Handbuch zur Schulung »Einführung in C«

Handbuch zur Schulung »Einführung in C«

Sascha Kliche

IBM Deutschland GmbH

HR University Education IBM Germany
Kst. 1710
Geb. 10178−80

16 Dec 2004

Version 8.2.0

Vorwort

Dieses Handbuch wurde als Begleitmaterial zu einer Einführungsschulung in die Programmiersprache C entwickelt. Die erste Fassung entstand im September 1994 in Zusammenarbeit mit Daniel Matthies im Rahmen einer Praxisphase in der BAB Hannover. Eine stark überarbeitete Fassung entstand im Juni 1997 und enthielt neben Fehlerbereinigungen einige neue Kapitel und Abschnitte. Die vorliegende Fassung wurde seitdem ebenfalls um einige Abschnitte erweitert und korrigiert. Als letztes kamen die Kapitel zur Socket Programmierung, Makefiles und dynamischen Bibliotheken (DLLs) hinzu sowie Ergänzungen im Bereich Felder.
Letzte Ergänzung: Non−Blocking Server

Berlin, Februar 2004

Sascha Kliche

Inhaltsverzeichnis

Fortgeschrittene Themen in C

Tabellen

Abbildungen

Einführung

In diesem Teil des Handbuches werden Hinweise zum Aufbau und Umgang mit diesem Handbuch sowie einleitende Informationen zur Sprache C gegeben.

1.0 Informationen zum Handbuch

Dieses Handbuch bietet eine Einführung in die Programmiersprache C sowie einen umfassenden Überblick über viele Standardfunktionen, die in fast jeder Implementation zu finden sind. Es wird weder ein Anspruch auf Vollständigkeit erhoben, noch werden sämtliche Möglichkeiten der Programmiersprache C behandelt, da dies den Rahmen dieses Dokumentes sprengen würde. Dieses Handbuch ist vielmehr als Ergänzung zur Einführungsschulung gedacht. Funktionen, Elemente, etc., die in den Übungsbeispielen vorkommen, werden hier vollständig behandelt. Zusätzlich sind Informationen über wichtige Funktionen und Elemente enthalten, die im Rahmen der Schulung nicht erwähnt werden, in der Praxis jedoch häufig Verwendung finden. Dieses Handbuch orientiert sich am ANSI−C Standard und dem IBM C/C++ Compiler Version 3.6 für OS/2, AIX und Windows NT(©).

Die folgende Aufstellung beinhaltet die Themen, die in diesem Handbuch keine Erwä;hnung finden bzw. nicht behandelt werden:

- Schlüsselwort `goto`,
- Schlüsselwort `volatile` und
- Bit−Operationen

1.1 Konventionen

In diesem Handbuch werden bestimmte Hervorhebungen zur Identifikation von Informationen benutzt, die im folgenden erläutert werden:

Schriftart	Verwendung
`Monospaced`	Befehle oder Text, der exakt so eingegeben werden muß, wie er abgebildet ist.
kursiv	Neue Begriffe sind kursiv gedruckt. Diese werden bei ihrem ersten Vorkommen im Text erläutert. Ebenfalls kursiv sind Namen und besondere Begriffe.
fett	Tasten oder Tastenkombinationen werden fett gedruckt.

Programmcode im Fließtext wird folgendermaßen dargestellt: Dieser Text enthält `markierten Programmcode`, der sich vom Rest abhebt. Beispiele werden vom Fließtext abgetrennt dargestellt. Beispiel:

```
Dies ist die Darstellung für Beispiele.
Beispiele umfassen in der Regel mehrere Zeilen und sind
deutlich vom restlichen Text abgetrennt.
```

In den Programmbeispielen werden C++ Kommentare (//) genutzt. Diese werden von allen modernen Compilern auch in C Programmen akzeptiert.

1.2 Aufbau des Handbuches

Kapitel 1.0, "Informationen zum Handbuch" enthält allgemeine Hinweise zu diesem Handbuch, wie z.B. Aufbau des Handbuches und Konventionen.

Kapitel 2.0, "Einleitung in C" enthält allgemeine Hinweise zur Sprache C wie z.B. die Historie.

In Kapitel 3.0, "Grundlagen der C Programmierung" werden grundsätzliche Aspekte der Programmierung in C, wie z.B. der verfügbare Zeichenvorrat, Operatoren, etc., erläutert.

In Kapitel 4.0, "Datentypen" werden die in C verfügbaren Datentypen aufgeführt und ihre Eigenschaften erläutert.

In Kapitel 5.0, "Auswahl" befinden sich Erläuterungen zur Programmierung von Auswahlentscheidungen.

Kapitel 6.0, "Wiederholungen − Schleifen" beschreibt die Möglichkeiten, die C zur Wiederholung von Programmteilen anbietet.

In Kapitel 7.0, "Eigene Funktionen" wird aufgezeigt, wie eigene Funktionen in C erstellt und angewendet werden können.

In Kapitel 8.0, "Formatierte Ein− und Ausgabe in C" werden die Standardmechanismen zur formatierten Ein− und Ausgabe in C++ vorgestellt.

In Kapitel 9.0, "Übersicht über wichtige Funktionen" werden wichtige Standardfunktionen von C besprochen.

Kapitel 10.0, "Dateibehandlung" beschreibt den Umgang mit Dateien. Dies umfaßt z.B. das Öffnen, Lesen und Schreiben von Dateien.

Kapitel 11.0, "Preprocessor−Anweisungen (Direktiven)" führt die in C verfügbaren Preprocessor−Anweisungen auf.

Kapitel 12.0, "Die häufigsten Programmierfehler in C" bietet eine Übersicht über die häufigsten Programmierfehler in C.

Kapitel 13.0, "Neuerung des ISO/IEC 9899:1999 Standards" stellt einige Neuerungen des ISO/IEC 9899:1999 Standards vor.

Kapitel 14.0, "Unterschiede zwischen C und C++" stellt eine Abgrenzung von C zu C++ dar. Es wird auf die grundsätzlichen Unterschiede und Gemeinsamkeiten von C und C++ eingegangen.

Kapitel 15.0, "Speicherverwaltung" stellt die Methoden zur Speicherverwaltung in C vor.

Kapitel 16.0, "Verbindungen zum Betriebssystem" geht auf die Möglichkeiten ein, Verbindungen zum Betriebssystem aufzubauen, z.B. um Betriebssystembefehle ausführen zu können.

Kapitel 17.0, "Dynamische Bibliotheken − DLL" ist eine Einleitung in die Erstellung dynamischer Bibliotheken (DLLs).

Kapitel 18.0, "Reguläre Ausdrücke" enthält eine Einführung in die Nutzung sog. regulärer Ausdrücke.

Kapitel 19.0, "Makefiles" stellt einfache Makefiles vor.

Kapitel 20.0, "Multithreading" beschäftigt sich mit Multithreading.

Kapitel 21.0, "Socketprogrammierung" zeigt anhand zweier Beispielanwendungen, wie mittels Socketprogrammierung Netzwerkverbindungen aufgebaut werden. Das erste Beispiel schickt HTTP Anfragen an einen Server und das zweite Beispiel ist ein dummer Server, der auf Anfrage einen Standardsatz verschickt.

Kapitel 22.0, "Graphische Anwendungen in C" demonstriert ein Beispiel der GUI−Programmierung unter C.

Kapitel 23.0, "Interaktion mit Java − Java Native Interface (JNI)" zeigt die Möglichkeiten zur Verbindung von C und Java auf.

Im Anhang befinden sich hilfreiche Aufstellungen, Übersichten und das Literaturverzeichnis.

2.0 Einleitung in C

2.1 Historie von C

Die Programmiersprache C wurde 1972 von Dennis Ritchie in den Bell Laboratories in New Jersey (USA) entworfen. Sie basiert auf der Programmiersprache B, welche von Ken Thompson aus der Sprache BCPL (**b**asic **c**ombined **p**rogramming **l**anguage) abgeleitet wurde. Beide Programmierer arbeiteten an der Entwicklung des Betriebssystems *Unix* und waren mit der verwendeten Sprache *Assembler* unzufrieden. Assembler ist sehr schnell, jedoch umständlich, schwierig und nicht portabel [1] Aus diesen Gründen entwickelten sie eine schnelle *Compilersprache*, die auf vielen verschiedenen Computersystemen einsetzbar ist.

Im Kapitel 13.0, "Neuerung des ISO/IEC 9899:1999 Standards" werden einige Neuerungen des Standards ISO/IEC 9899:1999, der im Dezember 1999 in Kraft trat, vorgestellt.

Dennis Ritchie fasst unter http://cm.bell−labs.com/cm/cs/who/dmr/chist.html die Geschichte von C zusammen.

2.2 Vom Quellcode zum lauffähigen Programm

Als Quellcode bezeichnet man eine endlich lang Folge von Befehlen zur Umsetzung von Algorithmen. Ein Algorithmus ist eine endlich lange Vorschrift, bestehend aus Einzelanweisungen. Eine endlich lange Vorschrift hat nicht zwangsweise eine endlich lange Laufzeit zur Folge, aber die Beschreibung der Vorschrift ist endlich. Ein Beispiel für einen Algorithmus ist ein Kochrezept ("Man nehme..."). Der Durchführende kennt die Bedeutung der Einzelanweisungen, welche deterministisch, also nicht zufällig, abgearbeitet werden.

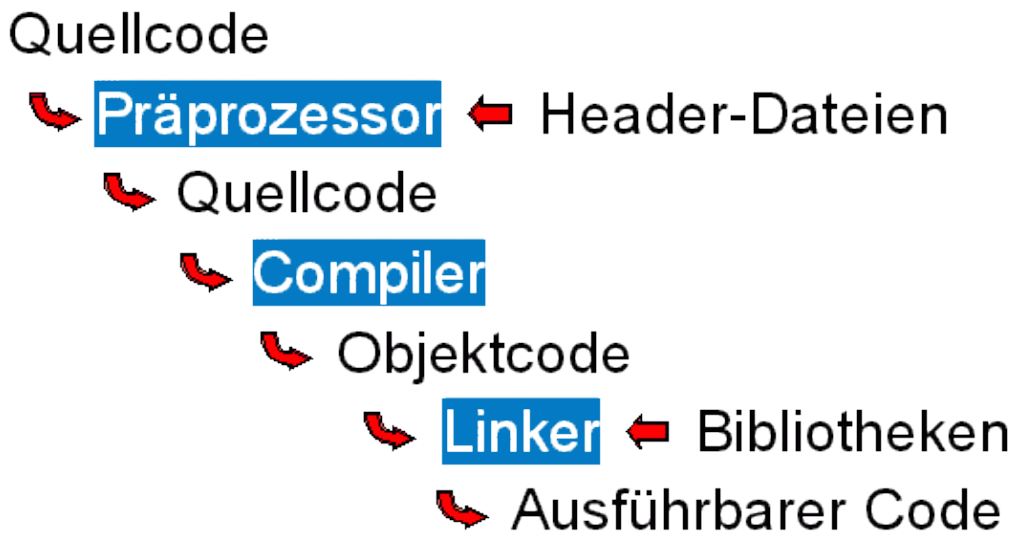
Programmiersprachen werden einerseits nach dem Zeitpunkt ihrer Übersetzung in Maschinensprache (Compiler− oder *Interpretersprache*), und andererseits nach ihrer Orientierung (prozedurale oder objektorientierte Sprache) unterschieden.

Ein, mit einer Compilersprache (z.B. C) erstelltes Programm ist nach dem sog. Link Vorgang, bei dem die verschiedenen Maschinensprachebestandteile eines Programms zusammengefügt werden, ohne weitere Zusätze eigenständig lauffähig. Im Gegensatz dazu ist ein, mit einer Interpretersprache (z.B. REXX) erstelltes Programm nicht ohne den Interpreter selbst lauffähig. Dies liegt daran, daß der Code vom Interpreter erst während der Laufzeit des Programms übersetzt wird, also immer als Quellcode vorliegt.
Eine Mischform stellen Programme dar, die zwar mit einer Compilersprache erzeugt werden, aber sogenannte *Laufzeitbibliotheken* (*Runtime−Libraries*) zur eigenständigen Verwendbarkeit benötigen, die als zusätzliche Dateien vorliegen müssen.

Um aus einem C Quellcode ein lauffähiges Programm zu erzeugen, muß dieser zunächst übersetzt werden, diese Aufgabe übernimmt der Compiler. Der Compiler durchsucht den Quelltext zunächst nach den sog. Preprozessoranweisungen und führt diese aus. Dies hat z.B. zur Folge, daß weitere Dateien in die Übersetzung mit einbezogen werden. Im nächsten Schritt werden der Quellcode und alle hinzugezogenen Dateien auf syntaktische Fehler untersucht und − wenn möglich − fast vollständig in Maschinensprache übersetzt. Das Ergebnis dieses Vorgangs ist der sog. Objektcode. Nach dem Compilieren muß der Linker den erzeugten Objektcode mit bereits existierendem Objektcode verbinden und erzeugt letztendlich das lauffähige Programm. Dieser komplette Ablauf kann bei komplexen Programmen sehr viel Zeit in Anspruch nehmen, so daß häufig nur einzelne Programmteile übersetzt und getestet werden. Eine weitere Möglichkeit bietet das sog. Prototyping per Interpretersprache. Hierbei werden Algorithmen zunächst in einer Interpretersprache (z.B. REXX) implementiert. Dies hat den Vorteil, daß das Programm, so unvollständig es auch ist, aufgerufen und getestet werden kann. Sind die Algorithmen (logisch) korrekt, werden sie in die Compilersprache

umgesetzt.

Abbildung 1. Ablauf Compile−Link−Go



2.3 Argumente für den Einsatz der Sprache C

Die Programmiersprache C bietet dem Programmierer ein Fülle von Vorteilen gegenüber anderen Programmiersprachen:

- C ist für fast alle Computersysteme erhältlich (PC/PS, Home−Computer, Mainframe, etc.).
- C ist leicht übertragbar auf andere Systeme (gute Portabilität).
- Eine Strukturierung der Programme ist in C möglich.
    - ♦ Problemstellungen können in C durch Untergliederung in Teilaufgaben bearbeitet werden.
    - ♦ Teilprogramme können wieder−/weiterverwendet werden.
    - ♦ Die Lesbarkeit des Quellcodes wird verbessert.
- Der erzeugte Code ist schnell, aber dennoch kompakt.
- C ist für viele Problembereiche einsetzbar.
- C ist leicht erlernbar.

Der entstehende Eindruck einer "eierlegenden Wollmilchsau" wird für diejenigen, die die Sprache erlernen wollen, jedoch durch deren Kryptik und Inkonsistenz getrübt. C zu erlernen ist grundsätzlich nicht sehr schwer. Den Code anderer Programmierer zu verstehen kann jedoch zur Nervenprobe werden: C−Programmierer sind vermutlich schreibfaul und C bietet viele Möglichkeiten, Befehle und Funktionen zu kombinieren und abzukürzen.

Die beiden folgenden Anweisungen leisten das gleiche:

```
(1)   return (a > b) ? a : b;

(2)   if (a > b) return a;
         else    return b;
```

Ähnlich ungewohnte Konstruktionen wie (1) erschweren das Verständnis, sind aber nach einiger Zeit beinahe selbstverständlich.

Grundlegende Sprachelemente

In diesem Teil des Handbuches werden die sprachlichen Grundlagen der Sprache C aufgezeigt.

3.0 Grundlagen der C Programmierung

In diesem Kapitel werden grundlegende Aspekte der Programmiersprache C wie z.B. der zur Verfügung stehende Zeichensatz und der strukturelle Aufbau eines C Quellcodes vorgestellt. Des weiteren wird auf die Sensibilität bezüglich Groß− und Kleinschreibung sowie auf Kommentare, Schlüsselwörter, Operatoren und Bezeichner eingegangen.

3.1 Zeichensatz

Folgende Zeichen sind in einem C Quellcode erlaubt:

```
Kleinbuchstaben       : a b C . . . z
Großbuchstaben        : A B C . . . Z
Ziffern               : 0 1 2 3 4 5 6 7 8 9
Trennzeichen          : z.B. blank, newline, tab
```

Tabelle 1. Sonderzeichen

Zeichen	englische/amerikanische Bezeichnung
+	plus sign
−	minus sign, hyphen
*	asterisk
/	forward slash
=	equal sign
(	left parenthesis
)	right parenthesis
{	left (curly) brace
}	right (curly) brace
[	left (square) bracket
]	right (square) bracket
<	left angle bracket
>	right angle bracket
'	single quotation
"	double quotation
!	exclamation mark
	ampersand
#	hash mark, number sign
%	percent sign
|	vertical bar
_	underscore
^	caret, and symbol
~	tilde
\	backslash

.	period, dot
,	comma
;	semicolon
:	colon
?	question mark
$	dollar sign
@	at sign

Diese Beschränkung gilt prinzipiell nicht für Text, den ein Programm ausgibt sowie für die Kommentare. Es sollte jedoch aus Gründen der Portabilität auf Sonderzeichen wie z.B. Umlaute verzichtet werden.

3.2 Groß− und Kleinschreibung

Der Compiler ist sehr empfindlich bezüglich der Groß− und Kleinschreibung, wenn es um Schlüsselworte, Variablen [2] oder Funktionen geht. Die Schreibweise, die in der Variablendefinition angewandt wird, muß das gesamte Programm über beibehalten werden. Dies bedeutet, daß eine Variable `GROSS` nicht erkannt wird, wenn sie in der Deklaration `gROSS` codifiziert wurde. Weiterhin sollte darauf geachtet werden, daß Schlüsselworte und Formatspezifikationen immer in Kleinbuchstaben (Bsp.: `scanf, fakultaet, %d`) geschrieben werden. Dahingegen wird die Übersichtlichkeit und Lesbarkeit verbessert, indem eigene Typen (Abschnitt 4.5, "Eigene Typen − typedef") in Großbuchstaben deklariert werden (Bsp.: `REAL, INTEGER`).

3.3 Schlüsselworte

Schlüsselworte sind reservierte Worte, die nicht neu definiert werden, d.h. als Bezeichner verwendet werden können.

ANSI−C definiert folgende Schlüsselworte:

```
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  wchar_t
while
```

Im folgenden werden die einzelnen Schlüsselworte mit Verweisen auf weitergehende Informationen vorgestellt.

Tabelle 2. Schlüsselworte

auto	Vereinbarung der Speicherklasse `auto` (siehe 4.6, "Speicherklassen")
break	Beendet die aktuelle Schleife und fährt mit der Programmausführung bei der nächsten Anweisung nach der Schleife fort. Beendet auch Anweisungszweige von `switch` Anweisungen. (siehe 6.4, "Die Anweisungen break und continue")
case	Anweisungszweig einer `switch` Anweisung. (siehe 5.2, "switch")
char	Datentyp für Zeichen und Zeichenketten. (siehe 4.1.3, "Textzeichen − char")
const	Vereinbarung einer Konstanten. (siehe 3.7, "Bezeichner")
continue	

	Bricht den aktuellen Schleifendurchlauf ab und springt zum Ende der Schleife. (siehe 6.4, "Die Anweisungen break und continue")
default	Anweisungszweig einer `switch` Anweisung, der ausgeführt wird, wenn kein `case` Zweig zutrifft bzw. vorher kein `break` auftrat. (siehe 5.2, "switch")
do	Beginnt eine `do while` Schleife, die Prüfung erfolgt am Ende eines Schleifendurchlaufs. (siehe 6.0, "Wiederholungen − Schleifen")
double	Vereinbarung einer rationellen Zahl. (siehe 4.1.2, "Rationale Zahlen / Fließkommazahlen")
else	Alternativzweig einer `if` Anweisung. (siehe 5.0, "Auswahl")
enum	Vereinbarung eines Aufzählungstyps. (siehe 4.3.2, "enum")
extern	Vereinbarung externer Bezeichner. (siehe 4.6, "Speicherklassen")
float	Vereinbarung einer rationellen Zahl. (siehe 4.1.2, "Rationale Zahlen / Fließkommazahlen")
for	Schleife mit Prüfung am Anfang eines Schleifendurchlaufs und integriertem Zähler. (siehe 6.0, "Wiederholungen − Schleifen")
goto	Unbedingter Sprung. (wird in diesem Handbuch nicht behandelt)
if	Abfrage einer Bedingung. (siehe 5.0, "Auswahl")
int	Vereinbarung einer ganzen Zahl. (siehe 4.1.1, "Ganze Zahlen − Integer")
long	Vereinbarung einer ganzen Zahl. (siehe 4.1.1, "Ganze Zahlen − Integer")
register	Vereinbarung einer Speicherklasse wie `auto`, jedoch Register statt Speicher, falls möglich. (siehe 4.6, "Speicherklassen")
return	Beendet Funktionen und übergibt ggf. einen Rückgabewert. (siehe 7.0, "Eigene Funktionen")
short	Vereinbarung einer ganzen Zahl. (siehe 4.1.1, "Ganze Zahlen − Integer")
signed	Vereinbarung einer ganzen Zahl oder eines Zeichens mit Vorzeichen. (siehe 4.1.1, "Ganze Zahlen − Integer" bzw. 4.1.3, "Textzeichen − char")
sizeof	Ermittlung der Länge eines Operanden. (siehe 3.6.7, "Hierarchie der Operatoren")
static	Vereinbarung einer Speicherklasse. Der Speicherbereich wird bei Programmbeginn zugeordnet. (siehe 4.6, "Speicherklassen")
struct	Vereinbarung einer Struktur. (siehe 4.2.2, "Strukturen")
switch	Beginn einer Auswahl. (siehe 5.2, "switch")
typedef	Definition eigener Datentypen. (siehe 4.5, "Eigene Typen − typedef" und 4.2.2.7, "Strukturen mit Typendefinitionen − typedef")
union	Vereinbarung verschiedener Datentypen im selben Speicherbereich. (siehe 4.2.3, "Unions")
unsigned	Vereinbarung einer ganzen Zahl oder eines Zeichens ohne Vorzeichen. (siehe 4.1.1, "Ganze Zahlen − Integer" bzw. 4.1.3, "Textzeichen − char")
void	Vereinbarung ohne Typ. (Nichts) (siehe 7.0, "Eigene Funktionen")
volatile	Vereinbarung eines unbeständigen Datentyps. Dieses Schlüsselwort teilt dem Compiler mit, daß die bezeichnete Variable durch Ereignisse außerhalb der Kontrolle des Programms verändert werden kann. Der Wert der Variablen muß deshlab vor jedem Zugriff neu aus dem Hauptspeicher gelesen werden, d.h. er darf nicht in einem Prozessorregister gespeichert werden. (Wird in diesem Handbuch nicht weiter behandelt.)
wchar_t	Datentyp für Zeichen und Zeichenketten in Zeichensätzen, bei denen ein Zeichen mehr als 8 Bit benötigt. (siehe 4.1.3, "Textzeichen − char")
while	Beginn einer `while` oder Ende einer `do while` Schleife. (siehe 5.0, "Auswahl")

Eine Reihe dieser Schlüsselworte wird in diesem Handbuch ausführlich erläutert. Zu beachten ist, daß sämtliche Zuweisungen, Funktionsaufrufe, Variablendeklarationen und eine Reihe der obigen Schlüsselworte mit einem Semikolon abgeschlossen werden müssen.
Beispiel:

```
return(ergebnis);
```

3.4 Formale Grundstruktur

Unter der formalen Grundstruktur versteht man den strukturellen Aufbau, der bei jedem C−Programm, welches nach bestimmten Grundsätzen geschrieben wurde, gleich ist. Da C eine relativ flexible Gestaltung des Quellcodes zuläßt, kann dies zu stark unterschiedlich aufgebauten Quellcodes führen. Die Einhaltung gewisser Standards erhöht jedoch die Lesbarkeit des Codes ungemein und sollte aus diesem Grund berücksichtigt werden.

Die übliche Grundstruktur stellt sich folgendermaßen dar:

```
/* Name, Beschreibung */                                (1)

#include <stdio.h>                                      (2)
...


Vereinbarungsteil globaler Variablen                    (3)
...


[typ] funktionsname ([typ]);                            (4a)
...


[typ] funktionsname ([typ])                             (4b)
{
   Vereinbarungsteil lokaler Variablen
   ...
   Anweisungsteil
   ...
}  /* Ende von xy */


[typ] main([typ])                                       (5)

{                                                       (6)
   Vereinbarungsteil lokaler Variablen
   ...
   Anweisungsteil
   ...
 } /* Ende von main */                                  (6)


[typ] funktionsname ([typ])                             (4c)
{
   Vereinbarungsteil lokaler Variablen
   ...
```

```
        Anweisungsteil
        ...
    }   /* Ende von xy */
```

(1) Alle von `/*` und `*/` eingeschlossenen Textkomponenten stellen Kommentare dar. Zur näheren Erläuterung siehe Abschnitt 3.4.1, "Der Einsatz von Kommentaren".

(2) `#include` ist eine sog. Preprocessoranweisung. Alle Anweisungen, die mit einer Raute (#) beginnen, werden vom sogenannten *Preprocessor* ausgeführt (siehe auch Abschnitt 11.0, "Preprocessor−Anweisungen (Direktiven)"). Die Anweisung `#include <stdio.h>` bewirkt, daß an dieser Stelle im Quelltext vor der Übersetzung in maschinenlesbaren Code die C−Definitionsdatei `stdio.h` eingefügt wird. Die Definitionsdatei `stdio.h` (**st**an**d**ard−**i**nput−**o**utput) befähigt den Compiler, die korrekte Verwendung der Funktionen zur Ein− und Ausgabe zu überprüfen.

(3) Der Vereinbarungsteil für globale Variablen schließt an die `#include`−Anweisungen an und steht noch vor `main()`. Lokale Variablen haben in ihrem Gültigkeitsbereich Vorrang vor globalen Variablen. Weiterführende Erläuterungen sind im Abschnitt 4.6.3, "Extern" zu finden.

(4a) In diesem Bereich werden sog. Prototypen für Funktionen aufgelistet. In C wird für Programmteile, die eine definierte Teilaufgabe ausführen, der Begriff *function* verwendet. In diesem Dokument wird die direkte deutsche Übersetzung *Funktion* verwendet.

Eine Funktion ist ein Teilprogramm, das im Programmablauf aufgerufen wird, eine bestimmte Aufgabe eigenständig ausführt und dann die Kontrolle wieder an die aufrufende Funktion zurückgibt. Sind diese Teilaufgaben auf verschiedene, getrennt voneinander compilierbare Dateien verteilt, so nennt man jede Datei ein *Modul* und nicht Funktion.

Es handelt sich bei einer Funktion um ein Unterprogramm, nicht um eine mathematische Funktion. In C sind alle Funktionen gleichberechtigt, das heißt, jede Funktion kann jede andere Funktion, mit Ausnahme von `main()`, aufrufen. Ruft eine Funktion sich selbst auf, so nennt man das *Rekursion*.

Die Prototypen dienen zunächst einmal dem Compiler als Prüfliste für sämtliche Funktionsaufrufe im Hinblick auf korrekte Anzahl und Typisierung von Parametern. Funktionsprototypen werden durch ein Semikolon nach der Parameterliste abgeschlossen. Der eigentliche Funktionscode kann nun folgen (4b) oder am Ende der Datei (4c).

(4b) Eigene Funktionen können vor der Hauptfunktion `main()` stehen, allerdings spricht einiges dagegen (siehe auch 7.0, "Eigene Funktionen").

(4c) Eigene Funktionen sollten nach `main()` aufgeführt werden und stehen somit hinter der Hauptfunktion. Auf jeden Fall benötigen diese Funktionen einen Prototypen (4a).

(5) `main()` ist ein Beispiel für eine solche Funktion. Die Besonderheit besteht darin, daß `main()` die Verbindung zum Betriebssystem herstellt. Bei Aufruf des Programms von der Betriebssystemebene aus wird die Funktion `main()` gestartet. Diese gibt die Kontrolle an untergeordnete Funktionen weiter und erhält sie zum Schluß zurück, um sie an das Betriebssystem zurückzugeben. Des weiteren ist `main()` die einzige Funktion, die Kommandozeilenparameter empfangen kann.

(6) Die geschweiften Klammern `{ }` (engl. curly braces,) fassen in C mehrere Anweisungen zu einem Block zusammen und können diese einem Befehl (z.B. `if`) oder einer Funktion zuordnen. Wird der Block einer Funktion zugeordnet, so können in dem Block auch Variablen deklariert werden. Die Funktion der Klammern ähnelt der von `Begin` und `End` in Pascal oder Cobol. Jede Anweisung (Befehl/Schlüsselwort oder Funktionsaufruf) muß durch ein Semikolon abgeschlossen werden.

3.4.1 Der Einsatz von Kommentaren

Der C−Compiler ist an einer Formatierung des Quellcodes nicht interessiert. Er kümmert sich nicht darum, wie der Text im Editor eingegeben wurde. Er liest von Semikolon zu Semikolon und von Funktion zu Funktion. Dabei ist auch die Menge der Leerzeichen unwichtig, sie werden einfach bis auf jeweils eins überlesen. Da der Compiler keinerlei Anforderungen an die Art und Weise der Formatierung des Quellcodes stellt, ist die Gefahr groß, für Menschen schlecht lesbaren Code zu schreiben.

Ein Beispiel:

```
#include<stdio.h>
#include<math.h>
#include<float.h>
#define FALSE 0
void main(void){float a,xalt,xneu;printf("%s,%s,%s","Berechnung der
Quadratwurzel einer Zahl a\n","nach einem Iterationsverfahren:\n\n",
"a=\n");scanf("%f",xneu=a;do{xalt=xneu;
xneu=(xalt+a/xalt)/2.0;}while(fabs(xalt−xneu)>FLT_EPSILON);
printf("\nWurzel aus a= %8.4f\n",xneu);}
```

Selbst dieses kurze Programm ist nicht ohne weiteres für den, der es nicht selbst geschrieben hat, verständlich. Durch einfache und wenig aufwendige Maßnahmen, insbesondere durch Kommentare, läßt sich dieses Programm in der Lesbarkeit stark verbessern. Die Befehle werden in späteren Kapiteln erläutert.

Das oben abgebildete Beispiel in formatierter Form:

```
/* wurzel.c
------------------------------------------------------------
Author: Daniel Wolkenhauer, Sascha Kliche
Datum : September 1994
Datei : WURZEL.C
Beschreibung:
berechnet die Quadratwurzel einer Zahl gemäß einer von
Archimedes stammenden Iterationsformel
------------------------------------------------------------
*/

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

#define FALSE 0

void main (void)
{

   float a, xalt, xneu;

   printf("%s,%s,%s",
          "Berechnung der Quadratwurzel einer Zahl a\n",
          "nach einem Iterationsverfahren :\n\n",
          "a = \n");
   scanf("%f",

   xneu=a;

   do
   {
      xalt = xneu;
```

```
        xneu = ( xalt + a/xalt) / 2.0;
      } while (fabs ( xalt - xneu ) > FLT_EPSILON);

      printf("\n Wurzel aus a= %8.4f\n", xneu);

   }  /* Ende von main */
```

3.4.1.1 Gestaltung von Kommentaren

Im folgenden werden einige Beispiele für die mögliche Verwendung von Kommentaren im Quellcode gezeigt. Zu beachten sind hierbei die Zeichenfolgen `/*` und `*/`, die Anfang und Ende des Kommentars kennzeichnen.

```
/* Kommentare */

/*************************************************************/
/* Trennlinie als Kommentar */

/*
 * Kommentar kann auch in dieser
 * Form geschrieben werden, um ihn
 * vom Code abzuheben.
 */

/*****************************************************
 *                                                   *
 *        Kommentar in einer Kommentarbox            *
 *                                                   *
 *****************************************************/
```

Manche Compiler erlauben auch die Verwendung der Zeichenfolge `//` zur Einleitung eines einzeiligen Kommentars. Diese, der Sprache C++ entstammende Notation läßt sich bei dem IBM C/C++ und fast allen anderen modernen Compilern problemlos in C Programmen verwenden.

```
// Jeglicher Text nach den Schrägstrichen wird als
// Kommentar interpretiert.
```

3.4.1.2 Was sollte kommentiert werden?

Folgendes sollte unbedingt kommentiert werden:

- Dateiname, Versionsnummer, Autor, Datum, Datum der letzten Änderung, Zweck des Programms (zu Beginn)
- der Zweck der verwendeten Variablen (bei deren Deklaration)
- die Aufgabe von einzelnen Programmabschnitten, insbesondere von Funktionen
- besondere Anweisungen
- spezielle Algorithmen

Nicht kommentiert werden sollte, was ohnehin offensichtlich ist. In den Kommentaren sollten nicht nur die Namen der Variablen auftauchen, sondern ebenfalls ihre Aufgabe.

Zum Abschluß soll ein Beispiel den sinnvollen Einsatz eines Kommentars zur Information am Programmkopf veranschaulichen:

```
/*****************************************************************/
/*        Programm: comment.c                                    */
/*        Version : 4.2.6b                                       */
/*        Autor   : Sascha Kliche, Daniel Wolkenhauer            */
/*        Datum   : September 1994                               */
/* letzte Änderung: 02.09.1994, 14.37h EST                       */
/*    Beschreibung: Dieses Programm besteht aus diesem Teil und  */
/*                  zwei weiteren externen Modulen               */
/*                  (leerzeil.c/copyride.c). Es demonstriert die */
/*                  Verwendung von Kommentaren.                  */
/*****************************************************************/
```

3.4.2 Formatierung

Zusätzlich zum Einsatz von Kommentaren gibt es einige Möglichkeiten, die Lesbarkeit des Programmcodes zu steigern. Dazu gehört das Einfügen von Leerzeilen und Leerzeichen sowie eine klare Strukturierung (Prototypen vor main(), Funktionscode nach main()).

Grundsätzlich sollte jeder Variablendeklaration mind. eine Leerzeile folgen. Leerzeichen sollten dazu benutzt werden, eine bündige Formatierung der Zeilen sicherzustellen. Jede Verschachtelung sollte durch mind. zwei Leerzeichen nach rechts eingerückt werden.

Der folgende Programmausschnitt demonstriert den sinnvollen Einsatz von Kommentaren, Leerzeilen und Leerzeichen.

```
/*---------------------------------------------------------------*/
/*        Programm: menue.c                                      */
/*        Version : 1.0.0                                        */
/*        Autor   : Sascha Kliche                                */
/*        Datum   : April 1997                                   */
/* letzte Änderung: 02.05.1997, 14.00h EST                       */
/*    Beschreibung: Dieses Programm stellt Funktionen zum Zeichnen  */
/*                  und Abfragen von Bildschirmmasken zur        */
/*                  Verfügung, die Menüs mit Auswahlbalken       */
/*                  enthalten können.                            */
/*---------------------------------------------------------------*/


/*---------------------------------------------------------------*/
/* Include-Dateien und Makros/symbolische Konstanten definieren  */
/*---------------------------------------------------------------*/

#include <stdio.h>
#include <conio.h>
#include <os2.h>                              /* OS/2 CP Definition */

#define INCL_DOSGETDATETIME                   /* OS/2 CP Zeit/Datum */

#define ESC 27                                /* Tastaturcodes      */
#define cursor_up 72
#define cursor_down 80
#define cursor_left 75
#define cursor_right 77
#define page_up 73
#define page_down 81
```

```
#define home 71
#define end 79
#define insert 82
#define del 83


/*--------------------------------------------------------------------*/
/* Globale Variablen deklarieren                                      */
/*--------------------------------------------------------------------*/

/* Menüpunkte für Hauptauswahlmenü */
char mpunkt[5][25]={" 1.   Menüpunkt 1  \0"," 2.   Menüpunkt 2  \0",
                    " 3.   Menüpunkt 3  \0"," 4.   Menüpunkt 4  \0",
                    " 5.   Menüpunkt 5  \0"};

/* Farben für Textvorder- und -hintergrund; hintergrund+=10 */
enum cols { black=30, red, green, yellow,
            blue, magenta, cyan,  white } colors;


/*--------------------------------------------------------------------*/
/* Funktionsprototypen                                                */
/*--------------------------------------------------------------------*/

void gotoxy(int, int);          /* Cursor plazieren, x=1-80, y=1-25 */
void clrscr(void);                          /* Bildschirm löschen */

/*--- helle Textfarben setzen ---*/
void color(enum cols forcolors, enum cols backcolors);
/*--- matte Textfarben setzen ---*/
void mcolor(enum cols forcolors, enum cols backcolors);

/*--- Maske mit Zeit/Datum zeichnen ---*/
void maske(void);
/*--- Balken zeichnen; Position neuer balken, Position alter balken */
void balken(int, int);


/*--------------------------------------------------------------------*/
/* Anfang Hauptprogramm (main)                                        */
/*--------------------------------------------------------------------*/

void main(void)
{
  /*--- Deklaration lokaler Variablen ---*/
  int taste=0,           /* Tastencode gedrückte Taste  */
      position=0,        /* Position des Auswahlbalkens */
      i=0,               /* Zählvariable für Schleifen  */
      max=5-1;           /* Anzahl Menüpunkte           */

/*--- Bildschirmmaske "Haupschirm" ausgeben ---*/
  clrscr();
  color(white,blue);
  maske();
  gotoxy(3,23);
  printf("Beliebige Taste drücken. Ende mit x.");
  fflush(stdout);

  for(i=0;i<=max;i++)    /* Menüpunkte ausgeben */
  {
     gotoxy(5,8+i);
     printf("%24s",mpunkt[i]);
  }                      /* Ende Menüpunkte ausgeben */

  balken(0,0);           /* balken auf erste position setzen */
```

```
  ...
} /* ende main */

/*------------------------------------------------------------------*/
/* Ende Hauptprogramm (main)                                        */
/*------------------------------------------------------------------*/


/*------------------------------------------------------------------*/
/* Anfang Funktionen                                                */
/*------------------------------------------------------------------*/

 ...

/*------------------------------------------------------------------*/
/* Ende Funktionen                                                  */
/*------------------------------------------------------------------*/
```

3.5 Ausdrücke

Ausdrücke setzen sich aus einem oder mehreren Operanden und einem Operator zusammen. Operanden können Variablen oder Konstanten sein, wobei ein Ausdruck sowohl Variablen als auch Konstanten beinhalten kann. Die Operatoren werden im folgenden erläutert.

Jeder vergleichende Ausdruck (z.B. `if(tag==4)`) wird auf die beiden Zustände wahr bzw. falsch abgebildet. Da C keine Wahrheitsvariablen (sog. boolsche Variablen) unterstützt, werden diese Zustände wiederum auf Integerwerte abgebildet. Falsch wird auf den Wert Null abgebildet und wahr auf einen beliebigen Wert ungleich Null. Dies hat zur Folge, daß bei vergleichenden Ausdrücken auch direkt Wahrheitswerte benutzt werden können: z.B.

```
int tag=4;

if (tag) printf("Tag ist <> 0.\n");
else     printf("Tag ist gleich Null.\n");
```

Dementsprechend liefern folgende Ausdrücke unterschiedliche Wahrheitswerte:

```
    int tag=0;

(1) if (tag==0) ... else ...

(2) if (tag) ... else ...

(3) if (!tag) ... else ...
```

Ausdruck (1) prüft, ob `tag` den Wert Null enthält. Da dies der Fall ist, ist der Ausdruck wahr, also ungleich Null und der Ausdruck nach `if` wird ausgeführt. Ausdruck (2) wird als Wahrheitswert Null, also falsch, betrachtet. Dementsprechend wird der Ausdruck nach `else` ausgeführt. Ausdruck (3) wird als Wahrheitswert 'nicht falsch' (!0), also wahr, betrachtet. Ausgeführt wird der Ausdruck nach `if`. Alles klar?

3.6 Operatoren

Die Gruppe der Operatoren wird in vier Untergruppen aufgeteilt:

- a) arithmetische Operatoren
- b) unäre oder monadische Operatoren
- c) vergleichende Operatoren
- d) logische Operatoren
- e) sonstige Operatoren

Im Anschluß an die Erläuterung der einzelnen Operatoren wird die Hierarchie der Operatoren untereinander aufgezeigt.

3.6.1 Arithmetische Operatoren

Mit Hilfe der arithmetischen Operatoren sind Gleitkomma−, Festkomma− und ganzzahlige Ausdrücke zu verarbeiten. In C sind die folgenden fünf definiert:

```
                Addition       :  +
                Subtraktion    :  -
                Multiplikation :  *
                Division       :  /
              ( Restwert       :  % )
```

Der Restwert−Operator ist in Klammern gesetzt, weil er nur für ganzzahlige Ausdrücke verwendet werden darf. Hierbei liefert er den ganzzahligen Rest einer Division:

```
10 / 5 = 2      3 / 2.0 = 1.5     36 / 5.0 = 7.2
10 % 5 = 0      3 % 2   = 1       36 % 5   = 1
```

Wird der Divisions−Operator mit zwei Integer−Werten verknüpft, so liefert er als Ergebnis wieder einen ganzzahligen Wert zurück:

```
 8 / 2 = 4     10 / 4 = 2        1 / 2 = 0
```

Bei Rechenoperationen immer daran denken, dass das Ergebnis einer Berechnung einer Variablen zugewiesen werden muss, damit man später damit weiter arbeiten kann. Beispiel für die Addition zweier Zahlen (eine in der Variablen `zahl1` und eine als Konstante):

```
int ergebnis = 0, zahl1 = 5;

ergebnis = zahl1 + 8;
```

Der Ausdruck `zahl1 + 8;` allein wäre zwar zulässig, das Ergebnis würde aber keiner Variablen zugewiesen und damit verloren.

3.6.2 Unäre oder monadische Operatoren

Unäre oder monadische Operatoren beziehen sich nur auf einen Operaden. Normalerweise stehen diese vor dem Operanden, können aber auch hinter dem Operanden stehen.

Beispiele für diese Art von Operatoren sind

- das unäre Minus −
- der Autoinkrement−Operator ++
- der Autodekrement−Operator −−
- sizeof
- der Typecast

Steht der Autoinkrement−/Autodekrement−Operator vor dem Operanden, so handelt es sich um einen Präinkrement bzw. −dekrement. In diesem Fall wird der Wert des Operanden geändert, bevor er verwendet wird. Steht er nach dem Operanden, so handelt es sich um einen Postinkrement bzw. −dekrement. In diesem Fall wird der Wert des Operanden geändert, nachdem er verwendet wird.
Beispiele:

```
a = 10;  /* a wird der Wert 10 zugewiesen */
b = --a; /* b wird der Wert  9 zugewiesen, der Wert von a wird VOR der Zuweisung um 1 verrin
c = a++; /* c wird der Wert  9 zugewiesen, erst dann erhaelt a den wert 10 */
```

3.6.3 Vergleichende Operatoren

Die vergleichenden Operatoren prüfen auf Gleichheit und Ungleichheit und haben immer zwei Operanden:

```
a)     Gleichheit:              Ausdruck == Ausdruck          zahl1 == zahl2
b)     Ungleichheit:            Ausdruck != Ausdruck          zahl1 != zahl2
c)     Kleiner als:             Ausdruck < Ausdruck           zahl1 < zahl2
d)     Kleiner oder gleich:     Ausdruck <= Ausdruck          zahl1 <= zahl2
e)     Größer:                  Ausdruck > Ausdruck           zahl1 > zahl2
f)     Größer oder gleich:      Ausdruck >= Ausdruck          zahl1 >= zahl2
```

Die Relation liefert den wahr (true) oder falsch (false) zurück. Da es in C keine boolschen Variablen gibt, wird für wahr ein Wert ungleich Null (0) und für falsch der Wert Null (0) vom Typ Integer zurückgegeben.

Anmerkung zu a):	Das einfache Gleichheitszeichen (=) erhält in C die Aufgabe einer Zuweisung. Der linke Operator einer Zuweisung muß ein "Lvalue" (eine gültige Speicherstelle wie z.B. eine Variable) sein. Bsp.: `a = 4` weist a den Wert vier zu.

3.6.4 Logische Operatoren

Bei logischen Operatoren liefert die Relation den Wert wahr oder falsch zurück.

Die logischen Operatoren werden nicht zum Vergleich von Werten, sondern zum Vergleich von Wahrheitswerten verwendet:

```
a)     !      (logisches NICHT)     !(zahl>0 &zahl<10)
b)     &&     (logisches UND)       zahl>10 || zahl==0
c)     ||     (logisches ODER)      zahl>0 &zahl<10
```

Die Charakteristik der einzelnen Operatoren wird mit Hilfe der Wahrheitstabelle dargestellt (wahr=1, falsch=0).

zu a) Das logische NICHT ist ein unärer Operator. Das bedeutet, daß sich der Operator nur auf einen Operanden bezieht.

Tabelle 3. Wahrheitstabelle (logisches NICHT)

x	!x
falsch (0)	wahr (1)
wahr (1)	falsch (0)

zu b) Das logische UND verknüpft Ausdrücke folgendermaßen miteinander:

Tabelle 4. Wahrheitstabelle (logisches UND)

Wert 1	Wert 2	Ergebnis
falsch (0)	falsch (0)	falsch (0)
falsch (0)	wahr (1)	falsch (0)
wahr (1)	falsch (0)	falsch (0)
wahr (1)	wahr (1)	wahr (1)

zu c) Das logische ODER wird durch folgende Wahrheitstabelle dargestellt:

Tabelle 5. Wahrheitstabelle (logisches ODER)

Wert 1	Wert 2	Ergebnis
falsch (0)	falsch (0)	falsch (0)
falsch (0)	wahr (1)	wahr (1)
wahr (1)	falsch (0)	wahr (1)
wahr (1)	wahr (1)	wahr (1)

3.6.5 Sonstige Operatoren

Die sonstigen Operatoren passen in keine der vorangegangenen Kategorien und sind für unterschiedliche Zwecke gedacht. Ihre Bedeutung kann Tabelle 19 entnommen werden.

3.6.6 Zusammengesetzte Zuweisungsoperatoren

Die zusammengesetzten Zuweisungsoperatoren dienen nicht nur einer verkürzten Schreibweise, sie werden auch in schnelleren Assemblercode übersetzt als die jeweils äquivalenten Ausdrücken.

Tabelle 6. Zusammengesetzte Zuweisungsoperatoren

Operator	Beispiel	Äquivalenter Ausdruck
+=	index += 2	index = index + 2
−=	index −= 2	index = index − 2
*=	index *= 2	index = index * 2
/=	index /= 2	index = index / 2
%=	index %= 2	index = index % 2
<<=	result <<= num	result = result << num
>>=	form >>= 1	form = form >> 1
=	mask =2	mask = mask 2
^=	test ^= pre_test	test = test ^pre_test
|=	flag |= on	flag = flag | on

3.6.7 Hierarchie der Operatoren

Beim Zusammentreffen mehrerer Operatoren in einer Relation muß definiert sein, welche Vorrangregeln gelten. Kommen mehrere Operatoren gleicher Priorität zusammen, so ist in der nachstehenden Tabelle die Auswertungsrichtung festgeschrieben.

Selbst wenn sich jemand diese Prioritäten merken kann, sollten zur besseren Lesbarkeit Klammern gesetzt werden.

Tabelle 7. Tabelle der Hierarchie der Operatoren

Operatoren	Verwendung	Auswertungsrichtung
( ) [] −> .	bildet einen Ausdruck bezeichnet ein Feldelement wählt eine Strukturkomponente aus wählt eine Strukturkomponente per Zeiger aus	links nach rechts
++ −− − ! ~ & * sizeof (typ)	Inkrementieren Dekrementieren bildet negativen Wert logische Negierung bitweises Negieren liefert Adresse einer Variablen/Konstanten greift über Zeiger auf Variable/Konstante zu liefer Größe eines Speicherbereiches wandelt in angegebenen Typ um (type cast)	rechts nach links
* / %	Multiplizieren Dividieren Restwert bilden	links nach rechts
+	Addieren	links nach rechts

−	Subtrahieren	
<< >>	verschiebt Bits nach links verschiebt Bits nach rechts	links nach rechts
< <= > >=	Vergleich auf kleiner Vergleich auf kleiner gleich Vergleich auf größer Vergleich auf größer gleich	links nach rechts
== !=	Vergleich auf gleich Vergleich auf nicht gleich	links nach rechts
&	Bitweises UND	links nach rechts
^	Bitweises XOR	links nach rechts
|	Bitweises ODER	links nach rechts
&	Logisches UND	links nach rechts
||	Logisches ODER	links nach rechts
? :	Bedingung	rechts nach links
= += −= *= /= <<= >>= &= ^= |= %=	einfache Zuweisung Addieren und Zuweisen Subtrahieren und Zuweisen Multiplizieren und Zuweisen Dividieren und Zuweisen Bits nach links verschieben und zuweisen Bits nach rechts verschieben und zuweisen Bitweise UND verknüpfen und zuweisen Bitweise XOR verknüpfen und zuweisen Bitweise ODER verknüpfen und zuweisen Restwert bilden und Zuweisen	rechts nach links
,	Operanden trennen	links nach rechts

3.6.8 Beispiele zur Verknüpfung der Operatoren

Im folgenden sind einige Beispiele für die Verknüpfung mehrerer Operatoren und deren Bedeutung aufgeführt:

```
a)     (c >= 'a') &(c <= 'z')     prüft, ob ein Zeichen ein Kleinbuchstabe ist
b)     (d >= 0) &(d <= 9)         prüft, ob ein Zeichen eine Ziffer ist
c)     (x > 0) &(y > 0)           prüft, ob x− und y−Koordinate positiv sind
```

3.7 Bezeichner

Ein Bezeichner dient dazu, einer Variablen oder einer Konstanten einen eindeutigen Namen zuzuweisen. Variablen und Konstanten dienen zum Speichern und Bearbeiten von Daten, wobei die Daten von Konstanten während des Programmlaufes nicht geändert werden können. In C können Konstanten für die Datentypen der ganzen Zahlen, der rationalen Zahlen und der Zeichen definiert werden. Der Deklaration von Konstanten geht das Schlüsselwort `const` voran. Variablen und Konstanten sind Objekte, die mit Hilfe ihrer Namen (Bezeichner), ihrer Datentypen und ihrer Größe in Byte einen bestimmten Speicherbereich im Hauptspeicher ansprechbar macht. Für die Vergabe der Bezeichner gibt es in C bestimmte Restriktionen:

1. Es dürfen nur Buchstaben, Ziffern und der Unterstrich benutzt werden, keine Umlaute und keine

Sonderzeichen. Der Unterstrich sollte nach Möglichkeit nur für eigene Typdefinitionen, nicht aber für eigentliche Variablen oder Konstanten benutzt werden, da dieser i.d.R. kennzeichnet, daß der Bezeichner von der Entwicklungsumgebung selbst bereitgestellt wird.

