesamten Ausdrucks ist wiederum der zugewiesene Wert.

$a = a$ **op** b kann in C effizienter (und auch kürzer) als a **op=** b geschrieben werden. Dann muss a nur einmal bewertet werden.

Programm 6.5: Zuweisungen (*zuweisung.c*)

```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
         * VORSICHT wg. der Reihenfolge der Parameterbewertung! */
    printf("a=%d, (a+=2)=%d\n", a, a += 2);

    return 0;
}
```

```
thales$ gcc -Wall zuweisung.c
thales$ a.out
a=13, b=13
a=15, (a+=2)=15
thales$
```

Tipp: Verlassen Sie sich nicht auch die Reihenfolge der Parameterbewertung. Verwenden Sie (außer max. einem Inkrement oder Dekrement) keine Anweisungen/Operatoren mit Nebeneffekten in anderen Anweisungen.

Kapitel 7

Datentypen

7.1 Überblick

Datentypen legen fest:

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

Abbildung 7.1 gibt einen Überblick über die verschiedenen Datentypen in C.

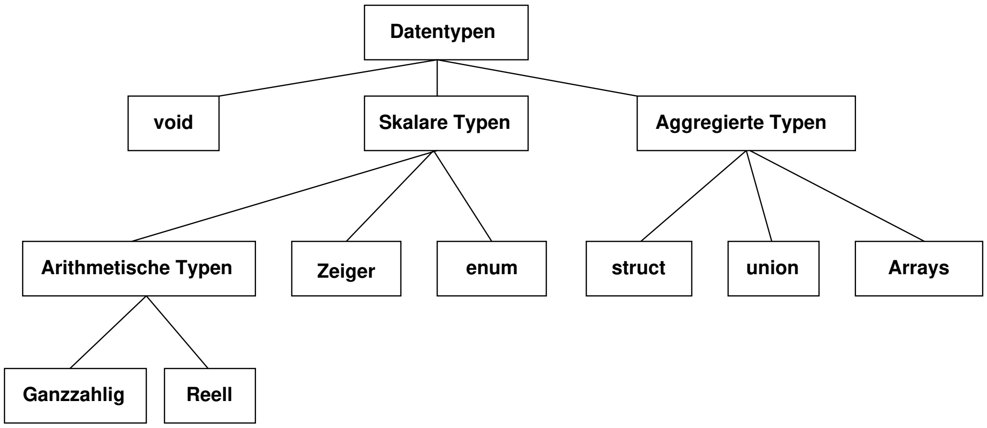


Abbildung 7.1: Datentypen – Eine Übersicht

7.2 Skalare Datentypen

Skalare Variablen lassen sich einteilen in

- *ganzzahlige Variablen* (*Integer-Variablen*):
  Schlüsselwort **int** mit den Modifikatoren **short**, **long**, **long long** (Konvertierung bei printf() mit **%lld**), **signed** und **unsigned**
- *Zeichen-Variablen* (*characters*):
  Schlüsselwort **char** mit den Modifikatoren **signed** und **unsigned**
- *Gleitkomma-Variablen* (*floats*):
  Schlüsselworte **float** und **double** mit dem Modifikator **long** (Konvertierung bei printf() z.B. mit **%Le**
- *Aufzählungs-Variablen* (*enums*):
  Schlüsselwort **enum**
- *Zeiger* (*pointers*)

In der folgenden Tabelle sind die Wertebereiche der skalaren Typen angegeben:

Typ	Größe (Bytes)	Wertebereich
int	4 (oft nur 2)	$-2^{31} \ldots 2^{31} - 1$
short int	2	$-2^{15} \ldots 2^{15} - 1$
long int	4	$-2^{31} \ldots 2^{31} - 1$
long long	8	$-2^{63} \ldots 2^{63} - 1$
unsigned short int	2	$0 \ldots 2^{16} - 1$
unsigned long int	4	$0 \ldots 2^{32} - 1$
signed char	1	$-2^{7} \ldots 2^{7} - 1$
unsigned char	1	$0 \ldots 2^{8} - 1$
float	4	$1.175494351E - 38 \ldots 3.402823466E + 38$
double	8	$2.2250738585072014E - 308 \ldots 1.7976931348623157E + 308$
long double	16	$3.362103143\ldots E - 4932 \ldots 1.18973149\ldots E + 4932$

7.2.1 Konstanten

Durch Voranstellen des Schlüsselwortes **const** kann man aus einem skalaren Typ einen konstanten skalaren Typ machen. Dies wird im folgenden Beispiel demonstriert:

Programm 7.1: 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);

  return 0;
}
```

```
thales$ gcc -Wall const.c
const.c: In function ‘main’:
const.c:6: warning: increment of read-only variable ‘i’
thales$ a.out
i=2
thales$
```

Anmerkung: C interpretiert das Schlüsselwort **const** nur als einen *Hinweis*. Die Variable lässt sich jedoch *trotzdem verändern*! Dabei kommt es lediglich zu einer Warnung des gcc.

7.2.2 Zeichen, Character (char)

Der Datentyp **char** ist im Prinzip ein 1-Byte-Integer-Typ (wie man auch schon am Wertebereich von **char** sehen konnte). Einzelne Zeichen, d. h. Zeichen-Konstanten, werden in einfache Hochkommas eingeschlossen. Bsp.: 'a', ':', ...

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		Beep
\0	NUL	Null-Byte
\ddd		ASCII-Code (oktal)

Zeichen-Variablen können auch wie Integer-Variablen verwendet werden, wie folgendes Beispiel zeigt:

Programm 7.2: Arbeiten mit Zeichen-Variablen (*char.c*)

```
#include <stdio.h>

int main() {
  char ch;
  int i;

    /* Ersatzdarstellung fuer den Zeilentrenner */
  ch = '\n'; printf ("ch='%c', ORD(ch)=%d\n", ch, ch);
```

```
    /* mit chars kann man wie mit Integern arbeiten
     * => kein CHR(...) und ORD(...) notwendig wie in Modula/Oberon! */
  ch = 'A'; printf("ch='%c', ORD(ch)=%d\n", ch, ch);
  ch = 65; printf("ch='%c', ORD(ch)=%d\n", ch, ch);
  ch = 'A'+1; printf("ch='%c', i=ORD(ch)=%d\n", ch, i = ch);

  return 0;
}
```

```
thales$ gcc -Wall char.c
thales$ a.out
ch='
', ORD(ch)=10
ch='A', ORD(ch)=65
ch='A', ORD(ch)=65
ch='B', i=ORD(ch)=66
thales$
```

7.2.3 Typ-Konvertierungen

Arithmetische Typen können in Ausdrücken (mit wenigen Ausnahmen) gemischt werden. Dabei finden oft teils automatische Konvertierungen (*implizite Konvertierungen*) und teils manuelle Konvertierungen (*explizite Konvertierungen*) mit dem (*type*)-Operator statt; siehe Abbildung 7.2.

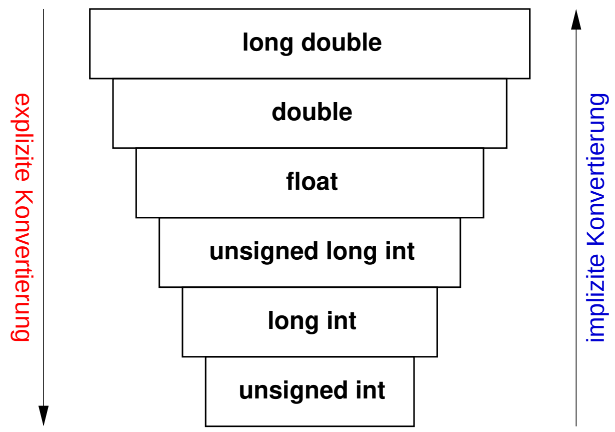


Abbildung 7.2: Typ-Hierarchie

- *Zuweisungs-Konvertierung:*
  Der Wert auf der rechten Seite wird konvertiert in den Typ auf der linken, sofern dies möglich ist.

- *Ganzzahlige Erweiterung:*

  – **char** und **short int** in einem Ausdruck werden in **int** konvertiert.

- **unsigned char** und **unsigned int** werden in **int** konvertiert (falls nicht zu groß, sonst **unsigned int**).

- In *arithmetischen Ausdrücken* werden die Operanden jeder Operation gemäß der Typ-Hierarchie in Abbildung 7.2 in den größeren der beiden Typen konvertiert.
  Bsp.: Enthält ein Operandenpaar eine **long double**, so wird der andere ebenfalls in **long double** konvertiert, usw.

Programm 7.3: Typ-Konvertierung (*convert.c*)

```c
#include <stdio.h>

int main() {
    char ch = 5;
    unsigned short s;
    int i = 71124;

        /* Nur die hinteren (niederwertigsten) 4 Bytes
         * von i werden s zugewiesen, da nicht mehr Platz ist!
         * (Dies entspricht dem Rest der Division i / 2^16,
         *  wobei 2^16 = 65536.) */
    s = i;
    printf("s=%08xh, i=%08xh\n", s, i);

        /* ch und s werden nach int konvertiert! */
    i = ch + s + 8;
    printf("i=%d, ch=%d, s=%d\n", i, ch, s);

        /* 882 passt nicht in eine char
         * => nur die niederwertigsten 8 Bit werden zugewiesen! */
    ch = 882;
    printf("ch='%c', ch=%02xh, 882=%02xh\n", ch, ch, 882);

    return 0;
}
```

```
thales$ gcc -Wall convert.c
convert.c: In function `main':
convert.c:21: warning: overflow in implicit constant conversion
thales$ a.out
s=000015d4h, i=000115d4h
i=5601, ch=5, s=5588
ch='r', ch=72h, 882=372h
thales$
```

7.2.4 Gleitkommazahlen (float, double)

Beim Umgang mit Gleitkomma-Zahlen ist hinsichtlich Genauigkeit etwas Vorsicht angebracht – dies zeigt das folgende Programm.

Programm 7.4: Genauigkeit (*float.c*)

```
/* RICHTIGES ERGEBNIS: +1 */
#include <stdio.h>

int main() {
    int x = 10864, y = 18817, z;
    float x1 = x, y1 = y, z1;
    double x2 = x, y2 = y, z2, z2a;
    double x3 = x, y3 = y, z3, u3, v3, w3;

    /* int */
    z  = 9 * x*x*x*x - y*y*y*y + 2 * y*y;
    printf("int: %d\n", z);

    /* float */
    z1 = 9 * x1*x1*x1*x1 - y1*y1*y1*y1 + 2 * y1*y1;
    printf("float: %f\n",z1);

    /* double (Formel umgestellt) */
    z2 = 9 * x2*x2*x2*x2 + 2 * y2*y2;
    z2a = z2 - y2*y2*y2*y2;
    printf("double (Formel umgestellt): %f\n",z2a);

    /* double ( schrittweise ) */
    u3 = 9 * x3*x3*x3*x3;
    v3 = 2 * y3*y3;
    w3 = u3 + v3;
    z3 = y3*y3*y3*y3;
    printf("u3 = %f\n", u3);
    printf("v3 = %f\n", v3);
    printf("w3 = %f\n", w3);
    printf("z3 = %f\n", z3);
    z3 = w3 - z3;
    printf("double (schrittweise): %f\n", z3);

    return 0;
}
```

```
thales$ gcc -Wall float.c
thales$ a.out
int: 1
float: 1.000000
double (Formel umgestellt): -1.000000
u3 = 125372283822342144.000000
v3 = 708158978.000000
w3 = 125372284530501120.000000
z3 = 125372284530501120.000000
double (schrittweise): 0.000000
thales$
```

Anmerkung: Es ist also angebracht, sich zu überlegen, ob sowohl die *Zwischenergebnisse* also auch das *Endergebnis* noch im *Wertebereich* des verwendeten Datentyps liegen.

Außerdem sollte man bei ==-Vergleichen mit Gleitkommazahlen sehr vorsichtig sein:

Programm 7.5: Äquivalenztest bei Gleitkommazahlen (*gleich.c*)

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

#define MAX(a,b) ((a) > (b) ? (a) : (b))

#define EPS 1.0e-9        /* Genauigkeit */

int gleich (double x, double y) {
    double delta = fabs(x-y);
    if (delta < EPS)                              /* absoluter Vergleich */
        return 1;
    else
        return delta / MAX(fabs(x), fabs(y)) < EPS;   /* relativer Vergleich */
}

int main() {
    double x, y;

    x = 2.0;
    y = (2.0 / 0.77777777777777);
    y *=       0.77777777777777;

    if (x == y) /* hoechst problematisch! */
        puts("gleich");
    else
        puts("ungleich");

    if (gleich(x, y))             /* Ausweg */
        puts("gleich");
    else
        puts("ungleich");

    return 0;
}
```

```
thales$ gcc -Wall -lm gleich.c
thales$ a.out
ungleich
gleich
thales$
```

7.2.5 Aufzählungen (enum)

```
EnumType   ::=   "enum" [ Identifier ] "{" Enumerator { "," Enumerator} "}"
               | "enum" Identifier
Enumerator ::= Identifier [ "=" Constant ] .
```

- Steht zwischen **enum** und der Aufzählung ein Bezeichner (*identifier*), so kann anschließend damit die Aufzählung bezeichnet werden.
- Der Wert der Konstanten ist beim ersten Aufzählungselement 0, beim zweiten 1, usw.
- Wird beim *Enumerator* mit „=“ eine Konstante angegeben, so wird obige Angabe (0, 1, ...) überschrieben.

Programm 7.6: Verwendung von Aufzählungen (*enum.c*)

```
#include <stdio.h>

enum farbe {rot, gruen, blau};   /* Typ */

void print_farbe (enum farbe f) {
    switch (f) {
        case rot:     puts("rot"); break;
        case gruen:   puts("gruen"); break;
        case blau:    puts("blau"); break;
        default:      puts("UNKNOWN"); break;
    }
}

int main() {
    enum {Apfel, Birne, Feige} obst;    /* Variable */
    enum farbe topf;                    /* Variable */

    topf = rot;
    printf("%d ", topf);        /* enums sind Integer */
    print_farbe(topf);

    topf++;                     /* sogar Inkr./Dekr. geht */
    printf("%d ", topf);
    print_farbe(topf);

    topf = 10;                  /* auch direkte Zuweisung ... */
    printf("%d ", topf);        /* ... obwohl keine gueltige Farbe! */
    print_farbe(topf);

    obst = Birne;
    puts(obst == Apfel ? "A" : obst == Birne ? "B" : "F");

    return 0;
}
```

```
thales$ gcc -Wall enum.c
thales$ a.out
0 rot
1 gruen
10 UNKNOWN
B
thales$
```

7.2.6 Zeiger

Zeiger referenzieren Variablen eines bestimmten Typs. Bei der Deklaration eines Zeigers muss man den Typ der Variablen angeben, auf den er zeigen soll. Der *Wert eines Zeigers* ist die Adresse der Variablen, auf die er zeigt.

Sei folgender Programmausschnitt gegeben:

```
int i = 13;
int *p = &i;
```

In Abbildung 7.3 ist diese Situation im Hauptspeicher veranschaulicht. In der Zuweisung $p = \&i$

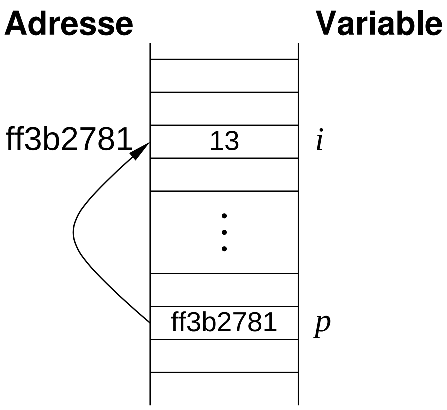


Abbildung 7.3: Beispiel für eine Zeiger-Variable

wird zunächst die Adresse von i mit dem *Adressoperator &* bestimmt und diese dann als Zeigerwert zugewiesen. Verwendet man nun die Zeigervariable p *ohne Dereferenzierung* , so ist ihr Wert gleich der Adresse von i (also eine **int** – abhängig vom System!). *Dereferenziert* man hingegen den Zeiger p mit dem *Dereferenzierungsoperator ** , also $*p$, so ist der Wert dieses Ausdrucks gleich dem *momentanen Wert der Variablen* (bzw. Speicherfläche), auf die der Zeiger zeigt. Ein dereferenzierter Zeiger, z. B. $*p$, ist ein *L-Wert*!

- Ein Zeiger, der auf *nichts zeigt* heißt *Null-Zeiger*. Konstanten hierfür sind 0 oder manchmal auch (**void***)0 bzw. als Makro *NULL* (in *stdlib.h* definiert).
- Ein Zeiger ist an den Typ gebunden, auf den gezeigt wird; er kann aber explizit auf andere Typen gewandelt werden (⇝ *type cast*).

Programm 7.7: 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=%08x (Adresse), *p=%d (Wert)\n", i, (int) p, *p);
```

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

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

    return 0;
}
```

```
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$
```

Mit Zeigern kann man in C auch *Arithmetik* betreiben (Addition und Subtraktion). Wird eine *Integer-Konstante* c auf einen Zeiger addiert bzw. von ihm subtrahiert, wobei der Zeiger mit $TYP * p$ vereinbart wurde, dann wird genau um c Objekte im Speicher weitergegangen, d. h. die Adresse des Zeigers wird um c * **sizeof**(TYP) inkrementiert bzw. dekrementiert. Addition und Subtraktion von *zwei Zeigern* ist nur erlaubt, wenn diesem vom *selben Typ* sind.

Programm 7.8: Zeiger-Arithmetik (*zeiger1.c*)

```
#include <stdio.h>

int main() {
    int i;

    long l = 0x12345678;
    unsigned char *p = (unsigned char*) &l;

    for (i = 0; i < 4; i++)
        printf("%02x\n", *(p+i));

    return 0;
}
```

```
thales$ gcc -Wall zeiger1.c # SPARC
thales$ a.out
12
34
56
78
thales$

chomsky$ gcc -Wall zeiger1.c # Intel
chomsky$ a.out
78
56
34
12
chomsky$
```

Anmerkung: In Programm 7.8 wird der Speicher wirklich byteweise „durchleuchtet“. Hierbei fällt auch auf, dass die intere Speicherung eines Integers bei unterschiedlichen Architekturen (SPARC und Intel x86) verschieden ist: *big endian* vs. *little endian*.

- **little endian**:
  das niedrigstwertige Byte wird an der niedrigsten Adresse abgelegt
- **big endian**:
  das niedrigswertige Byte wird an der höchsten Adresse abgelegt (siehe Abbildung 7.4)

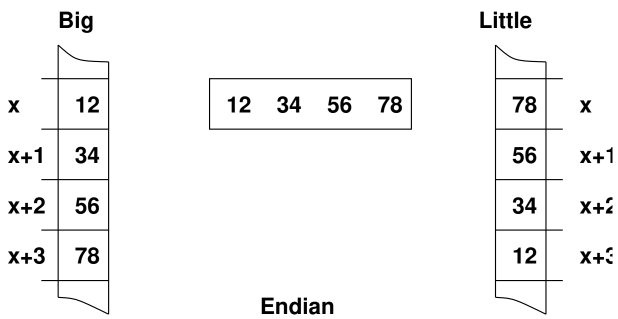


Abbildung 7.4: Little / Big Endian

7.3 Aggregierte Typen

7.3.1 Vektoren (Arrays)

```
Vector ::= Type Identifier "[" Constant "]" { "[" Constant "]" }
           [ Initializer ]
```

Ein *Array* ist in C ein *konstanter Zeiger* auf das *erste Element*. Der Typ ist ein *Zeiger auf den Elementtyp*. Bei mehrdimensionalen Arrays ist der Elementtyp auch wieder ein Arraytyp. Die *Indizierung* beginnt bei 0 und geht bis zur Größe des Arrays -1.

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

```
#include <stdio.h>

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

        /* Groesseninformation geht verloren, da nur Zeiger! */
    printf("SIZE=%d, sizeof(a)=%d, sizeof(p)=%d\n", SIZE, sizeof(a), sizeof(p));

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

    return 0;
}
```

```
thales$ gcc -Wall array1.c
thales$ a.out
SIZE=5, sizeof(a)=20, sizeof(p)=4
Summe: 15
thales$
```

Problem: Indizierungsfehler werden nicht durch eine Überprüfung erkannt! Evtl. tritt zur Laufzeit ein Fehler auf (⇝ *segmentation fault/violation*), wenn auf fremden Speicher (Adressraum außerhalb) zugegriffen wird.

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

```
#include <stdio.h>

#define SIZE 10

int main() {
    int x;
    int a[SIZE];
    int i, sum;

        /* wir wollen mal sehen, wo denn das alles im Speicher liegt */
    printf("&x = %08x, &i = %08x, &(a[0]) = %08x, &(a[SIZE+1]) = %08x\n",
        &x, &i, &(a[0]), &(a[SIZE+1]));

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

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

    /* typischer Fehler: ein Element zuviel */
    for (i = 0, sum = 0; i <= SIZE; i++)
        sum += a[i];
    printf("Summe: %d\n", sum);

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

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

    return 0;
}
```

```
thales$ gcc array1.c
thales$ a.out
&x = ffbff624, &i = ffbff5f4, &(a[0]) = ffbff5f8, &(a[SIZE+1]) = ffbff624
Summe: 285
Summe: 285
x=0, i=0
x=1, i=0
Segmentation Fault (core dumped)
thales$
```

7.3.1.1 Parameterübergabe

Der Name eines Arrays ist ein (konstanter) Zeiger auf das erste Element. Übergibt man also den Namen eines Arrays als Parameter (siehe Programm 7.11), so wird der Zeigerwert via Wertparameter übergeben; greift man über die lokale Kopie des Zeigerwerts auf den Vektor durch, ist man beim Orginal-Array!

