Dr. Helmut Herold
Dr. Jörg Arndt

C-Programmierung unter Linux / UNIX / Windows

Beispiele, Anwendungen, Programmiertechniken

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über `http://dnb.ddb.de` abrufbar.
ISBN 3-938626-17-7

Umschlaggestaltung: millin Verlag, `http://www.millin.de`
Gesamtlektorat: Nicolaus Millin
Fachlektorat: Dieter Bloms, Jörg Dippel, Klaas Freitag, Bruno Gerz, Bernhard Hoelcker, Björn Jacke, Andreas Jaeger, Dirk Pankonin, Wolfgang Rosenauer, Christian Steinrücken, Peter Varkoly
Satz: LaTeX
Druck: druckhaus köthen GmbH, Köthen (`http://www.koethen.de`
Printed in Germany on acid free paper.

Inhaltsverzeichnis

Vorwort

Ein paar Worte zu C und seiner Geschichte

Zu Beginn wird hier kurz auf die Historie von C und die verschiedenen Standardisierungen von C eingegangen.

Die Entstehung von C

Die Sprache C wurde im Jahre 1972 von *Dennis M.Ritchie* in den Bell-Laboratorien bei AT&T entwickelt und von *Brian W. Kernighan* in den Jahren 1973/74 weiter verbessert. Vorläufer von C waren die Sprachen *BCPL* (Basic Combined Programming Language) und *B*.
C war schon immer sehr eng mit dem Betriebssystem Unix verbunden, da die Sprache C auf diesem damals noch jungen Betriebssystem entwickelt wurde und Unix wiederum selbst mit allen seinen Dienstprogrammen nahezu vollständig in C geschrieben wurde. In den 80er Jahren hat sich aber C als eine universell einsetzbare Sprache entpuppt, was dazu führte, dass C heute auf nahezu jedem beliebigen Betriebssystem (Linux, Unix-Systeme, MS-DOS, Windows95/98/NT/..., VMS, OS/2, ...) angeboten wird.
In den 1970er und 1980er Jahren gab es keine Standardbeschreibung zur Sprache C. Statt dessen galt die 1. Ausgabe des Buches *„The C Programming Language"* von Kernighan und Ritchie (Prentice Hall, 1978) als die Bibel für alle C-Fragen. Diese „Bibel" ließ jedoch einige Fragen offen. So wurde bereits in den frühen 80er Jahren die Notwendigkeit für einen wirklichen C-Standard erkannt:

- **C89 – der erste Standard für C**
  Im Jahre 1983 begann das ANSI-Komitee X3J11 [1] mit dem Unterfangen, die Sprache C zu standardisieren. Im selben Jahr noch entschied das Komitee X3J11, dass nur *ein* C-Standard geschaffen werden soll, welcher von beiden Organisationen ANSI und ISO verabschiedet wird. C wurde zum ersten Mal Ende des Jahres 1989 mit der Annahme des ANSI-Standards standardisiert. Diese Version von C wird allgemein als *C89* bezeichnet. Dieser Standard wurde auch im Jahre 1990 von ISO übernommen. C89 wurde im Jahre 1995 leicht verbessert.
- **C99 – der neue Standard für C**
  Im Jahre 1999 wurde ein neuer Standard für C geschaffen. Diese Version nennt sich *ISO C99*, oder kurz auch nur *C99*. C99 enthält eine Reihe von Verbesserungen und einige neue Konstrukte, welche teilweise von C++ übernommen wurden.
  Alle C-Programme, die die von C99 vorgegebenen Vorschriften einhalten, werden *C99 kompatibel* genannt. Solche Programme haben den Vorteil, dass sie leicht portierbar sind, also ohne größere Änderungen von einem anderen C99-Compiler auf einer anderen Maschine in eine ablauffähige Version übersetzt werden können.

C als Vorstufe zu C++

C++ wurde – beginnend im Jahr 1979 – von *Bjarne Stroustrup* entwickelt. C++, das eine objektorientierte Version zu C ist, hat sich inzwischen sehr verbreitet. Da aber C++ nur eine Erweiterung zu C ist, muss auch jeder, der mit C++ programmieren will, die Sprache C richtig beherrschen, bevor er in die höheren Sphären der objektorientierten Programmierung aufsteigen kann.

[1] *ANSI* (**A**merican **N**ational **S**tandard **I**nstitute) ist eine amerikanische Organisation, welche Mitglied der **I**nternational **S**tandards **O**rganisation (ISO) ist.

Hinweise zu diesem Buch

Dieses Buch beschreibt die Theorie der Programmiersprache C anhand vieler Beispiele. Dabei begnügt es sich jedoch nicht allein mit der Vorstellung der einzelnen C-Elemente, sondern vermittelt dem C-Neuling auch Einblicke in wichtige Grundlagen der Informatik. Zudem gibt es auch zu den einzelnen C-Konstruktionen Programmiertechniken aus der C-Praxis, typische Anwendungsgebiete, Tipps und Fallgruben, die in C leider nicht allzu selten sind. Diese Grundkenntnisse bilden das Fundament, das für eine erfolgreiche Programmierung in C unverzichtbar ist.

Nach dem Durcharbeiten dieses Buches verfügt der Leser über ein gesundes und breites C-Fundament, und ist in der Lage, erfolgreich in C zu programmieren. Die Intention dieses Buches ist es nämlich:

- den C-Anfänger systematisch vom C-Basiswissen bis hin zu den fortgeschrittenen Techniken der Programmierung zu führen.
- dem bereits erfahrenen C-Programmierer – aufgrund der Vielzahl von Tipps, fundamentalen Algorithmen und nützlichen Programmiertechniken – eine Vertiefung bzw. Ergänzung seines C-Wissens zu ermöglichen.

Layout der Programme in diesem Buch

Es sei vorweg darauf hingewiesen, dass sich die in diesem Buch vorgestellten Programme nicht immer an die allgemein gültigen Programmier-Richtlinien halten. So befinden sich z. B. manchmal mehrere Anweisungen in einer Zeile. Dies wurde sowohl aus Platzgründen als auch aus Layout-Gesichtspunkten, die in einem solchen Buch einzuhalten sind, leider notwendig, denn ein Programmlisting, bei dem sich eventuell die entscheidenden Anweisungen über zwei Seiten verteilen, ist sicherlich aus didaktischen Gründen hinsichtlich Lesbarkeit ungünstiger als ein Programm, das sich vollständig auf einer Seite befindet, ohne dass der Leser ständig vor- und zurückblättern muss.

Downloads zu diesem Buch

Alle Programme in diesem Buch können von folgender Webseite

http://www.millin.de

heruntergeladen werden.

Windows-Kompatibilität

Auch wenn dieses Buch ursprünglich für die C-Programmierung unter Linux konzipiert wurde, so sind doch nahezu alle in diesem Buch vorgestellten Programme auch unter Windows-Compilern lauffähig. Einschränkungen gibt es lediglich bei einigen Windows-Compilern, die noch nicht den neuesten C99-Standard implementiert haben, aber das betrifft nur einige wenige Programme in diesem Buch.

Auch zu der ursprünglich für Linux eigens im Rahmen dieses Buches entwickelten Graphikbibliothek LCGI existiert zwischenzeitlich eine Windows-Version, die von der oben erwähnten Webseite heruntergeladen werden kann.

Kapitel 1

Einführendes Beispiel

1.1 Erste wesentliche C-Regeln

Das erste C-Programm `erst.c`:

```
/*************************************************************/
/*               Einfuehrendes Beispiel                      */
/*************************************************************/

#include  <stdio.h>

int main(void)
{
   printf("************************************************\n");
   printf("*                                              *\n");
   printf("*         Das ist mein erstes  C-Programm      *\n");
   printf("*                                              *\n");
   printf("************************************************\n");

   return 0;  /* auch moeglich:   return(0); */
}
```

Hier sind bereits einige grundlegende C-Regeln zu erkennen:

1. Vorläufig müssen alle C-Programme die Zeichenfolge `int main(void)` enthalten, was den eigentlichen Programmanfang kennzeichnet.
2. Der zu `main()` gehörige Programmteil ist in geschweiften Klammern einzubetten. Dabei kennzeichnet:
   - **`{`** den Anfang und
   - **`}`** das Ende des entsprechenden Programmteils.
3. Ein Kommentar wird mit `/*` ..... `*/` geklammert. Kommentare sind Informationen für den Programmierer und werden vom C-Übersetzer (auch C-Compiler genannt) nicht beachtet. Um ein Programm möglichst lesbar zu machen, sollte es möglichst gut kommentiert sein.
4. Bildschirmausgaben sind mit `printf("Text")` möglich. Auszugebender Text muss dabei mit Anführungszeichen geklammert sein.

5. Um die einzelnen C-Anweisungen voneinander zu trennen, muss immer ein Semikolon als Trennzeichen angegeben werden. Vor und nach einem Semikolon können beliebig viele Leerzeichen oder auch Leerzeilen angegeben werden; gilt übrigens auch für andere Wörter wie `main`, `printf` oder `return`.
6. `#include ‹stdio.h›` sollte zunächst immer am Anfang (vor `main()`) angegeben werden. Ebenso sollte vor der abschließenden geschweiften Klammer immer `return 0;` oder `return(0);` angegeben werden.
7. Die Angabe von `\n` im `Text` bei `printf("Text")` bewirkt, dass beim Programmablauf an dieser Stelle auf den Anfang der nächsten Zeile gesprungen wird. Das Steuerzeichen `\n`, das immer in Anführungszeichen anzugeben ist, wird also niemals als Text ausgegeben, sondern bewirkt einen Zeilenvorschub.

Wird als letztes Zeichen in einem auszugebenden Text nicht `\n` angegeben, so wird beim nächsten `printf` in der gleichen Zeile (an der alten Bildschirmposition) mit der Ausgabe fortgefahren, wie es das C-Programm `zweit.c` verdeutlicht:

```
#include  <stdio.h>

int main(void)
{
   printf("****\n*  *\n*  *\n****\n\n");
   printf("Das ist ein Viereck ");
   printf("aus Sternchen, ");
   printf("oder nicht ?");
   printf("\nE\nN\nD\nE");
   printf("N");
   printf("D");
   printf("E\n");
   return 0;
}
```

Startet man dieses Programm `zweit`, liefert es die folgende Bildschirmausgabe:

```
****
*  *
*  *
****

Das ist ein Viereck aus Sternchen, oder nicht ?
E
N
D
ENDE
```

Eine Angabe wie:

```
printf("Guten Ta
g");
```

ergibt einen Fehler, da ein Zeilenwechsel innerhalb der in "…" eingebetteten Zeichenkette nicht erlaubt ist.

1.2 Zeilen-Kommentare mit // (neu in C99)

Neben dem schon immer vorhandenen C-Kommentar, der in einem C-Programm mit der Zeichenfolge `/*` eingeleitet und mit der Zeichenfolge `*/` beendet wird, hat C99 zusätzlich noch wie C++ auch die Zeichenfolge `//` zum Kommentieren eingeführt: Alle Zeichen ab `//` bis zum Zeilenende werden dabei als Kommentar interpretiert:

```
printf("Frau Holle, "); /* Normaler C-Kommentar, der sich über
                           mehrere Zeilen erstrecken kann und
                           auch weiterhin verwendet werden kann.
                         */
printf("wie gehts?\n"); // Ab hier bis zum Zeilenende Kommentar (neu in C99)
                        // Bei mehrzeiligen Kommentare ist erneut // anzugeben
```

1.3 Gute Lesbarkeit von Programmen

Einrücken von untergeordneten Programmteilen

Wie einzurücken ist, wurde bereits in den vorherigen Programmbeispielen für die Anweisungen innerhalb der geschweiften Klammern von `main()` gezeigt. Es gibt keine allgemeingültige Regel, wie viele Leerzeichen einzurücken sind; üblich ist jedoch das Einrücken um drei, vier oder auch acht Leerzeichen.

Sinnvolle Kommentare

Da Kommentare in einem C-Programm der Mensch/Mensch-Kommunikation dienen, sind sie ein wichtiges Hilfsmittel, um die Lesbarkeit eines Programms zu erhöhen und somit das Verständnis zu erleichtern.

- **Kommentare sollten zusätzliche Information enthalten**
  Die Befehlsbedeutung ist nicht zu kommentieren. Kommentare wie
  `i=i+1; /* i wird um 1 erhoeht */`
  sind nicht nur überflüssig, sondern sogar schädlich, da sie den Programmtext unnötig aufblähen, was eher zur Unübersichtlichkeit und damit zu einer schlechteren Lesbarkeit führt.
- **Nicht zuviel Programmtext auf einmal kommentieren**
  Drei Seiten zusammenhängender Kommentar, um die nächsten drei Seiten Programmtext zu beschreiben, dient nicht der besseren Verständlichkeit eines Programms, sondern lässt sich auch nicht leicht ändern, wenn sich der betreffende Programmteil grundlegend ändert.
- **Ein falscher Kommentar ist schlimmer als gar kein Kommentar**
  Kommentare müssen bei jeder Änderung des Programms aktualisiert werden.
- **Einige gute treffende Sätze sind mehr wert als Romane**
  Bei Kommentaren geht es meist darum, den Nagel auf den Kopf zu treffen. Die Professionalität und Güte eines Programmierers lässt sich unter anderem an seinem Stil der Kommentierung erkennen.

1.4 Vermeiden von geschachtelten Kommentaren

Obwohl einige Compiler geschachtelte Kommentare, wie z. B.:

```
.....
/*
   .....
     /*
      .....
      .....
      */
   .....
*/
.....
```

zulassen, ist es nicht empfehlenswert, von solchen Schachtelungen Gebrauch zu machen, da diese nicht durch die C-Standards „abgesegnet“ sind. Verwendet man trotzdem solche Kommentar-Schachtelungen, dann verstößt man gegen die in der SW-Entwicklung wichtige Forderung der Portabilität, da ein solches Programm eventuell auf anderen C89- bzw. C99-Compilern nicht mehr ablauffähig ist.

Es gilt nämlich, dass ab dem ersten Vorkommen von /* bis zum ersten Auftreten von */ alles als Kommentar interpretiert wird. Für obige Angabe würde dies bedeuten, dass der fett gedruckte Teil vom Compiler als Kommentar interpretiert wird, so dass er dann mit dem Rest nichts mehr anfangen kann.