2. Das erste Zeichen muß immer ein Buchstabe oder der Unterstrich sein.

3. Ein Bezeichner darf beliebig lang sein. Allerdings sind nach ANSI−C nur die ersten 31 Zeichen signifikant, das heißt der C−Compiler läßt weitere Zeichen unberücksichtigt.

4. Es werden Klein− und Großbuchstaben unterschieden.

5. Die unter Punkt 3.3, "Schlüsselworte" angegebenen Schlüsselwörter dürfen nicht als Bezeichner verwendet werden, da der Compiler sonst nicht unterscheiden kann, ob eine Variable bzw. Konstante oder ein Schlüsselwort gemeint ist.

6. Ein Bezeichner darf in einem Anweisungsblock (eingegrenzt durch geschweifte Klammern) nur einmal vergeben werden. Es ist zwar möglich, in jedem Anweisungsblock gleiche Bezeichner zu benutzen, dies sollte i.d.R. aus Gründen der Übersichtlichkeit und Verständlichkeit jedoch vermieden werden, wenn es sich nicht um eine Zählvariable handelt.

7. In C dürfen Variablen und Konstanten nur am Blockanfang (d.h. unmittelbar nach einer öffnenden geschweiften Klammer) deklariert werden.

Beispiele:

```
Richtig                    Falsch

wort                       õwort
buchSTABE56                2buchSTABE
skanf_var                  scanf
```

Die verschiedenen, in C zur Verfügung stehenden, Datentypen werden im Kapitel 4.0, "Datentypen" vorgestellt.

4.0 Datentypen

Die Aufgabe von Datentypen besteht darin, Speicherbereiche in Bezug auf ihre Inhalte zu interpretieren. Der Inhalt dieses Speicherbereiches ist zum Beispiel der Wert einer Variablen. Der Compiler benötigt nun Informationen darüber, wieviel Speicherplatz er für die Variable reservieren muß. Dazu muß der Programmierer ihm den Datentyp der Variablen mitteilen.

Die Länge in Bytes, die die einzelnen Datentypen im Speicher beanspruchen, hängt von der verwendeten Hardware und Softwareentwicklungsumgebung ab. Die jeweiligen Größen sind in den Dateien `LIMITS.H` und `FLOAT.H` der verwendeten Entwicklungsumgebung festgelegt. Alle hier angegebenen Größen gelten für den IBM C/C++ Compiler. Die Ermittlung der dezimalen Zahlen aus den binären Angaben erfolgt mittels 2^x, wobei x die Anzahl der verwendeten Bits darstellt.

Wie viele andere Programmiersprachen auch, verfügt C über *einfache Datentypen* und *höhere Datentypen*. Bei den einfachen Datentypen unterscheidet man

- die ganzen Zahlen (Integer),
- die rationalen Zahlen (Float)
- und die Textzeichen (Character)

Alle diese Datentypen können in jeweils unterschiedlichen Formen auftreten.

Bei den höheren Datentypen unterscheidet man

- Felder (Array),
- Strukturen,
- Unions und
- Pointer.

Höhere Datentypen werden häufig auch als *komplexe* oder *strukturierte* Datentypen bezeichnet. Felder und Strukturen werden auch als zusammengesetzte Datentypen bezeichnet.

4.1 Einfache Datentypen

In diesem Abschnitt werden die einfachen Datentypen, ihr Verwendungszweck und ihre Deklaration erläutert.

4.1.1 Ganze Zahlen − Integer

Der Wertebereich der ganzen Zahlen besteht aus den natürlichen Zahlen. Ganze Zahlen können nicht nur in der üblichen dezimalen Schreibweise (z.B. 18), sondern z.B. auch in hexadezimaler Schreibweise (z.B. 0xFFFF bzw. 0XFFFF) angegeben werden.

Alle Integertypen lassen sich mit Vorzeichen (`signed`) oder ohne Vorzeichen (`unsigned`) definieren.

Die Integertypen sind weiter untergliedert in

- short int
- int
- long int

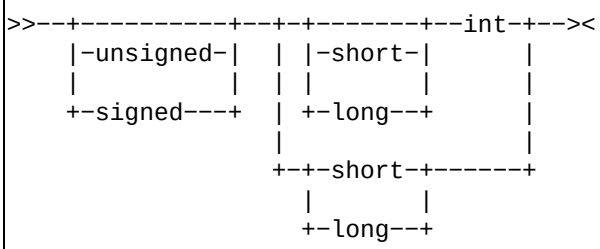
Die Untergliederung in die einzelnen Typen erfolgt HW− und Betriebssystemspezifisch nach der Anzahl Bits, die für die Speicherung der Daten verwendet wird. Wieviel Bits verwendet werden entscheidet aber letztendlich der Compilerhersteller. Die Header−Datei `limits.h` legt fest, welche Wertebereiche der Compiler für die einzelnen Integertypen verwendet.
Es gibt keine fest vorgegebenen Wertebereiche für die einzelnen Integer−Typen. Der Standard schreibt lediglich Mindestgrössen vor:

short int	16 Bit
int	16 Bit
long	32 Bit
long long	64 Bit

Weiterhin gilt `sizeof(short int) <= sizeof(int) <= sizeof(long int)`
Die im folgenden angegebenen Wertebereiche richten sich nach den unter 32−Bit Betriebssystemen üblicherweise verwendeten Grenzen (d.h. `int` mit 32 Bit Größe).

Syntaxdiagramm für die Integertypen:

Integertypen



4.1.1.1 short int

Für Diesen Datentyp werden zwei Byte (16 Bit) reserviert. Daraus resultiert, daß `signed short int` einen Wertebereich von −32.768 bis 32.767 (-2^{15} bis 2^{15}, das 16. Bit wird für das Vorzeichen verwendet) und `unsigned short int` einen Wertebereich von 0 bis 65.535 (2^{16}) zur Verfügung stellen.

Deklarationsbeispiele:
`short int x;` oder `signed short int x;`
Vorzeichenlos: `unsigned short int x;`

4.1.1.2 int

Die Variablen des Typs `int` werden mit vier Byte (32 Bit) dargestellt und haben einen Wertebereich von −2.147.483.648 bis 2.147.483.647 und somit 2^{32} Werten. Mittels der symbolischen Konstante `INT_MIN` kann der kleinste Wert (−2.147.483.648) angesprochen werden.

Deklarationsbeispiele:
`int x;` oder `signed int x;`
Vorzeichenlos: `unsigned int x;`

Der Wertebereich dieses Datentyps erstreckt sich von 0 bis 4.294.967.295.

4.1.1.3 long int

Der Wertebereich des Typs `long int` entspricht dem Wertebereich von `int`, da auch dieser Typ mit vier Byte (32 Bit) dargestellt wird.

Deklarationsbeispiele:
`long int x;` oder `signed long int x;`
Vorzeichenlos: `unsigned long int x;`

Einer Konstante des Typs `long int` folgt unmittelbar das Zeichen L, z.B. `518346L`, um zu kennzeichnen, daß es sich nicht um eine Konstante des Typs `int` handelt.

4.1.1.4 long long int

Dieser Datentyp ist nicht vom ANSI C Standard von 1989 abgedeckt, steht aber unter einigen Compilern zur Verfügung und ist im Standard von 1999 aufgenommen. "IBM C and C++ Compilers" definiert diesen Typ mit einer Länge von 8 Bytes und umfasst somit 2^{64} Werte.

Deklarationsbeispiele:
`long long int x;` oder `signed long long int x;`
Vorzeichenlos: `unsigned long long int x;`

Einer Konstante des Typs `long long int` folgen unmittelbar die Zeichen LL, z.B. `518346LL`, um zu kennzeichnen, daß es sich nicht um eine Konstante des Typs `int` handelt.

4.1.2 Rationale Zahlen / Fließkommazahlen

Bei den rationalen Zahlen unterscheidet man ebenfalls verschiedene Genauigkeitsstufen, Größen− und Wertebereiche:

- `float`,
- `double` und
- `long double`

`double` wurde erst durch den ANSI−C−Standard definiert, vorher hieß dieser Datentyp `longfloat`. Aus diesem Grund ist der Bezeichner für die Formatspezifikation nach wie vor `%lf`.

Die Header−Datei `float.h` legt fest, welche Wertebereiche der Compiler für die einzelnen Integertypen verwendet.

Bei Operationen mit rationalen Zahlen muß mindestens eine Zahl ein Komma enthalten, da sonst in Integer gerechnet wird!
Die Rechnung `ergebnis = 84 / 123;` liefert keine Nachkommastellen, da mit Integerkonstanten gerechnet wird und das Ergebnis dementsprechend auch vom Typ Integer ist. Benötigt man die Nachkommastellen, so liefert `ergebnis = 84.0 / 123;` das korrekte Ergebnis.

Variablen für rationale Zahlen sollten stets bei ihrer Deklaration initialisiert werden, da für sie nur bestimmte Bitkombinationen erlaubte Werte darstellen. Wird dies nicht beachtet, kann es zu Programmabbrüchen kommen.

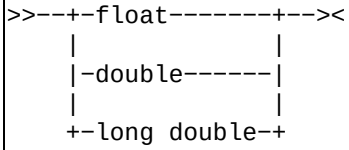
Es gibt drei verschiedene Möglichkeiten der Darstellung von Fließkommazahlen:

1. die Festkomma−Darstellung (z.B. 3.456),
2. die Gleit− oder Fließkomma−Darstellung (z.B. 3.423e7) und
3. die Darstellung, bei der der Compiler entscheidet, welche die kürzere ist.

Die Darstellungsform hat nur bei der Ein−/Ausgabe von rationalen Zahlen Auswirkungen und wird über den Formatstring festgelegt.

Syntaxdiagramm für die Fließkommatypen:

Fließkommatypen



4.1.2.1 Der Typ float

Der Typ `float` wird mit 32 Bit (4 Byte) dargestellt und ergibt einen Wertebereich von $1.17549*10^{-38}$ bis $3.40282*10^{38}$ (+/−).

Die Deklaration hat folgende Form:
```
float x=0.0;
```

Dieser Typ ist nicht für präzise Berechnungen geeignet, da seine Genauigkeit [3] (7 Bit) bereits nach wenigen Nachkommastellen nachläßt. In diesem Fall sind die Typen `double` bzw. `long double` zu benutzen. Beispiel:

```
float zahl1=0.0, zahl2=0.0, ergebnis=0.0;

scanf("%f+%f",
ergebnis=zahl1+zahl2;
printf("%f+%f=%f",zahl1,zahl2,ergebnis);

// Eingabe: 85.439+234.34
// Ausgabe: 85.439003+234.339996=319.778992
```

4.1.2.2 Der Typ double

Der Typ `double` wird mit 64 Bit (8 Byte) dargestellt und ermöglicht einen Wertebereich von $2.22507*10^{-308}$ bis $1.79769*10^{308}$ (+/−).
Die Deklaration hat folgende Form: `double x=0.0;`

Der Typ `double` hat mit 15 Bit eine wesentlich höhere Genauigkeit als `float` und sollte diesem grundsätzlich vorgezogen werden.

4.1.2.3 Der Typ long double

Der Typ `long double` wird mit 128 Bit (16 Byte) dargestellt und hat einen Wertebereich von $3.36*10^{-4932}$ bis $1.1897*10^{4932}$ (+/−).
Die Deklaration hat folgende Form: `long double x=0.0;`

4.1.2.4 Floating Point Konstanten

Floating Point Konstanten bestehen aus

- einem integralen Teil
- einem Dezimalpunkt
- einem Bruchteil
- einem Exponenten
- einem optionalen Suffix

Ein Plus oder ein Minus kann der Konstante vorangehen, ist jedoch nicht Teil der Konstante.

```
 Floating Point Konstante                    Wert

      5.3876e4                               53876
        4e-11                          0.00000000004
        1e+5                              100000
      7.321E-3                           0.007321
       3.2E+4                              32000
       0.5e-6                          0.0000005
        0.45                                0.45
       6.e10                          60000000000
```

4.1.3 Textzeichen − char

Ein Textzeichen wird in C mit 8 Bit verarbeitet. Dies ergibt $2^8 = 256$ verschiedene Zeichen. Diesen 256 Möglichkeiten sind international geläufige Zeichen zugeordnet. Diese gliedern sich wie folgt:

```
Zeichen 0   ...  31   Steuerzeichen
Zeichen 32  ...  47   Sonderzeichen
Zeichen 48  ...  57   Ziffern 0-9
Zeichen 58  ...  64   Sonderzeichen
Zeichen 65  ...  90   Großbuchstaben
Zeichen 91  ...  96   Sonderzeichen
Zeichen 97  ... 122   Kleinbuchstaben
Zeichen 123 ... 127   Sonderzeichen
Zeichen 128 ... 255   Zeichen des erweiterten Zeichensatzes
```

Die Zeichen 128 bis 255 hängen vom jeweiligen System und der verwendeten Code−Tabelle ab. Diese Zeichen umfassen i.d.R. nationale Umlaute und Sonderzeichen. Unter Windows wird für Kommandozeilenprogramme ASCII [4] verwendet, für die Programme der grafischen Oberfläche jedoch ANSI

[5] oder Unicode.

Im C Quellcode sollten immer die entsprechenden Zeichen anstatt der Zeichencodes (z.B. '2' statt 50) verwendet werden, da auf vielen Systemen andere Zeichensätze verwendet werden und die Zeichencodes dort eine andere Bedeutung haben. Weitere verbreitete Zeichensätze sind z.B. EBCDIC [6] (IBM Großrechnersysteme), ANSI bzw. ISO Latin 1 (z.B. Windows, UNIX) und Unicode (z.B. Windows, UNIX).

Eine Ausnahme stellt die Behandlung von Sonderzeichen (Umlaute, ...) dar. MS Windows benutzt z.B. für Konsolenprogramme den ASCII Zeichensatz, für Programme der graphischen Oberfläche jedoch den ANSI Zeichensatz. Werden Konsolenprogramme mit einem Editor erstellt, der auf der graphischen Oberfläche läuft, können die Umlaute über ihre Zahlenwerte erreicht werden. Diese Werte sind jedoch abhängig von der Sprache und dem Zeichensatz! Beispiel für die deutschen Umlaute im Extended ASCII Zeichensatz:

```
ä : \x84
Ä : \x8E
ö : \x94
Ö : \x99
ü : \x81
Ü : \x9A
ß : \xE1

printf("ae: \x84\nAe: \x8E\noe: \x94\nOe: \x99\nue: \x81\nUe: \x9A\nsz: \xE1\n");
bzw. (da die obere Variante innerhalb von Wörtern Probleme bereitet)
printf("ae: %c\nAe: %c\noe: %c\nOe: %c\nue: %c\nUe: %c\nsz: %c\n",
       \x84, \x8E, \x94, \x99, \x81, \x9A, \xE1);
```

Einfacher ist dagegen die Arbeit mit einem Editor, der beim Speichern die Möglichkeit bietet, den Zeichensatz zu wählen. In diesem Fall kann man die Umlaute ganz normal eingeben und wählt beim Speichern als Zeichensatz DOS bzw. ASCII.

Die Deklaration von Variablen oder Konstanten des Typs `char` hat folgende Form: `unsigned char buchstabe;`
`char` muß `unsigned` deklariert werden, wenn Zeichen mit einem Code größer 127 benutzt werden sollen! Wird `char buchstabe;` deklariert, können nur Zeichen mit Codes von 0 bis 127 benutzt werden. Manche Entwicklungsumgebungen definieren `char` grundsätzlich als `unsigned char`.

Achtung!
Der Datentyp `char` kann keine Zeichenketten (z.B. Wörter) aufnehmen, nur einzelne Zeichen.

Es ist sowohl möglich Zeichen zu vergleichen als auch mit Zeichen zu rechnen. Somit ist z.B. folgendes erlaubt:

```
    if (zeichen>='a' &zeichen<='z') ...
```

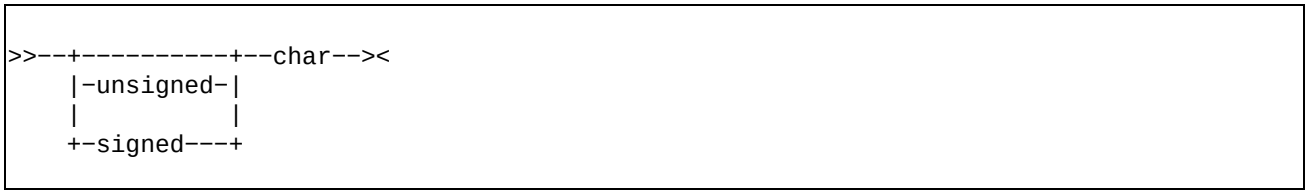
oder auch

```
    position = zeichen - 'a';
```

Da einige Zeichensätze wie z.B. Unicode wesentlich mehr als nur 256 Zeichen verwalten, existiert noch der Typ `wchar_t`, der mit bis zu vier Byte (Standard sind 2 Byte) dargestellt wird, um universelle Zeichensätze abbilden zu können. Eine `wchar_t` Konstante lautet z.B. `L'A'` anstatt `'A'`.

Syntaxdiagramm für die Zeichentypen:

Chartypen



4.1.3.1 Zeichenkonstanten und Zeichenkettenkonstanten

Konstanten sind Werte, die sich während des Programmdurchlaufs nicht ändern sollen. Sie können bei jedem Datentyp auftreten. Bei den Zeichen− und den Zeichenkettenkonstanten ist jedoch eine Besonderheit zu beachten.

(1) Zeichenkonstanten werden in einfache Hochkommata eingeschlossen.
Beispiele:

```
'd'
'C'
'2'
```

Diese Zeichenkonstanten werden zum Beispiel in der `switch`−Anweisung (siehe dazu Abschnitt 5.2, "switch") und in Vergleichen verwendet.

(2) Zeichenkettenkonstanten werden dahingegen in Anführungszeichen eingefaßt.
Beispiele:

```
"Fehler beim Öffnen der Datei aufgetreten!"
"rc=0"
"Bitte eine Zahl eingeben!"
```

Achtung!
Es ist nicht möglich, Zeichenketten mittels des Vergleichsoperators `==` zu vergleichen!

Zeichenkettenkonstanten finden besonders häufig in der `printf`−Anweisung (8.1, "printf − formatierte Ausgabe auf dem Bildschirm") Verwendung. Sie können beliebige Zeichen enthalten und laut ANSI−C−Standard 509 Zeichen lang sein. Diese Zahl ist jedoch abhängig vom verwendeten Compiler. Es können auch sog. Fluchtsymbolzeichen (8.1.3, "Fluchtsymbolzeichen") aufgenommen werden. Diese gelten nur als ein einzelnes Zeichen. Beim Ablegen im Speicher wird das Ende einer Zeichenkettenkonstanten durch eine binäre Null (\0) gekennzeichnet, die automatisch angefügt wird.

Zeichenkettenkonstanten werden in C vom Typ `char *` behandelt, in C++ vom Typ `const char *`.

4.2 Höhere Datenstrukturen

In diesem Abschnitt werden die höheren Datentypen Feld, Struktur, Union und Pointer anhand von ausführlichen Beispielen vorgestellt.

4.2.1 Felder − Array − Vektoren

Felder sind der erste Variablentyp der höheren Datentypen. Sie stellen eine aufeinanderfolgende Anordnung von Elementen des gleichen Typs dar. Der Zugriff auf die Elemente erfolgt mit Hilfe eines Index. Dabei

umfaßt der Wertebereich des Index die positiven ganzen Zahlen inklusive der Null, d.h. Felder haben einen 0−basierten Index (z.B. 8 Feldelemente, Index von 0 bis 7.) Eindimensionale Felder haben genau einen Index, mehrdimensionale zwei oder mehr Indizes.
Felder werden auf dem Stack angelegt. Da dessen Größe beschränkt ist, sollten große Felder mit dynamischer Speicherverwaltung (siehe 15.0, "Speicherverwaltung") verwaltet werden um Probleme zu vermeiden.

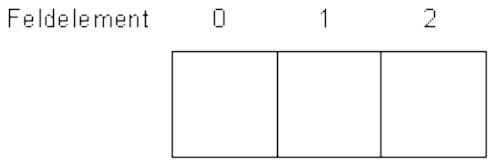
4.2.1.1 Eindimensionale Felder

Ein Feld wird über seine Anfangsadresse im Speicher, die Größe der Elemente und die Menge der Elemente vollständig definiert.
Beispiel:

```
int x[3];
```

definiert ein Feld x mit drei Elementen vom Typ Integer und der Größe 3*sizeof(int) Byte (i.d.R. 3*4 Bytes).

Abbildung 2. Beispiel für ein eindimensionales Feld



Die Deklaration `int x[3];` bewirkt, daß die Elemente `x[0], x[1]` und `x[2]` zur Verfügung stehen. `x` ist dabei definiert als `x = t>. Es beinhaltet die Speicheradresse des ersten Elementes x[0], also einen Zeiger auf die Position des Inhaltes der ersten Komponente im Speicher.`

4.2.1.2 Mehrdimensionale Felder

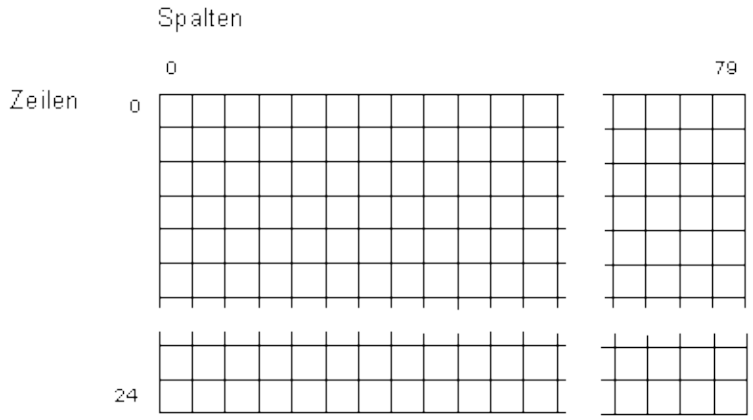
Für mehrdimensionale Felder werden mehrere Indizes verwendet. Typisch ist das zweidimensionale Feld, es wird auch *Matrize* oder *Tabelle* genannt.

Die Definition mehrdimensionaler Felder erfolgt im Gegensatz zu den meisten Programmiersprachen über mehrere, voneinander getrennte, eckige Klammern. C speichert (im Gegensatz zu Fortran) Felder in der sog. Row Major Order (sprich: Zeile hinter Zeile).
Beispiele:

```
(1)        int bildschirm [25] [80];
```

Mit dieser Anweisung wird ein zweidimensionales Feld mit dem Namen *bildschirm* erstellt. Es beinhaltet 25 Zeilen und 80 Spalten, genügend Platz, um einen kompletten Textbildschirm aufzunehmen:

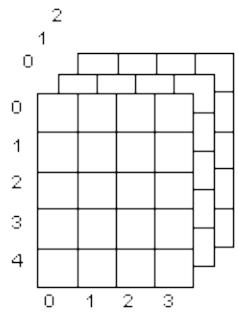
Abbildung 3. Beispiel für ein zweidimensionales Feld




```
(2)        int karten [5][4][3];
```

vereinbart ein dreidimensionales Feld mit den Dimensionen fünf, vier und drei:

Abbildung 4. Beispiel für ein dreidimensionales Feld

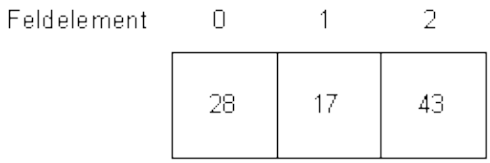


4.2.1.3 Initialisierung und Ansprache von Feldern

Die folgenden Beispiele illustrieren die Initialisierung ein−, zwei− und dreidimensionaler Felder bei deren Deklaration. Am Ende dieses Abschnittes befindet sich ein Beispiel für die Initialisierung eines zweidimensionalen Feldes mittels einer Schleife.

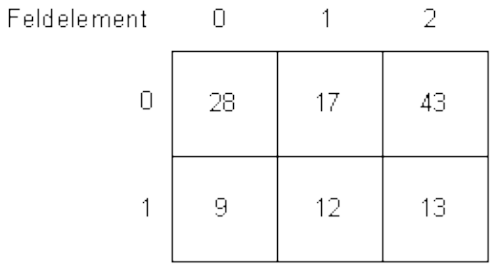
```
(1)   int datensatz[3] = {28,17,43};

                              /*       Feldelement 0: 28;
                                       Feldelement 1: 17;
                                       Feldelement 2: 43;       */
```

Abbildung 5. Initialisierung eines eindimensionalen Feldes



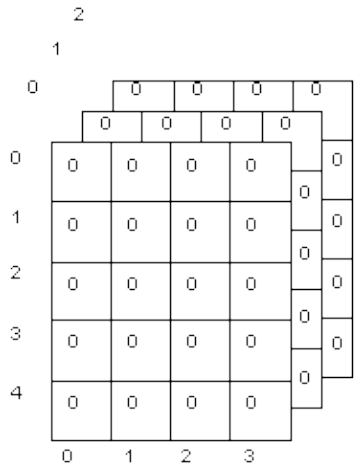

```
(2)   int matrix[2][3] = {{28,17,43},{9,12,13}};

                        /* Feldelemente 0,0 - 0,2: 28,17,43        */
                        /* Feldelemente 1,0 - 1,2: 9,12,13         */
```

Abbildung 6. Initialisierung eines zweidimensionalen Feldes




```
(3)   int karten[5][4][3] = {0};   /* alle Feldelemente werden 0 * /
```

Abbildung 7. Initialisierung eines dreidimensionalen Feldes



4.2.1.4 Ansprache von Feldelementen

Bei der Ansprache von Feldern muß berücksichtigt werden, daß einzelne Elemente angesprochen werden und nicht das gesamte Feld auf einmal.

Beispiel:

Dem Element '0,2' eines Feldes a, das zunächst mittels einer Schleife initialisiert wird, wird ein Wert zugewiesen:

```
int a[3][8], i, j;        /* Deklaration eines zweidimensionalen Feldes mit */
                          /* drei Zeilen mit je acht Spalten                */

for(i=0; i < 3; i++)      /* Start der äußeren Schleife für die Zeilen      */
{
   for(j=0; j < 8; j++)   /* Start der inneren Schleife für die Spalten     */
   {
       a[i][j] = 0;       /* Initialisierung der Spalten + Ende der inneren */
   }
}                         /* Ende der äußeren Schleife                      */
a[0][2] = 6;              /* Zuweisung des Wertes                           */
```

4.2.1.5 Übergabe von Feldern an Funktionen

Bei der Übergabe eines Feldes an eine Funktion (siehe auch Kapitel 7.0, "Eigene Funktionen") ist zu beachten, daß nicht das gesamte Feld kopiert und übergeben werden sollte, sondern nur die Adresse des ersten Elementes. Dies hat zur Folge, daß die Menge der Elemente als eigenständiger Wert übergeben werden muß, was die Angabe der Menge der Elemente innerhalb der Klammern ([ ]) überflüssig macht.

Beispiel:

```
/*********************************************************************/
/*** SUMME.C      summiert die Werte eines Feldes und übergibt das ***/
/***              Ergebnis an die aufrufende Funktion              ***/
/*********************************************************************/
#include <stdio.h>
double summieren (double a[], int n)        /* Deklarationskopf der    */
{                                           /* Funktion 'summieren'    */
   int i;                                   /* Deklaration der lokalen */
   double summe=0.0;                        /* Variablen i und summe   */

   for (i=0; i<n; i++)                      /* Start der Schleife      */
   {
        summe = summe + a[i];               /* Werte aufsummieren      */
   }

   /* for (i=0; i<n; summe +=a[i++]);          wäre auch möglich       */

   return(summe);                           /* Summe zurückgeben       */
}                                           /* Ende der Funktion       */

void main(void)                             /* Start des Hauptprogramms*/
{
   double x1[3] = {2, 4, 9};                /* Deklaration und Initi-  */
                                            /* alisierung des Feldes   */
   printf("\nSumme %f\n", summieren(x1,3)); /* Funktionsaufruf und Aus-*/
                                            /* gabe des Ergebnisses    */
```

```
}
```

Bei der Übergabe von Listen (zweidimensionale Felder) sieht es ähnlich aus:

```
int printListe(char liste[][50], int maxnr)
{
   int i=0;

   for (i=0; i<maxnr; i++)
   {
      printf("%d: %s\n",i,liste[i]);
   }
}

void main(void)
{
   char langeListe[20][50];                /* zwanzig zeilen mit 50 zeichen */
   ...                                     /* liste wird gefüllt            */
   printListe(langeListe, 20);
}
```

4.2.1.6 Zeichenketten − Strings

In C gibt es keinen eigenen Datentyp für Zeichenketten. Sie werden als Felder (Kapitel 4.2.1, "Felder − Array − Vektoren") von `char`−Werten behandelt. Die Deklaration

```
char zeichenkette[9];
```

definiert ein Feld mit dem Namen `zeichenkette` mit Speicherplatz für neun Zeichen. Sie können wie alle Felder mit `zeichenkette[0], zeichenkette[1]` bis `zeichenkette[8]` angesprochen werden.

Direkte Zuweisungen an Felder sind nicht zulässig, demnach sind auch Konstrukte wie `zeichenkette = "Hallo";` unzulässig. Hierfür müssen Hilfsfunktionen wie z.B. `sprintf()` oder `strcpy()` (siehe 9.1, "Funktionen zur Bearbeitung von Zeichenketten und Zeichen") genutzt werden.

Zu beachten ist, daß die letzte Feldkomponente für die binäre Null (\0) reserviert sein muß, wenn mit Funktionen zur Zeichenkettenbearbeitung (z.B. `strcpy()`) gearbeitet werden soll. Die obige Deklaration würde dann nur Platz für acht Zeichen bieten! Aus diesem Grund sollte immer folgendermaßen deklariert werden:

```
char zeichenkette[9+1];    /* 9 Zeichen + binäre Null \0 */
char zeichenkette2[10];    /* dito                       */
```

Nähere Informationen zum Umgang mit Zeichenketten befinden sich im Kapitel 9.1, "Funktionen zur Bearbeitung von Zeichenketten und Zeichen".

4.2.2 Strukturen

Im Gegensatz zum Feld werden in einer Struktur Daten verschiedenen Typs verbunden. Diese Verbindung nach logischen Gesichtspunkten wird auch *Datensatz* genannt.

Beispiele:

Spielkarte

Farbe	Karte	Punkte

Pferd

Alter	Geschlecht	Herkunft	Stall	Besitzer

Die Datentypen sind unterschiedlich: Alter = int, Geschlecht = char, etc.

Strukturen sind sehr mächtig und gleichzeitig einfach zu benutzen. Sie können einzeln, als Felder und auch verschachtelt genutzt werden. Mehr dazu in den folgenden Abschnitten.

4.2.2.1 Deklaration von Strukturen

Im Vereinbarungsteil kennzeichnet das Schlüsselwort `struct` den Beginn einer Strukturdefinition. Auf das Schlüsselwort folgt ein Name für die Struktur. Dieser Name bezeichnet **nicht** eine Variable! Es handelt sich vielmehr um einen Namen für eine Art Vorlage. Anhand dieser Vorlage können im späteren Programmverlauf einzelne Variablen oder Felder dieser Struktur erzeugt werden. Die Deklaration der Komponenten wird in geschweifte Klammern (`{ }`) eingefaßt.

Beispiel:

```
struct student {                            /* Name der Vorlage   */
       char         name[40+1];             /* Nachname           */
       char         vorname[40+1];          /* Vorname            */
       int          post_leit_zahl;         /* Postleitzahl       */
       char         wohnort[40+1];          /* Wohnort            */
       char         strasse_nr[40+1];       /* Strasse,Hausnummer */
       char         geb_dat[8+1];           /* Geb_Datum ttmmjjjj */
       int          mat_nr;                 /* Matrikelnr:        */
       float        noten[10];              /* Prüfungsnoten      */
};
```

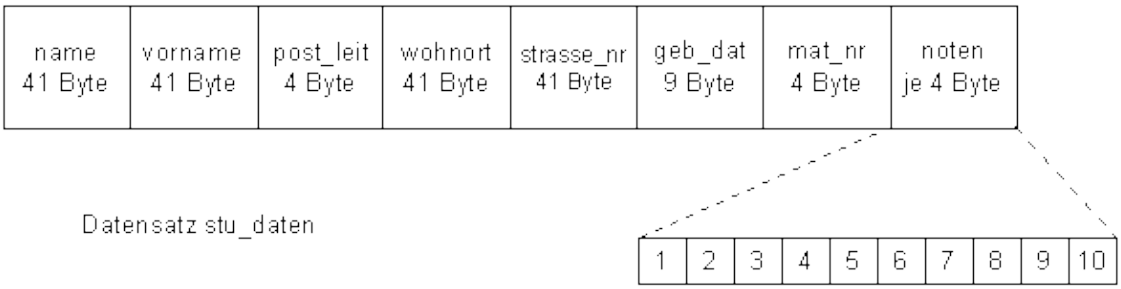
In dem obigen Beispiel wird ein Datensatz *student* definiert, allerdings noch kein Speicherplatz reserviert. Dem Compiler wird hierdurch nur der Name *student* und der Aufbau der Struktur bekannt gemacht. Es wurde also eine Vorlage mit dem Namen *student* definiert, anhand der Variablen erzeugt werden können, die dann diesen Aufbau haben.

Wird eine Variable dieser Struktur benötigt, so kann sie mit

```
struct student stu_daten;
```

definiert werden. Es wurde der Datensatz *stu_daten* unter Verwendung der Datenstruktur *student* mit einer Größe von 221 Byte vereinbart. Die acht Elemente des Datensatzes *stu_daten* sind somit folgende:

Abbildung 8. Datensatz stu_daten



Um die beiden Schritte Strukturdefinition und Datensatzvereinbarung zusammenzufassen, kann folgende Form gewählt werden:

```
struct student {
       char         name[40+1];              /* Nachname           */
       char         vorname[40+1];           /* Vorname            */
       int          post_leit_zahl;          /* Postleitzahl       */
       char         wohnort[40+1];           /* Wohnort            */
       char         strasse_nr[40+1];        /* Strasse,Hausnummer */
       char         geb_dat[8+1];            /* Geb_Datum ttmmjjjj */
       int          mat_nr;                  /* Matrikelnr:        */
       float        noten[10];               /* Prüfungsnoten      */
} stu_daten;
```

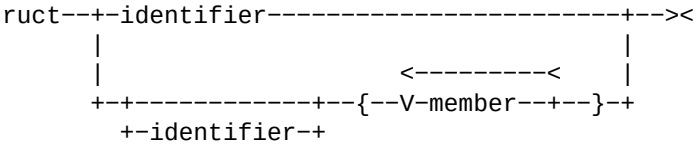
Sollen mehrere Datensätze (Variablen) mit der gleichen Struktur definiert werden, so können die Namen dieser Datensätze durch Kommata getrennt hintereinander angegeben werden.

```
struct student {
       char         name[40+1];              /* Nachname           */
       char         vorname[40+1];           /* Vorname            */
       int          post_leit_zahl;          /* Postleitzahl       */
       char         wohnort[40+1];           /* Wohnort            */
       char         strasse_nr[40+1];        /* Strasse,Hausnummer */
       char         geb_dat[8+1];            /* Geb_Datum ttmmjjjj */
       int          mat_nr;                  /* Matrikelnr:        */
       float        noten[10];               /* Prüfungsnoten      */
} stu_daten1, stu_daten2,
  stu_daten3;
```

Strukturen mit gleichem Aufbau, aber unterschiedlichem Namen sind in C kompatibel, in C++ hingegen nicht.

Syntaxdiagramm:

struct expression



struct member

```
                    <-,-----------<
>>--type specifier--V-declarator--+-->< 
```

4.2.2.2 Zugriff auf Strukturkomponenten

Der Zugriff auf einzelne Komponenten einer Struktur in C ist an die Vorgehensweise der vollqualifizierten Ansprache in Programmiersprachen wie z.B. PL/1 oder Cobol angelehnt:

```
strukturvariable.strukturkomponente
```

Beispiel:

```
strcpy(stu_daten.name, "Meyer");
stu_daten.noten[0] = 2;
```

Dem Element `stu_daten.name` wird die Zeichenkette `Meyer` und dem Element `stu_daten.noten[0]` wird die Zahl 2 zugewiesen.

Wenn über einen Zeiger (siehe 4.2.4, "Zeiger − Pointer") auf Strukturkomponenten zugegriffen werden sollen, muß folgende Syntax verwendet werden:

```
zeiger_auf_struktur->strukturkomponente

    oder

(*zeiger_auf_struktur).strukturkomponente
```

4.2.2.3 Felder von Strukturen

Soll ein Feld von Strukturen definiert werden, so ist dem Namen der Strukturvariablen die Felddefinition anzufügen:

```
struct student {
       char         name[40+1];              /* Nachname           */
       char         vorname[40+1];           /* Vorname            */
       int          post_leit_zahl;          /* Postleitzahl       */
       char         wohnort[40+1];           /* Wohnort            */
       char         strasse_nr[40+1];        /* Strasse,Hausnummer */
       char         geb_dat[8+1];            /* Geb_Datum ttmmjjjj */
       int          mat_nr;                  /* Matrikelnr:        */
       float        noten[10];               /* Prüfungsnoten      */
} stu_daten[10];
```

Hier wird ein Feld mit dem Namen `stu_daten` deklariert. Das Feld umfasst 10 Elemente, von `stu_daten[0]` bis `stu_daten[9]`. Der Aufbau der 10 Elemente entspricht den Vorgaben der Vorlage `student`.

Beim Zugriff auf einzelne Strukturelemente eines der Feldelemente muss im Gegensatz zum Zugriff auf einzelne Strukturelemente einer einzelnen Strukturvariablen die Elementnummer angegeben werden.

```
stu_daten[2].mat_nr = 158473;
// Das Strukturelement mat_nr im dritten Feldelement (stu_daten[2]) wird geändert.

stu_daten[1].noten[4] = 1.7;
// Das Strukturelement noten im zweiten Feldelement (stu_daten[1]) wird geändert.
// Da es sich bei noten wiederum um ein Feld handelt, muss auch hier das
// Feldelement angegeben werden. In diesem Beispiel das fünfte (noten[4]).
```

4.2.2.4 Verschachtelung von Strukturen

Es ist möglich, Strukturen als Bestandteile anderer Strukturen zu verwenden. Wichtig ist hierbei, dass die einzubindende Struktur als erstes deklariert wird. Im folgenden Beispiel wird in einer Struktur `person_` eine Strukturvariable der Struktur `gebu_` als Element deklariert. Beim Zugriff auf ein Element ist zu beachten, dass von aussen nach innen angesprochen wird, d.h. zuerst die Struktur, die die andere Struktur enthält, danach der Name der eingebundenen Struktur und zuletzt der Name des Elementes der eingebundenen Struktur.

```
/*
    Verschachtelung von Strukturen
*/

#include <stdio.h>


struct gebu_
{
  int tag;
  int monat;
  int jahr;
};

struct person_
{
  char   nachname[40];
  char   vorname[40];

  struct gebu_ geburtsdatum;   /* hier wird die andere Struktur eingebunden */

};

int main(int argc, char *argv[])
{
  struct person_ person;

  person.geburtsdatum.tag=12;
  person.geburtsdatum.monat=07;
  person.geburtsdatum.jahr=1981;

  printf("Geburtstag %d.%d.%d\n",

  person.geburtsdatum.tag,   /* Zugriff von aussen nach innen */
  person.geburtsdatum.monat,
  person.geburtsdatum.jahr);

  return 1;
} /* Ende main() */
```

4.2.2.5 Zeigerzugriff bei verschachtelten Strukturen

Beim Zeigerzugriff auf Strukturen wird der Pfeil−Operator (−>) benutzt. Dieser ist allerdings nur einmal erforderlich, Beispiel:

```
person->geburtsdatum.tag
```

4.2.2.6 Initialisierung von Strukturen

Eine Struktur kann direkt bei der Deklaration der Strukturvariablen mit Werten initialisiert werden. Beispiel:

(1)

```
struct datum {
       int  tag;
       char monat[10+1];
       int  jahr;
};

struct datum jahr_tag = {1,"Januar",1994};
```

(2)

```
struct datum {
       int  tag;
       char monat[10+1];
       int  jahr;
} jahr_tag = {1,"Januar",1994};
```

Hinweis
Nur als extern oder static (siehe Abschnitt 4.6, "Speicherklassen") definierte Strukturen dürfen wie in Beispiel 2 direkt im Anschluß an die Deklaration der Struktur initialisiert werden.

4.2.2.7 Strukturen mit Typendefinitionen − typedef

Wem die zusammengefaßte Schreibweise zu unübersichtlich ist, kann der Struktur auch einen eigenen Typ zuweisen:

```
typedef

struct {
       char         name[40+1];              /* Nachname           */
       char         vorname[40+1];           /* Vorname            */
       int          post_leit_zahl;          /* Postleitzahl       */
       char         wohnort[40+1];           /* Wohnort            */
       char         strasse_nr[40+1];        /* Strasse,Hausnummer */
       char         geb_dat[8+1];            /* Geb_Datum ttmmjjjj */
       int          mat_nr;                  /* Matrikelnr:        */
```

```
        float        noten[10];              /* Prüfungsnoten      */
} STUDENT;
```

Mit dieser Anweisung wird der Typ *Student* definiert und dem Compiler der Name und der Aufbau bekannt gemacht. Variablen können jetzt durch

```
STUDENT stu_daten;
```

vereinbart werden. Es ist somit ein neuer Datentyp angelegt worden. Um deutlich zu machen, daß es sich um einen neu definierten Typ handelt, sollte der Name (z.B. *STUDENT*) groß geschrieben werden.

4.2.3 Unions

Unions bestehen wie Strukturen aus mehreren Komponenten unterschiedlichen Datentyps. Für eine Struktur wird Speicherplatz zur Verfügung gestellt, der in seiner Größe der Summe der Größe der Strukturkomponenten entspricht. Für eine Union wird nur die Speicherkapazität reserviert, die notwendig ist, um die speicheraufwendigste der Union−Komponenten speichern zu können. Der Sinn besteht darin, Speicherplatz zur Verfügung zu stellen, der zu verschiedenen Zeiten Werte verschiedenen Datentyps aufnehmen kann.

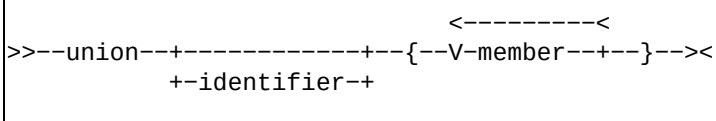
Der Speicherplatz jeder Union−Komponente beginnt an der gleichen Speicheradresse. Das bedeutet, daß immer nur der Wert einer Union−Komponente gespeichert werden kann, der vorherige Wert wird überschrieben. Syntaktisch unterscheiden sich die Unions nur durch das Schlüsselwort `union` von den Strukturen, die Variablendeklaration erfolgt nach dem Muster der Strukturvariablendefinition und auch der Zugriff erfolgt wie bei den Strukturen über den Punktoperator (vollqualifizierte Ansprache). Das folgende Beispiel zeigt, wie derselbe Speicherplatz einmal einen Integerwert und einmal einen Floatwert zugewiesen bekommt:

```
/*****************************************************************/
/* Programm: Union_1.c                                           */
/* Datum   : September 1994                                      */
/* Autor   : Sascha Kliche, Daniel Wolkenhauer                   */
/* Beschreibung:                                                 */
/* Die Wirkungsweise und die Verwendung von UNIONS wird an der   */
/* Zuweisung eines float- und eines integer-Wertes auf denselben */
/* Speicherplatz demonstriert.                                   */
/*****************************************************************/

#include <stdio.h>

/*
 * Deklaration der Komponenten der Union
 */
union my_union
{
   float float_num;
   int   int_num;
};

void main(void)
{
   /*
    * Deklaration einer Union-Variablen mit der Struktur my_union
    */
   union my_union union_komp;

   /*
```

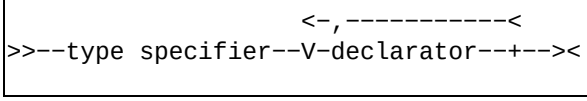
```
        * Zuweisung eines Integerwertes und Ausgabe
        */
       union_komp.int_num = 23;
       printf("Der aktuelle Wert der Union-Variablen ist: %d \n",
              union_komp.int_num);

       /*
        * Zuweisung eines Floatwertes und Ausgabe
        */
       union_komp.float_num = 19.87;
       printf("Der aktuelle Wert der Union-Variablen ist: %f \n",
              union_komp.float_num);
   }
```

Syntaxdiagramm:

union expression



union member



4.2.4 Zeiger − Pointer

Zeiger haben in C eine besondere Stellung, sie werden intensiver genutzt als in anderen Programmiersprachen. Das besondere an ihnen ist, daß sie als Inhalt bzw. Wert die Speicheradresse einer anderen Variablen [7] eines bestimmten Typs aufnehmen. Die Variablen eines Programmes stehen im Datenbereich ihres Prozesses. Der Prozeß sieht die Adressen nicht absolut zum Anfang des Hauptspeichers, sondern relativ zur Anfangsadresse des Datenbereichs des Prozesses. Für einen Prozeß ist demnach die Adresse einer Variablen die Nummer des ersten Bytes der Variablen relativ zum Beginn des Datenbereiches. Ein Pointer hat als Wert diese Adresse einer Variablen. Pointer werden vereinbart wie Variablen und erhalten wie diese Speicherplatz reserviert.

Abbildung 9. Zeiger

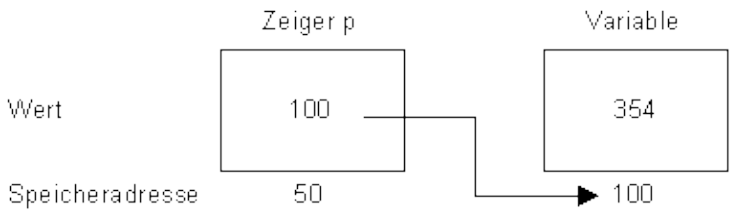


Tabelle 8. Zeiger im Speicher

Wert an der Speicheradresse	...		100		...		354		...
Speicheradresse	...	46	50	54	...	96	100	104	...

Aus diesem Beispiel ergibt sich:

- p ist ein Zeiger.
- Der Wert von p ist 100, dieser Wert wird als Adresse interpretiert.
- Der Zeiger p zeigt auf die Speicheradresse 100, an der der Wert 354 gespeichert ist. :* Der Zeiger p zeigt auf den Wert 354, der an der Adresse 100

Zeiger können einerseits sehr hilfreich sein, stellen andererseits ein hohes Risiko dar, da durch Fehler bei der Benutzung von Zeigern auf ungültige Speicheradressen zugegriffen werden kann. Diese Zugriffe führen unter Betriebssystemen mit geschützten Adreßräumen sofort zum Programmabsturz. Das einzig sichere ist ein Zeiger auf NULL, hier wird garantiert, daß der Zeiger definitiv auf keine Speicherstelle zeigt.

Daraus folgt: Zeiger immer bei der Deklaration auf NULL initialisieren (s.u.).

4.2.4.1 Vereinbarung von Zeigern

Zeiger müssen wie Variablen im Vereinbarungsteil definiert werden. Um deutlich zu machen, daß es sich um einen Zeiger handelt, sollte der Bezeichner mit einem p für Pointer oder z für Zeiger beginnen. Ein Zeiger wird durch das Zeichen `*` vor dem Bezeichner von Variablen unterschieden (Pflicht):

```
int   *zi = NULL;    /* Zeiger auf int    */
float *zf = NULL;    /* Zeiger auf float  */
```

Auch Zeigern wird ein Datentyp zugewiesen. Dieser interpretiert den Datentyp der Variablen, auf die der Zeiger zeigt, und nicht den eigenen Datentyp des Zeigers. Alle Adressen sind vom Typ Integer.

4.2.4.2 Anwendung von Zeigern

Um die Adresse einer Variablen zu ermitteln und sie dem Zeiger zu übergeben, verwendet C den unären Operator `&`. Nach der obigen Vereinbarung ergibt sich nun folgende mögliche Zuweisung:

```
int   n, *pi;
float x, *pf;

   pi = n
   pf = x
```

Nach dieser Zuweisung enthält `pi` die Adresse von `n` und `pf` die Adresse von `x`. Den Zugriff auf den Inhalt einer Adresse nennt man Dereferenzierung. Mit Hilfe der Dereferenzierung ist es möglich, nur unter Nutzung der Adresse eines anderen Objektes, dessen Wert zu ändern.

Beispiel:

```
/*****************************************************************/
/* Programm: zeiger.c                                            */
/* Autor   : Sascha Kliche, Daniel Wolkenhauer                   */
/* Datum   : September 1994                                      */
```

```
/* Beschreibung:                                                    */
/* Dieses Programm verdeutlich die Dereferenzierung: den Zugriff    */
/* auf den Inhalt einer Speicheradresse.                            */
/********************************************************************/
#include <stdio.h>
#include <conio.h>

void main(void)
{
   int  objekt = 0;          /* Int-Variable mit 0 initialisieren      */
   int *zeiger;              /* Zieger auf Int deklarieren             */

   zeiger  =      /* (1) Speicheradresse (objekt) an zeiger */
   objekt  = 4;           /* normale Zuweisung mit dem Wert 4       */

   *zeiger = 5;           /* (2) Speicherplatz von objekt = 5       */
}
```

In Zeile (1) wird der Zeigervariablen `zeiger` die Speicheradresse eines Objekts (hier: der Integervariablen `objekt`) übergeben. Dem Objekt wird dazu der unäre Adreßoperator ()vorangestellt. Mit seiner Hilfe wird nicht der Wert der Variablen `objekt` (zu dieser Zeit = 0) zugewiesen, sondern, die Adresse, an der der Wert im Speicher abgelegt ist.

Die eigentliche Dereferenzierung findet in Zeile (2) statt. Nur mit Hilfe der Adresse wird auf einen Speicherinhalt 'zugegriffen'. In diesem Fall wird der Inhalt (4) mit einem neuen Wert (5) überschrieben.