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

```
#include <stdio.h>

#define SIZE 10

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

```
        a[i] = i;
}

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

int summe2(int *a, int length) {
    int i, sum = 0;
    for (i = 0, sum = 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));

    return 0;
}
```

Erklärung: Das Element *a[i]* kann äquivalent auch via **(a+i)* über *Zeigerarithmetik* referenziert werden. So wird auch der Zugriff auf Array-Elemente auch tatsächlich intern umgesetzt.

7.3.1.2 Mehrdimensionale Arrays

Definition, Initialisierung und Zugriff:

Ein *zweidimensionales Array* kann man beispielsweise wie folgt anlegen:

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

Mit folgender Definition wird ein *initialisiertes zweidimensionales Array* angelegt:

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

Im *Speicher* sieht dieses Array dann wie folgt aus:

Element	Adresse	Inhalt
matrix[0][0]	1000	0
matrix[0][1]	1004	1
matrix[0][2]	1008	2
matrix[1][0]	100C	3
matrix[1][1]	1010	4
matrix[1][2]	1014	5

(Obige Adressierung stellt nur ein Beispiel dar. Von Bedeutung sind die relativen Positionen der Elemente zueinander!)

matrix ist ein 2-elementiges Array, dessen Elemente 3-elementige Arrays sind, deren Elemente wiederum Integer sind. Nach wie vor gilt, dass eine Array-Variable ein konstanter Zeiger auf das erste Element ist. Somit ist *matrix* ein konstanter Zeiger auf 3-elementige Integer-Arrays. (Der Elementtyp dieses Zeigers ist also ein 3-elementiges Integer-Array.) Somit bewirkt ein Inkrementieren dieses Zeigers um Eins eine Erhöhung der Adresse um $3 \cdot 4 = 12$ (3 Array-Elemente á 4 Bytes). Wird dieser Zeiger dereferenziert, so hat man ein 3-elementiges Integer-Array, also einen Zeiger auf das erste Element dieses Arrays. Der Typ dieses Zeigers ist nun **int** ∗, so dass ein Inkrementieren dieses Zeigers eine Erhöhung der Adresse um 4 (4 Bytes pro Integer) 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 Array-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 als ∗(*matrix* [1]+2) und dies wiederum als ∗(∗(*matrix* + 1) + 2).

Betrachten wir nun ein dreidimensionales Array als weiteres Beispiel für die Adressierung:

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

a ist ein 2-elementiges Array, dessen Elemente 3-elementige Arrays sind, dessen Elemente wiederum 4-dimensionale Integer-Arrays sind. Eine andere Sichtweise, die im Hinblick auf die Adressierung sehr nützlich ist: *a* ist ein Zeiger auf ein 3-elementiges Array, dessen Elemente 4-elementige Integer-Arrays 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 ein 3-elementiges Array, dessen Elemente 4-elementige Integer Arrays 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 ein 4-elementiges Integer-Array. Ein Inkrement dieses Zeigers erhöht somit die Adresse um $4 \cdot 4 = 16$. Schließlich ist ∗(∗(*a* + *i*) + *j*) ein Zeiger auf eine Integer 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 kann man auch „von Hand“ nachbilden: ∗((**int**∗)*a* + *i* ∗ 3 ∗ 4 + *j* ∗ 4 + *k*) ist somit äquivalent zu *a*[*i*][*j*][*k*]. Der *type cast* auf Integer-Zeiger war nötig, damit die Inkrements in Integer-Schritten passieren. Dementsprechend mussten die Indizes i und j noch mit den Schrittweiten multipliziert werden. Der Faktor 4 fiel jeweils weg, da bei einem Integer-Zeiger die Adresse bei jedem Inkrement um 4 erhöht wird! Abbildung 7.5 veranschaulicht, wie das Array *a* im Speicher abgelegt ist und wie der Zugriff erfolgt.

Parameterübergabe:

Im folgenden Quellcode-Fragment ist die *Parameterübergabe eines mehrdimensionalen Arrays* vorgeführt:

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

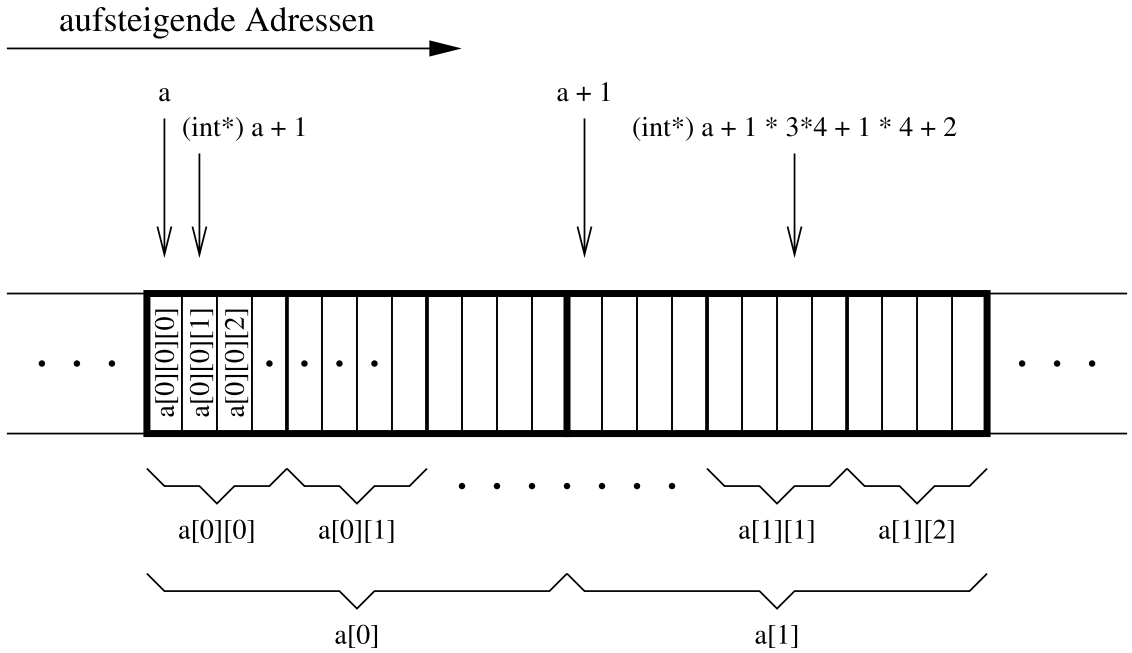


Abbildung 7.5: Visualisierung eines dreidimensionalen Arrays im Speicher

```
void f2(int b [][6][7], int size) {
    /* ... */

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

    /* ... */
}
```

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 ein 6-elementiges Array, dessen Elemente 7-elementige Integer-Arrays sind.

Bei obigem Beispiel ist jedoch vorausgestzt, dass die Dimensionen 2 und 3 bekannt und fest sind, nämlich 6 und 7. Auf dieser Basis kann der Compiler einen Zugriff $b[i][j][k]$ intern in $*((\textbf{int}*)b + i * 6*7 + j * 7 + k)$ umsetzen. Fehlt diese Information jedoch, d. h. die Funktion soll mehrdimensionale Arrays 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 muss man natürlich nun die Größen aller Dimensionen mitgeben.

Anmerkung: Das Array *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 einem *type case* in einen Integer-Zeiger „gecasted" werden. Dann kann man auf die Speicherfläche des Arrays „low level" zugreifen.

7.3.2 Zeichenketten (Strings)

Strings werden in C als *Arrays von Zeichen*, also vom Typ **char**[], repräsentiert. Das *String-Ende* wird durch ein *Null-Byte* ('\0') gekennzeichnet. Da es sich bei Strings also um Arrays handelt, werden bei der Parameterübergabe nur die Zeiger via *call by value* übergeben. Die Strings (= der Inhalt der Arrays) kann jedoch verändert werden.

String-Konstanten sind durch Doppelapostrophen eingeschlossene Zeichenfolgen, wie zum Beispiel "Ich␣bin␣ein␣String". String-Konstanten werden im Allgemeinen im *Read-Only-Speicherbereich* abgelegt, sie können also nicht verändert werden.

Das folgende Programm illustriert das Arbeiten mit Strings und String-Konstanten:

Programm 7.12: Strings und String-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'; */          /* i.a. nicht zulaessig , da die
                                  * Speicherflaeche , auf die s1
                                  * hier zeigt , im READ-ONLY-Bereich ist! */

    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 nicht im READ-ONLY wg. Array */
    printf("string: %s\n", string);
    printf("sizeof(string): %d\n", sizeof(string));    /* siehe oben */

    return 0;
}
```

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

Folgendes Beispiel-Programm zeigt, wie man die Länge von Strings bestimmen, Strings kopieren und vergleichen kann. Dies geschieht hier noch alles „von Hand“:

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

```
#include <stdio.h>

    /* Laenge eines Strings 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 eines Strings bestimmen */
int my_strlen2(char *s) {
    int len;
        /* bis zum abschliessenden Null-Byte laufen */
    for (len = 0; *s != '\0'; s++, len++);
    return len;
}

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

    /* Kopieren eines String
```

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

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

    /* Kopieren eines String
     * Vorauss.: genuegend Platz in t */
void my_strcpy4(char *t, char *s) {
    while (*t++ = *s++);          /* TRUE entspricht != 0 */
}

    /* Vergleich zweier Strings
     * 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 Strings
     * 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));

    return 0;
}
```

```
thales$ gcc -Wall strings1.c
strings1.c: In function ‘my_strcpy4’:
strings1.c:41: warning: suggest parentheses around assignment used as truth value
thales$ 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
thales$
```

Anmerkung: Weil das Ergebnis der Zuweisung $*t{+}{+} = *s{+}{+}$ als Wahrheitswert verwendet wird, gibt der Compiler eine Warnung aus, denn es könnte sich ja um die beliebte Verwechslung von = und == handeln.

In *strings.h* gibt es eine Reihe von Funktionen zum Arbeiten mit Strings (man muss also nicht immer alles selbst machen). Im Folgenden sind einige ausgewählte Funktionen vorgestellt. Nähere Informationen und weitere Funktionen siehe Manpage zu *strings.h* (z. B. `man strcpy`).

size_t strlen(const char *s)
: Liefert die Länge des Strings `s`. (*size_t* ist der Typ für Größenangaben – normalerweise identisch mit **int**.)

char *strcpy(char *s1, const char *s2)
: Kopiert den Inhalt von *s2* nach *s1* und gibt das Resultat zurück. (Es muss in *s1* genügend Platz vorhanden sein! Was, wenn nicht???)

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 Strings *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)
: Vergleich die beiden Strings, 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, das nur aus Zeichen von *s2* besteht. (Ein Präfix ist ein Teilstring, 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 des Strings *s2* (bis auf das abschließende Null-Byte) innerhalb von *s1* zurück. Ist *s2* kein Teilstring von *s1*, so ist das Ergebnis der Null-Zeiger. Ist *s2* der leere String (""), so ist das Ergebnis *s1*.

char *strtok(char *s1, const char *s2)

Die Funktion *strtok* () dient dazu, den String *s1* in Abschnitte (*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 Abschnitts (der inzwischen *nullterminiert* ist – *strtok* () *verändert also den String s1*) zurück. Um einen Zeiger auf das erste Zeichen des zweiten Abschnitts (und weiterer Abschnitte) zu erhalten, muss als *s1* ein Null-Zeiger übergeben werden. Hierbei kann derselbe String *s2* oder auch ein anderer verwendet werden. (*strtok* () speichert also intern die Position im String – über verschiedene Funktionsaufrufe hinweg.) Ein Null-Zeiger als Rückgabewert signalisiert, dass es keinen weiteren Abschnitt mehr gibt.

Anmerkung: Die Position innerhalb des Strings wird von *strtok* () in einer globalen Variable gespeichert. Bei einem Aufruf von *strtok* (), bei dem *s1* nicht der Null-Zeiger ist, geht die Position im vorigen String verloren. *strtok* () kann also *nicht gleichzeitig für zwei verschiedene Strings* verwendet werden!

Programm 7.14: Verwendung von Funktionen der String-Bibliothek (*strings2.c*)

```c
#include <stdio.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));

    return 0;
}
```

```
thales$ gcc -Wall strings2.c
thales$ 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!"
thales$
```

Und nun noch zum Abschluss ein Beispiel für die Verwendung der Funktion *strtok* ():

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

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

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

    /* String in Abschnitte (tokens) zerlegen */

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

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

    /* Inhalt des Arrays text nach strtok() ausgeben */

    for (i = 0; i < len; i++)
        printf("%c", text[i]);
    puts("");

    return 0;
}
```

```
thales$ gcc -Wall strings3.c
thales$ a.out
Das
macht
doch
riesigen
Spass
oder
-----------------------------------------
DasmachtdochriesigenSpass oder!
thales$ a.out | tail -1
DasmachtdochriesigenSpass oder!
thales$ a.out | tail -1 | od -bc
0000000 104 141 163 000 155 141 143 150 164 000 144 157 143 150 000 162
          D   a   s  \0   m   a   c   h   t  \0   d   o   c   h  \0   r
0000020 151 145 163 151 147 145 156 000 123 160 141 163 163 000 040 157
          i   e   s   i   g   e   n  \0   S   p   a   s   s  \0       o
0000040 144 145 162 000 041 012
          d   e   r  \0   !  \n
0000046
thales$
```

7.3.3 Strukturen (Records)

7.3.3.1 Einfache Strukturen

Eine *Struktur* (*Record*) fasst mehrere *Elemente* zu einem Typ zusammen. Im Gegensatz zu Arrays können die Elemente *unterschiedlichen Typs* sein.

Mit dem Schlüsselwort **struct** kann man analog zu den Records in Oberon/Modula-2 eine Struktur wie folgt deklarieren:

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

datum ist der *Name der Struktur* und kein Typ. Die soeben deklarierte Struktur repräsentiert – wie der Name schon sagt – ein Datum. Jede Variable dieser Struktur besteht aus *drei ganzzahligen Komponenten*, dem Tag, dem Monat und dem Jahr.

Eine *Variable* *geburtsdatum* von diesem (Struktur-)Typ kann danach wie folgt angelegt werden:

```
struct datum geburtsdatum;
```

Analog zu Aufzählungen kann man auch Variablen von namenlosen Strukturen anlegen:

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

Man kann eine Struktur-Variable auch bei ihrer Definition schon *initialisieren*:

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

Auf die *Komponenten* einer Struktur kann man wie folgt *zugreifen*:

```
struct datum gebdat = ...;

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

struct datum *p = ...;

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

7.3.3.2 Geschachtelte Strukturen

Die Elemente von Strukturen können (beinahe) beliebigen Typs sein. Insbesondere ist es auch möglich, *Strukturen ineinander zu schachteln*:

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

Wenn dann eine Variable als `struct person p` vereinbart ist, dann kann man wie folgt *auf die Elemente zugreifen*:

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

Anmerkung: Man kann auch Variablen von *unbenannten Strukturen* in andere Strukturen einbetten:

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

7.3.3.3 Rekursive Strukturen

Zeiger auf Strukturen 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 *„declare before use“*, d. h., dass alles was verwendet wird *vorher deklariert* werden muss. Diese Ausnahme ermöglicht *rekursive* bzw. *zyklisch verkettete* Strukturen:

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

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

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

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

7.3.3.4 Zuweisung von Strukturen

Variablen vom gleichen Struktur-Typ 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 Strukturen (*struct1.c*)

```c
#include <stdio.h>

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

int main() {
    struct datum vorl_beginn = {14, 10, 2003};
    struct datum ueb_beginn = {21, 10, 2003};

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

        /* komponentenweise Zuweisung => Vorsicht bei Zeigern! */
    vorl_beginn = ueb_beginn;

    printf("nachher: %d.%d.%d\n", vorl_beginn.tag, vorl_beginn.monat, vorl_beginn.jahr);

    return 0;
}
```

```
thales$ gcc -Wall struct1.c
thales$ a.out
vorher: 14.10.2003
nachher: 21.10.2003
thales$
```

7.3.3.5 Strukturen als Funktionsargumente

Wie üblich kann man durch *Zeiger eine call-by-reference-Semantik* erreichen, wobei *normalerweise eine call-by-value-Semantik* vorliegt:

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

```c
#include <stdio.h>

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

    /* call-by-value-Semantik */
void ausgabe1(struct datum d) {
    printf("%d.%d.%d\n", d.tag, d.monat, d.jahr);
}

    /* call-by-reference-Semantik (hier egal) */
void ausgabe2(struct datum *d) {
    printf("%d.%d.%d\n", d->tag, d->monat, d->jahr);
}

    /* call-by-value-Semantik => aendert nichts, da d Kopie! */
void setJahr1(struct datum d, int jahr) {
    d.jahr = jahr;
}

    /* call-by-reference-Semantik => gewuenschtes Ergebnis */
void setJahr2(struct datum *d, int jahr) {
    d->jahr = jahr;
}

int main() {
    struct datum start = {13, 10, 2003};

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

    return 0;
}
```

```
thales$ gcc -Wall struct2.c
thales$ a.out
13.10.2003
13.10.2003
13.10.2004
thales$
```

7.3.3.6 Strukturen als Ergebnis von Funktionen

Funktionen können auch *Strukturen als Rückgabewerte* haben. *Vorsicht* ist jedoch angebracht, wenn *Zeiger auf Strukturen* zurückgegeben werden:

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

```c
#include <stdio.h>

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

void ausgabe(struct datum d) {
    printf("%d.%d.%d\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; /* problematisch, 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 isses reiner Zufall!! */
    ausgabe(*p);        /* sollte eigentliche dasselbe ausgeben :-( */

    return 0;
}
```

```
thales$ gcc -Wall struct3.c
struct3.c: In function 'init2':
struct3.c:18: warning: function returns address of local variable
thales$ a.out
1.1.1900
1.1.1900
-215.-16784.1900
thales$
```

Erklärung: Die Variable *d* in der Funktion *init2* () ist eine lokale Variable (also auf dem *Stack*). *Sie existiert nur solange diese Funktion ausgeführt wird.* Danach wird dieser Speicherplatz evtl. anderweitig verwendet (siehe Abbildung 7.6). In unserem Fall findet beim ersten Aufruf von *ausgabe()*, bei dem das referenzierte Objekt noch unverändert als Parameter kopiert werden kann, und dem dabei erfolgenden (Unter-)Funktionsaufruf *printf()* eine anderweitige Verwendung des referenzierten Speicherplatzes statt. Folglich ist das „Datum“, welches beim zweiten Aufruf von *ausgabe()* ausgegeben wird, durch den ersten Aufruf von *ausgabe()* (und *printf()*) auf den Stack gelangt (siehe Abbildung 7.6).

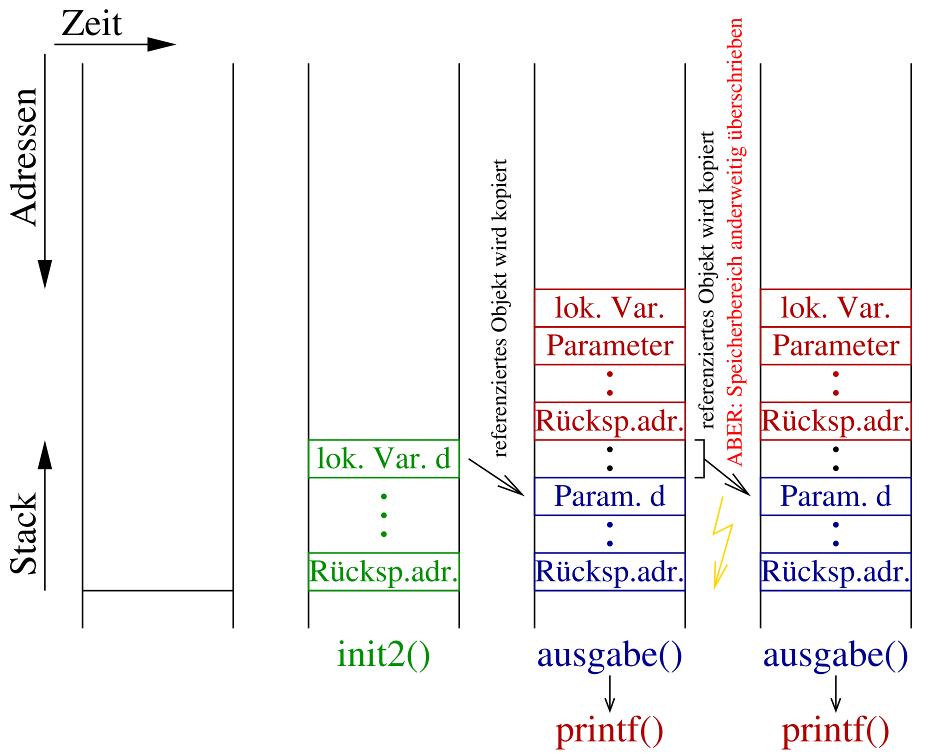


Abbildung 7.6: Funktionsaufrufe und lokale Variablen

7.4 Unions (union)

Analog zu Strukturen gibt es *Unions* (*union*), die *mehrere Elemente* zu einem Typ *zusammen fassen*. Der Unterschied zwischen Strukturen und Unions ist, dass die Elemente einer Struktur an *verschiedenen Stellen* im Speicher stehen, wohingegen die Elemente einer Union an *derselben Stelle* im Speicher stehen. Somit kann man bei einer Struktur alle Elemente gleichzeitig verwenden (d. h. mit Werten belegen), wohingegen man bei einer Union zu einem Zeitpunkt nur ein Element verwenden (d. h. mit einem Wert belegen) kann.

In folgendem Beispielprogramm wird eine Union zur Repräsentation einer IP-Adresse verwendet. Daher enthält diese Union eine Integer und ein 4-elementiges Byte-Array, die an derselben Stelle im Speicher stehen. Somit kann man auf die IP-Adresse einmal byteweise und ein anderes Mal integerweise zugreifen.

Programm 7.20: Verwendung von Unions (*union.c*)

```c
#include <stdio.h>

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

int main() {
    union IPAddr a;


    a.ip = 0x86423c05;  /* 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("%u.%u.%u.%u ", a.b[0], a.b[1], a.b[2], a.b[3]);
    printf("[%x.%x.%x.%x]\n", a.b[0], a.b[1], a.b[2], a.b[3]);

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

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

    printf("a:    %x\n", (unsigned int) &a);

    printf("ip:   %x\n", (unsigned int) &a.ip);

    printf("b[0]: %x\n", (unsigned int) &a.b[0]);
    printf("b[1]: %x\n", (unsigned int) &a.b[1]);
    printf("b[2]: %x\n", (unsigned int) &a.b[2]);
    printf("b[3]: %x\n", (unsigned int) &a.b[3]);

    return 0;
}
```

Ausführung auf einer *SPARC-Maschine*:

```
thales$ gcc -Wall union.c
thales$ a.out
2252487685 [86423c05]
134.66.60.5 [86.42.3c.5]

Speicherplatzbedarf: 4

Position im Speicher:
a:    ffbef634
ip:   ffbef634
b[0]: ffbef634
b[1]: ffbef635
b[2]: ffbef636
b[3]: ffbef637
thales$
```

Ausführung auf einer *Intel-x86-Maschine*:

```
chomsky$ gcc -Wall union.c
chomsky$ a.out
2252487685 [86423c05]
5.60.66.134 [5.3c.42.86]

Speicherplatzbedarf: 4

Position im Speicher:
a:    bffff348
ip:   bffff348
b[0]: bffff348
b[1]: bffff349
b[2]: bffff34a
b[3]: bffff34b
chomsky$
```

Warnung: Vorsicht beim Arbeiten mit Unions! In obigem Beispiel sieht man, dass auf unterschiedlichen Architekturen (SPARC und Intel x86) ein anderes Ergebnis geliefert wird. Der Grund dafür ist die unterschiedliche interne Repräsentation einer Integer im Speicher. Einmal kommt das höchstwertige Byte zuerst (*big endian* bei SPARC) und ein anderes Mal kommt das niederwertigste Byte zuerst (*little endian* bei Intel x86).

7.5 Eigene Typnamen (typedef)

Mit dem Schlüsselwort **typedef** können *eigene Datentypnamen* bzw. *Synonyme* vereinbart werden.