Kapitel 2

Elementare Datentypen

2.1 Die Grunddatentypen in C

In C existieren die in Abbildung 2.1 gezeigten elementaren Datentypen. Da ein Computer Zeichen – wie z. B. Buchstaben – ganz anders behandelt werden als Gleitpunktzahlen – wie z. B. die Zahl π=3.1415…–, wurde eine Klassifikation dieser unterschiedlichen Daten notwendig.

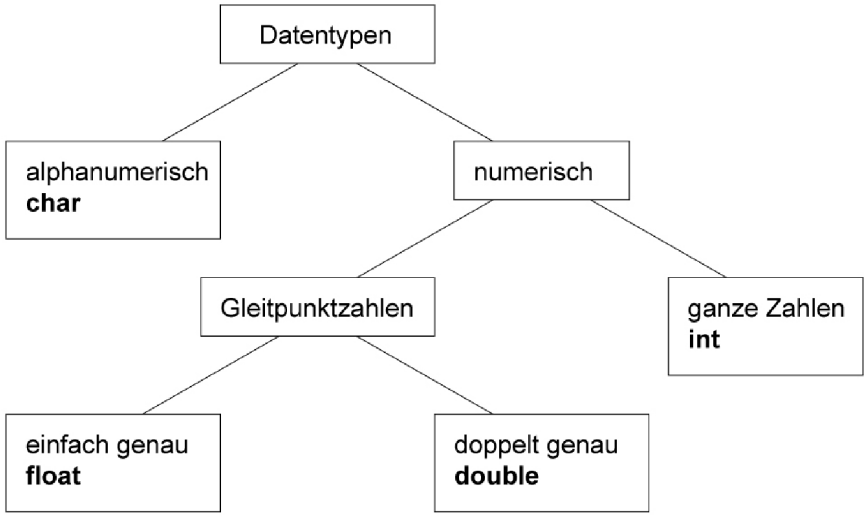


Abbildung 2.1: Die elementaren C-Datentypen

Ordnet man nun in einem Programm Daten bestimmten Klassen wie *Zeichen, ganze Zahl, einfach/doppelt genaue Gleitpunktzahl* usw. zu, dann teilt man dem Rechner deren *Datentyp* mit. In C existieren die Grunddatentypen: `char, int, float` und `double`, wobei es für die Typen `int` und `double` Versionen mit unterschiedlichen Genauigkeiten gibt:

char Daten dieses Typs belegen ein Byte Speicherplatz und repräsentieren „ Zeichen“. In einem Byte (8 Bit) kann genau ein Zeichen des ASCII-Zeichenvorrats gespeichert werden. Vor `char` darf dabei eines der beiden folgenden Schlüsselwörter angegeben werden:

signed	1. Bit ist Vorzeichenbit.
unsigned	1. Bit ist kein Vorzeichenbit.

int Dieser Datentyp repräsentiert „ganze Zahlen“; dafür sind im allgemeinen 4 Bytes vorgesehen, also viermal soviel Speicherplatz wie für `char`. Mit diesen 4 Bytes (32 Bit) können Zahlen im Bereich `-2 147 483 648` ... `+2 147 483 647` dargestellt werden.
Vor `int` darf dabei eines der folgenden Schlüsselwörter angegeben werden:

short Durch Voranstellen dieses Schlüsselworts (`short int`) werden typischerweise 2 Bytes (16 Bits) reserviert, was dem Zahlenbereich `-32 768 ...+32 767` entspricht.

long Durch Voranstellen dieses Schlüsselworts (`long int`) wird auf 32-Bit Maschinen üblicherweise ein Speicherplatz von 4 Bytes (32 Bit) reserviert. Auf 64-Bit Computern hat `long` aber normalerweise eine Genauigkeit von 64 Bit, also 8 Byte, was einem Wertebereich von etwa $\pm 9.22 \ldots \cdot 10^{18}$ entspricht.

long long (nur in C99) Ein `long long int` entspricht sowohl auf 32-, als auch auf 64-Bit Architekturen einer 8-Byte Genauigkeit. Dieser Datentyp bietet also für 32-Bit Computer die Möglichkeit, mit einem größeren Wertebereich zu arbeiten, als ihn die Prozessorregister bieten.

unsigned Das Voranstellen dieses Schlüsselworts (`unsigned int`) bewirkt, dass das erste Bit nicht als Vorzeichenbit, sondern als echte Stelle interpretiert wird. Mit `unsigned` kann also festgelegt werden, dass in einem Speicherplatz nur ganze positive, also keine negativen Zahlen stehen können.

Die obigen Schlüsselwörter können auch alleine, d. h. ohne `int`, angegeben werden, was vom C-Compiler genauso wie mit Angabe von `int` interpretiert wird: `short` entspricht `short int`, `long` entspricht `long int`, `long long` entspricht `long long int` und `unsigned` entspricht `unsigned int`.

float Dieser Datentyp ist für Gleitpunktzahlen mit einfacher Genauigkeit vorgesehen; dazu werden im allgemeinen 4 Bytes (32 Bit) reserviert.

double Daten dieses Typs belegen 8 Byte (64 Bit) Speicherplatz und sind Gleitpunktzahlen mit doppelter Genauigkeit. Wird `long double` angegeben, so bedeutet dies meist, dass ein Speicherplatz von 96 Bits (12 Bytes) reserviert wird.

Wie viele Bytes ein bestimmter Datentyp auf einer gegebenen Architektur tatsächlich belegt, kann mit dem folgenden Programm ermittelt werden (Beispiel für `long double`):

```
#include <stdio.h>
int main(void)
{
    printf( "Der Datentyp belegt %d Bytes \n", sizeof(long double) );
    return 0;
}
```

2.2 Wertebereiche für die einzelnen Datentypen

Tabelle 2.1 zeigt für die einzelnen Datentypen die auf 32-Bit Maschinen üblicherweise verwendeten Bitzahlen und die daraus resultierenden Wertebereiche. Tabelle 2.2, welche die äquivalente Information für 64-Bit Architekturen enthält, verdeutlicht, dass sich auf 64-Bit Computern lediglich der Wertebereich des Datentyps `long` ändert.

Tabelle 2.1: Typische Wertebereiche für die einzelnen Datentypen auf 32-Bit-Architekturen

Datentyp-Bezeichnung	Bitanzahl	Wertebereich
`char`, `signed char`	8	-128 .. 127
`unsigned char`	8	0 .. 255
`short`, `signed short`, `short int`, `signed short int`	16	-32 768 .. 32 767
`unsigned short`, `unsigned short int`	16	0 .. 65 535
`int`, `signed`, `signed int`	32	-2 147 483 648 .. 2 147 483 647
`unsigned`, `unsigned int`	32	0 .. 4 294 967 295
`long`, `signed long`, `long int`, `signed long int`	32	-2 147 483 648 .. 2 147 483 647
`unsigned long`, `unsigned long int`	32	0 .. 4 294 967 295
`long long`, `signed long long` (C99) `long long int`, `signed long long int` (C99)	64	-9 223 372 036 854 775 808 .. .. 9 223 372 036 854 775 807
`unsigned long long` (C99) `unsigned long long int` (C99)	64	0 .. 18 446 744 073 709 551 615
`float`	32	$1.2 \cdot 10^{-38}$..$3.4 \cdot 10^{38}$
`double`	64	$2.2 \cdot 10^{-308}$..$1.8 \cdot 10^{308}$
`long double`	96	$3.4 \cdot 10^{-4932}$..$1.1 \cdot 10^{4932}$

Tabelle 2.2: Typische Wertebereiche für die einzelnen Datentypen auf 64-Bit-Architekturen

Datentyp-Bezeichnung	Bitanzahl	Wertebereich
`int`, `signed`, `signed int`	32	-2 147 483 648 .. 2 147 483 647
`long`, `signed long` **`long int`, `signed long int`**	**64**	**-9 223 372 036 854 775 808 ..** **.. 9 223 372 036 854 775 807**
`unsigned long`, `unsigned long int`	**64**	**0 .. 18 446 744 073 709 551 615**
`unsigned long long`, (C99) `unsigned long long int` (C99)	64	0 .. 18 446 744 073 709 551 615
`long long`, `signed long long` (C99) `long long int`, `signed long long int` (C99)	64	-9 223 372 036 854 775 808 .. .. 9 223 372 036 854 775 807

Kapitel 3

Konstanten

Die Eigenschaft von Konstanten ist, dass sie – im Gegensatz zu Variablen – einen festen Wert besitzen, der einem der gerade besprochenen Datentypen zugeordnet ist. Jede Konstante hat dabei einen Datentyp, welcher durch ihre Form und Wert bestimmt wird (engl. *self-typing*).

3.1 char-Konstanten

Der Datentyp `char` wird meist durch 1 Byte realisiert. `char`-Konstanten müssen mit einfachem Hochkomma (z. B.: 'a') geklammert werden. Im Byte des `char`-Datentyps wird dann der zugehörige ASCII-Wert gespeichert, der dieses Zeichen repräsentiert, wie z. B.:

`char`-Konstante	Dezimaler ASCII-Wert	Dualdarstellung im **`char`**-Byte
'a'	97	01100001
'W'	87	01010111
'*'	42	00101010
'8'	56	00111000

Bei `char` handelt es sich anders als bei `float` und `double` um einen Ganzzahltypen. C ist nun bei weitem nicht so typstreng wie z. B. PASCAL. So ist es in C ohne weiteres möglich, im `char`-Datentyp auch einen numerischen Wert zu speichern. Es ist dabei lediglich zu beachten, dass bei einem numerischen Wert dann nicht der ASCII-Wert wie bei einer `char`-Konstante, sondern die Dualdarstellung dieser Zahl abgespeichert wird. Während also z. B. für `'8'` der ASCII-Wert `56`, der dieses Zeichen repräsentiert als Dualzahl (`00111000`) abgelegt wird, wird dagegen für die Dezimalzahl 8 die dieser Zahl entsprechende Dualzahl (`00001000`) im `char`-Datentyp abgelegt.

3.2 Ganzzahlige Konstanten

Es gibt drei verschiedene Arten von ganzzahligen Konstanten:

- *Dezimal-Konstanten*, die immer mit einer von `0` verschiedenen Ziffer beginnen
- *Oktal-Konstanten*, die immer mit einer `0` beginnen müssen
- *Hexadezimal-Konstanten*, die immer mit `0x` oder `0X` beginnen müssen.

Kapitel 4

Variablen

4.1 Variablen und die C-Regeln für Variablennamen

Die Eigenschaft von Variablen ist, dass sich ihre Werte ständig ändern können. Es handelt sich bei Variablen um eine Art Beschriftung eines Speicherbereichs. Während aber die Beschriftung immer gleich bleibt, kann sich der Inhalt eines Speicherbereichs ändern.

Die *Beschriftung eines Speicherplatzes*, d. h. die Vergabe eines Namens an eine Variable, wird vom Programmierer vorgenommen. Natürlich muss festgelegt werden, um welche Art von Variable (siehe auch Datentypen in Kapitel 2 ab Seite 7) es sich handelt. Diese Festlegung wird ebenfalls vom Programmierer vorgenommen; wie das geschieht, wird im nächsten Kapitel gezeigt. Um nun einen Speicherplatz zu „beschriften“, muss diesem ein Name, der *Variablenname*, gegeben werden. Die Vergabe von Variablennamen ist in C an bestimmte Regeln gebunden:

1. Es dürfen nur Buchstaben (keine Umlaute ä, ü, ö, Ä, Ü, Ö oder ß), Ziffern und der Unterstrich (_) verwendet werden.
2. Das erste Zeichen eines Variablennamens muss immer ein Buchstabe oder ein Unterstrich sein.
3. Ein Variablenname darf beliebig lang sein.
4. Wie jede höhere Programmiersprache verfügt auch C über einen festen Wortschatz von Befehlen (siehe unten). Solche Befehle dürfen natürlich nicht als Variablennamen verwendet werden, weil sonst der C-Compiler nicht entscheiden könnte, ob ein Befehl oder ein Variablenname gemeint ist.
5. Obwohl nach der ersten Regel erlaubt, sollten Variablennamen niemals mit einem Unterstrich beginnen, da es eine Reihe intern vordefinierter Namen dieser Form gibt.

Wie oben erwähnt, dürfen C-Schlüsselwörter nicht als Variablennamen verwendet werden. In folgender Tabelle sind alle C-Schlüsselwörter angegeben, wobei die in C99 neu hinzugekommenen Schlüsselwörter fett gedruckt sind. Während die Verwendung von Schlüsselwörtern als alleinstehende Variablennamen verboten ist, dürfen sie jedoch in Variablennamen eingebettet sein; z. B. wäre `automobil` ein erlaubter Variablenname.

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

Kapitel 5

Ausdrücke und Operatoren

5.1 Der einfache Zuweisungsoperator

5.1.1 Allgemeines zum einfachen Zuweisungsoperator

Der Zuweisungsoperator hinterlegt einen Wert in einer Variablen. Durch die Anweisung

```
zahl_var = 5;
```

wird z. B. in der Variablen `zahl_var` der Wert `5` abgelegt. Soll nun der Wert `-17` in `zahl_var` gespeichert werden, so ist lediglich

```
zahl_var = -17;
```

zu schreiben, wodurch der alte Wert von `zahl_var` überschrieben wird. Der Operator = ist *nicht* mathematisch als Gleichheitszeichen zu verstehen, sondern mehr im Sinne von „ergibt sich aus“:

```
zaehler = 19;
```

Der Wert der Variablen `zaehler` ergibt sich aus `19`, d. h.: im Speicherplatz mit dem Namen `zaehler` wird der Wert `19` gespeichert.

Beispiel:

In der Variablen `zahl_1` sei der Wert `4` und in der Variablen `zahl_2` der Wert `-24` gespeichert. Durch die Zuweisung:

```
zahl_1 = zahl_2;  /* Speichere in zahl_1 den Wert von zahl_2) */
```

gilt dann Folgendes:

Vor dieser Zuweisung:

zahl_1	4
zahl_2	-24
	...

Nach dieser Zuweisung:

zahl_1	~~4~~ -24
zahl_2	-24
	...

Der ursprüngliche Inhalt von `zahl_1` wurde mit dem Inhalt der Variablen `zahl_2` überschrieben.