Folgende Zuweisungen sind beispielsweise erlaubt:

```
int zahl=3; zahl2=0, *ptr_zahl=NULL, *ptr_zahl2=NULL;

ptr_zahl=         /* adresse von zahl zuweisen   */
ptr_zahl2=ptr_zahl;     /* adresse von zahl übernehmen */
zahl=(int)ptr_zahl;     /* adresse von zahl an zahl übergeben */

*ptr_zahl=5;            /* zahl den wert 5 zuweisen, entspricht zahl=5 */
zaehl2=*ptr_zahl;       /* wert von zahl an zahl2 übergeben            */

ptr_zahl2=(int *)*ptr_zahl; /* inhalt von zahl als adresse an
                               ptr_zahl2 übergeben */
```

Ungültig sind beispielsweise folgende Zuweisungen:

```
int zahl=3; zahl2=0, *ptr_zahl=NULL, *ptr_zahl2=NULL;

zahl=ptr_zahl;          /* eine adresse kann nicht einer int
                           variablen übergeben werden         */
*ptr_zahl=         /* dito                                 */

ptr_zahl=*zahl;         /* zahl enthält keine adresse         */

ptr_zahl2=*ptr_zahl;    /* einem zeiger muß eine adresse
                           übergeben werden                   */
```

Beispielprogramm:

```
void main(void)
{
  int zahl=3, *ptr_zahl=NULL;
```

```
  ptr_zahl=

  printf("\nAdresse von ptr_zahl = %d",
  printf("\nInhalt von ptr_zahl = %d",ptr_zahl);
  printf("\nAdresse von zahl = %d",
  printf("\nInhalt von zahl = %d",zahl);
  printf("\nWert von *ptr_zahl = %d",*ptr_zahl);
}
/*
   Ausgabe: (Adressenangaben ändern sich)

   Adresse von ptr_zahl = 166004
   Inhalt von ptr_zahl = 166008
   Adresse von zahl = 166008
   Inhalt von zahl = 3
   Wert von *ptr_zahl = 3
*/
```

4.2.4.3 Zeiger bei Funktionsaufrufen

Beim Aufruf von Funktionen werden die Parameter entweder per call−by−value oder call−by−reference übergeben. Letzterer Fall bewerkstelligt dies mit der Hilfe von Zeigern. Auf die Unterscheidung wird auch im Zusammenhang mit Funktionen (7.3, "Call by value und call by reference") eingegangen.

```
/*
   Beispiel "Call−by−reference"

   Die Funktion tausche() tauscht den Inhalt zweier Integer−Variablen.
   Dieser Tausch erfolgt mittels Zeigerzugriff.
*/


#include <stdio.h>


/*−−− Funktionsprototyp −−−*/
void tausche(int *, int *);


/*−−−Hauptfunktion−−−*/
int main(int argc, char *argv[])
{
   int a=3, b=7;

   printf("Vertauschung zweier Zahlen\n\nBitte geben Sie die erste Zahl ein : ");
   fflush(stdout);
   fflush(stdin);
   scanf("%d",
   printf("Bitte geben Sie die zweite Zahl ein: ");
   fflush(stdout);
   fflush(stdin);
   scanf("%d",

   /* Ausgabe: a=3 und b=7 */
   printf("\nTausche a=%d und b=%d −> ",a,b);

   /*
      tausche() benötigt Zeiger auf die Variablen, deren Werte getauscht werden sollen.
      Zu diesem Zweck wird der Adressoperator ()verwendet.
      tausche() erhält somit die Adressen der Variablen a und b im Hauptspeicher als Parame
```

```
   */
   tausche(

   /* Ausgabe: a=7 und b=3 */
   printf("a=%d und b=%d\n",a,b);


   return 1;
} /* Ende main() */


/*---Implementation der Funktion tausche()---*/
void tausche(int *zahl1, int *zahl2)
{
   int zwischen=0;

   /* Inhalt der ersten Variable sichern */
   /* *zahl1 bedeutet: hole das, was im Hauptspeicher an der Stelle steht, deren Adresse de
   /*            der zeiger zahl1 enthält die Hauptspeicheradresse der Variablen a und zeig
   zwischen = *zahl1;

   /* Inhalt der ersten Variable mit Inhalt der zweiten Variable überschreiben */
   *zahl1 = *zahl2;

   /* Inhalt der zweiten Variable mit gesichertem Inhalt der ersten Variable überschreiben
   *zahl2 = zwischen;


   return;
} /* Ende tausche() */
```

4.2.4.4 Zeiger und Felder

Zeiger und Felder sind eng miteinander verbunden. Die Deklaration eines Feldes bewirkt, daß im Stack ein Speicherbereich der angegebenen Größe reserviert wird und der Bezeichner gleichzeitig einen Zeiger auf diesen Speicherbereich darstellt.

```
char zeichen[10];
char ptr;
```

`zeichen` enthält die Adresse von `zeichen[0]`. entspricht also `ptr`. `zeichen` entspricht ebenfalls `ptr`. Daraus ergibt sich, daß `zeichen` entspricht. Die Adresse einer Zeichenkette kann einem Zeiger auf Zeichen also nur als Adresse eines Elementes der Zeichenkette übergeben werden (z.B. `ptr=zeichen;` oder `ptr=` oder `ptr=`). Hier ist auf die korrekte Benutzung des Adreßoperators ()zu achten.

Äquivalente Ausdrücke:

```
ptr
ptr           zeichen
*ptr          zeichen[0]
*(ptr+1)      zeichen[1]
*(ptr+n)      zeichen[n]

zeichen
```

Eine nützliche Eigenschaft von Zeigern im Zusammenhang mit Feldern ist die Möglichkeit, mit Zeigern zu rechnen. Dabei ist es möglich, mittels Inkrement− und Dekrementoperatoren immer das nächste Feldelement

zu erreichen, da diese beim Erhöhen des Zeigerwertes (Veränderung der Hauptspeicheradresse, auf die der Zeiger zeigt) die Grösse das Datentyps berücksichtigen. Erhöht man z.B. mit dem Inkrementoperator einen Zeiger auf einen Integer und einen Zeiger auf einen Double, so verändert sich der Integer um 4 (32 Bit) und der Double um 8 (64 Bit).
Beispiel für Inkrementoperator bei Zeigern mit Speicheradressen

```
pIntZahl      == 4000   /* Zeiger pIntZahl zeigt auf Speicherstelle 4000   */
pIntZahl++    -> 4004   /* Nach inkrement (int = 4 Byte) zeigt er auf 4004 */

pDoubleZahl   == 6000   /* Zeiger pDoubleZahl zeigt auf Speicherstelle 6000   */
pDoubleZahl++ -> 6008   /* Nach inkrement (double = 8 Byte) zeigt er auf 6008 */
```

Das folgende Beispiel demonstriert, wie mittels Zeigerarithmetik ein Zeichenfeld mit der Deklaration `char **argv` bzw. `char *argv[]` analysiert werden kann. Das Beispiel nutzt Zeiger intensiv und sollte nur dann angewendet werden, wenn man es wirklich versteht, da es auch einige Fallstricke beinhaltet.

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

void main(int argc, char *argv[])
{
  int anzparms=0, t_parameter=0;
  char name_in[255+1], name_out[255+1];
  char usage[35]="Hier steht ein Hilfekommentar.\0";

  argv++;                           /* auf zweiten parameter wechseln    */
                                    /* erster Parameter ist Programmname */
  argc--;                           /* anzahl parameter dekrementieren   */
  while(argc>0)
  {
    if (**argv=='-' || **argv=='/')  /* erstes Zeichen gleich - oder / ?  */
    {
      (*argv)++;                    /* ein zeichen weiter                */
      if(**argv!='\0')              /* zugriff auf zeichen nach - oder / */
        switch(**argv)
        {
          case 'h':
          {
            printf("%s",usage);
            exit(1);
            break;
          }
          case 't': t_parameter=1;
                    break;
          default : printf("Unrecognized parameter '-%c' ignored.",**argv);
                    break;
        }                           /* ende switch                       */
    }                               /* ende - oder / im parameter        */
    else
    {                               /* dateinamen zuweisen               */
       anzparms++;
       if(anzparms<3)
       {
         if(anzparms==1) strcpy(name_in,*argv);
         if(anzparms==2) strcpy(name_out,*argv);
       }
       else
         printf("Unrecognized parameter '%s' ignored.\n",*argv);
    }                               /* ende dateinamen zuweisen          */
    argv++;
```

```
      argc--;
    }                                   /* while argc>0                    */
  }                                     /* ende main                       */
```

Abbildung 10. Zeiger auf Integer

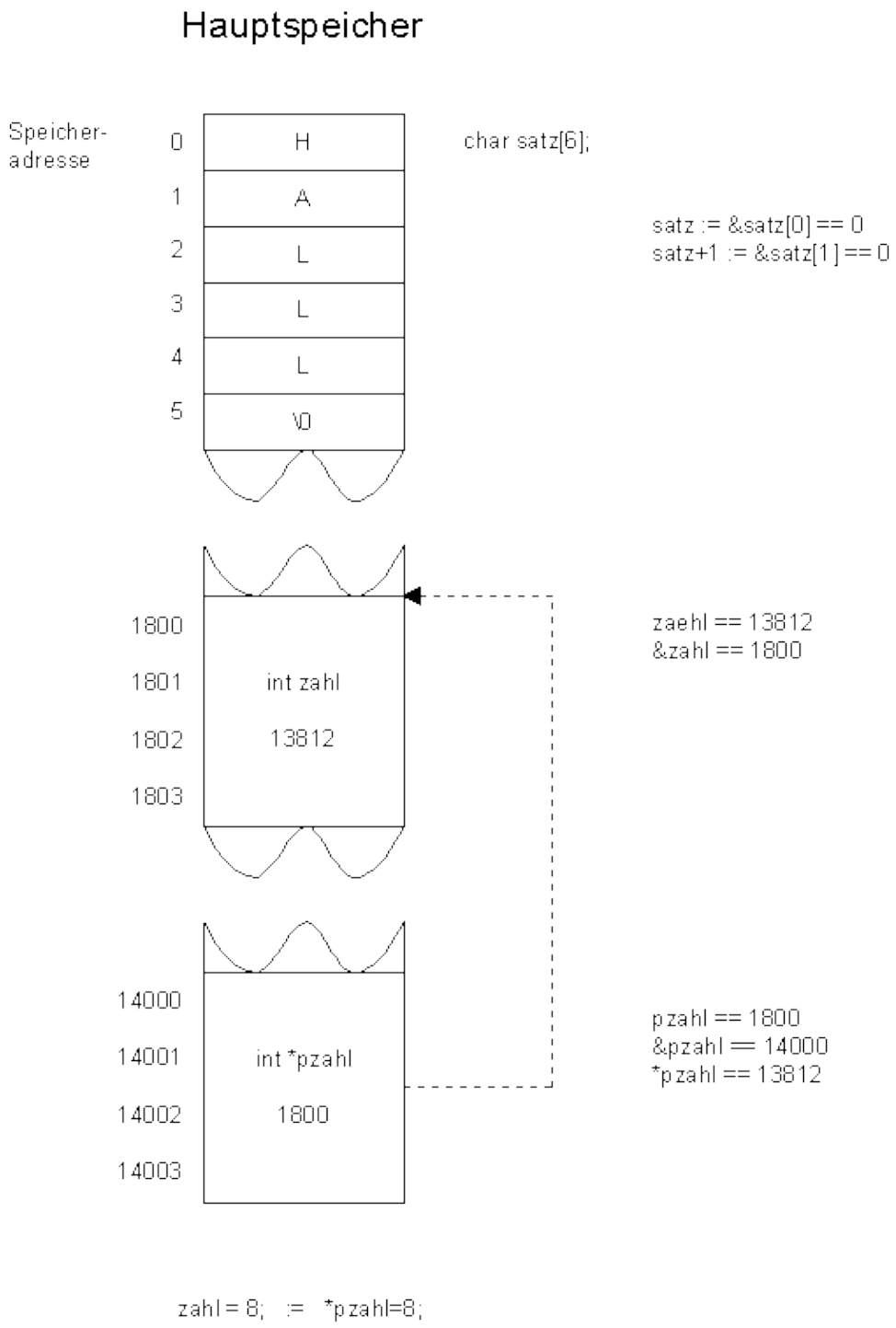
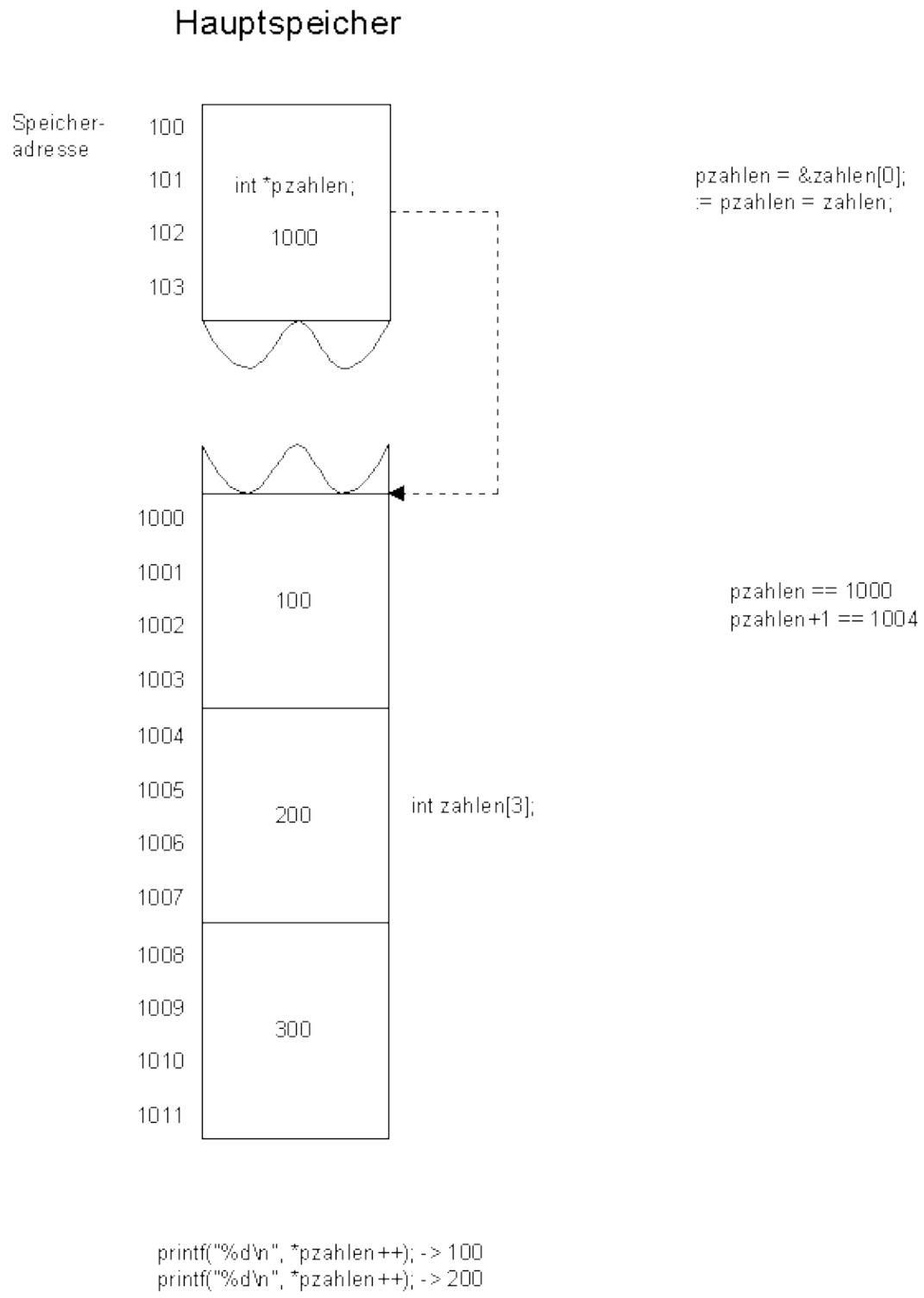


Abbildung 11. Zeiger auf Integer−Feld



4.3 Weitere Datentypen

4.3.1 void

Der Datentyp `void` hat keinen Wertebereich und kommt immer dann zum Einsatz, wenn kein bestimmter

Datentyp erforderlich ist bzw. um zu kennzeichnen, daß Funktionen keine Parameter erwarten oder keinen Wert zurückgeben.

4.3.2 enum

Eine Enumeration (Aufzählung) repräsentiert eine bestimmte Menge an Werten, die der Programmierer definiert. Eine Enumeration ist in C immer vom Typ `int`. Die definierten Werte einer Enumeration werden nacheinander durchnumeriert und somit auf den Datentyp `int` umgelegt. Die Deklaration umfaßt das Schlüsselwort `enum` gefolgt von einem Bezeichner und einer Liste der Aufzählungen. Die Elemente der Aufzählung werden auch enumeration constants genannt. Die Liste der Aufzählungen wird in geschweifte Klammern eingefaßt und jede Aufzählung ist durch ein Komma abgetrennt.
Beispiel:

```
enum monate { Januar=1, Februar, Maerz, April, Mai, Juni, Juli,
              August, September, Oktober, November, Dezember };
```

Die Monatsbezeichnungen werden, bei eins beginnend, aufsteigend numeriert, Januar=1 bis Dezember=12. Normalerweise beginnt die Numerierung mit dem ersten Element bei Null. Diese Numerierung ermöglicht die Benutzung der Bezeichnungen anstatt der Nummern im Programmcode. Jedes Aufzählungselement darf in seinem Gültigkeitsbereich nur einmal definiert werden.

Die Numerierung kann bei jedem Element der Aufzählungsliste beeinflußt werden:

```
enum status { run, create, delete=5, suspend };
```

Der Datentyp `status` repräsentiert die folgenden Werte:

```
 Enumeration         Integer
 Constant            Representation

 run                 0
 create              1
 delete              5
 suspend             6
```

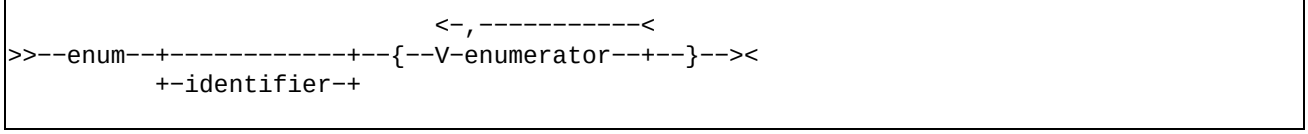
Enumeration Variablen werden wie folgt definiert:

```
enum monate { Januar=1, Februar, Maerz, April, Mai, Juni, Juli,
              August, September, Oktober, November, Dezember } monat1, monat2;

enum monate monat3 = April;
```

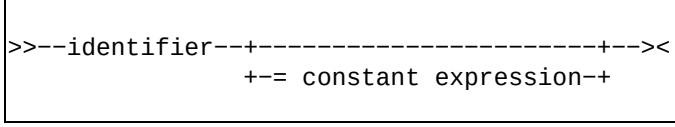
In diesem Beispiel werden die Variablen `monat1, monat2` und `monat3` definiert, wobei der Variablen `monat3` der Wert `April` (repräsentiert die Nummer 4) zugewiesen wird.

Syntaxdiagramm:

enum



enumerator



4.4 Datentypumwandlung

Bei der Datentypumwandlung wird danach unterschieden, ob der Programmierer regelnd eingreift (explizit) oder ob das System von sich aus (implizit) ohne Dazutun des Programmierers eine Umwandlung durchführt.

4.4.1 Implizite Datentypumwandlung

Der Compiler wandelt verschiedene Datentypen grundsätzlich in festgelegte andere Datentypen um, um die Möglichkeiten der jeweils verwendeten Hardware maximal ausnutzen zu können. Welche Datentypen in welche umgewandelt werden, ist der Dokumentation des jeweiligen Compilers zu entnehmen.

Weiterhin gilt, daß bei der Verwendung von verschiedenen Datentypen bei einer Zuweisung das Ergebnis der Berechnung rechts vom Gleichheitszeichen immer in den Datentyp des Operanden links vom Gleichheitszeichen umgewandelt wird. Dies gilt auch, wenn links mit `int`−Werten und rechts mit rationalen Zahlen gerechnet wird! Wichtig ist jedoch zu Bedenken, dass erst das Ergebnis umgewandelt wird.

Abschließend ist zu beachten, daß beim Zusammentreffen verschiedener Datentypen in einer Berechnung immer mit dem speicheraufwendigeren gerechnet wird.

4.4.2 Explizite Datentypumwandlung

Der Programmierer kann einen Operanden allerdings auch gezielt verändern. Dazu kann ein sogenannter *Cast* (oder *typecast*) benutzt werden. Ein Cast ist eine Datentypbezeichnung, die dem zu verändernden Operanden vorangestellt wird.
Beispiel:

```
int    ganz_1=3, ganz_2=4;
double dopp=0.0;

dopp = (float)ganz_1 * (long)ganz_2;
```

Die Variablen `ganz_1` und `ganz_2` werden als Integer definiert. Durch den Cast−Operator wird `ganz_1` in eine rationale Zahl und `ganz_2` in den Typ `long` explizit umgewandelt. Allerdings verändert der Compiler `ganz_2` eigenständig in den speicheraufwendigeren Typ (`float`). `float` wird allerdings gleich in `double` konvertiert. Das Ergebnis der Berechnung `ganz_1 * ganz_2` wäre ohnehin implizit in `double` umgewandelt worden, da es sich bei dem links vom Gleichheitszeichen stehenden Operanden um eine Gleitkommavariable mit doppelter Genauigkeit handelt.

Des weiteren können Funktionen zur Typkonvertierung benutzt werden. Zwei Beispiele sind die Funktionen `atoi()` zur Umwandlung von Zeichenketten in Integer und `atof()` zur Umwandlung von Zeichenketten in Float.

Beispiel für die Benutzung von `atof()`:

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

int main(int argc, char *argv[])
{
  char eingabe[20];
  double zahl=0.0;

  printf("Bitte eine Kommazahl eingeben: ");
  fflush(stdout);
  fflush(stdin);
  /* gets(eingabe); */                                          /* gets ist unsic
  fread(eingabe,sizeof(char),20,stdin);                         /* fread ist sich
  zahl=atof(eingabe);
  printf("%s (%d) ergibt %lf!\n",eingabe,strlen(eingabe),zahl);

  fflush(stdin);  /* leert den Puffer nach einer fread() operation nicht! */
  gets(eingabe);
  printf("%s\n",eingabe);

  return 1;
} /* Ende main() */

/*
   atof konvertiert Komma-Angaben (z.B. 123,45) nicht!
*/
```

Folgende Funktionen sollten gemäß ANSI C Standard mindestens zur Verfügung stehen.

Tabelle 9. Funktion zur Typkonvertierung − ANSI C

atof	Konvertiert eine Zeichenkette in einen Fließkommawert (double).
atoi	Konvertiert eine Zeichenkette in einen Integer.
atol	Konvertiert eine Zeichenkette in einen Long Integer.
strtod	Konvertiert eine Zeichenkette in einen Fließkommawert (double).
strtol	Konvertiert eine Zeichenkette in einen Long Integer.
strtoul	Konvertiert eine Zeichenkette in einen Unsigned Long Integer.

Als Erweiterung des Sprachstandards stehen u.U. folgende Funktionen (abhängig vom jeweiligen Compiler) zur Verfügung.

Tabelle 10. Funktion zur Typkonvertierung − nicht ANSI C

_atold	Konvertiert eine Zeichenkette in einen Fließkommawert (long double).
atoll	Konvertiert eine Zeichenkette in einen Long Long Integer.
ecvt	Konvertiert einen Fließkommawert in eine Zeichenkette.
fcvt	Konvertiert einen Fließkommawert in eine Zeichenkette. Rundung erfolgt gemäß FORTRAN F Format.
gcvt	Konvertiert einen Fließkommawert in eine Zeichenkette. Rundung erfolgt gemäß FORTRAN F oder FORTRAN E Format.
_itoa	Konvertiert einen Integer in eine Zeichenkette.
_ltoa	Konvertiert einen Long Integer in eine Zeichenkette.
strtold	Konvertiert eine Zeichenkette in einen Fließkommawert (long double).
strtoll	Konvertiert eine Zeichenkette in einen Long Long Integer.

strtoull	Konvertiert eine Zeichenkette in einen Unsigned Long Long Integer.
_ultoa	Konvertiert einen Unsigned Long Integer in eine Zeichenkette.
_ulltoa	Konvertiert einen Unsigned Long Long Integer in eine Zeichenkette.

4.5 Eigene Typen − typedef

Es besteht in C die Möglichkeit, eigene Datentypen zu deklarieren (siehe auch Beispiel in 4.2.2.7, "Strukturen mit Typendefinitionen − typedef"). Dies dient vor allem der Übersichtlichkeit, da nicht wirklich neue Datentypen erzeugt werden, sondern lediglich alte Datentypen einen neuen Namen erhalten. Dazu wird das Schlüsselwort `typedef` verwendet. Der neu vereinbarte Name kann daraufhin als neuer Datentypbezeichner verwendet werden.

```
typedef float REAL;
typedef int INTEGER;
typedef int *PTR_TO_INT;
```

Dadurch werden die Typbezeichner REAL, INTEGER und PTR_TO_INT vereinbart. Sie können in nachfolgenden Vereinbarungen als Datentypbezeichner verwendet werden:

```
REAL x, y;              /* entspr.: float x, y;   */
INTEGER a, b,;          /* entspr.: int a, b      */
PTR_TO_INT p1, p2;      /* entspr.: int *p1, *p2  */
```

Die Großschreibung dient der Unterscheidung zwischen eigenen und C−Datentypbezeichnern und sollte beibehalten werden.

Mitunter können diese eigenen Typen aber auch enorm zur Unübersichtlichkeit beitragen. `typedef` sollte mit Augenmaß eingesetzt werden.

4.6 Speicherklassen

Die Speicherklassen definieren genaue Bedingungen für die Art und Weise der Speicherung der Daten eines Programmes. Durch diese Bedingungen werden

- der Gültigkeitsbereich (Programm, Modul, Lokal),
- die Lebensdauer (dauerhaft, Blockdurchlauf) und
- der Speicherort (Stack, Heap)

vorgegeben.

Ein Stack ist ein kleiner Speicherbereich für lokale Variablen, wird also pro Funktion angelegt. Auf diesem werden einfach Datentypen wie int und double, aber auch komplexe Datentypen wie statische Felder und Zeiger abgelegt. Da die Größe des Stacks beschränkt ist, sollten große Felder und Datenstrukturen nur mit Hilfe der dynamischen Speicherverwaltung (siehe 15.0, "Speicherverwaltung") angelegt werden.
Der Heap ist ein großer Speicherbereich für dynamisch allokierten Speicher, der notfalls auch wachsen kann. In diesem Fall wird zur Laufzeit des Programms weiterer Speicher vom Betriebssystem geordert. Auf dem Heap landen alle dynamisch erzeugten Speicherblöcke (siehe 15.0, "Speicherverwaltung").

4.6.1 Automatic

Gültigkeitsbereich:	Blocklokal
Lebensdauer:	Dauer des Blockdurchlaufs
Speicherort:	Stack (Stapel)

Datenelemente, die nicht explizit einer anderen Speicherklasse angehören und innerhalb eines Blockes, der durch geschweifte Klammern eingefaßt ist, vereinbart werden, werden vom Compiler automatisch der Speicherklasse `auto` zugewiesen. Diese Datenelemente sind nur in dem Programmteil (Block) verfügbar, in dem sie definiert werden. Erst bei Aufruf des Programmteils wird für sie Speicher auf dem sog. Stack reserviert, der beim Verlassen allerdings wieder freigegeben wird. Der Inhalt dieses Datenelements bleibt nicht bis zum nächsten Blockaufruf erhalten.

Hinweis: In der Regel kann der Begriff Block mit dem Begriff Funktion gleichgesetzt werden, da innerhalb einer Funktion normalerweise kein neuer Block definiert wird, was jedoch zulässig ist und bei einigen Problemstellungen von Vorteil ist. Ein Datenelement der Speicherklasse `auto` ist somit in der Regel für die Dauer eines Funktionsdurchlaufes und nur innerhalb dieser Funktion verfügbar.
Beispiel:

```
auto int a,b,c;    oder

     int a,b,c;
```

4.6.2 Static

Bei Datenelementen der Speicherklasse `static` unterscheidet man, wo diese definiert werden:

(1) Innerhalb einer Funktion

Gültigkeitsbereich:	Blocklokal
Lebensdauer:	gesamter Programmdurchlauf
Speicherort:	Heap

(2) Außerhalb einer Funktion

Gültigkeitsbereich:	Modul
Lebensdauer:	gesamter Programmdurchlauf
Speicherort:	Heap

`static` (=dauerhafte) Datenelemente können innerhalb oder außerhalb von Funktionen definiert werden. Ein weiterer Unterschied zu Datenelementen der Speicherklasse `auto` besteht darin, daß `static`−Variablen, die innerhalb einer Funktion vereinbart wurden, nicht aus dem Speicher gelöscht werden, wenn die Funktion beendet wird. Damit stehen die Werte bei einem erneuten Funktionsaufruf wieder zur Verfügung und werden auf dem sog. Heap abgelegt.

Ein außerhalb einer Funktion definiertes `static` Datenelement ist in dem ganzen Modul − nicht aber in anderen Modulen, wie `extern` Datenelemente − verfügbar.

`static` Datenelemente sollten direkt nach den Preprozessoranweisungen (z.B. `#include`) deklariert werden.

4.6.3 Extern

Gültigkeitsbereich:	Programm
Lebensdauer:	gesamter Programmdurchlauf
Speicherort:	Heap

Es besteht die Möglichkeit, globale Datenelemente in allen Modulen zu benutzen. Diese Datenelemente gelten dann für alle Module und sind von der Speicherklasse `extern`. Innerhalb von anderen Modulen kann auf extern definierte Elemente mit einem Verweis (siehe Beispiel unten) zugegriffen werden.

Auch Funktionen sind vom Typ `extern`.
Beispiel:

```
Datei fc1.c

    #include <stdio.h>

    static int x=99;      /* nur im aktuellen Modul gültig        */
    int a=1, b=2, c=3;    /* Externe Variablen für alle Module    */

    int addieren(void);   /* Funktionsprototyp                    */

    void main(void)
    {

       printf("a=%3d b=%3d c=%3d\n", a,b,c);          /* 1  2  3 */
       printf("a+b+c=%3d\n",addieren());              /*    7    */
       printf("a=%3d b=%3d c=%3d\n", a,b,c);          /* 1  3  3 */

    }  /* Ende von main */


Datei fc2.c

    int addieren(void)
    {
       extern int a, b, c;     /* Variablen aus anderem Modul          */
       static int g;           /* Wert der Var. bleibt erhalten        */
       int ergebnis;           /* Lokale Variable                      */

       g++;
       ergebnis=a+b+c;
       a=g;
       b+=g;
       return (a+b+c);
    }  /* Ende addieren */
```

Lokale Variablen haben in ihrem Gültigkeitsbereich Vorrang vor globalen Variablen.

4.6.4 Register

Für Datenelemente der Speicherklasse `register` gilt dasselbe wie für Datenelemente der Speicherklasse `auto`, allerdings werden diese, sofern die Hardware dies zuläßt, in den Prozessorregistern gespeichert. Der Zugriff auf die Prozessorregister ist um ein Vielfaches schneller, als der Zugriff auf den Hauptspeicher.

5.0 Auswahl

Bei der Auswahl sind zwei mögliche Varianten zu betrachten:

1.	if / else	einfache Auswahl oder Alternativauswahl
2.	switch	mehrfache Auswahl (Liste)

5.1 if/else

`if` bzw. `if/else` werden immer dann genutzt, wenn die Abarbeitung einer oder mehrerer Anweisungen von einer Bedingung abhängt. Z.B. kann die Abbuchung eines Betrages von einem Konto davon abhängig gemacht werden, ob das Konto gedeckt ist. `if` wird immer dann gewählt, wenn nur interessiert, ob die Bedingung erfüllt wird. `if/else` wird dann gewählt, wenn ebenfalls das Nichteintreten der Bedingung von Interesse ist. Im Falle der Kontoabbuchung sollte reagiert werden, wenn das Konto nicht gedeckt ist, dementsprechend wäre `if/else` notwendig. In Prosa könnte man z.B. sagen: Wenn das Konto gedeckt ist, darf abgebucht werden. Wenn dies nicht der Fall ist, soll eine Fehlermeldung ausgegeben werden. Die Umsetzung in Programmcode sieht folgendermaßen aus:

Abbildung 12. Bedingung

... als C Code formuliert:

```
if ( kontostand - betrag > limit )
{
   kontostand = kontostand - betrag;
}
else
{
   printf("Das Konto ist nicht gedeckt, Abbuchung nicht möglich!\n");
}
```

Die nach `if` in Klammern stehende Bedingung muß wahr sein, damit die darauffolgende Anweisung ausgeführt wird. Sollen mehrere Anweisungen ausgeführt werden, müssen diese in geschweifte Klammern ({}) eingefaßt werden.

5.1.1 Bedingte Anweisung

Eine Anweisung oder eine Gruppe von Anweisungen wird nur in dem Fall ausgeführt, wenn die Bedingung wahr (ungleich 0) ist.

1. Beispiel:

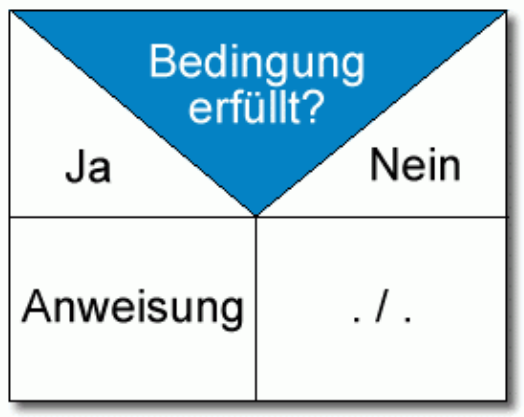
```
if ( kontostand-abbuchungsbetrag < kreditlinie )
{
   zugriff = 0;
   printf("Abbuchung nicht möglich!");
}
```

Wenn der Kontostand kleiner oder gleich Null ist, wird der Zugriff auf das Konto gesperrt und ein Hinweis ausgegeben.

2. Beispiel:

```
if ( !kontostand )
{
   zugriff = 0;
   printf("Exakt 0 DM auf dem Konto!");
}
```

Abbildung 13. Nassi Shneiderman Diagramm − if



5.1.2 Alternativ− oder Zweifachauswahl

Wenn die Bedingung nicht zutrifft, wird der Anweisungsblock nach `else` ausgeführt.

Beispiel für Alternativ− oder Zweifachauswahl:

```
if ( alter > 50 )
{
   printf("Bald gibt's Rente!");
}
else
{
   printf("Ein bischen dauert's noch.");
}
```

Bei zwei aufeinander folgenden `if`−Anweisungen mit nur einem `else` ist eigentlich nicht eindeutig, zu welchem `if` das `else` gehört:

```
if ( bedingung1 )
   if ( bedingung2 )
      anweisung1;
```

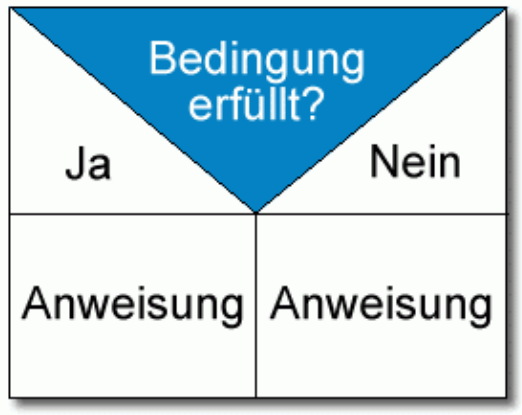
```
    else
       anweisung2;
```

In C gilt die Regel, dass ein `else` immer zu dem vorhergehenden `if` gehört, solange nicht dur Klammerung eine andere Zugehörigkeit erzwungen wird.

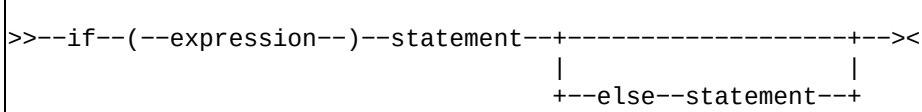
Anmerkung
Wie bei der for−Schleife wird auch hier nach der abschlierunden Klammer kein Semikolon, sondern nur eine geschweifte Klammer (Block−Zeichen) gesetzt.

Achtung
Bei Test auf Gleichheit `==` anstatt `=` benutzen!

Abbildung 14. Nassi Shneiderman Diagramm − if / else



Syntaxdiagramm:

if − statement


5.2 switch

ACHTUNG! Der folgende Teil ist für Programmieranfänger vollkommen unverständlich.

Mit Hilfe der `switch`−Anweisung kann eine Alternative unter einer Liste von verschiedenen Möglichkeiten gewählt werden. Diese Art der bedingten Auswahl wäre auch mit einer `if/else`−Kombination erreichbar, allerdings käme es zu einer Verschachtelung, die auch mit guten Kommentaren und guter Strukturierung nicht mehr lesbar wäre.

Dem Schlüsselwort `switch` folgt ein geklammerter Ausdruck, dessen Wert für die Auswahl zur Verfügung steht. Der Ausdruck in `switch ( ausdruck )` muss einen Integer ergeben (Integer−Variable, −Konstante oder −Ergebnis einer Berechnung). Keine Felder, also auch keine Zeichenketten!

Dem Ausdruck folgt der *switch body*, der in einfacher und in komplexer Form existiert.

Die einfache Form enthält einen oder mehrere `case`−Zweige und einen `default`−Zweig (optional) mit den Anweisungen. Die komplexe Form enthält zudem Variablendeklarationen und benötigt dafür geschweifte Blockklammern.

Syntax der einfachen Form:

```
switch ( Ausdruck )
{

   case konstanter Ausdruck : Anweisung;
                              [break;]
   case konstanter Ausdruck :
           Anweisung 1;
           Anweisung 2;
           ...
           Anweisung n;
           [break;]
   default : Anweisung;
             [break;]
}
```

Syntax der komplexen Form:

```
switch ( Ausdruck )
{

   case konstanter Ausdruck : Anweisung;
                              [break;]
   case konstanter Ausdruck :
        {
           Variablendeklarationen;

           Anweisung 1;
           Anweisung 2;
           ...
           Anweisung n;
           [break;]
        }
   default : Anweisung;
             [break;]
}
```

Der Ausdruck in `case ausdruck` muss ebenfalls einen Integer ergeben. Bereichsabfragen wie z.B. `case 10−15` oder `case 'a'−'z'` sind nicht möglich! Der `default` Zweig wird immer dann ausgeführt, wenn kein `case` zutrifft. Auch wenn man glaubt, alle Zustände mit `case` abzudecken, sollte man sicherheitshalber `default: break;` anhängen, da im Fall der Fälle das Verhalten des Programms nicht definiert ist!

Beispiel:

```
switch(zeichen)
{
  case 'a': printf("Kleinbuchstabe\n"); break;
  case 'b': printf("Kleinbuchstabe\n"); break;
  ...
  case 'z': printf("Kleinbuchstabe\n"); break;
  case 'A': printf("Grossbuchstabe\n"); break;
  case 'B': printf("Grossbuchstabe\n"); break;
  ...
```

```
    case 'Z': printf("Grossbuchstabe\n"); break;
    case '0': printf("Ziffer\n"); break;
    case '1': printf("Ziffer\n"); break;
    ...
    case '9': printf("Ziffer\n"); break;
    default : printf("Anderes Zeichen\n"); break;
  }
```

In diesem Beispiel wird eine Variable `zeichen` (definiert als `char zeichen;`) daraufhin überprüft, ob es sich um einen Großbuchsteben, Kleinbuchstaben oder eine Ziffer handelt. Die Zeilen mit ... kennzeichnen lediglich, daß im Beispiel einige Zeilen ausgelassen wurden. Es besteht keine Möglichkeit, Bereiche in der `case`−Bedingung anzugeben.

`break` wurde bereits in Abschnitt 6.4, "Die Anweisungen break und continue" beschrieben. Bei `switch` sorgt dieses Schlüsselwort dafür, daß der `switch`−Durchlauf abgebrochen wird. Wird dieses Schlüsselwort nicht verwendet und der Ausdruck einer der `case`−Zweige trifft zu, so durchläuft das Programm nicht nur diesen Zweig, sondern auch sämtliche nachfolgende, bis es auf ein `break` trifft. Dadurch können wir uns einigen Aufwand im obigen Beispiel sparen:

```
  switch(zeichen)
  {
    case 'a':
    case 'b':
    ...
    case 'z': printf("Kleinbuchstabe\n"); break;
    case 'A':
    case 'B':
    ...
    case 'Z': printf("Grossbuchstabe\n"); break;
    case '0':
    case '1':
    ...
    case '9': printf("Ziffer\n"); break;
    default : printf("Anderes Zeichen\n"); break;
  }
```

Dieses Verhalten von `break` wird auch im nächsten Beispiel zur Ausgabe der Anzahl der Tage eines Monats benutzt. Nach Eingabe des Monats soll ausgegeben werden, ob der Monat 30, 31 oder 28 Tage hat.

```
 #include <stdio.h>

 void main(void)
 {
    int monat=0;

    printf("Bitte den gewünschten Monat (1−12) eingeben: ");
    fflush(stdout);
    scanf("%d",

    switch (monat)
    {
      case 1  :
      case 3  :
      case 5  :
      case 7  :
      case 8  :
      case 10 :
      case 12 : printf("Der Monat hat 31 Tage.\n");    break;
      case 2  : printf("Der Monat hat 28/29 Tage.\n"); break;
```

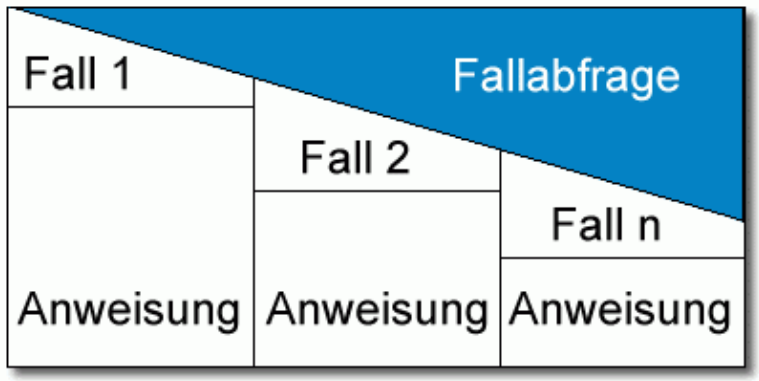
```
        default : printf("Der Monat hat 30 Tage.\n");    break;
      } /* Ende switch */

   }  /* Ende main */
```

Wird Januar, Maerz, Mai, Juli, August oder Oktober durch Eingabe der entsprechenden Zahl ausgewählt, so wird ebenfalls in den Zweig Dezember verzweigt. Dadurch wird für alle Monate, die 31 Tage haben, die korrekte Anzahl der Tage mit nur einer Anweisung ausgegeben. Die Anweisung `break` beendet den Durchlauf.
Handelt es sich bei der Eingabe um Februar, so verzweigt das Programm nicht in die ersten sechs Zweige, sondern erst in den Zweig Februar und gibt 28/29 als Anzahl der Tage aus. Die Anweisung `break` beendet auch hier den Durchlauf.
Für alle übrigen Eingaben wird 30 als Anzahl der Tage ausgegeben, auch wenn z.B. 100 eingegeben wurde.

Wird `default` ohne eine folgende Anweisung verwendet, so muß ein Semikolon folgen:

```
   default:;
```

Abbildung 15. Nassi Shneiderman Diagramm − switch



Syntaxdiagramme:

switch

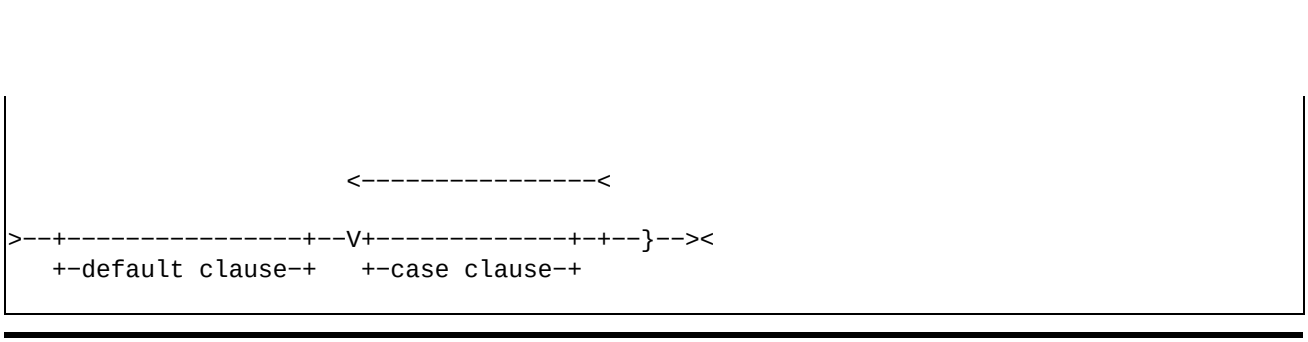
```
>>--switch--(--expression--)--switch body--><
```

switch body − einfache Form

```
>>--+-case label - statement-----------------------------------+--><
    |                                                         |
    +-+------------+--default label--+------------+--statement-+
      +-case label-+                 +-case label-+
```

switch body − komplexe Form

```
       <-----------------------------<  <----------------<

>>--{--V+--------------------------+-+--V+-------------+-+-->
        |-type definition----------|     +-case clause-+
        |                          |
        |-extern declaration-------|
        |                          |
        +-internal data definition-+
```


>--+-----------------+--V+-------------+-+--}--><
+-default clause-+ +-case clause-+

6.0 Wiederholungen − Schleifen

Um bestimmte Programmteile wiederholen zu können, werden die Wiederholungsanweisungen bzw. Schleifen verwendet. Sie gliedern sich wie folgt auf:

1. while − Schleife,
2. do−while − Schleife und
3. for − Schleife.

Allen Schleifen ist gemein, dass sie sich in einen Schleifenkopf (der bei der `do while`−Schleife aus zwei Teilen besteht) und einen Schleifenrumpf aufgliedern.

- Im Schleifenkopf befindet sich die Bedingung zur Ausführung eines Schleifendurchlaufs. Für diese Bedingung hat sich die Bezeichnung Abbruchkriterium etabliert. Diese Bezeichnung ist jedoch dahingehend irreführend, dass sie definiert, wann die Schleife weiterhin ausgeführt werden soll. (solange die Bedingung wahr ist, führe die Schleife aus)
- Im Schleifenrumpf befinden sich die Anweisungen, die bei jedem Schleifendurchlauf abgearbeitet werden sollen. Der Schleifenrumpf wird solange ausgeführt, wie der Ausdruck in der Klammer wahr ist.
  Befindet sich mehr als eine Anweisung im Schleifenrumpf, müssen diese Anweisungen, die bei jedem Schleifendurchlauf ausgeführt werden, in geschweifte Klammern ({}) eingefaßt sein.

Im folgenden werden die drei Schleifentypen vorgestellt. Für welche Schleife man sich zur Lösung eines Problems entscheidet, ist vorrangig Geschmackssache. Es gibt keinen Fall für den gilt, dass er nur mit einer der Schleifen zu lösen ist. Dementsprechend gibt es auch keine Empfehlungen, wann welche Schleife eingesetzt werden sollte. Die vielleicht komfortabelste ist die `for`−Schleife, da man bei ihr nicht Gefahr läuft, z.B. die Veränderung der Schleifenvariablen zu vergessen. Für die `do while` Schleife kann man sich entscheiden, wenn sichergestellt sein soll, dass die Schleife mindestens einmal durchlaufen wird.

6.1 while − Schleife

Die `while`−Schleife ist eine kopfgesteuerte Schleife. Das heißt, dass die Bedingung vor dem Schleifendurchlauf auf Wahrheit überprüft wird.
Beispiel:

```
n = 18;                                 /* Initialisierung der              */
i = 1;                                  /* Schleifenvariablen               */

while (i <= n)                          /* Start der Schleife / Schleifenkopf */
{
   printf("Ergebnis: %d\n", i*i);       /* Anweisungen im Schleifenrumpf      */
   i = i + 1;                           /* Veränderung der Schleifenvariablen */
}                                       /* Ende der Schleife                */
```

Diese Schleife gibt solange das Ergebnis von `i * i` aus, bis `i` den Wert von `n` (18) erreicht hat. Wenn `i` den Wert von `n` erreicht hat, wird der letzte Schleifendurchlauf gestartet. `i` wird bei jedem Schleifendurchlauf um eins erhöht.

Syntaxdiagramm:

while − statement
>>--while--(--expression--)--statement--><

Abbildung 16. Nassi Shneiderman Diagramm − while Schleife

6.2 do while − Schleife

Die `do while`−Schleife wird ebenfalls solange ausgeführt, wie der Ausdruck in der Klammer wahr ist. Allerdings wird erst nach dem Schleifendurchlauf geprüft, ob die Bedingung wahr ist. Es handelt sich somit um eine fußgesteuerte Schleife. Dementsprechend wird die Schleife mindestens einmal durchlaufen. Beispiel:

```
n = 18;
i = 1;

do
{
   printf("Ergebnis: %d\n", i*i);
   i = i + 1;
} while (i <= n);
```

Bei diesem Beispiel gibt es keinen Unterschied zwischen der `do while`−Schleife und der oben dargestellten `while`−Schleife. Eine kleine Änderung führt jedoch zu einem anderen Verhalten als es bei der `while`−Schleife der Fall wäre:

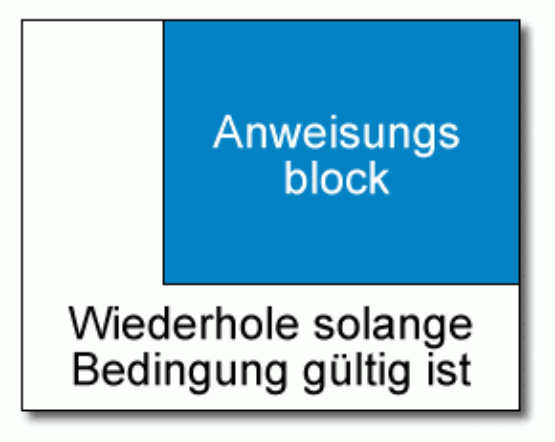
```
n = 0;

i = 1;

do
{
   printf("Ergebnis: %d\n", i*i);
   i = i + 1;
} while (i <= n);
```

Diese Schleife wird einmal durchlaufen obwohl `i` größer als `n` ist, da die Überprüfung des Abbruchkriteriums erst nach dem Durchlauf erfolgt.