```
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* also ein Synonym für **int**. Damit sind **int** $i, j;$ und *Laenge* $i, j;$ äquivalente Vereinbarungen.

Mit **typedef** wird *kein neuer Datentyp* (wie in Modula-2 oder Oberon mit TYPE) konstruiert. Es wird nur ein Name für einen anderweitig auch existierenden Datentyp eingeführt.

typedef ist nützlich im Bezug auf *Portabilität*: Führt man Datentypen mittels **typedef** ein, so muss ggf. nur die Typ-Vereinbarung (bei **typedef**) geändert werden.

Mittels **typedef** kann ein Programm auch *besser dokumentiert* werden (sprechende Typ-Namen).

typedef verhält sich ähnlich wie *define* — nur wird *typedef* vom Compiler verarbeitet, während *define* vom Preprozessor verarbeitet wird.

```
#define USHORT unsigned int
```

ist (nahezu) äquivalent zu

```
typedef unsigned int USHORT;
```

Aber Vorsicht:

```
#define PINT int *
PINT p1, p2;
```

wird expandiert zu **int** $* \ p1, \ p2;$ damit ist nur *p1* ein **int**-Zeiger, *p2* ist eine **int**-Variable!

Mit

```
typedef int * PINT;
PINT p1, p2;
```

werden hingegen zwei **int**-Zeiger vereinbart.

7.6 Komplexe Deklarationen

C ist eine ganz schön kryptische Sprache. In diesem Abschnitt werden wir diese Seite von C zu genüge beleuchten. Dies soll nicht dazu anregen unnötigerweise komplizierte Konstrukte entwerfen. Vielmehr ist es das *Ziel*, C-Programme auch mit *komplexeren Deklarationen zu verstehen* und selbst *einigermaßen komplexe Deklarationen zu entwickeln*. Dies üben wir in den folgenden Beispielen. Hierzu ist es gut, immer mal wieder einen Blick in die *Tabelle mit den Prioritäten der einzelnen Operatoren* zu werfen (siehe Abschnitt 6.2.1)!

Beispiel:

x ist ein Array von Zeigern auf **char**:

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

Dies entziffern wir nun Stück für Stück:

`x[]`	Array
`*x[]`	Array von Zeigern
`char *x[]`	Array von Zeigern auf **char**

Beachte: `[]` bindet stärker (hat höhere Priorität) als $*$!

Beispiel:

x ist ein Zeiger auf eine Funktion, die einen Zeiger auf ein 5-elementiges Array liefert, dessen Elemente wiederum Zeiger auf **int**s sind:

$$\textbf{int}\ *(*(*x)\,()\,)\,[5];$$

Versuchen wir das nun von innen nach außen zu entziffern:

`*x`	Zeiger
`*(*x)()`	`()` bindet stärker als `*`, also ist `*(*x)()` ein Zeiger auf eine Funktion, die einen Zeiger liefert.
`int *(*(*x)())[5]`	`[]` bindet stärker als `*`, also ist `x` Zeiger auf eine Funktion, die einen Zeiger auf ein Array liefert, dessen 5 Elemente Zeiger auf `int`s sind.

Transparenter wird dies durch stufenweisen Aufbau mit **typedef**:

$$\textbf{typedef}\ \textbf{int}\ *AP[5];$$

AP ist damit Name für ein 5-elementiges Array mit Zeigern auf **int**.

$$\textbf{typedef}\ AP\ *FP();$$

FP ist damit ein Name für eine Funktion, die Zeiger auf *AP* liefert.

$$FP\ *\ x;$$

Nun ist x ein Zeiger auf eine Funktion vom Typ *FP*.

Beachte:

- `[]` und `()` haben *höhere Priorität* als $*$!.
- [] und () werden von *links nach rechts* gruppiert, $*$ von *rechts nach links*.

Beispiel:

Man beachte, dass Klammern den Vorrang ändern. Dies wird im Folgenden deutlich:

$$\textbf{int}\ (*x\,[])\,()\,;$$

In diesem Fall ist x ein Array von Zeigern auf Funktionen, deren Rückgabewert von Typ **int** ist. Dies können wir auch wiederum schrittweise entwickeln:

`*x[]`	Array von Zeigern (`[]` bindet stärker als `*`)
`(*x[])()`	Array von Zeigern auf Funktionen
`int (*x[])()`	Array von Zeigern auf Funktionen, die **int** liefern

Ohne Klammern sieht es dann aber anders aus. In diesem Fall ist x ein Array von Funktionen, deren Rückgabewert vom Typ Zeiger auf **int** ist.

$$\textbf{int}{*x}\,[]()\,;$$

Man beachte jedoch, dass diese Definition *nicht zulässig* ist, da Arrays mit Funktionen als Elementen nicht erlaubt sind!

Beispiel:

x ist ein Zeiger auf ein Array, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Arrays liefern, deren Elemente Strukturen mit dem Namen `S` sind:

$$\textbf{struct}\ \ S\ \ (*(*(*\,x)\,[])\,()\,)\,[];$$

Dies entziffern wir nun wieder schrittweise:

`*x`	Zeiger
`(*x)[]`	Zeiger auf Array
`*(*x)[]`	Zeiger auf Array, dessen Elemente Zeiger sind
`(*(*x)[])()`	Zeiger auf Array, dessen Elemente Zeiger auf Funktionen sind
`*(*(*x)[])()`	Zeiger auf Array, dessen Elemente Zeiger auf Funktionen sind, die Zeiger liefern
`(*(*(*x)[])())[]`	Zeiger auf Array, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Arrays liefern
`struct S (*(*(*x)[])())[]`	Zeiger auf Array, dessen Elemente Zeiger auf Funktionen sind, die Zeiger auf Arrays liefern, deren Elemente Strukturen mit dem Namen *S* sind

Beispiele für unzulässige Deklarationen:

`int af[]()`	Array von Funktionen, die `int` liefern
`int fa()[]`	Funktion, die ein Array mit `int`s liefert (`int *fa()` wäre ok)
`int ff()()`	Funktion, die eine Funktion liefert, welche wiederum `int` liefert

Kapitel 8

Funktionen

Funktionen sind in C wie folgt definiert:

```
FunctionDefinition ::= FunctionType FunctionHeader CompoundStatement
FunctionType       ::= Type
FunctionHeader     ::= Identifier "(" [ ArgumentList ] ")"
ArgumentList       ::= Type Identifier { "," ArgumentList }
Identifier         ::= Letter { Letter | Digit }
Letter             ::= "a" | ... | "z" | "A" | ... | "Z" | "_"
Digit              ::= "0" | ... | "9"
```

- Die Definition einer Funktion besteht aus dem *Typ des Rückgabewertes*, dem *Namen der Funktion*, einer möglicherweise leeren *Liste von Argumenten* (d. h. jeweils Typ und Argumentname) und dem *Anweisungsblock*.
- Funktionen sind immer *global*, d. h. sie dürfen nicht geschachtelt werden (im Gegensatz zu Modula-2 oder Oberon).
- Die Parameterübergabe ist immer *call by value*.
- Bei *Prozeduren* (= Funktionen ohne Resultat) sollte *void* als Typ des Rückgabewertes angegeben werden. Ist kein Ergebnistyp angegeben, so wird *implizit int* angenommen!
- Der *Name einer Funktion* ist zugleich ein *Zeiger* auf diese Funktion. (Dies ist aber nicht portabel. Man konstruiert einen Zeiger besser mit dem Adressoperator „&".)

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

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

```c
#include <stdio.h>

int fib(int n) {          /* Berechnung der Fibonacci-Zahlen */
    if (n == 0 || n == 1)     /* Ende der Rekusion */
        return 1;
    else                      /* Berechnung mittel rekursiver Aufrufe */
```

```
        return fib(n-1) + fib(n-2);
}

int main() {
    int n = 7;

    printf("fib(%d) = %d\n", n, fib(n));

    return 0;
}
```

Ein *return* beendet die aktuelle Funktion (wie bei Modula-2 und Oberon). Der Wert des angegebenen Ausdrucks ist der *Rückgabewert* der Funktion und somit der Wert des Funktionsaufrufs. Ein *return* innerhalb von *main()* hat die *Terminierung* (Beendigung) des Programms zur Folge.

8.1 Variablen-Parameter (call by reference)

Die Parameterübergabe erfolgt in C – wie bereits erwähnt – via *call by value*. Jedoch lässt sich die Semantik von *call by reference* sehr einfach mit Hilfe von Zeigern emulieren, wobei dies jedoch *nicht transparent* ist, denn Variablen-Parameter müssen – infolge der Emulation mit Zeigern – *anders gehandhabt* (d. h. verwendet bzw. zugegriffen) werden als Werte-Parameter.

Programm 8.2 enthält eine Methode *swap*() zum Vertauschen der Werte zweier Integer-Variablen. Hierbei wird durch die Verwendung von Zeigern die Semantik von *call by reference* emuliert:

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

```
#include <stdio.h>

    /*
     * Diese Prozedur vertauscht die Werte der
     * int-Variablen, auf die a und b zeigen;
     * a und b sind quasi Variablen-Parameter!
     * *a und *b sind Links-Werte, die die (als Zeiger)
     * uebergebenen Variablen bezeichnen.
     */
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);

        /*
         * Aus x und y Zeiger konstruieren
         * fuer die Variablen-Parameter-Uebergabe
         */
    swap(&x, &y);
```

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

    return 0;
}
```

Da man die Zeiger dereferenzieren muss, kann man Variablen-Parameter in C nicht genauso wie die übergebenen Variablen handhaben (⇝ *nicht transparent*). 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-Deklaration von Funktionen (forward declarations)

In C muss eine Variable oder eine Funktion erst deklariert sein, bevor sie verwendet werden kann. Wir haben Vorab-Deklarationen zum Beispiel bei rekursiven Strukturen kennen gelernt. Wenn zwei oder mehr Funktionen sich jeweils gegenseitig aufrufen, dann gibt es keine Reihenfolge für diese Funktions-Definitionen, so dass immer nur bereits definierte Funktionen verwendet werden. Dieses Problem kann man lösen, indem man zunächst die Funktionen deklariert. Das folgende Beispielprogramm zeigt, wie in diesem Fall die Funktion/Prozedur *proc2* () vor ihrer Definition deklariert wird. Dies nennt sich dann *Vorab-Deklaration* (*forward declaration*).

Programm 8.3: Vorab-Deklarationen bei Funktionen (*forward.c*)

```
#include <stdio.h>

    /*
     * Vorab-Deklaration notwendig, da
     * proc1 und proc2 jeweils einander aufrufen!
     */
void proc2(int);

    /* Prozedur proc1: basiert auch Prozedur proc2 */
void proc1(int a) {
    printf ("proc1: a = %d\n", a);
    if (a > 0)
        proc2(a-1);
}

    /* Prozedur proc2: basiert auch Prozedur proc1 */
void proc2(int b) {
    printf ("proc2: b = %d\n", b);
    if (b > 0)
        proc1(b-1);
}

int main() {
        /* Kette von Funktionsaufrufen anstossen */
    proc1(4);

    return 0;
}
```

```
thales$ gcc -Wall forward.c
thales$ a.out
proc1: a = 4
proc2: b = 3
proc1: a = 2
proc2: b = 1
proc1: a = 0
thales$
```

8.3 Funktionszeiger

Der *Name einer Funktion* ist zugleich auch ein *Zeiger auf diese Funktion*. (Dies ist aber nicht portabel. Man konstruiert einen Zeiger besser mit dem Adressoperator „&".) 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 8.4 verwendet wird, zum leichteren Verständnis illustriert.

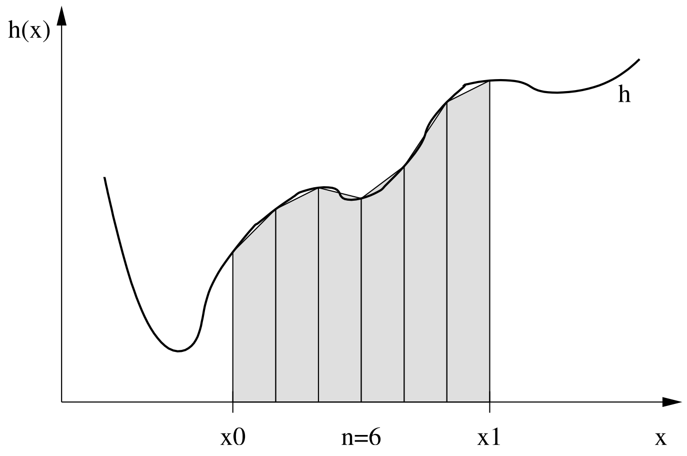


Abbildung 8.1: Integration nach der Trapezregel

Dabei 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 $\Delta := (x1 - x0)/n$. Die obige Formel lässt sich wie folgt umschreiben:

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

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

$$= \left(\left(\sum_{i=0}^{n} h(x0+i\Delta)\right) - \frac{h(x0)+h(x1)}{2}\right)\Delta$$

Diese letzte Formel wird in Programm 8.4 verwendet.

Programm 8.4: Funktionszeiger: Integration nach der Trapezregel (*mathfun.c*)

```c
#include <stdio.h>

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

   /*
    * Numerische Integration der Funktion h nach der Trapezregel
    * h: 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;
   int i;
   for (i = 0, x = x0; 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));

   return 0;
}
```

```
thales$ gcc -Wall mathfun.c
thales$ a.out
Numerische Integration:
f: 0.333350
g: 0.166700
thales$
```

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! (Statt $(*h)(x)$ kann man auch einfach $h(x)$ schreiben, was aber wiederum nicht so portabel ist.)

Kapitel 9

Dynamische Datenstrukturen

9.1 Allozieren und Freigeben von Speicher

Analog zu NEW oder ALLOCATE (in Oberon oder Modula-2) und DISPOSE oder DEALLOCATE (in Modula-2) gibt es in C ebenfalls Funktionen, mit denen dynmisch Speicherplatz beschafft bzw. wieder frei gegeben werden kann. Dies sind im Einzelnen die folgenden Funktionen, die in *stdlib.h* deklariert sind (d. h. es ist ein **#include** <stdlib .h> erforderlich):

void *calloc(size_t nelem, size_t elsize)
: Diese Funktion liefert einen Zeiger auf eine mit Null (d. h. Null-Bytes) initialisierte Speicherfläche für *nelem* Objekte der Größe elsize (Bytes). Ist nicht gegnügend Speicherplatz vorhanden, so ist das Ergebnis der Null-Zeiger.

void *malloc(size_t size)
: Diese Funktion liefert einen Zeiger auf eine Speicherfläche der Größe *size* (Bytes). Falls nicht mehr genügend (zusammenhängender) Speicherplatz vorhanden ist, wird der Null-Zeiger zurückgegeben.
Achtung: Aufgrund der Speicherinitialisierung bei *calloc()* und von Implementierungsfehlern von *malloc()* bei manchen Compiler, ist *calloc()* der Funktion *malloc()* vorzuziehen!

void *realloc(void *ptr, size_t size)
: Ändert die Große der (bisher) über ptr referenzierten Speicherfläche auf die Größe *size* (Bytes). Der Rückgabewert ist ein Zeiger auf die in der Größe veränderte Speicherfläche. Es kann sein, dass sich die Speicherfläche an einer anderen Stelle befindet als vor dem Aufruf. Der Inhalt (soweit bisher definiert) bleibt aber in jedem Fall erhalten. *realloc()* liefert einen Null-Zeiger, falls nicht genügend Speicher vorhanden ist.

void free(void *ptr)
: Gibt den Speicherplatz, der sich hinter *ptr* befindet und vorher mit *calloc()*, *malloc()* oder *realloc()* alloziert wurde, wieder frei. Wenn *ptr* ein Null-Zeiger ist, passiert gar nichts.

Folgendes Beispielprogramm liest von der Standardeingabe eine Integer. Diese Integer wird dann als Größe eines Zeichen-Arrays (Strings) interpretiert und ein Zeichen-Array dieser Größe dynamisch alloziert. Danach wird eine (möglicherweise nicht vollständige) Zeile von der Standardeingabe in dieses Array eingelesen und sofort wieder ausgegeben. Zum Schluß wird noch der Speicherplatz wieder frei gegeben.

Programm 9.1: Dynamisches Allozieren von Speicher für ein Array (*alloc.c*)

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

int main() {
    int len;
    char *s;

    /* Groesse des Arrays von stdin lesen */
    scanf("%d\n", &len);

    /* Array allokieren mit Fehlerbehandlung */
    if ((s = (char*) calloc(len, sizeof(char))) == NULL) {
        fprintf(stderr, "Not enough memory available\n");
        return 1;
    }

    /* Zeile von stdin in neuen String einlesen ... */
    fgets(s, len, stdin);

    /* ... und gleich wieder ausgeben */
    printf("Eingabe: \"%s\"\n", s);

    /* Speicherplatz wieder freigeben */
    free(s);
    s = NULL; /* => zur Sicherheit => kein Zugriff mehr mgl.! */

    return 0;
}
```

```
thales$ gcc -Wall alloc.c
thales$ a.out
10
Dies ist eine kleiner Test
Eingabe: "Dies ist "
thales$
```

Anmerkung: Der Zeiger, der von den Allozierungsfunktionen zurückgegeben wird, ist vom Typ **void** ∗. Er sollte erst noch mittels *type cast* in den gewünschten Zeigertyp umgewandelt werden (oben z. B. **char** ∗). Es empfiehlt sich, wie in obigem Beispiel, Zeiger, die noch auf freigegebene Speicherflächen zeigen, zur Sicherheit mit *NULL* zu überschreiben.

9.2 Dynamische Arrays – Ein Beispiel

Ein Problem von C ist ja, dass Arrays einfach Zeiger auf das erste Element sind und keine Größeninformation enthalten. Folgendes Programm versucht diese Schwäche zu beheben, indem es eine dynamische Datenstruktur für Integer-Arrays einführt. Außerdem überprüft es auch, ob bei einem Zugriff auf eine Array-Element der Index im erlaubten Bereich ist.

Programm 9.2: Integer-Array mit Größeninformation (*dynarray.c*)

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

#define ERROR -99999999

typedef struct {
            int *array; /* Zeiger auf das erste Element des Arrays */
            int size;   /* Groesse des allokierten Speichers */
            int used;   /* tatsaechliche Groesse des Arrays; used <= size! */
        } Array;

/*
 * Neues Integer-Array mit der angegebenen
 * Groesse erzeugen und zurueck geben
 *
 * Parameter "Array **a" notwendig, um Array-Zeiger
 * call by reference zu uebergeben!
 */
void arrayNew(Array **a, int len) {
    *a = (Array*) calloc (1, sizeof(Array));
    (*a)->size = (*a)->used = len;
    (*a)->array = (int*) calloc (len, sizeof(int));
}

/*
 * Integer-Array wieder freigeben
 */
void arrayDelete (Array **a) {
    free ((*a)->array);
    (*a)->array = NULL;
    free (*a);
    *a = NULL;
}

/*
 * Groesse des Integer-Arrays veraendern
 */
void arrayResize (Array *a, int len) {
    if (a->size < len) {
        a->array = (int*) realloc (a->array, sizeof(int) * len);
        a->size = a->used = len;
    }
    else
        a->used = len;
}

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

```c
/*
 * Element des Arrays lesen
 */
int arrayGet(Array *a, int i) {
    if (i < 0 || i >= a->used)
        return ERROR; /* :-( */
    return a->array[i];
}

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

int main() {
    int i;
    Array *a;

    arrayNew(&a, 5);

    for (i = 0; i < arrayLength(a); i++)
        arraySet(a, i, i);

    for (i = 0; i < arrayLength(a); i++)
        printf("a[%d] = %d\n", i, arrayGet(a, i));

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

    arrayResize(a, 3);

    for (i = 0; i < arrayLength(a); i++)
        printf("a[%d] = %d\n", i, arrayGet(a, i));

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

    arrayResize(a, 7);
    arraySet(a, 5, 10);

    for (i = 0; i < arrayLength(a); i++)
        printf("a[%d] = %d\n", i, arrayGet(a, i));

    arrayDelete(&a);

    return 0;
}
```

```
thales$ gcc -Wall dynarray.c
thales$ a.out
a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4
-----------
a[0] = 0
a[1] = 1
a[2] = 2
-----------
a[0] = 0
a[1] = 1
a[2] = 2
a[3] = 3
a[4] = 4
a[5] = 10
a[6] = 0
thales$
```

9.3 Dynamische Strings

Zum Erzeugen dynamischer Strings gibt es eine eigene Funktion *strdup* (). Diese Funktion bekommt einen String übergeben, beschafft dann mit *malloc* () Speicherplatz für eine Kopie und kopiert den String danach. Sie liefert dann einen Zeiger auf die Kopie bzw. einen Null-Zeiger, wenn kein Speicher alloziert werden konnten. Somit passiert bei

```c
char *s1 = strdup(s);
```

und bei

```c
char *s1 = malloc( strlen (s)+1);
if (s1) /* nur bei Erfolg ( also s1 != NULL) kopieren! */
    strcpy(s1, s);
```

also genau dasselbe, nur dass die erste Schreibweise mit *strdup* () deutlich kürzer ist.