Beispiel:

Das C-Programm `tausch.c` vertauscht die Werte von zwei `int`-Variablen:

```
#include  <stdio.h>

int main(void)
{
      int variab_1, variab_2, hilf;  /* Deklaration von drei ganzzahligen Variablen mit den Namen
                                        'variab_1', 'variab_2', 'hilf'   */
      variab_1 = 105;                /*          _________________
                                          variab_1 I     105     I
                                                   I_____________I      */
      variab_2 = -14;                /*          _________________
                                          variab_1 I     105     I
                                                   I_____________I
                                          variab_2 I     -14     I
                                                   I_____________I      */

      /************************************* Vertauschen *********************************/
      /* Zwischenspeichern der Werte von  'variab_1' in der "Hilfsvariable" 'hilf'  */
      hilf = variab_1;               /*          _________________
                                          variab_1 I     105     I
                                                   I_____________I
                                          variab_2 I     -14     I
                                                   I_____________I
                                          hilf     I     105     I
                                                   I_____________I      */
      /* Speichern des Wertes von 'variab_2' in 'variab_1', (d. h. der urspruengliche
          Wert von 'variab_1' wird ueberschrieben).
          Der urpruengliche Wert von  'variab_1' wurde zuvor in  'hilf' gesichert             */
      variab_1 = variab_2;           /*          _________________
                                          variab_1 I     -14     I
                                                   I_____________I
                                          variab_2 I     -14     I
                                                   I_____________I
                                          hilf     I     105     I
                                                   I_____________I      */
      /* Speichern des in 'hilf' gespeicherten Wertes in 'variab_2'. Nach dieser Zuweisung sind
         die urspruenglichen Werte von 'variab_1' und 'variab_2' vertauscht.                 */
      variab_2 = hilf;               /*          _________________
                                          variab_1 I     -14     I
                                                   I_____________I
                                          variab_2 I     105     I
                                                   I_____________I
                                          hilf     I     105     I
                                                   I_____________I     */
   return 0;
}
```

5.5.5 Bit-Operatoren nur für ganzzahlige Datentypen erlaubt

Die Bit-Operatoren `~`, `&`, `|` und `^` dürfen nicht auf Operanden angewendet werden, die vom Datentyp `float` oder `double` sind. Z. B. würde für das folgende C-Programm `bitgleit.c` ein Fehler gemeldet:

```
#include  <stdio.h>

int main(void) {
   int x;
   float zahl = 10*23;
   x = zahl & 200; /* zahl vom Typ float --> keine Bit-Operatoren dafuer erlaubt */
   printf("%d\n", x);
   return 0;
}
```

5.5.6 Fallgruben

Logische Operatoren und Bit-Operatoren unterscheiden sich

Das Ergebnis einer logischen Operation ist `FALSE` (in C: 0) oder `TRUE` (in C: 1). Bei den Bit-Operatoren dagegen werden die einzelnen Bits verglichen, und das Ergebnis ist dann das Bitmuster, und nicht unbedingt 0 oder 1. Die nachfolgenden Beispiele sollen dies verdeutlichen. Hat z. B. die `short`-Variable `z1` den Wert `2` und die `short`-Variable `z2` den Wert `5`, so haben die beiden Operatoren `&` und `&&` unterschiedliche Auswirkungen:

`z = z1 & z2;`

	Vorzeichen	= 16384 2^{14}	= 8192 2^{13}	= 4096 2^{12}	= 2048 2^{11}	= 1024 2^{10}	= 512 2^{9}	= 256 2^{8}	= 128 2^{7}	= 64 2^{6}	= 32 2^{5}	= 16 2^{4}	= 8 2^{3}	= 4 2^{2}	= 2 2^{1}	= 1 2^{0}	
z1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**= 2**
z2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	**= 5**
z1 & z2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**= 0**

`z = z1 && z2;`

	Vorzeichen	= 16384 2^{14}	= 8192 2^{13}	= 4096 2^{12}	= 2048 2^{11}	= 1024 2^{10}	= 512 2^{9}	= 256 2^{8}	= 128 2^{7}	= 64 2^{6}	= 32 2^{5}	= 16 2^{4}	= 8 2^{3}	= 4 2^{2}	= 2 2^{1}	= 1 2^{0}	
z1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**= 2** (TRUE)
z2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	**= 5** (TRUE)
z1 && z2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	**= 1** (TRUE)

TRUE && TRUE = TRUE (in C durch 1 dargestellt)

Hat z. B. die `short`-Variable `z1` den Wert `2` und die `short`-Variable `z2` den Wert `5`, so haben die beiden Operatoren | und || unterschiedliche Auswirkungen:

z = z1 | z2;

	Vorzeichen	= 16384 2^{14}	= 8192 2^{13}	= 4096 2^{12}	= 2048 2^{11}	= 1024 2^{10}	= 512 2^9	= 256 2^8	= 128 2^7	= 64 2^6	= 32 2^5	= 16 2^4	= 8 2^3	= 4 2^2	= 2 2^1	= 1 2^0	
z1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**= 2**
z2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	**= 5**
z1 | z2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	**= 7**

z = z1 || z2;

	Vorzeichen	= 16384 2^{14}	= 8192 2^{13}	= 4096 2^{12}	= 2048 2^{11}	= 1024 2^{10}	= 512 2^9	= 256 2^8	= 128 2^7	= 64 2^6	= 32 2^5	= 16 2^4	= 8 2^3	= 4 2^2	= 2 2^1	= 1 2^0	
z1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**= 2** (TRUE)
z2	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	**= 5** (TRUE)
z1 || z2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	**= 1** (TRUE)

TRUE || TRUE = TRUE (in C durch 1 dargestellt)

Niedrige Priorität der dyadischen Bit-Operatoren

Die Bitoperatoren lassen sich – wie in Tabelle 5.2 gezeigt - in die Prioritätstabelle der bisher kennengelernten Operatoren einordnen.

In Tabelle 5.2 ist erkennbar, dass die Priorität der Bitoperatoren sehr niedrig ist. Dies führt häufig zu äußerst ärgerlichen und schwer auffindbaren Fehlern in der praktischen Programmierung.

( )	
! ~ (Tilde) + (Vorz.) − (Vorz.)	höhere
* / %	Priorität
+ −	
< <= > >=	
== !=	
&	
^	
|	
&&	niedrigere
||	Priorität
=	

Tabelle 5.2: Priorität der Bit-Operatoren

Kapitel 6

Symbolische Konstanten

Die Programmiersprache C sieht die Möglichkeit vor, an Konstanten Namen zu vergeben. Wie wir im nachfolgenden sehen werden, bringt die Verwendung von Konstanten viele Vorteile mit sich, wie z. B. bessere Lesbarkeit von C-Programmen, Arbeitsersparnis beim Programmieren und leichtere Änderbarkeit von Programmen. In C können auf zwei verschiedene Arten symbolische Namen an Konstanten vergeben werden:

- Definition mit `#define` oder
- Definition unter Verwendung des Schlüsselworts `const`.

6.1 Konstanten-Definition mit #define

6.1.1 Die Direktive #define

Nehmen wir einmal an, dass Sie ein Programm erstellen sollen, in dem die Zahl π immer mit 9 Stellen Genauigkeit nach dem Komma verwendet werden muss. Sie müssten dann bei jedem Vorkommen von π die Zahlenschlange `3.141592654` eintippen. Mit der Verwendung von `#define` ist es nun jedoch möglich, am Anfang des Programms einem Zahlenwert einen Namen zuzuordnen, wie z. B.

```
#define PI 3.141592654
```

Anstelle des Zahlenwerts `3.141592654` können Sie dann bei den entsprechenden Anweisungen im gesamten Programm den symbolischen Namen `(PI)` verwenden. Dies kann eine erhebliche Arbeitsersparnis für das restliche Programm bedeuten, da nicht ein ständiges Nachschlagen und konzentriertes Abtippen der 10 Ziffern notwendig ist, sondern anstelle dessen nur der Name `PI` anzugeben ist.

Hinweis:

`#define`-Angaben werden nicht wie die anderen C-Anweisungen mit Semikolon abgeschlossen.

Kapitel 7

Ein- und Ausgabe

Bevor wir die beiden Bibliotheks-Funktionen `printf()` und `scanf()` in diesem Kapitel ausführlich behandeln, werden wir uns zunächst mit den sogenannten Headerdateien beschäftigen.

7.1 Headerdateien und #include

7.1.1 Bibliotheken und Headerdateien

Zu jedem C-Compiler wird eine Bibliothek mitgeliefert, in der Funktionen hinterlegt sind. Wie der Aufbau einer Funktion ist, können wir an `main()` sehen:

Funktionskopf, evtl. mit Parametern
Funktionsrumpf

```
int main(void)
{
...
...
...
}
```

Der Anfang des Funktionsrumpfs wird durch `{` und das Ende durch `}` gekennzeichnet.

Nun haben wir aber in den vorherigen Programmen bereits Funktionsaufrufe wie `printf()` oder `scanf()` benutzt, zu denen der Funktionsrumpf fehlte. Diese Funktionen sind in einer Bibliothek hinterlegt, in der sich der Linker, nachdem kompiliert wurde, die zum entsprechenden Funktionsaufruf gehörige Funktion sucht und zu Ihrem Programm dazubindet. Sie können also z. B. die Funktion `printf()` benutzen und Bildschirmausgaben veranlassen, obwohl Sie den dazugehörigen Programmteil (Funktion) nicht selbst programmiert haben; dies hat der Compiler-Hersteller bereits für Sie getan.

Seit C89 sind die Funktionsnamen und deren Funktionalität, die jedes C-Kompilierungssystem anbieten muss, genau festgelegt. Diese Funktionen sind üblicherweise in einer Standardbibliothek hinterlegt. Diese Standardbibliothek ist im Prinzip eine Datei, die den für den Linker bindbaren Objekt-Code dieser Funktionen enthält.

Damit aber auch bereits dem C-Compiler Informationen zu solchen Standardfunktionen bekannt sind, existieren so genannte Headerdateien, in denen die Funktionsköpfe der entspre-

Kapitel 8

Datentypumwandlungen

8.1 Implizite Datentypumwandlungen

8.1.1 Der sizeof-Operator

Bevor wir uns genauer mit den impliziten Datentypumwandlungen beschäftigen, werden wir einen neuen Operator kennenlernen: den `sizeof`-Operator. Mit diesem Operator kann die Größe eines Datentyps, einer Variable oder eines Ausdrucks in Bytes ermittelt werden. So sind z. B. die folgenden Angaben möglich:

`sizeof(double)`	liefert Anzahl von Bytes, die vom Datentyp `double` belegt werden.
`sizeof(var)`	liefert Anzahl von Bytes, die von der Variable `var` belegt werden.
`sizeof(a+b)`	liefert Anzahl von Bytes, die zur Berechnung von `a+b` benötigt werden.

Um festzustellen, wie viele Bytes Ihr Compiler für die Grunddatentypen verwendet, kann z. B. das folgende Programm `typgroes.c` dienen.

```
#include <stdio.h>

int  main(void) {
   printf("%20s: %d Bytes\n", "char", sizeof(char));
   printf("%20s: %d Bytes\n", "signed char", sizeof(signed char));
   printf("%20s: %d Bytes\n", "unsigned char", sizeof(unsigned char));
   printf("----------------------------------------\n");
   printf("%20s: %d Bytes\n", "short", sizeof(short));
   printf("%20s: %d Bytes\n", "signed short", sizeof(signed short));
   printf("%20s: %d Bytes\n", "unsigned short", sizeof(unsigned short));
   printf("----------------------------------------\n");
   printf("%20s: %d Bytes\n", "int", sizeof(int));
   printf("%20s: %d Bytes\n", "signed int", sizeof(signed int));
   printf("%20s: %d Bytes\n", "unsigned int", sizeof(unsigned int));
   printf("----------------------------------------\n");
   printf("%20s: %d Bytes\n", "long", sizeof(long));
   printf("%20s: %d Bytes\n", "signed long", sizeof(signed long));
   printf("%20s: %d Bytes\n", "unsigned long", sizeof(unsigned long));
   printf("----------------------------------------\n");
```

Kapitel 9

Die Headerdateien <limits.h und <float.h>

9.1 <limits.h> - Grenzwerte von Ganzzahltypen

Diese Headerdatei definiert die Grenzwerte für die verschiedenen Ganzzahltypen. C legt für jeden dieser Werte nur einen Mindestwert fest, wie dies in Tabelle 9.1 gezeigt ist. Der absolute Betrag dieses Mindestwerts (mit gleichem Vorzeichen) darf von keinem Compiler unterschritten werden, der sich C89- bzw. C99-Compiler nennt.

Tabelle 9.1: Konstanten mit Mindestwert aus <limits.h>

Konstante	Mindestwert	Beschreibung
CHAR_BIT	8	maximale Bitanzahl für ein Byte
SCHAR_MIN	-127	Minimalwert für signed char
SCHAR_MAX	+127	Maximalwert für signed char
UCHAR_MAX	255	Maximalwert für unsigned char
CHAR_MIN	SCHAR_MIN oder 0[1]	Minimalwert für char
CHAR_MAX	SCHAR_MAX oder UCHAR_MAX[2]	Maximalwert für char
MB_LEN_MAX	1	maximale Bytes für Vielbyte-Zeichen (länderspezifische Zeichen wie z. B. japanische oder chinesische Zeichen)
SHRT_MIN	-32767	Minimalwert für short int
SHRT_MAX	+32767	Maximalwert für short int
USHRT_MAX	65535	Maximalwert für unsigned short
INT_MIN	-32767	Minimalwert für int
INT_MAX	+32767	Maximalwert für int
UINT_MAX	65535	Maximalwert für unsigned int
LONG_MIN	-2147483647	Minimalwert für long int
LONG_MAX	+2147483647	Maximalwert für long int
ULONG_MAX	4294967295	Maximalwert für unsigned long
LLONG_MIN	-9223372036854775807	(neu in C99) Minimalwert für long long int
LLONG_MAX	+9223372036854775807	(neu in C99) Maximalwert für long long int
ULLONG_MAX	18446744073709551615	(neu in C99) Maximalwert für unsigned long long

[1] Abhängig davon, ob char ein vorzeichenbehafteter oder vorzeichenloser Datentyp ist
[2] Abhängig davon, ob char ein vorzeichenbehafteter oder vorzeichenloser Datentyp ist

Kapitel 10

Anweisungen und Blöcke

Fügt man zu einem Ausdruck wie `schieb >>= 4` oder `scanf("%d", &zahl)` oder `zaehl = 7` oder `nieder--` ein Semikolon hinzu, so erhält man eine *C-Anweisung*, wie z. B. folgende Anweisungen:

```
schieb >>= 4;
scanf("%d", &zahl);
zaehl = 7;
nieder--;
```

Um mehrere Anweisungen zu einem Block zusammenfassen zu können, müssen die geschweiften Klammern { und } verwendet werden. Ein solcher Block wird dann wie eine einzelne Anweisung interpretiert, wie z. B.

```
{
   zahl1 = -7;
   zahl2 = 15;
   sum = zahl1+zahl2;
   printf("%d + %d = %d\n", zahl1, zahl2, sum);
}
```

Hinweise:

1. In C beendet ein Semikolon eine Anweisung, während z. B. in Pascal ein Semikolon ein Trennzeichen zwischen den einzelnen Anweisungen ist.
2. Wie wir später sehen werden, können in einem Block auch Variablen vereinbart werden.