Syntaxdiagramm:

do while − statement
>>--do--statement--while--(--expression--)--;--><

Abbildung 17. Nassi Shneiderman Diagramm − do while Schleife



6.3 for − Schleife

Die `for`−Schleife wird häufig dann eingesetzt, wenn zu Beginn der Schleife bekannt ist, wie oft sie durchlaufen werden soll. Wie bei der `while`−Schleife handelt es sich hier um eine kopfgesteuerte Schleife, d.h. vor dem Schleifendurchlauf wird das Abbruchkriterium geprüft.
Die folgende Abbildung demonstriert den Aufbau einer `for`−Schleife und die einzelnen Schritte, die beim Abarbeiten der Schleife durchlaufen werden.

Abbildung 18. Ablauf der for−Schleife

```
          1            2                4
for (zaehler=1; zaehler<= n; zaehler=zaehler+1)
{
    ...3
}
```

1. Initialisierung der Zählvariablen
2. Prüfung des Abbruchkriteriums
3. Abarbeitung des Schleifenkörpers
4. Veränderung der Zählvariablen
5. − weiter bei 2. −

Im Vergleich zu anderen Programmiersprachen sind drei Charakteristika bemerkenswert:

1. Die Codierweise der Addition.

2. Die Semikola innerhalb der runden Klammern. Sie grenzen die einzelnen Ausdrücke gegeneinander ab und sind zwingend erforderlich. Hinter dem dritten Ausdruck folgt jedoch kein Semikolon.

3. Nach den runden Klammern folgt kein Semikolon, sondern nur ein Blockzeichen (geschweifte Klammer), wenn der Schleifenrumpf nicht in den Schleifenkopf integriert wird. Dies wird weiter unten demonstriert.

Beispiele:

```
aufwärts zählend:

            for (i = 1; i <= n; i++)
            {
               ...anweisungen...
            }


abwärts zählend:

            for (i = n; i > n; i--)
            {
               ...anweisungen...
            }
```

Besonders häufig wird die `for`−Schleife zur Verarbeitung von Feldern (Arrays) verwendet. Das Beispiel zeigt die Initialisierung eines Feldes mit 50 Komponenten:

```
int zahlenfeld[50];
int index = 0, n = 50;

for (index = 0; index < n; index++)
{
   zahlenfeld[index] = index;
}
```

Die Angabe sämtlicher Ausdrücke in den Klammern des Schleifenkopfes ist nicht zwingend. Des weiteren können Anweisungen aus dem Schleifenrumpf häufig in den Schleifenkopf eingebunden werden. Die beiden folgenden Codesegmente sind funktionell identisch:

```
int zahlenfeld[50];
int index = 0, n = 50;

for (index = 0; index < n; index++)
{
  zahlenfeld[index] = index;
  printf("\n%d", zahlenfeld[index]);
}

// entspricht

for (index = 0, n = 50; index < n; printf("\n%d", zahlenfeld[index++] = index));
```

Bei derartigen Vereinfachungen sollte man jedoch aufpassen, dass es nicht zu Wechselwirkungen, insbesondere durch das Inkrementieren der Zählvariable, zwischen den einzelnen Operationen kommt.

Syntaxdiagramm:

for − statement

```
>>--for--(--+---------------+--;--+---------------+--;--+---------------+--)-->
            |               |     |               |     |               |
            +--expression1--+     +--expression2--+     +--expression3--+


>--statement--><
```

- `expression1` stellt die Initialisierung der Zählvariable dar.
- `expression2` stellt die Abbruchbedingung dar.
- `expression3` stellt die Veränderung der Zählvariablen dar.

Abbildung 19. Nassi Shneiderman Diagramm − for Schleife

6.4 Die Anweisungen break und continue

Die `continue`− und die `break`−Anweisung werden ausschließlich bei `switch` (siehe 5.2, "switch") und bei Schleifen und dort nur im Schleifenrumpf, dem Anweisungsteil, genutzt. `break` bewirkt einen sofortigen Abbruch der gesamten Schleife bzw. von `switch`. Die Durchführung wird mit dem Befehl nach der abschließenden geschweiften Klammer des Schleifenblocks fortgesetzt. `continue` ist bei `switch` nicht zulässig bzw. wirkungslos.

Bei der `continue`−Anweisung wird nur der aktuelle Schleifendurchlauf abgebrochen, d.h. es wird nicht die gesamte Schleifenverarbeitung beendet, sondern ausschließlich mit dem nächsten Schleifendurchlauf begonnen.

Beispiel für `break`:

```
for (i = 0; i < 21; i++)
{
   if (zeichenkette [i] == '#')
      break;
```

```
    position++;
}
```

Die Länge der Zeichenkette wird solange hochgezählt, bis das Zeichen # auftaucht.

Beispiel für `continue`:

```
for (i = 1; i <= 100; i++)
{
   if (i % 5 == 0) continue;
   printf("\n%d", i);
}
```

Es werden sämtliche Zahlen von 1 bis inklusive 100 ausgegeben, die nicht durch 5 teilbar sind.

6.5 Schleifen − Zusammenfassendes Beispiel

Im folgenden Beispiel werden die drei verschiedenen Schleifenformen nacheinander abgearbeitet.

ACHTUNG! Das Beispiel macht noch nicht so mordsmässig Sinn. Ein Beispiel, dass den Unterschied in der Anzahl Durchläufe etc. demonstriert, wäre angebracht.

```
#include <stdio.h>

void main(void)
{
    int n = 18, zaehler = 1;

    for ( zaehler = 1; zaehler <= n; zaehler = zaehler+1 )
    {
        printf("Ergebnis for: %d\n", zaehler*zaehler );
    }

    zaehler = 1; n = 18;
    do
    {
        printf("Ergebnis do-while: %d\n", zaehler*zaehler );
        zaehler = zaehler + 1;
    } while (zaehler <= n);

    zaehler = 1; n = 18;
    while (zaehler <= n)
    {
        printf("Ergebnis while: %d\n", zaehler*zaehler );
        zaehler = zaehler + 1;
    }

}
```

6.6 Häufige Fehler beim Einsatz von Schleifen

Ein häufiger Fehler ist die Angabe des Semikolons hinter einem Schleifenkopf, wenn ein Schleifenrumpf folgt:

```
for (i=0; i<100; i++) ;   /* <- dieses Semikolon ist falsch */
```

```
{
   ergebnis=ergebnis+i;
}
```

In diesem Beispiel wird der scheinbare Schleifenrumpf nur ein einziges Mal abgearbeitet und zwar mit dem Wert `i == 100`, da die Schleife durch das Semikolon abgeschlossen wird. Sie zählt lediglich `i` von 0 bis 100.

Eine weitere beliebte Fehlerquelle sind unbeabsichtigte Endlosschleifen:

```
int i=0, ergebnis=0;

while (i<100)
{
   ergebnis=ergebnis+i;
}
```

In diesem Beispiel wird die Zählvariable nicht verändert, so dass das Abbruchkriterium endlos gilt. Bei allen Schleifen ist darauf zu achten, dass die Zählvariable verändert wird, ansonsten wird eine Endlosschleife erzeugt!

7.0 Eigene Funktionen

Es wurde bereits erwähnt, dass Funktionen eigenständig eine definierte Aufgabe ausführen und dann die Kontrolle wieder an die aufrufende Funktion zurückgeben. Sie sind somit sehr nützlich wenn es darum geht eine Aufgabe mehrmals innerhalb eines oder verschiedener Programme zu erledigen.

Die Programmierung einer eigenen Funktion wird anhand einer Funktion zur Berechnung der Fakultät dargestellt. Der verwendete Compiler bietet unter Umständen keine Funktion zur Berechnung der Fakultät. Diese Berechnung wird jedoch häufig benötigt. Die einfachste Möglichkeit besteht nun darin, eine eigene Funktion zu formulieren und die Lösung des Problems somit auszulagern.

7.1 Definition eigener Funktionen

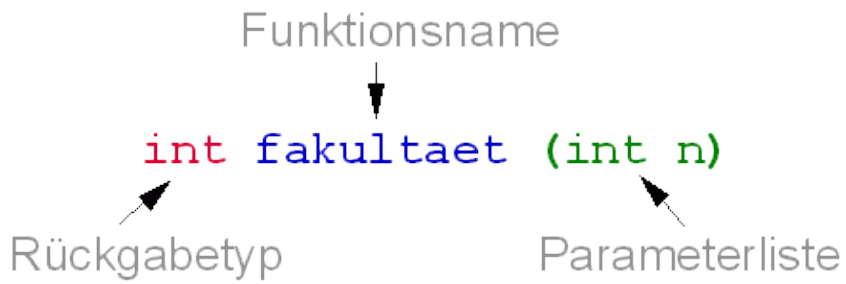
Eine Funktion besteht aus dem *Funktionskopf* und dem *Funktionsrumpf*. Der Funktionskopf beinhaltet Informationen über den Namen der Funktion, den Typ des Rückgabewertes und die Parameter, die an die Funktion übergeben werden sollen. Der Funktionsrumpf beinhaltet die Deklaration lokaler Variablen (optional), die Anweisungen, die die Funktion ausführen soll, und die Anweisung zur Beendigung der Funktion, die die Rückgabevariable enthält (optional).

Beispiel Berechnung der Fakultät:

```
int fakultaet (int n)          /* Funktionskopf                */
{                              /* Beginn Funktionsrumpf        */
   int i=0, produkt=1;         /* Deklaration der lokalen Var. */

   for (i=1; i <= n; ++i)
   {
       produkt *= i;           /* entspricht: produkt=produkt*i */
   }

   return(produkt);            /* Funktion beenden und den     */
                               /* Rückgabewert zurückgeben     */
}                              /* Ende Funktionsrumpf          */
```

`int fakultaet` vereinbart den Namen der Funktion (`fakultaet`) und den Typ des Resultats, das von der Funktion an die aufrufende Funktion zurückgegeben wird (`int`).

Abbildung 20. Bestandteile des Funktionskopfes



Wird kein Resultat zurückgegeben, muß vor dem Funktionsnamen `void` stehen. Dann darf dem Schlüsselwort `return` kein geklammerter Ausdruck folgen. Somit wird die Funktion beendet und kein Wert zurückgegeben.

`(int n)` stellt die Parameterliste dar. Die Parameterliste beschreibt diejenigen Variablen, die von der aufrufenden Funktion an die Funktion `fakultaet` übergeben werden sollen. Der Funktion `fakultaet` wird ein Argument vom Typ `int` beim Aufruf mitgegeben. Mehrere Parameter werden durch Kommata getrennt.
Würde es sich um eine Funktion handeln, die keine Variablen übergeben bekommt, müßte in den Klammern `void` stehen. Dies würde bedeuten, daß die Paramaterliste leer ist.

Mit `return(produkt);` wird die Funktion beendet und das Resultat `produkt` an die aufrufende Funktion zurückgegeben. Es ist auch `return produkt;` (ohne runde Klammern) möglich. Mit `return` kann immer nur ein Wert zurückgegeben werden.

Ein möglicher Aufruf wäre:

```
fak = fakultaet(7);
```

Das Resultat des Funktionsaufrufes `fakultaet(7)` wird durch das Schlüsselwort `return` an die aufrufende Funktion zurückgegeben und dabei der Variablen `fak` zugewiesen.

Ein weiteres Beispiel soll die Verwendung des Schlüsselwortes `void` verdeutlichen:

```
void copyright (void)

{
   printf("%s\n,%s\n,%s\n,%s\n,%s\n",
   " /**************************************/",
   " /* Copyright by    DaKliSaWo          */",
   " /* BAB Hannover    IBM intern         */",
   " /* (c) 1994                           */",
   " /**************************************/");
}
```

Das erste `void` legt fest, daß die Funktion `copyright()` kein Resultat an die aufrufende Funktion zurückgibt. Daß die Funktion keine Argumente erwartet, wird durch das zweite `void` in der Parameterliste festgelegt. Der Aufruf lautet:

```
copyright();
```

Bei der Definition eigener Funktionen ist auf den Gültigkeitsbereich der Variablen zu achten. Siehe dazu auch Speicherklassen im Abschnitt 4.6. "Speicherklassen".

7.2 Konstante Parameter

Konstante Parameter sind Parameter, deren Wert während der Abarbeitung der Funktion nicht verändert werden kann. Dies bietet sich in einigen Situationen als Sicherheitsmaßnahme an, wenn z.B. Konfigurationsparameter an Funktionen übergeben werden, die nicht geändert werden dürfen.
Beispiel: Eine Kontonummer wird an eine Funktion `abbuchen()` übergeben. Die Funktion soll zwar eine Abbuchung eines Betrages von einem Konto (identifiziert durch die Kontonummer durchführen), aber nicht − auch nicht ungewollt − die Kontonummer ändern.

```
#include <stdio.h>

int abbuchen(const int kontoNummer, int betrag)
```

```
{
   /* kontoNummer=2000; */ /* Compiler Fehlermeldung: Operand must be a modifiable lvalue.
   printf("%d\n",kontoNummer);

   ...

   return kontostand;
}

void main(void)
{
   int kontoNummer = 1000, betrag = -200;

   abbuchen(kontoNummer, betrag);
}
```

7.3 Call by value und call by reference

Bei Funktionen unterscheidet man *call by value* Funktionen und *call by reference* Funktionen. Die oben vorgestellte Funktion `fakultaet()` ist eine call by value Funktion, da an sie der Wert einer Variablen übergeben wird. Eine call by reference Funktion erwartet nicht den Wert einer Variablen, sondern deren Adresse in Form eines Zeigers. In diesem Fall wird der Wert der Variablen nicht ein zweites Mal im Speicher abgelegt und mit dieser Kopie gearbeitet, sondern es wird direkt mit dem Original gearbeitet.
Der Vorteil von call by reference gegenüber call by value ist, dass ohne Probleme die Werte der Variablen geändert werden können und diese geänderten Werte auch in der aufrufenden Funktion Gültigkeit haben. Bei call by value ist eine derartige dauerhafte Änderung nur mit Rückgabe des Wertes möglich. Da aber immer nur ein Wert per `return` zurückgegeben werden kann, muss bei mehreren Werten mit call by reference gearbeitet werden.

Beispiel für call by value:

```
int add(int, int);

void main(void)
{
   int a=3, b=7, ergebnis;

   ergebnis = add(a,b);
   printf("Ergebnis von %d + %d = %d\n",a,b,ergebnis);
}

int add(int x, int y)
{
   int ergebnis;

   ergebnis = x + y;
   return(ergebnis);
}
```

Beispiel für call by reference mit Rückgabewert per `return`:

```
int add(int *, int *);

void main(void)
{
   int a=3, b=7, ergebnis;
```

```
   ergebnis = add(
   printf("Ergebnis von %d + %d = %d\n",a,b,ergebnis);
}

int add(int *x, int *y)
{
   int ergebnis;

   ergebnis = *x + *y;
   return(ergebnis);
}
```

Beispiel für "call by reference" mit Rückgabewert per Zeiger:

```
void add(int *, int *, int *);

void main(void)
{
   int a=3, b=7, ergebnis;

   add(
   printf("Ergebnis von %d + %d = %d\n",a,b,ergebnis);
}

void add(int *x, int *y, int *ergebnis)
{
   *ergebnis = *x + *y;
}
```

7.4 Prototypen

Bei der Definition von Funktionen ist darauf zu achten, wo man sie im Quelltext definiert. Funktionen, die man vor dem Ort ihres Aufrufes definiert, bereiten keine Probleme. Der Compiler, der den Quelltext von oben nach unten durchgeht, hat zum Zeitpunkt des Aufrufes die Funktionsdefinition bereits analysiert und weiß, welchen Typ der Rückgabewert der Funktion hat und wieviele Parameter welchen Typs in welcher Reihenfolge von dieser Funktion erwartet werden.

Wird die Funktion erst nach ihrem ersten Aufruf definiert, so nimmt der Compiler an, daß ihr Rückgabewert vom Typ `int` ist. Dies wird als implizite Funktionsdeklaration bezeichnet. Ist in der Definition ein anderer Typ angegeben, so läuft das Programm spätestens beim Linker auf einen Fehler (z.B. auf einen *redefinition error*). Wie dieses Problem, daß auch auftritt, wenn Funktionen sich gegenseitig aufrufen, umgangen werden kann, wird im folgenden erläutert.

Um Funktionen nach dem Ort ihres ersten Aufrufes definieren zu können, bietet C die Möglichkeit der Prototypen. Prototypen stellen eine Vorgehensweise dar, wie dem Compiler zu Beginn des Quelltextes mitgeteilt werden kann, welche Funktionen in einem Programm vorkommen, welchen Typs ihre Rückgabewerte sind, welche Parameter sie erwarten und welche Namen diese Funktionen haben. Dies ermöglicht dem Compiler, die korrekte Verwendung der Funktionen in einem Programm zu überprüfen. Die Prototypen sollten direkt nach den Preprocessoranweisungen folgen.

Funktionsname und Parametertypliste ohne Rückgabewert werden als Signatur bezeichnet.

Das folgende Beispiel demonstriert den Einsatz von Prototypen:

```
/*****************************************************************
 *  proto.c                                                      *
 *  demonstriert den einsatz von prototypen                      *
 *****************************************************************/

#include <stdio.h>

double berechnung(double, double);              /*   Prototyp    */

void main(void)
{
     double a1=0.0, a2=0.0;

     a1=3.14;
     a2=218;

     printf("\nErgebnis = %lf", berechnung(a1,a2));
}

double berechnung(double a1, double a2)
{
   return (a1 * a2);
}
```

In dem obigen Beispiel wird ein Prototyp für die Funktion `berechnung()` eingeführt. Der Prototyp enthält folgende Elemente:

1. den Typ des Rückgabewertes,
2. den Namen der Funktion,
3. die Typen der Parameter, die die Funktion erwartet, in Klammern und
4. ein abschließendes Semikolon.

Die generelle Verwendung von Prototypen für sämtliche Funktionen in jedem Programm empfiehlt sich aus mehreren Gründen:

- Die Prototypen nehmen wenig Platz ein. In Verbindung mit einem Kommentar ermöglicht eine Reihe von Prototypen eine nützliche Übersicht über die einzelnen Funktionen, ihren Sinn, den Typ ihres Rückgabewertes und ihre Parameter in einem Programm. Der Reihe der kommentierten Prototypen kann die Funktion `main()` folgen, was das Programm sehr übersichtlich und leicht lesbar macht. Die eigentliche Funktionsdefinition kann im Anschluß erfolgen.
- Funktionen können nach ihrem ersten Aufruf definiert werden und sind dabei nicht auf den Typ `int` für den Rückgabewert beschränkt.
- Funktionen können sich gegenseitig aufrufen.

Die oben genannten Vorteile stellen die wichtigsten Gründe für die Verwendung von Prototypen dar. Die Verwendung ist jedoch nicht zwingend, sämtliche Funktionen können ohne Prototypen definiert werden.

Achtung!
Da implizite Funktionsdeklarationen in C++ nicht erlaubt sind, ist in C++ die Verwendung von Funktionsprototypen zwingend.

7.5 Variable Anzahl an Parametern beim Funktionsaufruf

Die bisher vorgestellten Möglichkeiten sind auf Funktionen mit einer fest definierten Anzahl an Parametern begrenzt. Funktionen wie `printf()` bieten jedoch die Möglichkeit, sie mit einer variablen Anzahl an Parametern aufzurufen:

```
printf("\nAufruf mit einem Parameter.\n");
printf("\nAufruf mit %d Parametern.\n",2);
printf("\n%d. Aufruf mit %s Parametern.\n",anzahl,typ);
```

Das folgende Beispiel demonstriert die grundsätzliche Vorgehensweise anhand der Funktion `name()`, die als Parameter einen Integerwert und eine unbestimmte Anzahl an Zeichenketten entgegennimmt.

```
/*
   Bearbeitung einer unbestimmten Anzahl Parameter, die bei
   einem Funktionsaufruf angegeben werden.
*/

#include <stdarg.h>

int name(int i, ...)
{
  va_list arg_ptr;
  char *zeichenkette;

  /* arg_ptr für Zugriffe durch va_arg und va_end initialisieren */
  /* i ist der letzte benannte Parameter vor ... und in diesem   */
  /* Fall der einzige Parameter, der angegeben werden muß.       */
  va_start(arg_ptr, i);

  /* solange einen Parameter holen und in Zeichenkette ablegen, */
  /* bis keine Parameter mehr vorhanden sind                    */
  while((zeichenkette=va_arg(arg_ptr, char*))!='\0')
  {
       printf("%s",zeichenkette);
       i++;
  }

  fflush(stdout);

  /* parameter suche beenden */
  va_end(arg_ptr);

  return i;
}

void main(void)
{
  int i=0;

  i=name(200,"Hallo",", dies"," ist ein ","Test.");
}
```

8.0 Formatierte Ein− und Ausgabe in C

ACHTUNG! Überarbeiten!

Das ist insgesamt zu verwirrend und unübersichtlich. Manche Beispiele sind auch nicht sehr hilfreich.

8.1 printf − formatierte Ausgabe auf dem Bildschirm

Die Funktion `printf()` dient zur formatierten Ausgabe von Daten jeglicher Art. Es ist z.B. möglich, die Form, in der Zahlen ausgegeben werden zu beeinflussen.

Die Syntax von `printf()` lautet:

printf(const char *format−string, argument−list);

- `const char` stellt eine Zeichenkonstante dar, die so ausgegeben wird, wie sie als Konstante vorliegt. Diese Konstante kann sogenannte Fluchtsymbolzeichen (s.u.) beinhalten. Soll das Prozentzeichen ausgegeben werden, muß es doppelt (%%) in der Zeichenkonstante vorkommen, da es ansonsten einen Formatstring einleitet.
  `*format string` stellt die Formatspezifikation dar, die festlegt, in welchem Format das jeweilige Argument ausgegeben werden soll. Diese Formatstrings werden innerhalb der Zeichenkonstante verwendet.

- Die `argument−list` benennt die Variablen, die mit der Anweisung ausgegeben werden sollen. Die Zuordnung Formatstring zu Variable verläuft von links nach rechts. Das bedeutet, daß der zuerst auftauchende Formatstring dem ersten Datenelement der argument−list zugeordnet wird, der zweite Formatstring dem zweiten Datenelement u.s.w. Die einzelnen Datenelemente werden durch Kommata voneinander getrennt.

Beispiel:

```
printf("Das %d. Ergebnis ist : %d\n", nummer, ergebnis);
```

`Das Ergebnis ist :` stellt die Zeichenkonstante dar, die unverändert ausgegeben wird.

`%d` stellt den Formatstring dar, der sich aus dem Prozentzeichen (`%`) und dem Umwandlungszeichen `d` zusammensetzt. Das Prozentzeichen kennzeichnet den Beginn einer Formatspezifikation. Das `d` bedeutet, daß die hinter dem Komma stehende Variable als Integervariable interpretiert und daß der Wert der Variablen an der durch `d` bezeichneten Stelle ausgegeben werden soll. Der Formatstring kann noch weitere Zeichen enthalten, die weiter unten vorgestellt werden.

Dem ersten `%d` wird `nummer` zugeordnet, dem zweiten `%d` wird `ergebnis` zugeordnet.

Wenn Nummer den Wert 4 und Ergebnis den Wert 20 enthält, sieht die Ausgabe wie folgt aus:

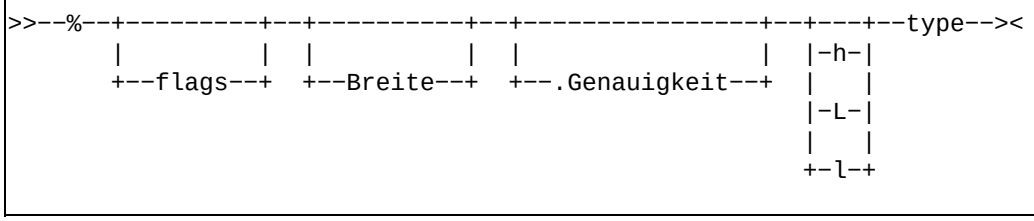
```
Das 4. Ergebnis ist : 20
```

Der Cursor steht nach der Ausgabe am Anfang der nächsten Zeile.

8.1.1 Erläuterung des Formatstrings

Die folgende Formatspezifikation stellt den Aufbau des Formatstrings dar.

printf() − Formatspezifikation



Die Formatspezifikationen beginnen mit einem Prozentzeichen (%) und enden mit dem Umwandlungszeichen (*type*; z.B. d, s, c, f oder e). Dazwischen können folgende Zeichen angeführt werden:

1. Folgende Steuerzeichen (*flags*) in beliebiger Reihenfolge
    a. '−': linksbündige Ausgabe des Datenelements
    b. '+': Zahlen werden immer mit Vorzeichen ausgegeben
    c. '0': leere Stellen im Ausgabefeld mit führenden/angehängten Nullen füllen
2. eine Zahl, die die minimale Feldbreite festlegt (bei Bedarf wird die Feldbreite vergrößert)
3. einen Punkt, der die Feldbreite von der Genauigkeit trennt und eine Zahl für die Genauigkeit mit folgender Bedeutung:
    a. maximale Anzahl von Zeichen für eine Zeichenkette
    b. Anzahl der Nachkommastellen für Fließkommazahlen (z.B. `%.0lf`)
    c. die minimale Anzahl von Ziffern für Ganzzahlenwerte
4. die Größe des Argumentes
    - h − Präfix für die Integertypen (d, i, n, o, u, x und X), der festlegt, daß das Argument `short int` oder `unsigned short int` ist.
    - l − Präfix für die Typen d, i, n, o, u, x und X, der festlegt, daß das Argument `long int` oder `unsigned long int` ist. Dieser Präfix kann ebenfalls bei den Typen e, f und g benutzt werden (`double` Typen).
    - L − Präfix für die Typen e, f und g, der festlegt, daß das Argument `long double` ist.
5. Der *type* kann aus folgenden Zeichen bestehen:

c	für char
d	für signed int (dezimal)
e	für float−Variablen, allerdings wird der Wert in Exponentenschreibweise ausgegeben.
E	wie e, aber E für den Exponent statt e

f für float−Variablen, Ausgabe in Festkommaschreibweise

g für float−Variablen. Der Compiler entscheidet abhängig von der Länge, ob Exponential− oder Festkommadarstellung gewählt wird.

G wie g, aber E für den Exponent statt e

i für signed int (dezimal)

n Nummer der Zeichen, die bis jetzt erfolgreich auf die Ausgabe geschrieben wurde.

o für signed int (oktal)

p für Zeigervariablen

s für Zeichenketten (char−Felder)

u für unsigned int (dezimal)

x für unsigned int (hexadezimal, a−f)

X für unsigned int (hexadezimal, A−F)

8.1.2 printf() − Beispiele

Die folgenden Abschnitte enthalten Beispiele zur Anwendung von `printf()`.

8.1.2.1 printf() − Beispiele − Zahlen

Das folgende Beispiel demonstriert die Ausgabe von Zahlen.

```
/* einfache Ein− und Ausgabe */

#include <stdio.h>

int main ( int argc, char *argv[] )
{
   int    intZahl1=3,      intZahl2=4,      intErgebnis=0;
   double doubleZahl1=3.2, doubleZahl2=4.1, doubleErgebnis=0.0;

   intErgebnis = intZahl1 + intZahl2;
   printf ("Zahl1 %d + Zahl2 %d = %d\n",
            intZahl1,  intZahl2,  intErgebnis);

   doubleErgebnis = doubleZahl1 + doubleZahl2;
   printf ("Zahl1 %.2lf + Zahl2 %.2lf = %.2lf (double)\n",
            doubleZahl1, doubleZahl2, doubleErgebnis);

   return 1;

}  /* ende von main() */
```

8.1.2.2 printf() − Beispiele − Zeichenketten

Die grundsätzliche Verwendungsweise soll mit Hilfe einiger Beispiele an einer Zeichenkette verdeutlicht werden, die Feldbreite der Zeichenkette beträgt 21 Zeichen: `EIN−, AUSGABEFUNKTION`

Die Beispiele sind folgendermaßen aufgebaut: Nummer des Beispiels, Formatstring, Ausgabe. Eine Beschreibung folgt in der jeweils nächsten Zeile. Leerzeichen, die ausgegeben werden, werden durch den Unterstrich (_) gekennzeichnet.

```
(1)  %s           EIN-, AUSGABEFUNKTION
```
Ist kein Feldformat verlangt, hat das Feld die Breite der Zeichenkettenkonstante.

```
(2)  %21s         EIN-, AUSGABEFUNKTION
```
Die Feldbreite entspricht der Länge der Zeichenkonstante.

```
(3)  %-21s        EIN-, AUSGABEFUNKTION
```
Eine Linksausrichtung ist nicht möglich, da die Feldbreite der Länge der Zeichenkettenkonstanten entspricht.

```
(4)  %15s         EIN-, AUSGABEFUNKTION
```
Die Feldbreite von 15 gibt nur den minimalen Wert der Feldbreite an.

```
(5)  %-15s        EIN-, AUSGABEFUNKTION
```
Eine Linksausrichtung ist nicht möglich.

```
(6)  %25s         ____EIN-, AUSGABEFUNKTION
```
Da die Feldbreite größer als die Länge der Zeichenkettenkonstanten ist, wird diese rechtsbündig eingefügt.

```
(7)  %-25s        EIN-, AUSGABEFUNKTION____
```
Linksausrichtung der Zeichenkettenkonstanten.

```
(8)  %25.10s      _______________EIN-, AUSG
```
In ein Feld mit 25 Elementen werden die ersten 10 Zeichen der Zeichenkettenkonstanten rechtsbündig ausgegeben.

```
(9)  %-25.10s     EIN-, AUSG_______________
```
10 Zeichen der Zeichenkettenkonstanten werden linksbündig ausgegeben.

```
(10)  %-025.10s    EIN-, AUSG000000000000000
```
Wie oben, allerdings werden Leerstellen mit Nullen aufgefüllt.

```
(11)  %.10s        EIN-, AUSG
```
Die Feldbreite entspricht der geforderten Zeichenanzahl.

```
```

```c
/*
printf() Demonstration

1. Beispiel - Zeichenketten:
   - Demonstration von nummerierten Parametern (%n$)
   - Demonstration von minimaler Ausgabebreite (ns)
   - Demonstration von maximaler Ausgabebreite (.ns)
2. Beispiel - Zeichenketten:
   - wie erstes Beispiel, aber ohne nummerierte Parameter
3. Beispiel - Zeichenketten:
   - über Variable gesteuerte Ausgabebreite (%*)
*/

#include <stdio.h>


int main(int argc, char *argv[])
{
  char zeichenkette[80]="Hallo";
  int  anzahl=3;

  /*
      %1$s
      ====
      % : Anfang der Formatspezifikation
      1$: 1. Parameter der Parameterliste
      s : Auszugebender Parameter ist vom Typ char* (->Zeichenkette)

      %1$3s
      =====
      % : Anfang der Formatspezifikation
      1$: 1. Parameter der Parameterliste
      3 : Ausgabebreite mindestens 3 Zeichen
      s : Auszugebender Parameter ist vom Typ char* (->Zeichenkette)

      %1$.3s
      ======
      % : Anfang der Formatspezifikation
      1$: 1. Parameter der Parameterliste
      .3: Ausgabebreite maximal 3 Zeichen
      s : Auszugebender Parameter ist vom Typ char* (->Zeichenkette)

  */
  printf("Beispiel 1:\n%1$s\n%1$3s\n%1$.3s\n",zeichenkette);

  /* Gleiche Ausgabe ohne nummerierte Parameter */
  printf("Beispiel 2:\n%s\n%3s\n%.3s\n",zeichenkette,zeichenkette,zeichenkette);

  /*
     Ausgabe mit variabler Anzahl vorangestellter Leerzeichen.
     Hier wird die Möglichkeit ausgenutzt, die Ausgabebreite durch eine Variable
     in der Parameterliste (hier: anzahl) zu steuern.
     Da die Ausgabebreite mit Leerzeichen aufgefüllt wird, kann mit der gleichen
     Variante keine Voranstellung von z.B. Punkten erreicht werden.
  */
  printf("Beispiel 3:\n%*c%s\n",anzahl,'',zeichenkette);

  return 1;
} /* Ende main() */
```

8.1.3 Fluchtsymbolzeichen

In den obigen Beispielen ist mehrfach die Zeichenfolge `\n` aufgetreten. Sie ist eines von mehreren Fluchtsymbolzeichen, auch *Escape−Sequenz* genannt, die der Formatierung der Ausgabe dienen und aus einem Backslash (`\`) und einem nachfolgenden Buchstaben bestehen. Mögliche Zeichenfolgen sind:

- \\ gibt den Backslash aus
- \' gibt das einfache Hochkomma (') aus
- \" gibt die Anführungszeichen bzw. das doppelte Hochkomma (") aus
- \0 Die binäre Null schließt eine Zeichenkette ab, wird von Funktionen zur Zeichenkettenbearbeitung automatisch generiert.
- \a *alarm*; gibt einen Warnton aus.
- \b *backspace*; bewegt den Cursor um eine Stelle nach links.
- \f *formfeed*; veranlaßt einen Seitenvorschub, d.h. daß der nachfolgende Text auf einer neuen Seite ausgegeben wird.
- \n markiert einen Zeilenvorschub (*new line*). Die nachfolgende Ein− oder Ausgabe wird in einer neuen Zeile begonnen.
- \r *carriage return*/Wagenrücklauf; bewegt den Cursor in derselben Zeile an den Zeilenanfang.
- \t setzt ein Tabulatorzeichen. Der folgende Text wird um einen definierten Wert, der in der Entwicklungsumgebung eingestellt wird, nach rechts verschoben.

Anmerkung: Um die Zeichen `\` und `%` darzustellen, werden sie doppelt eingegeben.

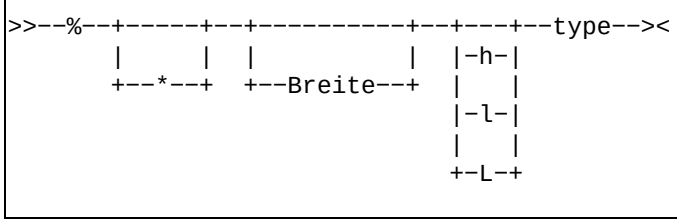
Bsp: `printf("Wechsel in das directory \"C:\\OS2\\APPS\" verläuft ...");` ergibt folgende Ausgabe: `Wechsel in das directory "C:\OS2\APPS"` verläuft...

8.2 scanf − formatiertes Lesen vom Standardeingabegerät

scanf(const char *format−string, argument−list);

Die Bedeutung der einzelnen Komponenten entspricht der von `printf()` (siehe oben).

scanf() − Formatspezifikation



Der Aufbau soll anhand des folgenden Beispiels erläutert werden:

```
Bscanf("%d",
```

`%d` bezeichnet wie bei der `printf()`−Funktion eine Formatspezifikation. Theoretisch gelten alle bei `printf()` angesprochenen Formatierungsmöglichkeiten auch für `scanf()`. Dem Benutzer sollte jedoch

vor der Eingabe mitgeteilt werden, wieviele Zeichen und in welchem Format er die Zeichen eingeben soll.

Die einzigen Unterschiede liegen bei der Breite des Eingabefeldes, des optionalen Sterns (*) und den Typen `X, E` und `G`. Die Angabe der Breite bei `printf()` gibt die Mindestbreite des Ausgabefeldes an. Bei `scanf()` wird hier die maximale Breite des Eingabefeldes festgelegt. Der optionale Stern nach dem Prozentzeichen unterdrückt die Zuweisung des nächsten Eingabefeldes. Das Feld wird analysiert, aber nicht gespeichert. Die Typen `X, E` und `G` stehen bei `scanf()` nicht zur Verfügung.

`/font> ist zusammengesetzt aus und der Variablen ergebnis, die den zu lesenden Wert aufnimmt. Das eichen markiert in C den Zugriff auf eine Adresse (siehe auch 4.2.4, "Zeiger − Pointer"). Die Argumente bei scanf() müssen Zeiger sein, da sonst die Werte der Variablen und nicht deren Adressen an scanf() übergeben werden. Bezeichnern von Felder (siehe Abschnitt 4.2.1, "Felder − Array − Vektoren") wird kein vorangestellt, da sie stets einen Zeiger auf den eigentlichen Inhalt darstellen.`

8.2.1 scanf() − Beispiele

1. Beispiel

```
int x = 0;

scanf("%5d",
Eingabe: 1234567
```

`x` bekommt nur die ersten fünf Ziffern (12345) der Eingabe zugewiesen, da die Feldbreite nicht überschritten werden darf. Die Ziffern 67 befinden sich jedoch noch im Eingabepuffer der Standardeingabe (stdin). Folgt der oben angegebenen Anweisung eine weitere, so wartet diese nicht eine erneute Eingabe ab, sondern nimmt lediglich das, was bereits im Eingabepuffer steht!

2. Beispiel

```
float zahl1 = 0.0, zahl2 = 0.0;

scanf("%5f+%5f",
1. Eingabe: 12345+6789
2. Eingabe: 123456+123456
```

Die erste Eingabe weist den Variablen die entsprechenden Werte zu. Zu beachten ist, daß hierbei eine komplette Addition eingegeben wird und nicht mehrere Werte nacheinander. Diese Form erscheint zunächst sehr komfortabel, birgt aber einige Gefahren in sich. Bei der zweiten Eingabe werden lediglich die ersten fünf Ziffern (12345) zugewiesen, da die Feldbreite nur fünf Zeichen beträgt. Die Zahl 6, das Additionszeichen und die Zahlen 123456 befinden sich noch im Eingabepuffer. Der Variablen `zahl2` wird jedoch kein Wert zugewiesen, sie behält den Initialisierungswert Null.

3. Beispiel

```
char ch = 0;

scanf("%c",
Eingabe: IBM
```

`ch` bekommt nur `I` übergeben, da eine Variable des Typs `char` nur ein Zeichen aufnehmen kann.

4. Beispiel

```
/*
   einfache Ein- und Ausgabe mit printf() / scanf()

   je zwei Integerwerte und zwei Doublewerte von der Standardeingabe
   lesen, addieren und das Ergebnis auf der Standardausgabe ausgeben

   printf(): Daten formatiert auf Standardausgabe ausgeben
   scanf() : Daten von Standardeingabe lesen
   fflush(): Ein-/Ausgabepuffer leeren
*/

#include <stdio.h>

int main ( int argc, char *argv[] )
{
   int    intZahl1=3,      intZahl2=4,      intErgebnis=0;
   double doubleZahl1=3.2, doubleZahl2=4.1, doubleErgebnis=0.0;

   printf ("Bitte intZahl1 eingeben: ");
   fflush (stdout);          /* Standardausgabepufferinhalt ausgeben */
   fflush (stdin);           /* Standardeingabepufferinhalt löschen  */
   scanf  ("%d",  /* intZahl1 von Standardeingabe lesen   */

   printf ("Bitte intZahl2 eingeben: ");
   fflush (stdout);          /* Standardausgabepufferinhalt ausgeben */
   fflush (stdin);           /* Standardeingabepufferinhalt löschen  */
   scanf  ("%d",  /* intZahl2 von Standardeingabe lesen   */

   intErgebnis = intZahl1 + intZahl2;
   printf ("Zahl1 %d + Zahl2 %d = %d\n",
            intZahl1,  intZahl2,  intErgebnis);

   printf ("Bitte doubleZahl1 eingeben: ");
   fflush (stdout);
   fflush (stdin);
   scanf  ("%lf",

   printf ("Bitte doubleZahl2 eingeben: ");
   fflush (stdout);
   fflush (stdin);
   scanf  ("%lf",

   doubleErgebnis = doubleZahl1 + doubleZahl2;
   printf ("Zahl1 %.2lf + Zahl2 %.2lf = %.2lf (double)\n",
            doubleZahl1, doubleZahl2, doubleErgebnis);

   return 1;

}  /* ende von main() */
```

8.2.2 Zeiger Dereferenzierung bei scanf()

```
/*
   Beispiel: Zeiger Dereferenzierung bei scanf()

   scanf() benötigt als zweiten Parameter die Hauptspeicheradresse der Variablen, in der
   der eingegebene Wert gespeichert werden soll.
```

```
    * Normalerweise gilt:

      int zahl=0;
      scanf("%d",       <- Adressoperator ()notwendig

    * ABER: Wenn wir einen Zeiger auf die Variable haben, in der wir den eingegebenen Wert
            speichern möchten, darf KEIN Adressoperator folgen, da wir bereits die
            Hauptspeicheradresse haben...

      int *zahl;              <- Zeiger auf Integer, enthält nach Funktionsaufruf die Adres
      scanf("%d", zahl);

*/


#include <stdio.h>


int eingabe(int *zahl);


int main(int argc, char *argv[])
{
  int zahl=0;

  zahl=eingabe(
  printf("Eingegebene Zahl: %d\n",zahl);

  return 1;
} /* Ende main() */



int eingabe(int *zahl)
{

  printf("Bitte eine Zahl eingeben: ");
  fflush(stdout);
  fflush(stdin);
  scanf("%d", zahl);

  return *zahl;
} /* Ende eingabe() */
```

8.3 Eingabe− und Ausgabepuffer leeren

Syntax: `fflush(FILE *);`

Die Funktion `fflush()` dient zum Leeren der Eingabe− bzw. Ausgabepuffer. Beim Leeren des Ausgabepuffers werden die noch enthaltenen Daten auf den Bildschirm geschrieben.

Eingabepuffer leeren:

```
fflush(stdin);
```

Ausgabepuffer leeren:

```
fflush(stdout);
```

Puffer einer Datei leeren:

```
fflush(FILE *);       /* Dateizeiger angeben, z.B. fflush(quelle); */
```

Das Leeren des Ausgabepuffers empfiehlt sich vor jeder `scanf()` (oder vergleichbaren) Anweisung und das Leeren des Eingabepuffers empfiehlt sich nach jeder `scanf()` (oder vergleichbaren) Anweisung.

9.0 Übersicht über wichtige Funktionen

Im folgenden werden einige Beispiele für wichtige Funktionen vorgestellt, die C bereits mitliefert. Bei diesen handelt es sich um sogenannte *Standardbibliotheksfunktionen*. Standardbibliotheksfunktionen sind Funktionen, die C vordefiniert zur Lösung bestimmter Aufgaben zur Verfügung stellt. Sie sind in den Definitionsdateien (auch Include− oder Headerdateien, Dateien mit der Erweiterung `.h`) definiert.

9.1 Funktionen zur Bearbeitung von Zeichenketten und Zeichen

In diesem Abschnitt wird detailliert auf die Bearbeitung von Zeichenketten eingegangen. Am Ende dieses Abschnittes befindet sich eine Übersicht über die gebräuchlichsten Funktionen zur Zeichenkettenmanipulation.

In einem Zeichenfeld soll die Zeichenkette `Hallo` gespeichert werden. Im Speicher ergibt sich dann folgendes Bild:

Element	0	1	2	3	4	5
Inhalt	H	a	l	l	o	\0

Die binäre Null wird von der jeweiligen Zeichenkettenfunktion selbsttätig an das Ende der Zeichenkette gesetzt. Wird eine Zeichenkette manuell erzeugt, sollte die binäre Null aus Sicherheitsgründen manuell angefügt werden.

```
/*********************************************************************/
/* Programm: zeichen.c                                               */
/* Autor   : Sascha Kliche, Daniel Wolkenhauer                       */
/* Datum   : September 1994                                          */
/* Beschreibung:                                                     */
/* Dieses Programm verdeutlich die automatische Benutzung der        */
/* binären Null und führt in einzelne Zeichenkettenfunktionen        */
/* ein.                                                              */
/*********************************************************************/

#include <stdio.h>
#include <string.h>                                              (1)

void main(void)
{
   /* Vereinbarungsteil der lokalen Variablen */
   char satz[80];                                                (2)

   /* Lesen der Zeichenkette */
   printf("Bitte einen Satz eingeben!\n");
   gets(satz);                                                   (3)

   /* Ausgabe der Zeichenkette */
   printf("%s\n", satz);

   /* Ermitteln der Länge der Zeichenkette und Länge ausgeben */
   printf("Der Satz beinhaltet %i Zeichen!\n", strlen(satz));    (4)
   printf("satz[4] wird jetzt mit der binären Null gefüllt.");

   /* Das fünfte Element mit der binären Null füllen. */
   satz[4]='\0';                                                 (5)
   printf("%s", satz);

   /* Neue Länge der Zeichenkette ermitteln und ausgeben. */
   printf("Der Satz besteht nun nur noch aus %i Zeichen!", strlen(satz));
```

```
}
```

(1) Für einige der Zeichenketten−Funktionen wird eine eigene Standardbibliotheksdatei benötigt. Die entsprechende Definitionsdatei hat den Namen `string.h`.

(2) Die Anweisung `char satz[80];` reserviert Speicherplatz für 80 Zeichen. Bei der Arbeit mit Funktionen zur Zeichenkettenverarbeitung sollte jedoch bedacht werden, daß diese automatisch die binäre Null an das Ende der Zeichenkette anhängen. Soll eine Zeichenkette mit 80 Zeichen mit Funktionen zur Zeichenkettenverarbeitung bearbeitet werden, muß die Deklaration `char satz[81];` lauten. In diesem Fall sollen 79 nutzbare Zeichen reichen.

(3) Die Funktion `gets()` liest Zeichenketten von der Standardeingabe (Tastatur) in das innerhalb der Klammern angegebene char−Feld. Zusätzlich setzt es bei Eingabe von **ENTER** automatisch die binäre Null an das Ende der Zeichenkette.

(4) Mit Hilfe der Funktion `strlen()` ist es möglich, die Länge einer Zeichenkette zu ermitteln. Dabei werden alle Zeichen einschließlich der Leerzeichen bis zur ersten binären Null gezählt.

(5) Durch das Füllen des fünften Elementes mit der binären Null wird das Ende der Zeichenkette verlegt. Sämtliche Zeichen hinter dem fünften Element werden nicht mehr ausgegeben.

Im folgenden werden die verwendeten Funktionen ausführlicher vorgestellt.

9.1.1 Die Funktion gets()

Die Funktion `gets()` dient der Eingabe von Zeichenketten. Als ersten und einzigen Parameter erwartet `gets()` einen Zeiger auf ein char−Feld. `gets()` prüft − wie leider die meisten Funktionen − nicht, ob der Speicherbereich für die Daten groß genug ist!

```
char eingabe[80]={""};
  ...
printf("Bitte den Namen eingeben: ");
fflush(stdout); fflush(stdin);

gets (eingabe);

printf("Eingabe: %s\n",eingabe);
```

9.1.2 Die Funktion strlen()

Die Funktion `strlen()` ermittelt die Länge einer Zeichenkette, in dem die Anzahl Zeichen bis zum ersten Auftauchen einer binären Null gezählt wird.

```
char eingabe[80]={""};
int laenge=0;
  ...
printf("Bitte den Namen eingeben: ");
fflush(stdout); fflush(stdin);
gets(eingabe);

laenge=strlen(eingabe);

printf("%s enthält %d Zeichen\n",eingabe,laenge);
```

9.1.3 Die Funktion sprintf()

Die Funktion `sprintf()` dient wie `printf()` der formatierten Ausgabe von Daten, allerdings werden die Daten nicht auf die Standardausgabe sondern an die Speicheradresse des ersten Parameters geschrieben. Dementsprechend muss der erste Parameter ein Zeiger auf einen Speicherbereich sein, der gross genug ist um die Daten aufzunehmen. `sprintf()` prüft nicht, ob der Speicherbereich für die Daten gross genug ist!

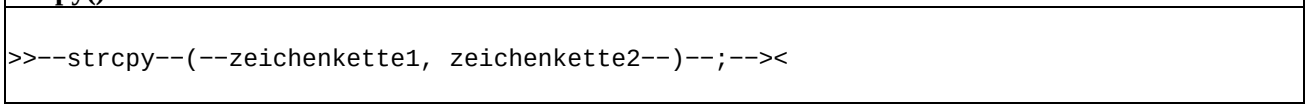
```
char text[80]={""};
int  anzahl = 4;
   ...
sprintf(text, "%d Sahneis\0", anzahl);
/* (sicherer als strcpy, wenn \0 angegeben wird) */
```

9.1.4 Die Funktionen strcpy() und strcat()

Die Funktionen `strcpy()` und `strcat()` dienen zum Kopieren von Zeichenketten in andere Zeichenketten. `strcpy()` kopiert Zeichenketten immer an den Anfang von Zeichenketten und `strcat()` hängt Zeichenketten an das Ende anderer Zeichenketten an.

Die Syntax von `strcpy()` lautet: ***char *strcpy(char *zeichenkette1, const char *zeichenkette2);***

strcpy()



`zeichenkette2` wird an den Speicherplatz von `zeichenkette1` kopiert.
Nach der Anweisung `strcpy(satz, "Sahne");` hat der reservierte Platz im Speicher folgendes Aussehen (vorausgesetzt, die Vereinbarung hatte die Form: `char satz[9]`):

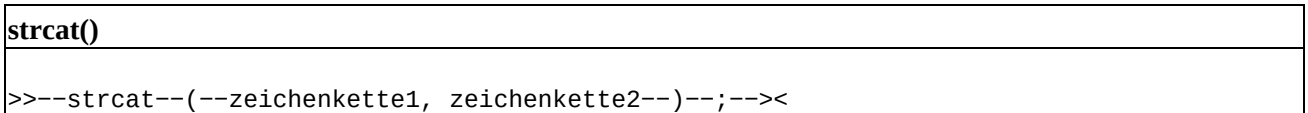
Element	0	1	2	3	4	5	6	7	8
Inhalt	S	a	h	n	e	\0			

Je nach Compiler hängt die Funktion `strcpy()` die binäre Null automatisch an das Ende der Zeichenkette an. Verlassen kann man sich darauf jedoch nicht.

Um an `zeichenkette1` die Zeichenkette `eis` anzuhängen, wird die Funktion `strcat` benutzt, die folgende Syntax hat:

char *strcat(char *zeichenkette1, const char *zeichenkette2);

strcat()



Das Ergebnis von

```
strcat(satz, "eis");
```

hat folgende Auswirkung:

Element	0	1	2	3	4	5	6	7	8
Inhalt	S	a	h	n	e	e	i	s	\0

Mit dem ersten Zeichen des anzuhängenden Textes wird die binäre Null überschrieben und an das neue Ende gesetzt.