9.4 Speicher-Operationen

Die Bibliothek *string.h* enthält außer String-Operationen auch eine Reihe von *Speicher-Operationen*, die oft im Kontext von dynamischen Datenstrukturen Verwendung finden. Im Einzelnen sind dies u. A. folgende Operationen:

void *memset(void *s, int c, size_t n)
: Initialisiert die ersten n Bytes des Speichers, auf den s zeigt, mit c (konvertiert in den Typ **unsigned char**). Der Rückgabewert ist s.

void *memcpy(void *s1, const void *s2, size_t n)
Kopiert n Bytes von $s2$ nach $s1$. (Diese Operation nur bei nicht überlappenden Speicherflächen $s1$ und $s2$ verwenden.) Der Rückgabewert ist $s1$.

void *memmove(void *s1, const void *s2, size_t n)
Kopiert n Bytes von $s2$ nach $s1$. Diese Operation kopiert selbst dann korrekt, wenn die Speicherflächen $s1$ und $s2$ überlappen. Der Rückgabewert ist $s1$.

int memcmp(const void *s1, const void *s2, size_t n)
Vergleicht die ersten n Bytes (als **unsigned char**) der Speicherflächen auf die $s1$ und $s2$ zeigen und gibt einen Wert (negativ, Null, positiv) analog zur *strcmp()*.

void *memchr(const void *s, int c, size_t n)
Sucht das erste Vorkommen von c (als **unsigned char**) in den ersten n Bytes des Speicherbereichs, auf den s zeigt. Der Rückgabewert ist ein Zeiger auf das erste Vorkommen und der Null-Zeiger, falls c nicht in den ersten n Bytes vorkommt.

Programm 9.3: Arbeiten mit Speicher-Operationen (*mem.c*)

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

#define MAX 50

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

    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);                      /* ... Speicher wieder freigeben */

    return 0;
}
```

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

Kapitel 10

Kommandozeilen-Argumente

10.1 Parameter der main-Funktion

Bisher haben wir bei der *main-Funktion* keine Parameter gehabt. Tatsächlich bekommt die *main-Funktion* aber mehrere Parameter, nämlich die Anzahl der Kommandozeilen-Argumente und die Kommandozeilen-Argumente selbst jeweils inkl. des Kommando-Namens (und außerdem noch einen Parameter mit den Umgebungsvariablen). *main()* sieht also wie folgt aus:

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

Hier nun die Erklärung der beiden Parameter *argc* und *argv*:

int argc
: Dieser Parameter enthält die Anzahl der Kommandozeilen-Parameter, wobei der Kommando-Name auch dazu gerechnet wird. D. h. im Fall *argc=1* gibt es kein Kommandozeilen-Argument, sondern nur den Kommando-Namen, im Fall *argc=2* gibt es ein Kommandozeilen-Argument, etc.

char *argv[]
: Dieser Parameter ist ein Array von Zeigern auf **char**, also ein Array von Strings. In *argv[0]* ist der Kommando-Name enthalten. In *argv[1]* steht das erste Argument, in *argv[2]* das zweite, usw.

In Abbildung 10.1 ist für die Kommandozeile *a.out 1*, wobei a.out der Kommando-Name ist, dargestellt, wie *argv* etwa im Speicher aussieht. (Dabei sind jedoch Speicherstellen für Bytes/Zeichen und solche für Zeiger der Einfachheit halber nicht unterschiedlich dargestellt!)

10.2 Ausgabe der Kommandozeilen-Argumente

Das folgende Beispiel-Programm illustriert die Verwendung von *argc* und *argv* und gibt einfach den Kommando-Namen, die Anzahl der Kommandozeilen-Argumente und die Kommandozeilen-Argumente selbst aus.

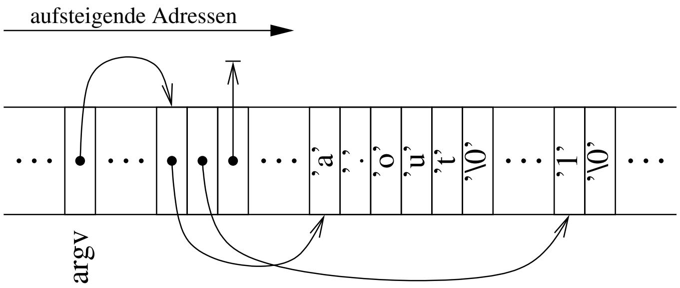


Abbildung 10.1: Repräsentation von argv im Hauptspeicher

Programm 10.1: Ausgabe der Kommandozeilen-Argumente und des Kommando-Namens (*args.c*)

```c
#include <stdio.h>

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

   printf("Kommando-Name: %s\n", argv[0]);

   printf("Anzahl der Argumente: %d\n", argc-1);

   for (i = 1; i < argc; i++)
      printf("Argument %d: %s\n", i, argv[i]);

   return 0;
}
```

```
thales$ gcc -Wall args.c
thales$ a.out
Kommando-Name: a.out
Anzahl der Argumente: 0
thales$ a.out test 1 2
Kommando-Name: a.out
Anzahl der Argumente: 3
Argument 1: test
Argument 2: 1
Argument 3: 2
thales$ gcc -Wall -o args args.c
thales$ args
Kommando-Name: args
Anzahl der Argumente: 0
thales$
```

Anmerkung: Wie man in Abbildung 10.1 schon sieht, ist *argv*[*argc*] der NULL-Zeiger. Somit gibt es also noch eine alternative Möglichkeit, das Ende des Arrays *argv* zu bestimmen, nämlich statt *i* < *argc* könnte man auch prüfen *argv*[*i*] != *NULL*.

10.3 Verarbeiten von Optionen

Ein ganz wichtiges Unix-Kommando ist *grep*, damit kann man Zeilen in Dateien und der Standard-Eingabe suchen, die auf einen regulären Ausdruck „passen“. Wir wollen nun anhand einer vereinfachten grep-Version die Verarbeitung von Optionen auf der Kommandozeile üben. Unser grep erhält als einziges Kommandozeilen-Argument ein Suchmuster (String). Es soll dann alle Zeilen der Standard-Eingabe (*stdin*) ausgeben, die dieses Muster als Teilstring enthalten (analog zum Kommando fgrep). Dies lässt sich wie folgt umsetzen:

Programm 10.2: Ein vereinfachtes grep (*mygrep.c*)

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

#define MAXLEN 256

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

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

        /* bei EOF oder Fehler liefert fgets() NULL! */
    while (fgets(line, MAXLEN, stdin))
        if (strstr(line, argv[1]))
            fputs(line, stdout);

    return 0;
}
```

```
thales$ gcc -Wall mygrep.c
thales$ a.out
Usage: a.out pattern
thales$ cat test.txt
Dies ist
ein kleines Test-Beispiel
für unser eigenes
grep-Kommando.
thales$ a.out ie < test.txt
Dies ist
ein kleines Test-Beispiel
thales$
```

Nun sollen aber Optionen ins Spiel kommen. Wenn wir unserem „Mini-grep“ die Option *-v* übergeben, so soll es alle Zeilen ausgeben, die nicht das Muster enthalten. Die Option *-n* bewirkt, dass in der Ausgabe jeweils die Nummern der Zeilen vorangestellt werden. Die spezielle Option *--* beendet die Suche nach Optionen. Alles was danach kommt wir nicht mehr als Option behandelt. Somit ist es auch möglich, ein Suchmuster der Form „-n“ anzugeben!

Programm 10.3: Ein vereinfachtes grep mit Optionen (*mygrep1.c*)

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

#define MAXLEN 256

int main(int argc, char *argv[]) {
    char line[MAXLEN];
    char *cmdname = argv[0];      /* Kommando-Name */
    char *s;
    int lineno = 0;               /* Zeilennummer in der Eingabe */
    int opt_v = 0;                /* Option -v gesetzt? */
    int opt_n = 0;                /* Option -n gesetzt? */

        /* Optionen verarbeiten */
    while (--argc > 0 && (*++argv)[0] == '-') {
        /* nach "--" kommen keine Optionen mehr (per Definition) */
        if ((*argv)[1] == '-' && (*argv)[2] == '\0') {
            argc--; argv++; break;
        }
        /* steht nur "-", dann fehlt etwas ;-) */
        if ((*argv)[1] == '\0') {
            fprintf(stderr, "%s: empty option\n", cmdname);
            return 1;
        }
        for (s = *argv + 1; *s != '\0'; s++)
            switch (*s) {         /* jede einzelne Option abarbeiten ... */
                case 'v': opt_v = 1; break;
                case 'n': opt_n = 1; break;
                default:          /* unbekannte Option! */
                    fprintf(stderr, "%s: illegal option '%c'\n", cmdname, *s);
                    return 2;
            }
    }

    if (argc != 1) {        /* argc und argv wurden veraendert! */
        fprintf(stderr, "Usage: %s [-nv] pattern\n", cmdname);
        return 3;
    }

    while (fgets(line, MAXLEN, stdin)) {
            /* das Resultat von strstr(), ein Zeiger,
             * wird mittels ! in 0 bzw. 1 umgewandelt => Vergleich erst dann mgl. */
        if (!strstr(line, argv[0]) == opt_v) {
            if (opt_n)
                printf("%d: ", lineno);
            fputs(line, stdout);
        }
        lineno++;
    }
    return 0;
}
```

Beachte: Statt $(*{+}{+}argv)\,[0]$ könnte man auch schreiben $**{+}{+}argv$. (Zur Erinnerung: *argv* ist ein Array von Zeigern auf **char**.) Jedoch sind $(*{+}{+}argv)\,[0]$ und $*{+}{+}argv[0]$ nicht äquivalent, da Letzteres aufgrund der Vorrang-Regeln wie $*{+}{+}(argv\,[0])$ interpretiert wird.

Programm 10.3 kann wie folgt verwendet werden:

```
thales$ gcc -Wall mygrep1.c
thales$ a.out
Usage: a.out [-nv] pattern
thales$ a.out -o ie
a.out: illegal option 'o'
thales$ a.out - ie
a.out: empty option
thales$ a.out ie < test.txt
Dies ist
ein kleines Test-Beispiel
thales$ a.out -v ie < test.txt
für unser eigenes
grep-Kommando.
thales$ a.out -nv ie < test.txt
2: für unser eigenes
3: grep-Kommando.
thales$ a.out -n ie < test.txt
0: Dies ist
1: ein kleines Test-Beispiel
thales$ a.out -- -B < test.txt
ein kleines Test-Beispiel
thales$
```

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

Kapitel 11

Der Preprozessor

Der Preprozessor, der vor der Übersetzung eines C-Programms ausgeführt wird, macht im Wesentlichen folgendes:

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

Die Anweisungen an den Preprozessor nennt man *Preprozessor-Direktiven*. Sie beginnen mit einem # als *erstem Nicht-Whitespace* in einer Zeile. Es gibt eine ganze Reihe von Direktiven, die wir in den folgenden Abschnitten näher kennen lernen werden.

11.1 Einbinden von Dateien

Die Direktive *#include <dateiname>* bzw. *#include "dateiname"* wird durch den Inhalt der angegebenen Datei ersetzt. Die beiden Varianten unterscheiden sich dahingehend, wo der Preprozessor nach der Datei sucht. Bei der ersten Variante findet die Suche an bestimmten Stellen im System (i. A. */usr/include*) und an weiteren angegebenen Stellen (*Option „-I"*) statt. Im anderen Fall wird zunächst im aktuellen Katalog, dann in den System-Verzeichnissen und zuletzt an den Orten, die mit *-I* angegeben sind, gesucht.

Fazit: *#include "dateiname"* verwendet zum Einbinden eigener Header-Dateien; die andere Variante wird für System-/Bibliotheks-Header-Dateien (wie z. B. *stdio.h*) benötigt.

Nähere Informationen – besonders zu *-I-* – gibt es (wie immer) via *man cpp*.

11.2 Makros

11.2.1 Definition und Verwendung von Makros

Mit der Direktive *#define NAME(args,...) body* können *Makros* definiert werden, wobei die Argumente auch fehlen können. Ein *Makro* ist ein String, der an jedem Vorkommen *nach seiner Definition* durch den *Ersatztext* ersetzt wird. Dabei werden jeweils die Argumente in den Ersatztext eingefügt, sofern es Argumente gibt.

Makro-Namen bestehen per Konvention nur aus *Großbuchstaben*. Auf diese Weise kann man sie nicht mit den Namen von Variablen oder Funktionen verwechseln.

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

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

```c
#include <stdio.h>

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

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

    printf("Maximum: %d\n", MAX(a, b));

    return 0;
}
```

```
thales$ gcc -E makros.c
# ... verkuerzte Ausgabe ;-)
int main() {
    int a = 4, b = 3;

    printf("Maximum: %d\n", ((a) > (b) ? (a) : (b)));

    return 0;
}
thales$ gcc -Wall makros.c
thales$ a.out
Maximum: 4
thales$
```

Anmerkungen: Die Argumente von Makros haben keinen Typ. Somit findet bei der Ersetzung keine Typprüfung statt – sehr wohl aber danach durch den Compiler, aber eben auf dem Ergebnis der Ersetzung. Außerdem sind „Makro-Aufrufe“ schneller als Funktionsaufrufe, da der „aufgerufene“ Code ja direkt an die Aufruf-Stelle kopiert wird ($\leadsto$ keine Rekursion möglich).

11.2.2 Fehlerquellen

Es gibt eine ganze Reihe von Fehlerquellen und Stolperfallen bei **#define**. Diese werden an folgendem 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 die leere Anweisung hinzu kommt.

a2) In diesem Fall führt der versehentlich hinzugefügte Strichpunkt aber zu einem Compiler-Fehler, der aber möglicherweise schwer zu finden ist, wenn man sich nicht die Ausgabe des Preprozessors ansieht.

b) Genauso wie bei a2) fällt das „=“ hier durch einen Compiler-Fehler auf, den man aber am besten versteht, wenn man die Ausgabe des Preprozessors kennt.

c) Als Parameter können nicht nur Variablen „übergeben“ werden, sondern beliebiger Text – in diesem Fall ein arithmetischer Ausdruck. Das Ergebnis ist hier, dass die Auswertung in einer Reihenfolge erfolgt wie das bestimmt nicht beabsichtigt war (wegen „Punkt vor Strich“).

d1) Klammert man die Argumente bei ihrer Verwendung, dann hat man in der Regel das beabsichtigte Resultat.

d2) Aber das Klammern der Argumente reicht manchmal noch nicht aus! Hier z. B. hat der Operator `!` höhere Prioriät als der Operator `*`. Daher wird nur der erste Faktor komplementiert, also äquivalent zu `(!(0))*(0)`.

e1) Dem kann man entgehen, wenn man nicht nur die Argumente klammert, sondern auch noch dem kompletten Ausdruck – wie dies bei QUADRIERE der Fall ist.

e2) Hier sieht man, dass trotz Klammerung noch Probleme auftauchen können, da der Parameter hier einen Seiteneffekt hat und mehrmals in einen arithmetischen Ausdruck kopiert wird. Das Resultat dieses Ausdrucks ist bestimmt nicht das beachsichtigte Resultat.

f) Zwischen dem Namen des Makros und der Liste der formalen Parameter darf kein Whitespace stehen, sonst wird die Parameterliste als zum Ersatztext gehörig betrachtet.

11.2.3 Makrodefinition auf der Kommandozeile

Zu Debugging-Zwecken ist es ganz geschickt, dass Makros auch auf der Kommandozeile definiert werden können. Die Option *-D'MAKRONAME=WERT'* ist gleichbedeutend mit einem *#define MAKRONAME WERT* im Quellcode. (Die einfachen Hochkommas bei *-D* kann man sich sparen, wenn in dem String keine Zeichen vorkommen, die von der Shell interpretiert werden. Man sollte sie aber lassen, wenn man sich nicht ganz sicher ist.)

In folgendem Beispiel wird die Option *-D* verwendet, um ggf. Debug-Ausgaben zu produzieren:

Programm 11.2: Debug-Ausgaben mit Makros (*debug.c*)

```c
#include <stdio.h>

int main() {
    DEBUG(puts("Start");)
    puts("... ich arbeite ...");
    DEBUG(puts("Ende");)
    return 0;
}
```

Will man die Debug-Ausgaben nicht einbinden, so definiert man den leeren String als Ersatztext für das Makro:

```
thales$ gcc -E -D'DEBUG(x)=' debug.c
# ... verkuerzte Ausgabe ;-)
int main() {

    puts("... ich arbeite ...");

    return 0;
}
thales$ gcc -Wall -D'DEBUG(x)=' debug.c
thales$ a.out
... ich arbeite ...
thales$
```

Andernfalls ersetzt man einfach das Makro durch den Parameter und schon sind die Debug-Ausgaben da:

```
thales$ gcc -E -D'DEBUG(x)=x' debug.c
# ... verkuerzte Ausgabe ;-)
int main() {
    puts("Start");
    puts("... ich arbeite ...");
    puts("Ende");
    return 0;
}
thales$ gcc -Wall -D'DEBUG(x)=x' debug.c
thales$ a.out
Start
... ich arbeite ...
Ende
thales$
```

11.2.4 Entfernen von Makros

Mit der Direktive *#undef*, die als Parameter den Namen eines Makros erwartet, kann man eine Makrodefinition aufheben. Ab *#undef MAKRO* ist das angegebene Makro (in diesem Fall „MAKRO“) nicht mehr definiert (bis es evtl. neu definiert wird). Folgendes Programm demonstriert die Verwendung von *#undef*:

Programm 11.3: Entfernen von Makros (*undef.c*)

```c
#include <stdio.h>

#define CONST 3

int main() {
    printf("CONST = %d\n", CONST);
#undef CONST
#define CONST 4
    printf("CONST = %d\n", CONST);
    return 0;
}
```

```
thales$ gcc -Wall undef.c
thales$ a.out
CONST = 3
CONST = 4
thales$
```

11.2.5 Vordefinierte Makros

Es gibt einige vordefinierte Makros, wie z. B. *__FILE__*, das durch den Namen der Datei, in der es steht, ersetzt wird, oder *__LINE__*, das durch die Zeilennummer, in der es steht, ersetzt wird. Das folgende Programm demonstriert eine sinnvolle Verwendung dieser Makros zur Überprüfung von Zusicherungen:

Programm 11.4: Zusicherungen überprüfen (*assert.c*)

```c
#include <stdio.h>

#define ASSERT(cond) \
            if (!(cond)) \
                fprintf(stderr, "%s (%d): assertion \"%s\" failed\n", \
                    __FILE__, __LINE__, #cond);

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

```
thales$ gcc -E assert.c
/* gekuerzte Ausgabe mit manuellem Umbruch (unten) ;-) */
int main() {
    int a = 3;
    if (!(a < 0))
        fprintf((&__iob[2]), "%s (%d): assertion \"%s\" failed\n",
                    "assert.c", 10, "a < 0");
    return 0;
}
thales$ gcc -Wall assert.c
thales$ a.out
assert.c (10): assertion "a < 0" failed
thales$
```

Anmerkungen: Bei obigem Programm sieht man außerdem eine weitere Verwendung des „Backslashs“. Steht dieser nämlich direkt vor dem Zeilenumbruch, so wird der *Backslash samt Zeilenumbruch und darauf folgenden Whitespace* durch den *leeren String ersetzt*. Auf diese Weise kann es Makrodefinitionen geben, die sich über mehrere Zeilen erstrecken. Des Weiteren kann man bei der Definition eines Makros einem Argument des Makros auch das Zeichen *# voranstellen*. Dann wird der eingefügte Text noch mit Anführungszeichen quotiert (siehe Programm 11.4).

11.3 Bedingte Übersetzung

11.3.1 Test auf Makro-Existenz

Mit den Direktiven *#ifdef MAKRONAME* bzw. *#ifndef MAKRONAME* wird ein Block begonnen, welcher von der Direktive *#endif* (oder *#else* etc.) beendet wird. Dieser Block wird nur unter der Bedingung übersetzt, dass ein Makro mit dem angegebenen Namen definiert bzw. nicht definiert ist. (Genauer müsste man eigentlich sagen, dass der Teil nicht unter der Bedingung übersetzt wird, sondern unter der Bedingung nicht durch den leeren String ersetzt wird und somit zur Übersetzung bleibt.)