Kapitel 11

Die if-Anweisung

11.1 Die zweiseitige if-Anweisung

Die `if`-Anweisung wird für Programmverzweigungen benötigt: `if` (deutsch *Wenn*) und `else` (deutsch *Sonst*). Die Syntax für die vollständige (zweiseitige) `if`-Anweisung ist:

if *(ausdruck)*
  anweisung1
else
  anweisung2
anweisung1 und *anweisung2* kann ein mit {} geklammerter Block von Anweisungen sein.

Der nach `if` in Klammern `( )` angegebene *ausdruck* ist meist eine Bedingung. Ist die Bedingung erfüllt bzw. der Wert dieses Ausdrucks von 0 verschieden, dann wird *anweisung1*, sonst *anweisung2* ausgeführt, denn es gilt ja in C immer:

- Der Wert 0 entspricht `FALSE` (falsch).
- Ein Wert verschieden von 0 entspricht `TRUE` (wahr).

Die Funktionsweise der `if`-Anweisung kann am besten durch einen so genannten Programmablaufplan veranschaulicht werden, wie er in Abbildung 11.1 gezeigt ist.

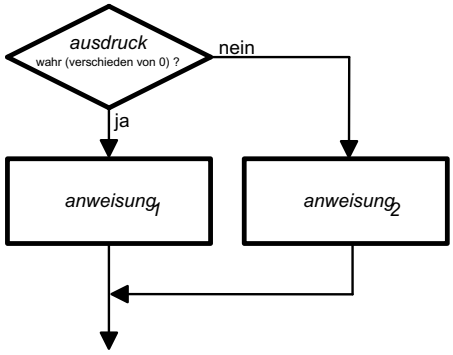


Abbildung 11.1: Programmablaufplan zur `if`-Anweisung

Kapitel 12

Die bedingte Bewertung ?:

Wenn wir das Maximum der beiden Variablen `zahl1` und `zahl2` an die Variable `max` zuweisen wollen, so würden wir mit unseren bisherigen Kenntnissen folgenden Programmteil erstellen:

```
   if (zahl1 > zahl2)
      max=zahl1;
   else
      max=zahl2;
```

Ist der Wert von `zahl1` größer als der Wert von `zahl2`, so wird der Wert von `zahl1` der Variablen `max`, andernfalls der Wert von `zahl2` der Variablen `max` zugewiesen.

Mit der bedingten Bewertung hätten wir diesen Programmteil auch wie folgt angeben können:

```
    max = (zahl1 > zahl2) ? zahl1 : zahl2;
```

Zunächst wird hier der Ausdruck (`zahl1 > zahl2`) bewertet. Trifft diese Bedingung zu, dann wird der Variablen `max` der Wert von `zahl1`, andernfalls der Wert von `zahl2` zugewiesen.

ausdr1 ? *ausdr2* : *ausdr3*

Zunächst wird dabei der *ausdr1* bewertet.
Ist sein Wert verschieden von 0, so wird *ausdr2* bewertet, andernfalls *ausdr3*.
In jedem Fall wird nur einer der beiden Ausdrücke *ausdr2* oder *ausdr3* bewertet;
dieser Wert ist dann das Ergebnis der gesamten bedingten Bewertung.

Da der Operator `? :` eine sehr niedrige Priorität besitzt, sind runde Klammern bei *ausdr1, ausdr2* und *ausdr3* meist nicht notwendig, erhöhen aber die Lesbarkeit.

Das folgende C-Programm `mannweib.c` zeigt eine Anwendung des bedingten Operators:

```
#include  <stdio.h>
int  main(void) {
    char   zeichen;
    printf("Geben Sie den Buchstaben   m  oder  w   ein: ");
    printf("----> Sie haben   ***%s***   gewaehlt\n",
           ((zeichen=getchar()) == 'm') ? ("maennlich") : ("weiblich"));
    return 0;
}
```

Kapitel 13

Die switch-Anweisung

13.1 Die switch-Anweisung

Mit der `switch`-Anweisung kann unter mehreren Alternativen, nicht nur unter zwei wie bei der `if`-Anweisung, ausgewählt werden. Die Syntax für die `switch`-Anweisung ist:

```
switch (ausdruck)
{
      case ausdr1:      anweisungen1
      case ausdr2:      anweisungen2
    ...
      case ausdrN:      anweisungenN
      default:          anweisungenD
}
```

Es wird der bei `switch` angegebene (*ausdruck*) ausgewertet und das Ergebnis mit den einzelnen `case`-Ausdrücken, die `int`- oder `char`- Werte (Konstanten) liefern müssen, verglichen.

Wird keine Übereinstimmung gefunden, so verzweigt das Programm zur `default`-Marke, falls diese angegeben wurde.

Wird keine Übereinstimmung gefunden und es ist keine `default`-Marke angegeben, so wird keine Anweisung des `switch`-Blocks ausgeführt. Fehlt also die `default`-Marke, so wird der gesamte `switch`-Block übersprungen, falls keine Übereinstimmung in den `case`-Marken vorliegt.

Die `switch`-Anweisung ist mit einem Programm-Schalter zu vergleichen und wird im Struktogramm - wie in Abbildung 13.1 gezeigt – dargestellt.

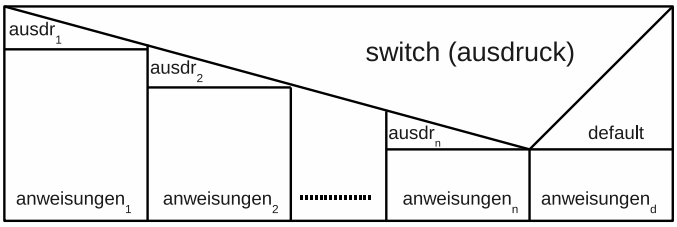


Abbildung 13.1: Struktogramm zur `switch`-Anweisung

Kapitel 14

Der Komma-Operator

14.1 Der Komma-Operator

Der Komma-Operator fasst mehrere Ausdrücke syntaktisch zu einem einzigen Ausdruck zusammen.

Sind mehrere Ausdrücke durch ein Komma (`,`) getrennt, so werden sie von links nach rechts bewertet; Datentyp und Wert des Gesamtausdrucks ist dann gleich dem Datentyp und Ergebnis der letzten Operation.

Beispiel:

`zeich = 'J', zaehl = 9, pi = 3.1415;`
: Alle drei Zuweisungen werden nacheinander berechnet, das Ergebnis des Gesamtausdruckes ist gleich `3.1415`.

Im Zusammenhang mit Schleifenvariablen, die wir im nächsten Kapitel behandeln werden, wird dieser Komma-Operator häufig verwendet. Der Komma-Operator erlaubt nämlich die Unterbringung von mehreren Zuweisungen an einer Stelle, wo sonst nur eine erlaubt wäre.

Mann kann den Komma-Operator auch innerhalb einer `if`-Anweisung verwenden, um sich die geschweiften Klammern einer Blockanweisung zu sparen, wie z. B.:

```
if (...)
   zaehl++, printf ("\n Zaehler erhoeht\n");
```

Sonst hätte man schreiben müssen:

```
if (...) {
   zaehl++;
   printf ("\n Zaehler erhoeht\n");
}
```

Aus Gründen der besseren Lesbarkeit ist aber immer die zweite Vorgehensweise vorzuziehen.

Kapitel 15

Die for-Anweisung

15.1 Die for-Anweisung

Syntaktisch sieht die `for`-Schleife wie folgt aus:

`for` (*ausdruck1; ausdruck2; ausdruck3)*
anweisung
Die *anweisung* kann ein Block von Anweisungen sein, der mit `{...}` zu klammern ist.
- *ausdruck1* initialisiert die Schleifenvariable(n).
- *ausdruck2* legt das Abbruchkriterium für die Schleife fest, wenn *ausdruck2* nicht erfüllt ist.
- *ausdruck3* reinitialisiert die Schleifenvariable(n).

Die Komponenten (*ausdrücke*) können einzeln oder auch insgesamt fehlen, jedoch müssen die Semikolons in der Klammer an den richtigen Stellen verbleiben.

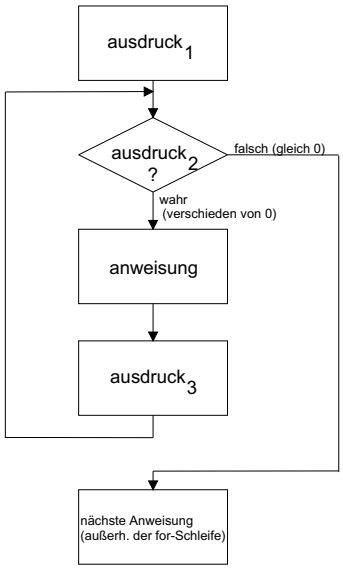


Abbildung 15.1: Programmablaufplan zur `for`-Schleife

Jetzt wollen wir nochmals die geometrische Reihe berechnen, wobei wir alle erforderlichen Schleifenanweisungen im `for`-Schleifenkopf unterbringen, so dass überhaupt keine Schleifenanweisung mehr übrigbleibt.

```
#include <stdio.h>

int  main(void)
{
   int     zaehl,  n;
   double  reih_teil,  summe;

   printf("Geben Sie n ein !\n");
   scanf("%d",&n);

   for (reih_teil = 1, summe = 1, zaehl = 1;
        zaehl <= n;
        reih_teil /= 2, summe += reih_teil, ++zaehl)
       ; /* leere Anweisung */

   printf("\n\n\nDie Summe der "
          "geometrischen Reihe bis %d ist:\n", n);
   printf("%lf\n", summe);
   return 0;
}
```

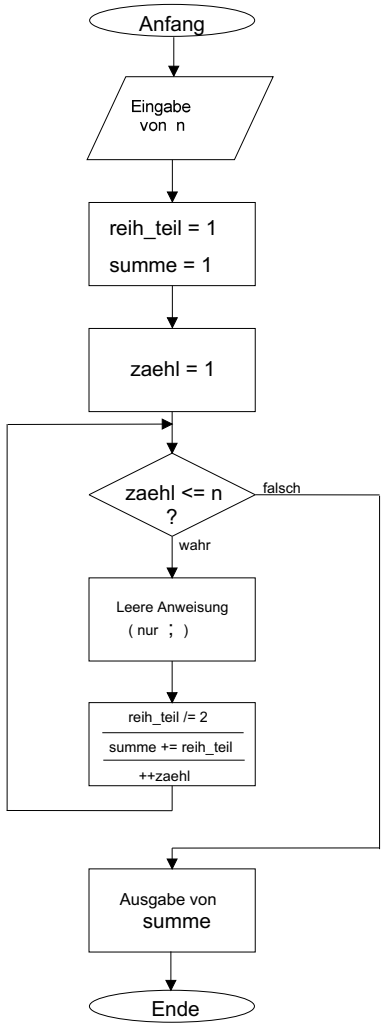


Abbildung 15.6: C-Programm `georeih3.c`

Hier haben wir als einzige Schleifenanweisung eine leere Anweisung. Für dieses Programm ist `reih_teil` mit dem Wert `1` vorzubesetzen.

Der Komma-Operator erlaubt also, mehrere Ausdrücke zu einem einzigen Ausdruck zu verschmelzen. Diese Technik lässt sich vor allen Dingen bei der `for`-Schleife anwenden, wenn mehrere Schleifenvariablen zu initialisieren und/oder zu reinitialisieren sind.

Generell sollte man jedoch darauf achten, den `for`-Schleifenkopf nicht zu überladen. Um die Übersichtlichkeit und Lesbarkeit eines Programms zu wahren, ist es des öfteren besser, Variablen in eigenen Anweisungen zu initialisieren oder zu verändern, wenn sie nicht unmittelbar zur Schleifensteuerung gehören.

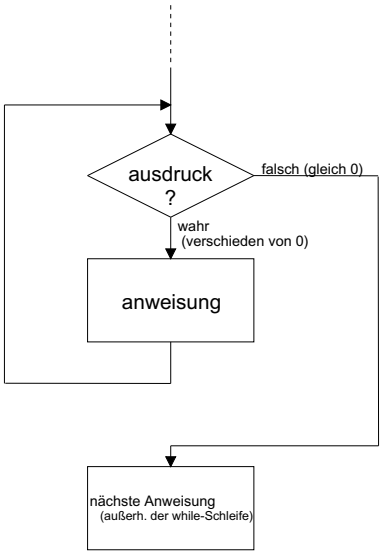
Kapitel 16

Die while-Anweisung

16.1 Die while-Anweisung

while (*ausdruck*)
    anweisung
anweisung kann wieder ein mit `{..}` zu klammernder Block von Anweisungen sein.

Die Funktionsweise der `while`-Schleife lässt sich am besten anhand der Abbildung 16.1 zeigen.



Vor jedem Schleifendurchlauf wird *ausdruck* berechnet. Solange das Ergebnis verschieden von 0 (TRUE) ist, wird die Schleifen-*anweisung* ausgeführt. Erst, wenn die Auswertung von *ausdruck* 0 (FALSE) liefert, wird die Schleife beendet und mit der nächsten nicht zur Schleife gehörigen Anweisung fortgefahren.
Ist das Ergebnis von *ausdruck* bereits bei der ersten Berechnung gleich 0 (FALSE), so wird die Schleife gar nicht durchlaufen, sondern sofort mit der nächsten nicht zur Schleife gehörigen Anweisung fortgefahren.