Die gleiche Anweisung mit `strcpy()` hätte die ersten vier Speicherstellen durch `eis\0` ersetzt. Die Speicherstellen fünf bis neun behielten ihren Wert, würden jedoch von vielen Funktionen nicht mehr beachtet, da eine binäre Null davor stünde.

Element	0	1	2	3	4	5	6	7	8
Inhalt	e	i	s	\0	e				

Wird dem Feld ein Wert zugewiesen, der länger als der reservierte Speicherplatz ist, so werden diese überzähligen Zeichen nicht abgeschnitten, sondern in den angrenzenden Speicherbereich geschrieben, so daß dort andere Variablenwerte oder Rücksprungadressen verloren gehen können. Der Compiler prüft nicht auf ausreichende Länge der Variable! Der Programmierer muß dafür sorgen, daß der reservierte Speicherbereich groß genug für die Aufnahme der Zeichenkette ist, anderenfalls kann es zu unvorhergesehenen Fehlern bis hin zu Programmabstürzen kommen. Diese Form von Fehler (*buffer overflow*) ist eine der Hauptursachen für Programme mit Sicherheitslücken.

9.1.5 Typumwandlung nach Dateneingabe per Zeichenkette

Die in 8.2, "scanf − formatiertes Lesen vom Standardeingabegerät" vorgestellte Funktion `scanf()` hat einige Nachteile. Diese führen dazu, dass häufig der Weg gewählt wird, zunächst eine Zeichenkette mit einer Funktion wie z.B. `gets()` einzulesen und dann in den Zieldatentyp wie z.B. `int` zu konvertieren. Die Funktion `gets()` erwartet als einzigen Parameter den Namen eines `char`−Feldes.

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

int main(int argc, char *argv[])
{
  char eingabe[80];
  int kontonummer=0;
  int kontostand=2000;
  double betrag=0.0;

  printf("Bitte geben Sie die Kontonummer ein: ");
  fflush(stdout); fflush(stdin);
  gets(eingabe);
  printf("Ihre Eingabe: %s\n",eingabe);

  /* kontonummer=(int)eingabe; <- typecast geht hier nicht */
  kontonummer=atoi(eingabe);
  printf("Kontonummer: %d\n",kontonummer);

  printf("Bitte geben Sie den Abbuchungsbetrag ein: ");
  fflush(stdout); fflush(stdin);
  scanf("%lf",
  printf("Alter Kontostand: %d − Abbuchungsbetrag: %lf\n",kontostand,betrag);

  kontostand = kontostand − (int)betrag;
  printf("Neuer Kontostand: %d\n",kontostand);

  return 1;
}
```

9.1.6 Auswahl häufig benutzter Funktionen zur Zeichen− und Zeichenkettenbearbeitung

Die Programmiersprache C bietet in ihren Standardbibliotheken verschiedene Funktionen zur Manipulation von Zeichen(ketten) an (Auswahl)

Funktionsname	Erklärung

`tolower (zeichen)`	Wandelt `zeichen` in einen Kleinbuchstaben um, wenn es ein Großbuchstabe ist. Ansonsten wird der Kleinbuchstabe unverändert zurückgeliefert. (Definitionsdatei: `ctype.h`)
	`char klein = 0;`
	`klein = tolower ('A');`

`toupper (zeichen)`	Wandelt `zeichen` in einen Großbuchstaben um, wenn es sich um einen Kleinbuchstaben handelt. Ansonsten wird der Großbuchstabe unverändert zurückgeliefert. (Definitionsdatei: `ctype.h`)
	`char gross = 0;`
	`gross = toupper ('a');`

`strcat (kette1, kette2)`	Kopiert die Zeichenkette `kette2` an das Ende der Zeichenkette `kette1`, überschreibt dabei die binäre Null und setzt sie an das neue Ende. `strcat` geht davon aus, daß in `kette1` genügend Platz vorhanden ist. Die Funktion liefert einen Zeiger auf den Anfang der Gesamtkette als Ergebnis zurück. (Definitionsdatei: `string.h`)
`strncat (kette1, kette2, n)`	Kopiert n Zeichen von `kette2` an das Ende von `kette1`. (Definitionsdatei: `string.h`)
`strcmp (kette1, kette2)`	Vergleicht `kette1` und `kette2` byteweise. Es wird ein positiver Wert zurückgeliefert, wenn `kette1` größer ist als `kette2`, ein negativer, wenn sie kleiner ist und Null, wenn sie identisch sind. Siehe auch stricmp und strnicmp. (Definitionsdatei: `string.h`)
`strncmp(kette1, kette2, n)`	Vergleicht nur n Zeichen in `kette1` und `kette2`. Die Bedeutung der Rückgabewerte ist identisch mit denen von `strcmp`. (Definitionsdatei: `string.h`)
`strlen (kette)`	Liefert als Rückgabewert die Länge der Zeichenkette `kette` zurück, ohne die binäre Null mitzuzählen. (Definitionsdatei: `string.h`) `length = strlen(string);`
`strcpy (kette1, kette2)`	Kopiert `kette2` in `kette1`. (Definitionsdatei: `string.h`)

10.0 Dateibehandlung

In diesem Kapitel wird die Arbeit mit Dateien unter C vorgestellt. Es wird auf die grundlegenden Funktionen, wie z.B. Dateien öffnen/schließen, Daten aus Dateien lesen, Daten in Dateien schreiben und Daten an bestehende Dateien anhängen, eingegangen. Des weiteren wird auf die Unterschiede in der Benutzung von Text− und Binärdateien hingewiesen.

Das grundsätzliche Vorgehen bei der Arbeit mit Dateien sieht wie folgt aus:

1. Datei(en) öffnen
2. auf Fehler beim Öffnen prüfen
3. Daten aus Datei(en) lesen / in Datei(en) schreiben
4. Datei(en) schliessen

10.1 Datei öffnen/schließen

Bevor Dateien verarbeitet werden können, müssen sie von dem Programm für die Bearbeitung vorbereitet werden. Diese Vorbereitung nennt man Datei öffnen. Das Betriebssystem wird veranlaßt Speicher bereitzustellen, in dem Informationen bezüglich der geöffneten Datei abgelegt werden. Nach der Nutzung der Datei muß diesie wieder geschlossen werden, damit die neuen Informationen dauerhaft auf dem Datenträger gespeichert werden. Anhand eines Beispiels wird die Vorgehensweise erläutert.

Dieses Beispiel öffnet eine Datei und schließt sie wieder, wenn der Versuch, sie zu öffnen, erfolgreich verlaufen ist:

```
/*  Datei_io.c                            */
/*  demonstriert die funktionen zum öffnen */
/*  und schließen von dateien in c        */

#include <stdio.h>

void main(void)
{
   /* Zeiger für die Datei deklarieren      */
   /* Für jede gleichzeitig geöffnete Datei    */
   /* muß ein derartiger Zeiger existieren.    */
   FILE *dateiZeiger=NULL;                              (1)

   /* Datei öffnen und Return−Code annehmen */
   dateiZeiger = fopen("TEST.TXT","w");                 (2)

   /* Return−Code auswerten                 */
   if (dateiZeiger != NULL)
   {
      /* wenn kein fehler, datei schließen  */
      fclose(dateiZeiger);                              (3)
   }
}
```

zu (1): Mit `FILE *dateiZeiger;` wird ein Zeiger mit dem Namen *dateiZeiger* definiert. Für jede gleichzeitig geöffnete Datei muß ein Zeiger mit einem eigenen Namen deklariert werden, dessen Name in

allen folgenden Operationen (Beispiele hier: (2),(3)) eingesetzt wird, die sich auf die dazugehörige Datei beziehen. Die Anweisung kann gelesen werden als: `dateiZeiger` ist ein Zeiger auf ein Objekt des Typs `FILE`, hinter dem sich eine Struktur verbirgt. `FILE` ist ein vordefinierter Typ aus der Definitionsdatei `stdio.h`, der immer groß geschrieben wird und für Dateizugriffe benötigt wird.

zu (2): Die Funktion `fopen()` öffnet die Datei `TEST.TXT`. Die Syntax der Funktion lautet:

```
zeiger = fopen(dateiname, zugriffsmodus);
```

`dateiZeiger` nimmt den Rückgabewert der Funktion `fopen()` auf. Dieser ist entweder ein Zeiger auf die Dateistruktur der geöffneten Datei oder `NULL`, wenn beim Öffnen der Datei ein Fehler aufgetreten ist. `TEST.TXT` ist der Dateiname der Datei, die geöffnet wird und `w` der Zugriffsmodus. Der Zugriffsmodus bestimmt die Art und Weise, in der auf die Datei zugegriffen werden kann. Die folgende Liste enthält die zulässigen Werte für den Dateizugriffsmodus:

Modus	**Beschreibung**
r	Die Datei wird nur zum Lesen geöffnet. Voraussetzung ist, daß die Datei bereits existiert. Ansonsten bekommt der Zeiger `NULL` zugewiesen.
w	Eine leere Datei wird zum Schreiben geöffnet. ACHTUNG! Falls die Datei bereits besteht, wird die Dateilänge auf Null gesetzt. Dies entspricht dem Löschen des Inhaltes der Datei.
a	Die Datei wird nur zum Schreiben geöffnet. Die neuen Daten werden am Ende angehängt, wenn die Datei existiert, anderenfalls wird sie erstellt.
r+	Eine existierende Datei wird zum Lesen und Schreiben geöffnet.
w+	Eine Datei wird zum Lesen und zum Schreiben geöffnet und erstellt, falls sie noch nicht existiert. Anderenfalls wird der Inhalt gelöscht.
a+	Eine Datei wird sowohl zum Lesen als auch zum Anhängen an das Dateiende geöffnet. Existiert sie noch nicht, so wird sie erstellt.

Außer diesen Zugriffsmodi muß dem Compiler mitgeteilt werden, ob die Datei im binären Modus oder im Text−Modus geöffnet werden soll. Dies geschieht durch das Anhängen eines `b` für binär an das Kürzel für den Zugriffsmodus. Für den Zugriff im Text−Modus ist kein Kürzel erforderlich. Dennoch ist es bei einigen Compilern möglich, ein `t` für den Text−Modus anzuhängen. Auf dieses Kürzel sollte jedoch verzichtet werden, um die Portabilität des Codes zu gewährleisten, da einige Compiler dieses Kürzel nicht unterstützen. So übersetzen manche Compiler ein Programm, das dieses Kürzel enthält, zwar anstandslos und ohne Fehlermeldung oder Warnung, beim Betrieb des Programms ist dieses dann jedoch nicht in der Lage, Dateien zu öffnen.

zu (3): Mit `fclose(dateiZeiger);` wird die Datei geschlossen. Dabei führt das Betriebssystem eine Reihe von Schritten aus:

1. Die zur Zwischenpufferung genutzten Speicher werden geleert und die noch nicht gespeicherten Daten auf den Datenträger geschrieben.
2. Die Verbindung zwischen Datei und Dateizeiger wird unterbrochen, so daß der Zeiger für eine andere Datei verwendet werden kann.
3. Der Systembereich der Festplatte (oder eines anderen Massenspeichers) wird aktualisiert (Größe, Position, Datum/Zeitpunkt der letzten Modifikation, etc.).

10.2 Zeichen und Zeichenketten aus Dateien lesen

Um den Inhalt einer Datei lesen zu können, muß sie im Lesezugriff geöffnet und eine Variable deklariert werden, die die gelesenen Daten aufnimmt. Im folgenden Beispiel wird die Datei `TEST.TXT` geöffnet und die Variable `zeichen` zur Aufnahme der Daten deklariert: Die Datei wird zeichenweise gelesen, das heisst, dass bei jedem Lesevorgang immer nur ein einzelnes Zeichen aus der Datei gelesen wird.

```
/*   datei_2.c                                              */
/*   demonstriert das lesen von textdaten aus einer datei   */
/*   der Inhalt wird zeichenweise gelesen                   */

#include <stdio.h>


void main(void)
{
   /* deklaration lokaler variablen */
   FILE *dateiZeiger=NULL;                  /* Dateizeiger   */
   char datei[20]="TEST.TXT";               /* Dateiname     */
   char zugriff[3]="r";                     /* Zugriffsmodus */
   char zeichen;

   /* datei oeffnen und return-code auswerten */
   dateiZeiger = fopen(datei , zugriff);

   if (dateiZeiger == NULL)       /* öffnen fehlgeschlagen */
   {
     printf("Die Datei %s konnte nicht geöffnet werden!", datei);
     exit(1);
   }
   else                           /* öffnen geglückt */
   {
     /* zeichen aus der datei lesen */
     zeichen=fgetc(dateiZeiger);
     while (!feof(dateiZeiger))
     {
           putchar(zeichen);      /*  zeichen auf der standardausgabe
                                     ausgeben */
           zeichen=fgetc(dateiZeiger);
     }

     /* datei schliessen */
     fclose(dateiZeiger);
     printf("Die Datei %s wurde geschlossen!", datei);

   } /* ende öffnen geglückt */

} /* ende main */
```

In der `while`-Schleife werden mit `fgetc()` solange einzelne Zeichen aus der, mit `dateiZeiger` referenzierten, Datei gelesen und der Variablen `zeichen` zugewiesen, bis sie den Wert `EOF` [8] erreicht hat. Ist das Ende der Datei noch nicht erreicht, wird jedes einzelne Zeichen ausgegeben.

Der Funktionsaufruf `feof(dateiZeiger)` prüft, ob diese Dateiendemarkierung erreicht wurde oder nicht.

Anmerkung: 'Zeichen' wird in der Funktion 'datei_lesen' als Integer-Wert deklariert, da es jedes beliebige ASCII-Zeichen inklusive dem EOF-Zeichen annehmen können muß. In einigen C Implementierungen liegt der Wert von EOF außerhalb des Wertebereichs des Typs char, da diese standardmäßig unsigned char verwenden.

Die Funktion `exit(Rückgabewert)` beendet das C−Programm mit sofortiger Wirkung. Es werden alle Puffer geleert und alle geöffneten Dateien automatisch geschlossen. Den Rückgabewert `1` kann ein aufrufendes Programm abfragen.

Beim Lesen von Zeichenketten aus einer Datei ergeben sich nur geringfügige Änderungen gegenüber dem Lesen von einzelnen Zeichen.

In diesem Beispiel wird die Dateiverarbeitung vollständig in Funktionen ausgelagert. Das Hauptprogramm hat nur noch aufrufende und auf Fehler testende Funktion. Die Dateifunktionen zum Öffnen, Lesen und Schließen wurden bereits erläutert.

```
/*   zeichenk_lesen.c                                          */
/*   demonstriert das lesen von zeichenketten aus einer datei  */
/*                                                             */

#include <stdio.h>

FILE *dateiZeiger=NULL;

void datei_oeffnen (char datei[], char zugriff[])
{
  dateiZeiger = fopen(datei , zugriff);
  return;
}

/* die einzigen unterschiede zwischen dem lesen einzelner zeichen  */
/* und dem lesen von strings aus einer datei sind in der funktion  */
/* datei_lesen zu finden. es werden andere variablendeklarationen  */
/* und die funktion fgets benutzt.                                 */
void datei_lesen (void)
{
   char satz[256];
   do
   {
      fgets(satz, 256, dateiZeiger);
      fputs(satz, stdout);
   } while (!feof(dateiZeiger));
  return;
}

void datei_schliessen (void)
{
  fclose(dateiZeiger);
  return;
}

void main(void)
{

   char datei[12]="TEST.TXT";
   char zugriff[3]="r";

   datei_oeffnen(datei, zugriff);
   if (dateiZeiger == NULL)
   {
     printf("Die Datei %s konnte nicht geöffnet werden!", datei);
     exit(1);
   }
   else
   {
     datei_lesen();
```

```
      datei_schliessen();
      printf("Die Datei %s wurde geschlossen!", datei);
   }
}
```

Die Änderungen wirken sich nur in der Funktion `datei_lesen` aus. Statt `fgetc()` wird `fgets()` und statt `putchar()` wird `fputs()` verwendet. Dabei ist zu bemerken, daß `fgets()` und `fputs()` mehr Parameter benötigen, als die Funktionen zur zeichenweisen Behandlung. Die Funktion `fgets()` erwartet drei Parameter. Der erste Paramater stellt die Adresse des Daten aufnehmenden Feldes dar, der zweite Parameter die Größe dieses Feldes und der dritte Parameter kennzeichnet als Zeiger die Quelle der zu lesenden Zeichenkette.

Die Angabe des Größenparameters ist notwendig, da nicht mehr Zeichen eingelesen werden dürfen, als `satz` aufnehmen kann. Die Datei wird immer bis zum Ende (EOF, End Of File) gelesen und mit Hilfe von `fputs()` ausgegeben. Diese Funktion übernimmt zwei Parameter. Der erste Parameter ist die auszugebende Zeichenkette und der zweite Parameter bezeichnet die Ausgabe (hier: Standard Output/ i.d.R. der Bildschirm).

10.3 Zeichen in Dateien schreiben

Im folgenden werden die Funktionen zum Schreiben von einzelnen Zeichen und Zeichenketten vorgestellt. Sie sind ohne Probleme in die oben genannten Programme einbindbar.

Streng genommen ist das Schreiben von Daten bereits behandelt worden, nur nicht in eine Datei, sondern auf den Bildschirm. Der `putc()` Funktion muß lediglich die Adresse der Datei mit Hilfe des Zeigers (`dateiZeiger`) gegeben werden.

In dem folgenden Beispiel wird zum Lesen der Zeichen von der Tastatur die Funktion `getchar()` verwendet.

```
void datei_schreiben(void)
{
   char zeichen;

   /* solange zeichen einlesen, bis enter gedrückt wird */
   while ((zeichen = getchar()) != '\n')
         /* zeichen in die datei schreiben */
         putc(zeichen, dateiZeiger);
  return;
}
```

10.3.1 Zeichenketten in eine Datei schreiben

Wie beim Lesen von Zeichenketten aus einer Datei, sind auch beim Schreiben von Zeichenketten in eine Datei einige zusätzliche Angaben notwendig.

```
void datei_schreiben(void)
{
   char satz[256]

   /* satz mit einer maximalen länge von 256 zeichen einlesen */
   fgets(satz, 256, stdin);

   /* solange wiederholen, bis in einer leeren zeile enter    */
   /* gedrückt wird                                           */
```

```
    while(satz[0] == '\n')
    {
       /* satz in datei schreiben */
       fputs(satz, datei_ptr2);

       /* neuen satz lesen */
       fgets(satz, 256, stdin);
    }
  return;
}
```

Es werden solange Sätze mit einer maximalen Anzahl von 256 Zeichen von der Tastatur eingelesen und in die Datei geschrieben, bis in einer leeren Zeile die Taste **ENTER** gedrückt wird.

Die Funktion `fputs()` schreibt Zeichenketten in die mit `datei_ptr2` bezeichnete Datei.

10.3.2 Anhängen von Daten an eine bereits vorhandene Datei

Das Anhängen von Daten erfolgt nicht über eine spezielle Funktion, sondern über den Zugriffsmodus. Je nachdem, ob die Datei nur zum Anhängen oder auch zum Lesen geöffnet werden soll, verwendet man `a` oder `a+`. Zum Schreiben der Daten werden die bereits erwähnten Funktionen `fputs()` und `putc()` verwendet.

10.4 Das Arbeiten mit binären Dateien

Der Unterschied zwischen Text− und Binärdateien besteht in den unterschiedlichen Informationen, die in ihnen enthalten sind. Textdateien enthalten Zeichenfolgen, die für den Menschen verständlich sind. Sie bestehen aus alphanumerischen Zeichen, Satzzeichen und einigen Steuerzeichen. Diese Art von Dateien kann zeilenorientiert aufgebaut sein.

Binärdateien besitzen keine Zeilenstruktur und sind für den Menschen in der Regel unverständlich. Ein weiterer Unterschied besteht in der Codierung von Zahlen. In der binären Datei werden Zahlen abgelegt, wie sie im Hauptspeicher vorliegen (z.B. `int` mit 4 Byte in einer bestimmten Reihenfolge) und nicht − wie in Textdateien − als eine Folge von Ziffern.

Bei einer Binärdatei sieht der Modus zum Öffnen einer Datei wie folgt aus:

```
rb
wb
ab
rb+
wb+
ab+
```

10.4.1 Binäres Lesen von Integer Daten

Prinzipiell werden alle Datentypen in Blöcken à 1 Byte gespeichert, wobei die Reihenfolge der Bytes im Speicher je nach Betriebssystem und Prozessor unterschiedlich sein kann. Ein 4 Byte langer Integer mit dem Wert x wird binär wie folgt dargestellt (Big Endian):

```
 11110000 00001111 11111111 00000000
```

Auf bestimmten Systemen werden diese vier Bytes jedoch in umgekehrter Reihenfolge abgelegt (Little Endian):

```
00000000 11111111 00001111 11110000
```

Oder auch folgendermaßen:

```
11111111 00000000 11110000 00001111
```

Allgemein unterscheidet man die Varianten *Little Endian* (z.B. Intel x86, low Bytes −> high Bytes) und *Big Endian* (z.B. RS/6000, high Bytes −> low Bytes). Little Endian meint "least significant byte first", bei Integer und Floating Point. Die Reihenfolge der Bits ist gleich, lediglich die Reihenfolge der Bytes ist umgekehrt.

Der folgende Beispielcode liest vier Byte aus einer Datei und speichert diese als (long) Integer in Abhängigkeit einer symbolischen Konstante, die die Zielhardware spezifiziert.

```
void readINT32(FILE *file_in, UINT32 *integer)
{
  unsigned char p[4];

  p[0]=fgetc(file_in); p[1]=fgetc(file_in);
  p[2]=fgetc(file_in); p[3]=fgetc(file_in);

  # ifdef RS6000
  // Big Endian - RS6000
  *integer=(((((p[0] << 8) | p[1]) << 8) | p[2]) << 8) | p[3];
  # else
  // Little Endian - DOS, OS/2 on Intel x86
  *integer=(((((p[3] << 8) | p[2]) << 8) | p[1]) << 8) | p[0];
  # endif

  return;
} //--- end - readINT32()
```

Das folgende Beispiel liest zwei Byte aus einer binären Datei und interpretiert sie als short int.

```
void readINT16(FILE *bmp_in, UINT16 *integer)
{
  unsigned char p[2];

  p[0]=fgetc(bmp_in); p[1]=fgetc(bmp_in);

  # ifdef RS6000
  // Big Endian - RS6000
  *integer=(p[0] << 8) | p[1]; // Big Endian - AIX
  # else
  // Little Endian - DOS, OS/2 on Intel x86
  *integer=(p[1] << 8) | p[0]; // Little Endian - DOS, OS/2
  #endif

  return;
} //--- end - readINT16()
```

Mit dieser Funktion kann z.B. die Höhe und Breite einer Bitmap ausgelesen werden, deren Werte als 19−22. Bytes einer Bitmap gespeichert werden. Z.B. ergibt sich aus `FD 02 87 01` 765x391 Pixel (`02 FD`x`01`

87).

10.4.2 Binäres Lesen von Fließkommadaten

Für die Darstellung von Fließkommazahlen im Speicher existieren für fast alle Prozessorsysteme unterschiedliche Varianten, was den Datenaustausch erschwert. Von der IEEE existieren allerdings Standards wie z.B. IEEE−754.

10.5 Zusammenfassung Lesen/Ausgeben von Daten

Die folgende Tabelle zeigt die zum Lesen und Augeben möglichen Funktionen. Auf Funktionen, die noch nicht in den Beispielen vorgekommenen sind, wird im Anschluß kurz eingegangen.

Tabelle 11. Funktionen zum Lesen/Ausgeben

Schreiben	Lesen	Daten
putc(), fputc()	getc(), fgetc()	Zeichen
fputs()	fgets()	Zeichenkette
fprintf()	fscanf()	formatierte Daten
fwrite()	fread()	Block

Die Funktionen `fprintf()` und `fscanf()` werden für das Übertragen von Feldern mit einer festen Länge verwendet. Der Gebrauch unterscheidet sich nur in der Angabe der Datei von den Funktionen `printf()` und `scanf()`.

Beispiel:

```
fprintf(dateiZeiger, "Horst ist %d Jahre alt!\n", jahre);
```

Mit dieser Anweisung wird in der angegebenen Datei ein Satz mit einer variablen Angabe des Alters abgespeichert und das Zeilenendezeichen angefügt.

Die Funktionen `fwrite()` und `fread()` schreiben beziehungsweise lesen `n` Datenobjekte der Größe `size` aus/in einem/n Puffer aus/in die mit `dateiZeiger` angegebene Datei.

Beispiel:

(1) Syntax:

```
fwrite(buffer, size, n, *fp);
```

Diese Anweisung speichert n Objekte der Größe `size` aus dem Speicherbereich ab der Adresse von `buffer` in die Datei, auf die der Dateizeiger `*fp` verweist.

Anwendung:

```
fwrite(katalog, sizeof(BUCH), 200, dateiZeiger);
```

Diese Anweisung speichert 200 Objekte der Größe der Struktur `BUCH` aus dem Speicherbereich ab der Adresse von `katalog` in die Datei, auf die der Dateizeiger `dateiZeiger` verweist.

(2) Syntax:

```
fread(buffer, size, n, *fp);
```

Mit dieser Anweisung werden die Daten aus einer Datei in den Speicherbereich ab der Adresse von `buffer` geschrieben. `size` beschreibt die Größe und n die Anzahl der Objekte.

Anwendung:

```
fread(katalog, sizeof(BUCH), 200, dateiZeiger);
```

Mit dieser Anweisung werden die Daten aus einer Datei, auf die der Dateizeiger `dateiZeiger` verweist, in den Speicherbereich ab der Adresse von `katalog` geschrieben. Die Größe der Objekte soll der Größe der Struktur `BUCH` entsprechen. Es sollen 200 dieser Objekte gelesen werden.

11.0 Preprocessor−Anweisungen (Direktiven)

Der Preprocessor ist ein Programm, welches den Quellcode vor der eigentlichen Übersetzung bearbeitet. Alle Preprocessor−Anweisungen beginnen in Spalte eins mit dem Zeichen # und gehören nicht zum eigentlichen Sprachumfang von C. Sie werden vor dem Compilieren verarbeitet. Es stehen folgende Anweisungen zur Verfügung:

```
#define
#elif
#else
#endif
#error
#if
#ifdef
#ifndef
#include
#line
#pragma
#undef
```

Diese Anweisungen werden in den folgenden Abschnitten anhand von Beispielen vorgestellt.

11.1 Symbolische Konstanten − #define

Mit der Anweisung `#define suchtext ersatztext` besteht die Möglichkeit, einer Konstanten einen Namen zuzuordnen. Der Preprocessor ersetzt vor dem Compilieren alle Textstellen im Programm, die mit `suchtext` übereinstimmen, durch die in `ersatztext` angegebene Zeichenfolge.
Beispiel:

```
#include <stdio.h>
#define PI 3.14159265358979

void main(void)
{
   ... ;
   erg = (PI * radius) / 2;
   ...;
}
```

Sämtliche Textstellen im Quelltext, die `PI` lauten, werden durch `3.14159265358979` ersetzt.

Einige Konstanten sind bereits definiert, wie z.B. die folgenden:

`__DATE__`	das Datum der Compilierung als Zeichenkette (Format: "Mmm dd yyyy")
`__FILE__`	der Dateiname der aktuellen Quelltextdatei als Zeichenkette
`__LINE__`	die Zeilennummer in der aktuelle Quelltextdatei als Integer
`__TIME__`	die Uhrzeit der Compilierung als Zeichenkette (Format: "hh:mm:ss")
`__TIMESTAMP__`	der aktuelle Timestamp der Compilierung als Zeichenkette (Format: "Day Mmm dd hh:mm:ss yyyy")

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

```
void main(void)
{
   printf("Zeile %d in Datei %s am %s um %s.\n",
          __LINE__, __FILE__, __DATE__, __TIME__);
}
```

11.2 Makros − #define

Es gibt Fälle, in denen eine abgeschlossene Teilaufgabe so kurz ist, daß es sich nicht lohnt, eine vollständige Funktion dafür zu programmieren. In diesen Fällen benutzt man ein Makro.

Die Definition eines Makros hat die Form `#define name anweisungen`.

Beispiel:

```
#include <stdio.h>
#define max(A,B)   ((A) > (B) ? (A) : (B))

void main(void)
{
   int zahl_1, zahl_2;

   printf("\n\nGeben Sie 2 ganze Zahlen durch ein Komma getrennt ein!\n");
   scanf("%d,%d",
   printf("Das Maximum dieser beiden Zahlen ist %d\n", max(zahl_1,zahl_2));
}
```

Die Definition von `max(A,B)` prüft, ob der Ausdruck `A` größer als der Ausdruck `B` ist. Ist dies der Fall, wird der Ausdruck `A` zurückgegeben, ist dies nicht der Fall, so wird der Ausdruck `B` zurückgegeben. An jeder Stelle im Quelltext, an der von der Definition `max(A,B)` Gebrauch gemacht wird, wird das Makro ausgeführt.

11.3 Übersetzungsabbruch − #error

Durch diese Anweisung wird der Übersetzungsablauf unterbrochen. Diese Anweisung findet vor allem bei bedingter Compilierung Anwendung.

11.4 Einfügen von Dateien − #include

C bietet zum Einfügen von Dateien in den Quelltext die Preprocessor−Anweisung `#include`. Wird der Dateiname der einzufügenden Datei in Hochkommata eingeschlossen, so sucht der Compiler zuerst in der benutzereigenen Bibliothek und dann in der Standardbibliothek. Wird der Dateiname hingegen in spitze Klammern (<>) eingebettet, so wird sofort und ausschließlich in der Standardbibliothek gesucht.

11.5 #if

Syntax: `#if konstanter_ausdruck anweisungen`

Trifft der Ausdruck zu, wird der, bis zum `#endif` folgende Code compiliert.

11.6 #ifdef

Syntax: `#ifdef ausdruck anweisungen`

Ist der Ausdruck als Makro vorher definiert worden, wird der folgende Code compiliert.

11.7 #ifndef

Syntax: `#ifndef ausdruck anweisungen`

Ist der Ausdruck nicht vorher als Makro definiert worden, wird der folgende Code compiliert. Das Beispiel mit PI von oben sollte wie folgt codiert werden:

```
#ifndef PI
#define PI 3.14159265358979
#endif
```

11.8 #else

Syntax: `#else anweisungen`

Dient bei `#if, #ifdef` und `#ifndef` als `else`−Anweisung.

11.9 #endif

Syntax: `#endif`

Beendet `#if, #ifdef` oder `#ifndef`.

11.10 #undef

Syntax: `#undef name_der_definition`

Vorherige Definition mit `#define` wird annulliert.

11.11 #elif

Syntax: `#elif ausdruck anweisungen`

Wenn der Ausdruck von `#if` unwahr und der Ausdruck hinter `#elif` wahr ist, dann wird der, bis zum nächsten `#elif` oder `#else` auftauchende Code compiliert.

11.12 #line

Syntax: `#line nummer`

`line` beeinflußt die Zeilennummerierung des Quelltextes durch den Compiler. Die der Anweisung folgende Zeile wird mit der angegebenen Zeilennummer versehen. Im folgenden wird mit dieser Zeilennummer weitergezählt.

11.13 #pragma

Synatx: `#pragma zeichen_sequenz`

`pragma` ermöglicht eine Einbindung von Anweisungen, die den Compiler in seinem Übersetzungsvorgang beeinflussen. Diese Anweisungen sind von Compiler zu Compiler unterschiedlich.

12.0 Die häufigsten Programmierfehler in C

Die folgende Liste enthält eine Auswahl populärer Fehler der C Programmierung bzw. hilfreiche Hinweise. Häufig sind kaum Ansätze zu erkennen, warum ein Programm sich nicht so verhält, wie es sollte.

Keine Panik, wenn der Compiler viele Fehlermeldungen ausspuckt. Es interessiert zunächst immer nur die erste, alle weiteren können Folgefehler sein. Lesen Sie sich die erste Fehlermeldung aufmerksam durch und korrigieren Sie den Fehler. Starten Sie danach den Compiler erneut und beseitigen Sie dann den nächsten Fehler.

12.1 Formatierung oder wie ein sauber geschriebenes Programm Fehler vermeidet

- Beim Eingeben einer Klammer (egal, ob geschweift, rund oder eckig) immer als nächstes die dazugehörige schließende Klammer eingeben und erst danach die Anweisungen/Ausdrücke zwischen die Klammern setzen. Dadurch ist automatisch sichergestellt, daß zu jeder geöffneten Klammer auch eine geschlossene gehört.

- Hinter eine schliessende geschweifte Klammer einen Kommentar setzen, was überhaupt geschlossen wird. Dies erleichtert die Lesbarkeit bei mehrfach geschachtelten Klammerpaaren.

- Nach Möglichkeit immer Funktionsprototypen verwenden, um re−declaration Probleme zu vermeiden.

12.2 Vermeidung von Nebenwirkungen

- Variablen immer bei der Deklaration oder am Anfang eines Blocks (vor deren Verwendung) initialisieren.

- Zeichenketten immer vor deren Verwendung per Schleife oder memset() initialisieren.

- `printf()` entspricht einer gepufferten Ausgabe. Falls die Ausgabe nicht zu dem gewünschten Zeitpunkt erfolgt, muß diese mit `fflush(stdout);` erzwungen werden.

- Scheint ein Programm Eingabeaufforderungen (z.B. `scanf()` oder `getch()`) zu überspringen, so schafft ein `fflush(stdin);` vor der jeweiligen Anweisung oft für Abhilfe.

- Bei `scanf()` sollte die Feldbreite begrenzt werden.

- Bei `scanf()` sollte beachtet werden, daß Fehleingaben zur Zuweisung eines Nullwertes führen können. Das bedeutet, daß die Variable nach der Eingabe den Wert Null hat.

- Bei `scanf()` in Verbindung mit einfachen Datentypen an den Adreßoperator ()denken!

- Tests auf Buchstaben immer mit dem Zeichen, nie mit dem Code (z.B. `if (zeichen>='a' &zeichen<='z')`) durchführen.

- Abhängigkeiten bei Verwendung der Operatoren ++ und −− vermeiden.
  Vergleiche z.B.

```
for(i=0;i<20;printf("%d\n",block[i]=i++));
```

und

```
for(i=0;i<20;printf("%d\n",block[i++]=i));
```

- Zeichenketten immer mit der binären Null ('\0') abschliessen, damit Systemfunktionen nicht über das Ende der Zeichenkette hinausarbeiten.

12.3 Verwechslungsgefahren

- Einer Konstante des Typs `long int` folgt unmittelbar das Zeichen L (z.B. `25434L`). Im Formatstring von `printf`, `scanf` etc. wird dieser Typ jedoch mit dem Zeichen l (z.B. `%li`) gekennzeichnet.

- `==` bei Vergleichen verwenden (z.B. if a==1), nicht `=`.

- `=` bei Zuweisungen verwenden (z.B. anzahl=0), nicht `==`.

- Ein Semikolon macht einen Ausdruck zu einer Anweisung, ein Semikolon allein ist eine leere Anweisung. Nach if(..) und for(..) folgt i.d.R. kein Semikolon.
  Bei folgendem Beispiel sorgt das Semikolon für einen kompletten Durchlauf der Schleife, bevor der geklammerte Teil bearbeitet wird.

```
for (i=0; i<100; i++);
{
    a[i]=' '
}
```

Bei folgendem Beispiel macht das Semikolon Sinn:

```
for (i=0; i<100; a[i++]=' ');
```

- Konstante Zeichenkette (Strings) in doppelte Hochkomma ("..") einfassen, einzelne Zeichen in einfache Hochkomma ('..') einfassen.

12.4 Häufige Fehler

- Strukturdefinitionen in getrennt compilierten Modulen müssen exakt übereinstimmen, da der Linker nicht mehr überprüfen kann, ob es z.B. bei der Länge eines Strings Abweichungen gibt. Der Code läßt sich zwar compilieren und linken, aber zur Laufzeit kann es zu merkwürdigem Verhalten kommen (z.B. Pointer, die zwar innerhalb einer Funktion noch den korrekten Wert haben, außerhalb der Funktion aber den Wert NULL haben).

- Operationen (=, ==, !=, +, ...) sind mit Zeichenkette nicht möglich.

- Der Ausdruck nach `switch` muss eine ganzzahlige Konstante (z.B. 5, −3, 'a') ergeben.

- Beim Arbeiten mit Feldern immer die Indexgrenzen prüfen. Für `char feld[n];` gilt `0<=index<n`.

12.5 Häufige Compiler− oder Laufzeitfehlermeldungen

- Bricht ein Programm mit einer Exception [9] ab, so ist dies häufig ein Hinweis auf ungültige Zugriffsoperationen bei Zeichenketten bzw. Zeigern. Z.B. Zugriff auf das zehnte Element einer Zeichenkette, die mit `char xyz[5];` deklariert wurde oder %s in einem Formatstring, obwohl ein einzelnes Zeichen oder eine Zahl ausgegeben werden sollte.

    - Der Abbruchcode c005 (OS/2) bzw. c0000005 wird durch eine ungültige Zeigeroperation (s.o.) verursacht. Beispiel:

```
char x[5];

strcpy(x,"Dies ist eine ziemliche lange Zeichenkette");
printf("%s",x);
```

    - Der Abbruchcode c009a (OS/2) bzw. c0000094 wird durch Benutzung eines undefinierten bzw. ungültigen Wertes verursacht. Eine der häufigsten Ursachen ist der Versuch, mit nicht initialisierten float, double oder long double Werten zu rechnen. Beispiel:

```
float zahl;

zahl=zahl+1.0;
```

    - Der Abbruchcode c009b (OS/2) bzw. c0000094 wird durch Benutzung eines undefinierten bzw. ungültigen Wertes verursacht. Eine der häufigsten Ursachen ist der Versuch, mit nicht initialisierten Integer Werten zu rechnen. Eine weitere häufige Fehlerquelle ist die Division durch Null. Beispiel:

```
int eingabe, ergebnis;

scanf("%d", eingabe);   /* hier fehlt der adreßoperator */
ergebnis = eingabe / 111;
```

- Während des Programmlaufes erscheint die Fehlermeldung "The floating−point conversion code is not linked in.".
  Im Programm wird versucht, einen ganzzahligen Wert als Fließkommawert ein− oder auszugeben. Beispiel:

```
int menge=0;

printf ("Menge: %lf\n", menge); // %lf -> double, für int %i oder %d wählen
```

13.0 Neuerung des ISO/IEC 9899:1999 Standards

Im Dezember 1999 tratt der ISO/IEC 9899:1999 Standard in Kraft, der eine Weiterentwicklung der Sprache C darstellt und viele Sachen festschreibt, die diverse Compilerhersteller in der Zwischenzeit in ihre Produkte eingebaut hatten. Einige Compiler wie GCC 3.0 (http://www.gnu.org/software/gcc/gcc−3.0/c99status.html) beinhalten bereits Unterstützung für Teile dieses Standards, aber es wird mit Sicherheit noch einige Zeit dauern, bis alle verbreiteten Compiler diesen Standard voll unterstützen.

Dieses Kapitel stellt einige der Neuerungen vor. Der komplette Standard (ca. 550 Seiten) ist beim ANSI unter http://webstore.ansi.org/ansidocstore/product.asp?sku=ANSI/ISO/IEC+9899−1999 für 18 US Dollar erhältlich.

- Felder mit variabler Länge (variable length array, VLA)

```
int test(int n)
{
  char satz[n+3];
  [...]
}
```

- Felder mit variabler Länge (variable length array, VLA)
  können nur in Funktionsdeklarationen, die keine Defintionen sind, verwendet werden

```
  char satz[*]
```

- Datentyp `long long int` und entsprechende Funktionen
- Kommentare, die mit `//` eingeleitet werden
- Mixtur von Deklarationen und Code
- Datentyp Boolean (`_Bool`), Werte 0 und 1
- komplexe und imaginäre Zahlen
- wide character support (16 bit char) in wchar.h und wctype.h
- universal character names (\u, \U) gemäß ISO/IEC 10646.
- hexadezimale Fließkommakonstanten und entsprechende Formatspezifikationen für printf/scanf (%a und %A).
- zusätzliche Integertypen in <stdint.h> und Funktionen in <inttypes.h>
- macros mit variabler Anzahl Parameter
- snprintf(): sprintf() mit Angabe einer Obergrenze der zu schreibenden Zeichen

14.0 Unterschiede zwischen C und C++

ACHTUNG! Überarbeiten und korrigieren!

C++ stellt eine Erweiterung der Sprache C um die Möglichkeiten der objektorientierten Programmierung dar. Da es sich um eine Erweiterung der Sprache handelt, sind fast alle C−Programme auch mit einem C++−Compiler compilierbar. Des weiteren ist das meiste, was für C gilt auch für C++ gültig. In diesem Kapitel soll ein Überblick über Änderungen und Erweiterungen gegeben werden. Dieses Kapitel dient nicht der Erlernung der C++−Programmierung!

Wie bereits erwähnt, hebt man bei objektorientierten Sprachen die Trennung zwischen Daten und Algorithmen weitestgehend auf. Es wird nicht versucht, daß Problem an die Sprache anzupassen, sondern die Sprache an das Problem. Das Ziel der objektorientierten Programmierung ist die Erstellung von Programmen, die leicht wartbar und erweiterbar sind und deren Code wiederverwendbar ist.

Abgesehen von den neuen Eigenschaften und Fähigkeiten von C++ gibt es einige Unterschiede zwischen C und C++, die beachtet werden müssen, wenn C−Programmteile in C++−Programmen verwendet werden sollen:

1. Variablen können in C++ an jeder beliebigen Stelle und nicht nur zu Beginn eines Blockes deklariert werden.

2. Variablennamen können in C++ beliebig lang sein.

3. Zeichenkonstanten sind vom Typ `char` und nicht, wie in C, vom Typ `int`.

4. Einige Funktionen (wie z.B. `printf()`) sollten durch die C++Äquivalente (z.B. `cout()`) ersetzt werden, welche in der Regel mächtiger sind. Nichtsdestotrotz gelten diese Funktionen nach wie vor.

5. Kommentare können durch zwei Schrägstriche (// Kommentar) eingeleitet werden. Eine Endmarkierung ist nicht notwendig, da diese Kommentare nur bis zum Zeilenende gehen.

6. Die Datei `iostream.h` stellt eine Ersatzmöglichkeit für die Datei `stdio.h` dar. Sie enthält z.B. die Funktion `cout()`, die anstatt von `printf()` benutzt werden kann.

C++ enthält einige neue Eigenschaften, die die Sprache flexibler machen. Einige dieser Erweiterungen werden in den folgenden Abschnitten vorgestellt.

14.1 Operatorüberladung

Operatoren können unter C++ neue Bedeutungen zugewiesen werden. Welche der Bedeutungen jeweils gültig ist, wird aus dem Zusammenhang deutlich. So erhält z.B. der links−shift−Operator (<<) in der Funktion `cout()` die Eigenschaft, Daten auf die Ausgabeeinheit zu leiten. Beispiel:

```
cout << "\nHeutiger Tag:" << tag;
```

14.2 Funktionsüberladung

In einem Programm dürfen mehrere Funktionen mit dem gleichen Namen vorkommen. Diese Eigenschaft wird auch als *Polymorphismus* bezeichnet. Der Compiler untersucht beim jeweiligen Funktionsaufruf nicht nur den Namen sondern auch die Argumentenliste und den Zugehörigkeitsbereich (klassenbezogen) und entscheidet dann, welche Funktion aufgerufen wird.

14.3 Klassen

Klassen sind zentrale Elemente der objektorientierten Programmierung. Sie stellen eine Art Struktur dar, beinhalten jedoch im Gegensatz zu `struct` nicht nur die Datenelemente der Struktur, sondern zusätzlich die Funktionen, die auf diese Datenelemente zugreifen. Des weiteren ist es möglich, die Daten und die Funktionen, die auf die Daten zugreifen, vor dem Rest des Programms zu verbergen. Hierbei handelt es sich um `private`−Daten und − Funktionen. Es kann für sämtliche Elemente der Klasse festgelegt werden, ob sie `private` oder `public` sein sollen, wobei alle Komponenten standardmäßig `private` sind.

Die Definition einer Klasse beinhaltet somit folgende Elemente:

1. das Schlüsselwort `class`,
2. den Namen der Klasse,
3. die Datenelemente der Klasse,
4. die Funktionen der Klasse und
5. Informationen darüber, welche Elemente und Methoden `public` oder `private` sind. (Gekennzeichnet durch die entsprechenden Schlüsselwörter)

Die Funktionen werden in der Regel als Prototypen in die Klasse eingebunden und außerhalb der Klasse definiert. Die komplette Funktionsdefinition läßt sich zwar in eine Klassendefinition einbetten, dies wird jedoch nur bei sehr kurzen Funktionen realisiert.

Werden Klassenfunktionen außerhalb einer Klasse definiert, so wird durch den Namen der Klasse und zwei Doppelpunkte vor dem Funktionsnamen signalisiert, daß diese Funktionen zu einer bestimmten Klasse gehören. (Beispiel siehe unten)

Diese Klassenfunktionen werden auch als *Methoden* bezeichnet.

Beispiel für die Definition einer Klasse:

```
// Definition einer Klasse

class personen
{
   private:
      int nummer;
      char nachname[40];
      char vorname[40];
      float postleitzahl;
      char wohnort[40];
      void gehalt_ermitteln();
   public:
      void erfassen();
      void anzeigen();
};

void personen::erfassen()
{
   ...Code...
}
```

`private` und `public` können in einer Klassendefinition beliebig oft erscheinen; die obige Aufteilung hat sich jedoch durchgesetzt. Des weiteren werden die Daten einer Klasse in der Regel `private` deklariert und

die Methoden `public`. Im obigen Beispiel ist die Methode `gehalt_ermitteln` auch `private` definiert, da sie nach außenhin nicht sichtbar sein soll.

14.4 Konstruktor und Destruktor

Um Objekte einer Klasse bei ihrer Erzeugung zu initialisieren, wird ein Konstruktor benutzt. Er hat den gleichen Namen wie die Klasse und kann bei der Definition der Klasse angegeben werden.

Sobald ein Objekt aus dem Speicher entfernt wird, wird der Destruktor aufgerufen. Sein Name entspricht dem Namen der Klasse mit einer vorangehenden Tilde (~).

Fortgeschrittene Themen in C

Die folgenden Kapitel beschäftigen sich mit fortgeschrittenen Themen wie z.B. Netzwerkprogrammierung in C.

15.0 Speicherverwaltung

Die Speicherverwaltungsfunktionen kommen z.B. zum Einsatz, wenn die Größe des benötigten Speichers unbekannt ist. Speicher für Variablen und Konstanten wird bereits vom Compiler reserviert und ist dadurch statisch und in seiner Größe vorgegeben. Im folgenden wird eine Auswahl der zur Verfügung stehenden Funktionen vorgestellt.

15.1 Speicherplatz reservieren

Syntax:

```
void *malloc(size_t size);
```

Die Funktion `malloc` reserviert einen Block angegebener Größe im Speicher. Konnte Speicher reserviert werden, liefert `malloc` einen Zeiger auf den angelegten Speicherbereich zurück. Konnte kein Speicher reserviert werden oder wurde als Größe 0 Bytes angegeben, wird NULL zurückgegeben.

Beispiel 1:

```
/* Speicherplatz mit malloc reservieren */
#include <stdio.h>
#include <malloc.h>

void main(void)
{
  char *block=NULL; /* Zeiger auf Zeichen deklarieren */
  int i;

  /* Speicherbereich mit 4000 Bytes reservieren und */
  /* zurückgegebenen Zeiger in Zeiger auf Zeichen   */
  /* umwandeln                                      */
  block=(char *)malloc(4000);
  if(block!=NULL)
  {
    for(i=0;i<4000;block[i++]=' ');
    printf("Speicherblock angelegt und initialisiert.");
  }
  else
  {
    printf("Speicherblock konnte nicht angelegt werden.");
  }
}
```

Beispiel 2:

```
/* Speicherplatz mit malloc reservieren */
#include <stdio.h>
#include <malloc.h>

void main(void)
{
  int *block=NULL; /* Zeiger auf Integer deklarieren */
  int i;
```

```
  /* Speicherbereich mit 20*32 Bits reservieren und */
  /* zurückgegebenen Zeiger in Zeiger auf Integer   */
  /* umwandeln                                      */
  block=(int *)malloc(20*sizeof(int));
  if(block!=NULL)
  {
    for(i=0;i<20;printf("%d\n",(block[i++]=i)));
    printf("Speicherblock angelegt und initialisiert.");
  }
  else
  {
    printf("Speicherblock konnte nicht angelegt werden.");
  }
}
```

15.2 Speicherplatz reservieren und initialisieren

Syntax:

```
void *calloc(size_t num, size_t size);
```

Die Funktion `calloc` legt einen Speicherbereich an und initialisiert diesen mit 0. Ansonsten verhält sich `calloc` wie `malloc`.

Beispiel:

```
/* Speicherplatz mit calloc reservieren */
#include <stdio.h>
#include <malloc.h>

void main(void)
{
  int *block=NULL; /* Zeiger auf Integer deklarieren */
  int i;

  /* Speicherbereich mit 20*32 Bits reservieren und */
  /* zurückgegebenen Zeiger in Zeiger auf Integer   */
  /* umwandeln                                      */
  block=(int *)calloc(20,sizeof(int));
  if(block!=NULL)
  {
    for(i=0;i<20;printf("Inhalt von %d : %d\n",i,block[i++]));
    printf("Speicherblock angelegt und initialisiert.");
  }
  else
  {
    printf("Speicherblock konnte nicht angelegt werden.");
  }
}
```

15.3 Speicherplatzgröße verändern

Syntax:

```
void *realloc(void *ptr, size_t size);
```

Die Funktion `realloc` ändert die Größe eines bereits angelegten Speicherbereichs. Der Inhalt des alten (bei Vergrößerung) bzw. übrigbleibenden (bei Verkleinerung) Speicherbereichs bleibt erhalten. Die Funktion `realloc` liefert wie `malloc` einen Zeiger auf den Speicherbereich zurück.

Beispiel:

```
/* Speicherplatz mit calloc reservieren und mit realloc verändern */
#include <stdio.h>
#include <malloc.h>

void main(void)
{
  int *block=NULL; /* Zeiger auf Integer deklarieren */
  int i;