Mit diesen Direktiven kann man nun das Debugging-Beispiel noch etwas einfacher gestalten:

Programm 11.5: Debug-Ausgaben mit Makros – Zweite Variante (*debug1.c*)

```
#include <stdio.h>

#ifdef DBG
   # define DEBUG(x) x
#else
   # define DEBUG(x)
#endif

int main() {
   DEBUG(puts("Start");)
   puts(" ... ich arbeite ... ");
   DEBUG(puts("Ende");)
   return 0;
}
```

Will man die Debug-Ausgaben nicht einbinden, so muss man nichts tun, denn, wenn das Makro *DBG* (auf der Kommandozeile) nicht definiert wurde, dann wird es so definiert, dass das *DEBUG*-Makro durch nichts ersetzt wird:

```
thales$ gcc -E debug1.c
# ... verkuerzte Ausgabe ;-)
int main() {

    puts("... ich arbeite ...");

    return 0;
}
thales$ gcc -Wall debug1.c
thales$ a.out
... ich arbeite ...
thales$
```

Definiert man auf der Kommandozeile jedoch das Makro *DBG*, so wird *DEBUG* so definiert, dass das Makro durch das Argument ersetzt wird:

```
thales$ gcc -E -DDBG debug1.c
# ... verkuerzte Ausgabe ;-)
int main() {
    puts("Start");
    puts("... ich arbeite ...");
    puts("Ende");
    return 0;
}
thales$ gcc -Wall -DDBG debug1.c
thales$ a.out
Start
... ich arbeite ...
Ende
thales$
```

Anmerkung: Zwischen *#ifdef* bzw. *#ifndef* und *#endif* kann man die Direktive *#else* und *#elif* verwenden, um einen Block für den alternativen Fall zu beginnen (siehe Programm 11.5).

11.3.2 Weitere Tests

Bisher wurde nur bezüglich Existenz bzw. Nicht-Existenz von Makros unterschieden. Es können jedoch auch die Werte von Makros zur bedingten Übersetzung herangezogen werden. Hierzu existieren die Direktiven *#if BEDINGUNG*, *#elif BEGINGUNG*, *#else* und *#endif*, wobei *#elif* beliebig oft (auch gar nicht) und *#else* optional vorkommen kann. Die Bedingung muss ein konstanter Ausdruck sein, wobei evtl. vorkommende Makros vorher ersetzt werden.

Das folgende Programm illustriert die Verwendung der beschriebenen Direktiven:

Programm 11.6: Bedingte Übersetzung (*if.c*)

```c
#include <stdio.h>

int main() {
#if x == 1
    puts("1");
#elif x == 2
    puts("2");
#else
    puts("anderer Wert oder nicht definiert");
#endif
    return 0;
}
```

```
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* unterscheidet man zwischen der *Deklaration* und der *Definition*.

Deklaration: Macht die Variable oder Funktion *bekannt* und legt deren *Typ* fest.

Definition: Zur Deklaration kommt hier noch hinzu, dass auch *Speicherplatz* beschafft wird (im Fall von Variablen) bzw. die *Implementierung* erfolgt (im Fall von Funktionen).

Anmerkung: *Initialisierung* geht nur bei der *Definition*!

Durch Voranstellen des Schlüsselwortes **extern** kann man globale Variablen und Funktionen deklarieren. Dadurch wird dem Übersetzer (Compiler) signalisiert, dass es an anderer Stelle im Programm eine entsprechende Definition gibt. Ist dies nicht der Fall, so führt dies zu einem Fehler beim Binden („Linken“) des Programms. Das folgende Programm enthält einige Beispiele von Deklarationen und Definitionen:

Programm 12.1: Deklaration vs. Definition (*defs.c*)

```
extern void print();       /* Deklaration */

void work() {              /* Definition */
   /* ... */
   print();
   /* ... */
}

extern int count;          /* Deklaration */

void print() {             /* Definition */
   /* ... */
   count++;
   /* ... */
}
```

```
int count;              /* Definition */

int main() {            /* Definition */
    int i;              /* Definition */
    /* ... */
    return 0;
}
```

Die Deklarationen wirken in obigem Beispiel eher sehr künstlich erzeugt. Hier könnte man die Deklarationen durch Umstellen der Definitionen vermeiden. Jedoch geht das nicht mehr, wenn der Quellcode über mehrere Module verteilt ist, denn eine Variable darf nur *einmal definiert*, aber *beliebig oft deklariert* werden. (*Initialisieren* kann man eine Variable natürlich nur bei ihrer Definition, da es nur dann auch einen Speicherplatz gibt, in den man diesen Wert schreiben kann!)

Hier nun ein paar Beispiele zur Verwendung von Deklarationen bei der Verteilung eines Programms auf mehrere Quelltext-Dateien. Wenn die verwendeten Variablen oder Funktionen, die in einer anderen Datei definiert sind, gar nicht deklariert sind, dann kommt es zu einer Fehlermeldung des Übersetzers:

```
thales$ cat lib.c
float c = 4.5;
thales$ cat test1.c
#include <stdio.h>
/* Deklaration von c fehlt komplett */
int main() {
    printf("c = %f\n", c);
    return 0;
}
thales$ gcc -Wall test1.c lib.c
test1.c: In function `main':
test1.c:4: `c' undeclared (first use in this function)
test1.c:4: (Each undeclared identifier is reported only once
test1.c:4: for each function it appears in.)
thales$
```

Deklariert man die Variable zwar, aber vergisst den Typ dabei anzugeben, so geht der Übersetzer von **int** aus. In diesem Fall ist dies fatal:

```
thales$ cat test2.c
#include <stdio.h>
extern c; /* Deklaration (als int :-( -- implizit) */
int main() {
    printf("c = %f\n", c);
    return 0;
}
thales$ gcc -Wall test2.c lib.c
test2.c:2: warning: type defaults to ‘int’ in declaration of ‘c’
test2.c: In function ‘main’:
test2.c:4: warning: double format, different type arg (arg 2)
thales$ a.out
c = 1024.000974
thales$
```

Richtig gemacht wird es wie folgt, wo sowohl das Schlüsselwort **extern** zur Deklaration verwendet also auch der Typ der Variablen angegeben wird:

```
thales$ cat test3.c
#include <stdio.h>
extern float c; /* Deklaration (als float :-) */
int main() {
    printf("c = %f\n", c);
    return 0;
}
thales$ gcc -Wall test3.c lib.c
thales$ a.out
c = 4.500000
thales$
```

12.2 Die Speicherklasse static

12.2.1 Lokale Variablen

Normalerweise gehören lokale Variablen, d. h. Variablen, die innerhalb einer Funktion definiert werden, zur *Speicherklasse auto*. Somit ist ihre Lebensdauer auf die momentane Ausführung der Funktion beschränkt. Bei jedem Aufruf wird von Neuem für diese Variablen Speicherplatz alloziert, der dann nach Beendigung dieser Funktion automatisch wieder freigegeben wird.

Stellt man einer lokalen Variable jedoch das Schlüsselwort **static** voran, so gehört sie zur Speicherklasse *static*. In diesem Fall wird genau einmal für diese Variable Speicherplatz alloziert. Dieser Speicherplatz wird dann erst beim Ende des Programms wieder frei gegeben. Die *Initialisierung* einer lokalen Variable der Speicherklasse static findet *genau einmal* statt (und nicht bei jedem Funktionsaufruf erneut).

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

Programm 12.2: Speicherklasse static für lokale Variablen (*static.c*)

```
#include <stdio.h>

void work() {
    static int lokal = 0;
    printf("lokal = %d\n", lokal++);
}

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

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

12.2.2 Globale Variablen und Funktionen

Globalen Variablen und Funktionen kann man ebenfalls das Schlüsselwort **static** voranstellen. Dies bedeutet, dass der *Gültigkeitsbereich dieser Variablen und Funktionen auf die Datei, in der sie enthalten sind, beschränkt* ist. Von anderen Dateien aus können sie nicht verwendet werden – auch nicht nach vorheriger Deklaration! 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 (Linken) tritt ein Fehler auf*, da die deklarierte Variable bzw. Funktion nicht gefunden wird.

12.3 Module

12.3.1 Modularisierung – Warum?

Sobald Programme größer werden, können sie nicht mehr übersichtlich in einer *einzigen Datei* untergebracht werden. Deshalb empfiehlt sich die *Aufteilung in einzelne Module*. Dabei kann man nach verschiedenen *Kriterien* modularisieren – aus der *Datensicht* oder aus der *funktionalen Sichtweise*. Außerdem empfiehlt es sich, Implementierungsdetails eines Moduls nach außen hin unsichtbar zu halten (Prinzip des *Information Hiding*). Dann kann eine Implementierung geändert werden, ohne dass dies Auswirkungen auf deren Klienten hat. Dies hat auch weitere Vorteile. Eine bessere sog. *Kapselung* erleichtert auch das *Finden von Fehlern*. Schließlich lassen sich einzelne Module evtl. *wieder verwenden*, wohingegen dies bei einem ganzen Programm nicht direkt möglich ist. Es gibt also eine Reihe guter Gründe, zu modularisieren.

12.3.2 Modularisierung in Modula-2

Im Folgenden wird ein kleines Modula-2-Beispiel für Modularisierung vorgestellt. Das Modul *Datum* bietet einen Datentyp zur Respräsentation eines Datums und zwei Funktionen zum Arbeiten mit einem Datum an. (Der Datentyp ist hier aber noch nicht bestmöglich gekapselt! ;-)

Dieses Beispiel wird dann im folgenden Abschnitt verwendet, um Modularisierung in C zu erklären.

Das Programm ist wie folgt strukturiert: Es gibt das Hauptprogramm (*Main.m2*), das „Definitions-Modul“ (*Datum.d*) und die zugehörige Implemenierung (*Datum.m2*). Das Hauptprogramm *Main.m2* importiert und verwendet Funktionen, Variablen und Konstanten vom Modul *Datum*.

Programm 12.3: Öffentliche Definitionen des Datums-Moduls (in Modula-2) (*Datum.d*)

```
DEFINITION MODULE Datum;

    (*** Konstanten ***)
    CONST
        JAN = 1;
        FEB = 2;
        (*  ...  *)

    (*** Datentypen ***)
    TYPE
        Datum = RECORD
                    tag , monat, jahr : INTEGER;
                END;

    (*** Variablen ( Definitionen ) ***)
    VAR
        heute : Datum;

    (*** Funktionen ( Deklarationen ) ***)
    PROCEDURE print(d : Datum);
    PROCEDURE istSchaltjahr(jahr : INTEGER): BOOLEAN;

END Datum.
```

Programm 12.4: Implementierung des Datums-Moduls (in Modula-2) (*Datum.m2*)

```
IMPLEMENTATION MODULE Datum;

    FROM InOut IMPORT WriteInt, Write, WriteLn;

    (*-- PRIVAT (nur zur Implementierung) ----------*)

    CONST
        c1 = 4;
        c2 = 100;
        c3 = 400;

    PROCEDURE teilt(b, a : INTEGER): BOOLEAN;
    BEGIN
        RETURN a MOD b = 0
    END teilt ;

    (*-- OEFFENTLICH ----------------------------------*)

    (*** Funktionen ( Definitionen ) ***)
```

```
    PROCEDURE print(d : Datum);
    BEGIN
        WriteInt (d.tag , 1);
        Write(".");
        WriteInt (d.monat, 1);
        Write(".");
        WriteInt (d.jahr , 1);
        WriteLn
    END print;

    PROCEDURE istSchaltjahr(jahr : INTEGER): BOOLEAN;
    BEGIN
        RETURN teilt(c1, jahr) & (~ teilt (c2, jahr) OR teilt (c3, jahr))
    END istSchaltjahr ;

BEGIN
    (*** Initialisierung der Variablen ***)
    heute.tag := 24;
    heute.monat := 12;
    heute.jahr := 2003;
END Datum.
```

Programm 12.5: Hauptprogramm (in Modula-2) (*Main.m2*)

```
MODULE Main;

    FROM InOut IMPORT WriteString, WriteLn;
    FROM Datum IMPORT Datum, JAN, heute, print, istSchaltjahr ;

    VAR
        d : Datum;

BEGIN
    d.tag := 1;
    d.monat := JAN;
    d.jahr := 2004;

    print (d);
    print (heute);
    IF ( istSchaltjahr (heute.jahr)) THEN
        WriteString ("ja")
    ELSE
        WriteString ("nein")
    END;
    WriteLn
END Main.
```

In Modula-2 bietet die Sprache schon die Möglichkeit zur Modularisierung. Die Definitionen und Deklarationen, welche im „Definitions-Modul“ stehen, sind nach außen bekannt, alle anderen (der Implementierung) sind privat.

12.3.3 Modularisierung in C

Die Programmiersprache C hat kein eigenes Konzept, was Modularisierung angeht. Daher muss man diese über den Präprozessor realisieren. Im Folgenden diskutieren wir die Modularisierung in C am Beispiel aus dem letzten Abschnitt. Grundsätzlich erfolgt die Aufteilung analog in die Dateien *main.c*, *datum.h* und *datum.c*. Da es jedoch nicht die Möglichkeit eines Imports gibt, muss die Datei *datum.h* mittels **#include** ... sowohl in *datum.c*, also auch in *main.c* eingebunden (genauer: *hereinkopiert*) werden. Dies hat zur Folge, dass in *datum.h* nur Deklarationen von Variablen stehen dürfen, da sie sonst *doppelt definiert* würden, was nicht erlaubt ist. Damit die Header-Datei *datum.h* nicht mehrmals hereinkopiert werden kann, verwendet man üblicherweise die „**#ifndef** -**#define**-**#endif**-Konstruktion". Außerdem sollte man die *nicht-öffentlichen* Variablen und Funktionen in *datum.c* mittels **static** eigens schützen.

Programm 12.6: Öffentliche Definitionen des Datums-Moduls (in C) (*datum.h*)

```
#ifndef DATUM_H
#define DATUM_H /*** nur _einmal_ einbinden! ***/

/*** Konstanten ***/
#define JAN 1
#define FEB 2
/* ... */

/*** Datentypen ***/
typedef struct {
            short int tag, monat, jahr;
        } Datum;

/*** Variablen ( Deklarationen ) ***/
extern Datum heute;

/*** Funktionen ( Deklarationen ) ***/
extern void print(Datum);
extern int istSchaltjahr(short);

#endif
```

Programm 12.7: Implementierung des Datums-Moduls (in C) (*datum.c*)

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

/*-- PRIVAT (nur zur Implementierung) -----------*/

static const int c1 = 4;
static const int c2 = 100;
static const int c3 = 400;

static int teilt(int b, int a) { /* teilt b die Zahl a? */
    return a % b == 0;
}

/*-- OEFFENTLICH ---------------------------------*/
```

```
/*** Variablen ( Definitionen ) ***/
Datum heute = {6, 12, 2005};      /* sehr  simplifiziert  */

/*** Funktionen ( Definitionen ) ***/
void print (Datum d) {
    printf ("%d.%d.%d\n", d.tag, d.monat, d.jahr);
}

int istSchaltjahr (short jahr) {
   return teilt (c1, jahr) && (! teilt (c2, jahr) || teilt (c3, jahr));
}
```

Programm 12.8: Hauptprogramm (in C) (*main.c*)

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

int main() {
   Datum d = {1, JAN, 2006};

   print (d);
   print (heute);
   puts( istSchaltjahr (heute.jahr) ? "ja" : "nein");

   return 0;
}
```

```
thales$ gcc -Wall main.c datum.c
thales$ a.out
1.1.2006
6.12.2005
nein
thales$
```

12.4 Abstrakte Datentypen

12.4.1 Allgemein

Zu einem *abstrakten Datentyp* (*ADT*) gehören

- *Daten*,
- *Operationen* (d. h. Funktionen bzw. Prozeduren) auf diesen Daten und
- eine *semantische Beschreibung der Operationen* (oft in Prosa, manchmal über *Axiome*).

Dieses Konzept ist noch *unabhängig von der Programmiersprache*. Wollen wir einen ADT in C implementieren, so verwenden wir oft einen Struktur-Typ für die Daten. Von diesem Typ ist aber nur der Name öffentlich bekannt. *Die Implementierung der Funktionen und die Elemente des Struktur-Typs sind nicht öffentlich bekannt*. Somit hängen Klienten nicht von der Implementierung oder den „Typ-Details“ ab. Das *Prinzip des Information Hiding* ist somit – im Gegensatz zum vorigen Beispiel – voll umgesetzt.

12.4.2 Komplexe Zahlen – Ein erstes Beispiel

Als erstes Beispiel für einen abstrakten Datentyp betrachten wir den Typ *Complex* zur Repräsentation von komplexen Zahlen.

Programm 12.9: Header-Datei des Moduls für komplexe Zahlen (*complex.h*)

```c
#ifndef COMPLEX_H
#define COMPLEX_H

    /* Abstrakter Datentyp */
typedef struct _complex *Complex;

typedef double FLOAT;

extern void newComplex(Complex *c, FLOAT re, FLOAT im);
extern void deleteComplex (Complex *c);
extern FLOAT real(Complex c);
extern FLOAT imag(Complex c);
extern Complex add(Complex a, Complex b);
extern void print (Complex c);

#endif
```

Anmerkung: *Complex* ist ein Zeiger auf einen Struktur-Typ, der hier noch gar nicht bekannt ist (und auch nicht bekannt sein soll). Dies ist jedoch – wie wir bei Strukturen gesehen haben – kein Problem.

Programm 12.10: Implementierung des Moduls für komplexe Zahlen (*complex.c*)

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

struct _complex {
    FLOAT _re, _im;
};

void newComplex(Complex *c, FLOAT re, FLOAT im) {
    *c = (Complex) calloc (1, sizeof(struct _complex));
    if (*c != NULL) {   /* bei Erfolg ... ;-) */
        (*c)->_re = re;
        (*c)->_im = im;
    }
}

void deleteComplex (Complex *c) {
    free (*c);
    *c = NULL;
}

FLOAT real(Complex c) {
    return c->_re;
}
```

```
FLOAT imag(Complex c) {
    return c->_im;
}

Complex add(Complex a, Complex b) {
    Complex tmp;
    newComplex(&tmp, a->_re, a->_im);
    if (tmp != NULL) {
        tmp->_re += b->_re;
        tmp->_im += b->_im;
    }
    return tmp; /* nur Zeiger ist lokal! Objekt auf dem Heap! */
}

void print(Complex c) {
    printf("%f + i * %f\n", c->_re, c->_im);
}
```

Programm 12.11: Arbeiten mit komplexen Zahlen (*mainComplex.c*)

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

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

    newComplex(&a, 3, 4);
    newComplex(&b, -1, 2);

    if (a == NULL || b == NULL) {
        fprintf(stderr, "not enough memory\n");
        return 1;
    }

    print(a);
    print(b);

    c = add(a, b);

    if (c == NULL) {
        fprintf(stderr, "not enough memory\n");
        return 1;
    }

    print(c);

    deleteComplex(&a);
    deleteComplex(&b);
    deleteComplex(&c);

    return 0;
}
```

```
thales$ gcc -Wall mainComplex.c complex.c
thales$ a.out
3.000000 + i * 4.000000
-1.000000 + i * 2.000000
2.000000 + i * 6.000000
thales$
```

12.4.3 Behälterdatenstrukturen – Ein Stack

Nun folgt noch ein etwas komplizierterer ADT, nämlich ein *Stack*. Typische Operationen eines Stacks sind *push()*, *pop()* und auch *empty()* (bzw. oft auch *isEmpty()*). Im Folgenden werden wir zunächst ein kleines Beispiel für die axiomatische Beschreibung eines Stack kennen lernen. Sei E die Menge der Elemente des Stacks und S sie Menge aller möglichen Stacks mit den Elementen E. Für die Operationen

- $new : \emptyset \to S$
  Anlegen eines leeren Stacks
- $push : S \times E \to S$
  Element oben auf den Stack legen
- $pop : S \to S$
  Oberstes Element vom Stack entfernen
- $top : S \to E$
  Liefert oberstes Stack-Element, ohne es zu entfernen
- $empty : S \to \{true, false\}$
  Liefert true genau dann, wenn der Stack leer ist

kann man nun (u. a.) folgende Axiome angeben

- $empty(new)$
- $NOT\ empty(push(s,e))$
- $top(push(s,e)) = e$
- $pop(push(s,e)) = s$

wobei $s \in S$ und $e \in E$ seien. In der nun folgenden Implementierung verwenden wir den Namen *isEmpty()* statt *empty()*. Außerdem ist *pop()* in unserer Implementierung eine Kombination aus *pop()* und *top()*.

Da der Stack in unserer Implementierung *Elemente von beliebigem Typ* aufnehmen soll, verwenden wir als Element einen *void-Zeiger*. Dieser kann in jeden andern Zeiger umgewandelt werden und umgekehrt. Das Kopieren eines übergebenen Elementes kann aber, da wir nichts über den Elementtyp wissen können, nicht von der Stack-Implementierung selbst erfolgen. Daher verwenden wird eine *Kopierfunktion* vom Typ *COPYFUN*, die ein Element (Zeiger) übergeben bekommt, dynamisch Speicher alloziert für eine Kopie des Elements, die Kopie anfertigt und dann einen Zeiger auf die Kopie zurück gibt.