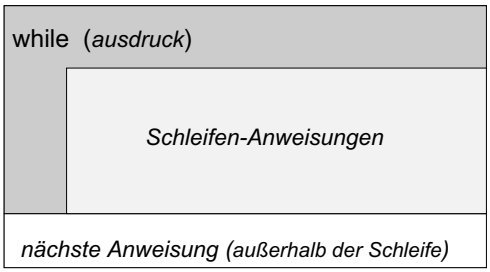


Abbildung 16.1: Programmablaufplan und Struktogramm zur `while`-Schleife

Da die Abfrage der Schleifen-Bedingung wie bei der `for`-Schleife am Anfang durchgeführt wird, wird die `while`-Schleife genauso wie die `for`-Schleife im Struktogramm dargestellt.

Kapitel 17

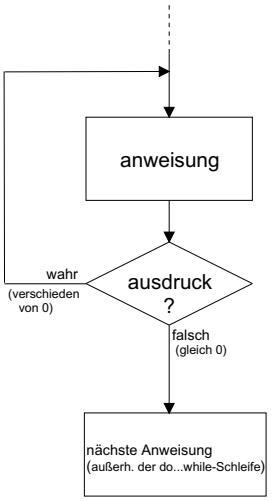
Die do. . . while-Anweisung

17.1 Die do. . . while-Anweisung

`do`
anweisung
`while` **(*ausdruck*);**

Die *anweisung* kann selbstverständlich wieder ein Block von Anweisungen sein, der dann mit `{..}` zu klammern ist.

Die Funktionsweise der `do...while`-Schleife lässt sich am besten anhand der Abbildung 17.1 zeigen.



Nach jedem Schleifendurchlauf wird *ausdruck* berechnet. Ist das Ergebnis verschieden von 0 (TRUE), so wird die Schleifen-*anweisung* nochmals ausgeführt. Erst, wenn die Auwertung von *ausdruck* 0 (FALSE) liefert, wird die Schleife beendet und mit der nächsten nicht zur Schleife gehörigen Anweisung fortgefahren.

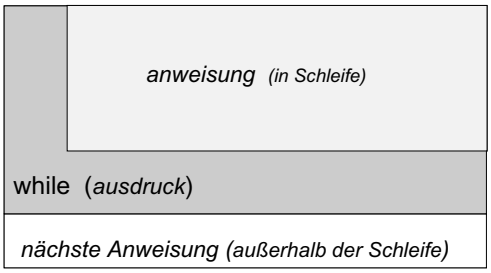


Abbildung 17.1: Programmablaufplan und Struktogramm zur `do...while`-Schleife

Da die Abfrage der Bedingung bei der `do...while`-Schleife am Ende stattfindet, wird diese Schleife im Struktogramm – wie rechts in Abbildung 17.1 gezeigt – dargestellt.

Kapitel 18

Die break-Anweisung

18.1 Die break-Anweisung

Nicht immer ist es sinnvoll, eine Schleife nur über das Abbruchkriterium zu verlassen. Die `break`-Anweisung dient dazu, um eine `for`-, `while`-, `do...while`-Schleife oder eine `switch`-Anweisung vorzeitig zu verlassen. Dazu muss lediglich `break` aufgerufen werden.

`for`-Schleife:

```
   for (ausdr1 ; ausdr2 ; ausdr3) {
      .....
      .....
 +----break;
 |    .....
 |    .....
 | }
 +->anweisung; /* ausserhalb for-Schleife */
```

`while`-Schleife:

```
   while (ausdruck) {
      .....
      .....
 +----break;
 |    .....
 |    .....
 | }
 +->anweisung; /* ausserhalb while-Schleife */
```

`do...while`-Schleife:

```
   do {
      .....
      .....
 +----break;
 |    .....
 |    .....
 | } while (ausdruck);
 +->anweisung    /* ausserhalb do...while */
```

`switch`-Anweisung:

```
   switch (ausdruck) {
       .....
      case x:
         ....
 +-------break;
 |     .....
 |     .....
 | }
 +->anweisung;    /* ausserhalb switch */
```

Kapitel 19

Die continue-Anweisung

19.1 Die continue-Anweisung

Die `continue`-Anweisung ist der `break`-Anweisung ähnlich. Sie bewirkt allerdings im Unterschied zur `break`-Anweisung nicht den Abbruch der gesamten Schleife, sondern nur den Abbruch des aktuellen Schleifendurchlaufs, also einen Sprung zum Schleifenende. `continue` leitet somit unverzüglich den nächsten Schleifendurchlauf ein und hat auch im Gegensatz zu `break` keine Auswirkung auf `switch`-Anweisungen.
Die `continue`-Anweisung bewirkt also eine sofortige Wiederholung der betreffenden Schleife. Bei `while` und `do...while` bedeutet dies, dass sofort wieder die Bedingung (*ausdruck*) ausgewertet wird. Bei `for` wird als nächstes die Reinitialisierung (*ausdr3*) durchgeführt und dann zur Schleifenbedingung (*ausdr2*) verzweigt.
Das folende C-Programm `vollkomm.c` gibt alle vollkommenen Zahlen zwischen `x` und `y` aus. Eine Zahl `z` ist vollkommen, wenn die Summe aller ihrer Teiler (ohne `z` selbst) gleich `z` ist.

```
#include <stdio.h>
int  main(void) {
    long int    sum, innen, aussen, von, bis;
    printf("Vollkommene Zahlen von? "); scanf("%ld", &von);
    printf("Vollkommene Zahlen bis? "); scanf("%ld", &bis);
    for (aussen = von; aussen <= bis; aussen++)  {
        sum = 0;
        for (innen = 1; innen <= aussen / 2; innen++) {
            if ((aussen % innen) != 0)
               continue;
            sum += innen;
            if (sum > aussen)
               break;
        }
        if (sum == aussen)
           printf("%ld\n",sum);
    }
    return 0;
}
```

Kapitel 20

Marken und die goto-Anweisung

20.1 Marken und die goto-Anweisung

Die Syntax der `goto`-Anweisung ist:

`goto` `marke;`
Diese Anweisung bewirkt einen Sprung zu der Programmstelle, an der
`marke:`
angegeben ist.

Eine `marke` hat dabei die gleiche Form wie ein Variablenname[1]; anschließend folgt ein Doppelpunkt:

```
marke: anweisung;
```

`marken` können vor jeder Anweisung stehen, unabhängig davon, ob sie innerhalb des Programms mit **goto** angesprungen werden oder nicht. Nach einer `marke` muss immer mindestens eine Anweisung angegeben sein, eventuell auch nur die leere Anweisung:

```
marke: ;
```

In einem Programm darf dieselbe Marke nur einmal angegeben sein.

20.2 Programmiertechniken

20.2.1 goto nur im äußersten Notfall

Mit Einführung der Strukturierten Programmierung in den 60er Jahren wurde die Springerei mit **goto** sehr verpönt. Damals waren Programme vor allen Dingen in Assembler geschrieben. Dort war und ist zum Teil noch der Sprung zu einer Code-Adresse ein fundamentales Prinzip. Später wurde dieser Stil vor allen Dingen in **BASIC** übernommen. Man stellte damals sehr schnell fest, dass Programme, die mit Vorwärts- und Rückwärtssprüngen gespickt sind, nicht nur schwer verständlich, sondern auch kaum wartbar waren. Dies führte zu einer erheblichen Verteuerung der Software. Ein Ausweg aus diesem Dilemma war die *Strukturierte Programmie-*

[1] muss mit Buchstabe oder Unterstrich beginnen und des weiteren dürfen nur Buchstaben, Ziffern oder Unterstriche folgen

Kapitel 21

Grafikprogrammierung unter Linux

Im Rahmen dieses Buches wurde eine einfache Grafik-Bibliothek namens LCGI entwickelt, welche von der Homepage heruntergelden werden kann.

21.1 Benutzung von LCGI

1. **Inkludieren von <graphics.h>**
   Da die hier vorgestellten Funktionen in der Headerdatei `graphics.h` deklariert sind, sollte immer Folgendes angegeben werden, wenn man von diesen Funktionen in seinem Programm Gebrauch macht.

   ```
   #include <graphics.h>
   ```

2. **Angabe von main() mit Parametern**
   Wenn Sie in Ihrem Programm Routinen aus diesem Kapitel aufrufen, müssen Sie statt

   ```
   int main(void)
   ```

   immer Folgendes angeben:

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

3. **Kompilieren und Linken von Grafikprogrammen**
   Wenn Sie ein Programm erstellt haben, das Routinen aus diesem Kapitel aufruft, müssen Sie dieses immer mit dem mitgelieferten Kommando `lcc` statt `gcc` bzw. `cc` kompilieren und linken.
   Wenn Sie z. B. ein Programm `polygon.c` entworfen haben und dieses nun kompilieren wollen, müssen Sie z. B. Folgendes aufrufen:

   ```
   user@linux:~ >   lcc -o polygon polygon.c ⏎
   ```

   `lcc` bietet die gleichen Optionen an wie das Kommando `gcc` bzw. `cc`.
   Geben Sie danach zum Testen die folgende Kommandozeile ein:

   ```
   user@linux:~ >  ./polygon ⏎
   ```

21.2 Grafikmodus ein- und ausschalten

`initgraph(int breite, int hoehe)`
: schaltet den Grafikmodus ein, indem ein Fenster eingeblendet wird, das `breite` Pixel breit und `hoehe` Pixel hoch ist.

`closegraph()`
: beendet den Grafikmodus. Da in diesem Fall auch das Grafikfenster gelöscht wird, wird meist vor `closegraph()` noch ein `getch()` angegeben, so dass erst auf einen Tastendruck des Benutzers hin der Grafikmodus verlassen wird.

Typisch für die Grafikprogrammierung ist deshalb z. B. folgender Programmausschnitt:

```
#include <graphics.h>
.......
int  main( int argc, char *argv[] )
{
    initgraph( 640, 480 );
       ........
         Grafik-Programmteil
       ........
    getch();  /* auf einen Tastendruck warten */
    closegraph();
}
```

21.3 Eingaben im Grafikmodus

Während man im Grafikmodus arbeitet, kann man nicht mehr die Standardroutinen für die Ein- und Ausgabe (`printf(), scanf(), getchar()` und `putchar()`) verwenden, sondern muss die eigens dafür angebotenen Routinen verwenden, welche nachfolgend vorgestellt sind:

`getcharacter(char *text, ...)`

`getstring(char *text, ...)`

`getint(char *text, ...)`

`getdouble(char *text, ...)`
: fordern den Benutzer durch Ausgabe des Textes `text` zur Eingabe eines Zeichens, eines Textes, einer ganzen Zahl oder einer Gleitpunktzahl auf und liefern diese Eingabe zurück.

`kbhit()`
: prüft, ob eine Taste gedrückt wurde. Falls eine Taste gedrückt wurde, liefert `kbhit()` einen `int`-Wert ungleich 0 (TRUE), falls nicht, liefert diese Funktion 0 (FALSE) zurück. Das dabei eingegebene Zeichen kann mit der nachfolgend vorgestellten Routine `getch()` nachträglich erfragt werden.

`getch()`
: liest ein Zeichen ein. Anders als bei `getchar()` findet hier jedoch keine Zwischenpufferung statt, sondern das Zeichen wird direkt von der Tastatur gelesen. Das eingegebene Zeichen wird dabei nicht am Bildschirm angezeigt. In jedem Fall liefert diese Routine

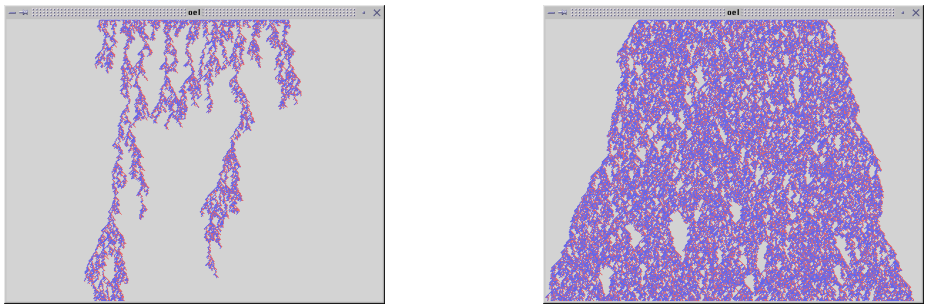


Abbildung 21.3: Wahrscheinlichkeit von 0.62 (links) und 0.7 (rechts) für das Eindringen von Öl

Beispiel:

Das folgende Programm `pixel.c` wählt zufällig eine Hintergrundfarbe und malt dann 1 000 kleine Vierecke an zufällige Bildschirmpositionen mit zufällig gewählten Farben. Ein Viereck sind dabei 9 Pixel:

```
xxx
xox
xxx
```

wobei nur der Mittelpunkt (o) zufällig gewählt ist. Danach wird der Inhalt dieses Fensters gelöscht, mit einer zufälligen Hintergrundfarbe gefüllt und wieder mit neuen zufälligen kleinen Rechtecken bemalt. Mit einem Tastendruck kann dieses Programm beendet werden.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <graphics.h>
#define PIXEL_ZAHL    1000
int  main( int argc, char *argv[] ) {
   int i, maxx, maxy, x, y, farbe, max_farbe;
    srand(time(NULL));
    initgraph( 640, 480 );
    maxx=getmaxx(); maxy=getmaxy(); max_farbe = getmaxcolor();
    while ( !kbhit()) {
       cleardevice( rand()%(max_farbe+1) );
       for (i=1 ; i<=PIXEL_ZAHL ; i++) {
          x = rand()%maxx;   y = rand()%maxy; farbe = rand()%(max_farbe+1);
          putpixel(x, y, farbe);   putpixel(x-1,y-1,farbe); putpixel(x,y-1,farbe);
          putpixel(x+1,y-1,farbe); putpixel(x+1,y,farbe);   putpixel(x+1,y+1,farbe);
          putpixel(x,y+1,farbe);   putpixel(x-1,y+1,farbe); putpixel(x-1,y,farbe);
       }
    }
    closegraph();
    return 0;
}
```

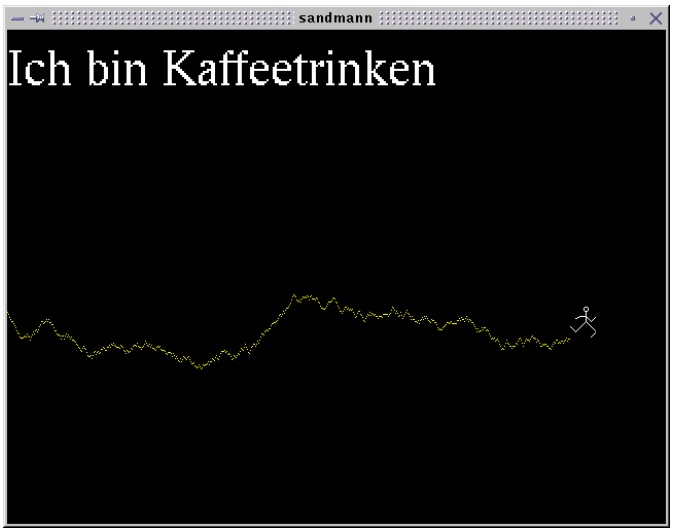


Abbildung 21.10: Anzeige des Programms `sandmann.c`

Abbildung 21.10 zeigt das von Programm `sandmann.c` angezeigte Fenster zu einem bestimmten Zeitpunkt. Dieses Programm könnte z. B. gestartet werden, wenn man seinen Bildschirmplatz längere Zeit verlässt, und man Information am Bildschirm darüber ausgeben lassen möchte, wo man sich gerade befindet und wann man wieder zurückkommt. Hierzu müsste man jedoch dann die auszugebende Meldung einlesen lassen. Eine entsprechende Anpassung dieses Programms an die speziellen eigenen Bedürfnisse sollte aber nicht allzu schwierig sein.

Beispiel:

Das folgende Programm `karten.c` blendet bei jedem Tastendruck immer sieben zufällig ausgewählte Karten ein, die es übereinander legt (siehe auch Abbildung 21.11). Mit der `ESC`-Taste kann dieses Programm beendet werden.

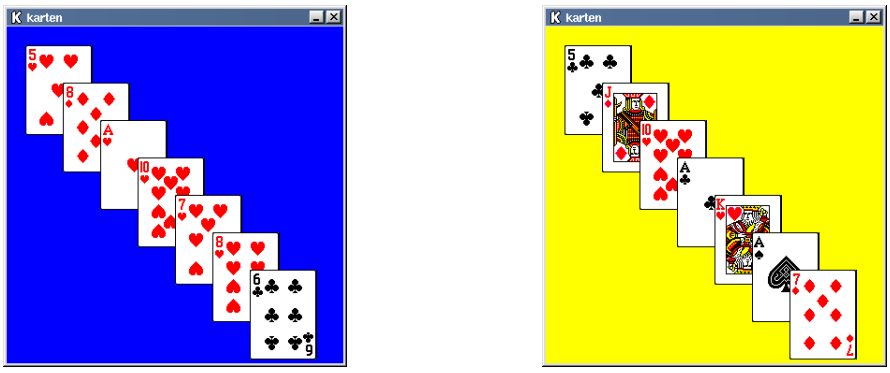


Abbildung 21.11: Unterschiedliche Anzeigen des Programms `karten.c` nach einem Tastendruck

Kapitel 22

Funktionen

In der Programmierpraxis sind häufig Probleme anzutreffen, die einander von der Aufgabenstellung her ähnlich sind. Dieser Tatsache wird in der Programmiersprache C mit so genannten Funktionen (*functions*) Rechnung getragen.

22.1 Allgemeines zu Funktionen

Wir haben bisher schon – ohne es ausführlich zu erwähnen – mit Funktionen gearbeitet. So haben wir z. B. zur Ausgabe auf dem Bildschirm die Funktion `printf()` aufgerufen. Hinter diesem Funktionsnamen verbirgt sich ein Programmteil, der in der Standardbibliothek definiert ist und bei jedem Aufruf von `printf()` ausgeführt wird.

22.1.1 Allgemeines Beispiel zu Funktionen

Am besten können wir das Prinzip von Funktionen an einem allgemeinen Beispiel nachvollziehen. In diesem Beispiel wird nicht die Programmiersprache C, sondern – zum besseren Verständnis – die deutsche Sprache verwendet. Es soll bei der Personalabteilung einer Firma immer ein einheitliches Papier für die Spesenabrechnung abgegeben werden. Wir nehmen dazu an, dass in der Standardbibliothek eine Funktion (Schablone) mit Namen `spesen()` definiert sei, was natürlich nicht für die wirkliche Programmbibliothek gilt.

```
  Funktionsname                 Liste der formalen Parameter
   |                                 |
   |                                 v
   v    ____________________________________________________________________
spesen(  name, stadt, preis, vor, haupt, nach, tagzeit, waehrung, datum )
{
    Drucke folgendes aus:
       "Herr/Frau  name hat am  datum in  stadt
        zu(m)  tagzeit:
             vor als Vorspeise
             haupt als Hauptspeise und
             nach als Nachspeise
        gegessen.
        Der Preis fuer dieses Essen betrug  preis waehrung."
}
```

In der Definition von `tausch()` werden mit der Deklaration

```
void  tausch(int *a, int *b)
```

zwei Speicherplätze für `int`-Zeiger und ein Speicherplatz für die lokale Variable `hilf` reserviert.

Mit dem Aufruf

```
   tausch(&zahl_1, &zahl_2);
```

werden dabei dann die Adressen von `zahl_1` und `zahl_2` als aktuelle Argumente an die Funktion `tausch()` übergeben; als Erinnerung: `&` = Adreßoperator. Bei dieser Übergabe werden also in die beiden reservierten Speicherplätze `a` und `b` der Funktion `tausch()` die Adressen von `zahl_1` und `zahl_2` abgelegt; siehe Abbildung 22.11.

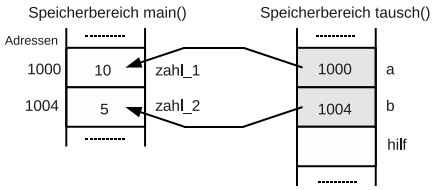
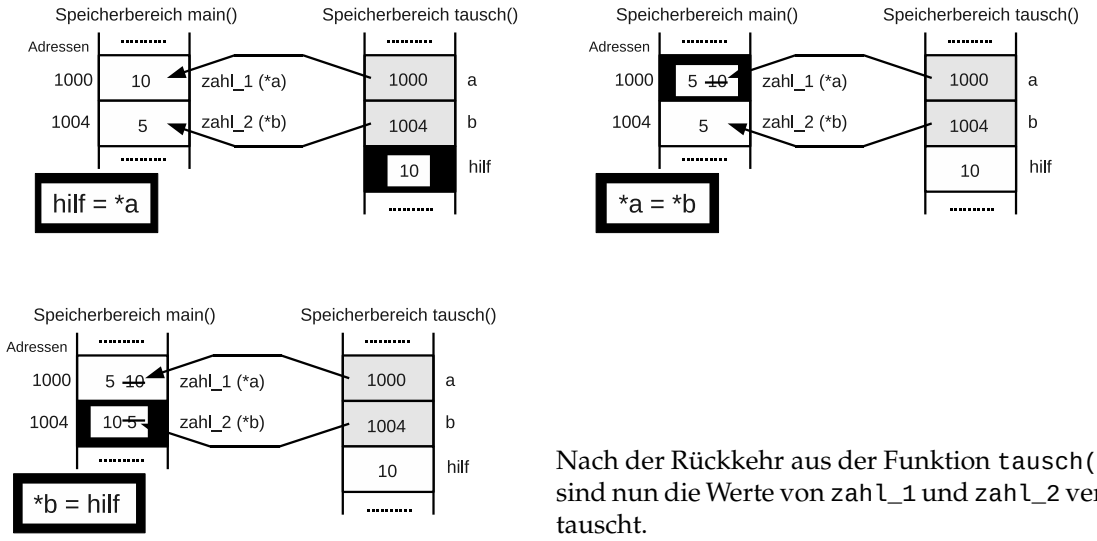


Abbildung 22.11: Übergabe der Adressen von `zahl_1` und `zahl_2` an `tausch()`

Der Ablauf innerhalb der Funktion `tausch()` ist bildlich in der Abbildung 22.12 dargestellt.

```
   hilf = *a;
   *a  = *b;
   *b  = hilf;
```



Nach der Rückkehr aus der Funktion `tausch()` sind nun die Werte von `zahl_1` und `zahl_2` vertauscht.

Abbildung 22.12: Call by Reference durch Übergabe von Adressen

und

```
float *welch_funktion(...);
     /* welch_funktion = Funktion, die Zeiger auf float-Wert liefert */
```

sollten Sie sich klarmachen.

Will man nun über den Zeiger `welch_funktion` eine Funktion aufrufen, so muss man `*welch_funktion` angeben, da `welch_funktion` ein Zeiger und folglich `*welch_funktion` die eigentliche Funktion ist.

Wenn z. B. `welch_funktion` auf die Funktion `zyl_vol()` zeigt, dann entspricht die Anweisung

```
return rad*(*welch_funktion)(rad);
```

der Anweisung

```
return rad * zyl_vol(rad);
```

wie dies auch in Abbildung 22.20 gezeigt ist.

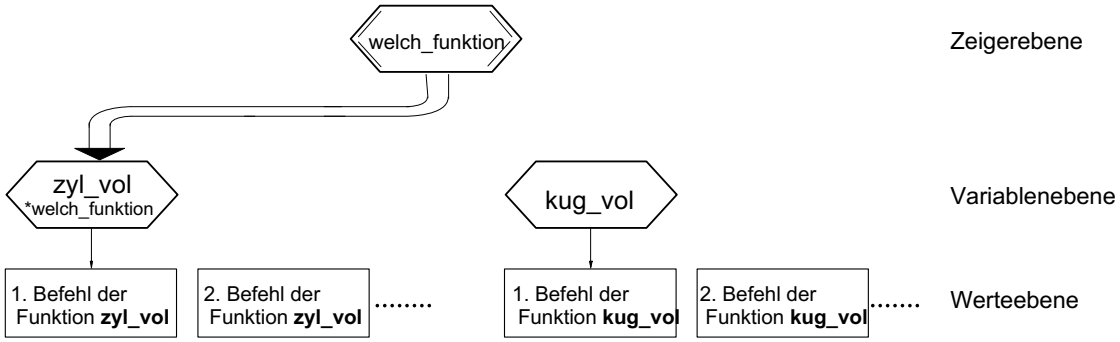


Abbildung 22.20: `*welch_funktion` entspricht `zyl_vol()`

Um das Zeigerprinzip bei Funktionen besser verstehen zu können, wollen wir ein weiteres Deklarations-Beispiel angeben:

```
float *(*welch_funktion)(...);
```

Hier wäre `welch_funktion` ein Zeiger auf eine Funktion, die einen Zeiger auf einen `float`-Wert liefert.

22.7.2 Typische Anwendungen

Auswahl von Funktionen zur Programmlaufzeit

Ein typischer Anwendungsfall für Zeiger auf Funktionen liegt vor, wenn erst zum Zeitpunkt der Programmausführung bekannt ist, welche Funktion auszuführen ist. Nehmen wir z. B. ein Sortierprogramm, bei dem erst während der Programmausführung bekannt ist, welche Daten zu sortieren sind: Ganze Zahlen, Zeichenketten oder sonstige Daten. In diesem Fall schreibt man nur eine Sortierfunktion, die neben anderen Parametern auch einen Zeiger auf eine Funktion erwartet, die für den Vergleich der einzelnen Daten zuständig ist. Ist dann während der

Kapitel 23

Speicherklassen und Modultechnik

Es wurde schon einmal erwähnt, dass sich C-Programme aus mehreren Dateien zusammensetzen können. Jede Programmdatei – auch *Modul* genannt – kann einzeln kompiliert werden, aber ist für sich allein nicht lauffähig.

Diese getrennt kompilierten Module können anschließend mit Hilfe des Linkers zu einem einzigen Programm zusammengebunden werden.

23.1 Gültigkeitsbereich, Lebensdauer, Speicherort

23.1.1 Gültigkeitsbereich

Unter dem Gültigkeitsbereich eines Objekts (*Funktionsname, Variablenname*) versteht man die Bereiche in den einzelnen Programmeinheiten, in denen es angesprochen werden kann.

Beim Gültigkeitsbereich wird zwischen folgenden Arten unterschieden:

- modulglobal
- programmglobal
- lokal (in einem Block von { . . . })

Modulglobale Gültigkeit

Ein *modulglobales* Objekt ist innerhalb der gesamten Programmdatei (Modul) ansprechbar, in der es deklariert ist, aber nicht in einer anderen Programmdatei (Modul). Jede Variable, die außerhalb einer Funktion deklariert ist, werden wir als *modulglobal* bezeichnen.

Da modulglobale Variablen außerhalb von Funktionen vereinbart sind und somit vielen Funktionen zur Verfügung stehen, kann innerhalb von Funktionen auf modulglobale Variablen zugegriffen werden, ohne dass diese als Parameter zu übergeben sind.

Beispiel:

In diesem Beispiel werden wir ein C-Programm `tausch3.c` erstellen, das mit Hilfe einer Funktion `tausch()` die Werte zweier Variablen vertauscht. In diesem Programm werden die entsprechenden Variablen jedoch nicht als Zeigervariablen übergeben.

Kapitel 24

Präprozessor-Direktiven

Der Präprozessor verarbeitet den Quelltext einer Programmdatei, wobei alle Präprozessor-Kommandos (Präprozessor-Direktiven) mit dem Zeichen # beginnen. Zwischenraum-Zeichen (engl. *whitespace: Leerzeichen,* `\f, \n, \r, \t` oder $\backslash v$) und Kommentare sind vor # zugelassen; zwischen # und Anfang der restlichen Präprozessor-Direktive sind nur *Leerzeichen,* $\backslash t$ und Kommentare zugelassen.

Beispiel:

So sind z. B. die folgenden drei Angaben erlaubte Präprozessor-Direktiven in C89 und C99:

```
/* Kommentar */  #if  TEST
  # /* Kommentar */  if TEST
  # if /* Kommen-
               tar */  TEST
```

Üblicherweise ruft der Compiler automatisch den Präprozessor auf, bevor er mit der Übersetzung beginnt.

C89 schreibt vor, dass der Präprozessor wie ein eigener Schritt vor dem eigentlichen Compilerlauf zu verstehen ist[1].

Der Präprozessor bietet nun die folgenden Dienstleistungen an:

- ❏ Bedingte Kompilierung (`#ifdef, ...`)
- ❏ Einkopieren von anderen Dateien (`#include`)
- ❏ Makro-Ersetzung (`#define`)
- ❏ Makrodefinition aufheben (`#undef`)
- ❏ Vordefinierte Makronamen (`__LINE__, __FILE__, ...`)
- ❏ Zeilenumerierung (`#line`)
- ❏ Fehlermeldungen (`#error`)
- ❏ Pragmas (`#pragma`)
- ❏ Null-Direktive (#)

[1] Das heißt nicht, dass der Präprozessor-Lauf als eigener Durchgang (wie es in heutigen Compilern oft der Fall ist) realisiert sein muss, sondern sich nur so verhalten muss

Beispiele:

```
#if bed1
     progteil1
#elif bed2
     progteil2
#elif bed3
     progteil3
#else
     progteil4
#endif
```

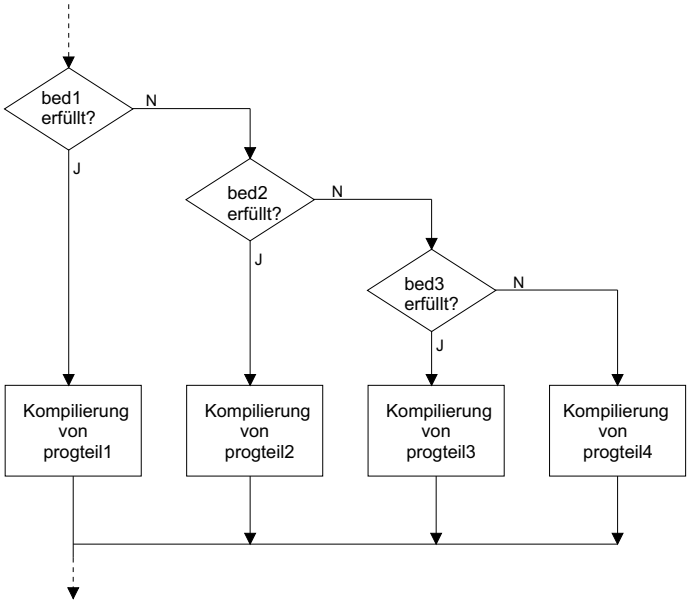


Abbildung 24.1: Programmablaufplan zu einer `#if`-Kaskade

Schachteln von `#if`-Konstruktionen ist erlaubt, wie z. B.:

```
#if beda
#   if beda1
       progteila1
#   else
       progteila2
#   endif
#else
#   if bedb1
       progteilb1
#   else
       progteilb2
#   endif
#endif
```

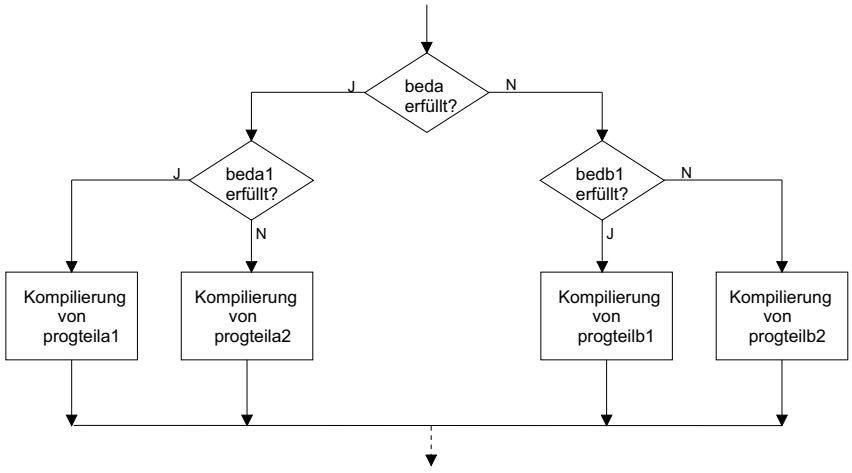


Abbildung 24.2: Programmablaufplan zu einer geschachtelten `#if`-Konstruktion

Kapitel 25

Zeiger und Arrays

In der Programmierpraxis sind häufig Probleme anzutreffen, bei denen man nicht mit einfachen Variablen auskommt, sondern einen ganzen Block von Variablen des gleichen Typs benötigt. Ein Beispiel dafür wären die Monatumsätze eines Unternehmens für ein Jahr. In diesem Fall ist es sehr nützlich, 12 `double`-Variablen in einem Block zu definieren, an dem nur ein Name vergeben wird. Zur Unterscheidung der Monate wird dann z. B. ein Index (ganze Zahl) verwendet.

25.1 Eindimensionale Arrays

Ein *Array* ist die namentliche Zusammenfassung einer Anzahl von gleichartigen Objekten eines Datentyps, wie z. B. `int`- oder `char`-Variablen. Anstelle des Begriffes Array wird manchmal auch der Begriff *Vektor* verwendet. Hier werden wir immer den Begriff Array benutzen.

25.1.1 Eindimensionale Arrays

Ein eindimensionales Array mit 26 Elementen vom Typ `char` könnte wie folgt vereinbart werden:

```
char buchst[26];
```

Mit dieser Angabe wird ein Array `buchst` mit 26 Elementen vom Typ `char` definiert. Die Anzahl der Elemente wird bei der Definition in eckigen Klammern angegeben. Unter

```
char buchst[26];
```

können wir uns die Reservierung eines Speicherblocks mit Namen `buchst` und mit 26 aufeinanderfolgenden Elementen (hier also 26 `char`-Variablen) vorstellen, wie dies auch links in Abbildung 25.1 gezeigt ist. Mit der Definition

```
char buchst[26];
```

werden also, wenn man so will, auf einmal 26 `char`-Variablen (`buchst[0], buchst[1], buchst[2], ..., buchst[25]`) festgelegt.

Werden n Elemente (hier: $n = 26$) für ein Array definiert, so erfolgt die Adressierung über so genannte Indizes von Element 0 bis Element $n - 1$.

Beachten Sie, dass jedes Array mit der Elementnummer 0 und nicht mit 1 beginnt!

Mit den Anweisungen

```
buchst[3] ='a';
buchst[0] ='!';
buchst[24] ='H';
```

ergäbe sich dann ein Speicherbild, wie es rechts in Abbildung 25.1 gezeigt ist.

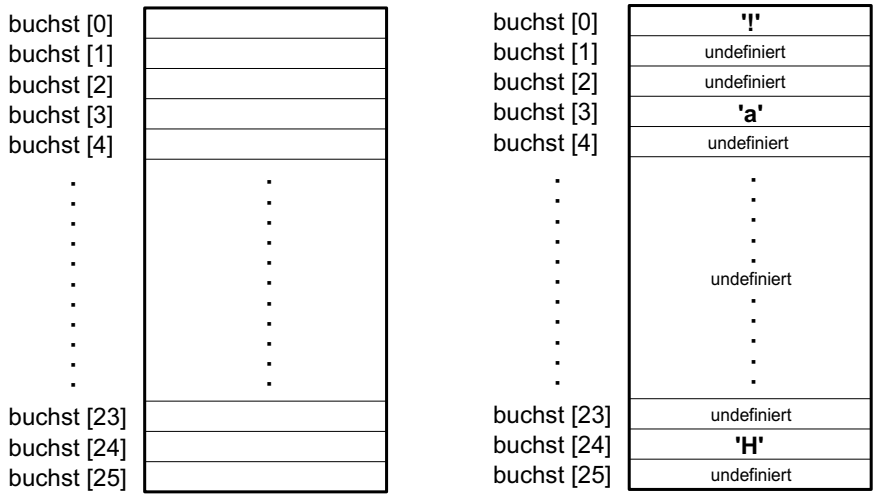


Abbildung 25.1: `char`-Array `buchst` mit 26 Speicherplätzen

Beispiele:

`int tage[32];`
definiert ein Array `tage` mit 32 `int`-Variablen `tage[0], ..., tage[31]`.

`int *zeig[20];`
definiert ein Array `zeig` mit 20 Zeigern `zeig[0],...,zeig[19]` auf `int`-Variablen.

Beispiel:

Wir wollen nun ein C-Programm `buchstnr.c` erstellen, das zunächst in einem Array `buchst` alle Kleinbuchstaben speichert. Danach kann der Benutzer eine Nummer eingeben, zu der ihm der entsprechende Kleinbuchstabe am Bildschirm ausgegeben wird. Unser Programm soll zur Ermittlung des Buchstabens auf das Array `buchst` zugreifen.

```
#include  <stdio.h>

int  main(void) {
   char   buchst[26];
   int    i,  zahl;
    for (i = 0; i < 26; i++)  /* Array mit Kleinbuchstaben belegen */
        buchst[i] = 'a' + i;
```

Kapitel 26

Argumente auf der Kommandozeile

26.1 Die Parameter argc und argv der Funktion main()

Jedes C-Programm muss – wie wir wissen – einen Hauptprogrammteil `main()` besitzen. `main()` ist aber nichts anderes als eine Funktion, die beim Start des C-Programms vom Betriebssystem oder dem C-Laufzeitsystem aufgerufen wird. Wir gingen bisher davon aus, dass `main()` ohne Parameter arbeitet:

```
int main(void)
```

In Wirklichkeit werden aber bei jedem Aufruf der Funktion `main()` zwei Argumente mitgegeben, die wie folgt definiert sind:

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

Die beiden Parameter `argc` und `argv` ermöglichen die Übergabe von Strings an das aufgerufene Programm, wobei gilt:

Erster Parameter **`argc`**
: ist die Anzahl der übergebenen Strings (Worte).

Zweiter Parameter **`argv`**
: ist ein Array von Zeigern, wobei jeder dieser Zeiger auf den Anfang eines Strings (Worts) aus der Kommandozeile zeigt.

Das folgende Programm `argtest.c` demonstriert die Verwendung von `argc` und `argv`:

```
#include  <stdio.h>
int  main(int argc, char *argv[]) {
  int  i;
   for (i=0; i<argc; i++)
      printf("Wort %d: %s\n", i, argv[i]);
   return 0;
}
```

Üblicherweise wird ein solches Programm, nachdem es kompiliert und gelinkt wurde, wie z. B.:

```
user@linux:~ >  cc -o argtest argtest.c ⏎
```

Kapitel 27

Dynamische Speicher-Reservierung und -Freigabe

27.1 Nachteile von statischen Arrays

Das folgende Programm `namsort1.c`, das bis zu 100 Namen einliest und diese dann sortiert ausgibt, zeigt die Nachteile von statischen Arrays in gewissen Anwendungen.

```
#include  <stdio.h>
#include  <string.h>
#define  MAX_NAMEN   100
#define  MAX_LAENGE   30
int  main(void) {
    int   i, j, n;
    char  nam[MAX_NAMEN][MAX_LAENGE], hilf[MAX_LAENGE];
    printf("Gib Deine Namen ein (Abschluss mit Leerzeile)\n"); /* Einlesen der Namen */
    for (n = 0; n < MAX_NAMEN; n++) {
       gets(nam[n]);
       if (strlen(nam[n]) == 0)
          break;
    }
    for (i = 0; i < n-1; i++) /* Sortieren der Namen */
       for (j = i+1; j < n; j++)
          if (strcmp(nam[i], nam[j]) > 0) {
             strcpy(hilf, nam[i]);
             strcpy(nam[i], nam[j]);
             strcpy(nam[j], hilf);
          }
    printf("---------------------- Sortierte Namensliste\n");
    for (i = 0; i < n; i++)
       printf("%s\n", nam[i]);
    return 0;
}
```

Kapitel 28

Strukturen

Unter einer Struktur versteht man die Zusammenfassung von einer oder mehreren Variablen, die – anders als bei Arrays – untereinander auch unterschiedliche Datentypen besitzen dürfen, zu einer Einheit.

28.1 Deklaration und Definition von Strukturen

28.1.1 Deklaration von Strukturen

Ein typisches Beispiel für eine einfache Struktur wäre die Zusammenfassung der Daten eines Studenten aus einer Studentenkartei zu einer einzigen Einheit, die wir `stu_daten` nennen. Eine solche Zusammenfassung wird in C z. B. folgendermaßen ausgedrückt:

```
struct stu_daten {
    char  name[20];       /* Nachname                  */
    char  vorname[20];    /* Vorname                   */
    long  postleit_zahl;  /* Postleitzahl              */
    char  wohnort[20];    /* Wohnort                   */
    char  strass_nr[20];  /* Strasse, Hausnummer       */
    char  geburtdat[11];  /* Geburtsdatum: tt.mm.jjjj  */
    long  matrikelnr;     /* Matrikelnummer            */
    short noten[10];      /* Prüfungsnoten             */
};
```

Bei der obigen Angabe handelt es sich um die *Deklaration einer Struktur*. Aus diesem Beispiel ist der allgemeine Aufbau einer Strukturdeklaration erkennbar:

- Eine Strukturdeklaration beginnt immer mit dem Schlüsselwort `struct`. Nach Schlüsselwort `struct` kann der Name der Struktur (oben: `stu_daten`) angegeben werden.
- Eine Strukturdeklaration besteht aus einer Liste von Komponenten (Einzelvariablen, wie z. B. `name, vorname, postleit_zahl` usw.).
- Jede Komponente wird mit ihrem Datentyp und ihrem Namen deklariert.
- Die ganze Komponentenliste ist mit `{...}` zu klammern.
- Jede Strukturdeklaration ist mit Semikolon abzuschließen.

Für unseren Fall ist die 1. Bedingung und somit auch die Gesamtbedingung erfüllt, so dass folgende Anweisung ausgeführt wird:

```
knot_zeig->links = einordnen(knot_zeig->links);
```

Die Funktion `einordnen()` ruft sich also selbst wieder auf, so dass sich das in Abbildung 28.23 gezeigte Bild ergibt.

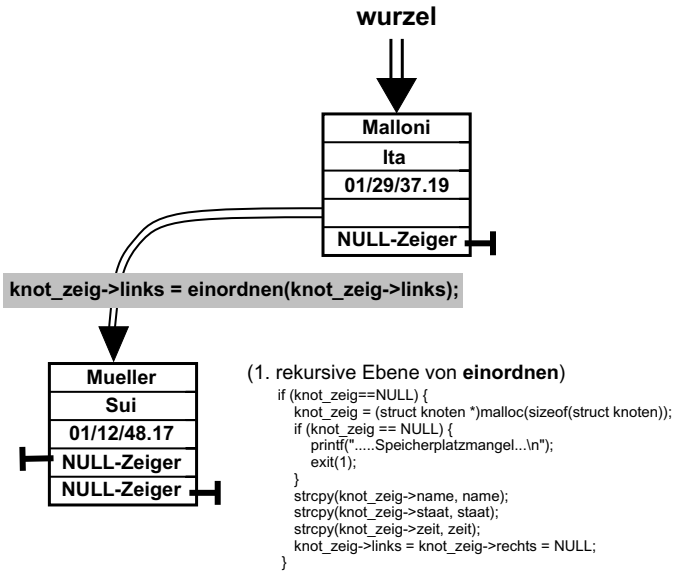


Abbildung 28.23: Einordnen von `Mueller` als linker Unterknoten von `Malloni`

Mit

```
return knot_zeig;
```

wird immer die Adresse des gerade behandelten Knotens an die übergeordnete Ebene zurückgegeben. Von der ersten rekursiven Ebene wird hier z. B. die Adresse des `Mueller`-Knotens zurückgegeben. Diese Adresse wird wegen

```
knot_zeig->links = einordnen(knot_zeig->links);
```

im `Malloni`-Knoten unter der Komponente `knot_zeig->links` (hier `wurzel->links`) gespeichert. Nach der Rückkehr nach `main()` wird dort unter Zuhilfenahme der Funktion `drucke_baum()` folgender Zwischenstand ausgegeben:

```
Zwischenstand nach 2 Laeufer
============================
       Platz Name            Land  Zeit
         1   Mueller          Sui  01/12/48.17
         2   Malloni          Ita  01/29/37.19
```

3. eintreffender Läufer In der `while`-Schleife von `main()` werden nun die Daten des nächsten eintreffenden Läufers eingelesen:

name: Hanson
staat: Swe
zeit: 01/20/10.33

Danach wird mit **`wurzel = einordnen(wurzel);`** wieder die Funktion `einordnen()` aufgerufen. In dieser Funktion wird nach Überprüfung der entsprechenden Bedingungen die Anweisung **`knot_zeig->links = einordnen(knot_zeig->links);`** ausgeführt. Bei diesem rekursiven Aufruf von `einordnen()` wird nach Überprüfung der vorgegebenen Bedingungen die Anweisung **`knot_zeig->rechts = einordnen(knot_zeig->rechts);`** ausgeführt, so dass sich das in Abbildung 28.24 gezeigte Bild ergibt.

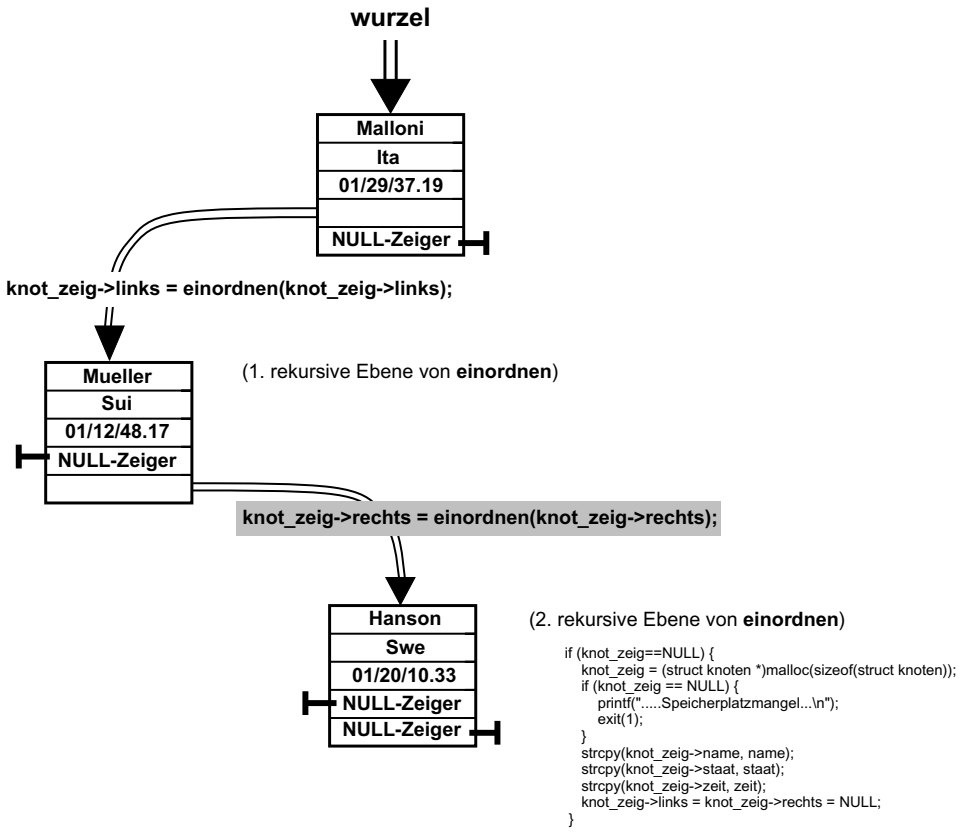


Abbildung 28.24: Einordnen von `Hanson` als rechter Unterknoten von `Mueller`

Nach der Rückkehr nach `main()` wird dort unter Zuhilfenahme der Funktion `drucke_baum()` folgender Zwischenstand ausgegeben:

```
Zwischenstand nach 3 Laeufer
============================
       Platz Name            Land  Zeit
         1   Mueller          Sui  01/12/48.17
         2   Hanson           Swe  01/20/10.33
         3   Malloni          Ita  01/29/37.19
```

4. eintreffender Läufer In der `while`-Schleife von `main()` werden nun die Daten des nächsten eintreffenden Läufers eingelesen:

```
name:   Bichler
staat:  Aut
zeit:   01/11/59.12
```

Danach wird mit **`wurzel = einordnen(wurzel);`** wieder die Funktion `einordnen()` aufgerufen. Der rekursive Ablauf dieser Funktion soll durch das in Abbildung 28.25 gezeigte Bild verdeutlicht werden.

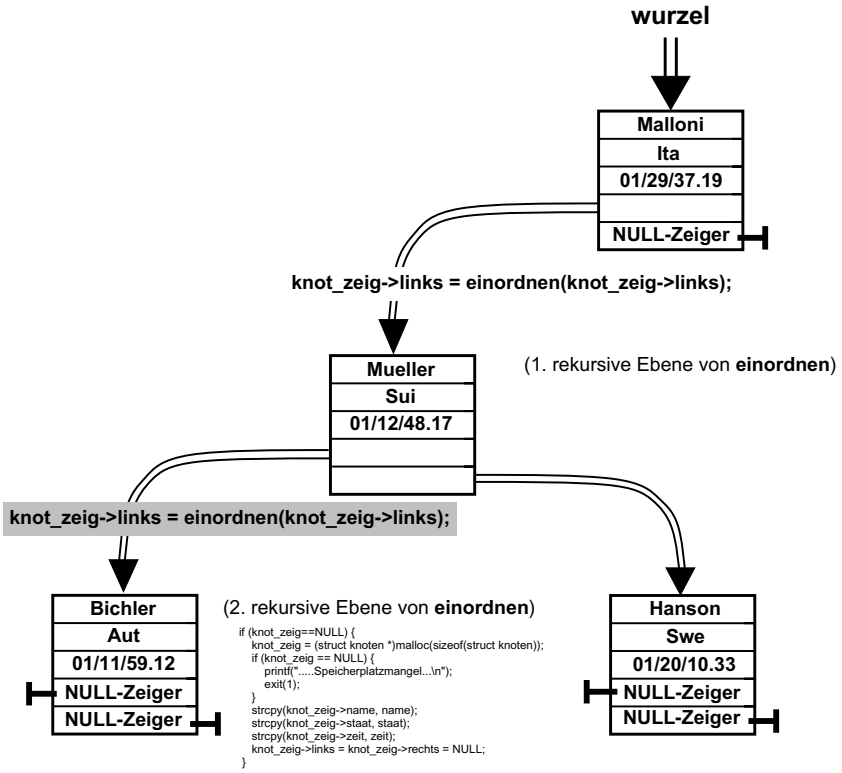


Abbildung 28.25: Einordnen von `Bichler` als linker Unterknoten von `Mueller`

Nach der Rückkehr nach `main()` wird dort unter Zuhilfenahme der Funktion `drucke_baum()` wieder der aktuelle Zwischenstand ausgegeben:

```
Zwischenstand nach 4 Laeufer
============================
       Platz Name            Land  Zeit
         1   Bichler          Aut  01/11/59.12
         2   Mueller          Sui  01/12/48.17
         3   Hanson           Swe  01/20/10.33
         4   Malloni          Ita  01/29/37.19
```

Hier können wir die Funktionsweise von `drucke_baum()` sehr schön erkennen: Es wird immer zuerst der vollständige linke Unterbaum ausgegeben, bevor ein Knoten selbst ausgegeben wird.

Kapitel 29

Eigene Datentypen

29.1 Definition eigener Datentypnamen mit typedef

29.1.1 Vergabe neuer Namen an existierende Datentypen mit typedef

Mit dem Schlüsselwort `typedef` können an Datentypen neue, vom Benutzer wählbare Namen vergeben werden, wie z. B.:

```
typedef  unsigned int  UINT;
typedef  unsigned long ULONG;
```

Mit diesen Angaben wird festgelegt, dass `UINT` ein Synonym für `unsigned int` und `ULONG` eines für `unsigned long` ist. Nach der obigen `typedef`-Angabe könnte man z. B. folgende Definitionen angeben:

```
UINT   zaehler, i;      /* entspricht:  unsigned int   zaehler, i;     */
UINT   *zeigarray[10];  /* entspricht:  unsigned int   *zeigarray[10]; */
ULONG  a, *b;           /* entspricht:  unsigned long  a, *b;          */
```

Man könnte nun glauben, dass das Schlüsselwort `typedef` völlig überflüssig ist, da man den gleichen Effekt mit `#define` hätte erreichen können, wie z. B.:

```
#define  UINT   unsigned int
#define  ULONG  unsigned long
```

Für diese einfachen Datentypdefinitionen ist dies zwar möglich, doch reicht bei komplexeren Datentypen der einfache Textersatz von `#define` nicht aus, wie z. B. bei:

```
#define STRING  char *
```

In diesem Fall würde die folgende Definition:

```
STRING  zgr1, zgr2;
```

nicht der erwarteten Definition

```
char  *zgr1, *zgr2;
```

entsprechen, sondern der folgenden Definition:

```
char  *zgr1, zgr2;  /* zgr2 ist nicht - wie erwartet - ein Zeiger */
```

Kapitel 30

Dateien

Eine *Datei* (engl. *file*) ist letztendlich nichts anderes als eine Ansammlung von Daten, die auf einem externen Speicher dauerhaft aufbewahrt werden. Aus Sicht von C kann man sich eine Datei als ein permanentes, großes `char`-Array vorstellen, in das ein Programm Daten schreiben und aus dem ein Programm Daten lesen kann.

Anders als in anderen Programmiersprachen, ist eine C-Datei nicht irgendwie strukturiert, sondern nur eine Folge von Bytes, weswegen eine Datei in C oft auch als Datenstrom (*byte stream*) bezeichnet wird. Jedes einzelne dieser Zeichen (Bytes) kann über eine Positionsnummer in der Datei lokalisiert werden. So besitzt das erste Zeichen einer Datei die Positionsnummer 0, das zweite die Positionsnummer 1 usw.

Man kann in C grundsätzlich auf zwei Ebenen mit Dateien operieren, auf einer „unteren“ (*low level*) oder einer „oberen“ (*high level*).

Die untere Ebene ist die der so genannten *elementaren Dateizugriffe* (*Elementare Ein-/Ausgabe*), bei denen Funktionen verwendet werden, die direkt auf die entsprechenden Routinen des Betriebssystems (*system calls*) für Dateioperationen zurückgreifen. Diese elementaren Funktionen sind natürlich vom Betriebssystem abhängig und gehören nicht zum C-Standard, was bei der Portierung von Programmen von großer Bedeutung ist.

Die obere Ebene ist die der so genannten *höheren Dateizugriffe* (*Höhere Ein-/Ausgabe*), bei denen Funktionen verwendet werden, die zum einen wesentlich komfortabler als die elementaren sind und zum anderen auch dem C-Standard entsprechen und somit vom jeweiligen Betriebssystem unabhängig sind.

Nachfolgend werden nur die *Höheren E/A-Funktionen* vorgestellt.

30.1 Höhere E/A-Funktionen

In diesem Kapitel werden E/A-Funktionen beschrieben, die sich in der Standard-Bibliothek befinden und in der Headerdatei `<stdio.h>` definiert sind. Die höheren E/A-Funktionen arbeiten im Gegensatz zu den elementaren E/A-Funktionen mit eigenen optimal eingestellten Puffern, so dass sich der Aufrufer darum nicht selbst kümmern muss. Ein solcher Dateipuffer ist ein Bereich im Arbeitsspeicher, in dem eine bestimmte Menge von Daten, die aus einer Datei gelesen oder in eine Datei geschrieben werden sollen, vor ihrer Übertragung zum Zielort zwischengespeichert werden kann. Dies ist ein Vorteil, denn auf diese Weise muss nicht für jedes

Kapitel 31

Anhang

31.1 Prioritätstabelle für die Operatoren

Operator (höchste Priorität oben; niedrigste unten)	**Assoziativität**
( ) [ ] -> . (Punktoperator)	von links her
! ~ (*Tilde*) ++ -- + (*Vorz.*) - (*Vorz.*) sizeof & (Adreßop.) * (Zeigerzugriff) (*casting*)	von rechts her
* / %	von links her
+ -	von links her
« »	von links her
< <= > >=	von links her
== !=	von links her
& (bitweise AND)	von links her
^	von links her
|	von links her
&&	von links her
||	von links her
?: (Bedingter Operator)	von rechts her
= += -= *= /= %= »= «= &= |= ^=	von rechts her
, (Kommaoperator)	von links her

31.2 C-Schlüsselwörter

In folgender Tabelle sind alle C-Schlüsselwörter angegeben, wobei die in C99 neu hinzugekommenen Schlüsselwörter fett gedruckt sind.

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

Index