  /* Speicherbereich mit 20*32 Bits reservieren und */
  /* zurückgegebenen Zeiger in Zeiger auf Integer   */
  /* umwandeln                                      */
  block=(int *)calloc(20,sizeof(int));
  if(block!=NULL)
  {
    for(i=0;i<20;printf("Inhalt von %d : %d\n",i,block[i++]));
    printf("Speicherblock angelegt und initialisiert.\n");
  }
  else
    printf("Speicherblock konnte nicht angelegt werden.");

  for(i=0;i<20;block[i++]=i);

  /* Speicherblock vergrößern */
  block=realloc(block,40*sizeof(int));
  if(block!=NULL)
  {
    for(i=0;i<40;printf("Inhalt von %d : %d\n",i,block[i++]));
    printf("Speicherblock geändert.\n");
  }

  /* Speicherblock verkleinern */
  block=realloc(block,10*sizeof(int));
  if(block!=NULL)
  {
    for(i=0;i<10;printf("Inhalt von %d : %d\n",i,block[i++]));
    printf("Speicherblock geändert.");
  }
}
```

15.4 Speicherblock freigeben

Wird ein reservierter Speicherblock nicht mehr benötigt, sollte dieser mit der Funktion `free` freigegeben werden, um nicht unnötig Speicherplatz zu blockieren.

Syntax:

```
void free(void *ptr);
```

Beispiel:

```
/* Speicherplatz mit malloc reservieren und mit free freigeben */
#include <stdio.h>
#include <malloc.h>

void main(void)
{
  char *block=NULL; /* Zeiger auf Zeichen deklarieren */
  int i;

  /* Speicherbereich mit 4000 Bytes reservieren und */
  /* zurückgegebenen Zeiger in Zeiger auf Zeichen   */
  /* umwandeln                                      */
  block=(char *)malloc(4000);
  if(block!=NULL)
  {
    for(i=0;i<4000;block[i++]=' ');
    printf("Speicherblock angelegt und initialisiert.");
  }
  else
  {
    printf("Speicherblock konnte nicht angelegt werden.");
  }

  /* speicherplatz freigeben */
  free(block);
}
```

15.5 Verkettete Listen

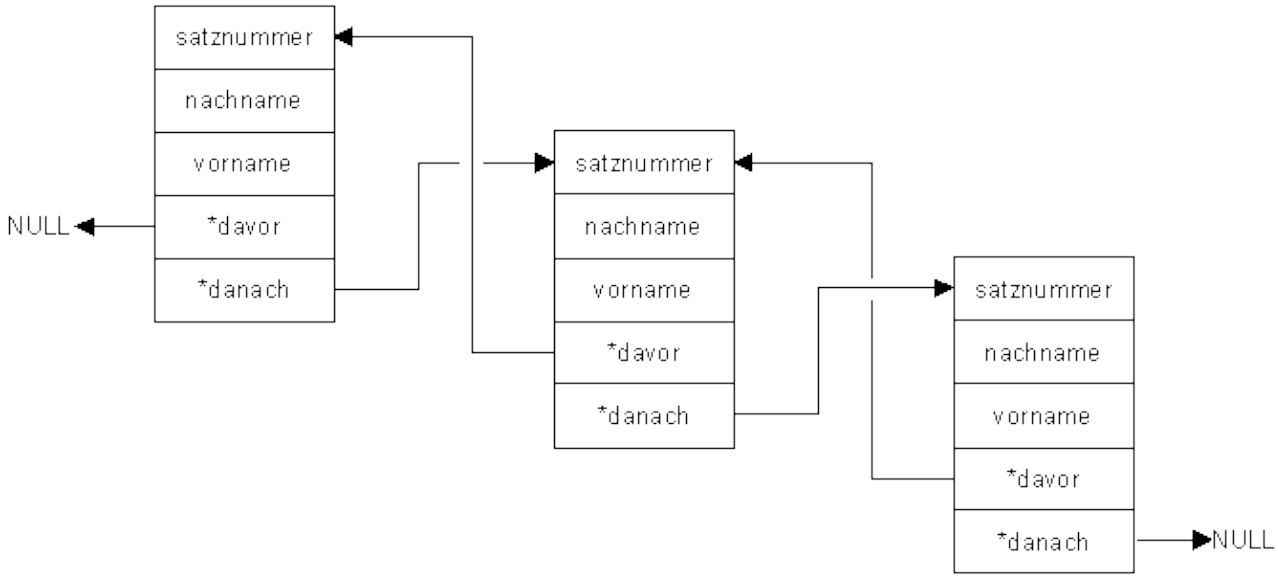
ACHTUNG! In den Beispielen wird mit globalen Variablen gearbeitet!

Mittels Zeiger und Speicherverwaltungsfunktionen lassen sich effektiv Listen aufbauen, ohne vorher zu wissen, wieviel Listenelemente benötigt werden. Des weiteren lassen sich flexibel Listenelemente einfügen und löschen.

Eine Liste besteht aus einer beliebigen Anzahl fest definierter Elemente. Diese Elemente sind Speicherbereiche, die i.d.R. als Strukturen mit einer festen Anzahl an Strukturelementen betrachtet werden. Eine derartige Liste kann z.B. zur Verwaltung von Studenten benutzt werden. Um eine Verbindung der einzelnen Listenelemente untereinander gewährleisten zu können, werden diese über Zeiger miteinander verbunden. Hierbei unterscheidet man einfach und doppelt verkettete Listen. Einfach verkettete Listen enthalten in der Struktur nur einen Zeiger, der auf das nachfolgende Listenelement zeigt (also ein Zeiger auf eine Struktur). Eine doppelt verkettete Liste enthält in der Struktur zusätzlich einen Zeiger, der auf das

vorherige Listenelement zeigt.

Abbildung 21. Doppelt verkettete Liste



Da die doppelt verkettete Liste wesentlich flexibler ist als die einfach verkettete Liste, wird auf die doppelt verkettete Liste ausführlich eingegangen und die einfach verkettete Liste lediglich der Vollständigkeit halber vorgestellt.

15.5.1 Doppelt verkettete Liste

In diesem Abschnitt wird vorgestellt, wie eine doppelt verkettete Liste erzeugt, Listenelemente hinzugefügt und gelöscht werden.

15.5.1.1 Doppelt verkettete Liste erzeugen

Der Vorgang zur Erzeugung einer doppelt verketteten Liste gliedert sich wie folgt:

1. Vorbereitungen
    a. Mittels `typedef` einen Strukturtyp definieren, der die Daten eines Listenelements aufnimmt.
    b. Eine Variable dieses Strukturtyps deklarieren.
    c. Mehrere Pointer dieses Strukturtyps deklarieren:
        ◊ Der Pointer `start` dient dazu, die Adresse des ersten Listenelements aufzunehmen.
        ◊ Der Pointer `ende` dient dazu, die Adresse des letzten Listenelements aufzunehmen.
        ◊ Der Pointer `momentan` dient dazu, die Adresse des aktuellen Listenelements aufzunehmen.
        ◊ Der Pointer `zwischen` dient dazu, die Adresse des vorherigen Listenelements aufzunehmen.
    d. Deklarierte Pointer mit NULL initialisieren.
2. Erstes Listenelement erzeugen
    a. Speicher mittels `malloc` reservieren (momentan=...)
    b. Strukturelemente mit Nutzdaten füllen.

c. Zeiger auf vorheriges Listenelement in der Struktur auf NULL setzen.
d. Zeiger `start` mit der Adresse des momentanen Speicherblocks füllen.
e. Zeiger `zwischen` mit der Adresse des momentanen Speicherblocks füllen.

3. Weitere Listenelemente erzeugen
   a. Speicher mittels `malloc` reservieren
   b. Strukturelemente mit Nutzdaten füllen.
   c. Zeiger auf vorheriges Listenelement in der Struktur mit dem Wert des Zeigers `zwischen` füllen.
   d. Zeiger auf vorheriges Listenelement in der Struktur, auf die der Zeiger `zwischen` zeigt, mit der Adresse des momentanen Speicherblocks füllen.
   e. Zeiger `zwischen` mit der Adresse des momentanen Speicherblocks füllen.
4. Letztes Listenelement erzeugen
   a. Vorgang wie oben, aber momentan−>danach=NULL

```
/* Doppelt verkettete Liste anlegen und benutzen */
#include <stdio.h>
#include <malloc.h>

/* Strukturtyp für Listenelement definieren und */
/* Variable DATEI deklarieren                   */
typedef struct datei
{
  int satznummer;
  char nachname[20+1];
  char vorname[20+1];
  struct datei *davor;
  struct datei *danach;
} DATEI;

DATEI *momentan, *zwischen, *start, *ende;

void speicher_anfordern(void);
void liste_fuellen(int);
void liste_ausgeben(void);

void main(void)
{
  int i;

  zwischen=NULL;                              /* Zeiger auf vorheriges Element */
  for(i=1;i<=3;i++)
  {
     speicher_anfordern();
     liste_fuellen(i);
     momentan->davor=zwischen;                /* Zeiger auf vorheriges Element */
     if(zwischen) zwischen->danach=momentan; /* zwischen<>NULL, ab 2.Element  */
     else start=momentan;                     /* zwischen==NULL, nur 1.Element */
     zwischen=momentan;                       /* momentane Adresse für Zeiger  */
                                              /* auf vorheriges Element retten */
  }
  momentan->danach=NULL;                      /* Zeiger auf nachfolgendes      */
                                          /* Element bei letztem Element   */
  ende=momentan;                              /* Zeiger auf letztes Element    */
  liste_ausgeben();
}

void speicher_anfordern(void)
{
   /* Speicher anfordern und Zeiger per Typecast ändern */
   if((momentan=(DATEI *)malloc(sizeof(DATEI))) == NULL)
   {
```

```
      printf("\nKein Speicherplatz zur Verfügung!\n");
      exit(8);
   }
   return;
}

void liste_fuellen(i)
{
   int t;
   /* Strukturelemente mit Nutzdaten füllen */
   momentan->satznummer=i;
   strcpy(momentan->nachname,"Nachname ");
   for(t=1;t<=i;t++)
   {
      strcat(momentan->nachname,"x");
   }
   strcpy(momentan->vorname,"Vorname");
   return;
}

void liste_ausgeben(void)
{
   printf("\nListe vorwärts ausgeben\n");
   /* Zeiger auf erstem Element positionieren */
   momentan=start;
   /* Solange es Elemente gibt, also bis momentan==NULL */
   while(momentan)
   {
      printf("\nSatz %d\nNachname %s\nVorname %s\n",
             momentan->satznummer,momentan->nachname,
             momentan->vorname);
      /* Zeiger auf nächstem Element positionieren */
      momentan=momentan->danach;
   }
   printf("\nListe rückwärts ausgeben\n");
   /* Zeiger auf letztem Element positionieren */
   momentan=ende;
   while(momentan)
   {
      printf("\nSatz %d\nNachname %s\nVorname %s\n",
             momentan->satznummer,momentan->nachname,
             momentan->vorname);
      /* Zeiger auf vorherigem Element positionieren */
      momentan=momentan->davor;
   }
   return;
}
```

15.5.1.2 Listenelement einfügen

Das Einfügen eines Listenelementes wird durch Änderung der Zeiger des vorhergehenden und des nachfolgenden Listenelementes erreicht. Die Zeiger `danach` des vorhergehenden Listenelementes und `davor` des nachfolgenden Listenelementes müssen auf das einzufügende Element zeigen. Der Zeiger `davor` des einzufügenden Elementes muß auf das vorhergehende Listenelement zeigen und der Zeiger `danach` auf das nachfolgende Listenelement.

Der folgende Beispielcode enthält nur die neue Funktion `liste_einfuegen(void)` und einen Beispielaufruf. Sämtliche mit ... angegeben Stellen sind mit dem vorhergehenden Quelltext identisch.

Hinweis

Dieser Beispielcode ist nicht auf alle möglichen Konstellationen ausgelegt!

```
. . .
void liste_einfuegen(void);
. . .
void main(void)
{
   . . .
   ende=momentan;
   liste_ausgeben();

   /* Neues Listenelement einfügen */
   zwischen=start->danach; /* zeiger vorbereiten, um element zwischen erstem
                              und zweitem element einzufügen */
   liste_einfuegen();
   printf("\n\nListe nach dem Einfügen eines Elements an zweiter Position:\n");
   liste_ausgeben();
}
 . . .
void liste_einfuegen(void)
{
   DATEI *hilfszeiger;

   /* 1.) Speicher für das neue Listenelement anfordern */
   speicher_anfordern();         /* momentan erhaelt adresse des neuen elements */

   /* 2.) Struktur mit Nutzdaten füllen */
   liste_fuellen(10);

   /* 3.) Zeiger aller drei Element 'korrigieren' */
   hilfszeiger=zwischen->davor;  /* zeiger auf vorheriges element */
   momentan->davor=hilfszeiger;  /* neues element auf vorheriges verweisen lassen */
   hilfszeiger->danach=momentan; /* vorheriges element auf neues verweisen lassen */
   momentan->danach=zwischen;    /* neues element auf nachfolgendes verweisen lassen */
   zwischen->davor=momentan;     /* nachfolgendes auf neues verweisen lassen */

   return;
}
 . . .
```

15.5.1.3 Listenelement löschen

Der folgende Beispielcode enthält nur die neue Funktion `liste_einfuegen(void)` und einen Beispielaufruf. Sämtliche mit ... angegeben Stellen sind mit dem vorhergehenden Quelltext identisch.

Hinweis

Dieser Beispielcode ist nicht auf alle möglichen Konstellationen ausgelegt!

```
 . . .
void liste_loeschen(void);
 . . .
void main(void)
{
   . . .
   ende=momentan;
   liste_ausgeben();

   /* Neues Listenelement einfügen */
   zwischen=start->danach; /* zeiger vorbereiten, um element zwischen erstem
                              und zweitem element einzufügen */
```

```
    liste_einfuegen();
    printf("\n\nListe nach dem Einfügen eines Elements an zweiter Position:\n");
    liste_ausgeben();

    zwischen=start->danach; /* zeiger vorbereiten, um element zwischen erstem
                               und zweitem element einzufügen */
    liste_loeschen();
    printf("\n\nListe nach dem Löschen eines Elements an zweiter Position:\n");
    liste_ausgeben();
}
 . . .
void liste_loeschen(void)
{
    DATEI *hilfszeiger;

    hilfszeiger=zwischen->davor; /* zeiger auf element vor dem zu löschenden
                                    element */
    hilfszeiger->danach=zwischen->danach;

    hilfszeiger=zwischen->danach;
    hilfszeiger->davor=zwischen->davor;
    return;
}
```

15.5.2 Einfach verkettete Liste

Im Gegensatz zur doppelt verketteten Liste wird bei der einfach verketteten Liste nur die Adresse zum nachfolgenden Element gesichert.

```
/* Einfach verkettete Liste anlegen und benutzen */
#include <stdio.h>
#include <malloc.h>

typedef struct datei
{
  int satznummer;
  char nachname[20+1];
  char vorname[20+1];
  struct datei *danach;
} DATEI;

DATEI *momentan, *zwischen, *start;

void speicher_anfordern(void);
void liste_fuellen(int);
void liste_ausgeben(void);

void main(void)
{
  int i=0;

  zwischen=NULL;
  for(i=1;i<=3;i++)
  {
     speicher_anfordern();
     liste_fuellen(i);
     if(zwischen) zwischen->danach=momentan; /* zwischen<>NULL, ab 2.Element  */
     else start=momentan;                    /* zwischen==NULL, nur 1.Element */
     zwischen=momentan;
```

```
    }
    momentan->danach=NULL;
    liste_ausgeben();
}

void speicher_anfordern(void)
{
   if((momentan=(DATEI *)malloc(sizeof(DATEI))) == NULL)
   {
      printf("\nKein Speicherplatz zur Verfügung!\n");
      exit(8);
   }
   return;
}

void liste_fuellen(i)
{
   int t=0;
   momentan->satznummer=i;
   strcpy(momentan->nachname,"Nachname ");
   for(t=1;t<=i;t++)
      strcat(momentan->nachname,"x");
   strcpy(momentan->vorname,"Vorname");
   return;
}

void liste_ausgeben(void)
{
   momentan=start;
   while(momentan)
   {
      printf("\nSatz %d\nNachname %s\nVorname %s\n",
             momentan->satznummer,momentan->nachname,
             momentan->vorname);
      momentan=momentan->danach;
   }
   return;
}
```

15.6 Speicherverwaltungsfunktionen für Debugging

Die folgenden Funktionen stellen einen Ersatz für malloc() und free() dar:

```
void *Malloc(size_t size);
void Free(char **ptr);
```

Wie anhand der Funktionsprototypen zu sehen ist, benötigt der Aufruf von Free() einen anderen Parameter als free().

```
free(puffer);
```

muß nun folgendermaßen aussehen:

```
Free( (char **) );
```

Die neuen Funktionen Malloc() und Free() geben folgende Funktionen bei ihrem Aufruf aus:

- Die Größe des Speicherblocks.
- Die Adresse des Speicherblocks.
- Den Namen der Source Code Datei, aus der die Funktion aufgerufen wurde.
- Die Zeile der Source Code Datei, aus der die Funktion aufgerufen wurde.

Die Funktionen werden jedoch nur dann benutzt, wenn das Symbol DEBUG_MEMORY per #define DEBUG_MEMORY definiert wurde. Anderenfalls werden alle Aufrufe von Malloc() und Free() durch die Aufrufe von malloc() und free() ersetzt.

Source Code für Malloc():

```
/*-------------------------------------------------------------------------*/
/* Malloc()                                                                */
/*-------------------------------------------------------------------------*/
#if DEBUG_MEMORY
 void *Malloc(size_t size, char *srcFile, int srcLine)
 {
   void *ptr;

   ptr=malloc( size );
   printf("Allocating %06d bytes of memory at %d in %s at line %05d\n",
          size, ptr, srcFile, srcLine);
   return ptr;
 }




#define Malloc( val ) Malloc( val, __FILE__, __LINE__)
#else
#   define Malloc(val) malloc(val)
#endif
/*--- End - Debugversion malloc() ---*/
```

Source Code für Free():

```
/*-------------------------------------------------------------------------*/
/* Free()                                                                  */
/*-------------------------------------------------------------------------*/
#ifdef DEBUG_MEMORY
 void Free(char **ptr, char *srcFile, int srcLine)
 {
   printf("Releasing  %06d bytes of memory at %d in %s at line %05d ",
           _msize(*ptr), *ptr, srcFile, srcLine);
   if( *ptr != NULL )
   {
      free( *ptr );
      *ptr = NULL;
   }
   printf("-> %d\n",*ptr);
 }
#define Free( val ) Free( val, __FILE__, __LINE__);
#else
#   define Free(val) free(* val);
#endif
```

```
    /*--- End Debugversion free() ---*/
```

16.0 Verbindungen zum Betriebssystem

Von Zeit zu Zeit ist es notwendig, von einem Programm aus ein anderes Programm zu starten, Betriebssystembefehle auszuführen oder dem Programm beim Aufruf von der Kommandozeile aus Daten (sog. *Kommandozeilenparameter*) mitzugeben.

In diesem Kapitel werden die dazu benötigten Funktionen vorgestellt und anhand von Beispielen verdeutlicht, wie unter C die Verbindung zum Betriebssystem hergestellt werden kann. Je nach verwendetem Betriebssystem bzw. verwendeter Entwicklungsumgebung werden weitere Funktionen zur Verfügung gestellt, wie z.B. `DosExecPgm` unter OS/2.

16.1 Kommandozeilenparameter auswerten

Eingangs wurde bereits erwähnt, daß die Funktion `main()` die einzige Funktion ist, die Kommandozeilenparameter empfangen kann. Kommandozeilenparameter sind Argumente, die beim Programmstart durch Leerzeichen voneinander getrennt hinter dem Programmnamen angegeben werden.

Beispiel:

```
DISKCOPY A: B:
```

In diesem Fall wird das Programm `DISKCOPY` aufgerufen, dem als Kommandozeilenparameter `A:` und `B:` mitgegeben werden. Solche Parameter werden in C der Funktion `main()` übergeben.

Die Funktionsweise soll anhand eines einfachen Beispiels erläutert werden:

```
/* KOMPARA.C     liest Kommandozeilenparameter ein und gibt */
/*               sie aus                                    */
#include <stdio.h>

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

   if (argc < 2)                  /* Parameter vorhanden ? */
   {
      printf("Es wurden keine Kommandozeilenparameter angegeben!\n");
      exit(4);
   }

   for (zaehler=1; zaehler<argc; zaehler++)
   {
       printf("\nParameter %d = %s", zaehler, argv[zaehler]);
   }
}
```

Die Funktion `main()` erhält vom Betriebssystem zwei Parameter: `argc` und `argv[]`. `argc` enthält die Anzahl der an das Programm übergebenen Parameter. `argv[]` enthält die eigentlichen Parameter. Enthält `argc` den Wert 1, so bedeutet dies, daß kein Parameter übergeben wurde, da grundsätzlich der Programmname als erster Parameter an das Programm übergeben wird. `argv[0]` enthält somit den Namen des Programms. Zusätzlich wird noch der Parameter `envp` übergeben, der Daten des sog. Environments beinhalten kann.

Die Namen der Variablen sind frei wählbar, allerdings stellen die oben angegeben Bezeichner einen Quasi−Standard dar.

Die Deklaration `char *argv[]` bedeutet: Zeiger auf Zeiger auf Zeichen. Diese Deklaration ist äquivalent zu `char **argv`. `argv` ist ein Zeiger, der auf ein Feld zeigt, welches wiederum Zeiger enthält, die ihrerseits auf die einzelnen Kommandozeilenparameter zeigen. Durch die leeren eckigen Klammern wird definiert, daß das Feld eine unbekannte Anzahl an Elementen hat.

16.2 Betriebssystembefehle ausführen

Damit Betriebssystembefehle von einem C−Programm aus aufgerufen werden können, verwendet C die Funktion `system`. Diese Funktion steht sowohl unter DOS als auch unter UNIX zur Verfügung und leitet sämtliche Systembefehle, die in der Kommandozeile des Betriebssystems eingegeben werden können, an das Betriebssystem weiter. Als Argument wird der Funktion die Kommandozeile in doppelten Hochkommata (") übergeben.

Beispiel:

```
resultat = system("dir c: > dirlist.txt");
```

Die Funktion übergibt den Rückgabewert des Systembefehls (erfolgreich==0, Fehler <>0) an die Variable *resultat*, die daraufhin ausgewertet werden kann.

16.3 Programme starten

Um andere Programme starten zu können, verwendet C in DOS−Umgebungen die Funktionen `spawn()` und `exec()`. Die beiden Funktionen unterscheiden sich nur in einem Punkt: bei der Verwendung von `spawn()` bleibt das aufrufende Programm im Speicher, während das aufgerufene (child−Prozess) ausgeführt wird. Nach Beendigung des aufgerufenen Programms wird die Kontrolle an das aufrufende Programm zurückgegeben. Bei `exec()` hingegen wird das aufrufende Programm im Speicher überlagert und nach Beendigung des Child−Prozesses auf die Betriebssystemebene zurückgekehrt. Da `spawn()` mit Hilfe eines Schalters dazu gebracht werden kann, sich wie `exec()` zu verhalten, wird in diesem Zusammenhang nur auf `spawn()` eingegangen.

Um eine vorher bekannte Anzahl von Parametern zu übergeben, wird die Funktion `spawnl()` verwendet. Steht die Anzahl der Parameter nicht fest, so ist die Funktion `spawnv()` zu benutzen. Im folgenden werden die einzelnen Varianten mit deren Syntax aufgeführt.

Die Funktion `spawnl()` existiert in den folgenden Varianten:

```
int spawnl(int mode, char *path, char *arg(),..., NULL);
int spawnlp(int mode, char *path, char *arg(),..., NULL);
int spawnle(int mode, char *path, char *arg(),..., NULL, char *envp[]);
int spawnlpe(int mode, char *path, char *arg(),..., NULL, char *envp[]);
```

Auf die Bedeutung der einzelnen Funktionsnamenszusätze und Argumente wird weiter unten eingegangen.

Die Funktion `spawnv()` existiert in den folgenden Varianten:

```
int spawnv(int mode, char *path, char *argv[]);
int spawnvp(int mode, char *path, char *argv[]);
int spawnve(int mode, char *path, char *argv[], char *envp[]);
int spawnvpe(int mode, char *path, char *argv[], char *envp[]);
```

Die Zusätze am Funktionsnamen haben die folgenden Bedeutungen:

Zeichen	Bedeutung
p	Die `spawn()`−Funktion sucht nach dem aufzurufenden Programm nicht nur im momentanen Verzeichnis, sondern in sämtlichen, im Betriebssystem mit `PATH` festgelegten Verzeichnissen.
e	ermöglicht die Übergabe eines eigenen Environments (eigene Umgebungsvariablen für das aufzurufende Programm.

Die folgende Aufstellung beschreibt die Parameter, die mitgegeben werden:

`mode` legt fest, wie sich das aufrufende Programm während der Ausführung des aufzurufenden Programmes verhält. Folgende Angaben sind möglich:

1. `P_WAIT`: das aufrufende Programm bleibt im Speicher, wird aber während der Ausführung des aufzurufenden Programmes stillgelegt.
2. `P_NOWAIT`: das aufrufende Programm und das aufzurufende Programm werden konkurrierend ausgeführt. Dieser Modus ist nur in Multitaskingumgebungen verfügbar.
3. `P_OVERLAY`: das aufrufende Programm wird vom aufzurufenden Programm überlagert und nach dessen Beendigung nicht weitergeführt. Entspricht der Funktion `exec()`.

`path` gibt den Namen des aufzurufenden Programmes an.

`argv` beinhaltet die Argumente, die an das aufzurufende Programm übergeben werden. `argv[0]` sollte den Namen des aufzurufenden Programmes beinhalten. Die weiteren Argumente sind wahlfrei.

`NULL` ist obligatorisch und schließt die Argumentenliste ab.

`envp` ist ein Zeigerfeld, dessen Inhalt übergeben wird, um ein Environment an das aufzurufende Programm übergeben zu können. Das höchste Element des eigentlichen Umgebungsfeldes muß `NULL` enthalten.

Beispiele:

Variante	Bedeutung
spawnl(P_WAIT,"c:\\os2\\e.exe","c:\\os2\\e.exe", "autoexec.bat",NULL);	Das aufzurufende Programm `e.exe` wird geladen und gestartet. Durch `P_WAIT` verbleibt das aufrufende Programm im Speicher. Der Parameter `autoexec.bat` wird an den Editor `e.exe` übergeben. Die Wiederholung der Angabe `e.exe` ist notwendig, da unter DOS der erste Parameter standardmäßig der Programmname des aufzurufenden Programmes sein sollte. Das Ende der Parameterliste wird durch `NULL`

	gekennzeichnet.
spawnlp(P_WAIT,"e.exe","e.exe","autoexec.bat",NULL);	Wie oben, nur daß im aktuellen Verzeichnis und gemäß der PATH−Angabe nach dem aufzurufenden Programm e.exe gesucht wird.
spawnlpe(P_OVERLAY,"e.exe","e.exe","autoexec.bat", NULL, environment);	Wie oben, nur daß im aktuellen Verzeichnis und gemäß der PATH−Angabe nach dem aufzurufenden Programm gesucht wird. Zusätzlich werden die Angaben des Zeigers environment an das Betriebssystem übergeben. environment muß als Zeigerfeld vom Typ char *environment[]; deklariert sein. Die Angabe P_OVERLAY bewirkt, daß spawn() sich wie exec() verhält.

Aufgrund der Komplexität der Erläuterungen soll noch ein Programmbeispiel angefügt werden:

```
/* SPAWN.C                                                    */
/* Dieses Programm demonstriert die Verwendungsweise der Funktion */
/* spawn() anhand der Variante spawnlp().                     */

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

/* Festlegung des Environments für das aufzurufende Programm */
char *environment[]=
{
    "TEMP=C:\\",
    "PATH=C:\\OS2;C:\\OS2\\APPS",
    NULL
};


void main(void)
{
    /* Deklaration einer lokalen Return-Code-Variablen */
    int res;

    /* Aufruf von E.EXE */
    res = spwanlp(P_WAIT,"e.exe","e.exe","autoexec.bat", NULL,
                  environment);

    /* Aufruf erfolgreich? */
    if(res)
    {
       printf("Der Prozeß konnte nicht gestartet werden!");
    }
    else
    {
```

```
        printf("spwanlp() beendet!");
    }
}
```

Das oben abgebildete Beispielprogramm ruft den Editor `e.exe` auf und gibt ihm ein eigenes Environment und Kommandozeilenparameter mit. Nach dem Aufruf wird anhand der Variablen `res`, die das Resultat des Aufrufes entgegennimmt, überprüft, ob der Aufruf erfolgreich war.

Der folgende Programmcode demonstriert die Verwendung einiger spawn Funktionen und wurde der CP Reference (Teil der Dokumentation der IBM C/C++ FirstStep Tools Version 2.01) entnommen.

ACHTUNG! Beispiel ist unsauber programmiert.

```
/*********************************************************************

This example shows calls to four of the eight _spawn routines.  When
called without arguments from the command line, the program first
runs the code for case PARENT.  It spawns a copy of itself, waits for
its child to run, then spawns a second child.  The instructions for the
child are blocked to run only if argv[0] and one parameter were
passed (case CHILD).  In its turn, each child spawns a grandchild as a
copy of the same program.  The grandchild instructions are blocked by
the existence of two passed parameters. The grandchild is permitted to
overlay the child.  Each of the processes prints a message identifying
itself.

*********************************************************************/

#include  <stdio.h>
#include  <process.h>
#define   PARENT        1
#define   CHILD         2
#define   GRANDCHILD    3

int main(int argc, char **argv, char **envp)
{
   int    result;
   char   *args[4];

   switch(argc)
   {
      case PARENT:
      /* no argument was passed:  spawn child and wait */
         result = _spawnle(P_WAIT, argv[0],
                           argv[0], "one", NULL, envp);
         if (result)
             abort();
         args[0] = argv[0];
         args[1] = "two";
         args[2] = NULL;

         /* spawn another child, and wait for it */
         result = _spawnve(P_WAIT, argv[0],
                     args, envp);
         if (result)
             abort();
         printf("Parent process ended. %d\n",result);
         _exit(0);

      case CHILD:
         /* one argument passed:  allow grandchild to overlay */
         printf("child process %s began\n", argv[1]);
```

```
            if (*argv[1] == 'o')            /* child one? */
            {
               _spawnl(P_OVERLAY, argv[0],
                           argv[0], "one", "two", NULL);
               abort();
               /* not executed because child was overlaid */
            }
            if (*argv[1] == 't')            /* child two? */
            {
                args[0] = argv[0];
                args[1] = "two";
                args[2] = "one";
                args[3] = NULL;
                _spawnv(P_OVERLAY, argv[0], args);
                abort();
                /* not executed because child was overlaid */
            }
            abort();       /* argument not valid */

         case GRANDCHILD:   /* two arguments passed */
            printf("grandchild %s ran\n", argv[1]);
            _exit(0);
      }

   /******  The output should be similar to:  *************

       child process one began
       grandchild one ran
       child process two began
       grandchild two ran
       Parent process ended
   */
   }
```

17.0 Dynamische Bibliotheken − DLL

Dynamic Link Libraries (DLL) dienen unter Microsoft Windows und OS/2 zum dynamischen Laden von Programmteilen, die u.U. von mehreren Anwendungen genutzt werden.

Dieses Kapitel beschreibt, wie DLLs erstellt und prinzipiell genutzt werden. Es wird nicht auf die Probleme eingegangen, die entstehen, wenn DLLs in Multithreading−Umgebungen eingesetzt werden.

17.1 Konzept der dynamischen Bibliotheken

Dynamische Bibliotheken werden vorzugsweise immer dann verwendet, wenn mehrere Programme gleiche Funktionen benötigen. Dadurch werden diese Funktionen nicht mehr in jedes Programm integriert sonder nur einmal als DLL zur Verfügung gestellt. Dies spart zum einen Speicherplatz und zum anderen erleichtert es die Pflege dieser Funktionen. Wird ein Fehler entdeckt, muß nur die DLL korrigiert und ausgetauscht werden. Nachteilig ist, daß man sich Gedanken darum machen muß, ob die Funktionen der DLL von mehreren Programmen gleichzeitig genutzt werden können. Probleme sind zu erwarten, wenn Daten auf dem Heap oder in statischem Speicher genutzt werden. Funktionen, die dies machen, werden als nicht reentrant bezeichnet. Hier müssen dann Techniken angewendet werden, um den Zugriff auf derartige Daten zu serialisieren. D.h., daß erst dann ein weiterer Thread mit den Daten arbeiten kann, wenn die Arbeit eines anderen beendet wurde. Unproblematisch sind alle als `auto` deklarierten Daten (die auf dem stack abgelegt werden) und sog. Thread−lokale Daten.

17.2 Schritte auf dem Weg zur DLL

Da die einzelnen Anweisungen vom Compiler abhängig sind, sind die folgenden Anweisungen als Beispiele für den IBM C/C++ Compiler zu verstehen. Andere Compiler wie z.B. GCC heissen anders und benutzen andere Parameter. Hier hilft nur ein Blick in die Compilerdokumentation.

17.2.1 DLL erstellen

1. DEF Datei erstellen

   Die Definitionsdatei enthält die Beschreibung der zu erstellenden DLL und listet die Funktionen und Datenelemente auf, die von anderen Programmen aus aufgerufen/genutzt werden können. Dies wird als Export bezeichnet.
   Beispiel für IBM C/C++ Compiler unter Windows:

   ```
   LIBRARY SUPPORT
   EXPORTS
       strip
       sstrip
       words
       wordx
       wordreplace
       wordindex
       word
       wordlength
       wordpos
   ```

   Es wird eine DLL mit dem Namen `SUPPORT` definiert, die die Funktionen, deren Namen unter `EXPORTS` aufgeführt sind, zur Verfügung stellt.

2. DLL compilieren und linken. Beispiel für IBM C/C++ Compiler:

```
icc /ge- /gd- support.c support.def /FEsupport.dll
```

Parameter:
`support.c` : Name(n) der Quelldatei(en)
`support.def` : DEF Datei für die DLL
`/ge-` : statisch gelinkte runtime library
`/gd-` : DLL erzeugen
`/FEsupport.dll` : Name der zu erzeugenden DLL

Der Aufruf erstellt DLL, EXP und LIB Dateien. Die LIB Datei wird vom Linker genutzt, um externe Referenzen vom Objektcode aufzulösen. Wenn der Linker im Objektcode eine Referenz auf eine Fuktion oder ein Datenelement findet, das auf externe Bibliotheken (Libraries) verweist, bindet der Linker diese Module automatisch ein. Für die sog. Standardbibliotheken ist es nicht notwendig, diese beim Linker anzugeben, sie werden automatisch eingebunden. Bei DLLs muss die Bibliothek beim Linken der Anwendung angegeben werden, die die DLL nutzen soll.

17.2.2 DLL benutzen

1. Die Funktionen, die aus einer DLL geladen werden sollen, mit `_Import` kennzeichnen. Die Namen müssen mit den Namen in der Definitionsdatei übereinstimmen.

   Möchte man eine bereits existierende DLL nutzen, so benötigt man die Funktionsdeklarationen vom Hersteller. Die DEF und LIB Dateien kann man u.U. mit Hilfsmitteln des Compilers erstellen (beim IBM C/C++ Compiler das Programm ilib).

   Beispiel:

```
int _Import words(char string[]);
```

2. EXE compilieren und linken. Beispiel für IBM C/C++ Compiler:

```
icc mymain.c support.lib
```

18.0 Reguläre Ausdrücke

Reguläre Ausdrücke (Regular Expressions) sind Beschreibungen, die der Suche von Mustern in Zeichenketten dienen. Viele Programme im Unix Umfeld nutzen Regular Expressions, z.B. egrep. Dementsprechend bieten die Dokumentation zu diesen Programmen gute Informationen zu Regular Expressions, Beispiel auch hier egrep. Regular Expressions gehören nicht zum Sprachumfang von ANSI C und sind somit nicht mit jedem Compiler übersetzbar. Jeder POSIX oder XPG4 kompatible Compiler sollte aber über die entsprechenden Funktionen verfügen.

18.1 Spezielle Zeichen in Regular Expressions

Zeichen	**Bedeutung**
\	Das nächste Zeichen seiner Sonderbedeutung berauben und als normales Zeichen betrachten.
\w	Jedes alphanumerische Zeichen inkl. Unterstrich. Entspricht [A−Za−z0−9_]
\W	Jedes nicht−alphanumerische Zeichen. Entspricht. [^A−Za−z0−9_]
\d	Jede Zahl. Entspricht [0−9]
\D	Jedes Zeichen, das keine Zahl ist. Entspricht [^0−9]
[xyz]	Jedes angegebene Zeichen. Ein Bereich kann durch Verwendung des − (z.B. [A−Fa−f0−9]) angegeben werden.
[^xyz]	Jedes nicht angegebene Zeichen. Ein Bereich kann durch Verwendung des − angegeben werden.
^	Kennzeichnet den Anfang der Eingabezeile.
$	Kennzeichnet das Ende der Eingabezeile.
*	0 oder mehr des vorangegangenen Zeichens.
+	1 oder mehr des vorangegangenen Zeichens. Entspricht {1,}
?	0 oder 1 des vorangegangenen Zeichens.
.	Jedes einzelne Zeichen aus newline.
x|y	Entweder 'x' oder 'y'.
{n}	Exakt n mal das vorangegangene Zeichen.
{n,}	Mind. n mal das vorangegangene Zeichen.
{n,m}	Mind. n mal, aber maximal m mal das vorangegangene Zeichen.
[\b]	Backspace
\B	nicht−Wort Grenze
\c X	'x' ist ein Kontrollzeichen
\f	Form feed
\n	Line feed
\r	carriage return
\s	einzelnes "white space" Zeichen (Leerzeichen, Tabulator, Form feed, Line Feed). Entspricht [\f\n\r\t\v]
\S	Einzelnes Zeichen, das kein "white space" Zeichen ist. Entspricht [^\f\n\r\t\v]
\t	Tabulatorzeichen
\v	Vertikaler Tabulator

\n	Rückwärtsreferenz auf einen Substring.

18.2 Beispiele für reguläre Ausdrücke

Ausdruck	Beschreibung
</?[a−zA−Z0−9]{1,9}>	Sucht HTML Befehle mit einer Länge von bis zu 9 Zeichen innerhalb der eckigen Klammern (/ nicht eingerechnet).
\([0−9]{3}\) \d{3}−\d{4}	Findet US amerikanische Telefonnummern, z.B. (212)−555−1212

Mit Vorsicht zu geniessen sind Konstrukte wie `test*`, bei dem DOS Benutzer davon ausgehen würden, daß alles, was mit `test` beginnt, gefunden wird. Um dies zu erreichen muß man allerdings `test.*` benutzen.

18.3 Quellcodebeispiele für reguläre Ausdrücke

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

int main(void)
{
   regex_t    preg;
   char       *string = "<TR><TD>Gesellschaft:</TD><TD>IBM Informationssysteme Deutschland
   char       *pattern = "</?[a-zA-Z0-9]{1,9}>"; // scan HTML tag up to 9 chars long incl.

   int        rc, offset=0;
   size_t     nmatch = 1;
   regmatch_t pmatch[1];

   if (0 != (rc = regcomp(pattern, REG_EXTENDED)))
   {
      printf("regcomp() failed, returning nonzero (%d)\n", rc);
      exit(EXIT_FAILURE);
   }

   if (0 != (rc = regexec(string, nmatch, pmatch, 0)))
   {
      printf("Failed to match '%s' with '%s',returning %d.\n",
      string, pattern, rc);
   }
   else
   {
      printf("-> offset: %d -> %s\n",offset,string+offset);
      do
      {
         printf("A matched substring \"%.*s\" is found at position %d to %d.\n",
                pmatch[0].rm_eo - pmatch[0].rm_so, .rm_so+offset],
                pmatch[0].rm_so, pmatch[0].rm_eo - 1);
         offset+=pmatch[0].rm_eo;
         printf("\n-> offset: %d -> %s\n",offset,string+offset);
      }
      while(0 == (rc = regexec(string+offset, nmatch, pmatch, 0)));
   }
   regfree(

   return 0;
}
```

19.0 Makefiles

In größeren Programmen wird man früher oder später den Quellcode auf mehrere Dateien aufteilen. Wenn nun lediglich an einer oder wenigen Dateien Änderungen gemacht wurden, lohnt es sich nicht, alle Dateien erneut zu übersetzen. Das Programm `make` übernimmt die Aufgabe herauszufinden, wie ein Programm übersetzt werden muß und welche Teile neu compiliert werden sollen. Als Eingabe benötigt `make` dazu das Makefile, dessen Dateiname auch `Makefile` lauten sollte.

Ein Makefile enthält Regeln, wie das Programm übersetzt werden soll. Den Großteil dieser Regeln machen die Abhängigkeiten, die *dependencies* aus. Diese regeln, welche Datei vor welcher compiliert werden muß und welche Dateien aus welchen erzeugt werden.

Zusätzlich enthält das Makefile Angaben, welche Programmaufrufe für die diversen Übersetzungsschritte zu tätigen sind.
Beispiel:

```
test: library.o test.o
```

`test` benötigt die Objectfiles `library.o` und `test.o` vor seiner eigenen Fertigstellung.

`test:` ist ein sogenanntes Ziel, das beim Aufruf von `make` angegeben werden kann (z.B. `make test`). Unterhalb des Zieles muß noch angegeben werden, wie `test` erzeugt werden soll. Beispiel:

```
        $(CC) -Wall -o $@ test.o library.o
```

`$(CC)` ist ein Platzhalter für den Compilernamen. `$@` ist der Name des Ziels, also `test`. `library.o` hat nun z.B. die folgenden Abhängigkeiten:

```
library.o:library.c support.c
        $(CC) -Wall -c -o library.c support.c
```

Einrückungen sollten sicherheitshalber mit der Tabulatortaste und nicht mit Leerzeichen erfolgen.

19.1 Besonderheiten

Eine Besonderheit ist folgendes Ziel in einem Makefile:

```
.c.o:
        $(CC) -Wall -c -o $*.o $<
```

Dies bedeutet, daß die Objectfiles aus einem C Quellcode mit gleichem Namen erzeugt werden, wobei `$*.o` den Namen des Objectfiles bezeichnet und `$<` der Name der Datei ist, von der das jeweilige Objectfile abhängt.

19.2 Komplettes Beispiel

```
CC     = gcc
OBJS   = library.o support.o test.o
CFLAGS = -Wall
```

```
all:    test.o

test:   library.o test.o
        $(CC) $(CFLAGS) -o $@ test.o

.c.o:
        $(CC) $(CFLAGS) -c -o $*.o $<

clean:
        rm -f *.o test
```

Dieses Beispiel kann mit `make` aufgerufen werden, um `test` zu übersetzen. Der Aufruf `make clean` löscht alle Objectfiles und die ausführbare Datei `test`

19.3 Weiterführende Informationen

- Hilfe zu GNU make unter http://www.gnu.org/manual/make/html_chapter/make_14.html
- "An Introduction to C Development on Linux" unter http://uw.physics.wisc.edu/~nextroom/introduction-to-c.html#make

20.0 Multithreading

ACHTUNG! Hier fehlt noch eine Menge Text und die Beispiele sind noch nicht ausgetestet!

TODO: Thread Konzept erläutern

Unterschied Runtime−Threads (_beginthread()), Betriebssystem−Threads (z.B. MS Windows: CreateThread()) und POSIX−Threads (fork())

Probleme beim Multithreading erläutern (Serialisierung, Koordinierung, Debugging, Abhängigkeit von Betriebsystemunterstützung, ...)

Beim Compilieren muss angegeben werden, dass Multithreading−Library gelinkt wird (IBM C/C++: /Gm+)

20.1 beginthread

Die Funktion `_beginthread()` ist wie folgt definiert:

```
#include <stdlib.h>   /* also in <process.h> */
int _beginthread(void (*start_address) (void *),
                    (void *)stack,
                    unsigned stack_size,
                    void *arglist);
```

Beispiel: (Quelle: IBM C/C++ Compilers Dokumentation)

```
/*
  Compilieren: icc /Gm+ dateiname.c
*/
#include <windows.h>                            /* MS Windows */

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

static volatile int wait = 1;

void _Optlink bonjour(void *arg)
{
   int i = 0;
   while (wait)              /* wait until the thread id has been printed     */
        Sleep(01);
   while (i++ <5)
      printf("Bonjour!\n");
}
int main(void)
{
   int tid;
   tid = _beginthread(bonjour, NULL, 8192, NULL);
   if (-1 == tid) {
      printf("Unable to start thread.\n");
      return EXIT_FAILURE;
   }
   else {
      printf("Thread started with thread identifier number %d.\n", tid);
      wait = 0;
   }
```

```
  WaitForSingleObject((HANDLE)tid, INFINITE); /* wait for thread bonjour to end */
                                              /*  before ending main thread     */
   return 0;
   /*************************************************************************
      The output should be similar to:
       Thread started with thread identifier number 2.
       Bonjour!
       Bonjour!
       Bonjour!
       Bonjour!
       Bonjour!
   *************************************************************************/
}
```

20.2 CreateThread

Die Funktion `CreateThread()` ist wie folgt definiert:

```
  HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  // pointer to thread security attributes
    DWORD dwStackSize,                         // initial thread stack size, in bytes
    LPTHREAD_START_ROUTINE lpStartAddress,     // pointer to thread function
    LPVOID lpParameter,                        // argument for new thread
    DWORD dwCreationFlags,                     // creation flags
    LPDWORD lpThreadId                         // pointer to returned thread identifier
   );
```

Weitere Informationen: http://msdn.microsoft.com/library/en-us/dllproc/base/createthread.asp

Beispiel: (Quelle: Microsoft Platform SDK)

```
#include <windows.h>
#include <conio.h>

DWORD WINAPI ThreadFunc( LPVOID lpParam )
{
    char szMsg[80];

    wsprintf( szMsg, "Parameter = %d.", *(DWORD*)lpParam );
    MessageBox( NULL, szMsg, "ThreadFunc", MB_OK );

    return 0;
}

VOID main( VOID )
{
    DWORD dwThreadId, dwThrdParam = 1;
    HANDLE hThread;
    char szMsg[80];

    hThread = CreateThread(
        NULL,                        // default security attributes
        0,                           // use default stack size
        ThreadFunc,                  // thread function
                        // argument to thread function
        0,                           // use default creation flags
```

```
                        // returns the thread identifier

   // Check the return value for success.

   if (hThread == NULL)
   {
      wsprintf( szMsg, "CreateThread failed." );
      MessageBox( NULL, szMsg, "main", MB_OK );
   }
   else
   {
      _getch();
      CloseHandle( hThread );
   }
}
```

21.0 Socketprogrammierung

ACHTUNG! Hier fehlt noch eine Menge Inhalt...

Das modulare Beispiel ist noch nicht funktionstüchtig!