Programm 12.12: Header-Datei des Stack-Moduls (*stack.h*)

```
#ifndef STACK_H
#define STACK_H

    /* abstrakter Datentyp */
typedef struct _stack *Stack;

    /* Stack-Elemente */
typedef void *ELEMENT;

typedef int BOOL;

    /* Kopierfunktion fuer die Stack-Elemente */
typedef ELEMENT (*COPYFUN)(ELEMENT);

extern void newStack(Stack *s, COPYFUN cp);
extern void deleteStack (Stack *s);
extern BOOL isEmpty(Stack s);
extern ELEMENT pop(Stack s);
extern void push(Stack s, ELEMENT elem);

#endif
```

Programm 12.13: Implementierung des Stack-Moduls (*stack.c*)

```
#include "stack.h"
#include <stdlib.h>

struct _stack {
    COPYFUN _copy;
    struct _stackElem *_top;
};

struct _stackElem {
    ELEMENT _elem;
    struct _stackElem *_next;
};

void newStack(Stack *s, COPYFUN cp) {
    *s = (Stack) calloc (1, sizeof(struct _stack));
    if (*s != NULL) {
        (*s)->_copy = cp;
        (*s)->_top = NULL;
    }
}

void deleteStack (Stack *s) {
    if (isEmpty(*s)) {
        free (*s);
        *s = NULL;
    }
}

BOOL isEmpty(Stack s) {
```

```
    return s->_top == NULL;
}

ELEMENT pop(Stack s) {
    ELEMENT elem = NULL;
    if (!isEmpty(s)) {
        struct _stackElem *st_elem = s->_top;
        elem = st_elem->_elem;
        s->_top = st_elem->_next;
        free(st_elem);
    }
    return elem;
}

void push(Stack s, ELEMENT elem) {
    struct _stackElem *st_elem;
    st_elem = (struct _stackElem*) calloc(1, sizeof(struct _stackElem));
    if (st_elem != NULL) {
        st_elem->_elem = s->_copy(elem);
        st_elem->_next = s->_top;
        s->_top = st_elem;
    }
}
```

Programm 12.14: Arbeiten mit String-Stacks (*mainStack.c*)

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

#define MAXLEN 80

int main() {
    char line[MAXLEN];
    Stack s;

        /*
         * char *strdup(char *)
         * dupliziert den uebergebenen String
         * und liefert einen Zeiger auf die Kopie.
         */
    newStack(&s, (COPYFUN) strdup);
    if (s == NULL) {
        fprintf(stderr, "not enough memory\n");
        return 1;
    }

        /*
         * Zwei unmittelbar aufeinanderfolgende,
         * die hoechstens durch Whitespace voneinander
         * getrennt sind, werden zu einer String-Konstanten!
         */
    puts("String-Stack\n"
```

```
        "------------\n"
        "Moegliche Kommandos:\n"
        " +elem  push\n"
        " -       pop\n"
        " ?       isEmpty\n"
        " EOF    ENDE\n");

    while (printf("> "), fgets(line, MAXLEN, stdin)) {
        int len = strlen(line);
        if (line[len-1] == '\n')          /* '\n' am Zeilenende entfernen */
            line[len-1] = '\0';
        switch (line[0]) {
            case '+':
                push(s, line+1);
                printf("push: elem = \"%s\"\n", line+1); /* oder &line[1] */
                break;
            case '-': {
                ELEMENT elem = pop(s);
                if (elem != NULL)
                    printf("pop: elem = \"%s\"\n", (char*) elem);
                else
                    puts("pop failed");
                break;
                }
            case '?':
                puts(isEmpty(s) ? "stack is empty" : "stack is  not  empty");
                break;
            default:
                puts("expected [+-?] as first character (or EOF)");
                break;
        }
    }

    puts("");

    while (!isEmpty(s))
        printf("pop: elem = \"%s\"\n", (char*) pop(s));

    deleteStack(&s);

    return 0;
}
```

```
thales$ gcc -Wall mainStack.c stack.c
thales$ a.out
String-Stack
------------
Moegliche Kommandos:
+elem  push
-      pop
?      isEmpty
EOF    ENDE

> +1
push: elem = "1"
> +2
push: elem = "2"
> -
pop: elem = "2"
> +3
push: elem = "3"
> ^D
pop: elem = "3"
pop: elem = "1"
thales$
```

Kapitel 13

Der Übersetzungs-Vorgang

13.1 Von der Quelle zum ausführbaren Programm

Den ersten Schritt auf dem Weg vom Quellcode zum ausführbaren Programm (siehe Abbildung 13.1) kennen wir bereits, den *Präprozessor*. Die Ausgabe des Präprozessors wird danach

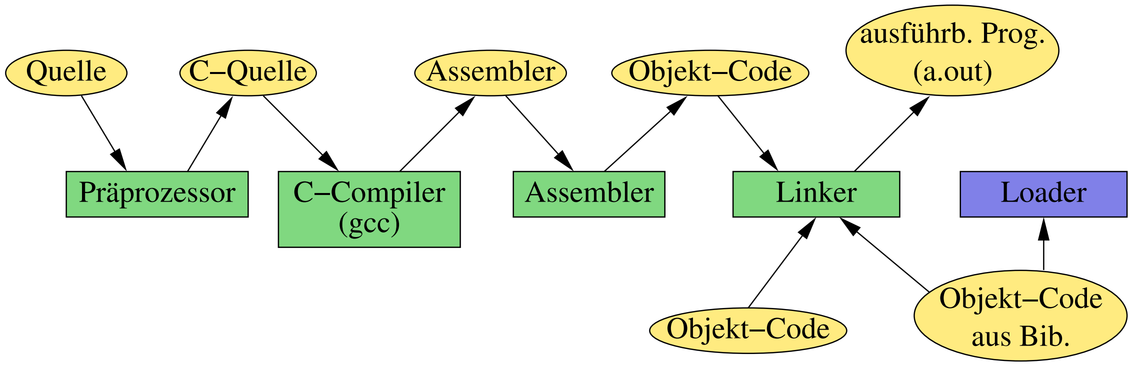


Abbildung 13.1: Der Weg von der Quelle zum ausführbaren Programm

vom *Übersetzer* (*Compiler*) verarbeitet. Dieser überprüft die syntaktische Korrektheit und auch die semantische Korrektheit (z. B. verwendete Variablen auch deklariert?) des Programms. Schließlich erzeugt er Assembler-Quelltext. Diese Ausgabe des GNU-C-Compilers kann man mit der Kommandozeilen-Option *-S* eigens erzeugen. (Normalerweise gibt es den Assembler-Quelltext nur temporär.)

Programm 13.1: Kleines Beispielprogramm (*hallo.c*)

```c
#include <stdio.h>

int main() {
    puts("Hallo Welt!");
    return 0;
}
```

```
thales$ gcc -S hallo.c
thales$ cat hallo.s
        .file   "hallo.c"
        .section        ".rodata"
        .align 8
.LLC0:
        .asciz  "Hallo Welt!"
        .section        ".text"
        .align 4
        .global main
        .type   main,#function
        .proc   04
main:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
        sethi   %hi(.LLC0), %o0
        or      %o0, %lo(.LLC0), %o0
        call    puts, 0
         nop
        mov     0, %o0
        mov     %o0, %i0
        nop
        ret
        restore
.LLfe1:
        .size   main,.LLfe1-main
        .ident  "GCC: (GNU) 3.2"
thales$
```

Der *Assembler* erzeugt nun aus dem Assembler-Quelltext den *Objekt-Code* (in *Maschinensprache*). Der *Linker* und der *Loader* kombinieren verschiedene Objekt-Codes (des Programms – evtl. mehrere Objekt-Codes! – und aus der Bibliothek), löst Referenzen zwischen verschiedenen Objekt-Codes auf und verschiebt die Daten. Das Ergebnis ist das *ausführbare Programm*.

13.2 Bibliotheken

Früher war es gebräuchlich, alle benötigten Objekt-Codes zu der ausführbaren Datei beim Kompilieren hinzuzubinden. Dies hatte – besonders als die Zahl und Größe der Bibliotheken zunahm – eine Reihe von Nachteilen:

- Die ausführbaren Dateien werden riesig groß, da sie den Objekt-Code der benötigten Bibliotheksfunktionen auch enthalten.
- Die Programme müssen neu kompiliert werden, wenn sich die Bibliotheken ändern und diese Änderungen auch im Programm wirksam sein sollen.
- Plugin-Systeme mit dynamisch hinzukommenden Modulen sind gar nicht möglich.

Aus diesen Gründen hat man sich vom *a.out-Format* „verabschiedet" und das *ELF-Format* (Executable and Linking Format) eingeführt. Heutzutage wird alles dynamisch gebunden (statische Bindung findet nur noch auf explizites Verlangen hin statt).

13.2.1 Verwendung dynamischer Bibliotheken

Wie bereits erwähnt, fügt der Linker/Loader evtl. auch Objekt-Code aus Bibliotheken hinzu. Dynamische Bibliotheken (*shared objects*), welche die Endung „.so“ besitzen (die für „shared object“ steht), enthalten Objekt-Code, der selektiv vom Loader extrahiert werden kann. Der Loader bindet nur den Teil der Bibliothek hinzu, der unbedingt nötig ist, um noch offene Referenzen aufzulösen – und dieses Hinzubinden geschieht erst kurz vor der Ausführung. Der Linke überprüft aber zur Übersetzungszeit, ob die entsprechenden Funktionen existieren (auch mit den entsprechenden Parameter-Typen etc.). Zur Übersetzungszeit müssen also lediglich der *Name der Bibliothek* und die *Schnittstelle der Bibliothek* (d. h. alle öffentlichen Funktionen und Variablen) bekannt sein. Die Mathe-Bibliothek `/usr/lib/libm.so`, die alle wichtigen mathematischen Funktionen enthält, wird standardmäßig nicht „hinzugelinkt“. Dies kann man aber mit Hilfe der Kommandozeilen-Option *-l*, in diesem Fall also *-l**m***, erreichen, wobei das „m“ hierbei aus dem Namen der dynamischen Bibliothek (lib**m**.so) stammt.

Programm 13.2: Verwendung mathematischer Funktionen (*math.c*)

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

int main() {
    double x = 3.1415;

    printf("sin(%f) = %f\n", x, sin(x));
    printf("arcsin(1) = %f\n", asin(1));

    return 0;
}
```

```
thales$ gcc -Wall math.c
/tmp/cca8LHmF.o: In function `main':
/tmp/cca8LHmF.o(.text+0x1c): undefined reference to `sin'
/tmp/cca8LHmF.o(.text+0x64): undefined reference to `asin'
collect2: ld returned 1 exit status
thales$ gcc -Wall math.c -lm
thales$ a.out
sin(3.141500) = 0.000093
arcsin(1) = 1.570796
thales$
```

13.2.2 Wer verwendet was?

Mit dem *Kommando ldd* kann man herausfinden, welche dynamischen Bibliotheken verwendet werden:

```
thales$ gcc -Wall math.c -lm
thales$ a.out
sin(3.141500) = 0.000093
arcsin(1) = 1.570796
thales$ ldd a.out
        libm.so.1 =>     /usr/lib/libm.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        /usr/platform/SUNW,Sun-Fire-V240/lib/libc_psr.so.1
thales$
```

13.2.3 Eigene dynamische Bibliotheken

Will man eine eigene dynamische Bibliothek erzeugen, so kann man die Option *-shared* von *gcc* verwenden. Dies wollen wir an einem kleinen Beispiel ausprobieren. Der Objekt-Code *mylib.o* soll in eine eigene Bibliothek kommen und diese wollen wir dann zu *main.o* „dazulinken".

Programm 13.3: Header-Datei des kleinen Moduls (*mylib.h*)

```c
#ifndef MYLIB_H
#define MYLIB_H

extern void hallo(char *vorname, char *name);

#endif
```

Programm 13.4: Implementierung des kleinen Moduls (*mylib.c*)

```c
#include <stdio.h>

void hallo(char *vorname, char *name) {
    printf("Hallo %s %s!\n", vorname, name);
}
```

Programm 13.5: Kleines Hauptprogramm (*main.c*)

```c
#include "mylib.h"

int main() {
    hallo("Johannes", "Mayer");
    return 0;
}
```

```
thales$ ls
main.c  mylib.c  mylib.h
thales$ gcc -Wall -shared -o libown.so mylib.c
thales$ ls
libown.so  main.c  mylib.c  mylib.h
thales$ gcc -Wall main.c
/tmp/ccGcWFuW.o: In function `main':
/tmp/ccGcWFuW.o(.text+0x14): undefined reference to `hallo'
collect2: ld returned 1 exit status
thales$ gcc -Wall main.c -L . -lown
thales$ a.out
ld.so.1: a.out: fatal: libown.so: open failed: No such file or directory
Killed
thales$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
thales$ a.out
Hallo Johannes Mayer!
thales$
```

Erklärung: Will man eine dynamische Bibliothek (shared object) erzeugen, so muss man dies dem gcc mit der Option *-shared* mitteilen. Die eigene Bibliothek wird zunächst zur Laufzeit nicht gefunden, da sie sich nicht in einem der Verzeichnisse befindet, die in der *Umgebungsvariable LD_LIBRARY_PATH* stehen, und auch nicht in den Verzeichnissen */usr/lib* und */lib* zu finden ist. Dieses Problem lässt sich leicht beheben, indem wir das aktuelle Verzeichnis („.“) an die Umgebungsvariable LD_LIBRARY_PATH anhängen. (Die einzelnen Verzeichnisse sind hierbei durch „:“ voneinander getrennt.)
Mit der Option *-l**own*** wird die eigene Bibliothek (lib**own**.so) „dazugelinkt“, wobei diese Bibliothek per default nicht im aktuellen Verzeichnis gesucht wird. Mit *-L .* erreicht man schließlich, dass der aktuellen Katalog auch in die Suche mit einbezogen wird.

Dass die eigene Bibliothek tatsächlich verwendet wird, sieht man an folgendem Beispiel:

```
thales$ ls
a.out  libown.so  main.c  mylib.c  mylib.h
thales$ ldd a.out
        libown.so =>     ./libown.so
        libc.so.1 =>     /usr/lib/libc.so.1
        libgcc_s.so.1 =>         /usr/local/lib/libgcc_s.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        /usr/platform/SUNW,Sun-Fire-V240/lib/libc_psr.so.1
thales$ rm libown.so
rm: remove `libown.so'? y
thales$ a.out
ld.so.1: a.out: fatal: libown.so: open failed: No such file or directory
Killed
thales$
```

13.2.4 Interna von dynamischen Bibliotheken

Folgendes Beispiel zeigt, wie eine dynamische Bibliothek zur Kompilezeit verlinkt wird. Dies können wir natürlich auch „von Hand“ machen.

Programm 13.6: Dynamische Bibliotheken „low level“ verwenden (*dynload.c*)

```
#include <stdio.h>
#include <dlfcn.h>        /* wg. dlopen(), ... */

int main() {
    void *handle;
    double (*sin)(double); /* Zeiger fuer die Funktion sin() */
    double x = 3.1415;

        /* dyn. Bibliothek (shared object) oeffnen */
    if (!(handle = dlopen("libm.so", RTLD_LAZY))) {
        fprintf(stderr, "%s\n", dlerror());
        return 1;
    }

        /* Zeiger auf Funktion in dyn. Bibliothek (shared object) beschaffen */
    if (!(sin = dlsym(handle, "sin"))) {
        fprintf(stderr, "%s\n", dlerror());
        return 1;
    }

    printf("sin(%f) = %f\n", x, (*sin)(x));

        /* dyn. Bibliothek (shared object) wieder schliessen */
    dlclose(handle);

    return 0;
}
```

```
thales$ gcc -Wall dynload.c
thales$ a.out
sin(3.141500) = 0.000093
thales$
```

Anmerkung: Es kann evtl. sein, dass man (mit der Option *-l**dl***) die Bibliothek *lib**dl**.a* hinzubinden muss.

Erläuterung: Die Verwendung dynamischer Bibliotheken erfordert immer das *Öffnen der Bibliothek* (mittels *dlopen()*), die *Abfrage der Position eines Symbols im Speicher* (mittels *dlsym()*) und das *Schließen der dynamischen Bibliothek* (mittels *dlclose()*). Dabei erhält *dlopen()* den Namen der dynamischen Bibliothek und die Flags – *RTLD_LAZY* bedeutet, dass eine Funktion erst dann geladen wird, wenn sie mittels *dlsym()* zum ersten Mal zugegriffen wird – und liefert ein Handle, das ungleich dem Null-Zeiger ist, wenn das Öffnen geklappt hat. Die Meldung zu einem evtl. Fehler liefert die Funktion *dlerror()* (aber nur genau einmal und danach NULL). Die Funktion *dlsym()* liefert einen Zeiger auf das angegebene Symbol (Funktion oder Variable) im Speicher. Ist das Ergebnis der Null-Zeiger, so wurde das Symbol nicht gefunden. Der Typ der Variable *sin* ist ein Zeiger-Typ auf Funktionen der „Bauart“ `double f(double x)` (wie etwa der Sinus-Funktion :-).

Ausblick: Am vorigen Beispiel haben wir gesehen, dass es gar nicht nötig ist, zur Kompilezeit überhaupt den Namen der Bibliothek zu kennen – die Schnittstelle (d. h. die öffentlichen

Funktionen und Variablen) genügt. Der Name der Bibliothek muss nur als String bekannt sein. Diesen kann man beispielsweise durch eine Suche im Dateisystem ermitteln. Damit lassen sich einfach plugin-basierte Systeme entwickeln. (Unter einem *Plugin* versteht man eine Software-Komponente, die dynamisch hinzugefügt werden kann.)

13.3 Makefiles

Wenn ein Programm aus mehr als einer Datei besteht, dann ist das Erzeugen der ausführbaren Datei keine Trivialität mehr. Wird ein solches Projekt größer, so kann der Übersetzungs-Vorgang u. U. ganz schön lange dauern. Daher zerlegt man die Übersetzung in kleine Einheiten (Erzeugung der einzelnen Objekt-Codes). Dies wird im Folgenden anhand des Stack-Beispiels aus dem vorigen Kapitel demonstriert.

```
thales$ gcc -Wall -c stack.c
thales$ gcc -Wall -c mainStack.c
thales$ gcc -o stack mainStack.o stack.o
thales$ stack
String-Stack
------------
Moegliche Kommandos:
 +elem  push
 -      pop
 ?      isEmpty
 EOF    ENDE

> ^D
thales$
```

Durch die Option *-c* wird erreicht, dass nur der Objekt-Code (Datei mit Endung „.o") erzeugt wird. Der letzte gcc-Aufruf (ohne die Option -c) führt erst zum Aufruf des Linkers und damit zur Erzeugung der ausführbaren Datei. Bei einer Änderung im Quelltext müssen nun nur die betroffenen Dateien erneut übersetzt werden. Erfolgt z. B. bei dem Stack-Beispiel eine *Änderung in der Datei stack.c*, so kann die erneute Erzeugung von *mainStack.o* unterbleiben, da hier ja keine Änderung erfolgte. Weil aber *stack.h* sowohl in *stack.c* also auch in *mainStack.c* eingebunden wird, sind bei einer Änderung in *stack.h* alle Objekt-Codes neu zu erzeugen.

Diese Übersetzung „von Hand" durchzuführen, ist ganz schön nervig. Daher gibt es *make*, mit dem die Erzeugung von Dateien (ganz allgemein) automatisiert werden kann. Sie kennen make schon von Modula-2 (mmm, d. h. „Make Makefile for Modula-2") bzw. von Oberon (mmo, d. h. „Make Makefile for Oberon"). Mit den Kommandos mmm bzw. mmo erfolgt die automatische Erzeugung eines Makefiles, das dann nur noch mittels make ausgeführt werden muss. So komfortabel geht's in C leider nicht. :-(Hier müssen wir das *Makefile* schon selbst erzeugen.

Wie ist ein Makefile überhaupt aufgebaut? Ganz einfach: Es besteht im Wesentlichen aus einer Menge von Zielen (*targets*). Zu jedem Ziel, das entweder der Name einer Datei oder ein anderer Bezeichner sein kann, ist angegeben, von welchen anderen Zielen bzw. Dateien es abhängt (*prerequisites*) und welche Kommandos (*commands*) zur Erzeugung bzw. Aktualisierung des Ziels notwendig sind. Dies sieht dann in einem Makefile syntaktisch wie folgt aus:

```
TARGET1: <WHITESPACE> {<WHITESPACE>} {PREREQUISITE}
<TAB> {<TAB>} COMMAND
<TAB> {<TAB>} COMMAND
...
TARGET2: <WHITESPACE> {<WHITESPACE>} {PREREQUISITE}
<TAB> {<TAB>} COMMAND
<TAB> {<TAB>} COMMAND
...
```

Mit dem Zeichen *"#"* kann man einen Kommentar beginnen, der bis zum Zeilenende geht.

Bleiben wir bei dem Stack-Beispiel aus dem vorigen Kapitel. Folgendes Makefile kann dazu verwendet werden, dieses Programm zu übersetzen:

Programm 13.7: Makefile für das Stack-Beispiel (*Makefile*)

```
stack:      stack.o mainStack.o
            gcc -o stack  stack.o mainStack.o

stack.o:            stack.c  stack.h
            gcc -c stack.c

mainStack.o:        mainStack.c  stack.h
            gcc -c mainStack.c

.PHONY: clean  realclean

clean:
            rm -f mainStack.o  stack.o  core

realclean:          clean
            rm -f stack
```

Anmerkung: Mit *.PHONY* teilt man make mit, dass es sich bei den angegebenen Bezeichnern nicht um Dateien handelt. Diese Targets werden also auch erzeugt, selbst wenn es eine gleichlautende Datei gäbe. Die Targets clean und realclean sind ganz nützlich, um erzeugte Dateien einfach wieder „loszuwerden“. Wie besprochen, hängen *stack.o* und *mainStack.o* nicht nur von der zugehörigen C-Quelle, sondern auch von der Header-Datei *stack.h* ab. Da make ursprünglich zur Übersetzung von C-Quellen entstanden ist, *müsste man die Abhängigkeit zwischen* stack.o *und* stack.c *etc. gar nicht angeben*!

Aus obigem Makefile erzeugt **make** einen *gerichteten azyklischen Graphen* (siehe Abbildung 13.2) entsprechend den angegebenen *Abhängigkeiten*. Dieser (baumartige) Graph wird dann von make postorder traversiert, also zunächst die Prerequisites und dann das Target. Bei jedem Target wird (nach dem Durchlaufen der Untergraphen – daher auch die Bezeichnung **post**order) überprüft, ob die Zieldatei schon existiert und nicht älter ist als die Dateien, von denen sie abhängt. Ist dies der Fall, so werden die Kommandos für dieses Zeil nicht ausgeführt – andernfalls schon.