21.1 Beispiel für Socketprogrammierung

21.1.1 Einfacher HTTP Client

```
      //
      // Dateien von einem Web Server per HTTP/1.0 laden und in Datei speichern
      //
      // Hostname des Webservers statisch in "char servername[]" angegeben.
      // Zu ladende Datei statisch in "char nachricht[]" angegeben.
      //
      // Compilieren: icc http.c ws_32.lib
      //

#include <stdio.h>
#include <windows.h>


int main(int argc, char *argv[])
{
  struct hostent *host = NULL;                              // Strukturzeiger für DNS Abfrage
  struct sockaddr_in server, client;                        // Adressstruktur für Server

  char servername[] = "www-bab.berlin-moabit.de.ibm.com";  // Hostname des Servers
  char nachricht[] = "GET /hilfe/ HTTP/1.0\r\n\r\n\0";     // HTTP Request Header
  int  anzahlBytes = 0;                                     // zählt die empfangenen Bytes
  char puffer[8192];                                        // Puffer für Empfangsdaten
  int  anzahlPakete=0;                                      // zählt die empfangenen Pakete
  FILE *ausgabeDatei;                                       // Strukturzeiger für zu sichernd
  int meinSocket = 0;                                       // lokaler Socket

  WSADATA wsadata;  // für Windows Socket Funktionen



      //
      // Programmstart
      //
      printf("%s vom %s\n",argv[0],__TIMESTAMP__);

      // Windows Socket Funktionen initialisieren
  WSASetLastError(0);
  WSAStartup(MAKEWORD(2,0),



      //
      // "Verwaltungsdaten" für Server
      //
  // DNS Abfrage
  host = gethostbyname(servername);

  if (host==NULL)
  {
     printf("Konnte %s nicht in IP Adresse aufloesen!\n",servername);
```

```
      // Name konnte nicht aufgelöst werden, es wird angenommen, dass servername
      // eine IP Adresse ist.
      // ACHTUNG! Solche Annahmen können unangenehme Auswirkungen haben.
      server.sin_addr.s_addr = inet_addr(servername);
   }
   else
   {
      // IP Adresse aus DNS Auflösung in Adressstruktur übernehmen
      memcpy( ), host->h_addr, host->h_length);
      // Adressart in Adressstruktur übernehmen (Art ist IP Adresse)
      server.sin_family = host->h_addrtype;

      printf("Konnte %s erfolgreich in IP Adresse aufloesen.\n", servername);
      printf("%s hat die IP Adresse %s\n", servername, inet_ntoa(server.sin_addr));
   } // Ende DNS Auflösung

   // Adressstruktur für Server vervollständigen
   // Portnummer des Servers angeben (Port 80 -> HTTP Serverdienst)
   server.sin_port = htons(80);
   // Rest der Adressstruktur aus Sicherheitsgründen mit Nullen auffüllen
   memset( server.sin_zero, 0, sizeof(server.sin_zero) );



     //
     // "Verwaltungsdaten" für Client
     //
     // Adressstruktur für Client mit Daten füllen
     client.sin_family = AF_INET;           // AF_INET    -> Internet Adresse (IP)
     client.sin_port   = INADDR_ANY;        // INADDR_ANY -> beliebige Portnummer
     client.sin_addr.s_addr = INADDR_ANY;   // "beliebige" IP Adresse

     // Socket für Client erzeugen
     meinSocket = socket (AF_INET, SOCK_STREAM, 0); // SOCK_STREAM -> TCP Socket

     if (meinSocket == ~0) // ~0 entspricht hier Windows Fehler INVALID_SOCKET
     {
   printf("Fehler bei socket(): %d\n", WSAGetLastError());
             exit (-2);
     } // Ende Socket konnte nicht erzeugt werden

   // Adressstruktur mit leerem Socket "verheiraten" (binden)
   if ( bind (meinSocket, (struct sockaddr *) sizeof(client) ) != 0 )
   {
      printf("Fehler bei bind(): %d\n", WSAGetLastError());
      exit (-3);
   }



     //
     // Kontaktaufnahme und Datenübertragung
     //
     // Server kontaktieren
   if ( connect (meinSocket, (struct sockaddr *) sizeof(server) ) != 0 )
   {
      printf("Fehler bei connect(): %d\n", WSAGetLastError());
      exit (-4);
   }

      printf("Server erfolgreich kontaktiert!\n");

   // HTTP Request Header abschicken
   anzahlBytes = send ( meinSocket, nachricht, strlen(nachricht), 0 );
   if ( anzahlBytes < 0 )
```

```
   {
      printf("Fehler bei send(): %d\n", WSAGetLastError());
      exit (-5);
   }

   // Paket wurde erfolgreich an Server geschickt...

   // Datei zum Speichern der Antwort anlegen
   ausgabeDatei = fopen( "datei.tmp", "w" );
   if ( ausgabeDatei==NULL )
   {
      printf("Datei konnte nicht geöffnet werden!\n");
      exit (-6);
   }

   // Empfangspuffer mit 0 initialisieren -> Puffer löschen
   memset ( puffer, 0, sizeof(puffer) );

   // Solange Daten mit recv() empfangen bis nichts mehr ankommt
   while ( anzahlBytes != 0 )
   {
      if ( ( anzahlBytes = recv(meinSocket, puffer, sizeof(puffer), 0) ) == -1)
      {
         printf("Fehler bei recv(): %d\n", WSAGetLastError());
         exit (-7);
      }

      // Im ersten Paket ist der HTTP-Header mit drin, den könnte man hier entfernen

      printf("Paket Nr. %d mit %d Bytes empfangen\n",
             ++anzahlPakete, strlen(puffer));
      fwrite ( puffer, sizeof(char), strlen(puffer), ausgabeDatei );

      // printf("%s\n", puffer);              // empfangene Daten am Bildschirm ausgeben
      memset ( puffer, 0, sizeof(puffer) ); // Puffer löschen

   } // Ende Schleife solange Daten vom Server kommen



       //
       // Übertragung abgeschlossen, aufräumen...
       //

   // Ausgabedatei schliessen
   fclose (ausgabeDatei);

   // Socket schliessen -> Ressourcen an Betriebssystem zurückgeben
   closesocket (meinSocket);



   return 1;
} // Ende main()
```

21.1.2 Häufig benötigte Funktionalität

21.1.2.1 Server kontaktieren

In allen Clientprogrammen tauchen am Anfang immer die gleichen Aufgaben auf:

- Adressstrukturen mit Daten füllen

- Socket erzeugen
- Socket binden
- Server kontaktieren

Diese Funktionalität wird in eine eigene Funktion, im folgenden unter dem Namen `kontaktierServer` geführt, ausgegliedert. Der folgende Programmcodeabschnitt beinhaltet die Angaben der Headerdatei `kontaktierServer.h`.

```
// Header für kontaktierServer.c

#ifndef _kontaktierServer_
#define _kontaktierServer_

int kontaktierServer ( char *servername, int portnummer );  // Prototyp

#endif
```

Der folgende Codeabschnitt enthält die gemeinsam genutzte Funktionalität.

```
//
// kontaktierServer.c
//
// int kontaktierServer(char *servername, int portnummer)
// - führt Verbindungsaufbau zu angegebenem Server (servername) auf angegebener
//   Portnummer (portnummer) durch
// - im Erfolgsfall wird die Socketnummer zurückgeliefert
// - im Fehlerfall wird 0 zurückgeliefert
//
// Beispielaufruf: socket = kontaktierServer("w3.ibm.com", 80);
//

#include <stdio.h>       // für sprintf()
#include <windows.h>     // für gethostbyname(), WSAxyz()

#include "kontaktierServer.h"


int kontaktierServer ( char *servername, int portnummer )
{
  struct hostent *host = NULL;                    // für DNS Abfrage
  struct sockaddr_in server, client;              // Adressstruktur für Server

  int meinSocket = 0;                             // Socket des Clients



  // DNS Abfrage
  host = gethostbyname (servername);

  if (host==NULL)
  {
     // printf("Konnte %s nicht in IP Adresse aufloesen.\n", servername);
     // Name konnte nicht aufgelöst werden, es wird angenommen, dass servername
     // eine IP Adresse ist.
     // ACHTUNG! Solche Annahmen können unangenehme Auswirkungen haben.
     server.sin_addr.s_addr = inet_addr(servername);
  }
  else
  {
     // IP Adresse aus DNS Auflösung in Adressstruktur übernehmen
```

```
      memcpy( ), host->h_addr, host->h_length);
      // Adressart in Adressstruktur übernehmen (Art ist IP Adresse)
      server.sin_family = host->h_addrtype;

      // printf("Konnte %s erfolgreich in IP Adresse aufloesen.\n", servername);
      // printf("%s hat die IP Adresse %s\n", servername, inet_ntoa(server.sin_addr));
   } // Ende DNS Auflösung

   // Adressstruktur für Server vervollständigen
   // Portnummer des Servers angeben (Port 80 -> HTTP Serverdienst)
   server.sin_port = htons (portnummer);
   // Rest der Adressstruktur aus Sicherheitsgründen mit Nullen auffüllen
   memset( server.sin_zero, 0, sizeof(server.sin_zero) );

   // Socket für Client erzeugen
   meinSocket = socket (AF_INET, SOCK_STREAM, 0); // SOCK_STREAM -> TCP Socket

   if (meinSocket == ~0) // ~0 entspricht hier Windows Fehler INVALID_SOCKET
   {
      printf("Fehler bei socket(): %d\n", WSAGetLastError());
      return (-2);
   } // Ende Socket konnte nicht erzeugt werden

   // Adressstruktur für Client mit Daten füllen
   client.sin_family = AF_INET;           // AF_INET    -> Internet Adresse (IP)
   client.sin_port   = INADDR_ANY;        // INADDR_ANY -> beliebige Portnummer
   client.sin_addr.s_addr = INADDR_ANY;   // "beliebige" IP Adresse

   // Adressstruktur mit leerem Socket "verheiraten" (binden)
   if ( bind (meinSocket, (struct sockaddr *) sizeof(client) ) != 0 )
   {
      printf("Fehler bei bind(): %d\n", WSAGetLastError());
      return (-3);
   }

   if ( connect (meinSocket, (struct sockaddr *) sizeof(server) ) != 0 )
   {
      printf("Fehler bei connect(): %d\n", WSAGetLastError());
      return (-4);
   }

   // printf("Server erfolgreich kontaktiert!\n\n");

   return meinSocket;
} // Ende kontaktierServer()
```

21.1.2.2 Gegenstelle identifizieren

Die Funktion `identifizieren()` ermittelt die eigene IP Adresse und den dazugehörigen Hostnamen.

```
// Header für identifizieren.c

#ifndef _gegenstelleIdentifizieren_
#define _gegenstelleIdentifizieren_

int identifizieren ( int socket, char *hostname, int sizeHostname );

#endif
```

Die Funktion `getsockname()` ermittelt die eigene IP Adresse und Portnummer anhand des angegebenen Sockets. Die Funktion `gethostbyaddr()` ermittelt den Hostnamen für die zuvor ermittelte IP Adresse und

kopiert diesen in das Feld `hostname`.

```
//
// identifizieren.c
//
// int identifizieren ( int socket, char *hostname, int sizeHostname );
// - ermittelt die eigene IP Adresse und den eigenen Hostnamen
// - im Erfolgsfall wird 1 zurückgeliefert
// - im Fehlerfall wird ein negativer Wert zurückgeliefert
//
// Beispielaufruf: rc = identifizieren ( socket, hostname, sizeof(hostname) );
//

#include <stdio.h>       // für sprintf()
#include <windows.h>     // f r gethostbyname(), WSAxyz()

#include "identifizieren.h"

int identifizieren ( int socket, char *hostname, int sizeHostname )
{
  int    sockaddrlength = 0;
  struct sockaddr_in local;
  struct hostent *DNS;

  sockaddrlength=sizeof(local);
  // getsockname(): eigene IP Adresse und Portnummer aus Socket extrahieren
  getsockname(socket, (struct sockaddr *)
  // getpeername(): IP Adresse und Portnummer des Servers aus Socket extrahieren
  // getpeername(socket, (struct sockaddr *)
  DNS = gethostbyaddr((char*),sizeof(struct in_addr),AF_INET);
  if ( DNS == NULL )
  {
     // printf("%s:%d\n", inet_ntoa(local.sin_addr), ntohs(local.sin_port));
     if ( sizeHostname>strlen(inet_ntoa(local.sin_addr)) )
     {
        strcpy ( hostname, inet_ntoa(local.sin_addr) );
     }
     else
     {
        return -1;
     }
  }
  else
  {
     // printf("%s [%s:%d]\n", DNS->h_name, inet_ntoa(local.sin_addr), ntohs(local.sin_port)
     if ( sizeHostname>strlen(DNS->h_name) )
     {
        strcpy ( hostname, DNS->h_name );
     }
     else
     {
        return -2;
     }
  }

  return 1;
} // Ende identifizieren()
```

21.1.2.3 Daten senden

```
// Header für datenSenden.c

#ifndef _datenSenden_
#define _datenSenden_

int datenSenden ( char *puffer, int laenge, int meinSocket );

#endif
```

```
//
// datenSenden.c
//
// int datenSenden ( char *puffer, int laenge, int meinSocket )
// - verschickt Daten über eine bestehende Verbindung
// - im Erfolgsfall wird 1 zurückgeliefert
// - im Fehlerfall wird ein negativer Wert zurückgeliefert
//
// Beispielaufruf: rc = datenSenden ( message, sizeof(message), socket );
//

#include <stdio.h>       // für sprintf()
#include <windows.h>     // f r gethostbyname(), WSAxyz()

#include "datenSenden.h"

int datenSenden ( char *puffer, int laenge, int meinSocket )
{
  int anzahlBytes = 0;

  if ( (anzahlBytes = send(meinSocket, puffer, laenge, 0)) < 0)
  {
     printf("Fehler bei send(): %d\n", WSAGetLastError());
     return (-2);
  }
  printf("Client: %s\n", puffer);

  return 1;
} // Ende datenSenden()
```

21.1.3 Einfacher SMTP Client

Der folgende Beispielcode für einen einfachen SMTP Server nutzt die oben aufgeführten Funktionen `kontaktierServer()` und `datenSenden()`. Der grundsätzliche Ablauf beim Verschicken einer E-Mail mittels SMTP ist im Kommentarblock am Anfang des Quellcodes dargestellt. Das Beispielprogramm führt keinerlei Authentifizierung durch! Anstelle von `smtpserver.com` ist ein gültiger SMTP Server Hostname anzugeben. Die E-Mail Adressen für Sender und Empfänger sowie der Text der E-Mail selbst sind selbstverständlich ebenfalls entsprechend anzupassen. Der Beispielcode führt keinerlei Konvertierungen (z.B. für Umlaute) durch und behandelt auch keine Sonderfälle. Ebenso werden keine Attachments unterstützt.

```
//
// SMTP Client (RFC 821)
// Mail an SMTP Server schicken
//
// 1. Server auf Port 25 kontaktieren, Server antwortet sofort
```

```
//    (220 d12relay01.de.ibm.com ESMTP Sendmail 8.11.1m3/NCO v5.01; Wed, 6 Fe...)
// 2. Client schickt HELO Kommando an Server, Server antwortet
//    HELO Identifikation
//    (250 d12relay01.de.ibm.com Hello dhcp2-42.berlin.de.ibm.com [9.165.174...)
// 3. Client schickt MAIL Kommando an Server, Server antwortet
//    MAIL FROM:<e-Mail Adresse>
//    (250 2.1.0 <shkliche@de.ibm.com>... Sender ok)
// 4. Client schickt RCPT Kommando an Server, Server antwortet
//     RCPT TO:<e-Mail Adresse>     <- An diese Adresse wird die Mail geschickt.
//    (250 2.1.5 <shkliche@de.ibm.com>... Recipient ok)
// 5. Client schickt DATA Kommando an Server, Server antwortet
//    (354 Enter mail, end with "." on a line by itself)
//    DATA
// 6. Client schickt Mailinhalt an Server
//    Beispiel:
//    Date: 6 Feb 2 15:19:03
//    From: Sascha Kliche <shkliche@de.ibm.com>
//    Subject: keins
//    To: fdicks@de.ibm.com
//
//    Hier kommt die Testmail.
//    Wir koennen auch mehrere Zeilen senden.
//
//    .
//    Server antwortet: 250 2.0.0 g16EKQw73202 Message accepted for delivery
// 7. Client schickt QUIT Kommando an Server, Server antwortet
//    QUIT
//    (221 2.0.0 d12relay01.de.ibm.com closing connection)
//
// Compileraufruf
// icc smtp-client.cpp kontaktierServer.cpp datenSenden.cpp identifizieren.cpp ws2_32.lib


#include <windows.h>
#include <stdio.h>
#include <time.h>

#include "smtp-client1.h"
#include "kontaktierServer.h"
#include "identifizieren.h"
#include "datenSenden.h"


int datenEmpfangen ( int meinSocket );
int endeSuchen(char *paket, int paketLaenge);

int main ( int argc, char *argv[] )
{
  char servername[300]="smtpserver.com";
  char hostname[300];
  int  meinSocket = 0;
  char nachricht[4096];
  char nachrichtKodiert[8192];
  char datum[100];

  WSADATA wsadata;
  SYSTEMTIME systemtime;

  // Ab hier geht's los...
  WSASetLastError ( 0 );
  WSAStartup ( MAKEWORD(2,0), );

  // Datum und Uhrzeit ermitteln
  memset ( datum, 0, sizeof(datum) );
  GetSystemTime( );
```

```
    sprintf ( datum, "Date: %s, %02d %s %04d %02d:%02d:%02d GMT\r\n\0",
              wkday[systemtime.wDayOfWeek-1], systemtime.wDay, month[systemtime.wMonth],
              systemtime.wYear, systemtime.wHour, systemtime.wMinute, systemtime.wSecond);

    // Server auf Port 25 (SMTP) kontaktieren
    meinSocket = kontaktierServer ( servername, 25 );

    memset ( hostname, 0, sizeof(hostname) );
    identifizieren ( meinSocket, hostname, sizeof(hostname) );

    // Begrüssung des Servers empfangen
    datenEmpfangen ( meinSocket );

    // Server begrüssen
    memset ( nachricht, 0, sizeof(nachricht) );
    sprintf ( nachricht, "HELO %s\r\n\0", hostname );
    datenSenden ( nachricht, strlen(nachricht), meinSocket );
    datenEmpfangen ( meinSocket );

    // Absender angeben
    memset ( nachricht, 0, sizeof(nachricht) );
    strcat ( nachricht, "MAIL FROM:<Absender@ibm.com>\r\n\0" );
    datenSenden ( nachricht, strlen(nachricht), meinSocket );
    datenEmpfangen ( meinSocket );

    // Empfänger angeben, an den die Mail zugestellt wird
    memset ( nachricht, 0, sizeof(nachricht) );
    strcat ( nachricht, "RCPT TO:<Empfaenger@ibm.com>\r\n\0" );
    datenSenden ( nachricht, strlen(nachricht), meinSocket );
    datenEmpfangen ( meinSocket );

    // Datenteil ankündigen
    memset ( nachricht, 0, sizeof(nachricht) );
    strcat ( nachricht, "DATA\r\n\0" );
    datenSenden ( nachricht, strlen(nachricht), meinSocket );
    datenEmpfangen ( meinSocket );

    // Datenteil (E-Mail-Header und Text der E-Mail)
    memset ( nachricht, 0, sizeof(nachricht) );
    strcat ( nachricht, datum );
    // Bei konvertierten Umlauten entsprechenden MIME Header einfügen, z.B. mit
    // strcat ( nachricht, "MIME-Version: 1.0\r\nContent-Type: text/plain; charset=ISO-8859-1\
    strcat ( nachricht, "From: Horst Hansen <Sender@ibm.com>\r\nSubject: Testmail\r\nTo: Empfa

    // Hier die Nachricht angeben, die verschickt werden soll, Zeilenumbrüche mit \r\n
    // Wenn Umlaute verwendet werden, muessen diese mit "Quoted Printable Encoding" umgewandel
    // RFC821 definiert noch weitere Fälle, die beachtet werden müssen.
    strcat ( nachricht, "Hier steht der E-Mail Text.\r\n\r\nAchtung!\r\nKeine Umlaute verwende

    strcat ( nachricht, "\r\n.\r\n\0" ); // Nachricht abschliessen
    datenSenden ( nachricht, strlen(nachricht), meinSocket );
    datenEmpfangen ( meinSocket );

    // Übertragung beenden
    memset ( nachricht, 0, sizeof(nachricht) );
    strcat ( nachricht, "QUIT\r\n\0" );
    datenSenden ( nachricht, strlen(nachricht), meinSocket );
    datenEmpfangen ( meinSocket );

    closesocket ( meinSocket );

    return 1;
  } // Ende main()
```

```
int datenEmpfangen ( int meinSocket )
{
  char puffer[4096];
  int  anzahlBytes = 0, endeGefunden = 0;

  do
  {
    memset ( puffer, 0, sizeof(puffer) );
    if ( (anzahlBytes = recv(meinSocket, puffer, sizeof(puffer), 0)) == -1 )
    {
     printf("Fehler bei recv(): %d\n", WSAGetLastError());
       return (-1);
    }
    printf("Server: %s\n", puffer);
    endeGefunden = endeSuchen ( puffer, anzahlBytes );
  } while ( puffer[3]=='-' &endeGefunden==0 );  // while ( puffer[3]!=' ');

  return 1;
} // Ende datenEmpfangen()



int endeSuchen(char *paket, int paketLaenge)
{
  int  zeilenAnfang = 0, paketZeile = 0;
  char zeile[512];
  int  zaehler = 0, zaehler2 = 0;
  int  endeGefunden = 0;

  // einzelne Paketzeilen auswerten
  memset ( zeile, 0, sizeof(zeile) );
  for ( zaehler=0; zaehler<paketLaenge; zaehler++ )
  {
    if ( paket[zaehler] == '\r' || paket[zaehler] == '\n' )
    {
      memcpy ( zeile, paket+zeilenAnfang, zaehler-zeilenAnfang );
      paketZeile++;
     printf("\n %d. Zeile: %s\n", paketZeile, zeile);
      zeilenAnfang = zaehler+2;
      zaehler++;
      if (zeile[3] == ' ')
      {
         endeGefunden = 1;
         break;
      }
      memset ( zeile, 0, sizeof(zeile) );
    } // Ende gefunden
  }
  // Ende einzelne Paketzeilen auswerten

  return endeGefunden;
} // Ende endeSuchen()
```

21.1.4 Einfacher POP3 Client

```
//
// POP3 Client (RFC 1939)
// Mail von POP3 Server empfangen
//
// 1. Server auf Port 110 kontaktieren, Server antwortet sofort
//    (+OK POP3 server ready)
```

```
// 2. Client schickt USER Kommando an Server, Server antwortet
//    USER name
//    (+OK name)
// 3. Client schickt PASS Kommando an Server, Server antwortet
//    PASS passwort
//    (+OK passwort)
// 4. Client schickt STAT Kommando an Server, Server antwortet
//    STAT
//    (+OK n m  (n: Anzahl Nachrichten, m: Größe in Oktetten))
// 5. Client schickt LIST Kommando an Server, Server antwortet
//    LIST [n]
//    (
//       +OK n messages (m octets)
//       1 m
//       2 m
//       [...]
//       .
//    )
// 6. Client schickt RETR Kommando an Server, Server antwortet
//    RETR n
//    (
//       +OK m octets
//       <entire message>
//       .
//    )
// 7. Client schickt QUIT Kommando an Server, Server antwortet
//    QUIT
//    (+OK POP3 off)
//
// Compileraufruf
// icc pop3.cpp kontaktierServer.cpp datenSenden.cpp ws2_32.lib

#include <windows.h>
#include <stdio.h>

#include "kontaktierServer.h"
#include "datenSenden.h"

int datenEmpfangen ( int meinSocket, int mehrzeilig );



int main ( int argc, char *argv[] )
{
  char servername[300] = "popserver.com";
  int  meinSocket = 0;
  char nachricht[4096];

  WSADATA wsadata;

  // Ab hier geht's los...
  WSASetLastError(0);
  WSAStartup ( MAKEWORD(2,0),

  // Server auf Port 110 (POP3) kontaktieren...
  meinSocket = kontaktierServer ( servername, 110 );

  // Begrüssung des Servers empfangen
  datenEmpfangen ( meinSocket, 0 ); // 0: einzeilige Antwort wird erwartet

  // Benutzername (Anmeldename) senden
  memset ( nachricht, 0, sizeof(nachricht) );
  strcat ( nachricht, "USER Sascha\r\n\0" );
  datenSenden ( nachricht, strlen(nachricht), meinSocket );
  datenEmpfangen ( meinSocket, 0 );
```

```
  // Passwort senden
  memset ( nachricht, 0, sizeof(nachricht) );
  strcat ( nachricht, "PASS geheim\r\n\0" );
  datenSenden ( nachricht, strlen(nachricht), meinSocket );
  datenEmpfangen ( meinSocket, 0 );

  // STAT senden
  memset ( nachricht, 0, sizeof(nachricht) );
  strcat ( nachricht, "STAT\r\n\0" );
  datenSenden ( nachricht, strlen(nachricht), meinSocket );
  datenEmpfangen ( meinSocket, 0 );

  // LIST senden
  memset ( nachricht, 0, sizeof(nachricht) );
  strcat ( nachricht, "LIST\r\n\0" );
  datenSenden ( nachricht, strlen(nachricht), meinSocket );
  datenEmpfangen ( meinSocket, 1 ); // 1: mehrzeilige Antwort erwartet

  // RETR senden
  memset ( nachricht, 0, sizeof(nachricht) );
  strcat ( nachricht, "RETR 1\r\n\0" );
  datenSenden ( nachricht, strlen(nachricht), meinSocket );
  datenEmpfangen ( meinSocket, 1 ); // 1: mehrzeilige Antwort erwartet

  // QUIT senden
  memset ( nachricht, 0, sizeof(nachricht) );
  strcat ( nachricht, "QUIT\r\n\0" );
  datenSenden ( nachricht, strlen(nachricht), meinSocket );
  datenEmpfangen ( meinSocket, 0 );

  closesocket ( meinSocket );

  return 1;
} // Ende main()



int datenEmpfangen ( int meinSocket, int mehrzeilig )
{
   int numBytes=1, block=0, quit=0;
   char buffer[8192];

   do
   {
     memset ( buffer,0,sizeof(buffer) );
     if ( (numBytes=recv(meinSocket, buffer, sizeof(buffer), 0)) == -1)
     {
       printf("Fehler bei recv(): %d\n", WSAGetLastError());
       exit(-2);
     } // send()

     block++;
     if(numBytes!=0)
     {
       printf("Server: %s\n", buffer);
     } // numBytes!=0

     // STATUS Indikator entweder +OK oder -ERR
     if (block==1 &buffer[0]!='+')
     {
        printf("Fehler!\n");
        exit(-1000);
     }
```

```
      if (mehrzeilig==0)
      {
        if ( (buffer[numBytes - 2] == '\r') &(buffer[numBytes - 1] == '\n') )
        {
          quit=1;
        }
      } // Ende mehrzeilig==0
      else
      {
        if ( (buffer[numBytes - 5] == '\r') &(buffer[numBytes - 4] == '\n') &
             (buffer[numBytes - 3] == '.')  &
             (buffer[numBytes - 2] == '\r') &(buffer[numBytes - 1] == '\n') )
        {
          quit=1;
        }
      } // Ende mehrzeilig==1

    } // while()
    // Ende einer Multilineresponse durch "CRLF.CRLF"
    while ( quit==0 );


    return 1;
  } // Ende datenEmpfangen()
```

21.1.5 Einfacher HTTP Server

Der Quellcode des HTTP Servers stellt das Grundgerüst eines Servers dar, der lediglich eine Anfrage nach der anderen bearbeitet. Soll der Server mehrere Anfragen gleichzeitig bearbeiten, müssen mehrere Prozesse oder Threads erzeugt werden.

```
//
// TCP Server
//
// Compileraufruf
// icc server1.c ws2_32.lib
//

#include <windows.h>
#include <stdio.h>


void ProcessRequest(int threadNr, int clientSocket);



int main(int argc, char *argv[])
{
  int    meinSocket=0, clientSocket=0, sockaddr_in_Groesse=0;
  int    anzahlBytes=0, nummer=0;
  char   puffer[4096];
  struct sockaddr_in server, client;

  WSADATA wsadata;

  // Hier geht's los...
  WSASetLastError(0);
  WSAStartup(MAKEWORD(2,0),

  meinSocket=socket (AF_INET, SOCK_STREAM, 0);
```

```
  if (meinSocket==~0)
  {
     printf("Fehler bei socket(): %d\n", WSAGetLastError());
  }
  else
  {
     server.sin_port=htons(80);
     server.sin_family = AF_INET;
     server.sin_addr.s_addr = INADDR_ANY;
     if (bind(meinSocket, (struct sockaddr *)sizeof(server)) != 0)
     {
        printf("Fehler bei bind(): %d\n", WSAGetLastError());
        exit (-1);
     }

    // Groesse von sockaddr_in ermitteln, wird von accept benötigt
    sockaddr_in_Groesse=sizeof(struct sockaddr_in);

    // queue für eingehende Anfragen vorbereiten
    listen(meinSocket,20);

    printf("Warte auf Anfragen...\n\n");

    // auf Anfragen warten
    while(1)
    {
       if ((clientSocket=accept(meinSocket, (struct sockaddr*)sse))==-1)
       {
          printf("Fehler bei accept(): %d\n", WSAGetLastError());
          WSASetLastError(0);
       } // accept
       else
       {
          ProcessRequest(++nummer, clientSocket);
       }
    } // Ende while(1)

    closesocket(meinSocket);

  } // socket()

  return 1;
} // Ende main()



void ProcessRequest(int nummer, int clientSocket)
{
   int    sockaddrlaenge = 0, anzahlBytes = 0;
   struct sockaddr_in local;
   struct hostent *DNSclient; // DNS structure
   char   puffer[4096];

   printf("Anfrage %d an Socket %d\n", nummer, clientSocket);

   sockaddrlaenge=sizeof(local);
   getpeername((int)clientSocket, (struct sockaddr *)
   DNSclient = gethostbyaddr((char*),sizeof(struct in_addr),AF_INET);

   if (DNSclient==NULL)
   {
      printf("Client Adresse: %s:%d\n", inet_ntoa(local.sin_addr), ntohs(local.sin_port));
   }
   else
   {
```

```
      printf("Client Adresse: %s [%s:%d]\n", DNSclient->h_name, inet_ntoa(local.sin_addr), r
   }

   // Nachricht des Clients empfangen
   memset ( puffer, 0, sizeof(puffer) );
   anzahlBytes = recv(clientSocket, puffer, sizeof(puffer), 0);
   if (anzahlBytes==SOCKET_ERROR)
   {
      printf("Fehler bei recv(): %d\n", WSAGetLastError());
      return;
   }
   printf("Client Nachricht:\n%s\n", puffer);

   // Antwort an Client schicken
   memset ( puffer, 0, sizeof(puffer) );
   strcat ( puffer, "HTTP/1.0 200 OK\r\nServer: Test/1.0 (Nix)\r\n\r\nWillkommen bei unserem
   if ((anzahlBytes=send(clientSocket, puffer, strlen(puffer), 0)) < 0)
   {
      printf("Fehler bei send(): %d\n", WSAGetLastError());
      return;
   }
   closesocket(clientSocket);

   return;
} // Ende ProcessRequest
```

21.1.6 Einfacher Non−Blocking HTTP Server

Der Quellcode des HTTP Servers stellt das Grundgerüst eines Servers dar, der lediglich eine Anfrage nach der anderen bearbeitet. Soll der Server mehrere Anfragen gleichzeitig bearbeiten, müssen mehrere Prozesse oder Threads erzeugt werden.

Normalerweise blockieren Aufrufe wie `accept()` die Ausführung des Programmes, bis tatsächlich Daten vorliegen. Damit das Programm in der Zwischenzeit reagieren kann, z.B. auf so etwas einfaches wie einen Tastendruck zum Beenden des Servers, muss ein anderer Mechanismus gefunden werden. Besonders wichtig ist dies, wenn bei bereits bestehenden Verbindungen Daten gesendet/empfangen werden müssen, aber gleichzeitig auch neue Verbindungen entgegen genommen werden sollen.

Das folgende Beispiel nutzt einen relativ einfachen Mechanismus. In einer vorgegebenen Datenstruktur (`fd_set`) werden die Sockets aufgenommen, die überwacht werden sollen. In diesem Fall ist es nur ein Socket. Innerhalb der `while`−Schleife wird die Datenstruktur initialisiert und mit dem Socket bestückt. Danach wird dieser Socket mit der Funktion `select()` überprüft, ob Daten vorliegen. `select()` modifizert die Datenstrukturen, die der Funktion mitgegeben werden. `FD_ISSET()` prüft, ob Daten an einem bestimmten Socket vorliegen.

```
//
// Non−Blocking TCP Server
//
// Compileraufruf
// icc server2.c ws2_32.lib
//

#include <windows.h>
#include <stdio.h>
#include <conio.h>


void ProcessRequest(int threadNr, int clientSocket);
```

```
int main(int argc, char *argv[])
{
  int    meinSocket=0, clientSocket=0, sockaddr_in_Groesse=0;
  int    anzahlBytes=0, nummer=0;
  char   puffer[4096];
  struct sockaddr_in server, client;
  struct timeval tv;
  fd_set readfds;

  WSADATA wsadata;

  // Hier geht's los...
  WSASetLastError(0);
  WSAStartup(MAKEWORD(2,0),

  meinSocket=socket (AF_INET, SOCK_STREAM, 0);

  if (meinSocket==~0)
  {
     printf("Fehler bei socket(): %d\n", WSAGetLastError());
  }
  else
  {
     server.sin_port=htons(80);
     server.sin_family = AF_INET;
     server.sin_addr.s_addr = INADDR_ANY;
     if (bind(meinSocket, (struct sockaddr *)sizeof(server)) != 0)
     {
        printf("Fehler bei bind(): %d\n", WSAGetLastError());
        exit (-1);
     }

    // Groesse von sockaddr_in ermitteln, wird von accept benötigt
    sockaddr_in_Groesse=sizeof(struct sockaddr_in);

    // queue für eingehende Anfragen vorbereiten
    listen(meinSocket,20);

    printf("Warte auf Anfragen...\n\n");

    // auf Anfragen warten
    while(1)
    {
       // Timeout einstellen
       tv.tv_sec = 1;
       tv.tv_usec = 500000;
       // Descriptor-Set einstellen, muss vor jedem select() geschehen
       FD_ZERO(
       FD_SET(meinSocket,

       if (select(meinSocket+1, NULL, NULL, R)
       {
          printf("select() fehlgeschlagen\n");
          exit(-100);
       }
       // bei Tastendruck Server (unsauber) beenden
       if (_kbhit())
       {
          exit(10);
       }
       // stehen an meinSocket Daten bereit?
       if (FD_ISSET(meinSocket,
```

```
        {
           if ((clientSocket=accept(meinSocket, (struct sockaddr*)sse))==-1)
           {
              printf("Fehler bei accept(): %d\n", WSAGetLastError());
              WSASetLastError(0);
           } // accept
           else
           {
              ProcessRequest(++nummer, clientSocket);
           }
        } // FD_ISSET
     } // Ende while(1)

     closesocket(meinSocket);

   } // socket()

   return 1;
} // Ende main()



void ProcessRequest(int nummer, int clientSocket)
{
   int    sockaddrlaenge = 0, anzahlBytes = 0;
   struct sockaddr_in local;
   struct hostent *DNSclient; // DNS structure
   char   puffer[4096];

   printf("Anfrage %d an Socket %d\n", nummer, clientSocket);

   sockaddrlaenge=sizeof(local);
   getpeername((int)clientSocket, (struct sockaddr *)
   DNSclient = gethostbyaddr((char*),sizeof(struct in_addr),AF_INET);

   if (DNSclient==NULL)
   {
      printf("Client Adresse: %s:%d\n", inet_ntoa(local.sin_addr), ntohs(local.sin_port));
   }
   else
   {
      printf("Client Adresse: %s [%s:%d]\n\n", DNSclient->h_name, inet_ntoa(local.sin_addr),
   }

   // Nachricht des Clients empfangen
   memset ( puffer, 0, sizeof(puffer) );
   anzahlBytes = recv(clientSocket, puffer, sizeof(puffer), 0);
   if (anzahlBytes==SOCKET_ERROR)
   {
      printf("Fehler bei recv(): %d\n", WSAGetLastError());
      return;
   }
   printf("Client Nachricht:\n%s\n", puffer);

   // Antwort an Client schicken
   memset ( puffer, 0, sizeof(puffer) );
   strcat ( puffer, "HTTP/1.0 200 OK\r\nServer: Test/1.0 (Nix)\r\n\r\nWillkommen bei unserem
   if ((anzahlBytes=send(clientSocket, puffer, strlen(puffer), 0)) < 0)
   {
      printf("Fehler bei send(): %d\n", WSAGetLastError());
      return;
   }
   closesocket(clientSocket);

   return;
```

```
} // Ende ProcessRequest
```

21.2 IPv6 Programmierung

ACHTUNG! Hier fehlt noch eine Menge Inhalt...

Beispiel:

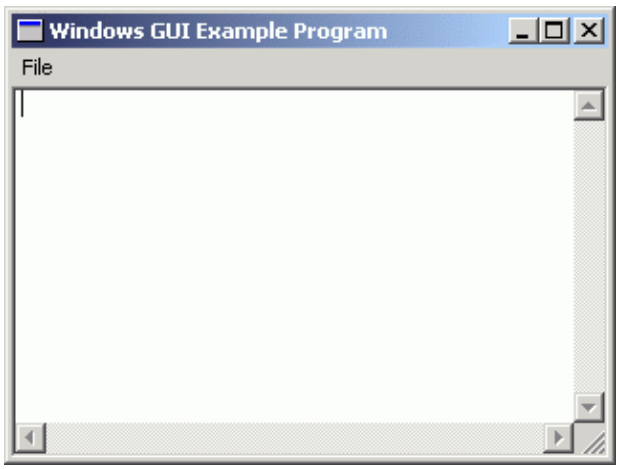
```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

void main(void)
{
   struct addrinfo *ai=NULL;
   int    res=0;
   int    sockfd=0;

   if ((res=getaddrinfo("www.ibm.com","http",0,
   {
      fprintf(stderr,"Kann keine Verbindung aufbauen: %s\n",gai_sterror(res));
      return 1;
   }

   while(ai)
   {
      if ((sockfd=socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol))>=0)
      {
         if (!connect(sockfd, ai->ai_addr, ai->ai_addrlen))
         {
            char buf[1024];
            FILE *f=fdopen(sockfd,"r+");

            fputs("GET / HTTP/1.0\r\nHost: www.ibm.com:80\r\n\r\n",f);
            while(fgets(buf,1000,f))
            {
               fputs(buf,stdout);

            }
            fclose(f);

            return 0;
         }
         else
         {
            close(sockfd);
         }
      }
      ai=ai->ai_next;
   } // Ende while(ai)

} // Ende main()
```

22.0 Graphische Anwendungen in C

ACHTUNG! Überarbeiten!

In diesem Abschnitt wird exemplarisch gezeigt, wie in C eine Anwendung mit graphischer Benutzeroberfläche für Windows aufgebaut ist.

Abbildung 22. Abbildung der Beispielanwendung GUI



22.1 Notwendige eigene Headerdateien

In der ersten Headerdatei `gui.h` werden Konstanten definiert, die den Menüeinträgen zugeordnet werden.

```
#define CM_FILE_EXIT  9070
#define CM_ABOUT      9071
```

Die zweite Headerdatei `gui.rc` enthält die Beschreibung der Ressourcen der Datei, in diesem Fall lediglich das Menü und die Menüeinträge.

```
#include "gui.h"

MAINMENU MENU
{
 POPUP "
 {
  MENUITEM "CM_ABOUT
  MENUITEM SEPARATOR
  MENUITEM "ECM_FILE_EXIT
 }
}
```

22.2 Hauptprogramm und Nachrichtenbehandlung

Bei Kommandozeilenprogrammen wird die Funktion `main()` vom Betriebsystem aufgerufen, bei Programmen für die graphische Oberfläche in MS Windows ist dies die Funktion `WinMain()`.

```
#include <windows.h>

#include "gui.h"

static char g_szClassName[] = "MyWindowClass";
static HINSTANCE g_hInst = NULL;

#define IDC_MAIN_TEXT   1001

LRESULT CALLBACK WndProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam)
{
   switch(Message)
   {
      case WM_CREATE:
         CreateWindow("EDIT", "",
            WS_CHILD | WS_VISIBLE | WS_HSCROLL | WS_VSCROLL | ES_MULTILINE | ES_WANTRETURN,
            CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
            hwnd, (HMENU)IDC_MAIN_TEXT, g_hInst, NULL);

         SendDlgItemMessage(hwnd, IDC_MAIN_TEXT, WM_SETFONT,
            (WPARAM)GetStockObject(DEFAULT_GUI_FONT), MAKELPARAM(TRUE, 0));
         break;
      case WM_SIZE:
         if(wParam != SIZE_MINIMIZED)
         {
            MoveWindow(GetDlgItem(hwnd, IDC_MAIN_TEXT), 0, 0, LOWORD(lParam), HIWORD(lParam),
         }
         break;
      case WM_SETFOCUS:
         SetFocus(GetDlgItem(hwnd, IDC_MAIN_TEXT));
         break;
      case WM_COMMAND:
         switch(LOWORD(wParam))
         {
            case CM_FILE_EXIT:
               PostMessage(hwnd, WM_CLOSE, 0, 0);
               break;
            case CM_ABOUT:
               MessageBox (NULL, "Windows GUI Example Program" , "About...", 0);
               break;
            default: break;
         }
         break;
      case WM_CLOSE:
         DestroyWindow(hwnd);
         break;
      case WM_DESTROY:
         PostQuitMessage(0);
         break;
      default:
         return DefWindowProc(hwnd, Message, wParam, lParam);
         break;
   }

   return 0;
} // End WndProc()


int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdSho
{
   WNDCLASSEX WndClass;
   HWND hwnd;
   MSG Msg;
```

```
    g_hInst = hInstance;

    WndClass.cbSize        = sizeof(WNDCLASSEX);
    WndClass.style         = 0;
    WndClass.lpfnWndProc   = WndProc;
    WndClass.cbClsExtra    = 0;
    WndClass.cbWndExtra    = 0;
    WndClass.hInstance     = g_hInst;
    WndClass.hIcon         = NULL;
    WndClass.hCursor       = LoadCursor(NULL, IDC_ARROW);
    WndClass.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    WndClass.lpszMenuName  = "MAINMENU";
    WndClass.lpszClassName = g_szClassName;
    WndClass.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    if(!RegisterClassEx(
    {
       MessageBox(0, "Window Registration Failed!", "Error!",
          MB_ICONEXCLAMATION | MB_OK | MB_SYSTEMMODAL);

       return 0;
    }

    hwnd = CreateWindowEx(
       WS_EX_CLIENTEDGE,
       g_szClassName,
       "Windows GUI Example Program",
       WS_OVERLAPPEDWINDOW,
       CW_USEDEFAULT, CW_USEDEFAULT, 320, 240,
       NULL, NULL, g_hInst, NULL);

    if(hwnd == NULL)
    {
       MessageBox(0, "Window Creation Failed!", "Error!",
          MB_ICONEXCLAMATION | MB_OK | MB_SYSTEMMODAL);

       return 0;
    }

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    while(GetMessage(NULL, 0, 0))
    {
       TranslateMessage(
       DispatchMessage(
    }

    return Msg.wParam;
 } // End WinMain()
```

22.3 Übersetzung des Programms

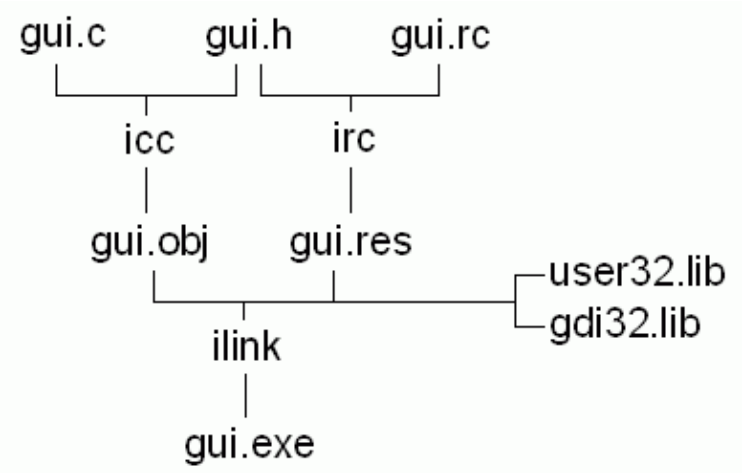
Zur Übersetzung des Programms sind folgende Schritte erforderlich:

1. Compilierung der Quellcodedatei (ohne Linkeraufruf!)
2. Compilierung der Ressourcendatei
3. Linken der Objekt− und Ressourcendateien.

Die Programmaufrufe sind von Compiler zu Compiler unterschiedlich. Zur Übersetzung des Programms mit dem IBM C/C++ Compiler sind folgende Kommandos notwendig:

1. `icc /c gui.c`
2. `irc gui.rc`
3. `ilink gui.obj gui.res user32.lib gdi32.lib`

Abbildung 23. Schaubild Übersetzung der Beispielanwendung GUI



23.0 Interaktion mit Java − Java Native Interface (JNI)

ACHTUNG! Hier fehlt noch eine Menge Inhalt...

Zur Interaktion von C mit Java gibt es verschiedene Möglichkeiten. Im folgenden werden zwei Varianten vorgestellt:

1. Zum einen kann C Programmcode aus Java heraus aufgerufen. Dazu wird das C Programm in Form einer DLL erzeugt, die bestimmten Konventionen folgen muss.

2. Zum anderen kann aus C Programmen heraus Java Programmcode aufgerufen werden. Hierzu wird innerhalb des C Programmes die Java Virtual Machine (VM) gestartet.

Bei beiden Varianten ist selbstverständlich der Austausch von Daten zwischen C und Java möglich.

23.1 C Programmcode in Java nutzen

Die Vorgehensweise zur Nutzung von C Programmcode in Java führt über den Umweg einer Dynamic Link Library (DLL, 17.0, "Dynamische Bibliotheken − DLL"):

1. Java−Klasse, die die native Methode deklariert und benutzt, schreiben und compilieren

   ```
   javac Klasse.java
   ```

2. Header−Datei für die native Methode mit "javah −jni" erstellen

   ```
   javah −jni Klasse
   ```

3. Native Methode in C implementieren und als DLL compilieren, z.B. IBM C/C++ Compiler:
   (der Dateiname des C−Quellcodes ist `dllcode.c`, die erzeugte DLL wird `dllcode.dll` heißen)

   ```
   icc /n2 /Ic:\j2sdk1.4.1\include;c:\j2sdk1.4.1\include\win32 /ge− /gd− dllcode.c /FEd
   ```

23.1.1 Beispiel einer Java−Klasse, die native Methoden deklariert und benutzt

Beispieldatei (`HelloWorld.java`):

```
class HelloWorld {

    //---> native code begins here
    // native Methode deklarieren
    // public, zwei Argumente vom Typ String, Rückgabewert vom Typ String
    // diese Methode muss später in C implementiert werden
    public native String displayHelloWorld(String text1, String text2);

    // native Bibliothek laden (DLL), Name "HelloWorld"
```

```
    // Name im Dateisystem libhello.so (Unix) bzw. hello.dll (Windows)
    static {
        System.loadLibrary("HelloWorld");
    }
    //---> native code ends here

    // main() ruft die native C-Methode auf
    public static void main(String[] args) {
        HelloWorld hW = new HelloWorld();
        System.out.println("Java: "+hW.displayHelloWorld("TestString1", "TestString2"));
    }
}
```

23.1.2 Beispiel einer Header-Datei

Der Inhalt der Header-Datei wird durch das Kommando :xphjavah generiert, diese Datei selbst zu erzeugen ist demnach nicht notwendig.

Beispieldatei (`HelloWorld.h`):

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    displayHelloWorld
 * Signature: (Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_HelloWorld_displayHelloWorld
  (JNIEnv *, jobject, jstring, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

23.1.3 Beispiel einer nativen Methode in C

Beim Schreiben der nativen Methoden muss die Defintion der Funktionen den Prototypen in der Header-Datei entsprechen. Den Prototyp in der Header-Datei kann man in den C-Quellcode kopieren und hinter die Typen die gewünschten Bezeichner setzen.

Bei der Übergabe von Daten ist zu beachten, dass diese von der Virtual Machine angefordert werden müssen und nach der Nutzung wieder freigegeben werden müssen. Dazu dienen die Aufrufe `GetStringUTFChars()` bzw. `ReleaseStringUTFChars()` (für Zeichenketten).