Soll in unserem Beispiel nun die ausführbare Datei *stack* erzeugt werden, so gibt es folgende interne Abläufe nach dem Aufruf von make:

1. Existiert *stack.o* und ist es nicht älter als *stack.c* und *stack.h*? Wenn nein, dann führe die Kommandos aus, um *stack.o* zu erzeugen.

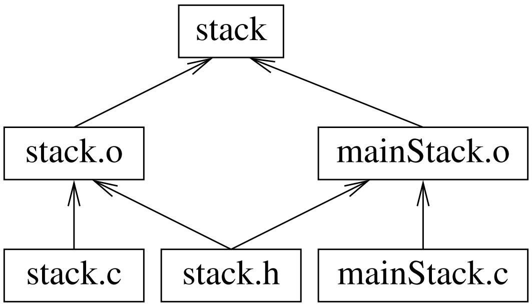


Abbildung 13.2: Von make erzeugter Abhängigkeitsgraph

2. Existiert *mainStack.o* und ist es nicht älter als *mainStack.c* und *stack.h*? Wenn nein, dann führe die Kommandos aus, um *mainStack.o* zu erzeugen.
3. Existiert *stack* und ist es nicht älter als *stack.o* und *mainStack.o*? Wenn nein, dann führe die Kommandos aus, um *stack* zu erzeugen.

```
thales$ ls
Makefile  mainStack.c  stack.c  stack.h
thales$ make
gcc -c stack.c
gcc -c mainStack.c
gcc -o stack stack.o mainStack.o
thales$ ls
Makefile  mainStack.c  mainStack.o  stack  stack.c  stack.h  stack.o
thales$ make
make: 'stack' is up to date.
thales$ touch mainStack.c
thales$ make
gcc -c mainStack.c
gcc -o stack stack.o mainStack.o
thales$ make
make: 'stack' is up to date.
thales$ touch stack.c
thales$ make
gcc -c stack.c
gcc -o stack stack.o mainStack.o
thales$ make
make: 'stack' is up to date.
thales$ touch stack.h
thales$ make
gcc -c stack.c
gcc -c mainStack.c
gcc -o stack stack.o mainStack.o
thales$ make clean
rm -f mainStack.o stack.o core
thales$ ls
Makefile  mainStack.c  stack  stack.c  stack.h
thales$ make realclean
rm -f mainStack.o stack.o core
rm -f stack
thales$ ls
Makefile  mainStack.c  stack.c  stack.h
thales$
```

Anmerkungen:

- Mit dem Kommando *touch* kann man Datum und Zeit der letzten Änderung einer Datei auf „jetzt“, also auf das aktuelle Datum und die aktuelle Uhrzeit, setzen.
- Wird make *ohne ein Kommandozeilen-Argument* aufgerufen, so wird das *erste Target im Makefile* erzeugt (in unserem Fall „stack“). Ist dies nicht gewünscht, so muss man das *Target als Kommandozeilen-Argument von make* angeben (siehe *make clean* bzw. *make realclean*).
- Standardmäßig sucht make nach einem Makefile, das *makefile* oder *Makefile* heißt. Gibt es keine solche Datei, so bricht es ab. Hat das Makefile einen anderen Namen, so kann man diesen mit der Option *-f* make mitteilen.
- Mit der Option *-n* gibt make nur die Kommandos aus, ohne sie jedoch tatsächlich auszuführen.

Nun werden wir das vorgestellte Makefile noch etwas verbessern. In Makefiles kann man auch Variablen definieren mittels *VARIABLE = WERT* und dann mit *$(VARIABLE)* verwenden. Dies werden wir verwenden und außerdem eliminieren wir die Abhängigkeiten, die make sowieso schon vermutet (z. B. die Abhängigkeit zwischen *stack.o* und *stack.c*). Muss ein Objekt-Code wie etwa *stack.o* nur mit dem Kommando „`$(CC) $(CFLAGS) -c stack.c`“ („`-o stack.o`“ wird dabei implizit angenommen) erzeugt werden, so kann man auch kein Kommando angeben. In diesem Fall führt make genau dieses Kommando aus.

Programm 13.8: Ein verbessertes Makefile für das Stack-Beispiel (*makefile*)

```
CC = gcc
CFLAGS = -Wall
OBJ = stack.o mainStack.o

stack: $(OBJ)
	$(CC) $(CFLAGS) -o stack $(OBJ)

stack.o:	stack.h

mainStack.o:	stack.h

.PHONY: clean realclean

clean:
	rm -f $(OBJ) core

realclean:	clean
	rm -f stack
```

```
thales$ ls
mainStack.c  makefile  stack.c  stack.h
thales$ make
gcc -Wall   -c -o stack.o stack.c
gcc -Wall   -c -o mainStack.o mainStack.c
gcc -Wall -o stack stack.o mainStack.o
thales$
```

Anmerkungen: Wie man sieht, wird für *stack.o* und *mainStack.o* jeweils genau das oben beschriebene Kommando ausgeführt. Man erkennt hier außerdem, dass *CC* und *CFLAGS* Variablen sind, die make schon kennt und in solchen Fällen verwendet, um den Compiler mit den gewünschten Flags aufzurufen.

Kapitel 14

Die C-Standards

14.1 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 defacto-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.

14.2 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.

14.2.1 Einzeilige Kommentare

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

Programm 14.1: Einzeilige Kommentare (*comment.c*)

```c
#include <stdio.h>

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

14.2.2 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 14.2: Mischen von Deklarationen/Definitionen und Anweisungen (*mixed.c*)

```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;
}
```

14.2.3 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 14.3: Variablen in for-Schleifen (*for.c*)

```c
#include <stdio.h>

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

14.2.4 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 14.4: 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$
```

14.2.5 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 14.5: Flexibles Array-Element in Strukturen (*struct.c*)

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

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

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

    int len = 3;
    struct my *s;

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

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

    return 0;
}
```

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

14.2.6 Nicht-konstante Initialisierer

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

Programm 14.6: Nicht-konstante Initialisierer (*init.c*)

```c
#include <stdio.h>

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

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

14.2.7 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.

14.2.7.1 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 14.7: 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.

14.2.7.2 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};
```

14.2.8 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 14.8: 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;
}
```

14.2.9 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 14.9: 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;
}
```

14.2.10 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 14.10: 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$
```

14.2.11 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 14.11: 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!

14.2.12 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 14.12: 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$
```

14.2.13 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 14.13: assert mit Name der aktuellen Funktion (*assert.c*)

```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$
```

14.2.14 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 14.14: Inline-Funktionen (*inline.c*)

```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 15

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.

15.1 Typische Schwachstellen

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

Programm 15.1: 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 übergebene Unix-Kommando ausführt. Das Kommando *cat* gibt einfach die Dateien auf die Standardausgabe aus, deren Namen es als Kommandozeilen-Argumente erhält.

Programm 15.1, 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 15.1 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 15.2: printf() und die Formatierungsangabe %n (*printf.c*)

```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.

15.2 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 15.3: 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.

15.3 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.

Teil II

Das Betriebssystem Unix

Kapitel 16

Das Aufbau des Betriebssystems Unix

16.1 Betriebssysteme allgemein

16.1.1 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.

16.1.2 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.

16.1.3 Schichtenmodell

Das *Schichtenmodell* teilt die *„erweiterte Maschine"* in die folgenden Schichten ein (siehe Abbildung 16.1):

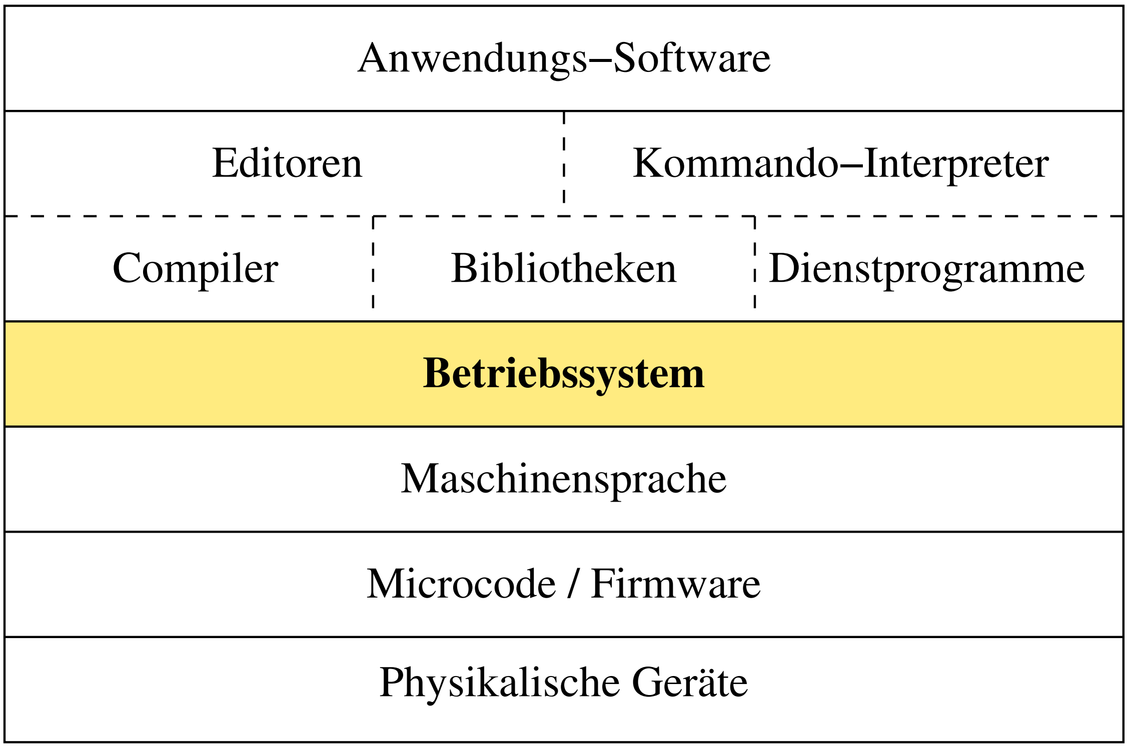


Abbildung 16.1: Das Schichtenmodell der „erweiterten Maschine"

- **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).

16.2 Unix-Schalenmodell

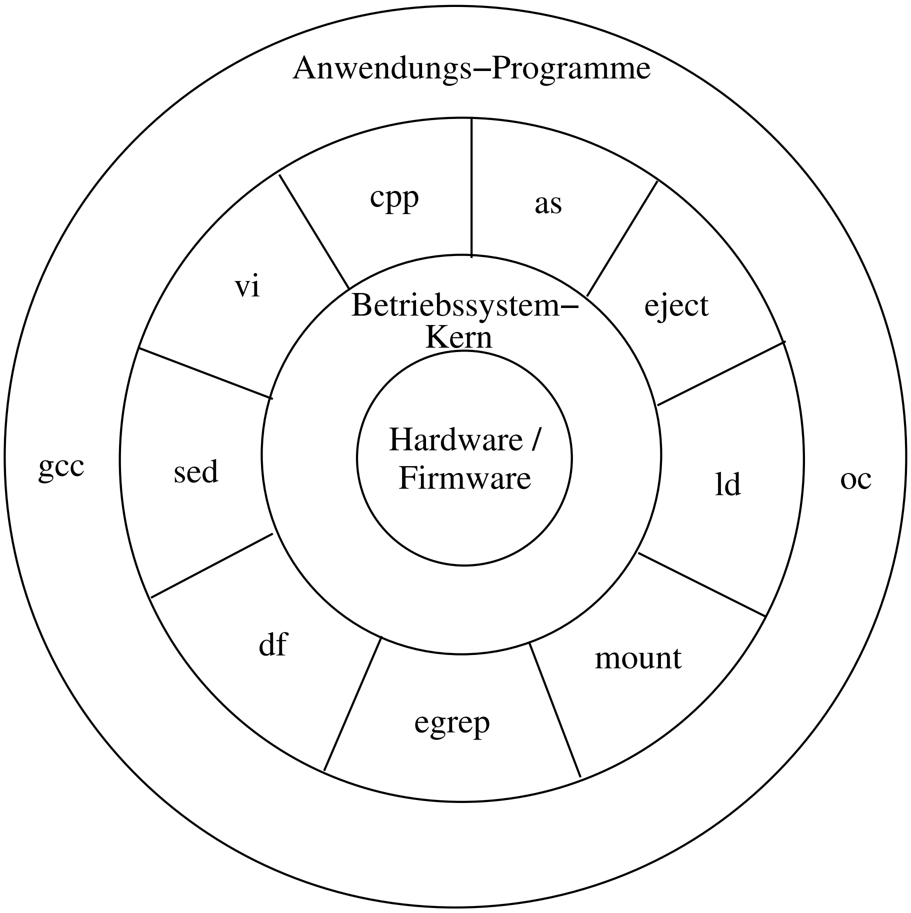


Abbildung 16.2: Das Unix-Schalenmodell

Abbildung 16.2 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.

16.3 Interner Aufbau von Unix

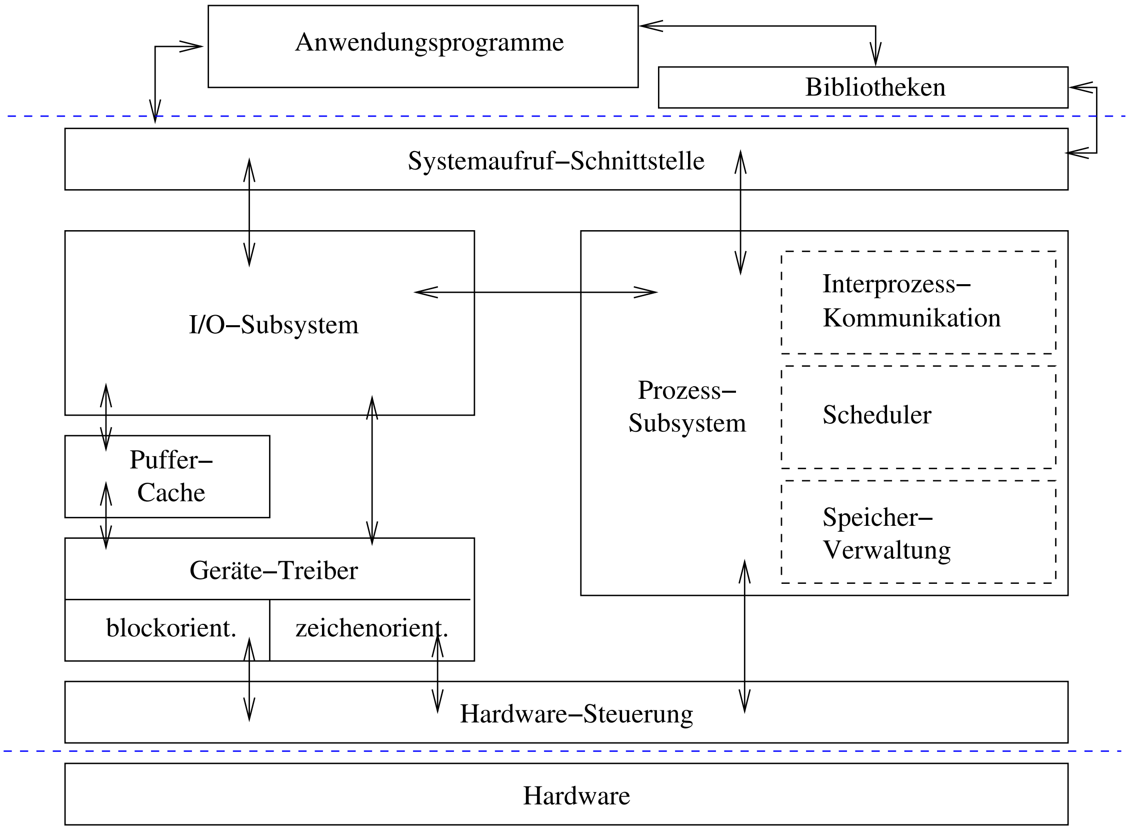


Abbildung 16.3: Der interne Aufbau von Unix

Der interne Aufbau des Betriebssystems Unix ist in Abbildung 16.3 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 17

Das I/O-Subsystem

17.1 Dateien

17.1.1 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“.

17.1.2 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

17.1.3 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

17.1.4 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)

17.1.5 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.

17.2 Dateisysteme

17.2.1 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*)

17.2.2 Netzwerk-Dateisysteme

17.2.2.1 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 17.1 zeigt ein Beispiel für ein verteiltes Dateisystem (nach [Handschuch93]).

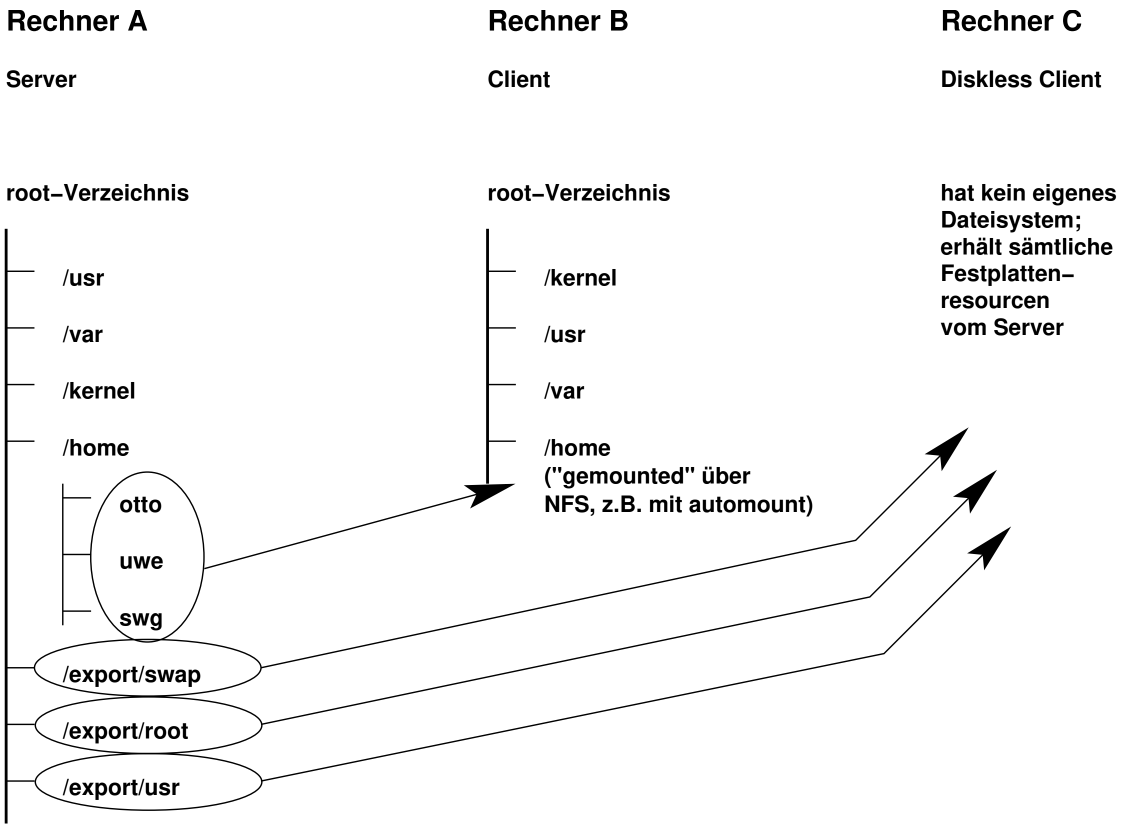


Abbildung 17.1: Beispiel für ein Netzwerk-Dateisystem

17.2.2.2 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!)

17.2.2.3 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.

17.2.2.4 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*.

17.2.3 Pseudo-Dateisysteme

17.2.3.1 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!).

17.2.3.2 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.

Boot–block	Super–block	Inode–Liste	Datenblöcke

Abbildung 17.2: Grobstruktur des Unix-Dateisystems

17.2.4 Das Unix-Dateisystem (UFS)

17.2.4.1 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 17.2 zeigt die Grobstruktur eines Unix-Dateisystems mit den folgenden 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$
```

17.2.4.2 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)	...

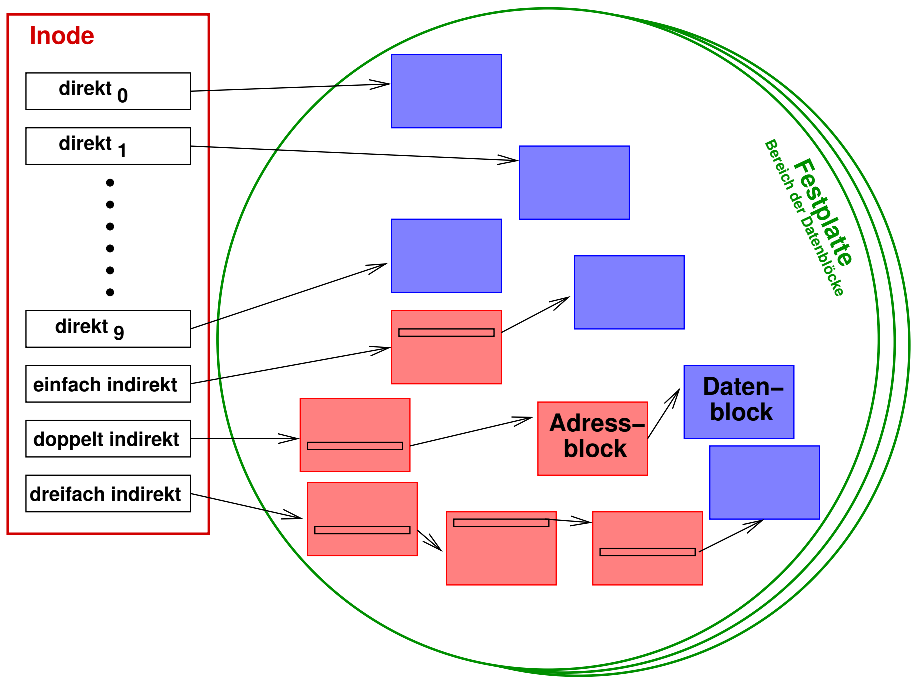


Abbildung 17.3: Adressierung der Datenblöcke

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 17.3). Bei indirekter Adressierung stehen in Datenblöcken, sog. 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. 17.4 und Abb. 17.5).

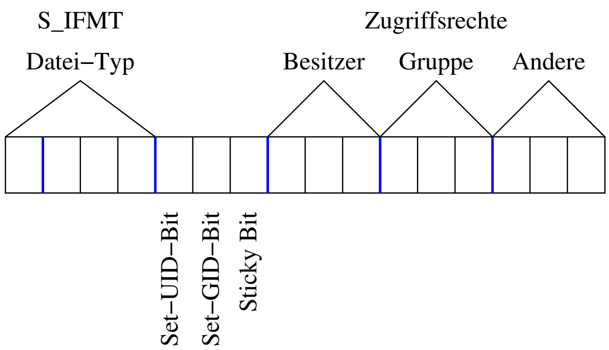
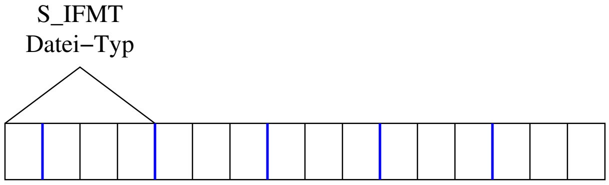


Abbildung 17.4: Kodierung der Komponente `st_mode`

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.



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 17.5: 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 17.1: Informationen über Dateien ermitteln – mittels stat() (*stat.c*)

```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).