Beispieldatei (`HelloWorld.c`):

```
#include <jni.h>
```

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


JNIEXPORT jstring JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj, jstring jString1, jstring jString2)
{
    char buffer[1024];
    const char *string1 = (*env)->GetStringUTFChars(env, jString1, 0); // String1 von Java erha
    const char *string2 = (*env)->GetStringUTFChars(env, jString2, 0); // String2 von Java erha

    printf("C: %s\n", string1);                                        // String1 von Java in C
    printf("C: %s\n", string2);                                        // String2 von Java in C

    (*env)->ReleaseStringUTFChars(env, jString1, string1);             // String1 von Java frei
    (*env)->ReleaseStringUTFChars(env, jString2, string2);             // String2 von Java frei

    strcpy(buffer, "Dies ist ein String aus dem C Programm!\n");       // String von C an Java

    return (*env)->NewStringUTF(env, buffer);                          // String von C an Java
}
```

23.2 Java Programmcode in C nutzen

23.3 JNI Informationen von Sun

- JNI Programmer's Guide and Specification
  http://java.sun.com/docs/books/jni/
- JNI Tips
  http://java.sun.com/products/jdk/faq/jnifaq.html
- JNI SDK 1.4.1
  http://java.sun.com/j2se/1.4.1/docs/guide/jni/spec/jniTOC.doc.html
- JNI 1.1 Java Tutorial
  http://java.sun.com/docs/books/tutorial/native1.1/

Anhänge

Partial Table−of−Contents

Anhang A. Übersicht Schlüsselworte

Tabelle 12. Schlüsselworte

auto	Vereinbarung der Speicherklasse `auto` (siehe 4.6, "Speicherklassen")
break	Beendet die aktuelle Schleife und fährt mit der Programmausführung bei der nächsten Anweisung nach der Schleife fort. Beendet auch Anweisungszweige von `switch` Anweisungen. (siehe 6.4, "Die Anweisungen break und continue")
case	Anweisungszweig einer `switch` Anweisung. (siehe 5.2, "switch")
char	Datentyp für Zeichen und Zeichenketten. (siehe 4.1.3, "Textzeichen − char")
const	Vereinbarung einer Konstanten. (siehe 3.7, "Bezeichner")
continue	Bricht den aktuellen Schleifendurchlauf ab und springt zum Ende der Schleife. (siehe 6.4, "Die Anweisungen break und continue")
default	Anweisungszweig einer `switch` Anweisung, der ausgeführt wird, wenn kein `case` Zweig zutrifft bzw. vorher kein `break` auftrat. (siehe 5.2, "switch")
do	Beginnt eine `do while` Schleife, die Prüfung erfolgt am Ende eines Schleifendurchlaufs. (siehe 6.0, "Wiederholungen − Schleifen")
double	Vereinbarung einer rationellen Zahl. (siehe 4.1.2, "Rationale Zahlen / Fließkommazahlen")
else	Alternativzweig einer `if` Anweisung. (siehe 5.0, "Auswahl")
enum	Vereinbarung eines Aufzählungstyps. (siehe 4.3.2, "enum")
extern	Vereinbarung externer Bezeichner. (siehe 4.6, "Speicherklassen")
float	Vereinbarung einer rationellen Zahl. (siehe 4.1.2, "Rationale Zahlen / Fließkommazahlen")
for	Schleife mit Prüfung am Anfang eines Schleifendurchlaufs und integriertem Zähler. (siehe 6.0, "Wiederholungen − Schleifen")
goto	Unbedingter Sprung. (wird in diesem Handbuch nicht behandelt)
if	Abfrage einer Bedingung. (siehe 5.0, "Auswahl")
int	Vereinbarung einer ganzen Zahl. (siehe 4.1.1, "Ganze Zahlen − Integer")
long	Vereinbarung einer ganzen Zahl. (siehe 4.1.1, "Ganze Zahlen − Integer")
register	Vereinbarung einer Speicherklasse wie `auto`, jedoch Register statt Speicher, falls möglich. (siehe 4.6, "Speicherklassen")
return	Beendet Funktionen und übergibt ggf. einen Rückgabewert. (siehe 7.0, "Eigene Funktionen")
short	Vereinbarung einer ganzen Zahl. (siehe 4.1.1, "Ganze Zahlen − Integer")
signed	Vereinbarung einer ganzen Zahl oder eines Zeichens mit Vorzeichen. (siehe 4.1.1, "Ganze Zahlen − Integer" bzw. 4.1.3, "Textzeichen − char")
sizeof	Ermittlung der Länge eines Operanden. (siehe 3.6.7, "Hierarchie der Operatoren")
static	Vereinbarung einer Speicherklasse. Der Speicherbereich wird bei Programmbeginn zugeordnet. (siehe 4.6, "Speicherklassen")
struct	Vereinbarung einer Struktur. (siehe 4.2.2, "Strukturen")
switch	Beginn einer Auswahl. (siehe 5.2, "switch")
typedef	Definition eigener Datentypen. (siehe 4.5, "Eigene Typen − typedef" und 4.2.2.7, "Strukturen mit Typendefinitionen − typedef")
union	Vereinbarung verschiedener Datentypen im selben Speicherbereich. (siehe 4.2.3,

	"Unions")
unsigned	Vereinbarung einer ganzen Zahl oder eines Zeichens ohne Vorzeichen. (siehe 4.1.1, "Ganze Zahlen − Integer" bzw. 4.1.3, "Textzeichen − char")
void	Vereinbarung ohne Typ. (Nichts) (siehe 7.0, "Eigene Funktionen")
volatile	Vereinbarung eines unbeständigen Datentyps. Dieses Schlüsselwort teilt dem Compiler mit, daß die bezeichnete Variable durch Ereignisse außerhalb der Kontrolle des Programms verändert werden kann. Der Wert der Variablen muß deshlab vor jedem Zugriff neu aus dem Hauptspeicher gelesen werden, d.h. er darf nicht in einem Prozessorregister gespeichert werden. (Wird in diesem Handbuch nicht weiter behandelt.)
wchar_t	Datentyp für Zeichen und Zeichenketten in Zeichensätzen, bei denen ein Zeichen mehr als 8 Bit benötigt. (siehe 4.1.3, "Textzeichen − char")
while	Beginn einer `while` oder Ende einer `do while` Schleife. (siehe 5.0, "Auswahl")

Anhang B. Übersicht über die Datentypen

Tabelle 13. Wertebereiche der Datentypen unter OS/2

Typ	Größe	Wertebereich von	Wertebereich bis
char	1 Byte	−128	+127
unsigned char	1 Byte	0	+255
signed char	1 Byte	−128	+127
int	4 Byte	−2.147.483.648	+2.147.483.649
unsigned int	4 Byte	0	+4.294.967.296
signed int	4 Byte	−2.147.483.648	+2.147.483.649
short int	2 Byte	−32.768	+32.767
unsigned short int	2 Byte	0	+65.535
signed short int	2 Byte	−32.768	+32.767
long int	4 Byte	−2.147.483.648	+2.147.483.649
unsigned long int	4 Byte	0	+4.294.967.296
signed long int	4 Byte	−2.147.483.648	+2.147.483.649
long long int	8 Byte	0x8000000000000000	0x7fffffffffffffff
unsigned long long int	8 Byte	0	0xffffffffffffffff
signed long long int	8 Byte	0x8000000000000000	0x7fffffffffffffff
float	4 Byte	$3.4*10^{-38}$	$3.4*10^{+38}$
double	8 Byte	$1.7*10^{-308}$	$1.7*10^{+308}$
long double	16 Byte	$3.4*10^{-4932}$	$1.1*10^{+4932}$

Note: Der Datentyp `long long int` ist erst im ANSI C 99 Standard enthalten und deshalb nicht portabel. Er wird allerdings von vielen modernen Compilern unterstützt (z.B. IBM C and C++ Compilers for OS/2, AIX, and for Windows NT 3.6).

Anhang C. Übersicht Operatoren

C.1 Arithmetische Operatoren

Tabelle 14. Arithmetische Operatoren

Zweck	Operator
Addition	+
Subtraktion	−
Multiplikation	*
Division	/
Restwert	%

C.2 Zusammengesetzte Zuweisungsoperatoren

Tabelle 15. Zusammengesetzte Zuweisungsoperatoren

Operator	Beispiel	Äquivalenter Ausdruck
+=	index += 2	index = index + 2
−=	index −= 2	index = index − 1
*=	index *= 2	index = index * 2
/=	index /= 2	index = index / 2
%=	index %= 2	index = index % 2
<<=	result <<= num	result = rsult << num
>>=	form >>= 1	form = form >> 1
=	mask =2	mask = mask 2
^=	test ^= pre_text	test = test ^pre_test
|=	flag |= on	flag = flag | on

C.3 Unäre Operatoren

Tabelle 16. Unäre oder monadische Operatoren

Operator	Bedeutung
++	Wert inkrementieren
−−	Wert dekrementieren
!	logisches Nicht
~	bitweises Komplement
	Adreßoperator

*	Verweisoperator (Inhalt von)
(typ)	Typumwandler (type cast)
sizeof	Größenermittler

C.4 Vergleichende Operatoren

Tabelle 17. Vergleichende Operatoren

Ausdruck	Bedeutung
Ausdruck == Ausdruck	gleich
Ausdruck != Ausdruck	ungleich
Ausdruck < Ausdruck	kleiner als
Ausdruck <= Ausdruck	kleiner oder gleich
Ausdruck > Ausdruck	größer
Ausdruck >= Ausdruck	größer oder gleich

C.5 Logische Operatoren

Tabelle 18. Logische Zuweisungsoperatoren

Operator	Bedeutung
!	logisches NICHT
&&	logisches UND
||	logisches ODER

C.6 Hierarchie der Operatoren

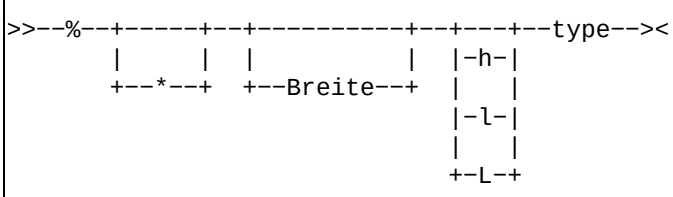
Tabelle 19. Tabelle der Hierarchie der Operatoren

Operatoren	Verwendung	Auswertungsrichtung
( ) [] −> .	bildet einen Ausdruck bezeichnet ein Feldelement wählt eine Strukturkomponente aus wählt eine Strukturkomponente per Zeiger aus	links nach rechts
++ −− − ! ~ & * sizeof (typ)	Inkrementieren Dekrementieren bildet negativen Wert logische Negierung bitweises Negieren liefert Adresse einer Variablen/Konstanten greift über Zeiger auf Variable/Konstante zu liefer Größe eines Speicherbereiches wandelt in angegebenen Typ um (type cast)	rechts nach links

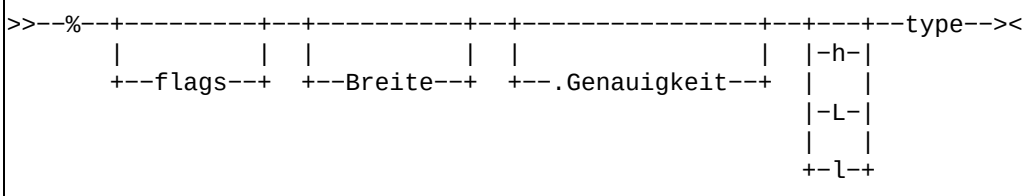
* / %	Multiplizieren Dividieren Restwert bilden	links nach rechts
+ −	Addieren Subtrahieren	links nach rechts
<< >>	verschiebt Bits nach links verschiebt Bits nach rechts	links nach rechts
< <= > >=	Vergleich auf kleiner Vergleich auf kleiner gleich Vergleich auf größer Vergleich auf größer gleich	links nach rechts
== !=	Vergleich auf gleich Vergleich auf nicht gleich	links nach rechts
&	Bitweises UND	links nach rechts
^	Bitweises XOR	links nach rechts
|	Bitweises ODER	links nach rechts
&	Logisches UND	links nach rechts
||	Logisches ODER	links nach rechts
? :	Bedingung	rechts nach links
= += −= *= /= <<= >>= &= ^= |= %=	einfache Zuweisung Addieren und Zuweisen Subtrahieren und Zuweisen Multiplizieren und Zuweisen Dividieren und Zuweisen Bits nach links verschieben und zuweisen Bits nach rechts verschieben und zuweisen Bitweise UND verknüpfen und zuweisen Bitweise XOR verknüpfen und zuweisen Bitweise ODER verknüpfen und zuweisen Restwert bilden und Zuweisen	rechts nach links
,	Operanden trennen	links nach rechts

Anhang D. Formatstring

scanf() − Formatspezifikation



printf() − Formatspezifikation



Die Formatspezifikationen beginnen mit einem Prozentzeichen (%) und enden mit dem Umwandlungszeichen (*type*; z.B. d, s, c, f oder e). Dazwischen können folgende Zeichen angeführt werden:

1. Folgende Steuerzeichen (*flags*) in beliebiger Reihenfolge
    a. '−': linksbündige Ausgabe des Datenelements
    b. '+': Zahlen werden immer mit Vorzeichen ausgegeben
    c. '0': leere Stellen im Ausgabefeld mit führenden/angehängten Nullen füllen
2. eine Zahl, die die minimale Feldbreite festlegt (bei Bedarf wird die Feldbreite vergrößert)
3. einen Punkt, der die Feldbreite von der Genauigkeit trennt und eine Zahl für die Genauigkeit mit folgender Bedeutung:
    a. maximale Anzahl von Zeichen für eine Zeichenkette
    b. Anzahl der Nachkommastellen für Fließkommazahlen (z.B. `%.0lf`)
    c. die minimale Anzahl von Ziffern für Ganzzahlenwerte
4. die Größe des Argumentes
    - h − Präfix für die Integertypen (d, i, n, o, u, x und X), der festlegt, daß das Argument `short int` oder `unsigned short int` ist.
    - l − Präfix für die Typen d, i, n, o, u, x und X, der festlegt, daß das Argument `long int` oder `unsigned long int`ist.
    - L − Präfix für die Typen e, f und g, der festlegt, daß das Argument `long double` ist.
5. Der *type* kann aus folgenden Zeichen bestehen:

c für char

d für signed int (dezimal)

e für float−Variablen, allerdings wird der Wert in Exponentenschreibweise ausgegeben.

E wie e, aber E für den Exponent statt e

f für float−Variablen, Ausgabe in Festkommaschreibweise

g für float−Variablen. Der Compiler entscheidet abhängig von der Länge, ob Exponential− oder Festkommadarstellung gewählt wird.

G wie g, aber E für den Exponent statt e

i für signed int (dezimal)

n Nummer der Zeichen, die bis jetzt erfolgreich auf die Ausgabe geschrieben wurde.

o für signed int (oktal)

p für Zeigervariablen

s für Zeichenketten (char−Felder)

u für unsigned int (dezimal)

x für unsigned int (hexadezimal, a−f)

X für unsigned int (hexadezimal, A−F)

Anhang E. Übersicht Fluchtsymbolzeichen

- \\ gibt den Backslash aus
- \' gibt das einfache Hochkomma (') aus
- \" gibt die Anführungszeichen bzw. das doppelte Hochkomma (") aus
- \0 Die binäre Null schließt eine Zeichenkette ab, wird von Funktionen zur Zeichenkettenbearbeitung automatisch generiert.
- \a *alarm*; gibt einen Warnton aus.
- \b *backspace*; bewegt den Cursor um eine Stelle nach links.
- \f *formfeed*; veranlaßt einen Seitenvorschub, d.h. daß der nachfolgende Text auf einer neuen Seite ausgegeben wird.
- \n markiert einen Zeilenvorschub (*new line*). Die nachfolgende Ein− oder Ausgabe wird in einer neuen Zeile begonnen.
- \r *carriage return*/Wagenrücklauf; bewegt den Cursor in derselben Zeile an den Zeilenanfang.
- \t setzt ein Tabulatorzeichen. Der folgende Text wird um einen definierten Wert, der in der Entwicklungsumgebung eingestellt wird, nach rechts verschoben.

Anmerkung: Um die Zeichen `\` *und* `%` *darzustellen, werden sie doppelt eingegeben.*
Bsp: `printf("Wechsel in das directory \"C:\\OS2\\APPS\" verläuft ...");`
ergibt folgende Ausgabe: `Wechsel in das directory "C:\OS2\APPS"` verläuft...

Anhang F. Übersicht C Funktionen zur Ein−/Ausgabe von Zeichen(−ketten)

fgetc

```
Syntax:     int fgetc(FILE *stream);
Bibliothek: stdio.h
Beispiel:   zeichen = fgetc(filein);
```

Von der mit stream verbundenen Datei wird ein Zeichen gelesen und zurückgegeben. Ist das Dateiende erreicht, wird ein negativer Wert zurückgeliefert.

fgets

```
Syntax:     char *fgets(char *buffer, int n, FILE *stream);
Bibliothek: stdio.h
Beispiel:   fgets(line, 256, filein);
```

Bis zum Erreichen von EOL werden alle Zeichen (inkl. \n), höchstens jedoch n−1 zeichen, aus der mit stream verbundenen Datei in den Puffer `buffer` eingelesen. Sind keine Fehler aufgetreten, liefert die Funktion einen Zeiger auf den Puffer zurück. Sind Fehler aufgetreten, liefert sie einen NULL−Zeiger zurück.

fprintf

```
Syntax:     int fprintf(FILE *stream, char *format,...);
Bibliothek: stdio.h
Beispiel:   fprintf(fileout, "%04d. Zeile: %s\n", linenr, line);
```

Gleiche Funktionsweise wie printf(), mit dem Unterschied, daß die Ausgabe auf die mit stream verbundene Datei erfolgt.

fputc

```
Syntax:     int fputc(int c, FILE *stream);
Bibliothek: stdio.h
Beispiel:   fputc('x', fileout);
```

Das Zeichen c wird auf die mit stream verbundene Datei geschrieben. Treten keine Fehler auf, wird das Zeichen c zurückgeliefert, ansonsten EOF.

fputs

```
Syntax:     int fputs(char *string, FILE *stream);
Bibliothek: stdio.h
Beispiel:   fputs(line, fileout);
```

Die Zeichenkette string wird (ohne das abschließende \0) in die mit stream verbundene Datei geschrieben. Treten Fehler auf, wird ein Wert ungleich 0 zurückgeliefert, ansonsten der Wert 0.

getc

```
Syntax:     int getc(FILE *stream);
Bibliothek: stdio.h
Beispiel:   zeichen = getc(filein);
```

Aus der mit stream verbundenen Datei wird das nächste Zeichen gelesen und zurückgegeben. Traten Fehler auf, wird EOF zurückgegeben.

getch

```
Syntax:     int getch(void);
Bibliothek: conio.h
Beispiel:   zeichen = getch();
```

Liest ein Zeichen von der Tastatur ein und gibt es zurück. Das Programm wird fortgeführt, nachdem eine Taste gedrückt wurde. Das eingegebene Zeichen erscheint nicht auf dem Bildschirm.

getchar

```
Syntax:     int getchar(void);
Bibliothek: stdio.h
Beispiel:   zeichen = getchar();
```

Von stdin (i.d.R. die Tastatur) wird das nächste Zeichen gelesen und zurückgegeben. Treten Fehler auf, wird EOF zurückgeliefert. Die Tastenkombination CTRL−Z (oder STRG−Z) wird als EOF interpretiert.

getche

```
Syntax:     int getche(void);
Bibliothek: conio.h
Beispiel:   zeichen = getche();
```

Liest ein Zeichen von der Tastatur, zeigt es auf dem Bildschirm an und gibt es zurück. Das Programm wird fortgeführt, nachdem eine Taste gedrückt wurde.

gets

```
Syntax:     char *gets(char *buffer);
Bibliothek: stdio.h
Beispiel:   gets(line);
```

Von stdin (i.d.R. Tastatur) werden solange Zeichen gelesen, bis \n (Return) oder EOF eingegeben wird. Die Zeichen werden in `buffer` gespeichert. Treten Fehler auf, wird ein NULL−Zeiger zurückgegeben, ansonsten ein Zeiger auf `buffer`.

printf

```
Syntax:     int printf(char *format,...);
Bibliothek: stdio.h
Beispiel:   printf("%04d. Zeile: %s\n",linenr,line);
```

Dient der formatierten Ausgabe von Daten auf stdout (i.d.R. der Bildschirm). Treten Fehler auf, wird −1 zurückgegeben, ansonsten die Anzahl Zeichen, die auf die Ausgabe geschrieben wurden.

putc

```
Syntax:     int putc(int c, FILE *stream);
Bibliothek: stdio.h
Beispiel:   putc(zeichen, fileout);
```

Das Zeichen c wird in die mit stream verbundene Datei geschrieben. Treten Fehler auf, wird EOF zurückgegeben, ansonsten das Zeichen c.

putchar

```
Syntax:     int putchar(int c);
Bibliothek: stdio.h
Beispiel:   putchar(zeichen);
```

Das Zeichen c wird auf die Standardausgabe geschrieben. Treten Fehler auf, wird EOF zurückgeliefert, ansonsten das Zeichen c.

puts

```
Syntax:     int puts(char *string);
Bibliothek: stdio.h
Beispiel:   puts(line);
```

Die Zeichenkette string wird auf die Standardausgabe geschrieben und der Cursor am Anfang der nächsten Zeile positioniert. Treten Fehler auf, wird ein Wert ungleich 0 zurückgegeben, ansonsten der Wert 0.

scanf

```
Syntax:     int scanf(char *format,...);
Bibliothek: stdio.h
Beispiel:   scanf("%d",
```

Von der Standardeingabe (i.d.R. Tastatur) werden Werte gemäß der Formatspezifikation eingelesen. Treten Fehler auf, wird EOF zurückgegeben, ansonsten die Anzahl Zeichen, die eingelesen wurden.

sscanf

```
Syntax:     int sscanf(char *string, char *format,...);
Bibliothek: stdio.h
Beispiel:   sscanf(string,"%d",
```

Gleiche Funktionsweise wie scanf(), mit dem Unterschied, daß die Werte aus einer Zeichenkette gelesen werden, auf die `string` weist.

Anhang G. C Funktionen der Standardbibliotheken

Tabelle 20. Ein−/Ausgabefunktionen

clearerr	setzt EOF− und Fehlerindikator einer Datei zurück	stdio.h
fclose	schließt eine Datei	stdio.h
feof	testet auf EOF	stdio.h
ferror	testet auf Dateifehler	stdio.h
fflush	erzwingt das Speichern des Puffers in eine Datei	stdio.h
fgetc	liest ein einzelnes Zeichen aus einer Datei	stdio.h
fgets	liest eine Zeile aus einer Datei in einen Puffer	stdio.h
fopen	öffnet eine Datei in einem bestimmten Modus	stdio.h
fprintf	schreibt Daten formatiert in eine Datei	stdio.h
fputc	schreibt ein einzelnes Zeichen in eine Datei	stdio.h
fputs	schreibt eine Zeichenkette in eine Datei	stdio.h
fread	liest mehrere Sätze aus einer Datei in einen Puffer	stdio.h
freopen	ordnet einem Stream (Dateizeiger) eine neue Datei zu	stdio.h
fscanf	liest Daten formatiert aus einer Datei	stdio.h
fseek	bewegt den Dateizeiger	stdio.h
ftell	ermittelt die aktuelle Position des Dateizeigers	stdio.h
fwrite	schreibt mehrere Sätze aus einem Puffer in eine Datei	stdio.h
getc	liest ein Zeichen von einem Stream	stdio.h
getch	liest ein Zeichen von der Tastatur	conio.h
getchar	liest ein Zeichen von der Standardeingabe	stdio.h
getche	liest ein Zeichen von der Tastatur und gibt dieses auf dem Bildschirm aus	conio.h
gets	liest eine Zeile von der Standardeingabe in einen Puffer	stdio.h
perror	schreibt eine Meldung auf stderr	stdio.h
printf	schreibt Daten formatiert auf die Standardausgabe	stdio.h
putc	schreibt ein Zeichen auf einen stream	stdio.h
putchar	schreibt ein Zeichen auf die Standardausgabe	stdio.h
puts	schreibt eine Zeile auf die Standardausgabe	stdio.h
remove	löscht eine Datei	stdio.h
rename	ändert den Namen einer Datei	stdio.h
rewind	setzt den Dateizeiger auf den Anfang zurück	stdio.h
scanf	liest Daten formatiert von der Standardeingabe	stdio.h
setbuf	definiert einen Ein−/Ausgabe−Puffer für eine Datei	stdio.h
setvbuf	definiert die Art der Pufferung für eine Datei	stdio.h
sprintf	schreibt Daten formatiert an eine Speicherstelle	stdio.h
sscanf	liest Daten formatiert aus einer Zeichenkette	stdio.h
tmpfile	erzeugt und öffnet eine temporäre Datei	stdio.h
tmpnam	erzeugt einen temporären Dateinamen	stdio.h
ungetc	Rückgängigmachen eines vorherigen getc	stdio.h

Tabelle 21. mathematische Funktionen

abs	ermittelt den Absolutwert	stdlib.h
acos	ermittelt den Arcuscosinus	math.h
asin	ermittelt den Arcussinus	math.h
atan	ermittelt den Arcustangens	math.h
atan2	ermittelt den Arcustangens eines Quotienten	math.h
ceil	rundet positive Zahlen auf, negative ab	math.h
cos	ermittelt den Cosinus	math.h
cosh	ermittelt den hyperbolischen Cosinus	math.h
exp	dient als Exponentialfunktion e^x	math.h
fabs	ermittelt den Absolutwert einer Fließkommazahl	math.h
floor	rundet positive Zahlen ab, negative auf	math.h
fmod	ermittelt den Rest einer Fließkommaoperation	math.h
frexp	bestimmt für float−Werte Mantisse und Exponent zur Basis 2	math.h
ldexp	berechnet das Produkt aus Mantisse und einer Zweierbasis	math.h
log	ermittelt den natürlichen Logarithmus	math.h
log10	ermittelt den dekadischen Logarithmus	math.h
pow	berechnet das Ergebnis von Potenzen	math.h
rand	erzeugt eine Zufallszahl	stdlib.h
sin	ermittelt den Sinus	math.h
sinh	ermittelt den hyperbolischen Sinus	math.h
sqrt	berechnet die Quadratwurzel	math.h
srand	initialisiert den Zufallsgenerator	stdlib.h
tan	ermittelt den Tangens	math.h
tanh	ermittelt den hyperbolischen Tangens	math.h

Tabelle 22. Speicherverwaltungsfunktionen

calloc	reserviert Speicherplatz für Felder	stdlib.h
free	gibt Speicherplatz frei	stdlib.h
malloc	reserviert Speicherplatz in Bytes	stdlib.h
realloc	verändert die Größe eines Speicherblockes	stdlib.h
longjmp	führt Rücksprung zur setjmp−Stelle durch	setjmp.h
setjmp	markiert Rücksprung−Stelle für longjmp	setjmp.h

Tabelle 23. Test integer values (ohne deutsche Umlaute)

isalnum	prüft auf alphanumerisch	ctype.h
isalpha	prüft auf alphabetisch	ctype.h
iscntrl	prüft auf Steuerzeichen	ctype.h
isdigit	prüft auf Ziffern	ctype.h

isgraph	prüft auf Grafikzeichen	ctype.h
islower	prüft auf Kleinbuchstaben	ctype.h
isprint	prüft auf druckbare Zeichen	ctype.h
ispunct	prüft auf Satzzeichen	ctype.h
isspace	prüft auf Trennzeichen	ctype.h
isupper	prüft auf Großbuchstaben	ctype.h
isxdigit	prüft auf Hexadezimalziffern	ctype.h

Tabelle 24. Umgebungssteuerung

abort	bricht das Programm mit einer Fehlermeldung ab	stdlib.h
assert	ermöglicht den Programmabbruch, wenn ein Testergebnis fehlschlägt	assert.h
exec	lädt und startet ein Programm	process.h
exit	beendet ein Programm ordnungsgemäß	stdlib.h
getenv	ermittelt die Werte der Umgebungsvariablen	stdlib.h
int86	Interruptaufruf auf x86er−Systemen (z.B. DOS)	dos.h
putenv	setzt eine neue Umgebungsvariable oder ändert den Wert einer bestehenden	stdlib.h
raise	sendet Signale an das Programm	signal.h
signal	Funktion zur Behandlung von Interrupts des Betriebssystems	signal.h
spawn	lädt und startet ein Programm	process.h
system	Betriebssystemkommandos ausführen	stdlib.h

Tabelle 25. Umwandlungsfunktionen

atof	konvertiert ASCII nach float	stdlib.h
atoi	konvertiert ASCII nach int	stdlib.h
atol	konvertiert ASCII nach long	stdlib.h
itoa	konvertiert int nach ASCII	stdlib.h
strtod	konvertiert Zeichenkette nach double	stdlib.h
strtol	konvertiert Zeichenkette nach long	stdlib.h
strtoul	konvertiert Zeichenkette nach unsigned long	stdlib.h
tolower	konvertiert in Kleinbuchstaben	ctype.h
toupper	konvertiert in Großbuchstaben	ctype.h

Tabelle 26. Zeichenketten− und Speicherfunktionen

bsearch	führt binäre Suche in einem sortierten Feld durch	stdlib.h
memchr	sucht in einem bestimmten Speicherbereich nach einem Zeichen	string.h
memcmp	vergleicht zwei Speicherbereiche miteinander	string.h
memset	initialisiert einen Speicherbereich	string.h
memcpy	kopiert n Bytes innerhalb des Speichers und liefert einen Zeiger auf das Ziel zurück	string.h
qsort	sortiert ein Feld nach dem QuickSort−Algorithmus	stdlib.h

strcat	verkettet zwei Zeichenketten miteinander	string.h
strchr	sucht ein Zeichen innerhalb einer Zeichenkette	string.h
strcmp / stricmp	vergleicht zwei Zeichenketten miteinander	string.h
strcpy	kopiert eine Zeichenkette in einen anderen Speicherbereich	string. h
strcspn	ermittelt die Anzahl Zeichen einer Zeichenkette bis zu einem Begren zer	string.h
strlen	ermittelt die Länge einer Zeichenkette	string.h
strncat	verkettet eine Zeichenkette mit einer Teilzeichenkette	string.h
strncmp / strnicmp	vergleicht n Zeichen zweier Zeichenketten	string.h
strncpy	kopiert eine Teilzeichenkette in einen anderen Speicherbereich	string.h
strpbrk	ermittelt die Position des ersten Begrenzerzeichens	string.h
strrchr	sucht von rechts nach links in einer Zeichenkette nach einem einzelnen Zeichen	string.h
strspn	sucht die Position eines nicht zur Vorgabemenge gehörigen Zeichens	string.h
strstr	sucht eine Teilzeichenkette in einer Zeichenkette	string.h
strtok	führt eine Token−Suche in einer Zeichenkette durch	string.h

Tabelle 27. Zeitfunktionen

asctime	konvertiert die Time−Struktur in eine Zeichenkette	time.h
ctime	liefert eine Zeichenkette mit Datum und Uhrzeit	time.h
difftime	liefert die Zeitdifferenz in Sekunden	time.h
gmtime	konvertiert die Time−Struktur in "Greenwich mean time"	time.h
localtime	konvertiert die Time−Struktur in "Local time"	time.h
time	aktualisiert die Time−Struktur	time.h

Anhang H. Übersicht über die C Include−/Header−Dateien

Diese Übersicht enthält u.U. nicht alle Funktionsnamen, die in den jeweiligen Dateien enthalten sind.

assert.h

Die Datei assert.h definiert nur das Test−Makro assert.

ctype.h

ctype.h enthält u.a. die folgenden Funktionsprototypen, die Integerwerte auf bestimmte Kennzeichen hin untersuchen:

```
isalnum     isdigit     isprint     isupper     toupper
isalpha     isgraph     ispunct     isxdigit
iscntrl     islower     isspace     tolower
```

float.h

Die Datei float.h enthält u.a. Definitionen der Fließkommakonstanten, z.B. den Wertebereich der einzelnen Fließkommadatentypen.

limits.h

Die Datei limits.h enthält u.a. Definitionen für integer und character.

math.h

Die Datei math.h enthält u.a. Funktionsprototypen für mathematische Funktionen und Makros:

```
acos        ceil        floor        log10        sqrt
asin        cos         fmod         modf         tan
atan        cosh        frexp        pow          tanh
atan2       exp         ldexp        sin
bessel      fabs        log          sinh
```

setjmp.h

Diese Datei definiert Prototypen für die Funktionen `setjmp` und `longjmp` sowie einen Buffertyp, den diese beiden Funktionen zum Speichern von Daten benötigen.

signal.h

Diese Datei enthält u.a. Prototypen für die Funktionen `signal` und `raise` sowie Wertzuordnungen für einzelne Signale.

stdarg.h

Diese Datei enthält u.a. Makrodefinitionen, die es ermöglichen, auf Parameter in Funktionen mit variablen Parameterlisten zuzugreifen. Die dazugehörigen Funktionen sind `va_arg, va_start` und `va_end`.

stdio.h

Diese Datei enthält u.a. Prototypen und Makrodefinitionen für die Ein− und Ausgabe von Daten. Sie umfaßt folgende Funktionen:

```
clearerr      fprintf      fwrite       puts         sscanf
fclose        fputc        getc         remove       tmpfile
feof          fputs        gets         rename       tmpnam
ferror        fread        getchar      rewind       ungetc
fflush        freopen      perror       scanf        vfprintf
fgetc         fscanf       printf       setbuf       vprintf
fgets         fseek        putc         setvbuf      vsprintf
fopen         ftell        putchar      sprintf
```

stdlib.h

Diese Datei enthält u.a. Prototypen für folgende Funktionen:

```
abort         atol         free         qsort        strtod
abs           bsearch      getenv       rand         strtol
atof          calloc       labs         realloc      system
atoi          exit         malloc       srand
```

string.h

Diese Datei enthält u.a. Prototypen für die Funktionen zur Zeichenkettenbearbeitung.

```
memchr        strcat       strcspn      strncpy      strstr
memcmp        strchr       strlen       strpbrk      strtok
memcpy        strcmp       strncat      strrchr
memset        strcpy       strncmp      strspn
```

time.h

Diese Datei beinhaltet die Prototypen für folgende Funktionen:

```
              asctime      difftime     localtime
              ctime        gmtime       time
```

Des weiteren befindet sich in dieser Datei die Definition der Struktur `tm`, die von einigen der oben genannten Funktionen benutzt wird.

Anhang I. Bedienung der Compiler Tools

In diesem Kapitel werden Informationen zu einigen Compilern bereitgehalten. Der IBM Compiler ist zwar etwas mühsam in der Installation und auch alles andere als klein, verfügt jedoch über eine excellente Online−Hilfe. Keiner der hier vorgestellten Compiler verfügt über eine integrierte Entwicklungsumgebung, bringt also von Haus aus keinen Editor mit.

I.1 IBM C/C++ Compiler V3.6

Dieser Compiler für Windows, OS/2 und AIX ist IBM intern unter ftp://ftp3.torolab.ibm.com/pub/vac++/ verfügbar. Die Windows Version befindet sich im Verzeichnis win−v3.6.5, die OS/2 Version im Verzeichnis os2−v3.6.5 und die AIX Version im Verzeichnis aix−v3.6. Bei der Installation unter Windows sind unbedingt einige wesentliche Schritte zu beachten, siehe unten.

Die folgenden Abschnitte behandeln

- die Installation unter Windows 95 bzw. die Installation unter Windows 2000,
- das Testen der Installation,
- Konfiguration und Test der Online Hilfe und
- den Compileraufruf.

I.1.1 Installation unter Windows 95

1. Installation mittels SETUP.EXE starten
   (in BAB Berlin unter `S:\C\Compiler\IBM C 3.6\Setup.exe`)
2. Next
3. Installationsverzeichnis C:\ibmcxxw beibehalten; ca. 220MB freier Platz erforderlich
4. Custom
5. Next
6. "IBM OpenClass Source" abwählen, alles andere beibehalten
7. Next
8. − Moment warten −
9. Next
10. Next
11. − Eine endliche Zeit lang warten −
12. .Am Ende erscheint u.U. die Meldung "Error during the NetQuestion Search Server initialization program. Please refer to the Search Server release notes."
13. Ok
14. Nein, ich möchte die ReadMe jetzt nicht lesen
15. Finish
16. Finish
17. Alle offenen Anwendungen schließen und Windows neu starten
18. My Computer öffnen
19. Installationslaufwerk (i.d.R. C:) öffnen
20. Ordner IBMCXXW öffnen
21. Ordner NTQ öffnen
22. Ordner INSTALL öffnen
23. Datei INITIMN.BAT mit der rechten Maustaste anklicken
24. Properties
25. Memory
26. Initial Environment auf 4096 setzen

27. Ok
28. Start Menu öffnen
29. Settings
30. Taskbar..
31. Lasche Start Menu Programs auswõhlen
32. Advanced
33. Ordner Programs öffnen
34. Ordner IBM C and C++ Compilers öffnen
35. Icon Command Line mit der rechten Maustaste auswählen
36. Properties
37. Lasche Programm auswählen
38. Bei Cmd Line den Text CMD.EXE durch COMMAND.COM ersetzen
39. Lasche Memory auswählen
40. Initial Environment auf 4096 setzen
41. Ok
42. My Computer öffnen
43. Installationslaufwerk (i.d.R. C:) ÷ffnen
44. Ordner IBMCXXW öffnen
45. Ordner BIN öffnen
46. SETENV.BAT mit der rechten Maustaste auswählen
47. EDIT
48. Vor der Zeile START %1 %2 %3 %4 %5 %6 %7 %8 %9 eine Zeile DOSKEY einfügen
49. In der Zeile START %1 %2 %3 %4 %5 %6 %7 %8 %9 den Text START entfernen
50. Menü File−>Save
51. Start Menü öffnen
52. Programs−>IBM C/C++ Compilers−>Command line
53. Folgendes Kommando eingeben:
    `C:\ibmcxxw\ntq\install\initimn.bat C: C:\ibmcxxw`
    (großgeschriebene Laufwerksbuchstaben ggf. an Installationslaufwerk anpassen)
54. − Moment warten −

Nun sollten unbedingt noch die beiden Abschnitte I.1.3, "Testen der Installation" und I.1.4, "Konfiguration und Test der Online Hilfe" durchgelesen und befolgt werden.

I.1.2 Installation unter Windows 2000

1. Installation mittels SETUP.EXE starten
   (in BAB Berlin unter `S:\C\Compiler\IBM C 3.6\Setup.exe`)
2. Next
3. Installationsverzeichnis C:\ibmcxxw beibehalten; ca. 220MB freier Platz erforderlich
4. Custom
5. Next
6. "IBM OpenClass Source" und den Debugger abwählen, alles andere beibehalten
   Wer einen Debugger benötigt findet unter http://oltdbg.torolab.ibm.com/debugger/ den IBM Distributed Debugger, der u.a. C/C++ und Java versteht.
7. Next
8. − Moment warten −
9. Next
10. Next
11. − Eine endliche Zeit lang warten −
12. Am Ende erscheint u.U. die Meldung "Error during the NetQuestion Search Server initialization program. Please refer to the Search Server release notes."
13. Ok
14. Nein, ich möchte die ReadMe jetzt nicht lesen

15. Finish
16. Finish
17. Alle offenen Anwendungen schließen und Windows neu starten

Nun sollten unbedingt noch die beiden Abschnitte I.1.3, "Testen der Installation" und I.1.4, "Konfiguration und Test der Online Hilfe" durchgelesen und befolgt werden.

I.1.3 Testen der Installation

1. Start Menü öffnen
2. Programs
3. IBM C and C++ Compilers
4. Command Line
5. ICC eingeben
6. wenn jetzt NICHT die Compilerhilfe erscheint, wurde etwas falsch gemacht...

I.1.4 Konfiguration und Test der Online Hilfe

Wenn der Compiler funktioniert muss noch die Hilfe lauffähig gemacht werden. Zunächst die Anleitung für den Netscape Navigator, weiter unten für den Internet Explorer:

1. Netscape Navigator starten
2. Menu Edit−>Preferences
3. Unterpunkt Advanced aufklappen
4. Proxies auswählen
5. Manual proxy configuration auswählen
6. View...
7. Im Feld "Do not use proxy servers for domains beginning with:" folgendes eingeben:
   `localhost:49213, 127.0.0.1:49213`
   (falls dort bereits etwas stand, einfach mit einem Komma anhängen)
8. Ok
9. Ok
10. ALLE Netscape Fenster schließen (Hilfe erscheint u.U. nur, wenn Netscape nicht geöffnet ist)
11. Start Menü öffnen
12. Programs
13. IBM C and C++ Compilers
14. Help Home Page
15. − Moment warten −
16. wenn jetzt NICHT die Onlinehilfe erscheint, wurde etwas falsch gemacht...

Anleitung für die Online Hilfe mit dem Internet Explorer

1. Menü "Tools"
2. Menüpunkt "Internet Options..."
3. Lasche "Connections"
4. "LAN Settings"
5. bei "Proxy Server" auf "Advanced"
6. In der Box "Exceptions" folgendes hinter die bestehenden Einträge setzen
   `localhost:49213; 127.0.0.1:49213` br (falls dort bereits etwas stand, einfach mit einem Semikolon anhängen)
7. Ok
8. Ok
9. Lasche "Advanced"
10. Bei "HTTP 1.1 settings" sowohl "Use HTTP 1.1" als auch

11. "Use HTTP 1.1 through proxy connections" deaktivieren
12. Ok
13. Start Menü öffnen
14. Programs
15. IBM C and C++ Compilers
16. Help Home Page
17. − Moment warten −
18. wenn jetzt NICHT die Onlinehilfe erscheint, wurde etwas falsch gemacht...

I.1.5 Compileraufruf

Der IBM C/C++ Compiler wird durch Eingabe von `icc` in der eigens dafür vorgesehenen Kommandozeile gestartet. In der Regel genügt es, auf `icc` den Namen der Quellcodedatei folgen zu lassen. Entdeckt der Compiler keine syntaktischen Fehler, wird automatisch der Linker ILINK aufgerufen.
Beispiele für Kommandozeilenschalter:

```
ICC xyz.c               // compilieren und linken
ICC xyz.c abc.obj       // compilieren, linken und abc.obj linken
ICC /c xyz.c            // xyz.c nur compilieren, keine .exe (o.ä.) erzeugen
ICC /N3 xyz.c           // nach 3 Fehlern abbrechen
ICC /Wall+              // sämtliche Warnungen ausgeben, die möglich sind
ICC /?                  // Hilfe
ICC /Fc+                // nur Syntax Check
ICC /Ti+                // Debug information generieren
ICC /Wgrp               // bestimmte Gruppen von Warnungen ausgeben (siehe Onlinehilfe)
ICC /Sa                 // ANSI konform
ICC /Ge−                // DLL anstatt EXE erzeugen
ICC /Gm+                // Multithreading Libraries binden
```

Wenn gleichzeitig mehrere Quellcodes compiliert werden sollen, werden die Namen der Quellcodedateien einfach durch Leerzeichen voneinander getrennt angegeben.

Wenn eine Quellcodedatei compiliert und mit einer Objektdatei gelinkt werden soll, wird der Name der Objektdatei einfach durch ein Leerzeichen vom Quellcodedateinamen abgetrennt angegeben.

```
ICC datei1.c datei2.c datei3.c      // 3 Dateien compilieren, datei1.exe wird erzeugt

ICC datei1.c wincursor.obj          // datei1.c compilieren, mit wincursor.obj linken,
                                    //   datei1.exe wird erzeugt
```

I.2 Borland C++Builder Compiler

Borland hat den Compiler seines Produktes C++Builder "frei" zur Verfügung gestellt. Was Borland unter "frei" versteht ist der dem Compiler beigefügten Lizenzvereinbarung zu entnehmen. Über einen Registrierungsprozess kann der Compiler unter http://www.borland.com/bcppbuilder/freecompiler/ geladen werden.

Aus Lizenzrechtlichen Gründen darf dieser Compiler nicht auf Servern innerhalb der IBM zur Verfügung gestellt werden.

Nach der Installation kann der Compiler innerhalb einer Kommandozeile mit `BCC32` gestartet werden. Wenn die Compilierung erfolgreich war wird automatisch der Linker gestartet.

Beispiele für Kommandozeilenschalter:

```
BCC32 xyz.c                // compilieren und linken
BCC32 xyz.c abc.obj        // compilieren, linken und abc.obj linken
BCC32 -c xyz.c             // nur compilieren
BCC32 -j3 xyz.c            // nach 3 Fehlern abbrechen
BCC32                      // Hilfe
BCC32 -v                   // Debug information generieren
```

I.3 MinGW (GCC Windows)

MinGW ist eine Minimalvariante des GCC Compilers für Windows. Homepage: http://www.mingw.org/

Die Datei MinGW−1.1.tar.gz (bzw. die jeweils aktuelle Variante) laden und in ein Verzeichnis ohne Leerzeichen im Namen entpacken. Diese Datei enthält alle benötigten Dateien. Das Verzeichnis bin muß in den Pfad aufgenommen werden.
Beispiele für Kommandozeilenschalter:

```
gcc -o xyz xyz.c           // compilieren und linken
gcc -c hello.c             // nur compilieren
```

Anhang J. Rechnung mit Bits und Bytes

Eine binäre Angabe lässt sich mittels einer einfachen Tabelle problemlos in eine Dezimalzahl übersetzen.

```
Bitposition      7    6    5    4   3   2   1   0
Bitwert        128   64   32   16   8   4   2   1
```

Beispiel

```
Bitposition      7    6    5    4   3   2   1   0
Bitwert        128   64   32   16   8   4   2   1


                 1    0    1    0   1   0   1   0
               128 +  0 + 32 +  0 + 8 + 0 + 2 + 0

Ergebnis:                                  170
```

Bei `signed` Variablen und Fließkommazahlen wird das Bit links außen, das sog. Sign Bit, als Vorzeichen verwendet.

Anhang K. Zeichensätze

K.1 Standard ASCII

Siehe auch http://www.asciitable.com/

Tabelle 28. Standard ASCII Zeichensatz

	0	1	2	3	4	5	6	7	8	9
000	NULL							BEL	???	TAB
010	LF	VT	FF	CR			>	<		!!
020	¶	§			^	V	>	<	*	
030		???	Blank	!	"	#	$	%	&	'
040	(	)	*	+	,	−	.	/	0	1
050	2	3	4	5	6	7	8	9	:	;
060	<	=	>	?	@	A	B	C	D	E
070	F	G	H	I	J	K	L	M	N	O
080	P	Q	R	S	T	U	V	W	X	Y
090	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

K.2 Extended ASCII

Tabelle 29. Extended ASCII Zeichensatz

−?− nicht darstellbar; ??? unbekannt										
	0	1	2	3	4	5	6	7	8	9
120									Ç	ü
130	é	â	ä	à	å	ç	ê	ë	è	ï
140	î	ì	Ä	Å	É	æ	Æ	ô	ö	ò
150	û	ù	_	Ö	Ü	ø	£	¥	_	
160	á	í	ó	ú	ñ	Ñ	ª	º	¿	_
170	¬	½	¼	¡	«	»				|
180	*									
190		*	*	*	*	*	_	*		
200										
210								*	*	
220						ß				
230	µ			???				???	???	
240		±	>=	<=			÷		°	.
250	.		_	²		−?−				

K.3 ISO−8859−1 − Latin 1 − ANSI

Der Zeichensatz ISO−8859−1 (Latin 1 − West European) erweitert den ASCII Zeichensatz um ein 8.Bit. Die Zeichen mit den Codes 0 bis 127 entsprechen dem ASCII Zeichensatz. Dieser Zeichensatz kommt unter Windows und UNIX (inkl. Linux) zum Einsatz.

Tabelle 30. ANSI Zeichensatz

nb=nicht benutzt; −?− nicht darstellbar										
	0	1	2	3	4	5	6	7	8	9
120									nb	nb
130	‚	−?−	"	...			^	−?−	−?−	<
140	OE	nb	nb	nb	nb	'	'	"	"	*
150	−	−−	~	(TM)	−?−	>	oe	nb	nb	
160	−?−	¡	¢	£	¤	¥	¦	§	¨	©
170	a	«	¬	−	®	¯	°	±	2	3
180	´	µ	¶	·	−?−	1	o	»	¼	½
190	¾	¿	À	Á	Â	Ã	Ä	Å	Æ	Ç
200	È	É	Ê	Ë	Ì	Í	Î	Ï		Ñ
210	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û
220	Ü	Ý	Þ	ß	à	á	â	ã	ä	å
230	æ	ç	è	é	ê	ë	ì	í	î	ï
240		ñ	ò	ó	ô	õ	ö	÷	ø	ù
250	ú	û	ü	ý	þ	ÿ				

K.4 EBCDIC

Tabelle 31. EBCDIC Zeichensatz

	0	1	2	3	4	5	6	7	8	9
060					BLANK					
070						.	<	(	+	|
080	&									
090	!	$	*	)	;	^	_	/		
100							|	,	%	_
110	>	?								
120			:	#	@	'	=	"		a
130	b	c	d	e	f	g	h	i		
140						j	k	l	m	n
150	o	p	q	r						
160			s	t	u	v	w	x	y	z
170										

180										
190			{	A	B	C	D	E	F	G
200	H	I							}	J
210	K	L	M	N	O	P	Q	R		
220					\		S	T	U	V
230	W	X	Y	Z						
240	0	1	2	3	4	5	6	7	8	9

K.5 Unicode

Frühere Zeichensätze deckten immer nur bestimmte Sprachen und ihre geläufigen Zeichen ab. US Zeichensätze enthielten z.B. i.d.R. keine europäischen Umlaute. Diese wiederum sind in europäischen Zeichensätzen häufig unterschiedlichen Nummern zugeordnet.
Unicode ist ein Zeichensatzstandard, der Unterstützung für ca. eine Million Zeichen umfaßt und eine einheitliche Position für alle Zeichen definiert. Unicode 3.0 enthält alle Zeichen, die von ISO/IEC 10646−1:2000 definiert werden. Dies umfaßt z.Zt. 49194 Zeichen, den sog. Basic Multilingual Plane (BMP).

Unicode definiert drei verschiedene Kodierungsformate, die alle dasselbe Zeichen darstellen und somit verlustfrei untereinander konvertiert werden können:

- UTF−8 (1 Byte pro Codeunit)
- UTF−16 (2 Byte pro Codeunit)
- UTF−32 (4 Byte pro Codeunit)

Alle Formate benötigen mind. 4 Byte Speicherplatz pro Zeichen. Das ü kann in Unicode z.B. als U+00FC "ü" kodiert werden.

Weitere Informationen sind unter http://www.unicode.org/versions/ zu finden.

Anhang L. Literaturverzeichnis

Erlenkötter, Helmut; Reher, Volker	Programmiersprache C Ein strukturierte Einführung Reinbek bei Hamburg: Rowohlt Taschenbuch Verlag GmbH ISBN 3 499 18166 5
IBM document	Common Programming Interface − C Reference Doc.Nr. SC26−4353−0 IBM document SAA − Writing Applications: A Design Guide Doc.Nr. SC26−4362
Kernighan, Brian W; Ritchie, Dennis M	Programmieren in C − Mit dem C−Reference Manual in deutscher Sprache 2. Ausgabe, ANSI C Muenchen: Hanser ISBN 3−446−15497−3
Kernighan, Brian W; Ritchie, Dennis M	The C Programming Language Englewood Cliffs: Prentice−Hall ISBN 0−13−110370−9

Anhang M. Online Ressourcen

M.1 Ressourcen im IBM Intranet − Foren

Im IBM Intranet stehen unter http://w3.ibm.com/forums/index.htm verschiedene Diskussionsforen bereit.

Die VM Diskussionsforen sind im IBM Intranet unter http://tr2.fishkill.ibm.com:8080/frames.htm verfügbar. Wählen Sie dort IBMPC bzw. NEWFORUM und dann IBMPC aus, dort finden Sie u.a. die Foren C−LANG und C++.

M.2 Ressourcen im Internet

Die folgenden Adressen stellen eine Auswahl des Angebotes an Seiten im Internet dar, die sich mit der Programmiersprache C befassen. Da nichts unvergänglich ist, können sich die Adressen jederzeit ändern. Stand der Liste: 15.04.1997

C Library Reference (Übersicht über die C Libraryfunktionen)

```
http://ebweb.tuwien.ac.at/gnu-docs/libc-1.12/libc_toc.html
```

The Electronic Developer Magazine for OS/2

Viele Informationen und Beispielcode zur Programmierung unter OS/2. Auch Produkttests und hervorragende Einführungen in C und C++.

```
http://www.edm2.com/
```

Learn C/C++ Today (A list of resources/tutorial)

```
http://www.lib.ox.ac.uk/internet/news/faq/archive/c-faq.learn-c-cpp-today.html
```

Online EE Textbook (C Kurs)

```
http://spectra.eng.hawaii.edu/Courses/EE150/Book/book.html
```

Programmiersprache C/C++ (C Kurs)

```
http://www.informatik.uni-halle.de/lehre/c/index.html
```

Programming in C (C Kurs)

```
http://www.cm.cf.ac.uk/Dave/C/CE.html
```

SIMTEL NET − C (Programme und Quelltexte)

```
http://www.simtel.net/simtel.net/msdos/c-pre.html
```

SNIPPETS (Exzellente Sammlung von Quelltexten und Informationsdateien)

```
http://www.brokersys.com/snippets/
```

Index

Sonderzeichen

A

B

C

Z

1

Portabilität bezeichnet die Möglichkeit, den Quellcode eines Programmes ohne Modifikation auf mehreren Computersystemen übersetzen zu lassen.

2

Eine Variable ist einfach ein praktischer Ort, an dem man etwas unterbringen kann. Ein Ort mit einem Namen, an dem Sie Ihr spezielles Etwas wiederfinden, wenn Sie zu einem späteren Zeitpunkt nachsehen. Wie im richtigen Leben gibt es verschiedene Arten von Orten, an denen Dinge gespeichert werden können. Die einen sind eher privater Natur, während die anderen der Öffentlichkeit zugänglich sind. Einige Orte existieren nur über kurze Zeit, während andere immer vorhanden sind. Informatiker lieben es, hier vom "Geltungsbereich" von Variablen zu sprechen, aber mehr als das gerade Gesagte ist nicht damit gemeint. (Programmieren mit Perl; Larry Wall, Tom Christiansen Randal L. Schwartz; O'Reilly Verlag 1997; Deutsche Ausgabe der 2. Auflage)
Eine singulare Variable ist ein "Skalar", eine plurale Variable ein "Array".

3

Die Genauigkeit bezieht sich auf die Anzahl Stellen (Vor− und Nachkomma), auf die die Zahl genau dargestellt werden kann.

4

American Standard Code for Information Interchange (siehe auch K.1, "Standard ASCII")

5

American National Standards Institute (siehe auch K.3, "ISO−8859−1 − Latin 1 − ANSI")

6

extended binary coded decimal interchange code (siehe auch K.4, "EBCDIC")

7

Mit Variablen sind hier auch komplexe Datentypen wie Strukturen gemeint.

[8]

End Of File, Dateiendemarkierung

[9]

Eine Exception ist eine, vom Betriebssystem ausgelöste, Ausnahmebedingung, die den Abbruch der Programmausführung zur Folge hat, sofern keine eigenen Routinen für die Behandlung von Exceptions (sog. Exception Handler) geschrieben wurden. Diese Ausnahmebedingung ist das Resultat eines Programmfehlers.