17.2.4.3 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 17.2: 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*.

17.2.4.4 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 -> hallo.txt
chomsky$
```

Konsequenzen

- Durch Hardlinks wird aus dem Dateibaum ein zyklenfreier gerichteter Graph.
- Durch Softlinks können auch Zyklen im Dateisystem entstehen (durch Softlinks auf Verzeichnisse)!

17.3 Systemaufrufe für I/O-Verbindungen – Erster Teil

17.3.1 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:

```c
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.

```c
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!)

```c
int fd;
if ((fd = open(dateiname, O_WRONLY | O_CREAT | O_EXCL, perms)) < 0) {
    /* Fehlerbehandlung, z.B. */
    perror("open()");
    exit(1);
}
```

Programm 17.3 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 17.3: 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).

17.3.2 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.

17.3.3 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 17.4: Umlenkung der Standardausgabe – mittels dup() (*dup.c*)

```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. ;-)

17.3.4 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 17.1 auf Seite 171 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 17.5: 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$
```

17.3.5 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 17.2.4.3 auf Seite 172 bereits erläutert und mit dem Programm 17.2 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.

17.3.6 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 17.6: Ausgabe eines Strings mit Hilfe von write() (*write.c*)

```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 17.7: Ausgabe eines Strings mit Hilfe von write() (*write1.c*)

```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;
}
```

17.3.7 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 17.8: Einlesen einer Zeile von der Standardeingabe – mittels read() (*read.c*)

```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*.

17.3.8 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 17.9: Fehlerbehandlung bei Systemaufrufen – Die Erste (*error1.c*)

```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 17.10: Fehlerbehandlung bei Systemaufrufen – Die Zweite (*error2.c*)

```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 */

/* ... */
```

17.4 Datenstrukturen für I/O-Verbindungen

17.4.1 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 17.6).

- **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.

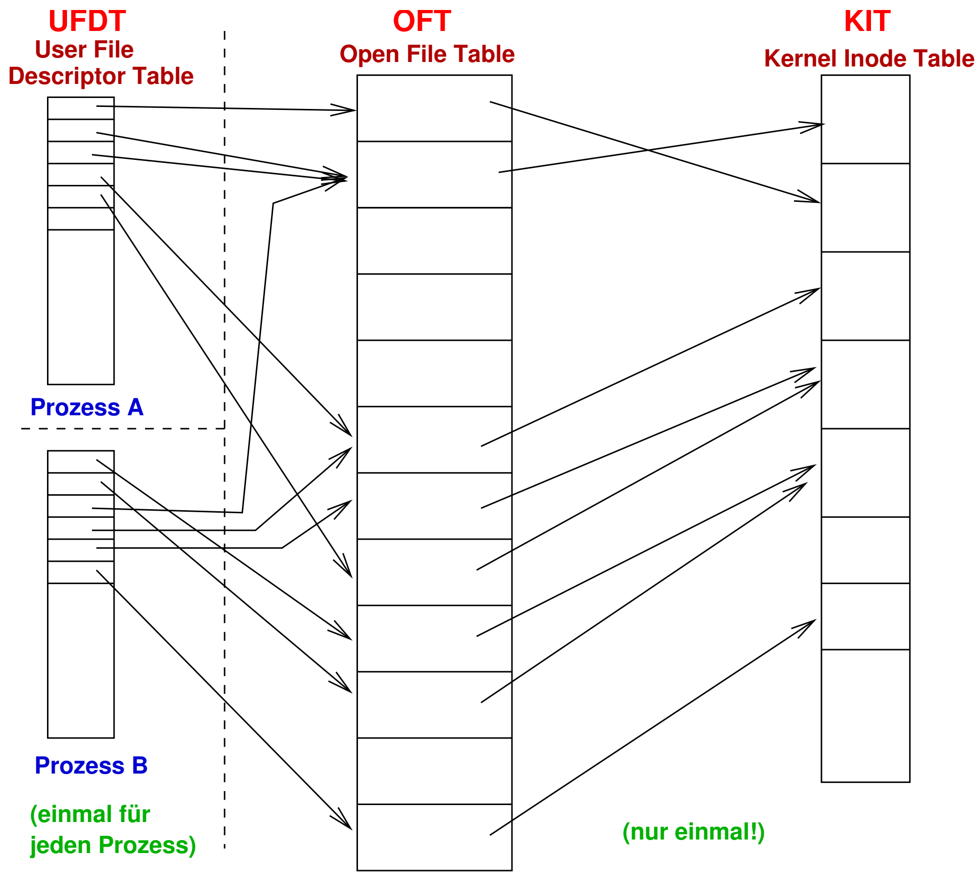


Abbildung 17.6: Datenstrukturen für I/O-Verbindungen

- **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*),
  - 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.

17.4.2 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.

17.4.2.1 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*“.

17.4.2.2 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.

17.4.2.3 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.

17.4.2.4 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.

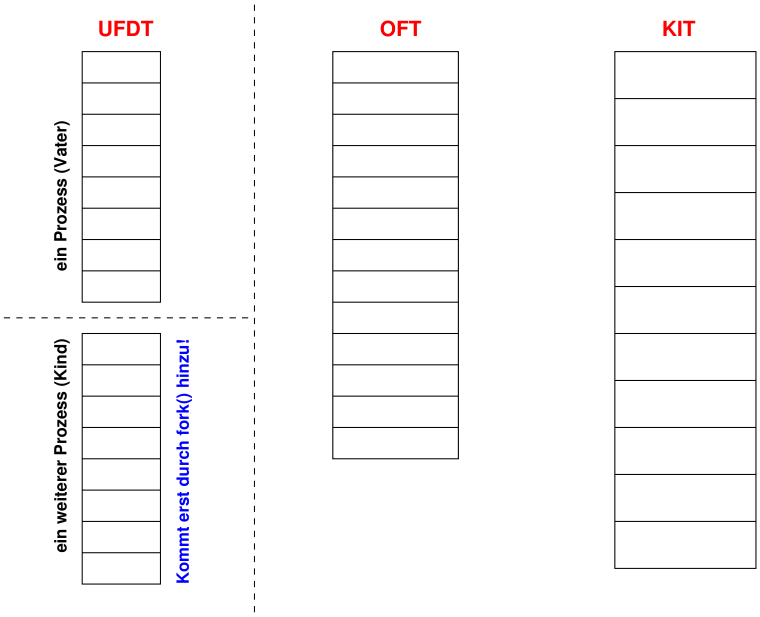


Abbildung 17.7: Kerneltabellen – Ausgangszustand

17.4.2.5 Beispiel

Im Folgenden soll die Belegung der **Kerneltabellen** über eine Folge von Systemaufrufen gezeigt werden. *Der Ausgangszustand ist in Abbildung 17.7 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 17.8.*

Vater:

```
fork();
```

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 17.9.*

Vater:

```
fd3 = open("file3", O_WRONLY);
```

Kind:

```
fd3 = open("file3", O_RDONLY);
```

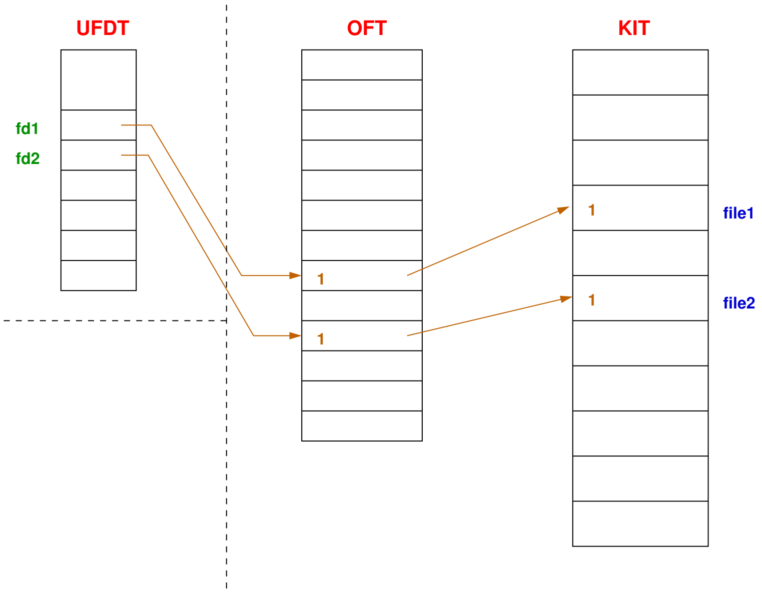


Abbildung 17.8: Kerneltabellen (1)

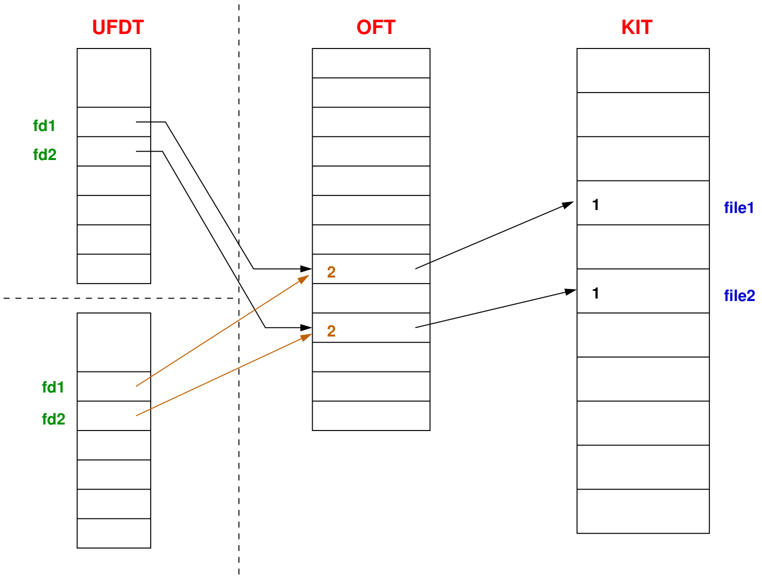


Abbildung 17.9: Kerneltabellen (2)

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 17.10.*

Vater:

```
fd4 = open(" file4 ", O_RDONLY);
```

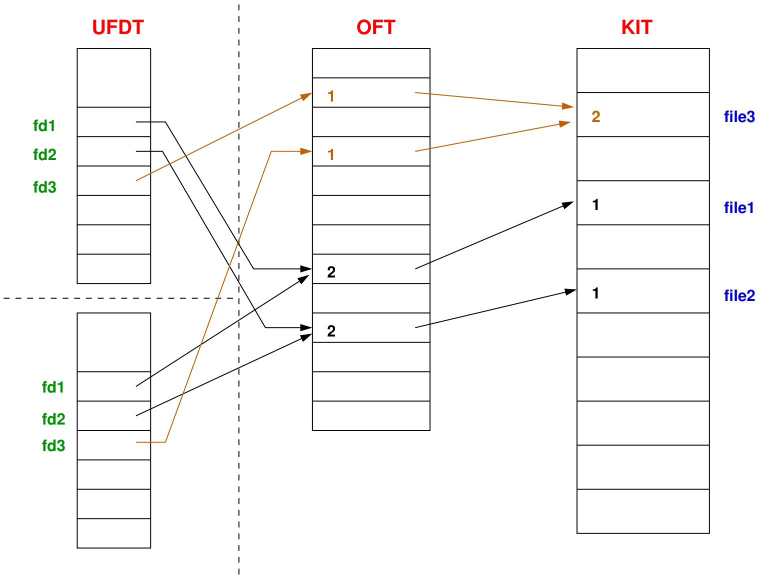


Abbildung 17.10: Kerneltabellen (3)

Kind:

fd4 = open(" file5 ", O_RDONLY);

Danach sieht die Situation wie in Abbildung 17.11.

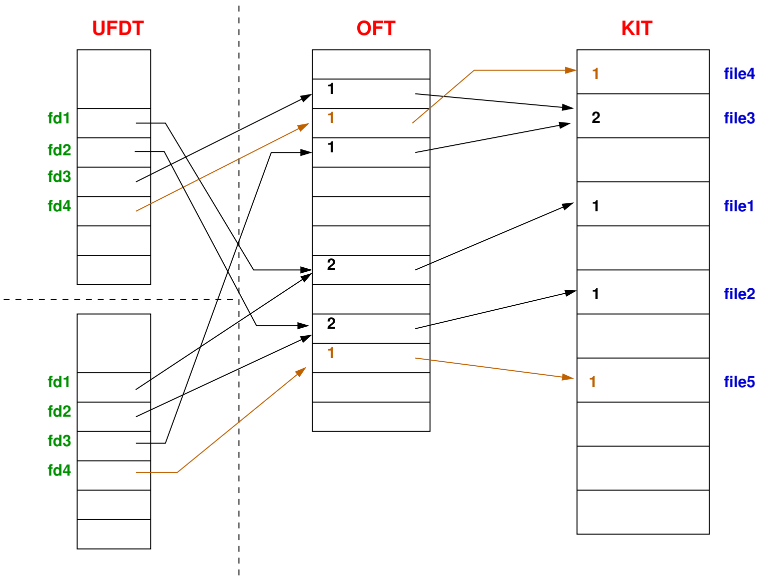


Abbildung 17.11: Kerneltabellen (4)

Kind:

close (fd1);

Vater:

close (fd2);

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 17.12.*

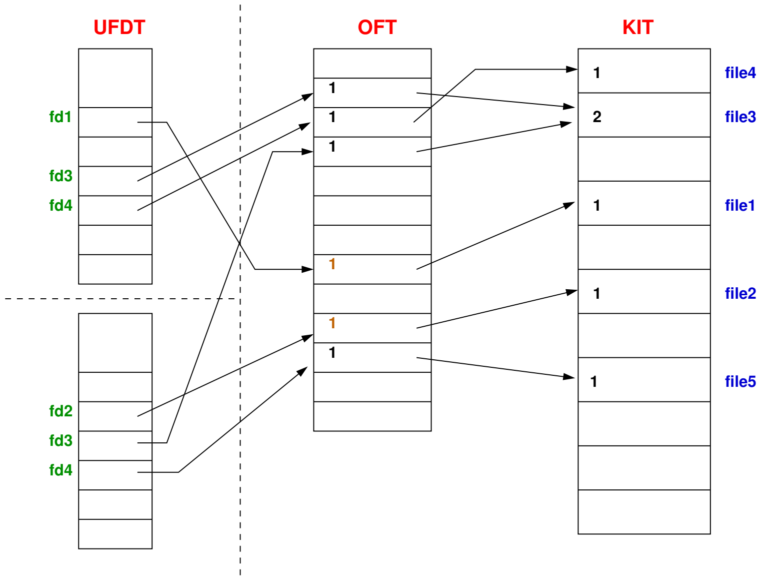


Abbildung 17.12: Kerneltabellen (5)

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 17.13.*

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 (⇝ auf Platte, wenn Link Count größer 0 und sonst Datei freigeben).

Vater:

close (fd1);
close (fd4);

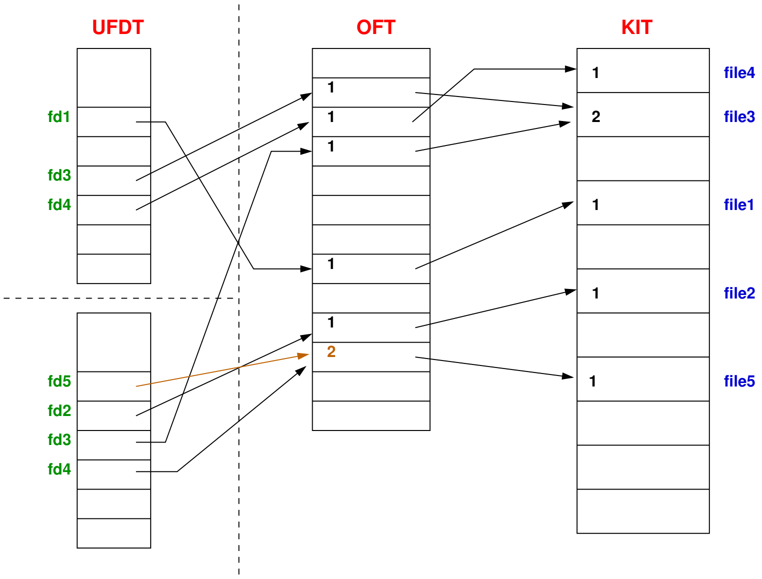


Abbildung 17.13: Kerneltabellen (6)

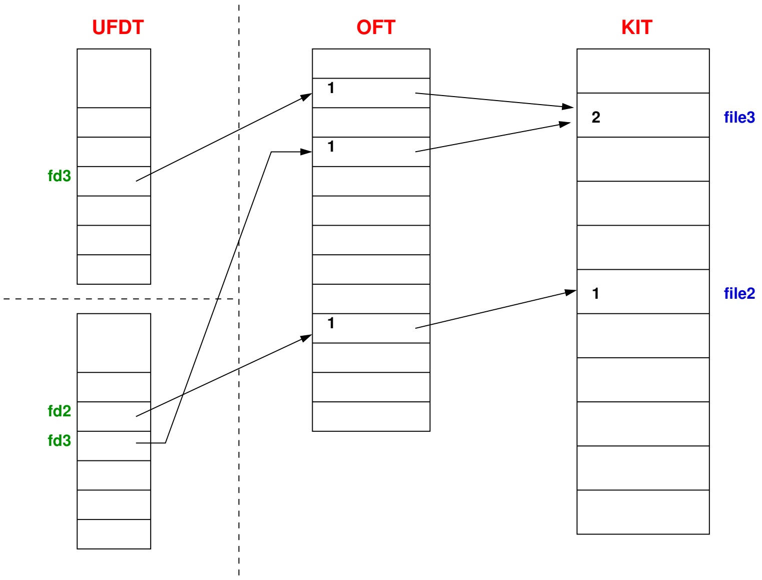


Abbildung 17.14: Kerneltabellen (7)

Nach jedem dieser *close()*-Aufrufe werden alle Slots freigegeben, da alle Referenz-Zähler auf 0 gehen. *Danach sieht die Situation wie in Abbildung 17.14.*

Kind:

close (fd2);
close (fd3);

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 17.7).

17.5 Systemaufrufe für I/O-Verbindungen – Zweiter Teil

17.5.1 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 17.11: 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.

17.5.2 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 ($\leadsto$ zur Synchronisation geeignet, genauso wie *open()* mit *O_EXCL* – später mehr dazu).

17.5.3 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 17.12: 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*).

17.5.4 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 17.13: 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$
```

17.5.5 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 17.14: Echo aus- und anschalten – mittels ioctl() (*echo.c*)

```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;
}
```

17.6 Synchronisation

17.6.1 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 17.15: Inkrementieren ohne Synchronisation (*nolock.c*)

```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.

17.6.2 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 17.16: Inkrementieren mit Synchronisation (*lock.c*)

```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.

17.6.3 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 ($\leadsto$ 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 17.17: Inkrementieren mit Synchronisation – mittels lockf() (*lockf.c*)

```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

[Bach86] M. J. Bach: *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[Comer84] D. Comer: *Operating System Design: The XNIU Approach.* Prentice Hall, 1984.

[Darnell01] P. A. Darnell und P. E. Margolis: *C: A Software Engineering Approach.* Springer, Dritte Auflage, 2001.

[Handschuch93] T. Handschuch: *SOLARIS 2 für den Systemadministrator.* Solaris Galerie, IWT-Verlag, 1993.

[Herold99] H. Herold: *Linux-Unix-Shells.* Addison-Wesley, 1999.

[Kernighan86] B. W. Kernighan und R. Pike: *Der UNIX-Werkzeugkasten.* Hanser, 1986.

[Kernighan90] B. W. Kernighan und D. Ritchie: *Programmieren in C.* Hanser, Zweite Auflage, 1990.

[Rochkind88] M. Rochkind: *UNIX-Programmierung für Fortgeschrittene.* Hanser Verlag, 1988.

[Stevens92] W. R. Stevens: *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992.

[Tanenbaum87] A. S. Tanenbaum: *Operating Systems - Design and Implementation.* Prentice Hall, 1987.

⇝ **jeweils auf aktuellste Ausgabe achten!**

Abbildungsverzeichnis

Beispiel-Programme

Index